COS 226

Algorithms and Data Structures
Princeton University
Spring 2010

Robert Sedgewick

Course Overview

- outline
- why study algorithms?
- usual suspects
- coursework
- resources

What is COS 226?

- Intermediate-level survey course.
- Programming and problem solving with applications.
- Algorithm: method for solving a problem.
- Data structure: method to store information.

topic	data structures and algorithms
data types	stack, queue, union-find, priority queue
sorting	quicksort, mergesort, heapsort, radix sorts
searching	hash table, BST, red-black tree
graphs	BFS, DFS, Prim, Kruskal, Dijkstra
strings	KMP, regular expressions, TST, Huffman, LZW
geometry	Graham scan, k-d tree, Voronoi diagram

Why study algorithms?

Their impact is broad and far-reaching.

Internet. Web search, packet routing, distributed file sharing, ...

Biology. Human genome project, protein folding, ...

Computers. Circuit layout, file system, compilers, ...

Computer graphics. Movies, video games, virtual reality, ...

Security. Cell phones, e-commerce, voting machines, ...

Multimedia. CD player, DVD, MP3, JPG, DivX, HDTV, ...

Transportation. Airline crew scheduling, map routing, ...

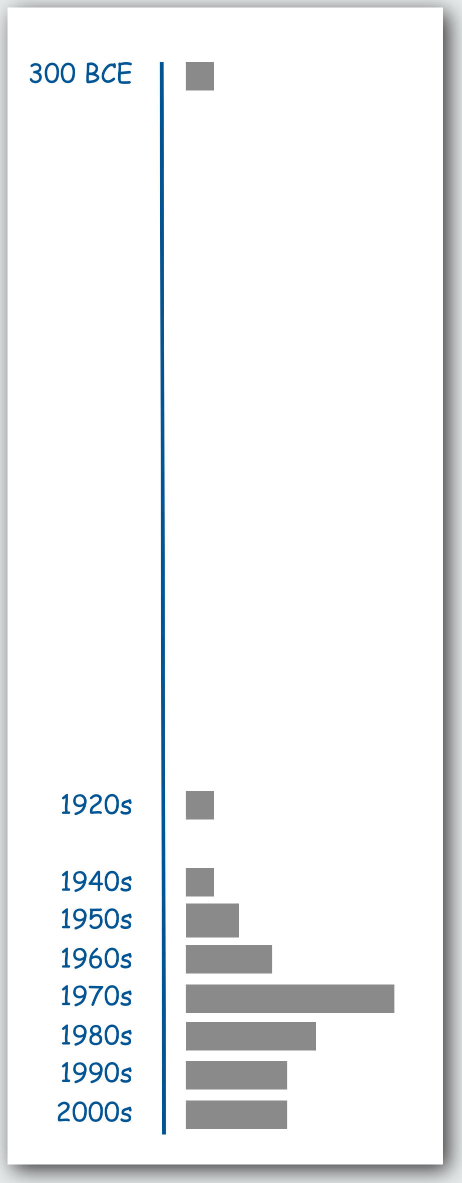
Physics. N-body simulation, particle collision simulation, ...

...

Why study algorithms?

Old roots, new opportunities.

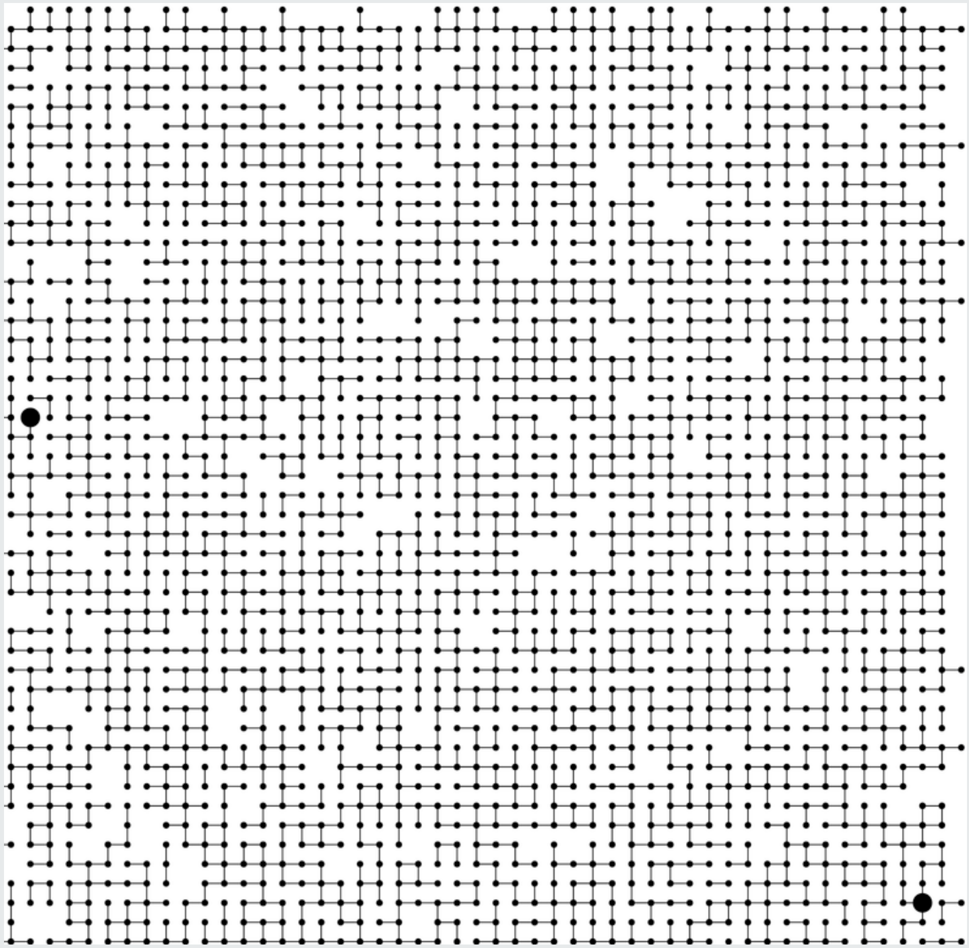
- Study of algorithms dates at least to Euclid.
- Some important algorithms were discovered by undergraduates!



Why study algorithms?

To solve problems that could not otherwise be addressed.

Ex. Network connectivity. [stay tuned]



Why study algorithms?

For intellectual stimulation.

> " *For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing.* " — Francis Sullivan

> " *An algorithm must be seen to be believed.* " — D. E. Knuth

Why study algorithms?

They may unlock the secrets of life and of the universe.

Computational models are replacing mathematical models in scientific inquiry.

$$E = mc^2$$

$$F = ma$$

$$F = \frac{G m_1 m_2}{r^2}$$

$$\left[-\frac{\hbar^2}{2m} \nabla^2 + V(r) \right] \Psi(r) = E \, \Psi(r)$$

20th century science
(formula based)

```
for (double t = 0.0; true; t = t + dt)
   for (int i = 0; i < N; i++)
   {
      bodies[i].resetForce();
      for (int j = 0; j < N; j++)
         if (i != j)
            bodies[i].addForce(bodies[j]);
   }
```

21st century science
(algorithm based)

" Algorithms: a common language for nature, human, and computer. " — Avi Wigderson

Why study algorithms?

For fun and profit.

Why study algorithms?

- Their impact is broad and far-reaching.
- Old roots, new opportunities.
- To solve problems that could not otherwise be addressed.
- For intellectual stimulation.
- They may unlock the secrets of life and of the universe.
- For fun and profit.

Why study anything else?

Coursework and grading

8 programming assignments. 45%

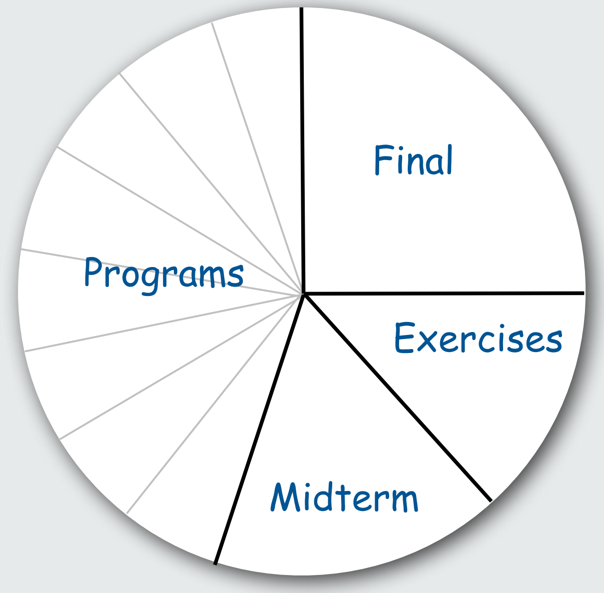
- Electronic submission.
- Due 11pm, starting Wednesay 9/23.

Exercises. 15%

- Due in lecture, starting Tuesday 9/22.

Exams.

- Closed-book with cheatsheet.
- Midterm. 15%
- Final. 25%



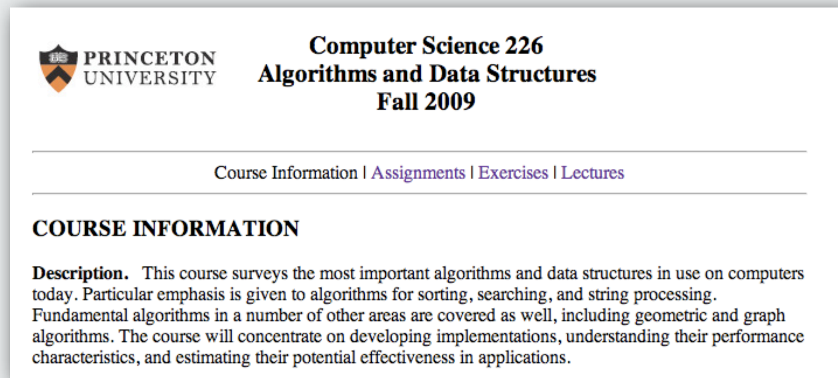
Staff discretion. To adjust borderline cases.

everyone needs to meet me in office hours

Resources (web)

Course content.

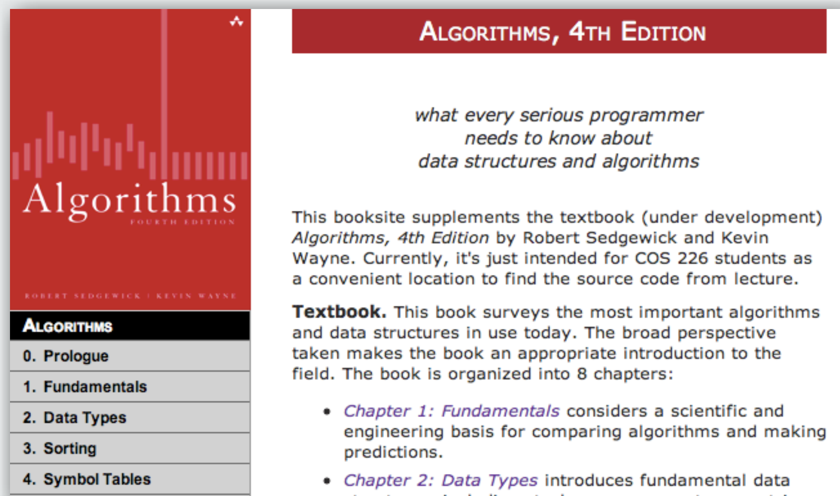
- Course info.
- Exercises.
- Lecture slides.
- Programming assignments.
- Submit assignments.



`http://www.princeton.edu/~cos226`

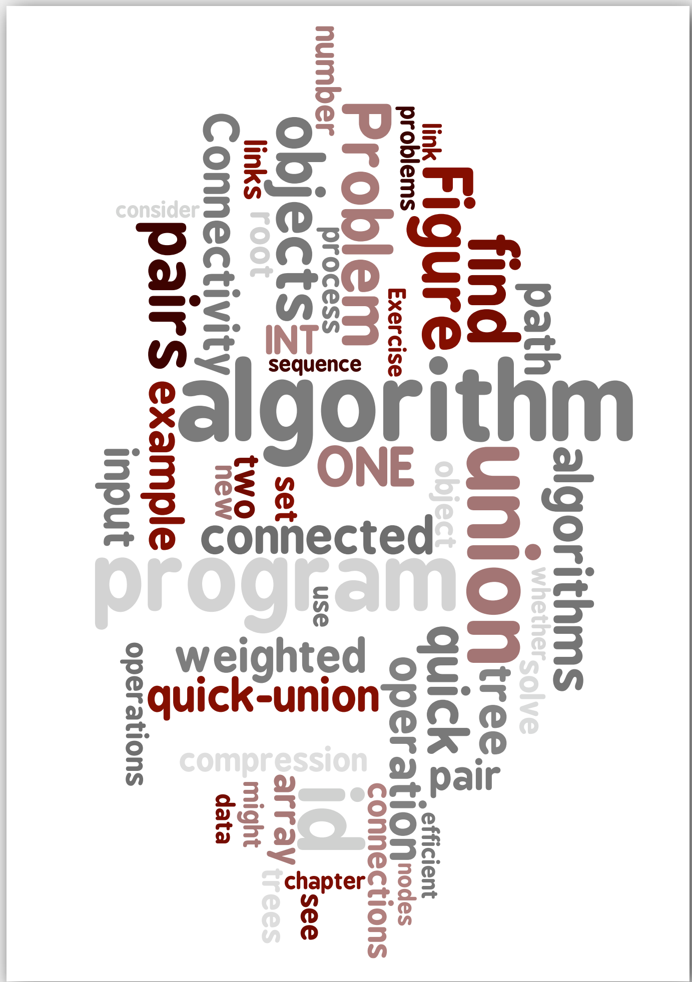
Booksites.

- Brief summary of content.
- Download code from lecture.



`http://www.cs.princeton.edu/IntroProgramming`
`http://www.cs.princeton.edu/algs4`

1.5 Case Study



- dynamic connectivity
- quick find
- quick union
- improvements
- applications

Subtext of today's lecture (and this course)

Steps to developing a usable algorithm.

- Model the problem.
- Find an algorithm to solve it.
- Fast enough? Fits in memory?
- If not, figure out why.
- Find a way to address the problem.
- Iterate until satisfied.

The scientific method.

Mathematical analysis.

- dynamic connectivity
- quick find
- quick union
- improvements
- applications

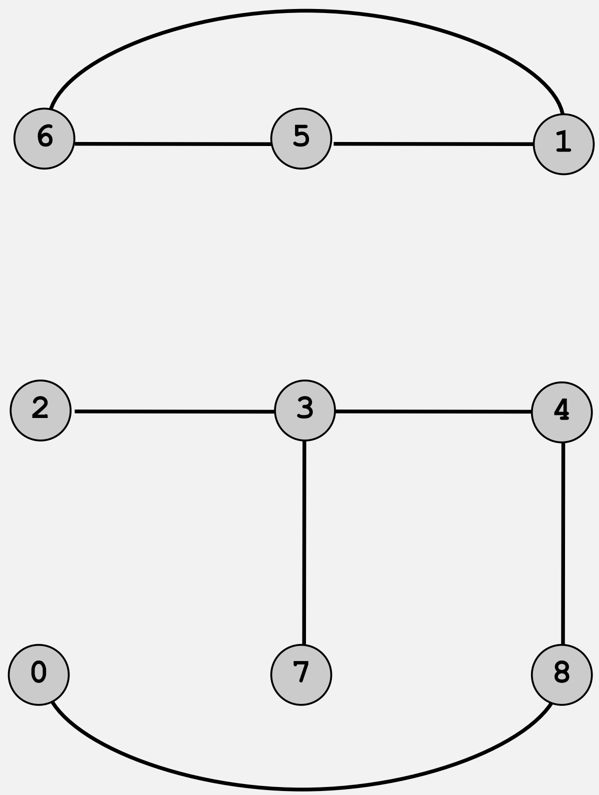
Dynamic connectivity

Given a set of objects

- Union: connect two objects.
- Find: is there a path connecting the two objects?

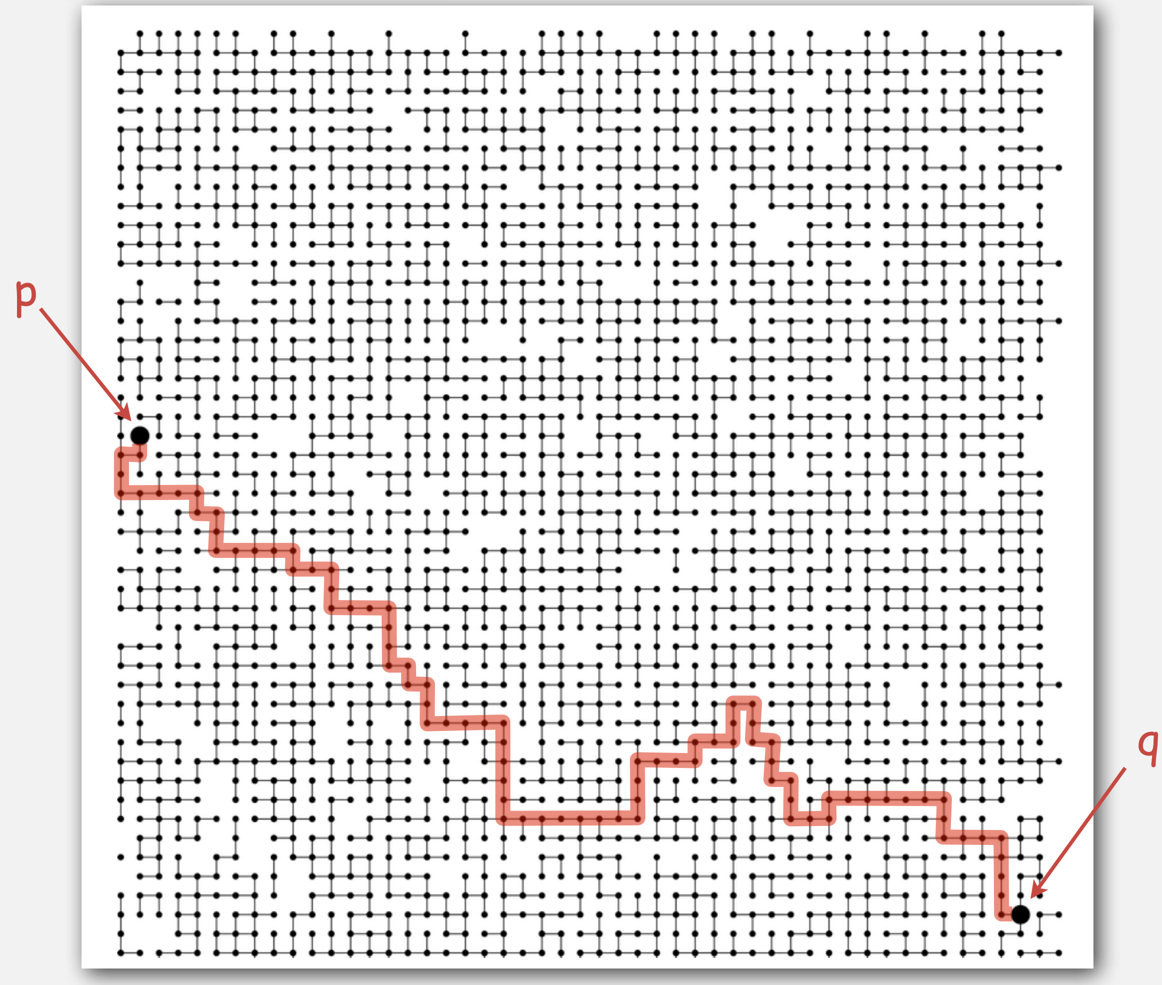
more difficult problem: find the path

```
union(3, 4)
union(8, 0)
union(2, 3)
union(5, 6)
 find(0, 2)      no
 find(2, 4)      yes
union(5, 1)
union(7, 3)
union(1, 6)
union(4, 8)
 find(0, 2)      yes
 find(2, 4)      yes
```



Network connectivity: larger example

Q. Is there a path from p to q?



A. Yes.

but finding the path is more difficult: stay tuned (Chapter 4)

Modeling the objects

Dynamic connectivity applications involve manipulating objects of all types.

- Variable name aliases.
- Pixels in a digital photo.
- Computers in a network.
- Web pages on the Internet.
- Transistors in a computer chip.
- Metallic sites in a composite system.

When programming, convenient to name objects 0 to N-1.

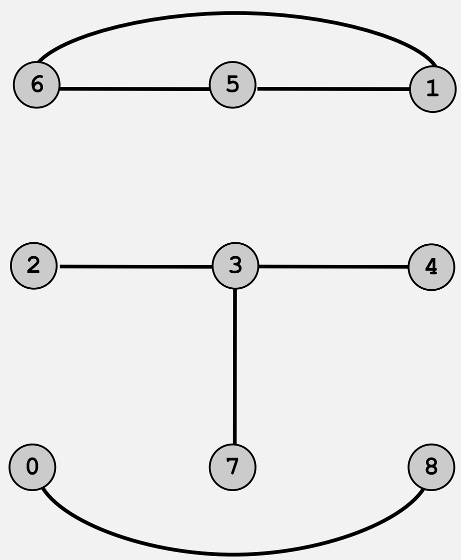
- Use integers as array index.
- Suppress details not relevant to union-find.

can use symbol table to translate from object names to integers (stay tuned)

Modeling the connections

Transitivity. If `p` is connected to `q` and `q` is connected to `r`, then `p` is connected to `r`.

Connected components. Maximal set of objects that are mutually connected.



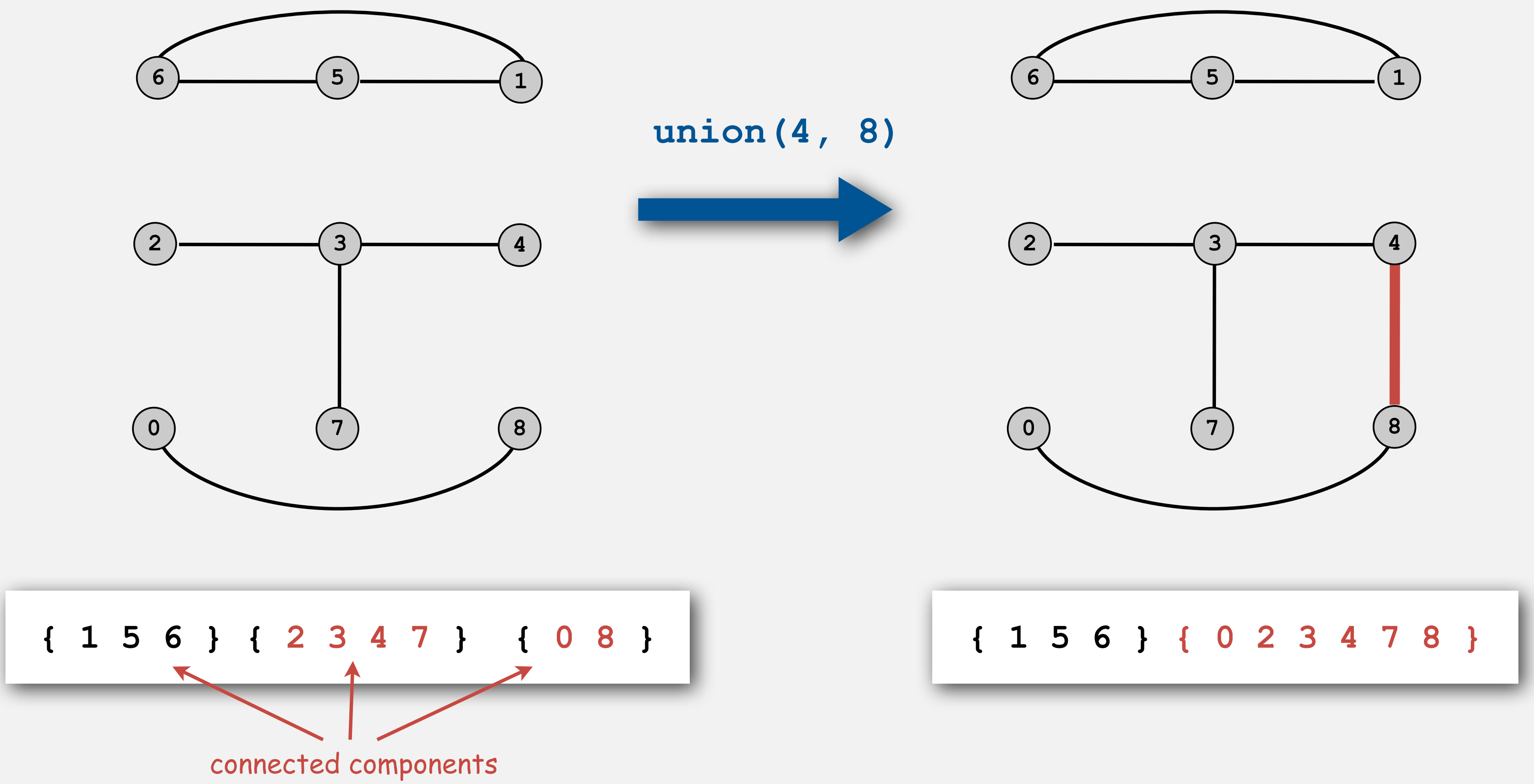
`{ 1 5 6 } { 2 3 4 7 } { 0 8 }`

connected components

Implementing the operations

Find query. Check if two objects are in the same set.

Union command. Replace sets containing two objects with their union.



Union-find data type (API)

Goal. Design efficient data structure for union-find.

- Number of objects N can be huge.
- Number of operations M can be huge.
- Find queries and union commands may be intermixed.

public class UnionFind		
	`UnionFind(int N)`	*create union-find data structure with N objects and no connections*
`boolean`	`find(int p, int q)`	*are p and q in the same set?*
`void`	`unite(int p, int q)`	*replace sets containing p and q with their union*

- dynamic connectivity
- **quick find**
- quick union
- improvements
- applications

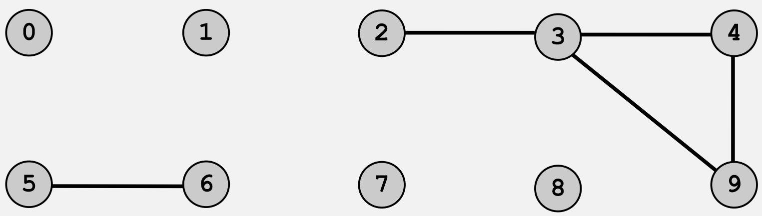
Quick-find [eager approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `p` and `q` are connected if they have the same id.

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	9	9	6	6	7	8	9

5 and 6 are connected
2, 3, 4, and 9 are connected



Quick-find [eager approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `p` and `q` are connected if they have the same id.

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	9	9	6	6	7	8	9

5 and 6 are connected
2, 3, 4, and 9 are connected

Find. Check if `p` and `q` have the same id.

`id[3] = 9`; `id[6] = 6`
3 and 6 not connected

Quick-find [eager approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `p` and `q` are connected if they have the same id.

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	9	9	6	6	7	8	9

5 and 6 are connected
2, 3, 4, and 9 are connected

Find. Check if `p` and `q` have the same id.

`id[3] = 9`; `id[6] = 6`
3 and 6 not connected

Union. To merge sets containing `p` and `q`, change all entries with `id[p]` to `id[q]`.

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	6	6	6	6	6	7	8	6

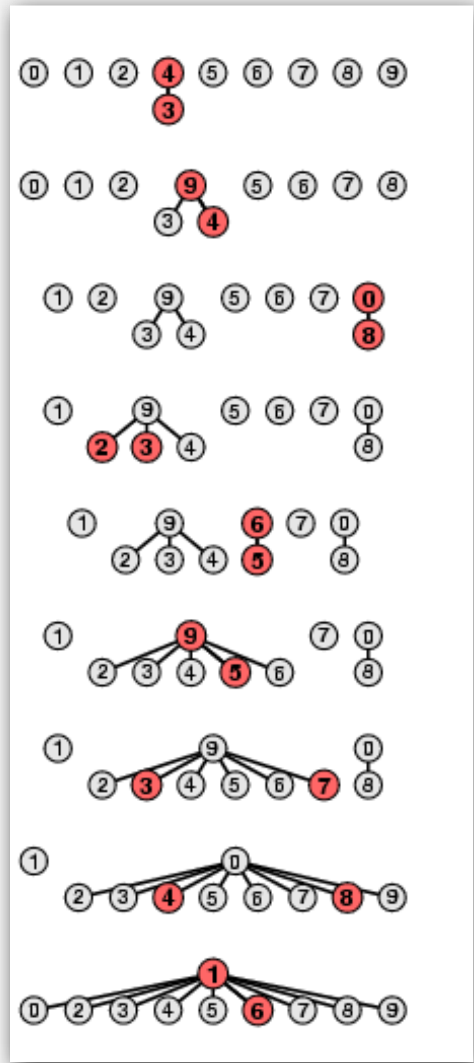
union of 3 and 6
2, 3, 4, 5, 6, and 9 are connected

problem: many values can change

Quick-find example

3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	9	9	5	6	7	8	9
8-0	0	1	2	9	9	5	6	7	0	9
2-3	0	1	9	9	9	5	6	7	0	9
5-6	0	1	9	9	9	6	6	7	0	9
5-9	0	1	9	9	9	9	9	7	0	9
7-3	0	1	9	9	9	9	9	9	0	9
4-8	0	1	0	0	0	0	0	0	0	0
6-1	1	1	1	1	1	1	1	1	1	1

problem: many values can change



Quick-find: Java implementation

```
public class QuickFind
{
   private int[] id;

   public QuickFind(int N)
   {
      id = new int[N];
      for (int i = 0; i < N; i++)
         id[i] = i;
   }

   public boolean find(int p, int q)
   {
      return id[p] == id[q];
   }

   public void unite(int p, int q)
   {
      int pid = id[p];
      for (int i = 0; i < id.length; i++)
         if (id[i] == pid) id[i] = id[q];
   }
}
```

set id of each object to itself
(N operations)

check if `p` and `q` have same id
(1 operation)

change all entries with `id[p]` to `id[q]`
(N operations)

Quick-find is too slow

Quick-find defect.

- Union too expensive (N operations).
- Trees are flat, but too expensive to keep them flat.

algorithm	union	find
quick-find	N	1

Ex. Takes N^2 operations to process sequence of N union commands on N objects.

Quadratic algorithms do not scale

Rough standard (for now).

- 10^9 operations per second.
- 10^9 words of main memory.
- Touch all words in approximately 1 second.

a truism (roughly) since 1950 !

Ex. Huge problem for quick-find.

- 10^9 union commands on 10^9 objects.
- Quick-find takes more than 10^{18} operations.
- 30+ years of computer time!

Paradoxically, quadratic algorithms get worse with newer equipment.

- New computer may be 10x as fast.
- But, has 10x as much memory so problem may be 10x bigger.
- With quadratic algorithm, takes 10x as long!

- dynamic connectivity
- quick find
- **quick union**
- improvements
- applications

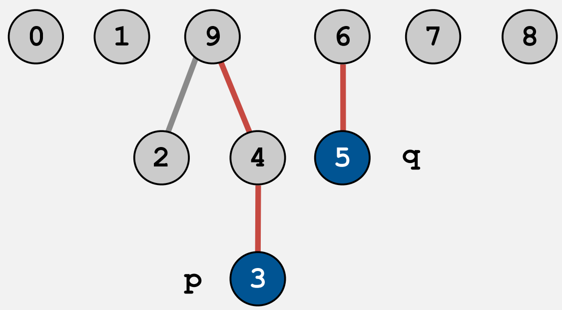
Quick-union [lazy approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `id[i]` is parent of `i`.
- Root of `i` is `id[id[id[...id[i]...]]]`.

keep going until it doesn't change

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	4	9	6	6	7	8	9



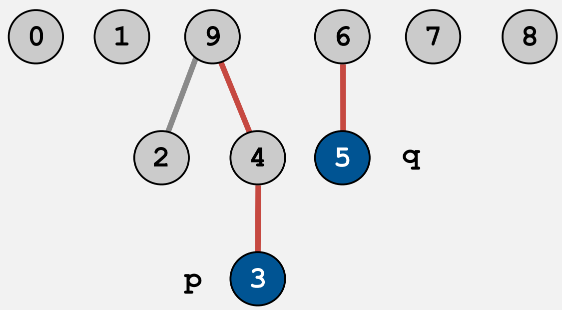
3's root is 9; 5's root is 6

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `id[i]` is parent of `i`.
- Root of `i` is `id[id[id[...id[i]...]]]`.

keep going until it doesn't change

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	4	9	6	6	7	8	9



3's root is 9; 5's root is 6
3 and 5 are not connected

Find. Check if `p` and `q` have the same root.

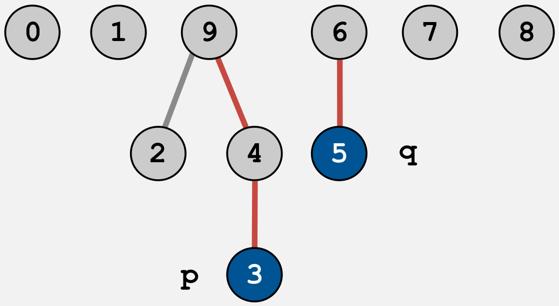
Quick-union [lazy approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `id[i]` is parent of `i`.
- Root of `i` is `id[id[id[...id[i]...]]]`.

keep going until it doesn't change

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	4	9	6	6	7	8	9



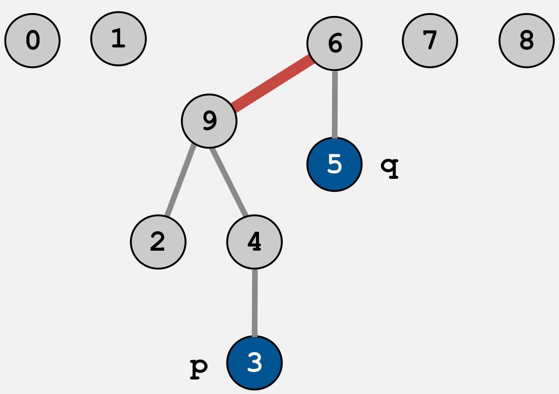
3's root is 9; 5's root is 6
3 and 5 are not connected

Find. Check if `p` and `q` have the same root.

Union. To merge sets containing `p` and `q`, set the id of `p`'s root to the id of `q`'s root.

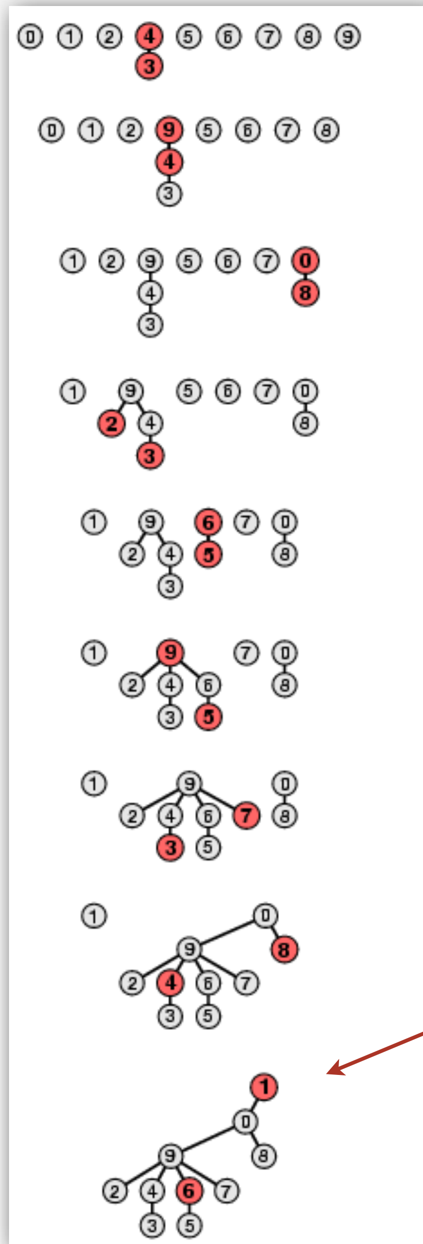
i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	4	9	6	6	7	8	6

only one value changes



Quick-union example

3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9
2-3	0	1	9	4	9	5	6	7	0	9
5-6	0	1	9	4	9	6	6	7	0	9
5-9	0	1	9	4	9	6	9	7	0	9
7-3	0	1	9	4	9	6	9	9	0	9
4-8	0	1	9	4	9	6	9	9	0	0
6-1	1	1	9	4	9	6	9	9	0	0



problem:
trees can get tall

Quick-union: Java implementation

```
public class QuickUnion
{
   private int[] id;

   public QuickUnion(int N)
   {
      id = new int[N];
      for (int i = 0; i < N; i++) id[i] = i;
   }

   private int root(int i)
   {
      while (i != id[i]) i = id[i];
      return i;
   }

   public boolean find(int p, int q)
   {
      return root(p) == root(q);
   }

   public void unite(int p, int q)
   {
      int i = root(p), j = root(q);
      id[i] = j;
   }
}
```

set id of each object to itself
(N operations)

chase parent pointers until reach root
(depth of i operations)

check if p and q have same root
(depth of p and q operations)

change root of p to point to root of q
(depth of p and q operations)

Quick-union is also too slow

Quick-find defect.

- Union too expensive (N operations).
- Trees are flat, but too expensive to keep them flat.

Quick-union defect.

- Trees can get tall.
- Find too expensive (could be N operations).

algorithm	union	find
quick-find	N	1
quick-union	N †	N

← worst case

† includes cost of finding root

- dynamic connectivity
- quick find
- quick union
- **improvements**
- applications

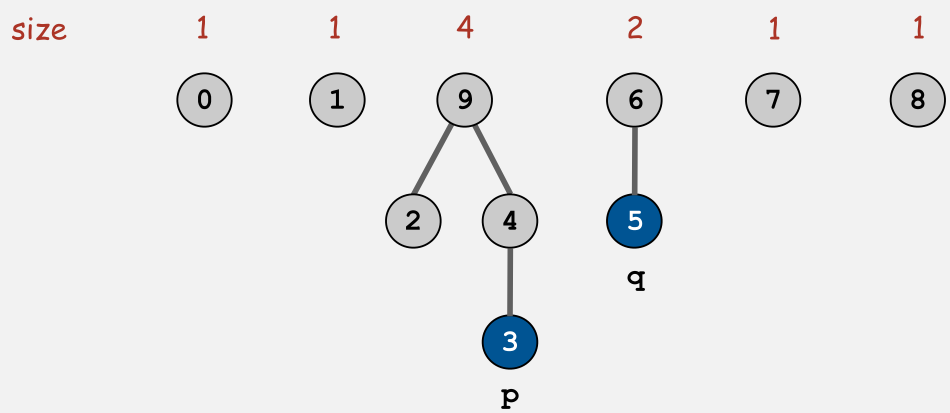
Improvement 1: weighting

Weighted quick-union.

- Modify quick-union to avoid tall trees.
- Keep track of size of each set.
- Balance by linking small tree below large one.

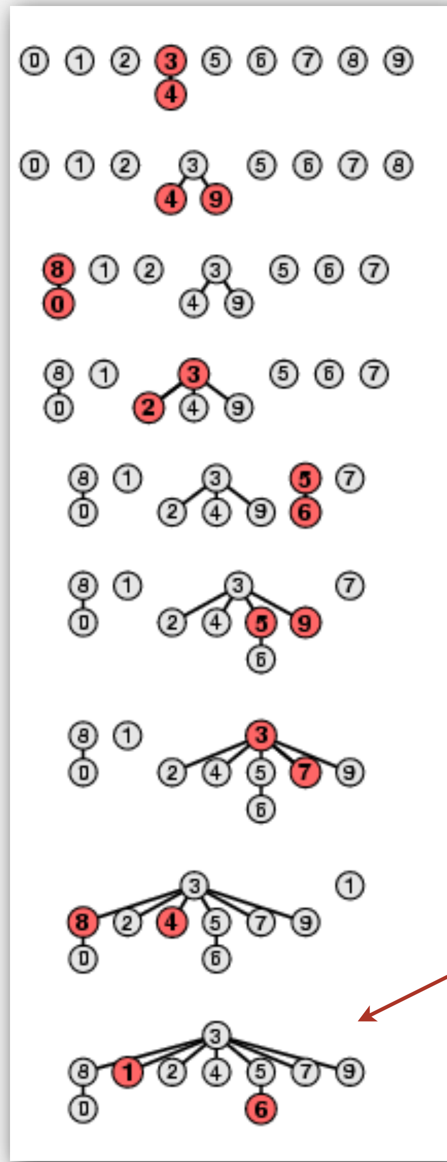
Ex. Union of 3 and 5.

- Quick union: link 9 to 6.
- Weighted quick union: link 6 to 9.



Weighted quick-union example

3-4	0	1	2	3	3	5	6	7	8	9
4-9	0	1	2	3	3	5	6	7	8	3
8-0	8	1	2	3	3	5	6	7	8	3
2-3	8	1	3	3	3	5	6	7	8	3
5-6	8	1	3	3	3	5	5	7	8	3
5-9	8	1	3	3	3	3	5	7	8	3
7-3	8	1	3	3	3	3	5	3	8	3
4-8	8	1	3	3	3	3	5	3	3	3
6-1	8	3	3	3	3	3	5	3	3	3



no problem:
trees stay flat

Weighted quick-union: Java implementation

Data structure. Same as quick-union, but maintain extra array `sz[i]` to count number of objects in the tree rooted at `i`.

Find. Identical to quick-union.

```
return root(p) == root(q);
```

Union. Modify quick-union to:

- Merge smaller tree into larger tree.
- Update the `sz[]` array.

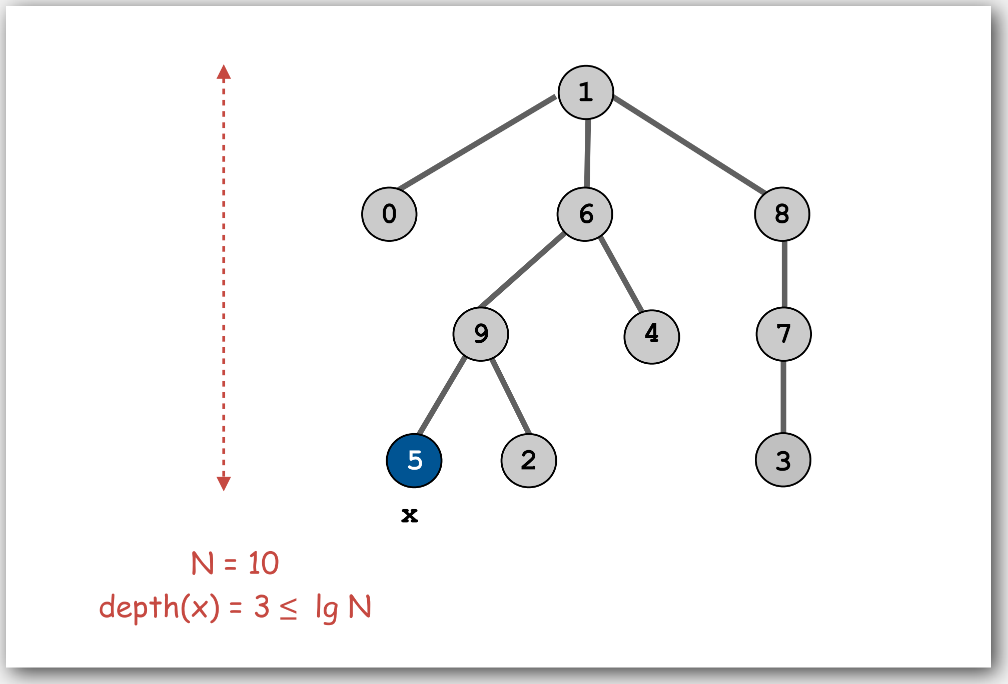
```
int i = root(p);
int j = root(q);
if  (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
else                { id[j] = i; sz[i] += sz[j]; }
```

Weighted quick-union analysis

Analysis.

- Find: takes time proportional to depth of p and q.
- Union: takes constant time, given roots.

Proposition. Depth of any node x is at most lg N.



Analysis.

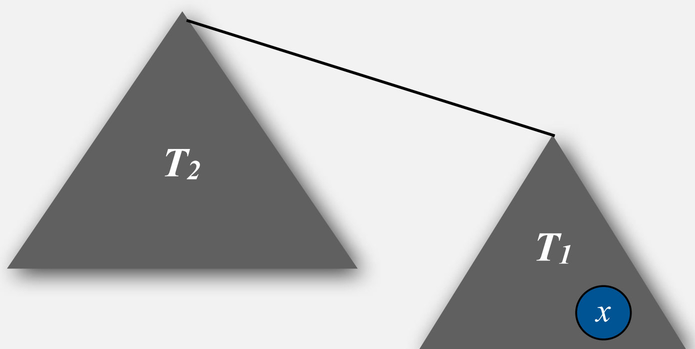
- Find: takes time proportional to depth of p and q.
- Union: takes constant time, given roots.

Proposition. Depth of any node x is at most lg N.

Pf. When does depth of x increase?

Increases by 1 when tree T_1 containing x is merged into another tree T_2.

- The size of the tree containing x at least doubles since $|T_2| \geq |T_1|$.
- Size of tree containing x can double at most lg N times. Why?



Weighted quick-union analysis

Analysis.

- Find: takes time proportional to depth of p and q.
- Union: takes constant time, given roots.

Proposition. Depth of any node x is at most lg N.

algorithm	union	find
quick-find	N	1
quick-union	N †	N
weighted QU	lg N †	lg N

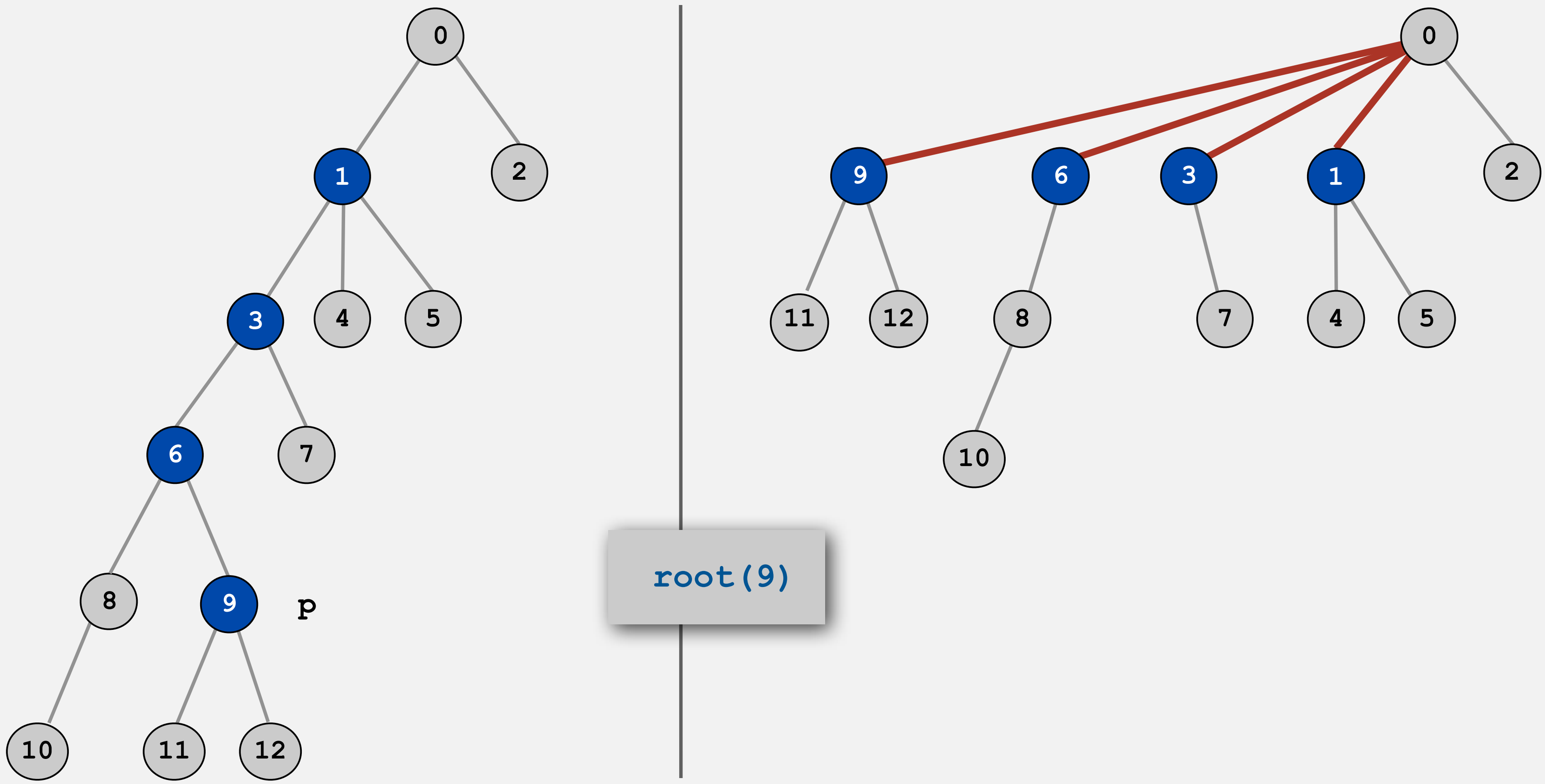
† includes cost of finding root

Q. Stop at guaranteed acceptable performance?

A. No, easy to improve further.

Improvement 2: path compression

Quick union with path compression. Just after computing the root of **p**, set the id of each examined node to **root(p)**.



Path compression: Java implementation

Standard implementation: add second loop to `root()` to set the `id[]` of each examined node to the root.

Simpler one-pass variant: halve the path length by making every other node in path point to its grandparent.

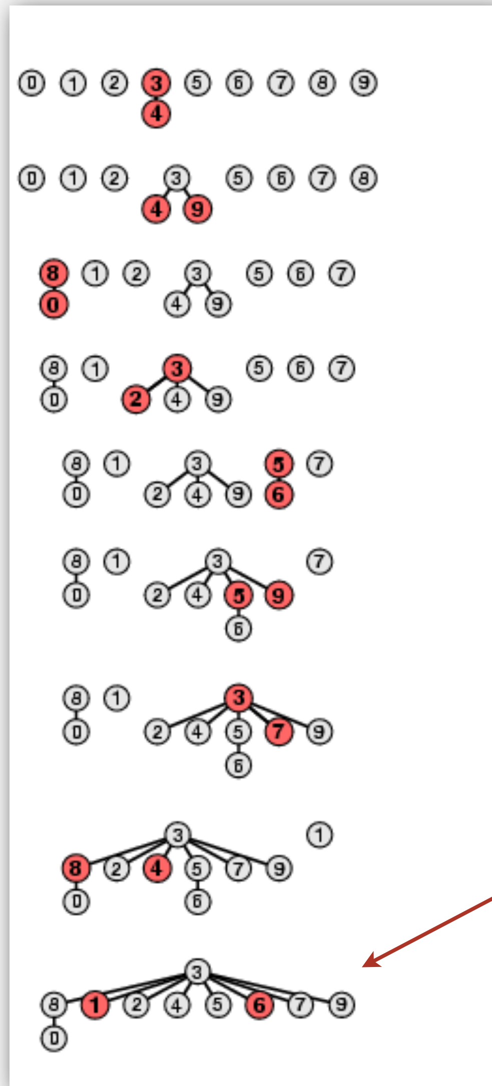
```
public int root(int i)
{
   while (i != id[i])
   {
      id[i] = id[id[i]];
      i = id[i];
   }
   return i;
}
```

only one extra line of code !

In practice. No reason not to! Keeps tree almost completely flat.

Weighted quick-union with path compression example

3-4	0	1	2	3	3	5	6	7	8	9
4-9	0	1	2	3	3	5	6	7	8	3
8-0	8	1	2	3	3	5	6	7	8	3
2-3	8	1	3	3	3	5	6	7	8	3
5-6	8	1	3	3	3	5	5	7	8	3
5-9	8	1	3	3	3	3	5	7	8	3
7-3	8	1	3	3	3	3	5	3	8	3
4-8	8	1	3	3	3	3	5	3	3	3
6-1	8	3	3	3	3	3	3	3	3	3



no problem:
trees stay VERY flat

WQUPC performance

Proposition. [Tarjan 1975] Starting from an empty data structure, any sequence of M union and find ops on N objects takes O(N + M lg* N) time.

- Proof is very difficult.
- But the algorithm is still simple!

actually O(N + M α(M, N))
see COS 423

Linear algorithm?

- Cost within constant factor of reading in the data.
- In theory, WQUPC is not quite linear.
- In practice, WQUPC is linear.

because lg* N is a constant in this universe

N	lg* N
1	0
2	1
4	2
16	3
65536	4
2^{65536}	5

lg* function
number of times needed to take
the lg of a number until reaching 1

Amazing fact. No linear-time linking strategy exists.

Summary

Bottom line. WQUPC makes it possible to solve problems that could not otherwise be addressed.

algorithm	worst-case time
quick-find	M N
quick-union	M N
weighted QU	N + M log N
QU + path compression	N + M log N
weighted QU + path compression	N + M lg* N

M union-find operations on a set of N objects

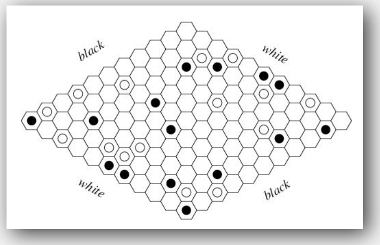
Ex. [10^9 unions and finds with 10^9 objects]

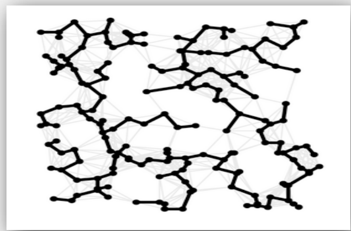
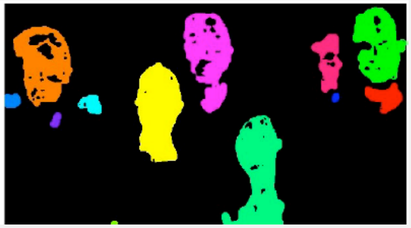
- WQUPC reduces time from 30 years to 6 seconds.
- Supercomputer won't help much; good algorithm enables solution.

- dynamic connectivity
- quick find
- quick union
- improvements
- **applications**

Union-find applications

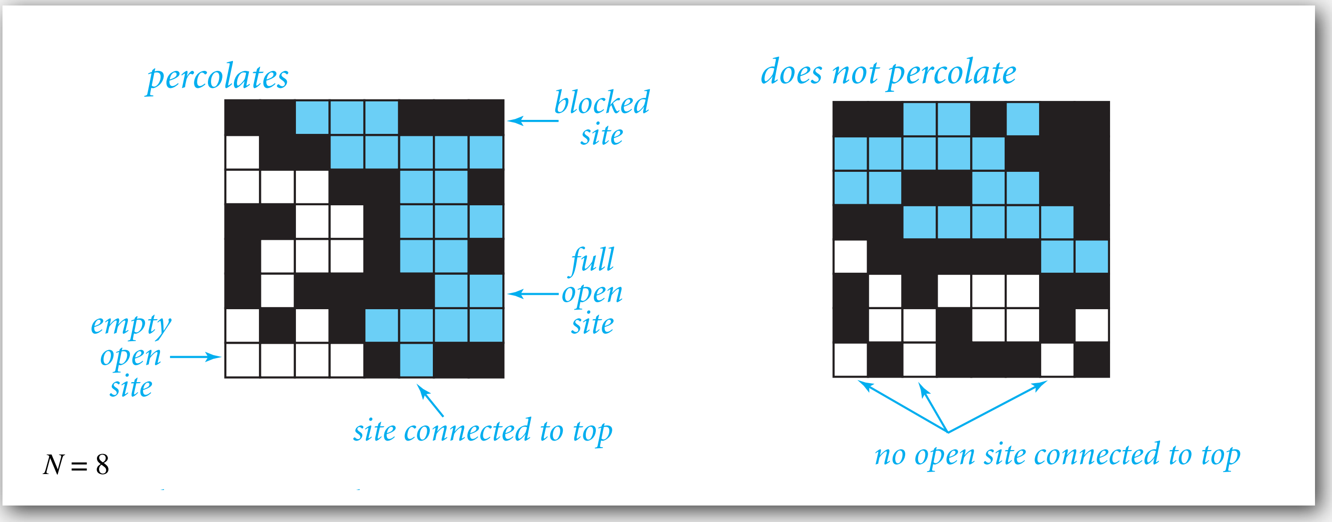
- Percolation.
- Games (Go, Hex).
- ✓ Network connectivity.
- Least common ancestor.
- Equivalence of finite state automata.
- Hoshen-Kopelman algorithm in physics.
- Hinley-Milner polymorphic type inference.
- Kruskal's minimum spanning tree algorithm.
- Compiling equivalence statements in Fortran.
- Morphological attribute openings and closings.
- Matlab's `bwlabel()` function in image processing.



A model for many physical systems:

- N-by-N grid of sites.
- Each site is open with probability p (or blocked with probability 1-p).
- System percolates if top and bottom are connected by open sites.



Percolation

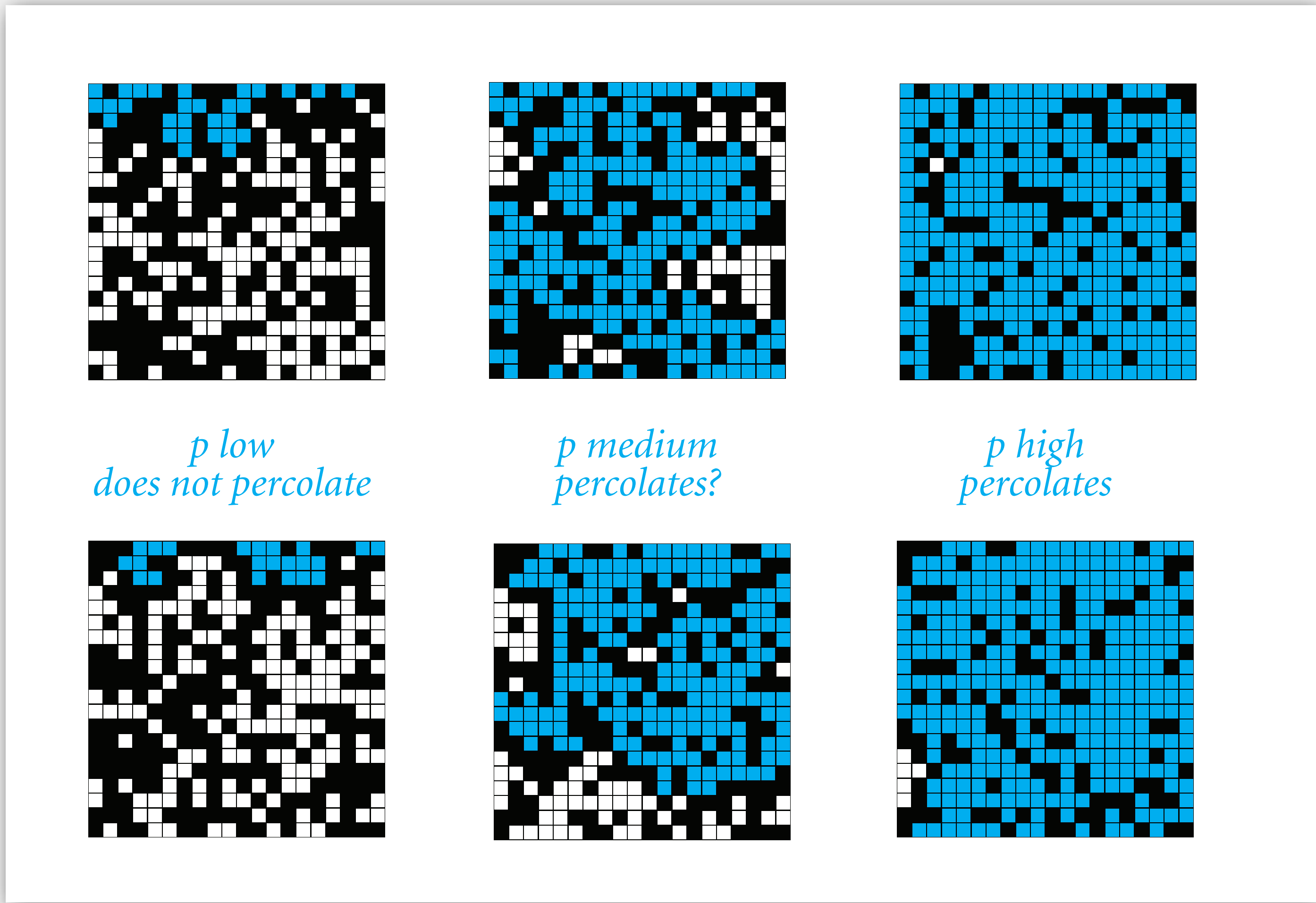
A model for many physical systems:

- N-by-N grid of sites.
- Each site is open with probability p (or blocked with probability 1-p).
- System percolates if top and bottom are connected by open sites.

model	system	vacant site	occupied site	percolates
electricity	material	conductor	insulated	conducts
fluid flow	material	empty	blocked	porous
social interaction	population	person	empty	communicates

Likelihood of percolation

Depends on site vacancy probability p.

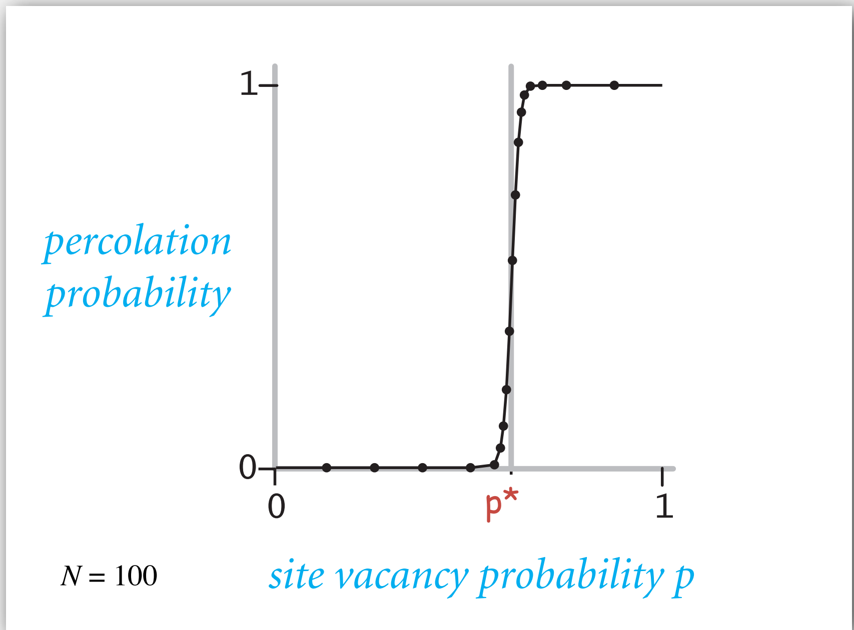


$N = 20$

When N is large, theory guarantees a sharp threshold p*.

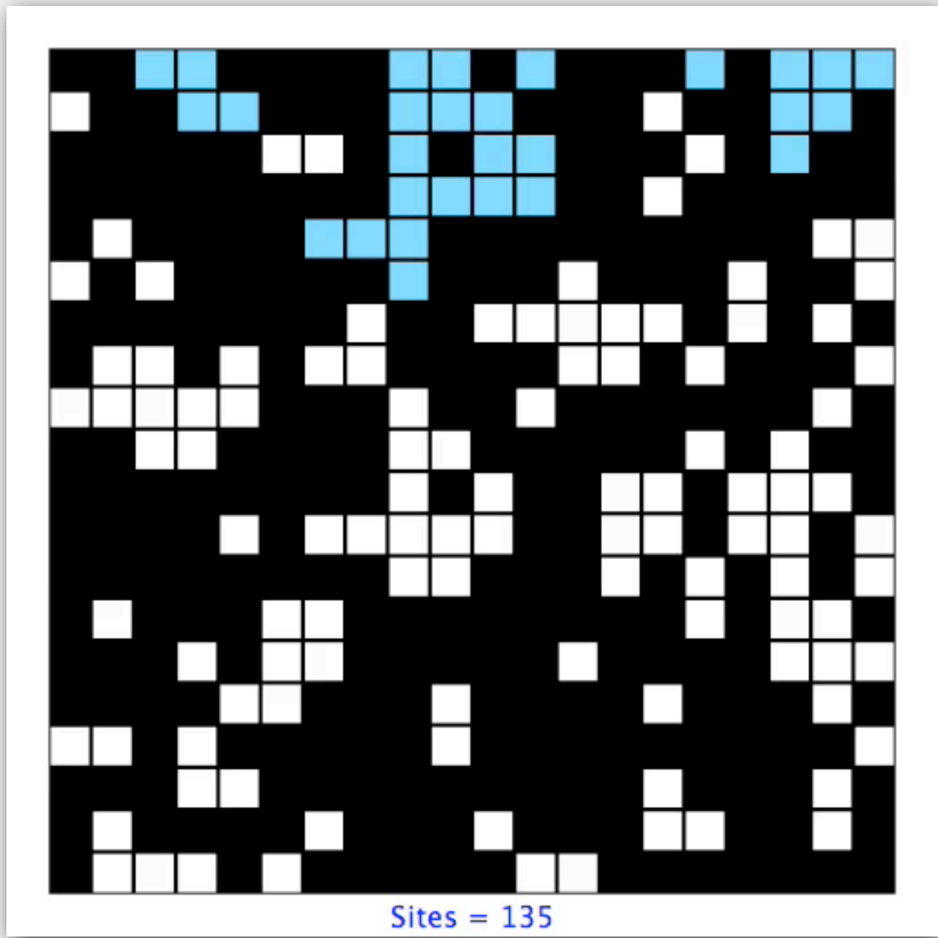
- p > p*: almost certainly percolates.
- p < p*: almost certainly does not percolate.

Q. What is the value of p* ?



Monte Carlo simulation

- Initialize N-by-N whole grid to be blocked.
- Declare random sites open until top connected to bottom.
- Vacancy percentage estimates p*.

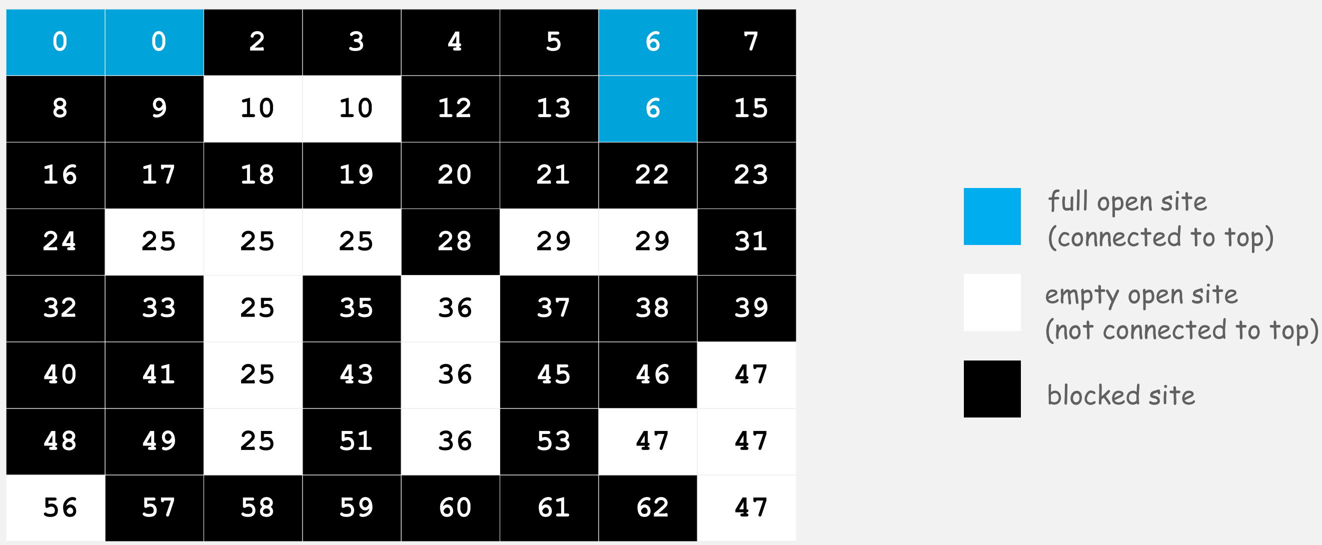


UF solution to find percolation threshold

How to check whether system percolates?

- Create an object for each site.
- Sites are in same set if connected by open sites.
- Percolates if any site in top row is in same set as any site in bottom row.

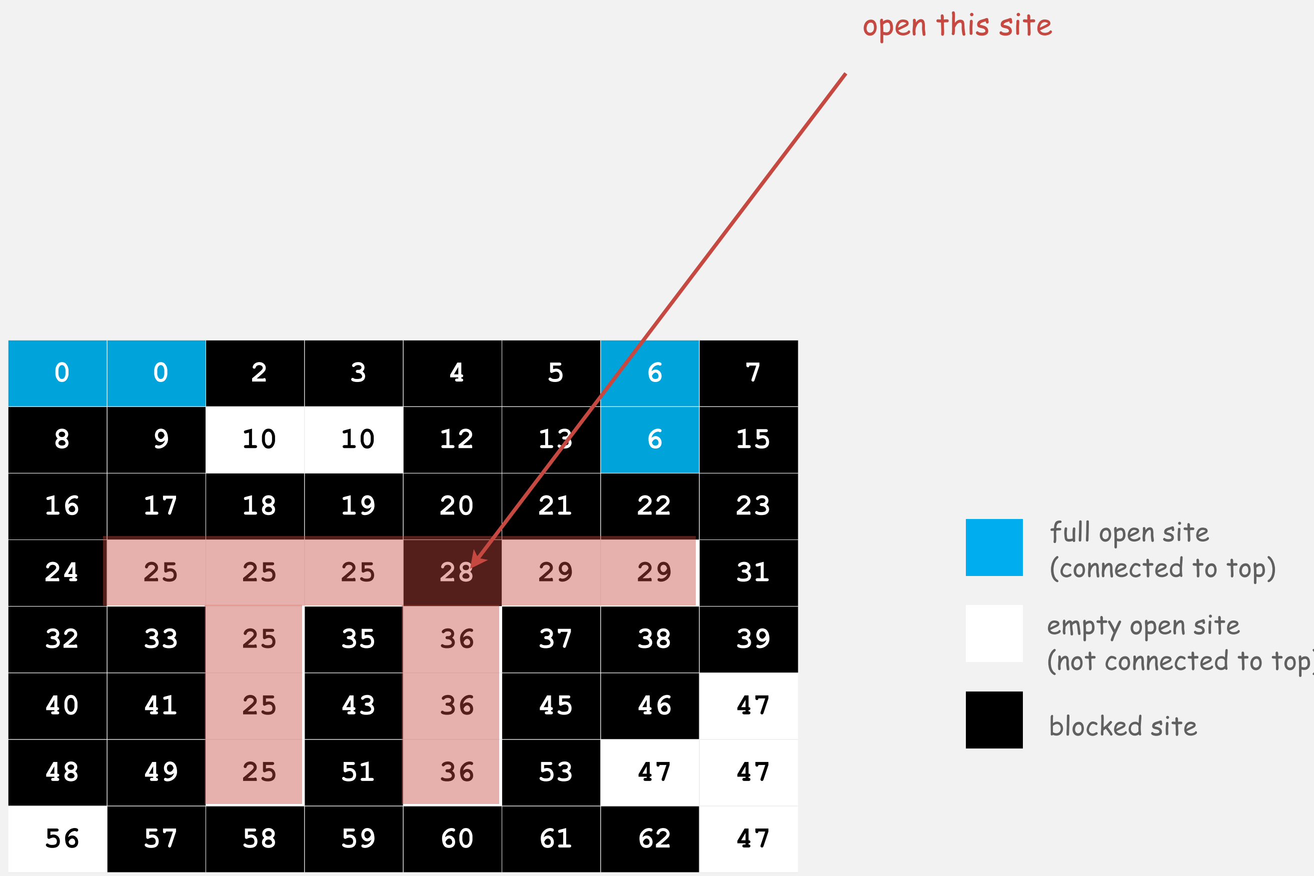
brute force algorithm needs to check N^2 pairs



$N = 8$

UF solution to find percolation threshold

Q. How to declare a new site open?



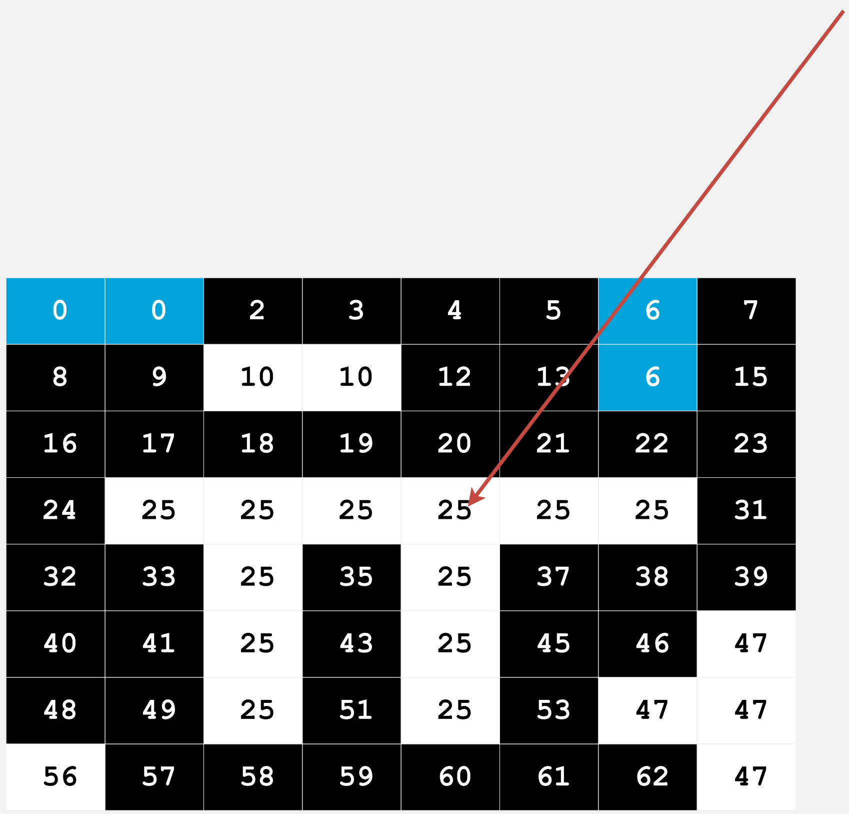
$N = 8$

UF solution to find percolation threshold

Q. How to declare a new site open?

A. Take union of new site and all adjacent open sites.

open this site



full open site (connected to top)

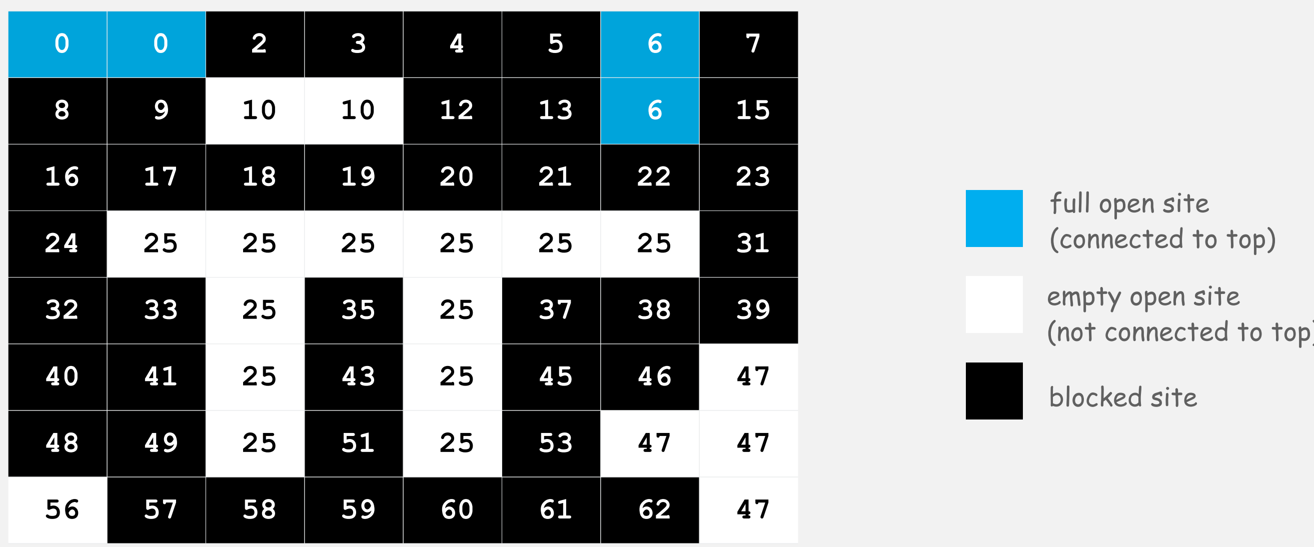
empty open site (not connected to top)

blocked site

$N = 8$

UF solution: a critical optimization

Q. How to avoid checking all pairs of top and bottom sites?

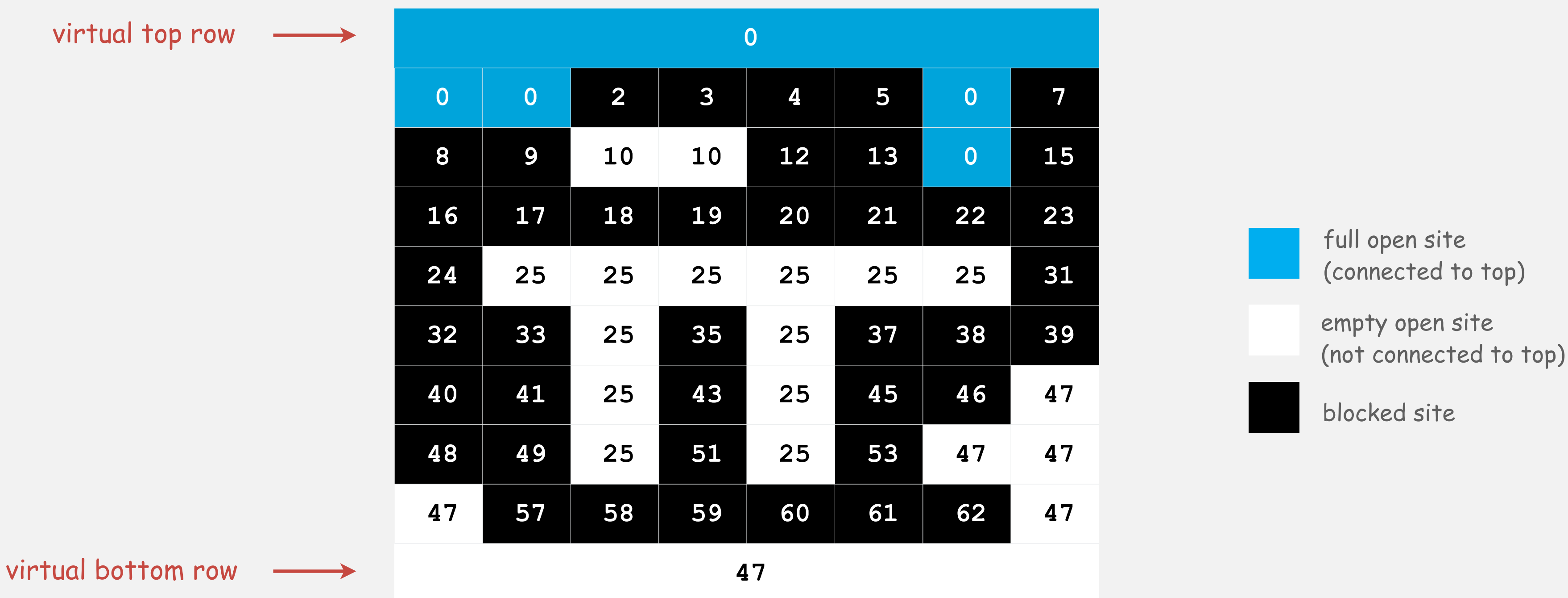


$N = 8$

UF solution: a critical optimization

Q. How to avoid checking all pairs of top and bottom sites?

A. Create a virtual top and bottom objects;
   system percolates when virtual top and bottom objects are in same set.



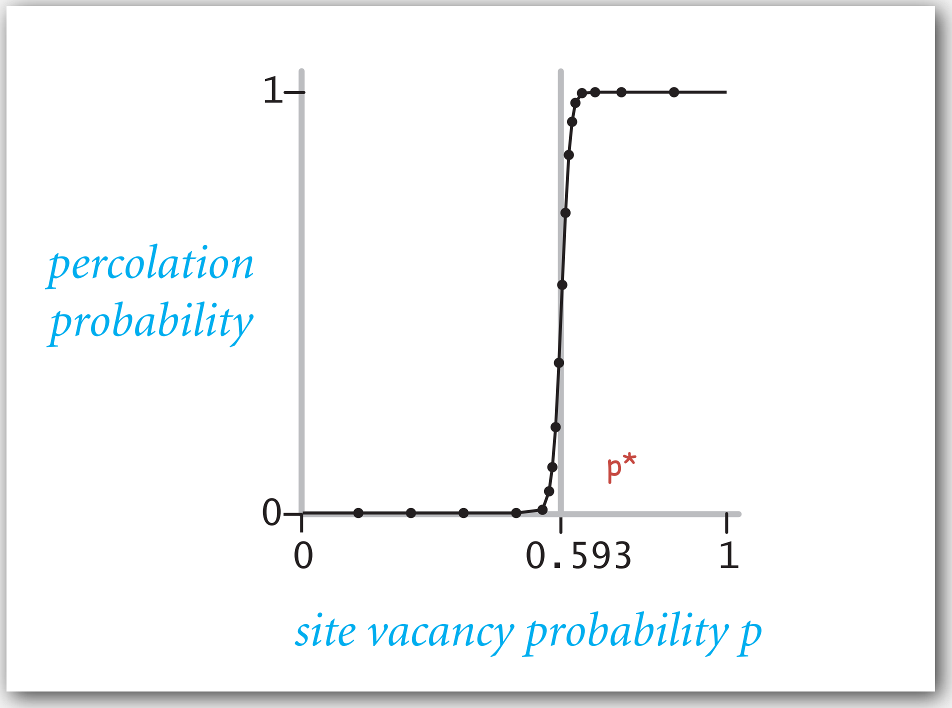
$N = 8$

Percolation threshold

Q. What is percolation threshold p* ?

A. About 0.592746 for large square lattices.

percolation constant known only via simulation



Subtext of today's lecture (and this course)

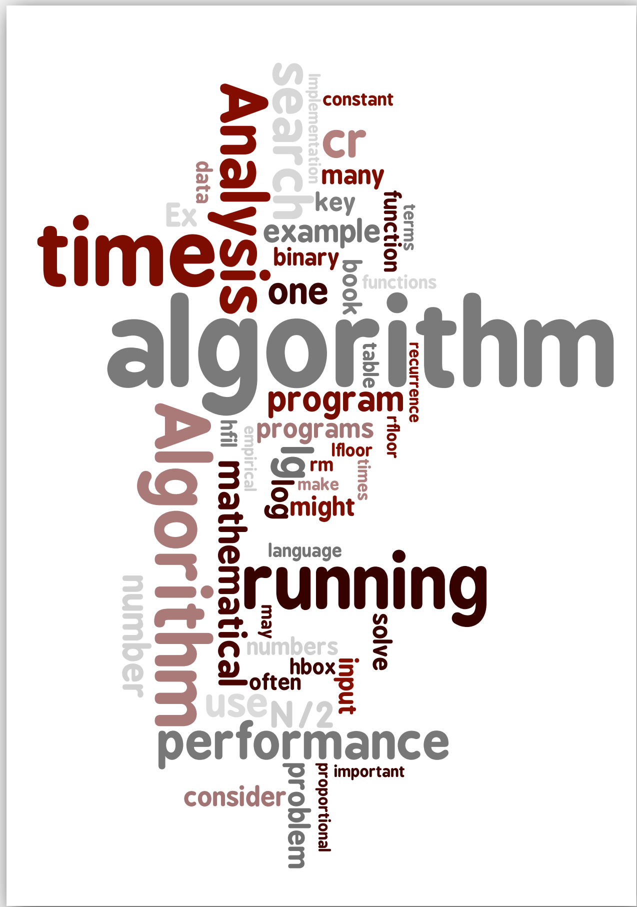
Steps to developing a usable algorithm.

- Model the problem.
- Find an algorithm to solve it.
- Fast enough? Fits in memory?
- If not, figure out why.
- Find a way to address the problem.
- Iterate until satisfied.

The scientific method.

Mathematical analysis.

1.4 Analysis of Algorithms



- estimating running time
- mathematical analysis
- order-of-growth hypotheses
- input models
- measuring space

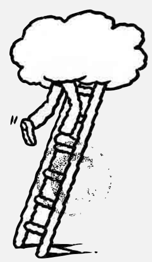
Reference: Intro to Programming in Java, Section 4.1

Cast of characters

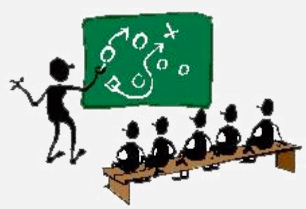
Programmer needs to develop a working solution.

Student might play any or all of these roles someday.

Client wants problem solved efficiently.



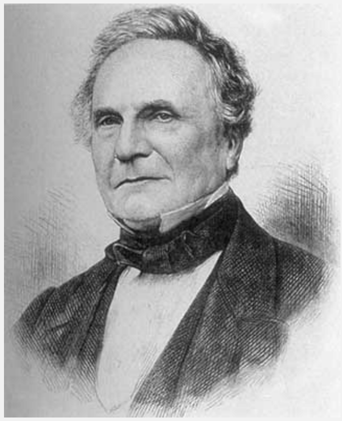
Theoretician wants to understand.



Basic blocking and tackling is sometimes necessary. [this lecture]

Running time

> *" As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time? "* — Charles Babbage



Charles Babbage (1864)

Analytic Engine

how many times do you have to turn the crank?

Reasons to analyze algorithms

Predict performance.

Compare algorithms.

Provide guarantees.

Understand theoretical basis.

this course (COS 226)

theory of algorithms (COS 423)

Primary practical reason: avoid performance bugs.

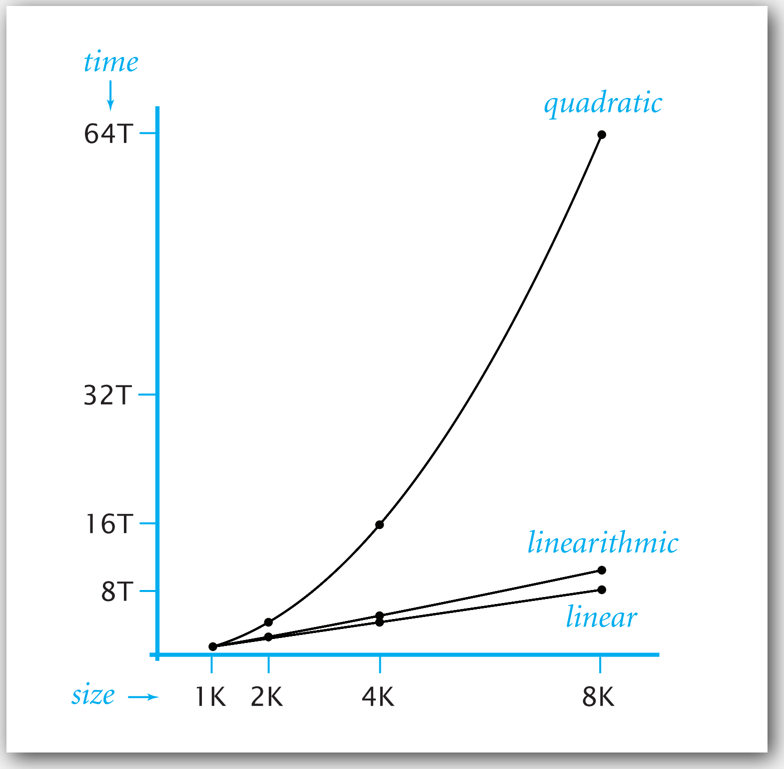
client gets poor performance because programmer did not understand performance characteristics

Some algorithmic successes

Discrete Fourier transform.

- Break down waveform of N samples into periodic components.
- Applications: DVD, JPEG, MRI, astrophysics, ….
- Brute force: N^2 steps.
- FFT algorithm: N log N steps, enables new technology.

Friedrich Gauss
1805

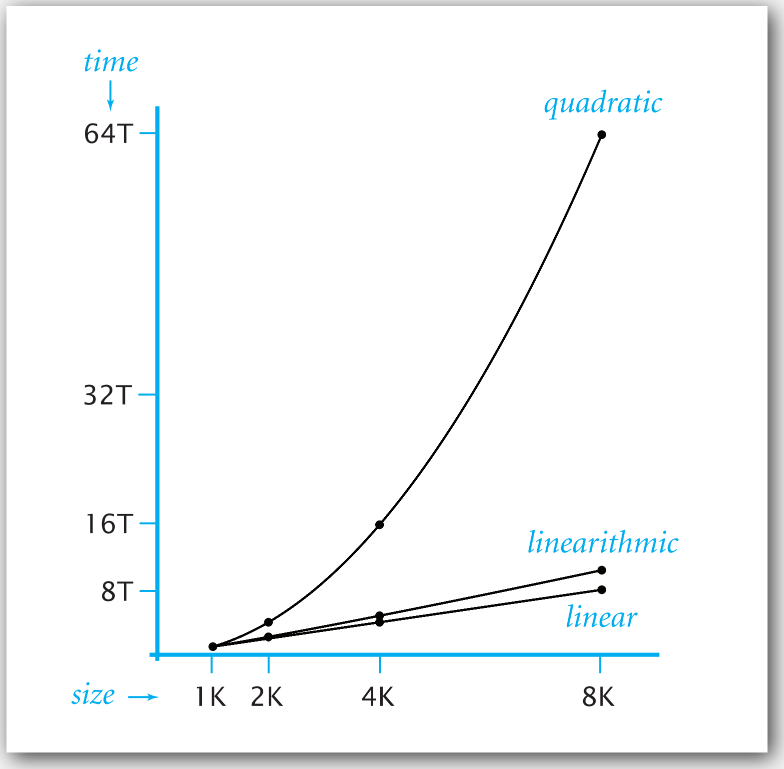


Some algorithmic successes

N-body Simulation.

- Simulate gravitational interactions among N bodies.
- Brute force: N^2 steps.
- Barnes-Hut: N log N steps, enables new research.

Andrew Appel
PU '81



- **estimating running time**
- mathematical analysis
- order-of-growth hypotheses
- input models
- measuring space

Scientific analysis of algorithms

A framework for predicting performance and comparing algorithms.

Scientific method.

- Observe some feature of the universe.
- Hypothesize a model that is consistent with observation.
- Predict events using the hypothesis.
- Verify the predictions by making further observations.
- Validate by repeating until the hypothesis and observations agree.

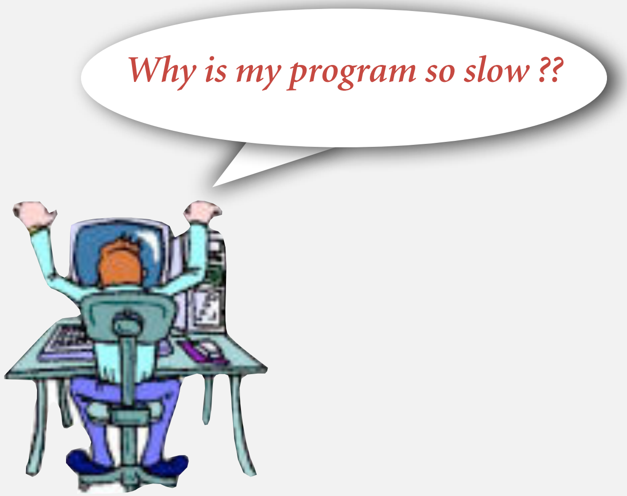
Principles.

- Experiments must be reproducible.
- Hypotheses must be falsifiable.

Universe = computer itself.

Experimental algorithmics

Every time you run a program you are doing an experiment!



First step. Debug your program!

Second step. Choose input model for experiments.

Third step. Run and time the program for problems of increasing size.

Example: 3-sum

3-sum. Given N integers, find all triples that sum to exactly zero.

```
% more input8.txt
8
 30 -30 -20 -10 40 0 10 5

% java ThreeSum < input8.txt
 4
 30 -30   0
 30 -20 -10
-30 -10  40
-10   0  10
```

Context. Deeply related to problems in computational geometry.

3-sum: brute-force algorithm

```
public class ThreeSum
{
   public static int count(int[] a)
   {
      int N = a.length;
      int cnt = 0;
      for (int i = 0; i < N; i++)
         for (int j = i+1; j < N; j++)
            for (int k = j+1; k < N; k++)
               if (a[i] + a[j] + a[k] == 0)
                  cnt++;
      return cnt;
   }

   public static void main(String[] args)
   {
      long[] a = StdArrayIO.readInt1D();
      StdOut.println(count(a));
   }
}
```

check each triple

ignore overflow

Empirical analysis

Run the program for various input sizes and measure running time.

ThreeSum.java

N	time (seconds) †
1000	0.26
2000	2.16
4000	17.18
8000	137.76

† Running Linux on Sun-Fire-X4100

Measuring the running time

Q. How to time a program?

A. Manual.

```
% java ThreeSum < 1Kints.txt
```

tick tick tick

```
0
% java ThreeSum < 2Kints.txt
```

tick tick tick tick tick tick
tick tick tick tick tick tick
tick tick tick tick tick tick
tick tick tick tick tick tick

```
2
391930676 -763182495 371251819
-326747290 802431422 -475684132
```

Measuring the running time

Q. How to time a program?

A. Automatic.

```
Stopwatch stopwatch = new Stopwatch();

ThreeSum.count(a);

double time = stopwatch.elapsedTime();
StdOut.println("Running time: " + time + " seconds");
```

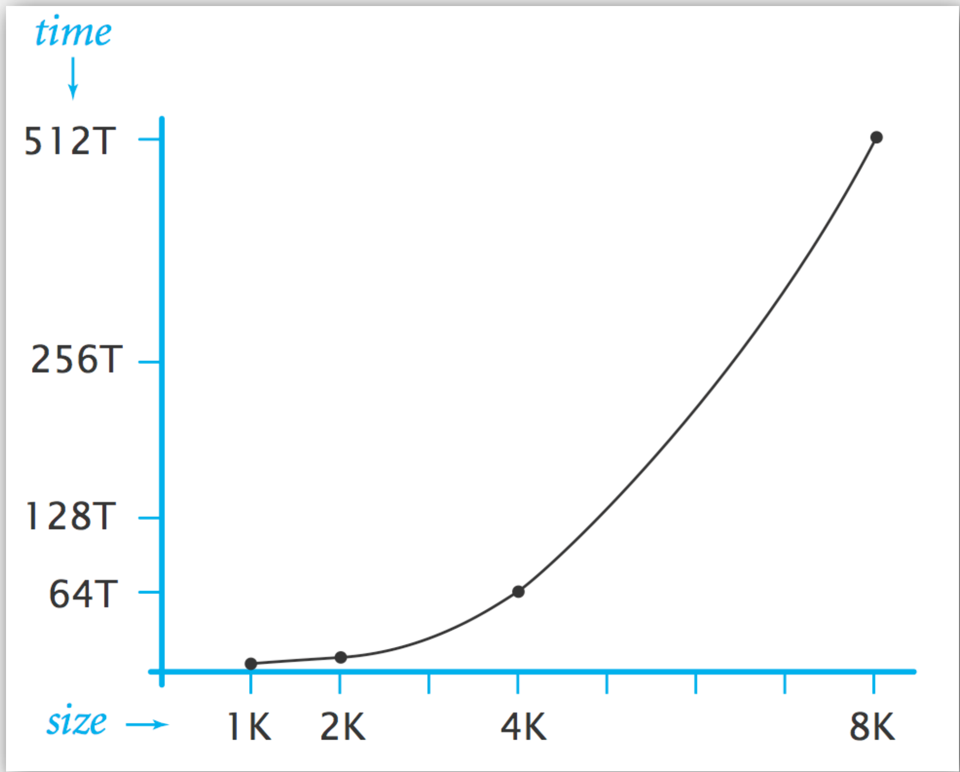
client code

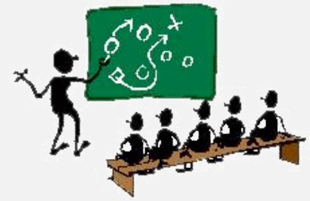
```
public class Stopwatch
{
   private final long start = System.currentTimeMillis();

   public double elapsedTime()
   {
      long now = System.currentTimeMillis();
      return (now - start) / 1000.0;
   }
}
```

implementation (part of `stdlib.jar`, see `http://www.cs.princeton.edu/introcs/stdlib`)

Data analysis

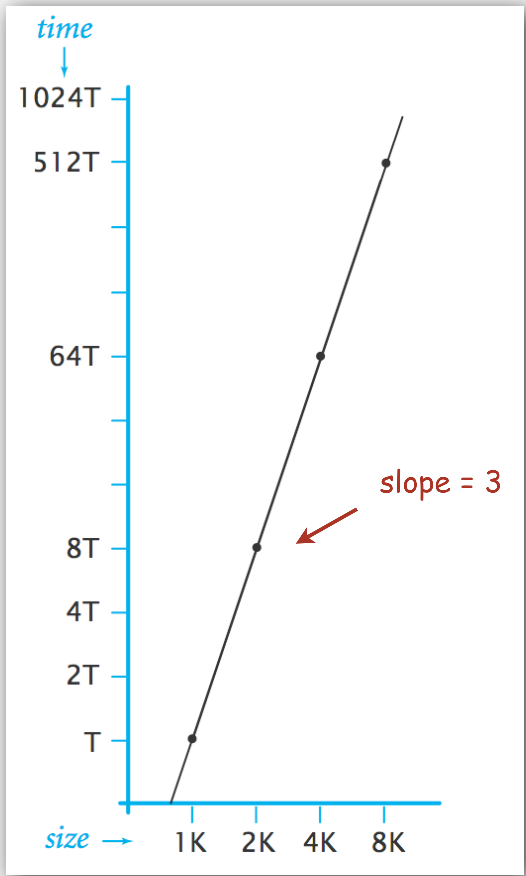
Plot running time as a function of input size N.





Data analysis

Log-log plot. Plot running time vs. input size N on log-log scale.



power law

Regression. Fit straight line through data points: $a N^b$.

slope

Hypothesis. Running time grows with the cube of the input size: $a N^3$.

Doubling hypothesis

Doubling hypothesis. Quick way to estimate b in a power law hypothesis.

Run program, doubling the size of the input.

N	time (seconds) †	ratio	lg ratio
500	0.03	-	
1,000	0.26	7.88	2.98
2,000	2.16	8.43	3.08
4,000	17.18	7.96	2.99
8,000	137.76	7.96	2.99

seems to converge to a constant b ≈ 3

Hypothesis. Running time is about $a N^b$ with b = lg ratio.

Caveat. Can't identify logarithmic factors with doubling hypothesis.

Prediction and verification

Hypothesis. Running time is about $a\,N^3$ for input of size N.

Q. How to estimate a?

A. Run the program!

N	time (seconds)
4,000	17.18
4,000	17.15
4,000	17.17

$$17.17 = a \times 4000^3$$
$$\Rightarrow \quad a = 2.7 \times 10^{-10}$$

Refined hypothesis. Running time is about $2.7 \times 10^{-10} \times N^3$ seconds.

Prediction. $1{,}100$ seconds for $N = 16{,}000$.

Observation.

N	time (seconds)
16384	1118.86

validates hypothesis!

Experimental algorithmics

Many obvious factors affect running time:

- Machine.
- Compiler.
- Algorithm.
- Input data.

More factors (not so obvious):

- Caching.
- Garbage collection.
- Just-in-time compilation.
- CPU use by other applications.

Bad news. It is often difficult to get precise measurements.

Good news. Easier than other sciences.

e.g., can run huge number of experiments

War story (from COS 126)

Q. How long does this program take as a function of N?

```
public class EditDistance
{
   String s = StdIn.readString();
   int N = s.length();
   ...
    for (int i = 0; i < N; i++)
       for (int j = 0; j < N; j++)
          distance[i][j] = ...
    ...
}
```

N	time
1,000	0.11
2,000	0.35
4,000	1.6
8,000	6.5

Jenny

N	time
250	0.5
500	1.1
1,000	1.9
2,000	3.9

Kenny

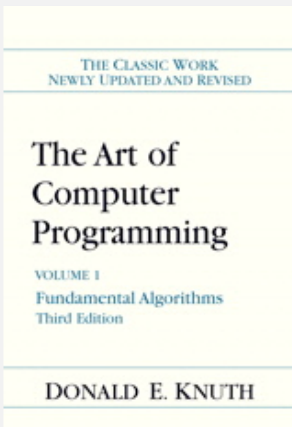
Jenny. $\sim c_1 N^2$ seconds.

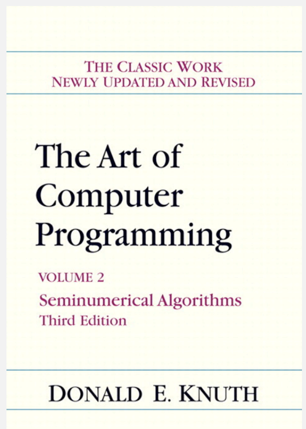
Kenny. $\sim c_2 N$ seconds.

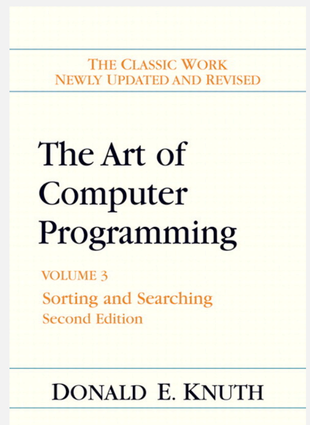
- estimating running time
- **mathematical analysis**
- order-of-growth hypotheses
- input models
- measuring space

Total running time: sum of cost × frequency for all operations.

- Need to analyze program to determine set of operations.
- Cost depends on machine, compiler.
- Frequency depends on algorithm, input data.

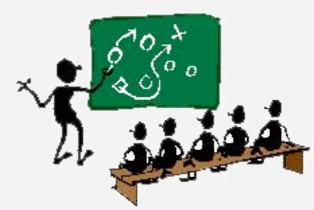







Donald Knuth
1974 Turing Award

In principle, accurate mathematical models are available.



Cost of basic operations

operation	example	nanoseconds †
integer add	`a + b`	2.1
integer multiply	`a * b`	2.4
integer divide	`a / b`	5.4
floating point add	`a + b`	4.6
floating point multiply	`a * b`	4.2
floating point divide	`a / b`	13.5
sine	`Math.sin(theta)`	91.3
arctangent	`Math.atan2(y, x)`	129.0
...	...	...

† Running OS X on Macbook Pro 2.2GHz with 2GB RAM

Cost of basic operations

operation	example	nanoseconds †
variable declaration	`int a`	c_1
assignment statement	`a = b`	c_2
integer compare	`a < b`	c_3
array element access	`a[i]`	c_4
array length	`a.length`	c_5
1D array allocation	`new int[N]`	$c_6\ N$
2D array allocation	`new int[N][N]`	$c_7\ N^2$
string length	`s.length()`	c_8
substring extraction	`s.substring(N/2, N)`	c_9
string concatenation	`s + t`	$c_{10}\ N$

Novice mistake. Abusive string concatenation.

Example: 1-sum

Q. How many instructions as a function of N?

```
int count = 0;
for (int i = 0; i < N; i++)
   if (a[i] == 0) count++;
```

operation	frequency
variable declaration	2
assignment statement	2
less than compare	$N + 1$
equal to compare	N
array access	N
increment	$\leq 2N$

between N (no zeros)
and 2N (all zeros)

Example: 2-sum

Q. How many instructions as a function of N?

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      if (a[i] + a[j] == 0) count++;
```

operation	frequency
variable declaration	$N + 2$
assignment statement	$N + 2$
less than compare	$1/2\ (N + 1)\ (N + 2)$
equal to compare	$1/2\ N\ (N - 1)$
array access	$N\ (N - 1)$
increment	$\leq\ N^2$

$$0 + 1 + 2 + \ldots + (N - 1) = \frac{1}{2} N\,(N - 1) = \binom{N}{2}$$

tedious to count exactly

Tilde notation

- Estimate running time (or memory) as a function of input size N.
- Ignore lower order terms.
  - when N is large, terms are negligible
  - when N is small, we don't care

Ex 1. $6N^3 + 20N + 16 \sim 6N^3$

Ex 2. $6N^3 + 100N^{4/3} + 56 \sim 6N^3$

Ex 3. $6N^3 + \underbrace{17N^2 \lg N + 7N}_{\text{discard lower-order terms}} \sim 6N^3$

discard lower-order terms
(e.g., N = 1000: 6 billion vs. 169 million)

Technical definition. $f(N) \sim g(N)$ means $\displaystyle\lim_{N \to \infty} \frac{f(N)}{g(N)} = 1$

Example: 2-sum

Q. How long will it take as a function of N?

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      if (a[i] + a[j] == 0) count++;
```

"inner loop"

operation	frequency	time per op	total time
variable declaration	$\sim N$	c_1	$\sim c_1 N$
assignment statement	$\sim N$	c_2	$\sim c_2 N$
less than comparison	$\sim 1/2\ N^2$	c_3	$\sim c_3 N^2$
equal to comparison	$\sim 1/2\ N^2$		
array access	$\sim N^2$	c_4	$\sim c_4 N^2$
increment	$\leq N^2$	c_5	$\leq c_5 N^2$
total			$\sim c N^2$

depends on input data

Example: 3-sum

Q. How many instructions as a function of N?

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      for (int k = j+1; k < N; k++)
         if (a[i] + a[j] + a[k] == 0)
            count++;
```

~ 1

$\sim N$

$\sim N^2 / 2$

$$\binom{N}{3} = \frac{N(N-1)(N-2)}{3!} \sim \frac{1}{6}N^3$$

"inner loop"

may be in inner loop, depends on input data

Remark. Focus on instructions in inner loop; ignore everything else!

Bounding the sum by an integral trick

Q. How to estimate a discrete sum?

A1. Take COS 340.

A2. Replace the sum with an integral, and use calculus!

Ex 1. 1 + 2 + ... + N.

$$\sum_{i=1}^{N} i \;\sim\; \int_{x=1}^{N} x\,dx \;\sim\; \frac{1}{2}N^2$$

Ex 2. 1 + 1/2 + 1/3 + ... + 1/N.

$$\sum_{i=1}^{N} \frac{1}{i} \;\sim\; \int_{x=1}^{N} \frac{1}{x} dx \;=\; \ln N$$

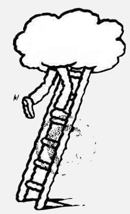
Ex 3. 3-sum triple loop.

$$\sum_{i=1}^{N}\sum_{j=i}^{N}\sum_{k=j}^{N} 1 \;\sim\; \int_{x=1}^{N}\int_{y=x}^{N}\int_{z=y}^{N} dz\,dy\,dx \;\sim\; \frac{1}{6}N^3$$

In principle, accurate mathematical models are available.

In practice,

- Formulas can be complicated.
- Advanced mathematics might be required.
- Exact models best left for experts.



costs (depend on machine, compiler)

Text

$$T_N = c_1 A + c_2 B + c_3 C + c_4 D + c_5 E$$

A = variable declarations
B = assignment statements
C = compare
D = array access
E = increment

frequencies
(depend on algorithm, input)

Bottom line. We use approximate models in this course: $T_N \sim c\ N^3$.

Common order-of-growth hypotheses

To determine order-of-growth:

- Assume a power law $T_N \sim a\,N^{\,b}$.
- Estimate exponent b with doubling hypothesis.
- Validate with mathematical analysis.

Ex. `ThreeSumDeluxe.java`

Food for precept. How is it implemented?

N	time (seconds)
1,000	0.26
2,000	2.16
4,000	17.18
8,000	137.76

`ThreeSum.java`

N	time (seconds)
1,000	0.43
2,000	0.53
4,000	1.01
8,000	2.87
16,000	11.00
32,000	44.64
64,000	177.48

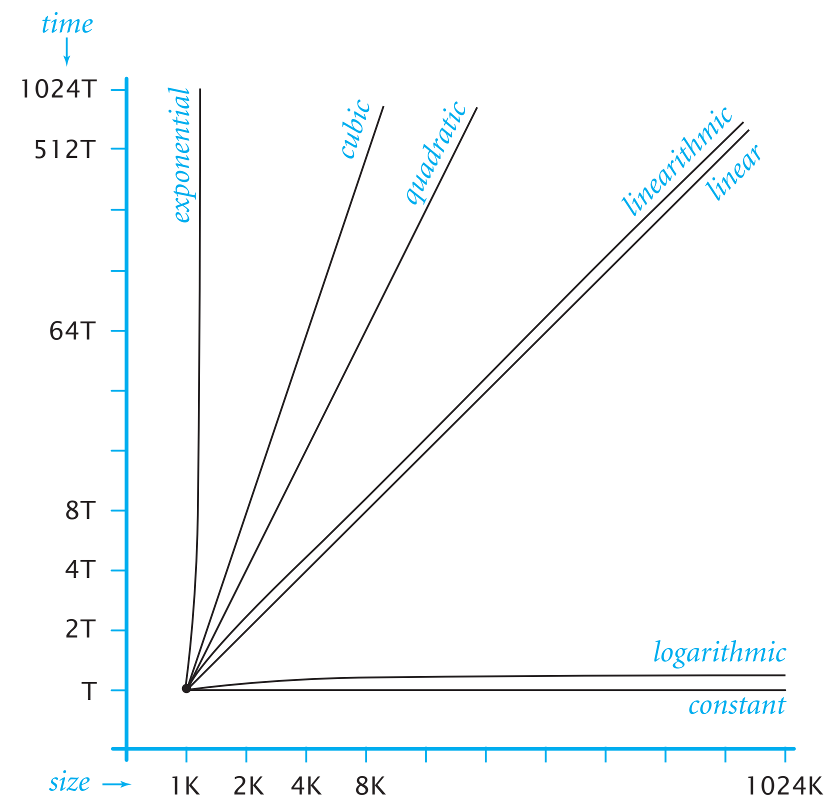
`ThreeSumDeluxe.java`

Common order-of-growth hypotheses

Good news. the small set of functions

$$1,\ \log N,\ N,\ N \log N,\ N^2,\ N^3, \text{ and } 2^N$$

suffices to describe order-of-growth of typical algorithms.



Orders of growth (log-log plot)

Common order-of-growth hypotheses

growth rate	name	typical code framework	description	example	T(2N) / T(N)
1	constant	`a = b + c;`	statement	add two numbers	1
log N	logarithmic	`while (N > 1) { N = N / 2; ... }`	divide in half	binary search	~ 1
N	linear	`for (int i = 0; i < N; i++) { ... }`	loop	find the maximum	2
N log N	linearithmic	[see mergesort lecture]	divide and conquer	mergesort	~ 2
N^2	quadratic	`for (int i = 0; i < N; i++)` `for (int j = 0; j < N; j++)` `{ ... }`	double loop	check all pairs	4
N^3	cubic	`for (int i = 0; i < N; i++)` `for (int j = 0; j < N; j++)` `for (int k = 0; k < N; k++)` `{ ... }`	triple loop	check all triples	8
2^N	exponential	[see combinatorial search lecture]	exhaustive search	check all possibilities	T(N)

Practical implications of order-of-growth

growth rate	name	description	effect on a program that runs for a few seconds	
			time for 100x more data	size for 100x faster computer
1	constant	independent of input size	-	-
log N	logarithmic	nearly independent of input size	-	-
N	linear	optimal for N inputs	a few minutes	100x
N log N	linearithmic	nearly optimal for N inputs	a few minutes	100x
N^2	quadratic	not practical for large problems	several hours	10x
N^3	cubic	not practical for medium problems	several weeks	4-5x
2^N	exponential	useful only for tiny problems	forever	1x

- estimating running time
- mathematical analysis
- order-of-growth hypotheses
- **input models**
- measuring space

Types of analyses

Best case. Lower bound on cost.

- Determined by "easiest" input.
- Provides a goal for all inputs.

Worst case. Upper bound on cost.

- Determined by "most difficult" input.
- Provides guarantee for all inputs.

Average case. "Expected" cost.

- Need a model for "random" input.
- Provides a way to predict performance.

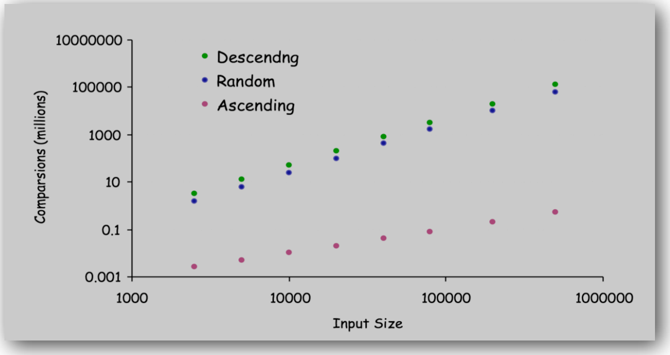
Ex 1. Array accesses for brute-force 3-sum.

- Best: $\sim \frac{1}{2}N^3$
- Average: $\sim \frac{1}{2}N^3$
- Worst: $\sim \frac{1}{2}N^3$

Ex 2. Compares for insertion sort.

- Best (ascending order): $\sim N$.
- Average (random order): $\sim \frac{1}{4} N^2$
- Worst (descending order): $\sim \frac{1}{2}N^2$

(details in Lecture 4)



Commonly-used notations

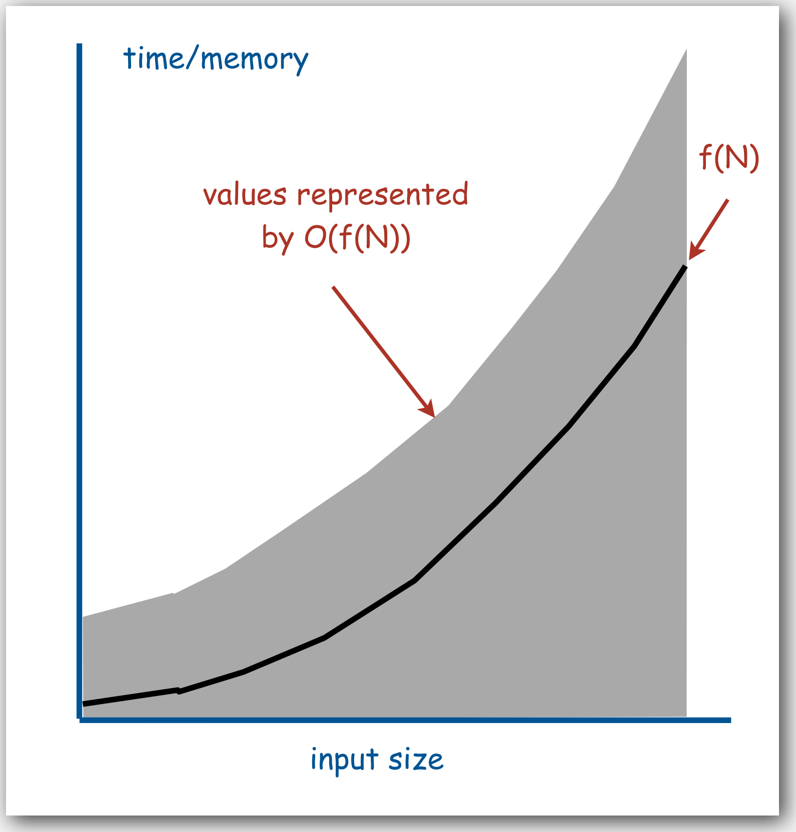
notation	provides	example	shorthand for	used to
Tilde	leading term	$\sim 10\ N^2$	$10\ N^2$ $10\ N^2 + 22\ N \log N$ $10\ N^2 + 2\ N + 37$	provide approximate model
Big Theta	asymptotic growth rate	$\Theta(N^2)$	N^2 $9000\ N^2$ $5\ N^2 + 22\ N \log N + 3N$	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	N^2 $100\ N$ $22\ N \log N + 3\ N$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$9000\ N^2$ N^5 $N^3 + 22\ N \log N + 3\ N$	develop lower bounds

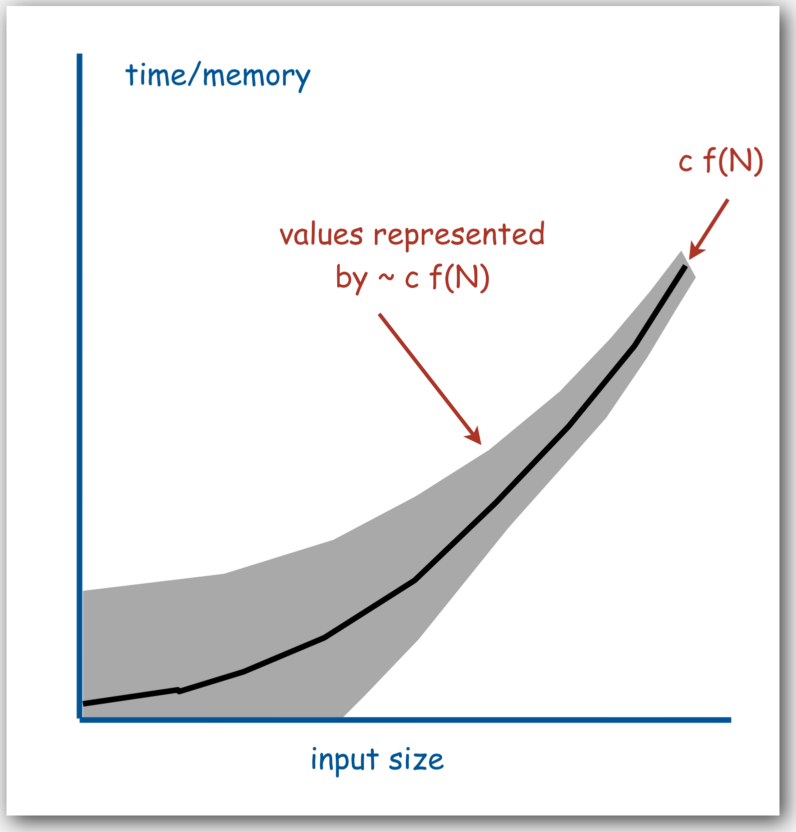
Common mistake. Interpreting big-Oh as an approximate model.

Tilde notation vs. big-Oh notation

We use tilde notation whenever possible.

- Big-Oh notation suppresses leading constant.
- Big-Oh notation only provides upper bound (not lower bound).





- estimating running time
- mathematical analysis
- order-of-growth hypotheses
- input models
- **measuring space**

Typical memory requirements for primitive types in Java

Bit. 0 or 1.

Byte. 8 bits.

Megabyte (MB). 1 million bytes.

Gigabyte (GB). 1 billion bytes.

type	bytes
`boolean`	1
`byte`	1
`char`	2
`int`	4
`float`	4
`long`	8
`double`	8

Typical memory requirements for arrays in Java

Array overhead. 16 bytes.

type	bytes
`char[]`	$2N + 16$
`int[]`	$4N + 16$
`double[]`	$8N + 16$

one-dimensional arrays

type	bytes
`char[][]`	$2N^2 + 20N + 16$
`int[][]`	$4N^2 + 20N + 16$
`double[][]`	$8N^2 + 20N + 16$

two-dimensional arrays

Ex. An N-by-N array of doubles consumes $\sim 8N^2$ bytes of memory.

Typical memory requirements for objects in Java

Object overhead. 8 bytes.

Reference. 4 bytes.

Ex 1. A `Complex` object consumes 24 bytes of memory.

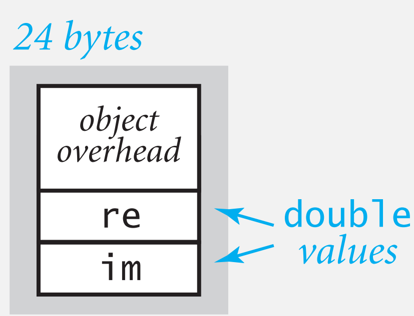
```
public class Complex
{
   private double re;
   private double im;
   ...
}
```

8 bytes overhead for object

8 bytes

8 bytes

24 bytes



Typical memory requirements for objects in Java

Object overhead. 8 bytes.

Reference. 4 bytes.

Ex 2. A virgin `String` of length N consumes ~ 2N bytes of memory.

```
public class String
{
   private int offset;
   private int count;
   private int hash;
   private char[] value;
   ...
}
```

8 bytes overhead for object

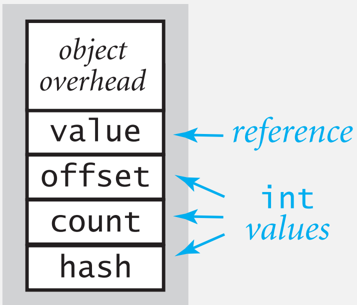
4 bytes

4 bytes

4 bytes

4 bytes for reference
(plus 2N + 16 bytes for array)

2N + 40 bytes



Example 1

Q. How much memory does `QuickUWPC` use as a function of N ?

A.

```
public class QuickUWPC
{
   private int[] id;
   private int[] sz;

   public QuickUWPC(int N)
   {
      id = new int[N];
      sz = new int[N];
      for (int i = 0; i < N; i++) id[i] = i;
      for (int i = 0; i < N; i++) sz[i] = 1;
   }

   public boolean find(int p, int q)
   { ... }

   public void unite(int p, int q)
   { ... }
}
```

Example 2

Q. How much memory does this code fragment use as a function of N?

A.

```
...
int N = Integer.parseInt(args[0]);
for (int i = 0; i < N; i++) {
    int[] a = new int[N];
    ...
}
```

Remark. Java automatically reclaims memory when it is no longer in use.

not always easy for Java to know

Turning the crank: summary

In principle, accurate mathematical models are available.

In practice, approximate mathematical models are easily achieved.

Timing may be flawed?

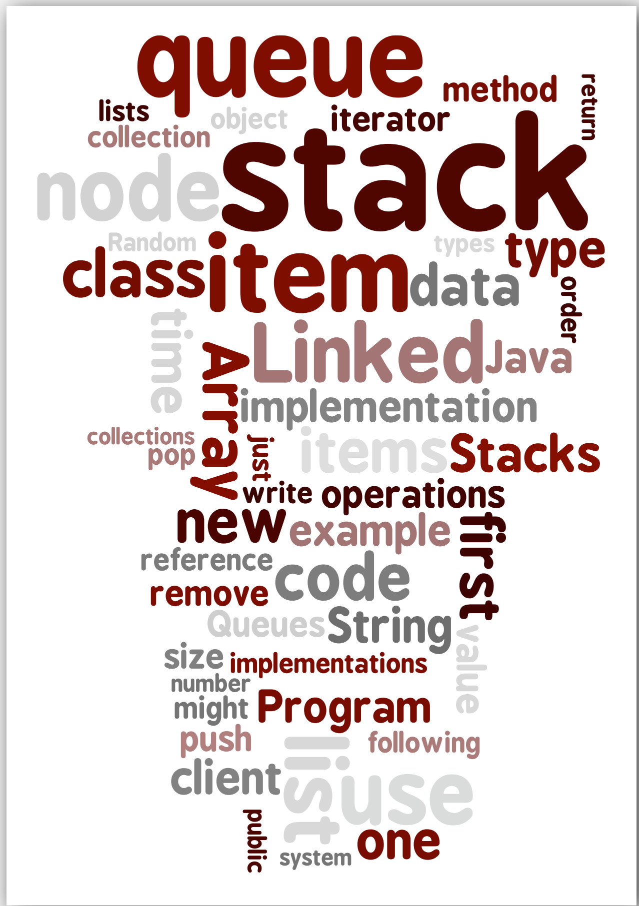
- Limits on experiments insignificant compared to other sciences.

- Mathematics might be difficult?
- Only a few functions seem to turn up.
- Doubling hypothesis cancels complicated constants.

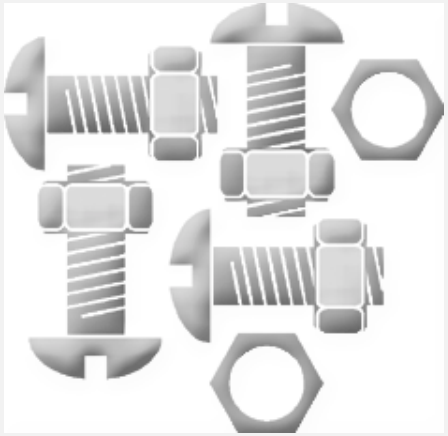
Actual data might not match input model?

- Need to understand input to effectively process it.
- Approach 1: design for the worst case.
- Approach 2: randomize, depend on probabilistic guarantee.

1.3 Stacks and Queues



- stacks
- dynamic resizing
- queues
- generics
- iterators
- applications



Fundamental data types.

- Values: sets of objects
- Operations: insert, remove, test if empty.
- Intent is clear when we insert.
- Which item do we remove?

LIFO = "last in first out"

Stack. Remove the item most recently added.

Analogy. Cafeteria trays, Web surfing.

FIFO = "first in first out"

Queue. Remove the item least recently added.

Analogy. Registrar's line.

Client, implementation, interface

Separate interface and implementation.

Ex: stack, queue, priority queue, symbol table, union-find, ...

Benefits.

- Client can't know details of implementation $\Rightarrow$ client has many implementation from which to choose.
- Implementation can't know details of client needs $\Rightarrow$ many clients can re-use the same implementation.
- Design: creates modular, reusable libraries.
- Performance: use optimized implementation where it matters.

> Client: program using operations defined in interface.
> Implementation: actual code implementing operations.
> Interface: description of data type, basic operations.

- stacks
- dynamic resizing
- queues
- generics
- iterators
- applications

Stacks

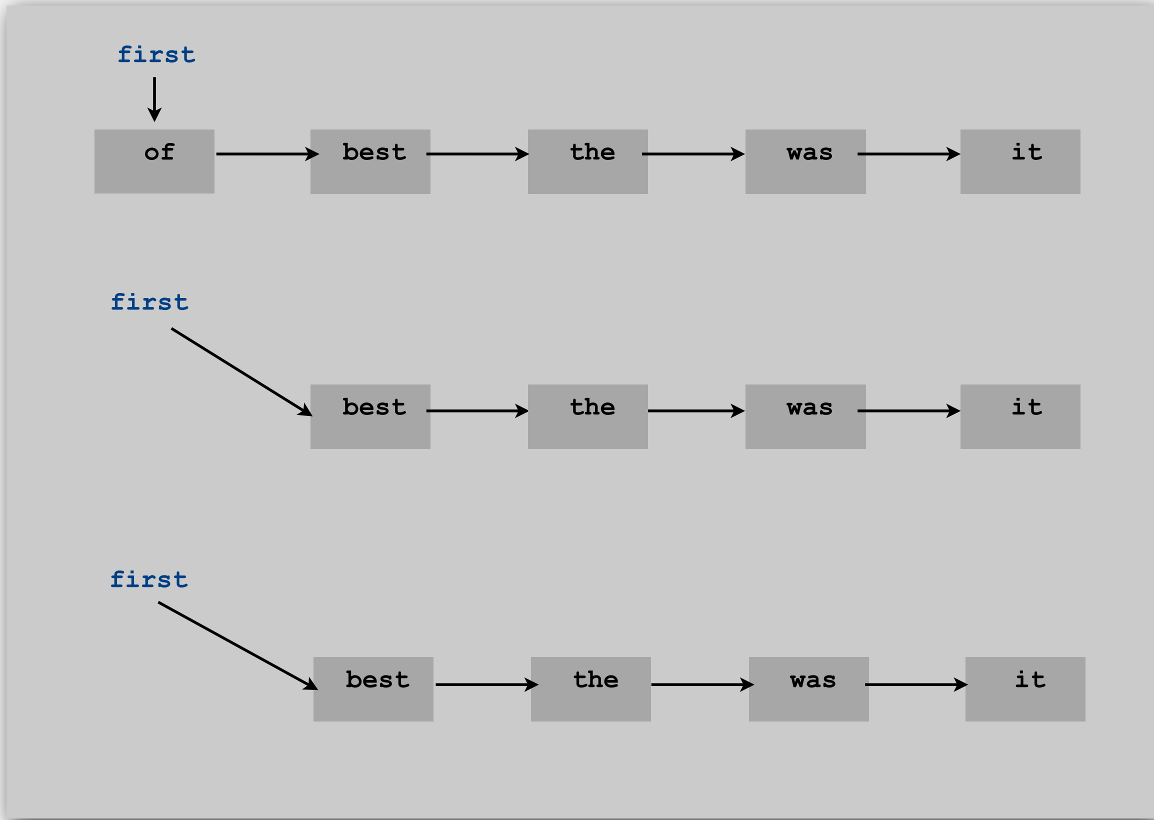
Stack operations.

- `push()` Insert a new item onto stack.
- `pop()` Remove and return the item most recently added.
- `isEmpty()` Is the stack empty?


```
public static void main(String[] args)
{
   StackOfStrings stack = new StackOfStrings();
   while (!StdIn.isEmpty())
   {
      String item = StdIn.readString();
      if (item.equals("-")) StdOut.print(stack.pop());
      else                  stack.push(item);
   }
}
```

```
% more tobe.txt
to be or not to - be - - that - - - is

% java StackOfStrings < tobe.txt
to be not that or be
```

Stack pop: linked-list implementation




```
String item = first.item;
```

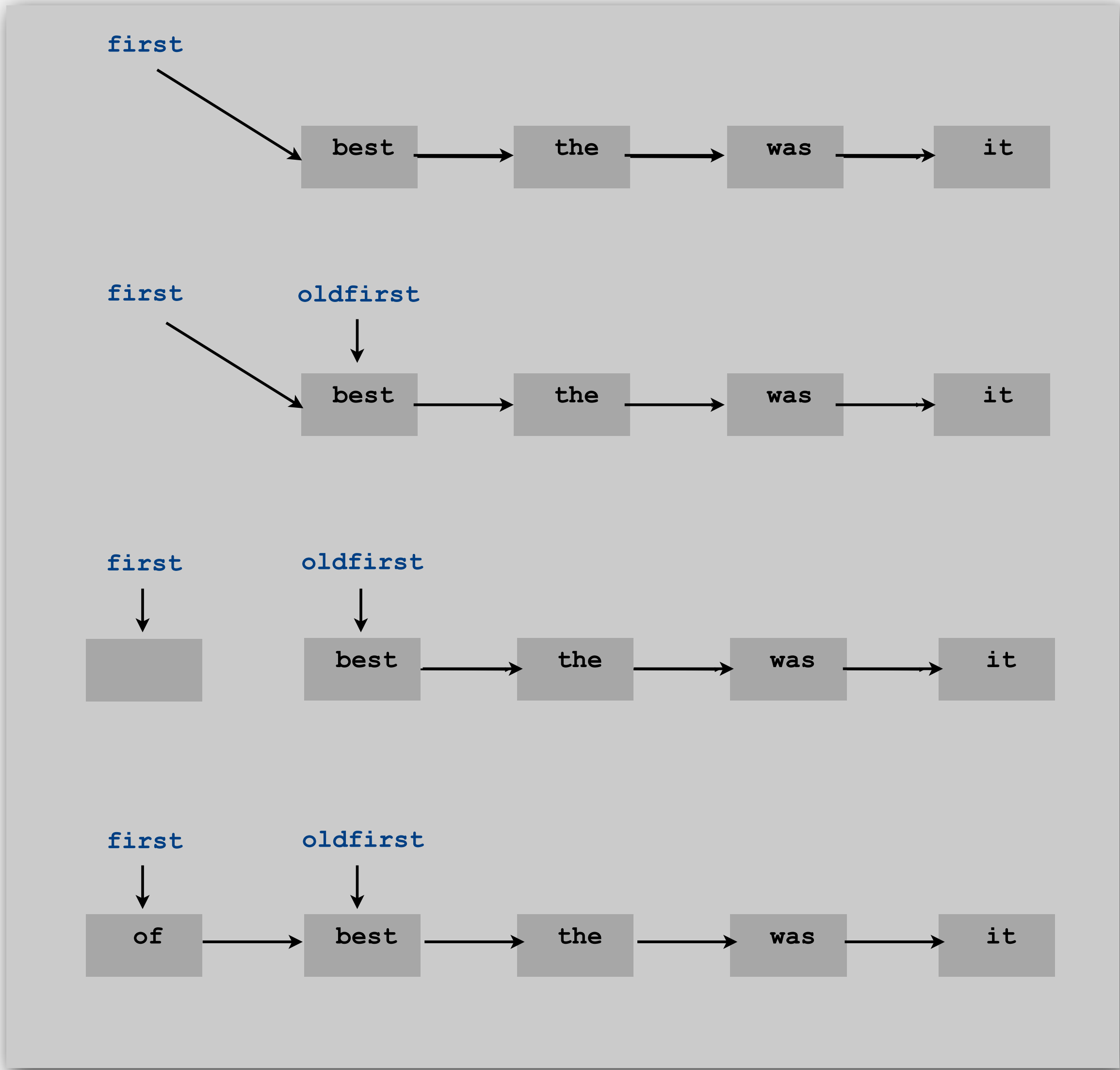
"of"

```
first = first.next;
```

```
return item;
```

"of"

Stack push: linked-list implementation




```
Node oldfirst = first;
```

```
first = new Node();
```

```
first.item = "of";
first.next = oldfirst;
```

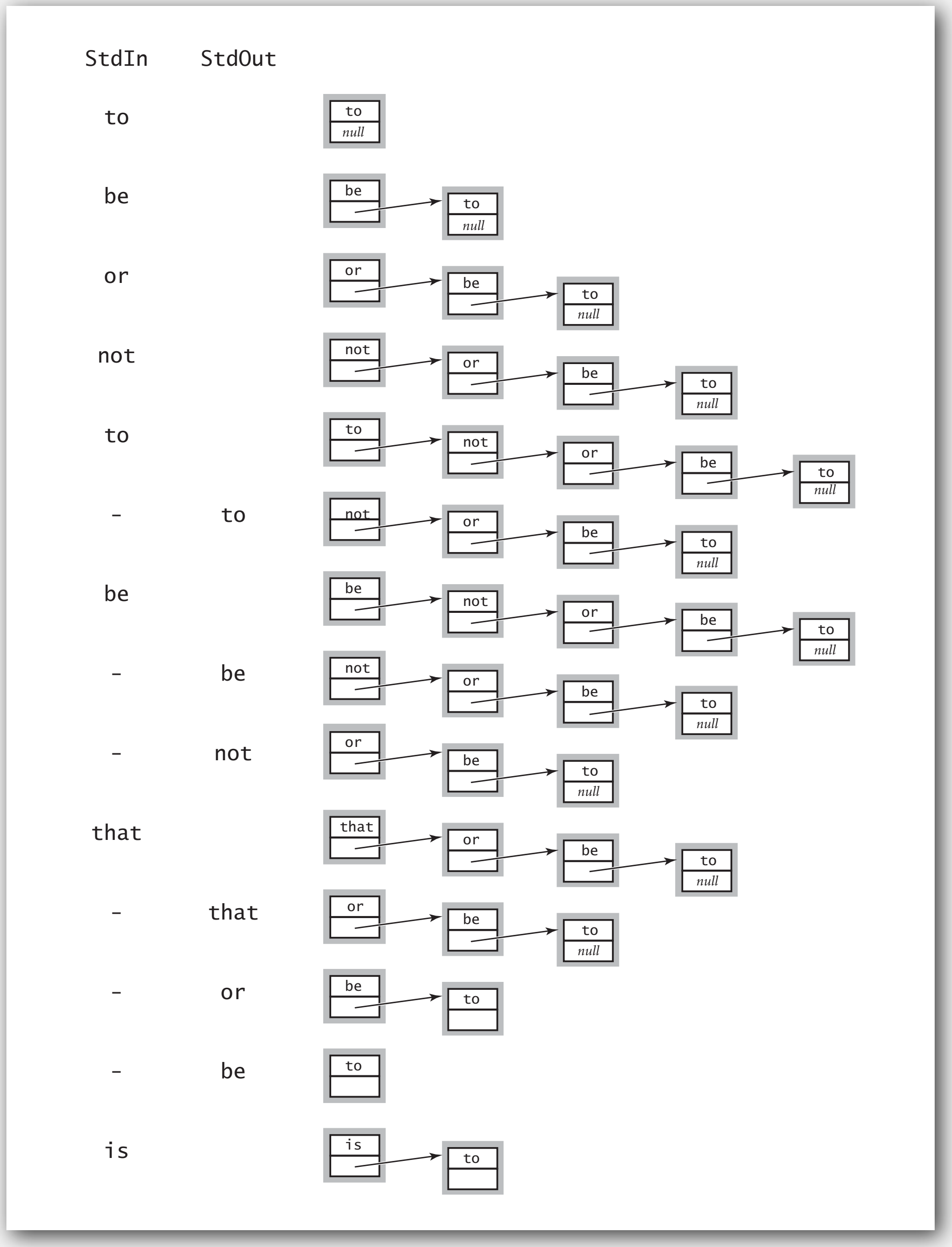
Stack: linked-list implementation

```
public class StackOfStrings
{
   private Node first = null;

   private class Node
   {
      String item;
      Node next;
   }

   public boolean isEmpty()
   {  return first == null;  }

   public void push(String item)
   {
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public String pop()
   {
      if (isEmpty()) throw new RuntimeException();
      String item = first.item;
      first = first.next;
      return item;
   }
}
```

"inner class"

stack underflow

Stack: linked-list trace



Stack: array implementation

Array implementation of a stack.

- Use array `s[]` to store `N` items on stack.
- `push()`: add new item at `s[N]`.
- `pop()`: remove item from `s[N-1]`.

`s[]`	`it`	`was`	`the`	`best`	`of`	`times`	*null*	*null*	*null*	*null*
	0	1	2	3	4	5	6	7	8	9
							`N`			`capacity = 10`

Stack: array implementation

```
public class StackOfStrings
{
   private String[] s;
   private int N = 0;

   public StackOfStrings(int capacity)
   {  s = new String[capacity];  }

   public boolean isEmpty()
   {  return N == 0;  }

   public void push(String item)
   {  s[N++] = item;  }

   public String pop()
   {  return s[--N];  }
}
```

a cheat
(stay tuned)

decrement N;
then use to index into array

```
public String pop()
{
   String item = s[--N];
   s[N] = null;
   return item;
}
```

this version avoids "loitering"

garbage collector only reclaims memory
if no outstanding references

- stacks
- **dynamic resizing**
- queues
- generics
- iterators
- applications

Stack: dynamic array implementation

Problem. Requiring client to provide capacity does not implement API!

Q. How to grow and shrink array?

First try.

- `push()`: increase size of `s[]` by 1.
- `pop()`: decrease size of `s[]` by 1.

Too expensive.

- Need to copy all item to a new array.
- Inserting first N items takes time proportional to $1 + 2 + \ldots + N \sim N^2/2$.

infeasible for large N

Goal. Ensure that array resizing happens infrequently.

Stack: dynamic array implementation

Q. How to grow array?

"repeated doubling"

A. If array is full, create a new array of twice the size, and copy items.

```
public StackOfStrings() {  s = new String[2];  }

public void push(String item)
{
   if (N == s.length) resize(2 * s.length);
   s[N++] = item;
}

private void resize(int capacity)
{
   String[] dup = new String[capacity];
   for (int i = 0; i < N; i++)
      dup[i] = s[i];
   s = dup;
}
```

$1 + 2 + 4 + \ldots + N/2 + N \sim 2N$

Consequence. Inserting first N items takes time proportional to N (not N^2).

Stack: dynamic array implementation

Q. How to shrink array?

First try.

- `push()`: double size of `s[]` when array is full.
- `pop()`: halve size of `s[]` when array is half full.

"thrashing"

Too expensive

- Consider push-pop-push-pop-... sequence when array is full.
- Takes time proportional to N per operation.

N = 5	`it`	`was`	`the`	`best`	`of`	*null*	*null*	*null*
N = 4	`it`	`was`	`the`	`best`				
N = 5	`it`	`was`	`the`	`best`	`of`	*null*	*null*	*null*
N = 4	`it`	`was`	`the`	`best`				

Stack: dynamic array implementation

Q. How to shrink array?

Efficient solution.

- **push()**: double size of **s[]** when array is full.
- **pop()**: halve size of **s[]** when array is one-quarter full.

```
public String pop()
{
   String item = s[--N];
   s[N] = null;
   if (N > 0 && N == s.length/4) resize(s.length / 2);
   return item;
}
```

Invariant. Array is always between 25% and 100% full.

Stack: dynamic array implementation trace

StdIn	StdOut	N	a.length	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
		0	1	*null*							
to		1	1	to							
be		2	2	to	be						
or		3	4	to	be	or	*null*				
not		4	4	to	be	or	not				
to		5	8	to	be	or	not	to	*null*	*null*	*null*
-	to	4	8	to	be	or	not	*null*	*null*	*null*	*null*
be		5	8	to	be	or	not	be	*null*	*null*	*null*
-	be	4	8	to	be	or	not	*null*	*null*	*null*	*null*
-	not	3	8	to	be	or	*null*	*null*	*null*	*null*	*null*
that		4	8	to	be	or	that	*null*	*null*	*null*	*null*
-	that	3	8	to	be	or	*null*	*null*	*null*	*null*	*null*
-	or	2	4	to	be	*null*	*null*				
-	be	1	2	to	*null*						
is		2	2	to	is						

Amortized analysis

Amortized analysis. Average running time per operation over a worst-case sequence of operations.

Proposition. Starting from empty data structure, any sequence of M push and pop ops takes time proportional to M.

running time for doubling stack with N items

	worst	best	amortized
construct	1	1	1
push	N	1	1
pop	N	1	1

doubling or shrinking

Remark. Recall, WQUPC used amortized bound.

Stack implementations: memory usage

Linked list implementation. ~ 16N bytes.

```
private class Node
{
   String item;
   Node next;
}
```

8 bytes overhead for object

4 bytes

4 bytes

16 bytes per item

Doubling array. Between ~ 4N (100% full) and ~ 16N (25% full).

```
public class DoublingStackOfStrings
{
   private String[] s;
   private int N = 0;
   …
}
```

4 bytes × array size

4 bytes

Remark. Our analysis doesn't include the memory for the items themselves.

Stack implementations: dynamic array vs. linked List

Tradeoffs. Can implement with either array or linked list; client can use interchangeably. Which is better?

Linked list.

- Every operation takes constant time in worst-case.
- Uses extra time and space to deal with the links.

Array.

- Every operation takes constant amortized time.
- Less wasted space.

- stacks
- dynamic resizing
- **queues**
- generics
- iterators
- applications

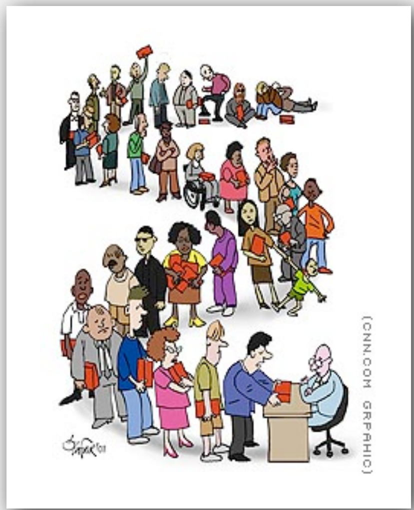
Queues

Queue operations.

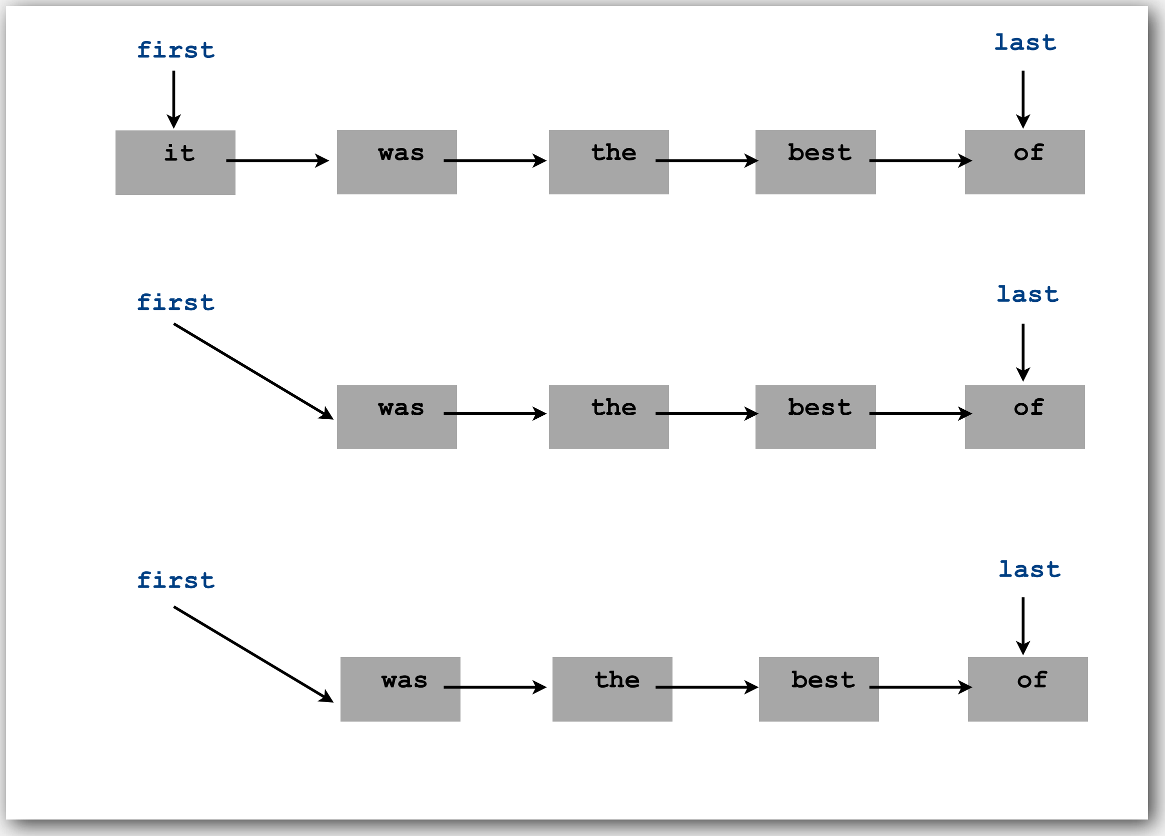
- `enqueue()` Insert a new item onto queue.
- `dequeue()` Delete and return the item least recently added.
- `isEmpty()` Is the queue empty?

```
public static void main(String[] args)
{
   QueueOfStrings q = new QueueOfStrings();
   while (!StdIn.isEmpty())
   {
      String item = StdIn.readString();
      if (item.equals("-")) StdOut.print(q.dequeue());
      else                  q.enqueue(item);
   }
}
```

```
% more tobe.txt
to be or not to - be - - that - - - is

% java QueueOfStrings < tobe.txt
to be or not to be
```



Queue dequeue: linked list implementation




```
String item = first.item;
```

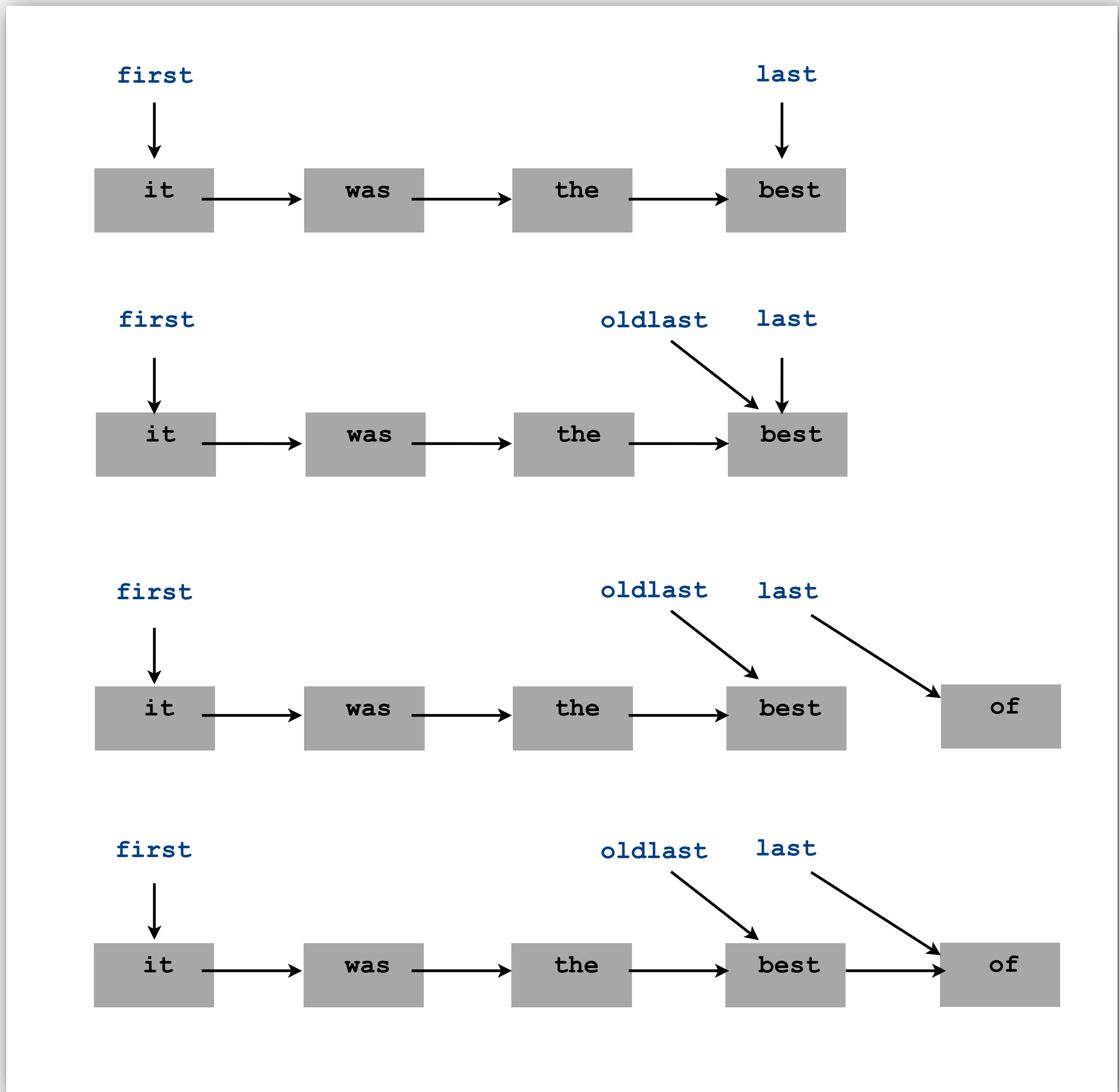
"it"

```
first = first.next;
```

```
return item;
```

"it"

Queue enqueue: linked list implementation




```
Node oldlast = last;
```

```
last = new Node();
last.item = "of";
last.next = null;
```

```
oldlast.next = last;
```

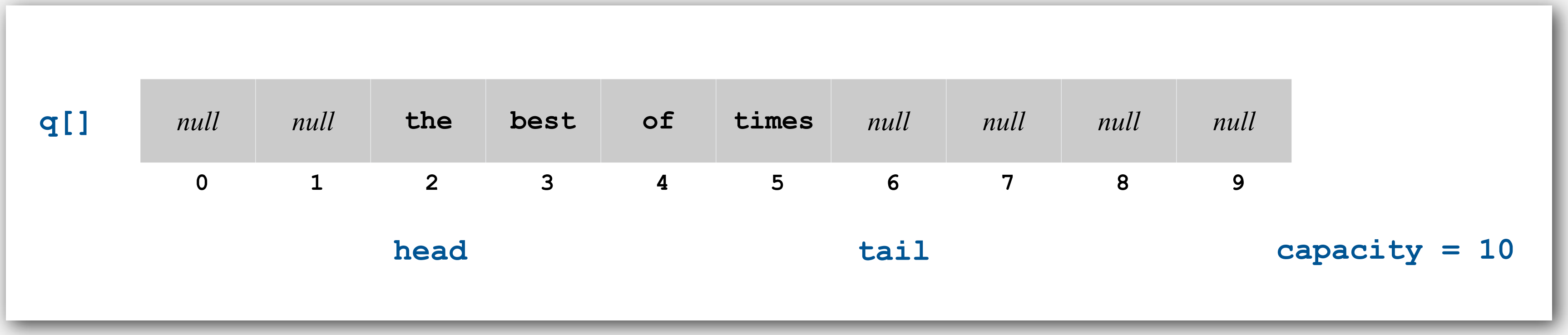
Queue: linked list implementation

```
public class QueueOfStrings
{
   private Node first, last;

   private class Node
   {  /* same as in StackOfStrings */  }

   public boolean isEmpty()
   {  return first == null;  }

   public void enqueue(String item)
   {
      Node oldlast = last;
      last = new Node();
      last.item = item;
      last.next = null;
      if (isEmpty()) first = last;
      else           oldlast.next = last;
   }

   public String dequeue()
   {
      String item = first.item;
      first       = first.next;
      if (isEmpty()) last = null;
      return item;
   }
}
```

Queue: dynamic array implementation

Array implementation of a queue.

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the `capacity`.
- Add repeated doubling and shrinking.



- stacks
- dynamic resizing
- queues
- generics
- iterators
- applications

Parameterized stack

We implemented: `StackOfStrings`.

We also want: `StackOfURLs`, `StackOfCustomers`, `StackOfInts`, etc?

Attempt 1. Implement a separate stack class for each type.

- Rewriting code is tedious and error-prone.
- Maintaining cut-and-pasted code is tedious and error-prone.

@#$*! most reasonable approach until Java 1.5.

[hence, used in Algorithms in Java, 3rd edition]

Parameterized stack

We implemented: `StackOfStrings`.

We also want: `StackOfURLs`, `StackOfCustomers`, `StackOfInts`, etc?

Attempt 2. Implement a stack with items of type `Object`.

- Casting is required in client.
- Casting is error-prone: run-time error if types mismatch.

```
StackOfObjects s = new StackOfObjects();
Apple  a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = (Apple) (s.pop());
```

run-time error

Parameterized stack

We implemented: `StackOfStrings`.

We also want: `StackOfURLs`, `StackOfCustomers`, `StackOfInts`, etc?

Attempt 3. Java generics.

- Avoid casting in both client and implementation.
- Discover type mismatch errors at compile-time instead of run-time.

type parameter

```
Stack<Apple> s = new Stack<Apple>();
Apple  a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = s.pop();
```

compile-time error

Guiding principles. Welcome compile-time errors; avoid run-time errors.

Generic stack: linked list implementation

```
public class LinkedStackOfStrings
{
   private Node first = null;

   private class Node
   {
      String item;
      Node next;
   }

   public boolean isEmpty()
   {  return first == null;  }

   public void push(String item)
   {
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public String pop()
   {
      String item = first.item;
      first = first.next;
      return item;
   }
}
```

```
public class Stack<Item>
{
   private Node first = null;

   private class Node
   {
      Item item;
      Node next;
   }

   public boolean isEmpty()
   {  return first == null;  }

   public void push(Item item)
   {
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public Item pop()
   {
      Item item = first.item;
      first = first.next;
      return item;
   }
}
```

generic type name

Generic stack: array implementation

```
public class ArrayStackOfStrings
{
   private String[] s;
   private int N = 0;

   public StackOfStrings(int capacity)
   {  s = new String[capacity];  }

   public boolean isEmpty()
   {  return N == 0;  }

   public void push(String item)
   {  s[N++] = item;  }

   public String pop()
   {  return s[--N];  }
}
```

```
public class ArrayStack<Item>
{
   private Item[] s;
   private int N = 0;

   public Stack(int capacity)
   {  s = new Item[capacity];  }

   public boolean isEmpty()
   {  return N == 0;  }

   public void push(Item item)
   {  s[N++] = item;  }

   public Item pop()
   {  return s[--N];  }
}
```

the way it should be

@#$*! generic array creation not allowed in Java

Generic stack: array implementation

```
public class ArrayStackOfStrings
{
   private String[] s;
   private int N = 0;

   public StackOfStrings(int capacity)
   {  s = new String[capacity];  }

   public boolean isEmpty()
   {  return N == 0;  }

   public void push(String item)
   {  s[N++] = item;  }

   public String pop()
   {  return s[--N];  }
}
```

```
public class ArrayStack<Item>
{
   private Item[] s;
   private int N = 0;

   public Stack(int capacity)
   {  s = (Item[]) new Object[capacity]; }

   public boolean isEmpty()
   { return N == 0; }

   public void push(Item item)
   {  s[N++] = item;  }

   public Item pop()
   {  return s[--N];  }
}
```

the way it is

the ugly cast

Generic data types: autoboxing

Q. What to do about primitive types?

Wrapper type.

- Each primitive type has a wrapper object type.
- Ex: `Integer` is wrapper type for `int`.

Autoboxing. Automatic cast between a primitive type and its wrapper.

Syntactic sugar. Behind-the-scenes casting.

```
Stack<Integer> s = new Stack<Integer>();
s.push(17);        // s.push(new Integer(17));
int a = s.pop();   // int a = s.pop().intValue();
```

Bottom line. Client code can use generic stack for any type of data.

Autoboxing challenge

Q. What does the following program print?

```
public class Autoboxing {

    public static void cmp(Integer a, Integer b) {
        if      (a <  b) StdOut.printf("%d <  %d\n", a, b);
        else if (a == b) StdOut.printf("%d == %d\n", a, b);
        else             StdOut.printf("%d >  %d\n", a, b);
    }

    public static void main(String[] args) {
        cmp(new Integer(42), new Integer(42));
        cmp(43, 43);
        cmp(142, 142);
    }
}
```

```
% java Autoboxing
42 >  42
43 == 43
142 >  142
```

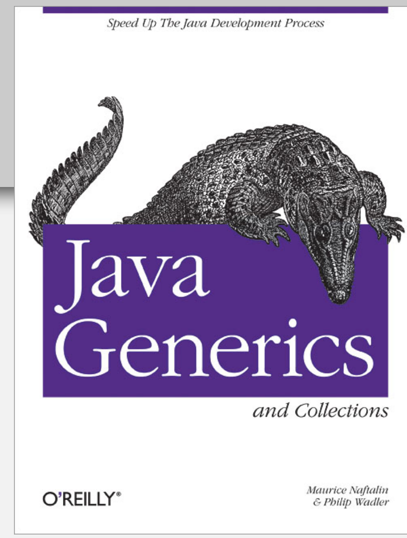
Best practice. Avoid using wrapper types whenever possible.

Generics

Caveat. Java generics can be mystifying at times.

```
public class Collections
{
   ...
   public static<T> void copy(List<? super T> dest, List<? extends T> src)
   {
      for (int i = 0; i < src.size(); i++)
         dest.set(i, src.get(i));
   }
}
```

mixing generics with inheritance

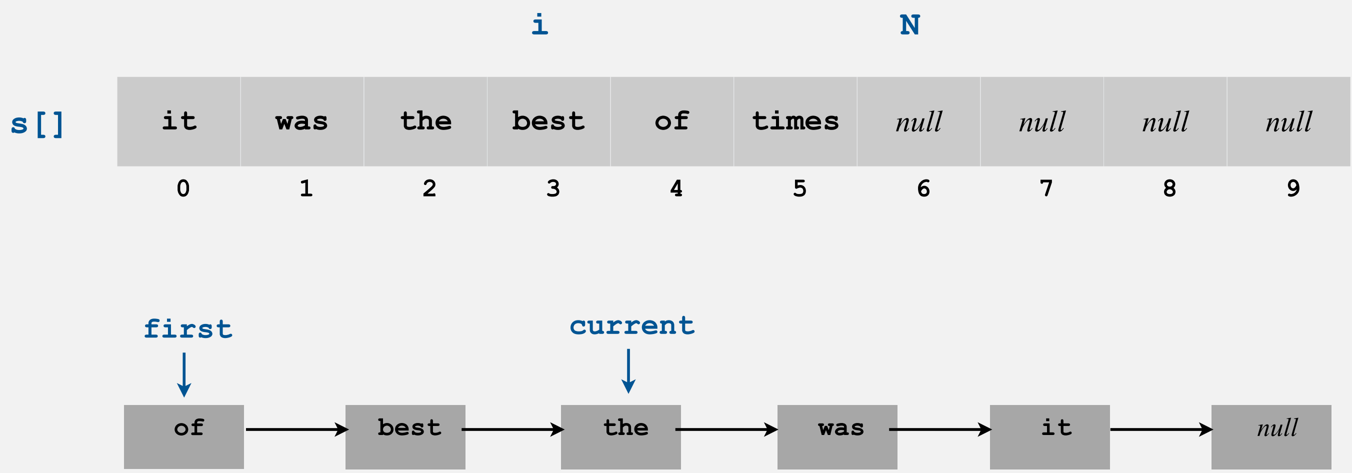


This course. Restrict attention to "pure generics."

avoid mixing generics with inheritance

Iteration

Design challenge. Support iteration over stack items by client, without revealing the internal representation of the stack.



Java solution. Make stack implement the `Iterable` interface.

Iterators

Q. What is an `Iterable` ?

A. Has a method that returns an `Iterator`.

```
public interface Iterable<Item>
{
   Iterator<Item> iterator();
}
```

Q. What is an `Iterator` ?

A. Has methods `hasNext()` and `next()`.

```
public interface Iterator<Item>
{
   boolean hasNext();
   Item next();
   void remove();   ← optional; use at your own risk
}
```

Q. Why make data structures `Iterable` ?

A. Java supports elegant client code.

"foreach" statement

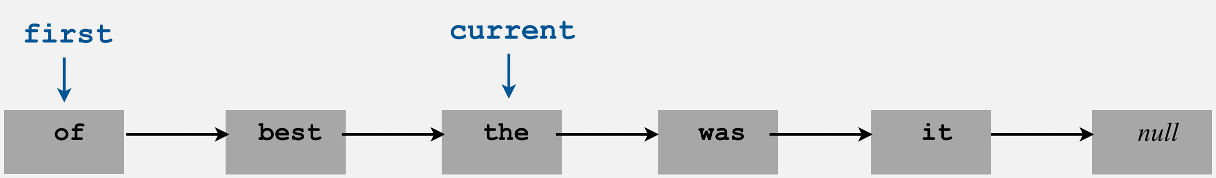
```
for (String s : stack)
   StdOut.println(s);
```

equivalent code

```
Iterator<String> i = stack.iterator();
while (i.hasNext())
{
   String s = i.next();
   StdOut.println(s);
}
```

Stack iterator: linked list implementation

```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    ...

    public Iterator<Item> iterator() { return new ListIterator(); }

    private class ListIterator implements Iterator<Item>
    {
        private Node current = first;

        public boolean hasNext() {  return current != null;  }
        public void remove()     {  /* not supported */      }
        public Item next()
        {
            Item item = current.item;
            current   = current.next;
            return item;
        }
    }
}
```



Stack iterator: array implementation

```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    ...

    public Iterator<Item> iterator() { return new ArrayIterator(); }

    private class ArrayIterator implements Iterator<Item>
    {
        private int i = N;

        public boolean hasNext() {  return i > 0;          }
        public void remove()     {  /* not supported */   }
        public Item next()       {  return s[--i];        }
    }
}
```

				i			N			
s[]	it	was	the	best	of	times	*null*	*null*	*null*	*null*
	0	1	2	3	4	5	6	7	8	9

- stacks
- dynamic resizing
- queues
- generics
- iterators
- **applications**

Java collections library

java.util.List API.

- **boolean isEmpty()** Is the list empty?
- **int size()** Return number of items on the list.
- **void add(Item item)** Insert a new item to end of list.
- **void add(int index, Item item)** Insert item at specified index.
- **Item get(int index)** Return item at given index.
- **Item remove(int index)** Return and delete item at given index.
- **Item set(int index Item item)** Replace element at given index.
- **boolean contains(Item item)** Does the list contain the item?
- **Iterator<Item> iterator()** Return iterator.
- ...

Implementations.

- **java.util.ArrayList** implements API using an array.
- **java.util.LinkedList** implements API using a (doubly) linked list.

`java.util.Stack`.

- Supports `push()`, `pop()`, `size()`, `isEmpty()`, and iteration.
- Also implements `java.util.List` interface from previous slide, e.g., `set()`, `get()`, and `contains()`.
- Bloated and poorly-designed API $\Rightarrow$ don't use.

`java.util.Queue`.

- An interface, not an implementation of a queue.

Best practices. Use our implementations of `Stack` and `Queue` if you need a stack or a queue.

War story (from COS 226)

Generate random open sites in an N-by-N percolation system.

- Jenny: pick (i, j) at random; if closed, repeat.
  Takes ~ $c_1 N^2$ seconds.
- Kenny: maintain a `java.util.ArrayList` of open sites.
  Pick an index at random and delete.
  Takes ~ $c_1 N^4$ seconds.

Q. Why is Kenny's code so slow?

Lesson. Don't use a library until you understand its API!

COS 226. Can't use a library until we've implemented it in class.

Real world applications.

- Parsing in a compiler.
- Java virtual machine.
- Undo in a word processor.
- Back button in a Web browser.
- PostScript language for printers.
- Implementing function calls in a compiler.

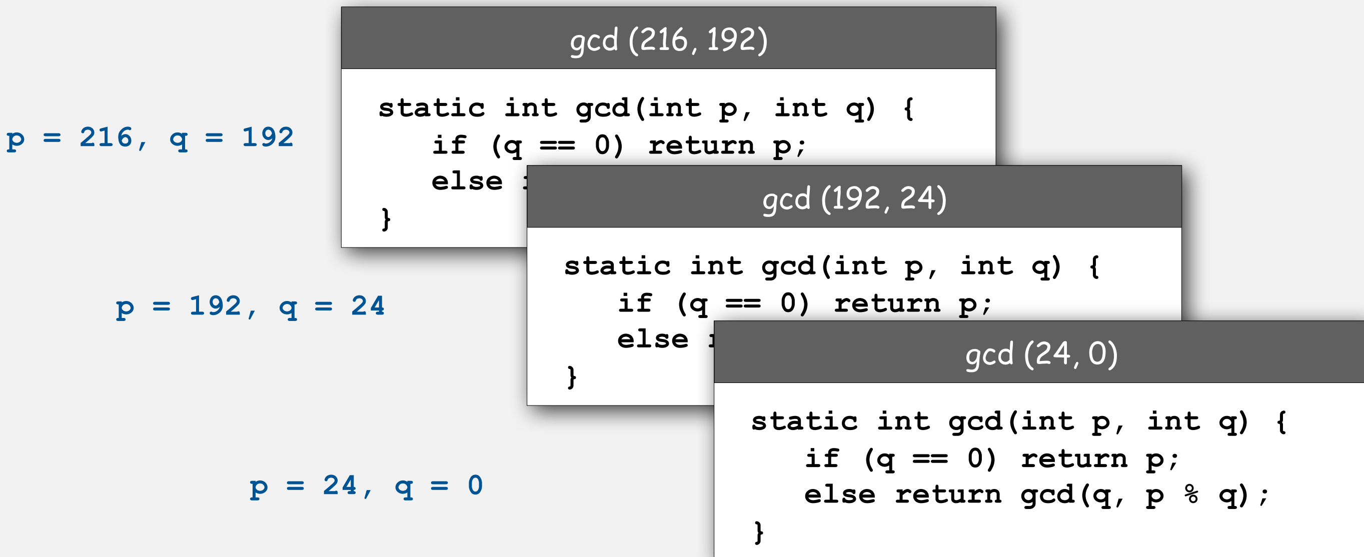
Function calls

How a compiler implements a function.

- Function call: push local environment and return address.
- Return: pop return address and local environment.

Recursive function. Function that calls itself.

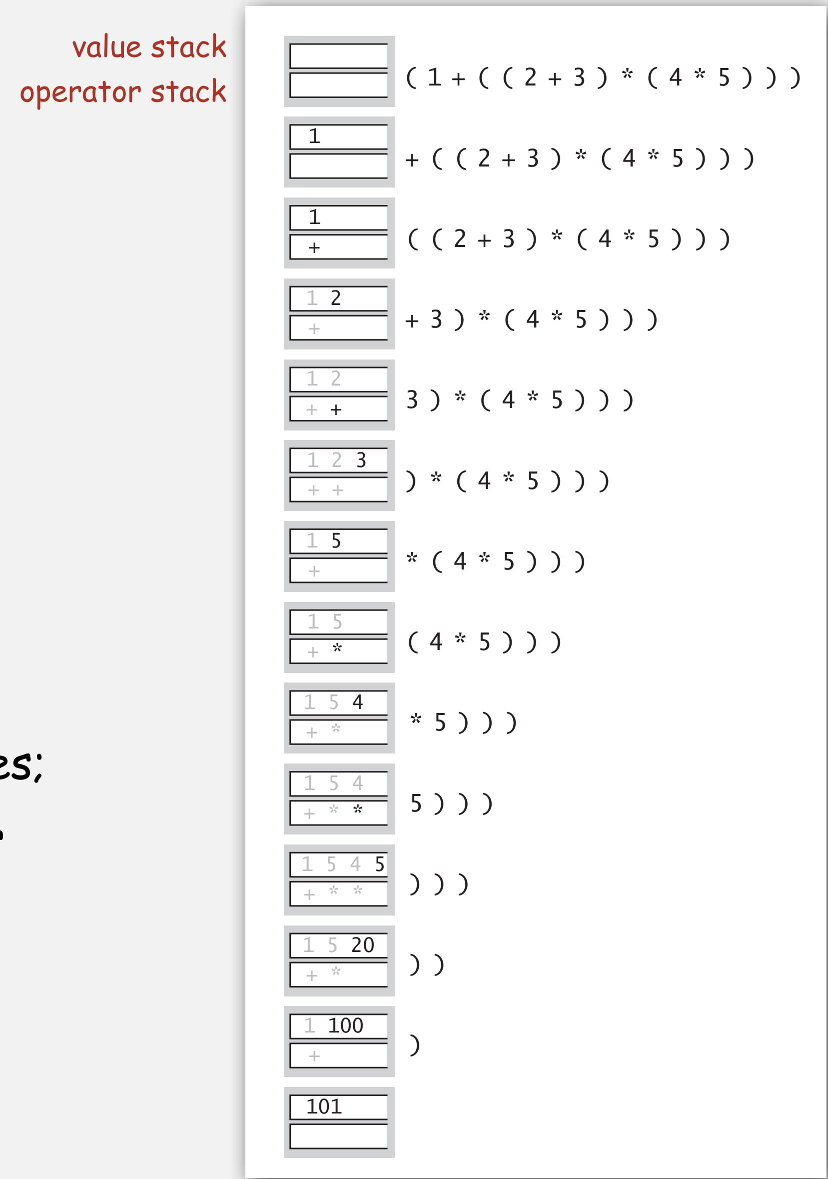
Note. Can always use an explicit stack to remove recursion.



Goal. Evaluate infix expressions.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

operand   operator



Two-stack algorithm. [E. W. Dijkstra]

- Value: push onto the value stack.
- Operator: push onto the operator stack.
- Left parens: ignore.
- Right parens: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

Context. An interpreter!

Arithmetic expression evaluation

```
public class Evaluate
{
   public static void main(String[] args)
   {
      Stack<String> ops  = new Stack<String>();
      Stack<Double> vals = new Stack<Double>();
      while (!StdIn.isEmpty()) {
         String s = StdIn.readString();
         if      (s.equals("("))               ;
         else if (s.equals("+"))    ops.push(s);
         else if (s.equals("*"))    ops.push(s);
         else if (s.equals(")"))
         {
            String op = ops.pop();
            if      (op.equals("+")) vals.push(vals.pop() + vals.pop());
            else if (op.equals("*")) vals.push(vals.pop() * vals.pop());
         }
         else vals.push(Double.parseDouble(s));
      }
      StdOut.println(vals.pop());
   }
}
```

```
% java Evaluate
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0
```

Correctness

Q. Why correct?

A. When algorithm encounters an operator surrounded by two values within parentheses, it leaves the result on the value stack.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

as if the original input were:

```
( 1 + ( 5 * ( 4 * 5 ) ) )
```

Repeating the argument:

```
( 1 + ( 5 * 20 ) )
( 1 + 100 )
101
```

Extensions. More ops, precedence order, associativity.

Stack-based programming languages

Observation 1. The 2-stack algorithm computes the same value if the operator occurs after the two values.

```
( 1 ( ( 2 3 + ) ( 4 5 * ) * ) + )
```

Observation 2. All of the parentheses are redundant!

```
1 2 3 + 4 5 * * +
```

Jan Lukasiewicz

Bottom line. Postfix or "reverse Polish" notation.

Applications. Postscript, Forth, calculators, Java virtual machine, …

PostScript

Page description language.

- Explicit stack.
- Full computational model
- Graphics engine.

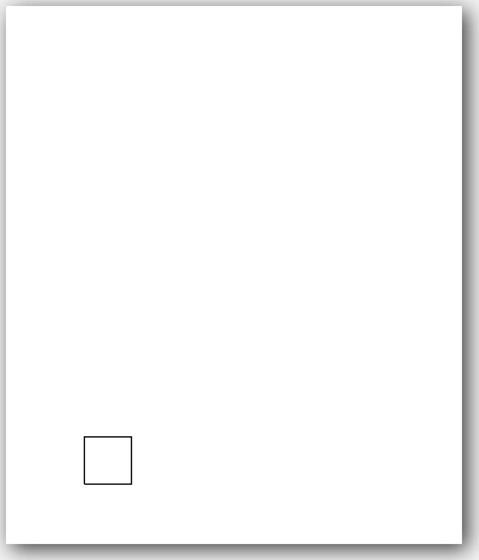
Basics.

- `%!`: "I am a PostScript program."
- Literal: "push me on the stack."
- Function calls take arguments from stack.
- Turtle graphics built in.

a PostScript program

```
%!
72 72 moveto
0 72 rlineto
72 0 rlineto
0 -72 rlineto
-72 0 rlineto
2 setlinewidth
stroke
```

its output



PostScript

Data types.

- Basic: integer, floating point, boolean, ...
- Graphics: font, path, curve, ...
- Full set of built-in operators.

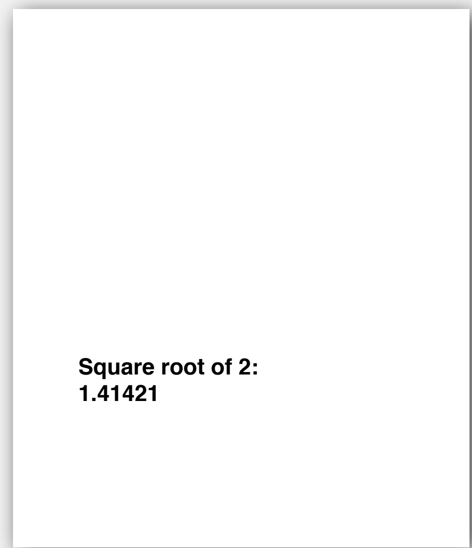
Text and strings.

`System.out.print()`

- Full font support.
- **`show`** (display a string, using current font).
- **`cvs`** (convert anything to a string).

`toString()`

```
%!
/Helvetica-Bold findfont 16 scalefont setfont
72 168 moveto
(Square root of 2:) show
72 144 moveto
2 sqrt 10 string cvs show
```



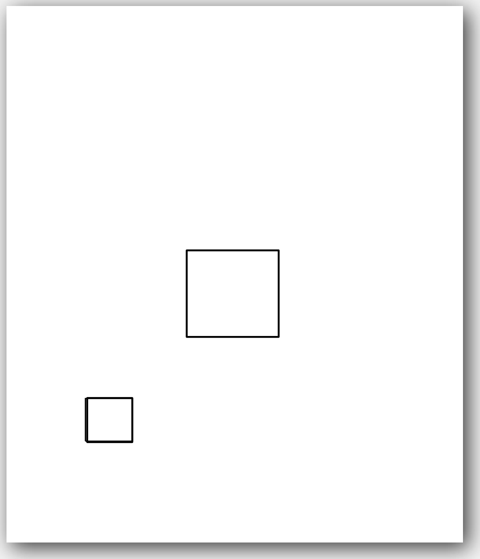
Variables (and functions).

- Identifiers start with `/`.
- `def` operator associates id with value.
- Braces.
- args on stack.

function definition

function calls

```
%!
/box
{
  /sz exch def
  0 sz rlineto
  sz 0 rlineto
  0 sz neg rlineto
  sz neg 0 rlineto
} def

72 144 moveto
72 box
288 288 moveto
144 box
2 setlinewidth
stroke
```



For loop.

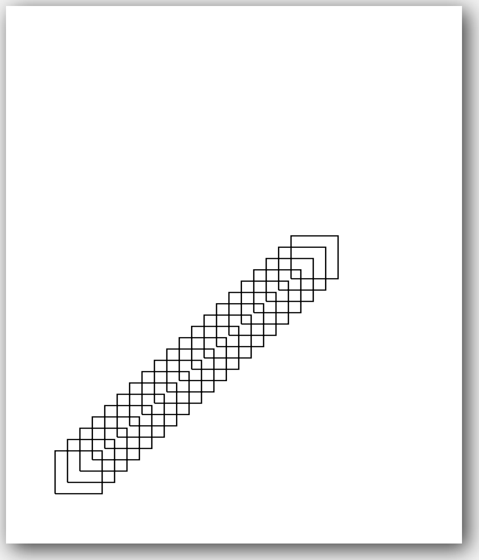
- "from, increment, to" on stack.
- Loop body in braces.
- `for` operator.

```
%!
\box
{
   ...
}

1 1 20
{ 19 mul dup 2 add moveto 72 box }
for
stroke
```

If-else conditional.

- Boolean on stack.
- Alternatives in braces.
- `if` operator.

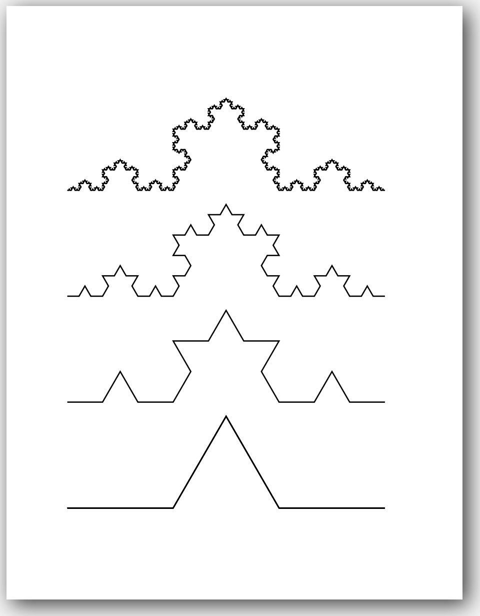
... (hundreds of operators)



PostScript

Application 1. All figures in Algorithms in Java, 3^{rd} edition: figures created directly in PostScript.

```
%!
72 72 translate

/kochR
  {
    2 copy ge { dup 0 rlineto }
      {
        3 div
        2 copy kochR 60 rotate
        2 copy kochR -120 rotate
        2 copy kochR 60 rotate
        2 copy kochR
      } ifelse
    pop pop
  } def

0   0 moveto   81 243 kochR
0  81 moveto   27 243 kochR
0 162 moveto    9 243 kochR
0 243 moveto    1 243 kochR
stroke
```



See page 218

Application 2. All figures in Algorithms, 4^{th} edition: enhanced version of `StdDraw` saves to PostScript for vector graphics.

Queue applications

Familiar applications.

- iTunes playlist.
- Data buffers (iPod, TiVo).
- Asynchronous data transfer (file IO, pipes, sockets).
- Dispensing requests on a shared resource (printer, processor).

Simulations of the real world.

- Traffic analysis.
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.

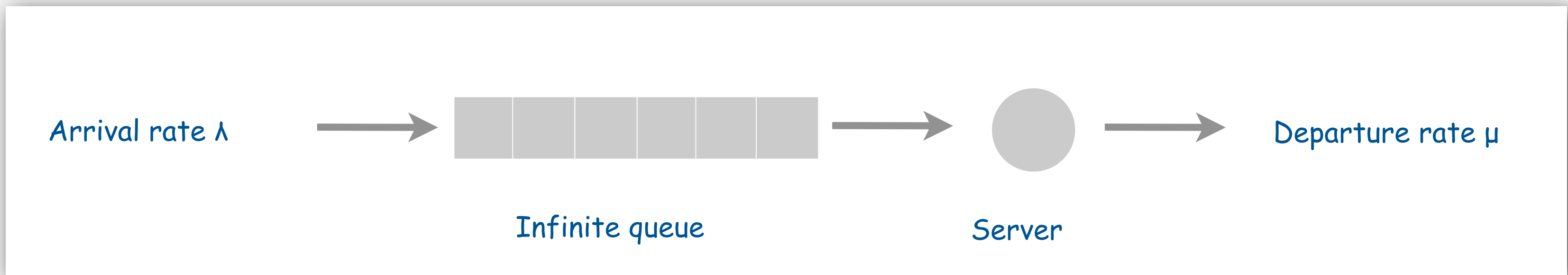
M/M/1 queuing model

M/M/1 queue.

- Customers arrive according to Poisson process at rate of λ per minute.
- Customers are serviced with rate of μ per minute.

interarrival time has exponential distribution $\Pr[X \le x] = 1 - e^{-\lambda x}$

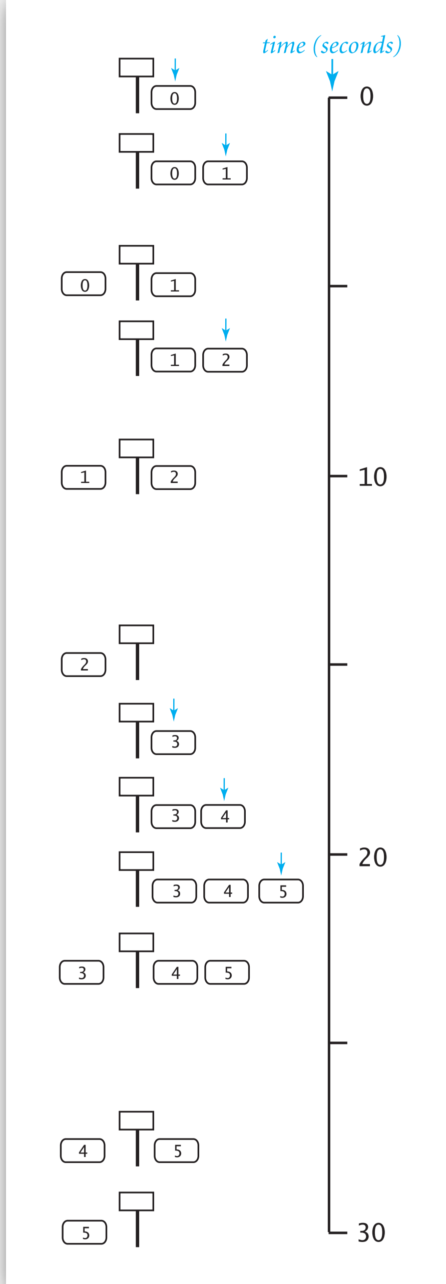
service time has exponential distribution $\Pr[X \le x] = 1 - e^{-\mu x}$



Q. What is average wait time W of a customer in system?

Q. What is average number of customers L in system?

M/M/1 queuing model: example simulation



	arrival	departure	wait
0	0	5	5
1	2	10	8
2	7	15	8
3	17	23	6
4	19	28	9
5	21	30	9

M/M/1 queuing model: event-based simulation

```
public class MM1Queue
{
    public static void main(String[] args) {
        double lambda = Double.parseDouble(args[0]);   // arrival rate
        double mu     = Double.parseDouble(args[1]);   // service rate
        double nextArrival = StdRandom.exp(lambda);
        double nextService = nextArrival + StdRandom.exp(mu);

        Queue<Double> queue = new Queue<Double>();
        Histogram hist = new Histogram("M/M/1 Queue", 60);

        while (true)
        {
            while (nextArrival < nextService)
            {
                queue.enqueue(nextArrival);
                nextArrival += StdRandom.exp(lambda);
            }

            double arrival = queue.dequeue();
            double wait = nextService - arrival;
            hist.addDataPoint(Math.min(60,  (int) (Math.round(wait))));
            if (queue.isEmpty()) nextService = nextArrival + StdRandom.exp(mu);
            else                 nextService = nextService + StdRandom.exp(mu);
        }
    }
}
```

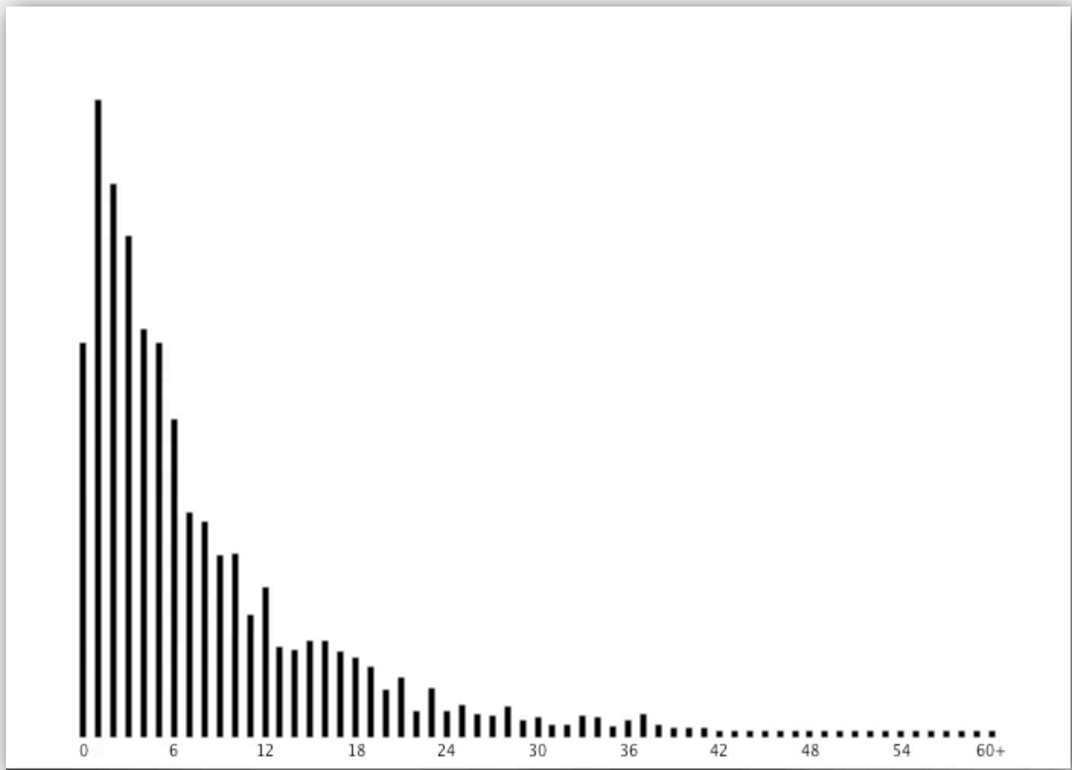
next event is an arrival

next event is a service completion

M/M/1 queuing model: experiments

Observation. If service rate μ is much larger than arrival rate λ, customers gets good service.

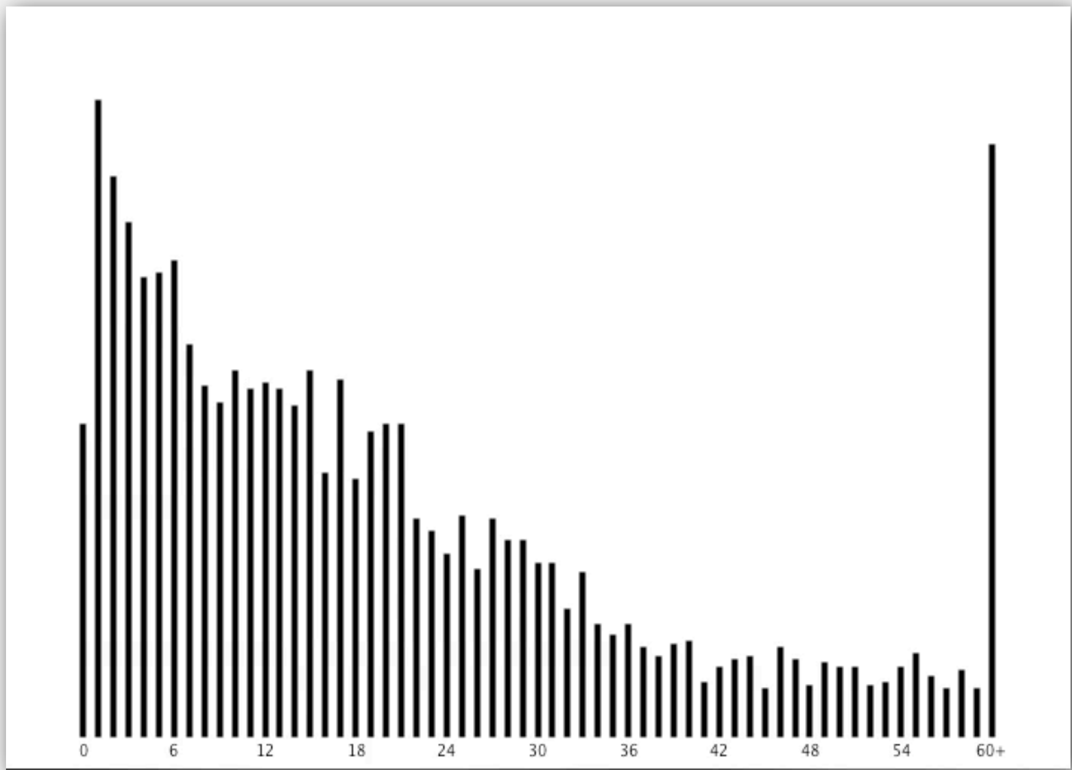
```
% java MM1Queue .2 .333
```



M/M/1 queuing model: experiments

Observation. As service rate μ approaches arrival rate λ, services goes to h***.

```
% java MM1Queue .2 .25
```



M/M/1 queuing model: experiments

Observation. As service rate μ approaches arrival rate λ, services goes to h***.

```
% java MM1Queue .2 .21
```

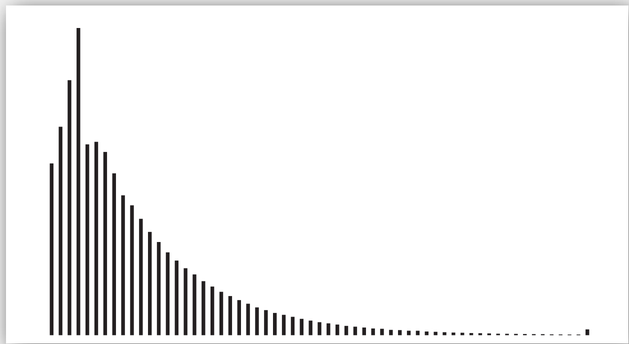
M/M/1 queuing model: analysis

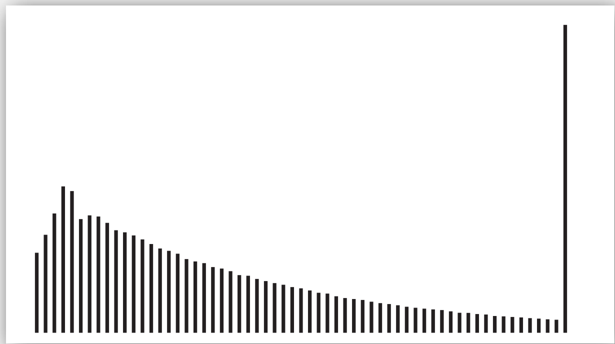
M/M/1 queue. Exact formulas known.

wait time W and queue length L approach infinity
as service rate approaches arrival rate

Little's Law

$$W = \frac{1}{\mu - \lambda}, \quad L = \lambda W$$

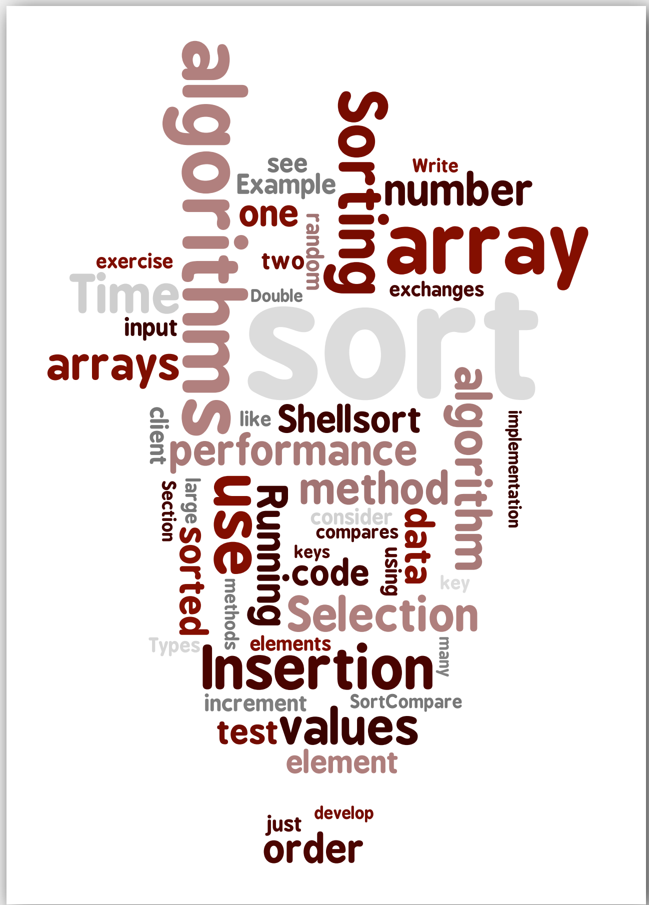




More complicated queueing models. Event-based simulation essential!

Queueing theory. See ORF 309.

2.1 Elementary Sorts



- rules of the game
- selection sort
- insertion sort
- sorting challenges
- shellsort

Sorting problem

Ex. Student record in a University.

file	Fox	1	A	243-456-9091	101 Brown
	Quilici	1	C	343-987-5642	32 McCosh
	Chen	2	A	884-232-5341	11 Dickinson
	Furia	3	A	766-093-9873	22 Brown
	Kanaga	3	B	898-122-9643	343 Forbes
record	Andrews	3	A	874-088-1212	121 Whitman
	Rohde	3	A	232-343-5555	115 Holder
	Battle	4	C	991-878-4944	308 Blair
key	Aaron	4	A	664-480-0023	097 Little
	Gazsi	4	B	665-303-0266	113 Walker

Sort. Rearrange array of N objects into ascending order.

Aaron	4	A	664-480-0023	097 Little
Andrews	3	A	874-088-1212	121 Whitman
Battle	4	C	991-878-4944	308 Blair
Chen	2	A	884-232-5341	11 Dickinson
Fox	1	A	243-456-9091	101 Brown
Furia	3	A	766-093-9873	22 Brown
Gazsi	4	B	665-303-0266	113 Walker
Kanaga	3	B	898-122-9643	343 Forbes
Rohde	3	A	232-343-5555	115 Holder
Quilici	1	C	343-987-5642	32 McCosh

Sample sort client

Goal. Sort any type of data.

Ex 1. Sort random numbers in ascending order.

```
public class Experiment
{
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      Double[] a = new Double[N];
      for (int i = 0; i < N; i++)
         a[i] = StdRandom.uniform();
      Insertion.sort(a);
      for (int i = 0; i < N; i++)
         StdOut.println(a[i]);
   }
}
```

```
% java Experiment 10
0.08614716385210452
0.09054270895414829
0.10708746304898642
0.21166190071646818
0.363292849257276
0.460954145685913
0.5340026311350087
0.7216129793703496
0.9003500354411443
0.9293994908845686
```

Sample sort client

Goal. Sort any type of data.

Ex 2. Sort strings from standard input in alphabetical order.

```
public class StringSorter
{
   public static void main(String[] args)
   {
      String[] a = StdIn.readAll().split("\\s+");
      Insertion.sort(a);
      for (int i = 0; i < a.length; i++)
         StdOut.println(a[i]);
   }
}
```

```
% more words3.txt
bed bug dad yet zoo ... all bad yes

% java StringSorter < words.txt
all bad bed bug dad ... yes yet zoo
```

Sample sort client

Goal. Sort any type of data.

Ex 3. Sort the files in a given directory by filename.

```
import java.io.File;
public class FileSorter
{
   public static void main(String[] args)
   {
      File directory = new File(args[0]);
      File[] files = directory.listFiles();
      Insertion.sort(files);
      for (int i = 0; i < files.length; i++)
         StdOut.println(files[i].getName());
   }
}
```

```
% java FileSorter .
Insertion.class
Insertion.java
InsertionX.class
InsertionX.java
Selection.class
Selection.java
Shell.class
Shell.java
ShellX.class
ShellX.java
```

Goal. Sort any type of data.

Q. How can sort know to compare data of type `String`, `Double`, and `File` without any information about the type of an item?

Callbacks.

- Client passes array of objects to sorting routine.
- Sorting routine calls back object's compare function as needed.

Implementing callbacks.

- Java: interfaces.
- C: function pointers.
- C++: class-type functors.
- ML: first-class functions and functors.

Callbacks: roadmap

client

```
import java.io.File;
public class FileSorter
{
   public static void main(String[] args)
   {
      File directory = new File(args[0]);
      File[] files = directory.listFiles();
      Insertion.sort(files);
      for (int i = 0; i < files.length; i++)
         StdOut.println(files[i].getName());
   }
}
```

object implementation

```
public class File
implements Comparable<File>
{
   ...
   public int compareTo(File b)
   {
      ...
      return -1;
      ...
      return +1;
      ...
      return 0;
   }
}
```

built in to Java

interface

```
public interface Comparable<Item>
{
   public int compareTo(Item that);
}
```

sort implementation

```
public static void sort(Comparable[] a)
{
   int N = a.length;
   for (int i = 0; i < N; i++)
      for (int j = i; j > 0; j--)
         if (a[j].compareTo(a[j-1]) < 0)
              exch(a, j, j-1);
         else break;
}
```

key point: no reference to `File` →

Comparable interface API

Comparable interface. Implement `compareTo()` so that `v.compareTo(w)`:

- Returns a negative integer if `v` is less than `w`.
- Returns a positive integer if `v` is greater than `w`.
- Returns zero if `v` is equal to `w`.
- Throw an exception if incompatible types or either is `null`.

```
public interface Comparable<Item>
{  public int compareTo(Item that);  }
```

Required properties. Must ensure a total order.

- Reflexive: $(v = v)$.
- Antisymmetric: if $(v < w)$ then $(w > v)$; if $(v = w)$ then $(w = v)$.
- Transitive: if $(v \leq w)$ and $(w \leq x)$ then $(v \leq x)$.

Built-in comparable types. `String`, `Double`, `Integer`, `Date`, `File`, ...

User-defined comparable types. Implement the `Comparable` interface.

Implementing the Comparable interface: example 1

Date data type. Simplified version of `java.util.Date`.

```
public class Date implements Comparable<Date>
{
   private final int month, day, year;

   public Date(int m, int d, int y)
   {
      month = m;
      day   = d;
      year  = y;
   }

   public int compareTo(Date that)
   {
      if (this.year  < that.year ) return -1;
      if (this.year  > that.year ) return +1;
      if (this.month < that.month) return -1;
      if (this.month > that.month) return +1;
      if (this.day   < that.day  ) return -1;
      if (this.day   > that.day  ) return +1;
      return 0;
   }
}
```

only compare dates
to other dates

Implementing the Comparable interface: example 2

Domain names.

- Subdomain: `bolle.cs.princeton.edu`.
- Reverse subdomain: `edu.princeton.cs.bolle`.
- Sort by reverse subdomain to group by category.

```
public class Domain implements Comparable<Domain>
{
   private final String[] fields;
   private final int N;

   public Domain(String name)
   {
       fields = name.split("\\.");
       N = fields.length;
   }

   public int compareTo(Domain that)
   {
      for (int i = 0; i < Math.min(this.N, that.N); i++)
      {
         String s = fields[this.N - i - 1];
         String t = fields[that.N - i - 1];
         int cmp = s.compareTo(t);
         if      (cmp < 0) return -1;
         else if (cmp > 0) return +1;
      }
      return this.N - that.N;
   }
}
```

only use this trick when no danger of overflow

subdomains

```
ee.princeton.edu
cs.princeton.edu
princeton.edu
cnn.com
google.com
apple.com
www.cs.princeton.edu
bolle.cs.princeton.edu
```

reverse-sorted subdomains

```
com.apple
com.cnn
com.google
edu.princeton
edu.princeton.cs
edu.princeton.cs.bolle
edu.princeton.cs.www
edu.princeton.ee
```

Two useful sorting abstractions

Helper functions. Refer to data through compares and exchanges.

Less. Is object `v` less than `w` ?

```
private static boolean less(Comparable v, Comparable w)
{  return v.compareTo(w) < 0;  }
```

Exchange. Swap object in array `a[]` at index `i` with the one at index `j`.

```
private static void exch(Comparable[] a, int i, int j)
{
   Comparable t = a[i];
   a[i] = a[j];
   a[j] = t;
}
```

Testing

Q. How to test if an array is sorted?

```
private static boolean isSorted(Comparable[] a)
{
   for (int i = 1; i < a.length; i++)
      if (less(a[i], a[i-1])) return false;
   return true;
}
```

Q. If the sorting algorithm passes the test, did it correctly sort its input?

A. Yes, if data accessed only through `exch()` and `less()`.

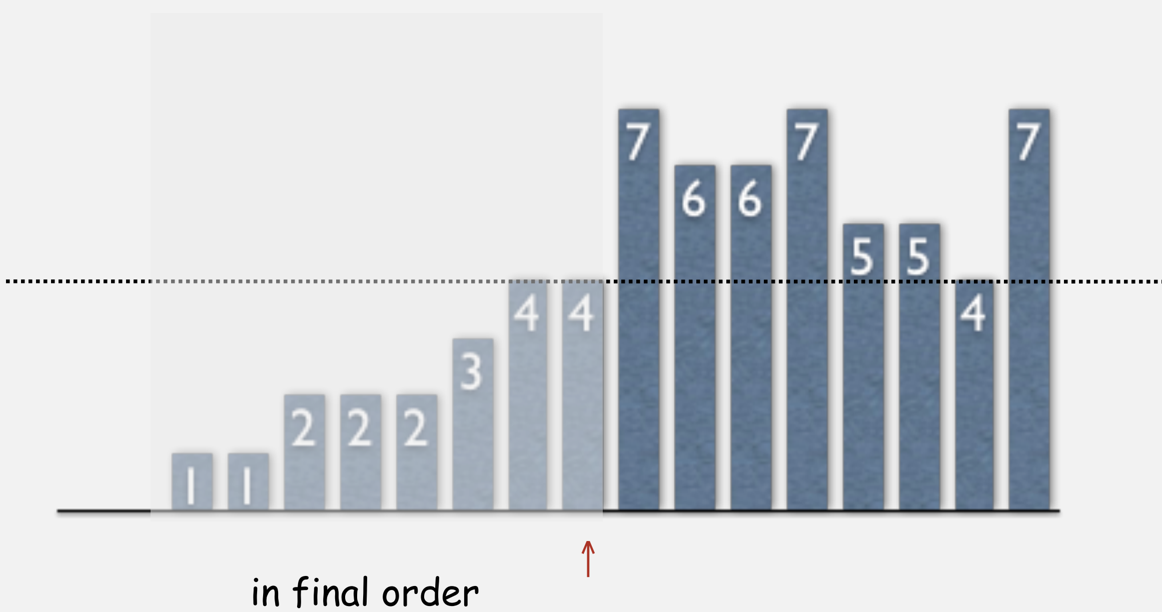
- rules of the game
- **selection sort**
- insertion sort
- sorting challenges
- shellsort

Selection sort

Algorithm. ↑ scans from left to right.

Invariants.

- Elements to the left of ↑ (including ↑) fixed and in ascending order.
- No element to right of ↑ is smaller than any element to its left.



Selection sort inner loop

To maintain algorithm invariants:

- Move the pointer to the right.

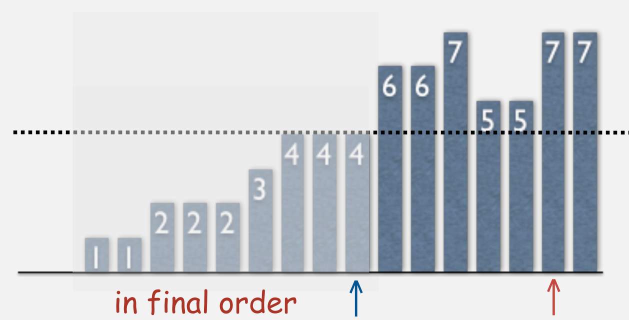
```
i++;
```

- Identify index of minimum item on right.

```
int min = i;
for (int j = i+1; j < N; j++)
   if (less(a[j], a[min]))
      min = j;
```

- Exchange into position.

```
exch(a, i, min);
```



Selection sort: Java implementation

```
public class Selection {

   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
      {
         int min = i;
         for (int j = i+1; j < N; j++)
            if (less(a[j], a[min]))
               min = j;
         exch(a, i, min);
      }
   }

   private static boolean less(Comparable v, Comparable w)
   {  /* as before */   }

   private static void exch(Comparable[] a, int i, int j)
   {  /* as before */   }
}
```

Selection sort: mathematical analysis

Proposition A. Selection sort uses $(N-1) + (N-2) + \ldots + 1 + 0 \sim N^2/2$ compares and N exchanges.

i	min	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

entries in black are examined to find the minimum

entries in red are a[min]

entries in gray are in final position

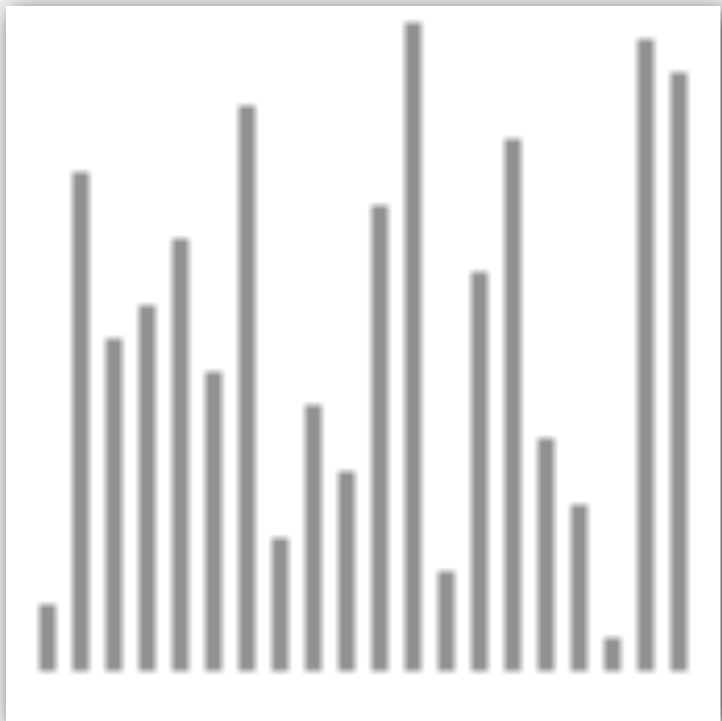
Trace of selection sort (array contents just after each exchange)

Running time insensitive to input. Quadratic time, even if array is presorted.

Data movement is minimal. Linear number of exchanges.

Selection sort animations

20 random elements



algorithm position

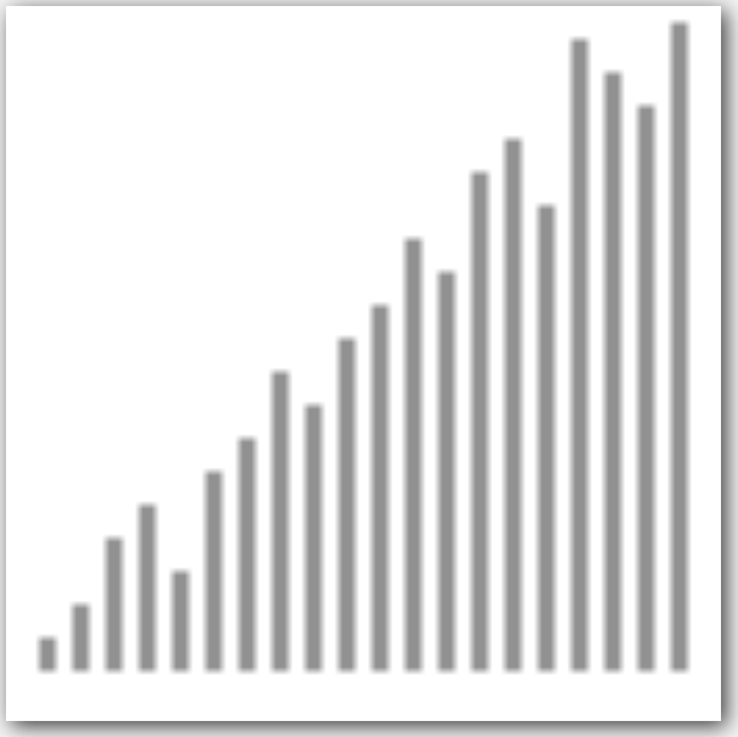
in final order

not in final order

http://www.sorting-algorithms.com/selection-sort

Selection sort animations

20 partially-sorted elements



▲ algorithm position

in final order

not in final order

http://www.sorting-algorithms.com/selection-sort

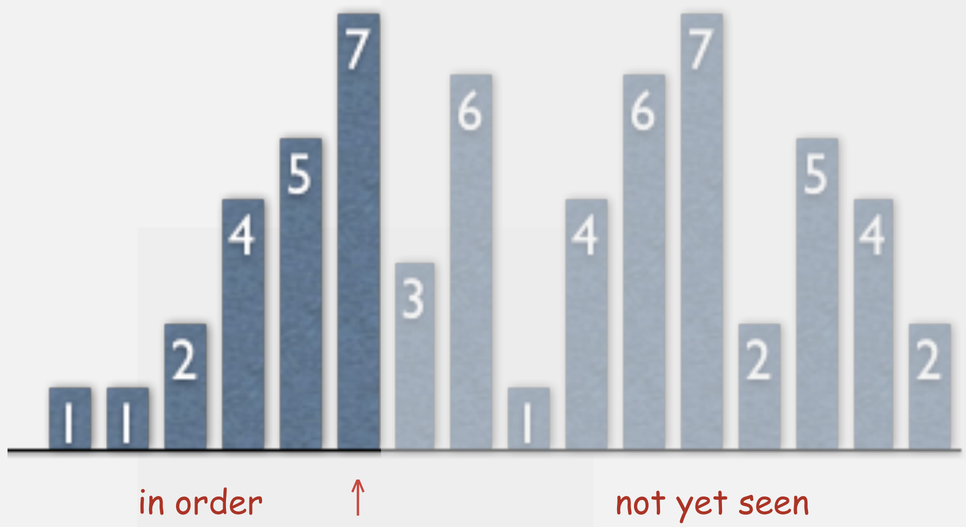
- rules of the game
- selection sort
- **insertion sort**
- sorting challenges
- shellsort

Insertion sort

Algorithm. ↑ scans from left to right.

Invariants.

- Elements to the left of ↑ (including ↑) are in ascending order.
- Elements to the right of ↑ have not yet been seen.



Insertion sort inner loop

To maintain algorithm invariants:

- Move the pointer to the right.

```
i++;
```

- Moving from right to left, exchange `a[i]` with each larger element to its left.

```
for (int j = i; j > 0; j--)
   if (less(a[j], a[j-1]))
        exch(a, j, j-1);
   else break;
```

Insertion sort: Java implementation

```
public class Insertion {

   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
         for (int j = i; j > 0; j--)
            if (less(a[j], a[j-1]))
               exch(a, j, j-1);
            else break;
   }

   private static boolean less(Comparable v, Comparable w)
   {  /* as before */  }

   private static void exch(Comparable[] a, int i, int j)
   {  /* as before */  }
}
```

Insertion sort: mathematical analysis

Proposition B. To sort a randomly-ordered array with distinct keys, insertion sort uses ~ $N^2/4$ compares and $N^2/4$ exchanges on average.

Pf. For randomly-ordered data, we expect each element to move halfway back.

a[]

i	j	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
1	0	O	S	R	T	E	X	A	M	P	L	E
2	1	O	R	S	T	E	X	A	M	P	L	E
3	3	O	R	S	T	E	X	A	M	P	L	E
4	0	E	O	R	S	T	X	A	M	P	L	E
5	5	E	O	R	S	T	X	A	M	P	L	E
6	0	A	E	O	R	S	T	X	M	P	L	E
7	2	A	E	M	O	R	S	T	X	P	L	E
8	4	A	E	M	O	P	R	S	T	X	L	E
9	2	A	E	L	M	O	P	R	S	T	X	E
10	2	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

Trace of insertion sort (array contents just after each insertion)

entries in gray do not move

entry in red is `a[j]`

entries in black moved one position right for insertion

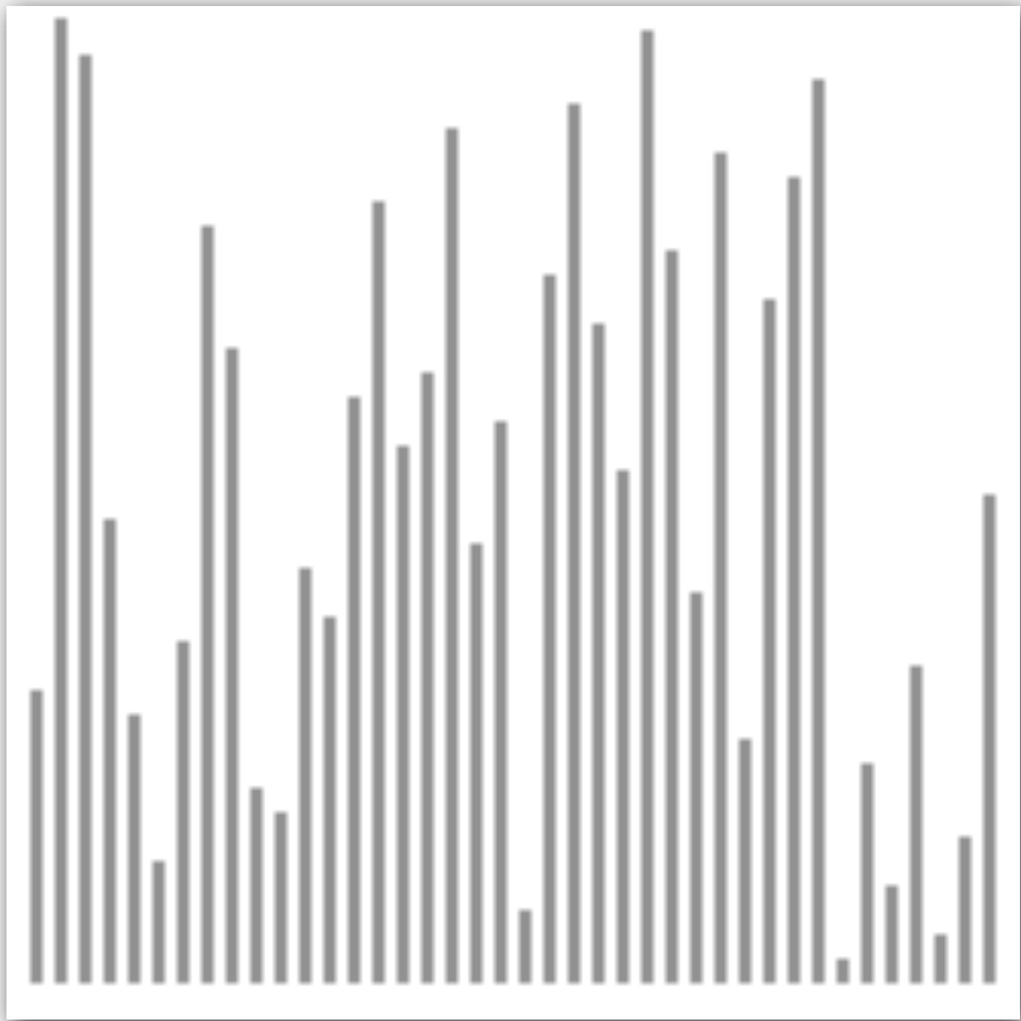
Insertion sort: trace

a[ ]

i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
		A	S	O	M	E	W	H	A	T	L	O	N	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
0	0	A	S	O	M	E	W	H	A	T	L	O	N	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
1	1	A	S	O	M	E	W	H	A	T	L	O	N	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
2	1	A	O	S	M	E	W	H	A	T	L	O	N	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
3	1	A	M	O	S	E	W	H	A	T	L	O	N	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
4	1	A	E	M	O	S	W	H	A	T	L	O	N	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
5	5	A	E	M	O	S	W	H	A	T	L	O	N	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
6	2	A	E	H	M	O	S	W	A	T	L	O	N	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
7	1	A	A	E	H	M	O	S	W	T	L	O	N	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
8	7	A	A	E	H	M	O	S	T	W	L	O	N	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
9	4	A	A	E	H	L	M	O	S	T	W	O	N	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
10	7	A	A	E	H	L	M	O	O	S	T	W	N	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
11	6	A	A	E	H	L	M	N	O	O	S	T	W	G	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
12	3	A	A	E	G	H	L	M	N	O	O	S	T	W	E	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
13	3	A	A	E	E	G	H	L	M	N	O	O	S	T	W	R	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
14	11	A	A	E	E	G	H	L	M	N	O	O	R	S	T	W	I	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
15	6	A	A	E	E	G	H	I	L	M	N	O	O	R	S	T	W	N	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
16	10	A	A	E	E	G	H	I	L	M	N	N	O	O	R	S	T	W	S	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
17	15	A	A	E	E	G	H	I	L	M	N	N	O	O	R	S	S	T	W	E	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
18	4	A	A	E	E	E	G	H	I	L	M	N	N	O	O	R	S	S	T	W	R	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
19	15	A	A	E	E	E	G	H	I	L	M	N	N	O	O	R	R	S	S	T	W	T	I	O	N	S	O	R	T	E	X	A	M	P	L	E
20	19	A	A	E	E	E	G	H	I	L	M	N	N	O	O	R	R	S	S	T	T	W	I	O	N	S	O	R	T	E	X	A	M	P	L	E
21	8	A	A	E	E	E	G	H	I	I	L	M	N	N	O	O	R	R	S	S	T	T	W	O	N	S	O	R	T	E	X	A	M	P	L	E
22	15	A	A	E	E	E	G	H	I	I	L	M	N	N	O	O	O	R	R	S	S	T	T	W	N	S	O	R	T	E	X	A	M	P	L	E
23	13	A	A	E	E	E	G	H	I	I	L	M	N	N	N	O	O	O	R	R	S	S	T	T	W	S	O	R	T	E	X	A	M	P	L	E
24	21	A	A	E	E	E	G	H	I	I	L	M	N	N	N	O	O	O	R	R	S	S	S	T	T	W	O	R	T	E	X	A	M	P	L	E
25	17	A	A	E	E	E	G	H	I	I	L	M	N	N	N	O	O	O	O	R	R	S	S	S	T	T	W	R	T	E	X	A	M	P	L	E
26	20	A	A	E	E	E	G	H	I	I	L	M	N	N	N	O	O	O	O	R	R	R	S	S	S	T	T	W	T	E	X	A	M	P	L	E
27	26	A	A	E	E	E	G	H	I	I	L	M	N	N	N	O	O	O	O	R	R	R	S	S	S	T	T	T	W	E	X	A	M	P	L	E
28	5	A	A	E	E	E	E	G	H	I	I	L	M	N	N	N	O	O	O	O	R	R	R	S	S	S	T	T	T	W	X	A	M	P	L	E
29	29	A	A	E	E	E	E	G	H	I	I	L	M	N	N	N	O	O	O	O	R	R	R	S	S	S	T	T	T	W	X	A	M	P	L	E
30	2	A	A	A	E	E	E	E	G	H	I	I	L	M	N	N	N	O	O	O	O	R	R	R	S	S	S	T	T	T	W	X	M	P	L	E
31	13	A	A	A	E	E	E	E	G	H	I	I	L	M	M	N	N	N	O	O	O	O	R	R	R	S	S	S	T	T	T	W	X	P	L	E
32	21	A	A	A	E	E	E	E	G	H	I	I	L	M	M	N	N	N	O	O	O	O	P	R	R	R	S	S	S	T	T	T	W	X	L	E
33	12	A	A	A	E	E	E	E	G	H	I	I	L	L	M	M	N	N	N	O	O	O	O	P	R	R	R	S	S	S	T	T	T	W	X	E
34	7	A	A	A	E	E	E	E	E	G	H	I	I	L	L	M	M	N	N	N	O	O	O	O	P	R	R	R	S	S	S	T	T	T	W	X
		A	A	A	E	E	E	E	E	G	H	I	I	L	L	M	M	N	N	N	O	O	O	O	P	R	R	R	S	S	S	T	T	T	W	X

Insertion sort animation

40 random elements



http://www.sorting-algorithms.com/insertion-sort

Insertion sort: best and worst case

Best case. If the input is in ascending order, insertion sort makes N-1 compares and 0 exchanges.

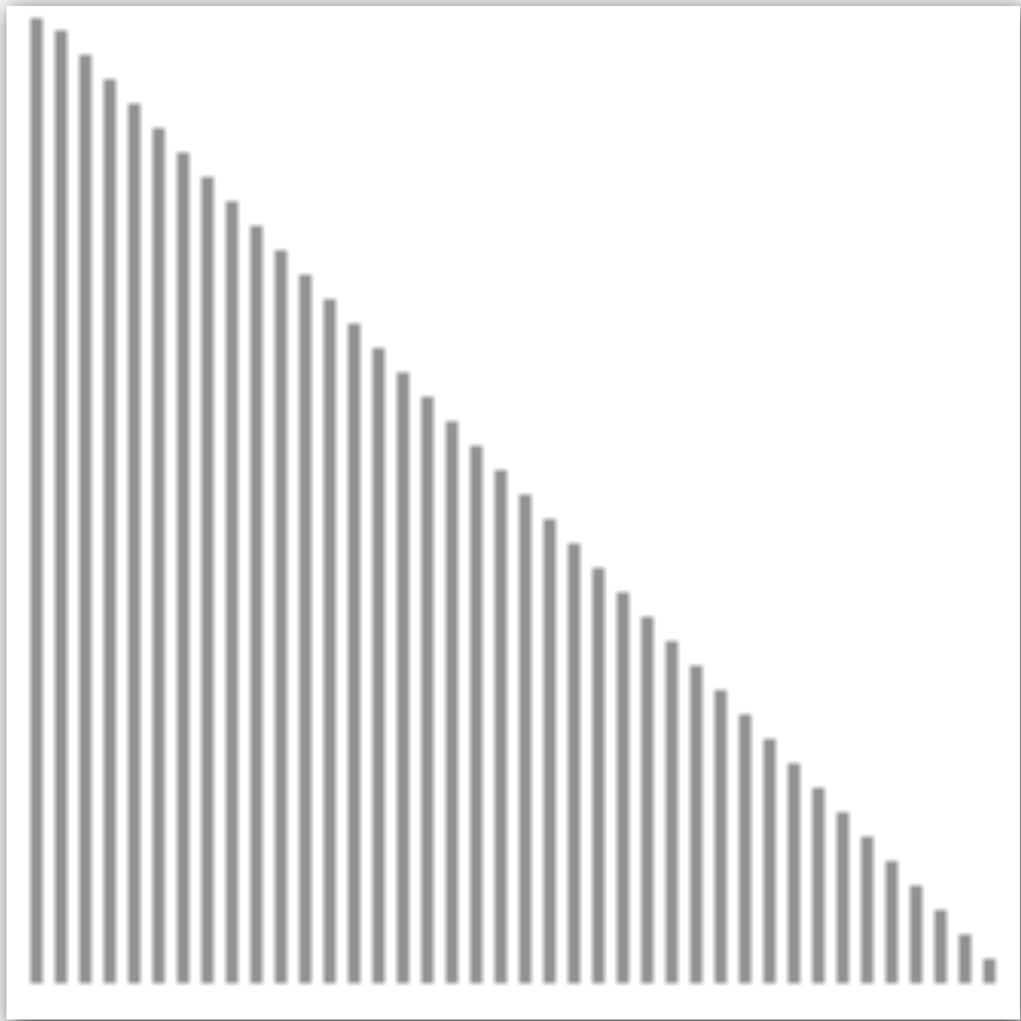
```
A E E L M O P R S T X
```

Worst case. If the input is in descending order (and no duplicates), insertion sort makes ~ $N^2/2$ compares and ~ $N^2/2$ exchanges.

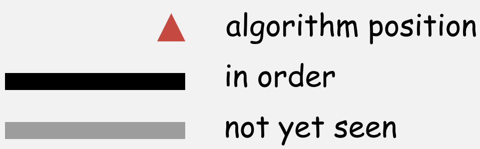
```
X T S R P O M L E E A
```

Insertion sort animation

40 reverse-sorted elements



http://www.sorting-algorithms.com/insertion-sort



Insertion sort: partially sorted inputs

Def. An inversion is a pair of keys that are out of order.

```
A E E L M O T R X P S
```

T-R T-P T-S R-P X-P X-S

(6 inversions)

Def. An array is partially sorted if the number of inversions is O(N).

- Ex 1. A small array appended to a large sorted array.
- Ex 2. An array with only a few elements out of place.

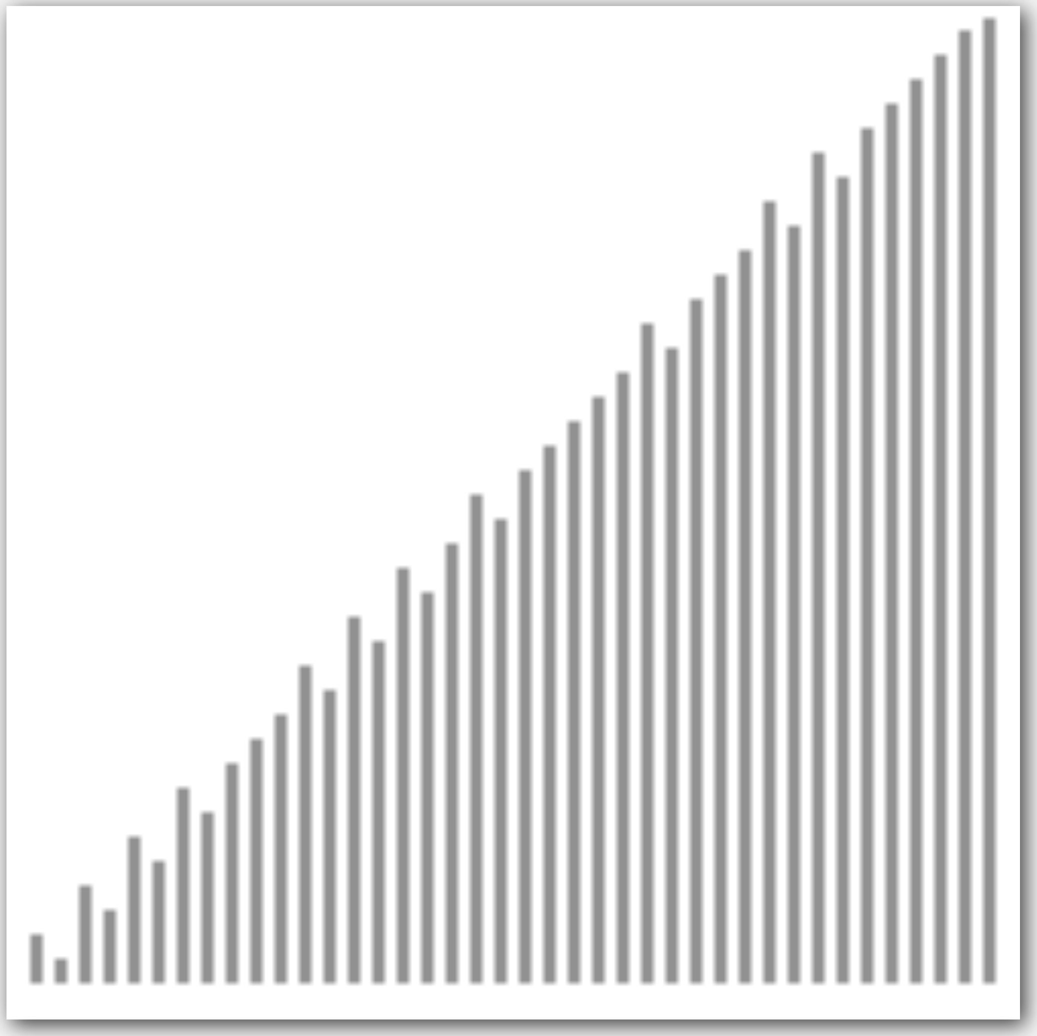
Proposition C. For partially-sorted arrays, insertion sort runs in linear time.

Pf. Number of exchanges equals the number of inversions.

number of compares = exchanges + (N-1)

Insertion sort animation

40 partially-sorted elements



algorithm position

in order

not yet seen

http://www.sorting-algorithms.com/insertion-sort

- rules of the game
- selection sort
- insertion sort
- **sorting challenges**
- shellsort

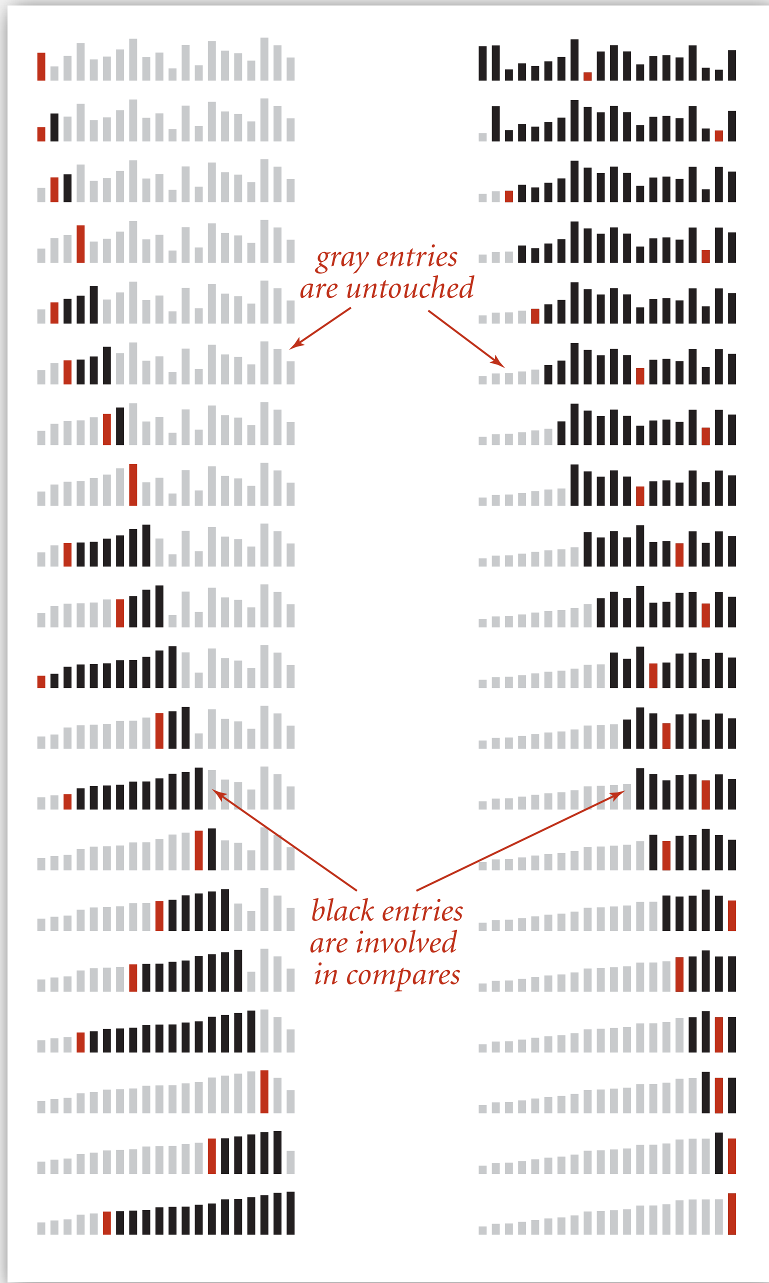
Sorting challenge 0

Input. Array of doubles.

Plot. Data proportional to length.

Name the sorting method.

- Insertion sort.
- Selection sort.



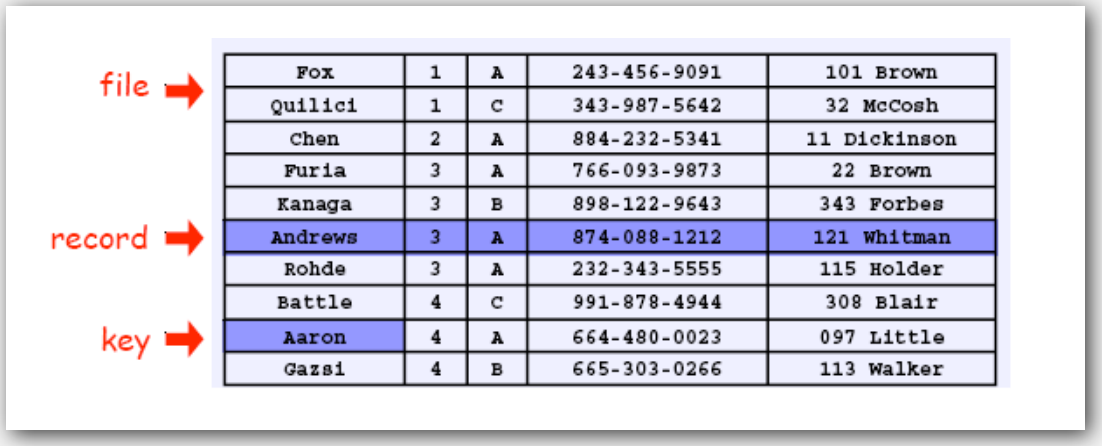
Sorting challenge 1

Problem. Sort a file of huge records with tiny keys.

Ex. Reorganize your MP3 files.

Which sorting method to use?

- System sort.
- Insertion sort.
- Selection sort.



Fox	1	A	243-456-9091	101 Brown
Quilici	1	C	343-987-5642	32 McCosh
Chen	2	A	884-232-5341	11 Dickinson
Furia	3	A	766-093-9873	22 Brown
Kanaga	3	B	898-122-9643	343 Forbes
Andrews	3	A	874-088-1212	121 Whitman
Rohde	3	A	232-343-5555	115 Holder
Battle	4	C	991-878-4944	308 Blair
Aaron	4	A	664-480-0023	097 Little
Gazsi	4	B	665-303-0266	113 Walker

Sorting challenge 2

Problem. Sort a huge randomly-ordered file of small records.

Ex. Process transaction records for a phone company.

Which sorting method to use?

- System sort.
- Insertion sort.
- Selection sort.

file	Fox	1	A	243-456-9091	101 Brown
	Quilici	1	C	343-987-5642	32 McCosh
	Chen	2	A	884-232-5341	11 Dickinson
	Furia	3	A	766-093-9873	22 Brown
	Kanaga	3	B	898-122-9643	343 Forbes
record	Andrews	3	A	874-088-1212	121 Whitman
	Rohde	3	A	232-343-5555	115 Holder
	Battle	4	C	991-878-4944	308 Blair
key	Aaron	4	A	664-480-0023	097 Little
	Gazsi	4	B	665-303-0266	113 Walker

Sorting challenge 3

Problem. Sort a huge number of tiny files (each file is independent).

Ex. Daily customer transaction records.

Which sorting method to use?

- System sort.
- Insertion sort.
- Selection sort.

file	Fox	1	A	243-456-9091	101 Brown
	Quilici	1	C	343-987-5642	32 McCosh
	Chen	2	A	884-232-5341	11 Dickinson
	Furia	3	A	766-093-9873	22 Brown
	Kanaga	3	B	898-122-9643	343 Forbes
record	Andrews	3	A	874-088-1212	121 Whitman
	Rohde	3	A	232-343-5555	115 Holder
	Battle	4	C	991-878-4944	308 Blair
key	Aaron	4	A	664-480-0023	097 Little
	Gazsi	4	B	665-303-0266	113 Walker

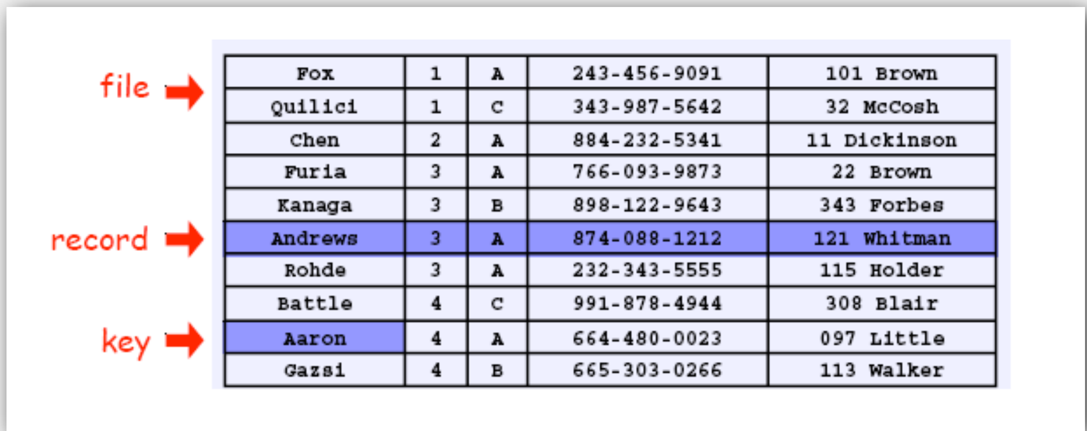
Sorting challenge 4

Problem. Sort a huge file that is already almost in order.

Ex. Resort a huge database after a few changes.

Which sorting method to use?

- System sort.
- Insertion sort.
- Selection sort.



Fox	1	A	243-456-9091	101 Brown
Quilici	1	C	343-987-5642	32 McCosh
Chen	2	A	884-232-5341	11 Dickinson
Furia	3	A	766-093-9873	22 Brown
Kanaga	3	B	898-122-9643	343 Forbes
Andrews	3	A	874-088-1212	121 Whitman
Rohde	3	A	232-343-5555	115 Holder
Battle	4	C	991-878-4944	308 Blair
Aaron	4	A	664-480-0023	097 Little
Gazsi	4	B	665-303-0266	113 Walker

- rules of the game
- selection sort
- insertion sort
- animations
- **shellsort**

Shellsort overview

Idea. Move elements more than one position at a time by h-sorting the array.

an h-sorted array is h interleaved sorted subsequences

h = 4

L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
L				M				P				T			
	E				H				S				S		
		E				L				O				X	
			A				E				L				R

Shellsort. h-sort the array for a decreasing sequence of values of h.

input	S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E
13-sort	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
4-sort	L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
1-sort	A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X

h-sorting

How to h-sort an array? Insertion sort, with stride length h.

3-sorting an array

M	O	L	E	E	X	A	S	P	R	T
E	O	L	M	E	X	A	S	P	R	T
E	E	L	M	O	X	A	S	P	R	T
E	E	L	M	O	X	A	S	P	R	T
A	E	L	E	O	X	M	S	P	R	T
A	E	L	E	O	X	M	S	P	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T

Why insertion sort?

- Big increments $\Rightarrow$ small subarray.
- Small increments $\Rightarrow$ nearly in order. [stay tuned]

Shellsort example: increments 7, 3, 1

input

S	O	R	T	E	X	A	M	P	L	E

7-sort

S	O	R	T	E	X	A	M	P	L	E
M	O	R	T	E	X	A	S	P	L	E
M	O	R	T	E	X	A	S	P	L	E
M	O	L	T	E	X	A	S	P	R	E
M	O	L	E	E	X	A	S	P	R	T

3-sort

M	O	L	E	E	X	A	S	P	R	T
E	O	L	M	E	X	A	S	P	R	T
E	E	L	M	O	X	A	S	P	R	T
E	E	L	M	O	X	A	S	P	R	T
A	E	L	E	O	X	M	S	P	R	T
A	E	L	E	O	X	M	S	P	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T

1-sort

A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	E	L	O	P	M	S	X	R	T
A	E	E	L	O	P	M	S	X	R	T
A	E	E	L	O	P	M	S	X	R	T
A	E	E	L	M	O	P	S	X	R	T
A	E	E	L	M	O	P	S	X	R	T
A	E	E	L	M	O	P	S	X	R	T
A	E	E	L	M	O	P	R	S	X	T
A	E	E	L	M	O	P	R	S	T	X

result

A	E	E	L	M	O	P	R	S	T	X

Shellsort: intuition

Proposition. A g-sorted array remains g-sorted after h-sorting it.

Pf. Harder than you'd think!

7-sort

M	O	R	T	E	X	A	S	P	L	E
M	O	R	T	E	X	A	S	P	L	E
M	O	L	T	E	X	A	S	P	R	E
M	O	L	E	E	X	A	S	P	R	T
M	O	L	E	E	X	A	S	P	R	T

3-sort

M	O	L	E	E	X	A	S	P	R	T
E	O	L	M	E	X	A	S	P	R	T
E	E	L	M	O	X	A	S	P	R	T
E	E	L	M	O	X	A	S	P	R	T
A	E	L	E	O	X	M	S	P	R	T
A	E	L	E	O	X	M	S	P	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T

still 7-sorted

What increments to use?

1, 2, 4, 8, 16, 32 . . .

No.

1, 3, 7, 15, 31, 63, . . .

Maybe.

→ 1, 4, 13, 40, 121, 364, . . .

OK, easy to compute 3x+1 sequence.

1, 5, 19, 41, 109, 209, 505, . . .

Tough to beat in empirical studies.

Interested in learning more?

- See Algs 3 section 6.8 or Knuth volume 3 for details.
- Consider doing a JP on the topic.

Shellsort: Java implementation

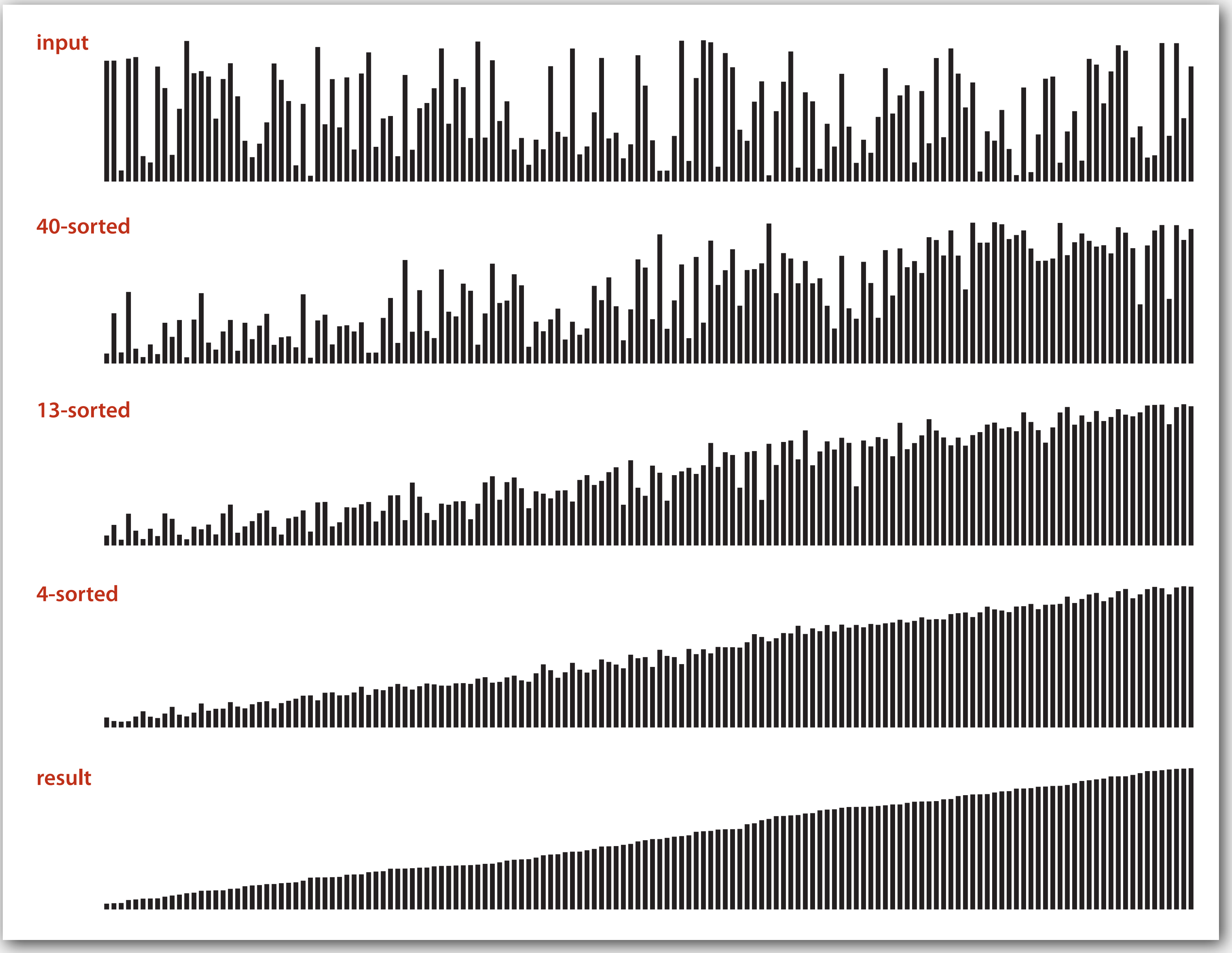
```
public class Shell
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;

      int h = 1;
      while (h < N/3) h = 3*h + 1; // 1, 4, 13, 40, 121, 364, 1093, ...

      while (h >= 1)
      {  // h-sort the array.
         for (int i = h; i < N; i++)
         {
            for (int j = i; j >= h && less(a[j], a[j-h]); j -= h)
               exch(a, j, j-h);
         }

         h = h/3;
      }
   }

   private static boolean less(Comparable v, Comparable w)
   { /* as before */ }
   private static boolean void(Comparable[] a, int i, int j)
   { /* as before */ }
}
```

magic increment sequence

insertion sort

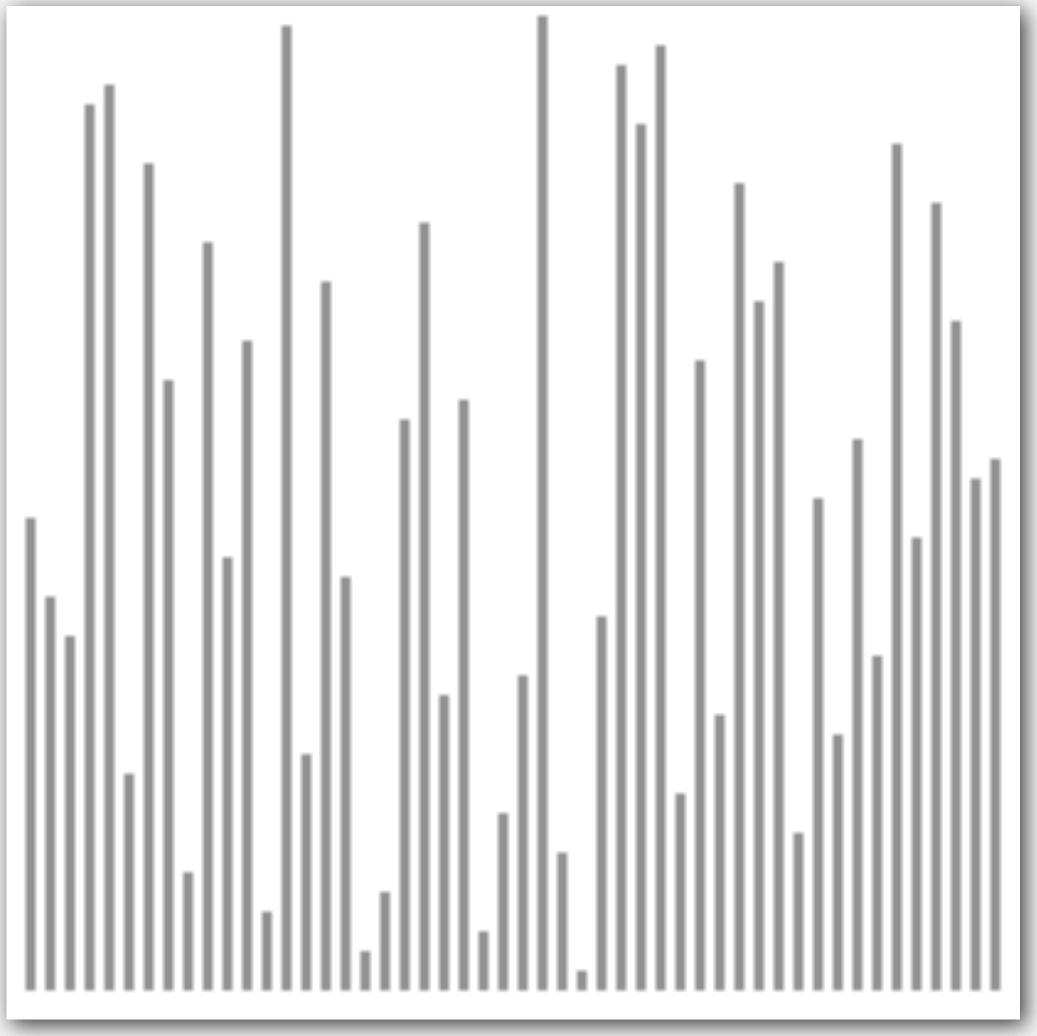
move to next increment

Visual trace of shellsort



Shellsort animation

50 random elements

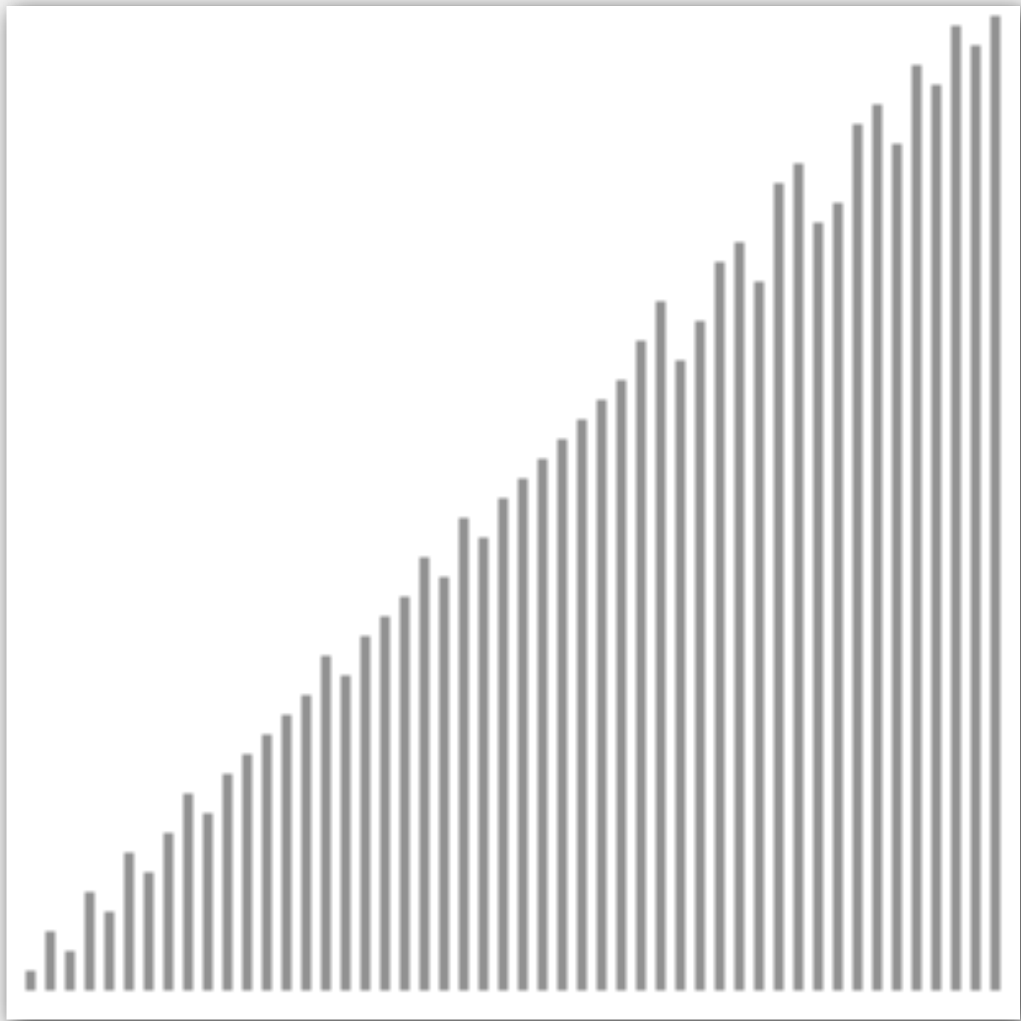


http://www.sorting-algorithms.com/shell-sort

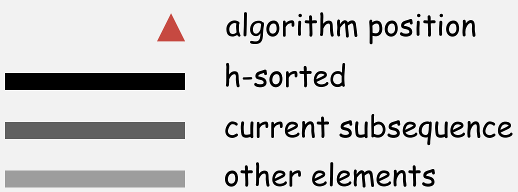
algorithm position
h-sorted
current subsequence
other elements

Shellsort animation

50 partially-sorted elements



http://www.sorting-algorithms.com/shell-sort



Shellsort: analysis

Proposition. The worst-case number of compares used by shellsort with the 3x+1 increments is $O(N^{3/2})$.

Property. The number of compares used by shellsort with the 3x+1 increments is at most by a small multiple of N times the # of increments used.

N	compares	$N^{1.289}$	2.5 N lg N
5,000	93	58	106
10,000	209	143	230
20,000	467	349	495
40,000	1022	855	1059
80,000	2266	2089	2257

measured in thousands

Remark. Accurate model has not yet been discovered (!)

Why are we interested in shellsort?

Example of simple idea leading to substantial performance gains.

Useful in practice.

- Fast unless array size is huge.
- Tiny, fixed footprint for code (used in embedded systems).
- Hardware sort prototype.

Simple algorithm, nontrivial performance, interesting questions.

- Asymptotic growth rate?
- Best sequence of increments? ← open problem: find a better increment sequence
- Average case performance?

Lesson. Some good algorithms are still waiting discovery.

2.2 Mergesort

- mergesort
- bottom-up mergesort
- sorting complexity
- comparators

Two classic sorting algorithms

Critical components in the world's computational infrastructure.

- Full scientific understanding of their properties has enabled us to develop them into practical system sorts.
- Quicksort honored as one of top 10 algorithms of 20th century in science and engineering.

Mergesort.

← today

- Java sort for objects.
- Perl, Python stable sort.

Quicksort.

← next lecture

- Java sort for primitive types.
- C qsort, Unix, g++, Visual C++, Python.

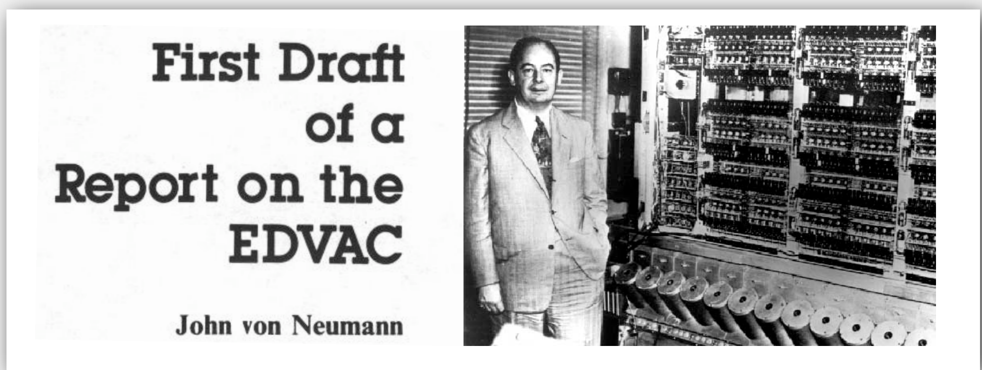
- **mergesort**
- bottom-up mergesort
- sorting complexity
- comparators

Basic plan.

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves.

input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Mergesort overview



Merging

Q. How to combine two sorted subarrays into a sorted whole.

A. Use an auxiliary array.

		a[]												aux[]									
	k	0	1	2	3	4	5	6	7	8	9	i	j	0	1	2	3	4	5	6	7	8	9
input		E	E	G	M	R	A	C	E	R	T			-	-	-	-	-	-	-	-	-	-
copy		E	E	G	M	R	A	C	E	R	T			E	E	G	M	R	A	C	E	R	T
												0	5										
	0	A										0	6	E	E	G	M	R	A	C	E	R	T
	1	A	C									0	7	E	E	G	M	R		C	E	R	T
	2	A	C	E								1	7	E	E	G	M	R			E	R	T
	3	A	C	E	E							2	7		E	G	M	R			E	R	T
	4	A	C	E	E	E						2	8			G	M	R			E	R	T
	5	A	C	E	E	E	G					3	8			G	M	R				R	T
	6	A	C	E	E	E	G	M				4	8				M	R				R	T
	7	A	C	E	E	E	G	M	R			5	8					R				R	T
	8	A	C	E	E	E	G	M	R	R		5	9									R	T
	9	A	C	E	E	E	G	M	R	R	T	6	10										T
merged result		A	C	E	E	E	G	M	R	R	T												

Abstract in-place merge trace

Merging: Java implementation

```
private static void merge(Comparable[] a, int lo, int mid, int hi)
{
   assert isSorted(a, lo, mid);     // precondition: a[lo..mid]   sorted
   assert isSorted(a, mid+1, hi);   // precondition: a[mid+1..hi] sorted

   for (int k = lo; k <= hi; k++)
      aux[k] = a[k];

   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)
   {
      if      (i > mid)              a[k] = aux[j++];
      else if (j > hi)               a[k] = aux[i++];
      else if (less(aux[j], aux[i])) a[k] = aux[j++];
      else                           a[k] = aux[i++];
   }

   assert isSorted(a, lo, hi);     // postcondition: a[lo..hi] sorted
}
```

copy

merge

	lo			i	mid			j		hi
aux[]	A	G	L	O	R	H	I	M	S	T
							k			
a[]	A	G	H	I	L	M				

Assertion. Statement to test assumptions about your program.

- Helps detect logic bugs.
- Documents code.

Java assert statement. Throws an exception unless boolean condition is ture.

```
assert isSorted(a, lo, hi);
```

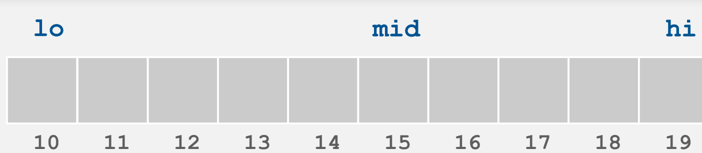
Can enable or disable at runtime. $\Rightarrow$ No cost in production code.

```
java -ea MyProgram   // enable assertions
java -da MyProgram   // disable assertions (default)
```

Best practices. Use to check internal invariants. Assume assertions will be disabled in production code (e.g., don't use for external argument-checking).

Mergesort: Java implementation

```
public class Merge
{
   private static Comparable[] aux;

   private static void merge(Comparable[] a, int lo, int mid, int hi)
   {  /* as before */  }

   private static void sort(Comparable[] a, int lo, int hi)
   {
      if (hi <= lo) return;
      int mid = lo + (hi - lo) / 2;
      sort(a, lo, mid);
      sort(a, mid+1, hi);
      merge(a, lo, m, hi);
   }

   public static void sort(Comparable[] a)
   {
      aux = new Comparable[a.length];
      sort(a, 0, a.length - 1);
   }
}
```



Mergesort trace

lo hi

a[]

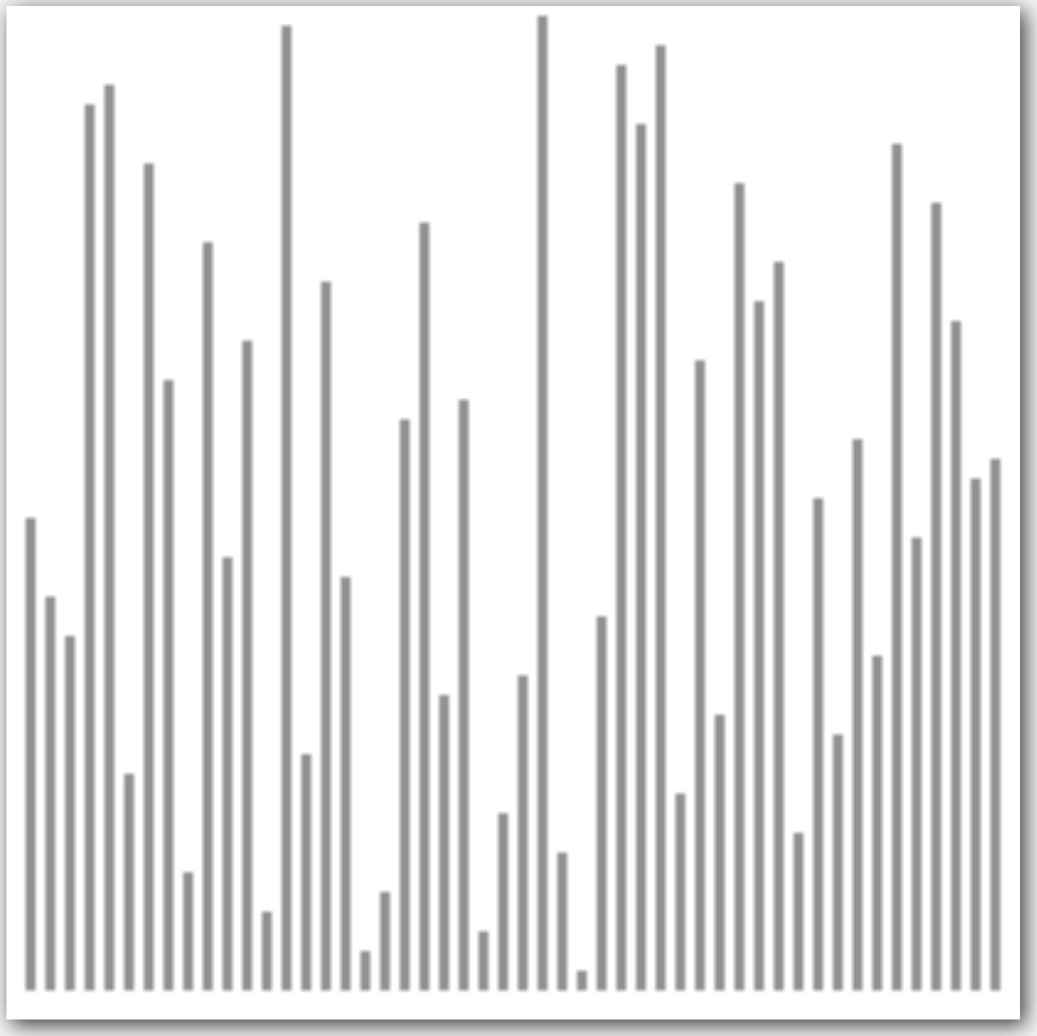
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, 0, 0, 1)`	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, 2, 2, 3)`	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, 0, 1, 3)`	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, 4, 4, 5)`	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, 6, 6, 7)`	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
`merge(a, 4, 5, 7)`	E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
`merge(a, 0, 3, 7)`	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
`merge(a, 8, 8, 9)`	E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
`merge(a, 10, 10, 11)`	E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
`merge(a, 8, 9, 11)`	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
`merge(a, 12, 12, 13)`	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
`merge(a, 14, 14, 15)`	E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
`merge(a, 12, 13, 15)`	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
`merge(a, 8, 11, 15)`	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
`merge(a, 0, 7, 15)`	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Trace of merge results for top-down mergesort

result after recursive call

Mergesort animation

50 random elements



http://www.sorting-algorithms.com/merge-sort

algorithm position

in order

current subarray

not in order

Mergesort animation

50 reverse-sorted elements

http://www.sorting-algorithms.com/merge-sort

algorithm position

in order

current subarray

not in order

Mergesort: empirical analysis

Running time estimates:

- Home pc executes 10^8 comparisons/second.
- Supercomputer executes 10^{12} comparisons/second.

	insertion sort (N^2)			mergesort (N log N)		
computer	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

Bottom line. Good algorithms are better than supercomputers.

Mergesort: mathematical analysis

Proposition. Mergesort uses $\sim 2\,N \lg N$ data moves to sort any array of size N.

Def. $\mathrm{D}(N)$ = number of data moves to mergesort an array of size N.

$$= \underbrace{\mathrm{D}(N/2)}_{\text{left half}} + \underbrace{\mathrm{D}(N/2)}_{\text{right half}} + \underbrace{2\,N}_{\text{merge}}$$

Mergesort recurrence. $\mathrm{D}(N) = 2\,\mathrm{D}(N/2) + 2\,N$ for $N > 1$, with $\mathrm{T}(1) = 0$.

- Not quite right for odd N.
- Similar recurrence holds for many divide-and-conquer algorithms.

Solution. $\mathrm{D}(N) \sim 2\,N \lg N$.

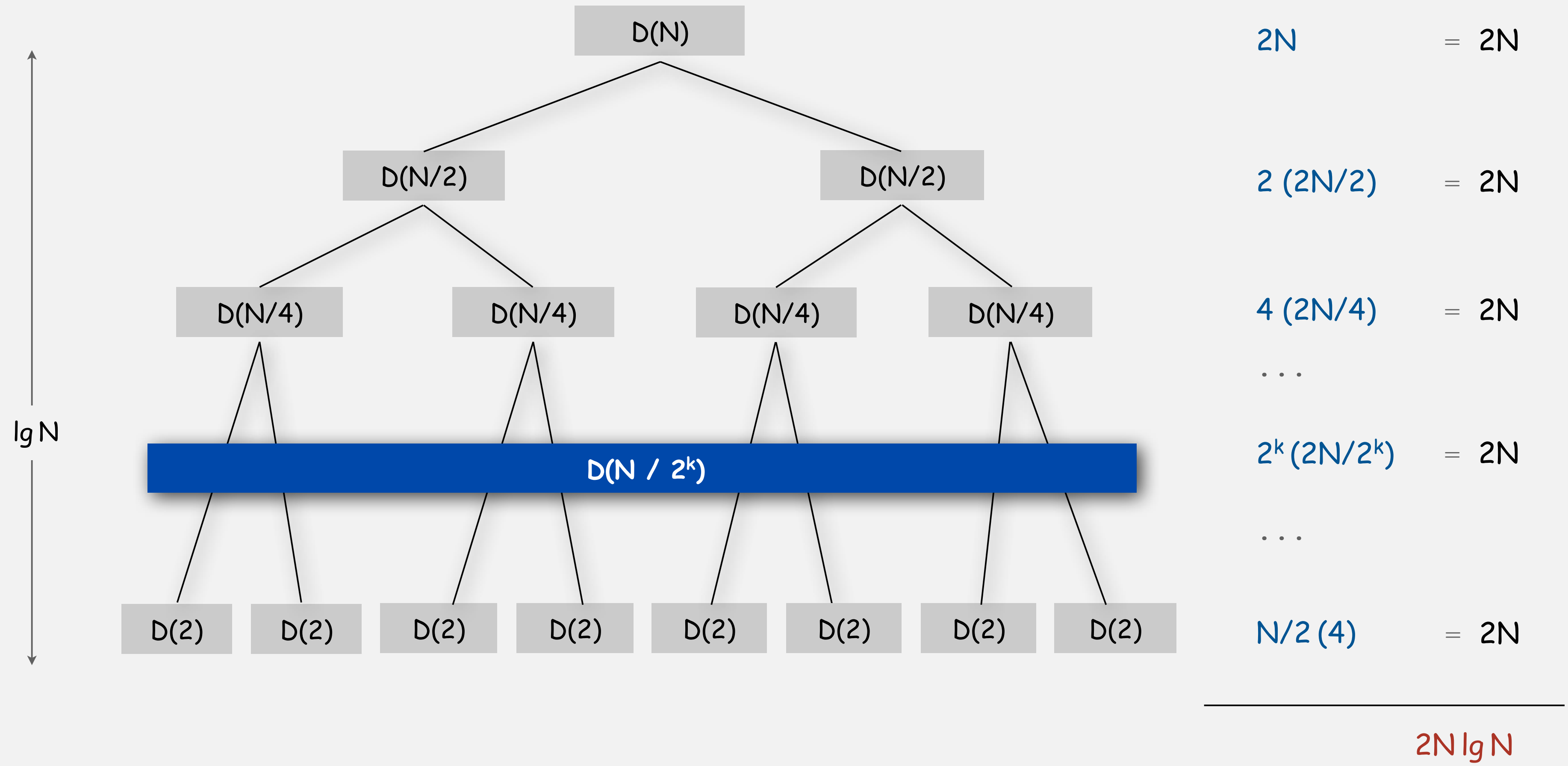
- For simplicity, we'll prove when N is a power of 2.
- True for all N. [see COS 340]

Mergesort recurrence: proof 1

Mergesort recurrence. $D(N) = 2\,D(N/2) + 2\,N$ for $N > 1$, with $D(1) = 0$.

Proposition. If N is a power of 2, then $D(N) = 2\,N \lg N$.

Pf.



Mergesort recurrence: proof 2

Mergesort recurrence. $D(N) = 2\,D(N/2) + 2\,N$ for $N > 1$, with $D(1) = 0$.

Proposition. If N is a power of 2, then $D(N) = 2\,N \lg N$.

Pf.

$D(N)$	$= 2\,D(N/2) + 2N$	given
$D(N) / N$	$= 2\,D(N/2) / N + 2$	divide both sides by N
	$= D(N/2) / (N/2) + 2$	algebra
	$= D(N/4) / (N/4) + 2 + 2$	apply to first term
	$= D(N/8) / (N/8) + 2 + 2 + 2$	apply to first term again
	$\ldots$	
	$= D(N/N) / (N/N) + 2 + 2 + \ldots + 2$	stop applying, T(1) = 0
	$= 2 \lg N$	

Mergesort recurrence: proof 3

Mergesort recurrence. $D(N) = 2\,D(N/2) + 2\,N$ for $N > 1$, with $D(1) = 0$.

Proposition. If N is a power of 2, then $D(N) = 2\,N \lg N$.

Pf. [by induction on N]

- Base case: $N = 1$.
- Inductive hypothesis: $D(N) = 2N \lg N$.
- Goal: show that $D(2N) = 2(2N)\lg(2N)$.

D(2N) = 2 D(N) + 4N	given
= 4 N lg N + 4 N	inductive hypothesis
= 4 N (lg (2N) - 1) + 4N	algebra
= 4 N lg (2N)	QED

Mergesort: number of compares

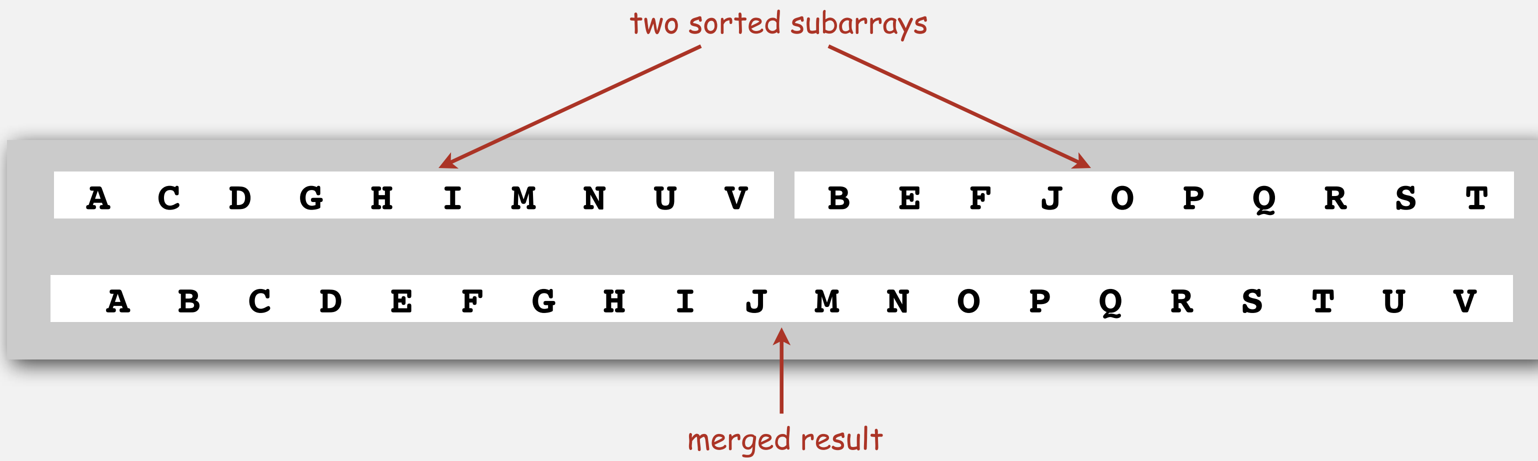
Proposition. Mergesort uses between $\frac{1}{2} N \lg N$ and $N \lg N$ compares to sort any array of size N.

Pf. The number of compares for the last merge is between $\frac{1}{2} N \lg N$ and N.

Mergesort analysis: memory

Proposition G. Mergesort uses extra space proportional to N.

Pf. The array `aux[]` needs to be of size N for the last merge.



Def. A sorting algorithm is in-place if it uses O(log N) extra memory.

Ex. Insertion sort, selection sort, shellsort.

Challenge for the bored. In-place merge. [Kronrud, 1969]

Mergesort: practical improvements

Use insertion sort for small subarrays.

- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 7 elements.

Stop if already sorted.

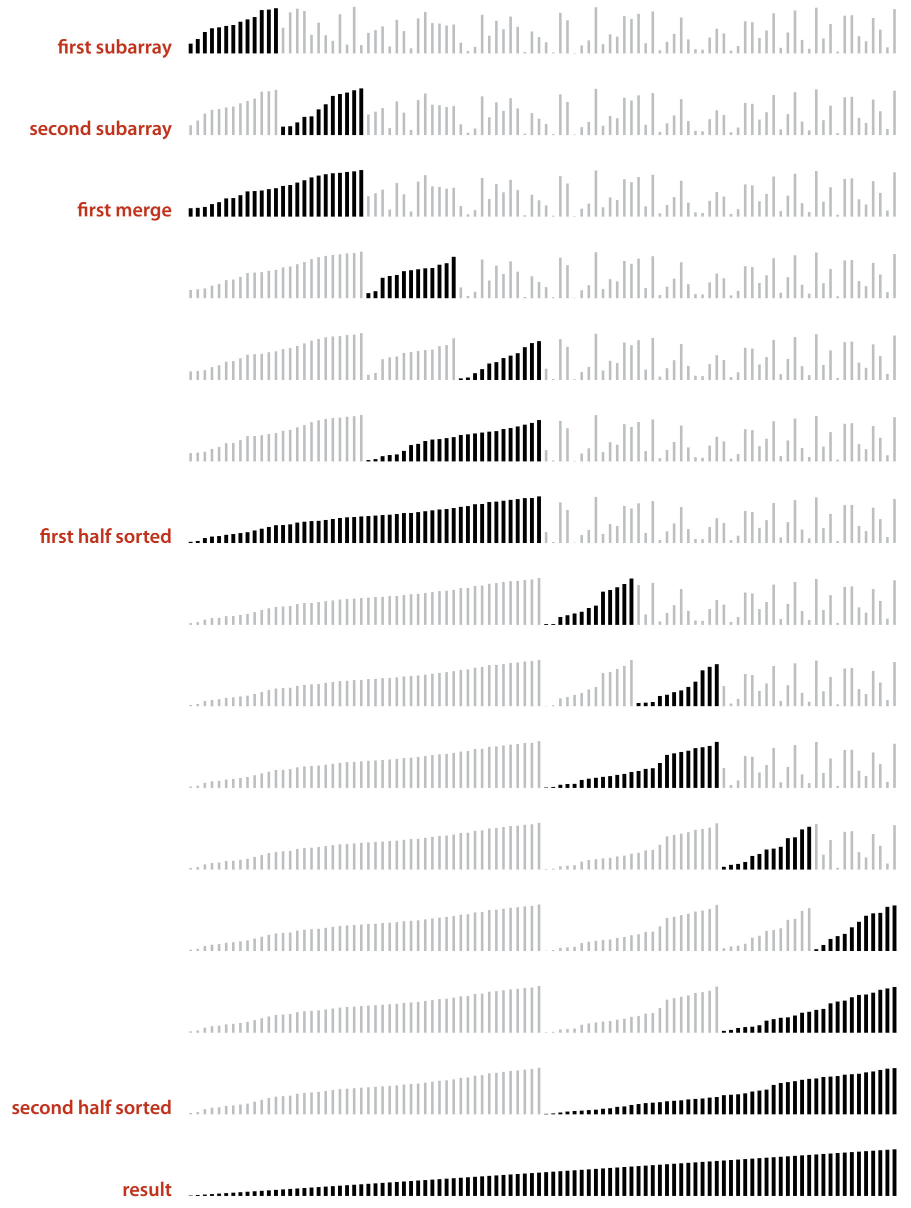
- Is biggest element in first half ≤ smallest element in second half?
- Helps for partially-ordered arrays.

A	B	C	D	E	F	G	H	I	J	M	N	O	P	Q	R	S	T	U	V
A	B	C	D	E	F	G	H	I	J	M	N	O	P	Q	R	S	T	U	V

Eliminate the copy to the auxiliary array. Save time (but not space) by switching the role of the input and auxiliary array in each recursive call.

Ex. See `MergeX.java` or `Arrays.sort()`.

Mergesort visualization



Visual trace of top-down mergesort for with cutoff for small subarrays

Bottom-up mergesort

Basic plan.

- Pass through array, merging subarrays of size 1.
- Repeat for subarrays of size 2, 4, 8, 16, ...

	a[i] 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sz = 2																
merge(a, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 4, 5)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 6, 6, 7)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 8, 8, 9)	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
merge(a, 10, 10, 11)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, 12, 12, 13)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, 14, 14, 15)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L
sz = 4																
merge(a, 0, 1, 3)	E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L
merge(a, 4, 5, 7)	E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L
merge(a, 8, 9, 11)	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
merge(a, 12, 13, 15)	E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P
sz = 8																
merge(a, 0, 3, 7)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
sz = 16																
merge(a, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Trace of merge results for bottom-up mergesort

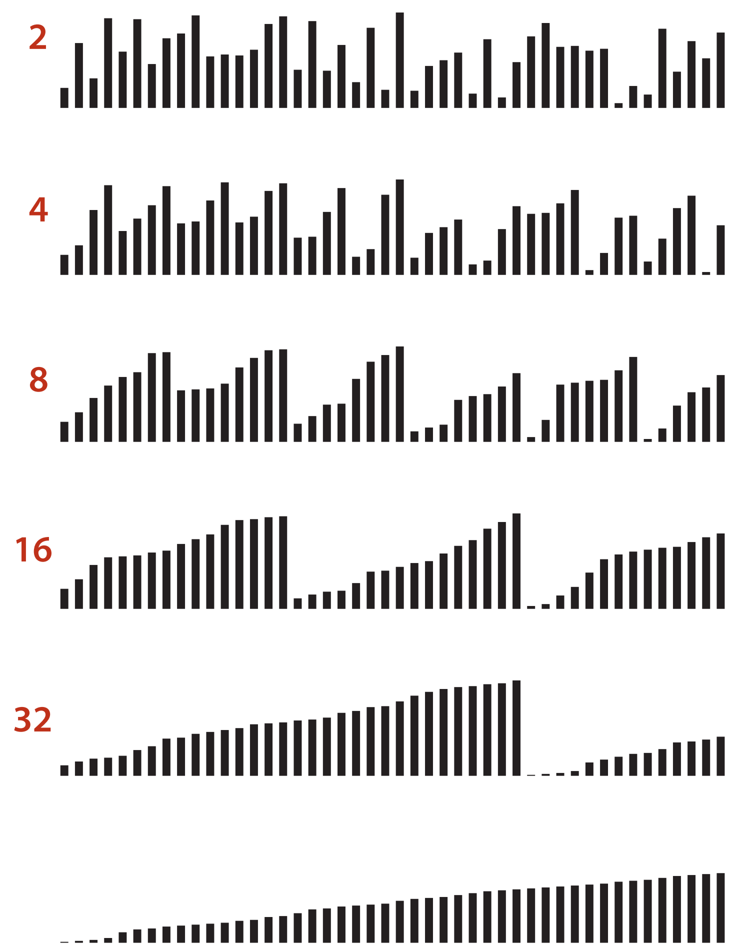
Bottom line. No recursion needed!

Bottom-up mergesort: Java implementation

```
public class MergeBU
{
   private static Comparable[] aux;

   private static void merge(Comparable[] a, int lo, int mid, int hi)
   {  /* as before */   }

   public static void sort(Comparable[] a)
   {
      int N = a.length;
      aux = new Comparable[N];
      for (int sz = 1; sz < N; sz = sz+sz)
         for (int lo = 0; lo < N-sz; lo += sz+sz)
            merge(a, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
   }
}
```

Bottom line. Concise industrial-strength code, if you have the space.

Bottom-up mergesort: visual trace



Visual trace of bottom-up mergesort

- mergesort
- bottom-up mergesort
- **sorting complexity**
- comparators

Complexity of sorting

Computational complexity. Framework to study efficiency of algorithms for solving a particular problem X.

Machine model. Focus on fundamental operations.

Upper bound. Cost guarantee provided by some algorithm for X.

Lower bound. Proven limit on cost guarantee of all algorithms for X.

Optimal algorithm. Algorithm with best cost guarantee for X.

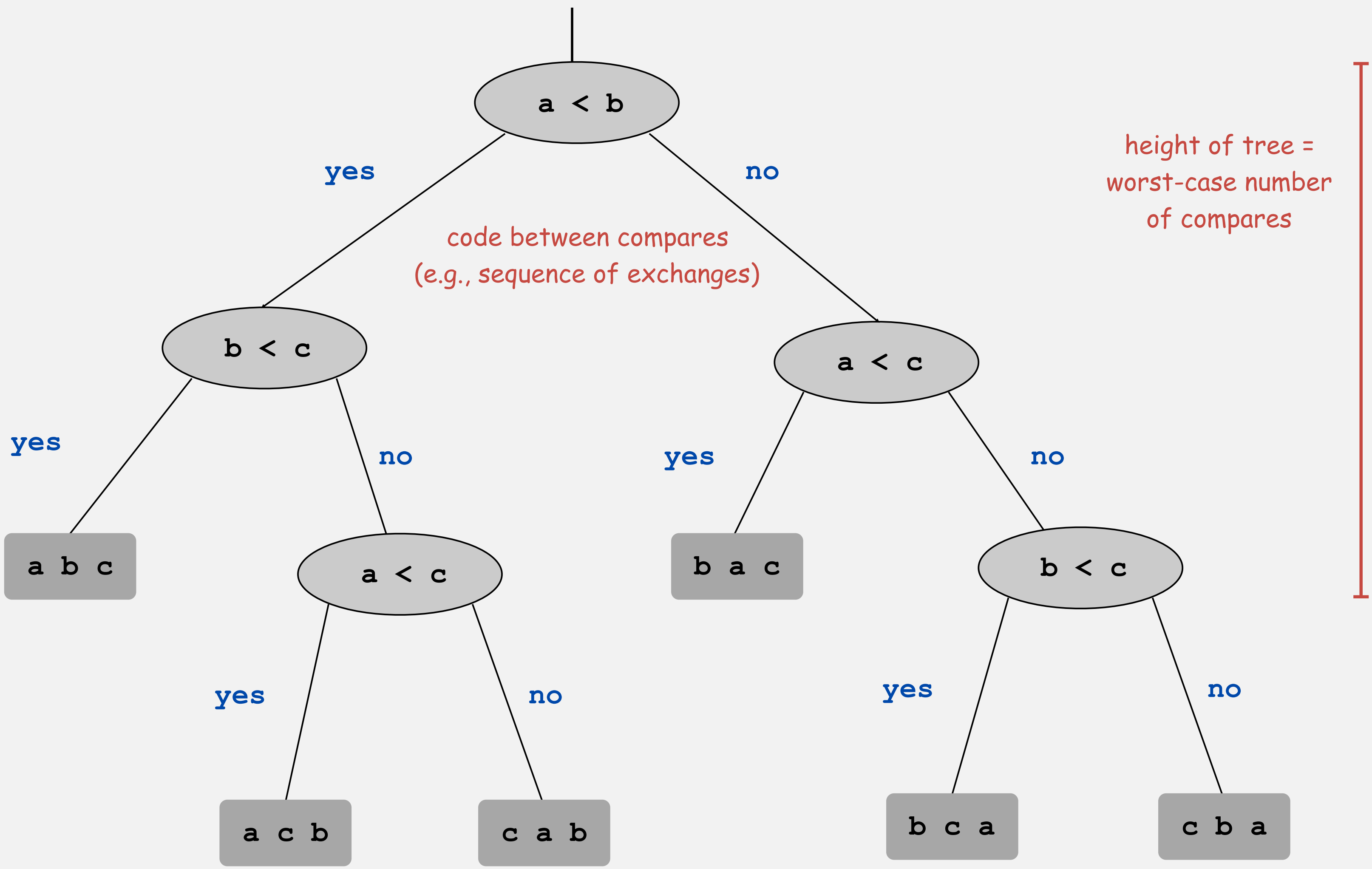
lower bound ~ upper bound

Example: sorting.

access information only through compares

- Machine model = # compares.
- Upper bound = ~ N lg N from mergesort.
- Lower bound = ~ N lg N ?
- Optimal algorithm = mergesort ?

Decision tree (for 3 distinct elements)

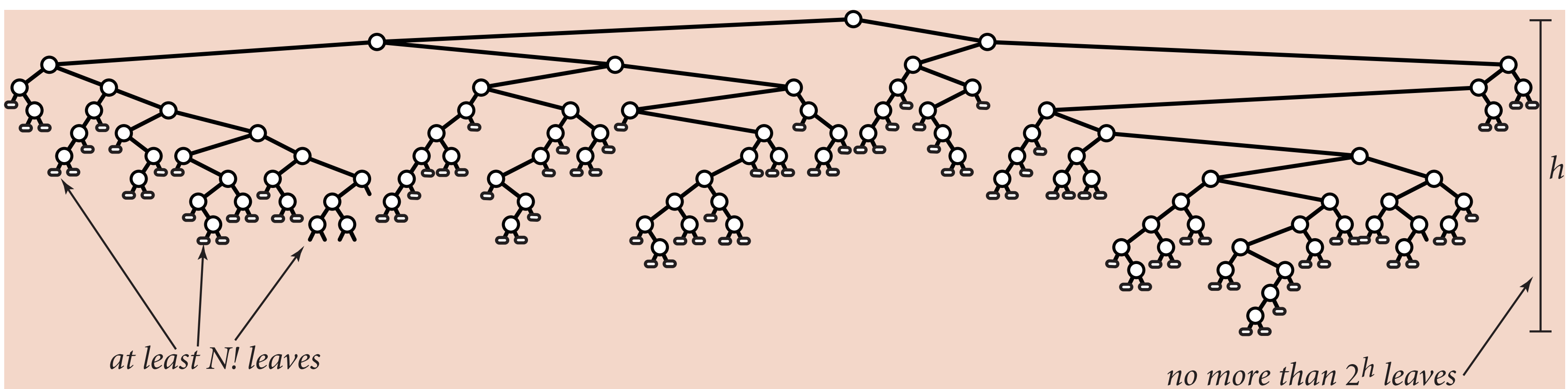


Compare-based lower bound for sorting

Proposition. Any compare-based sorting algorithm must use at least $\lg N! \sim N \lg N$ compares in the worst-case.

Pf.

- Assume input consists of N distinct values a_1 through a_N.
- Worst case dictated by height h of decision tree.
- Binary tree of height h has at most 2^h leaves.
- $N!$ different orderings $\Rightarrow$ at least $N!$ leaves.



Compare-based lower bound for sorting

Proposition. Any compare-based sorting algorithm must use at least $\lg N\,! \sim N \lg N$ compares in the worst-case.

Pf.

- Assume input consists of N distinct values a_1 through a_N.
- Worst case dictated by height h of decision tree.
- Binary tree of height h has at most 2^h leaves.
- $N\,!$ different orderings $\Rightarrow$ at least $N\,!$ leaves.

$$2^h \geq \#\,\text{leaves} \geq N\,!$$

$$\Rightarrow \; h \geq \lg N\,! \sim N \lg N$$

Stirling's formula

Complexity of sorting

Machine model. Focus on fundamental operations.

Upper bound. Cost guarantee provided by some algorithm for X.

Lower bound. Proven limit on cost guarantee of all algorithms for X.

Optimal algorithm. Algorithm with best cost guarantee for X.

Example: sorting.

- Machine model = # compares.
- Upper bound = ~ N lg N from mergesort.
- Lower bound = ~ N lg N.
- Optimal algorithm = mergesort.

First goal of algorithm design: optimal algorithms.

Complexity results in context

Other operations? Mergesort optimality is only about number of compares.

Space?

- Mergesort is not optimal with respect to space usage.
- Insertion sort, selection sort, and shellsort are space-optimal.

Challenge. Find an algorithm that is both time- and space-optimal.

Lessons. Use theory as a guide.

Ex. Don't try to design sorting algorithm that uses $\frac{1}{2} N \lg N$ compares.

Complexity results in context (continued)

Lower bound may not hold if the algorithm has information about:

- The initial order of the input.
- The distribution of key values.
- The representation of the keys.

Partially-ordered arrays. Depending on the initial order of the input, we may not need N lg N compares.

insertion sort requires only N-1 compares on an already sorted array

Duplicate keys. Depending on the input distribution of duplicates, we may not need N lg N compares.

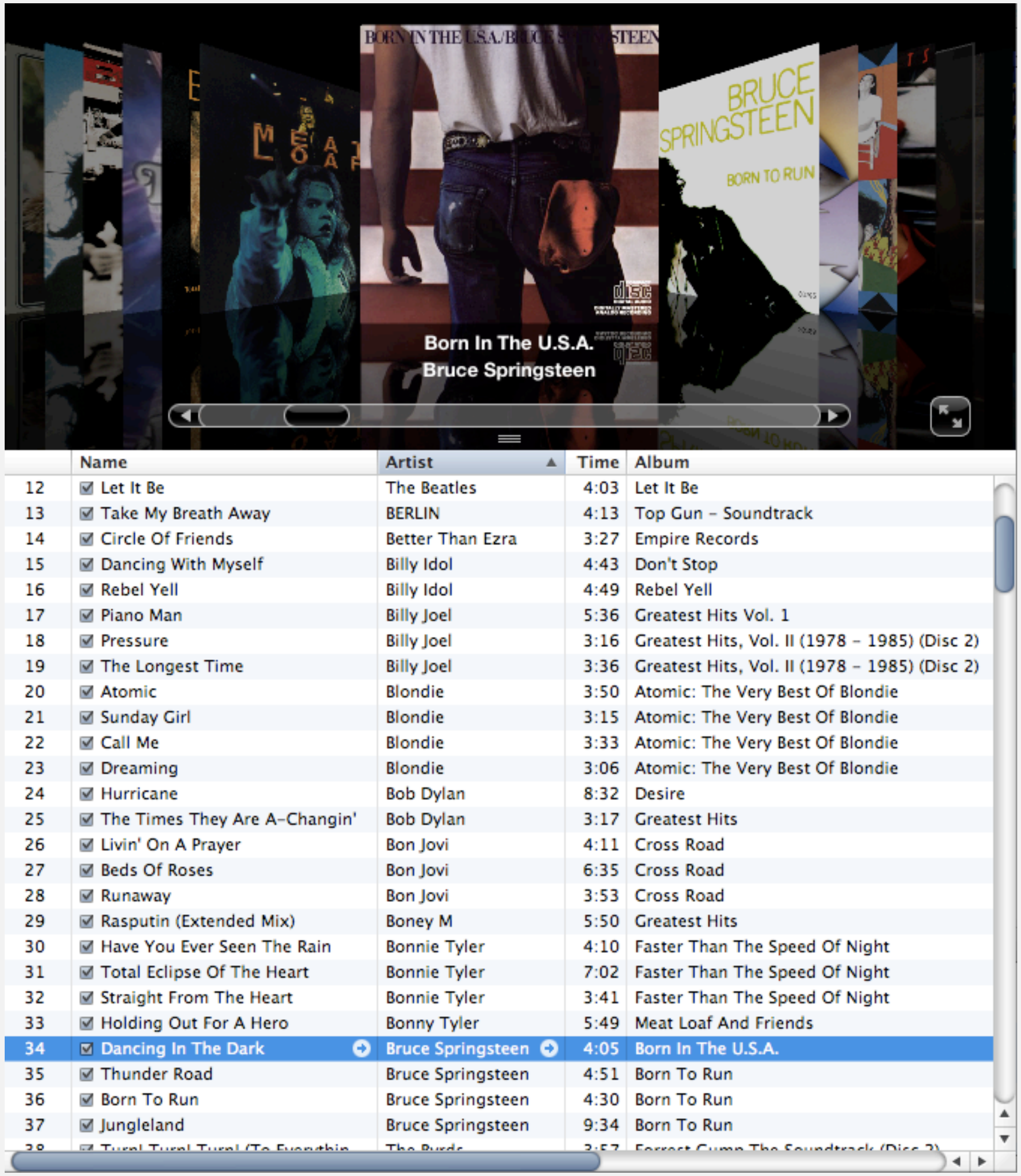
stay tuned for 3-way quicksort

Digital properties of keys. We can use digit/character compares instead of key compares for numbers and strings.

stay tuned for radix sorts

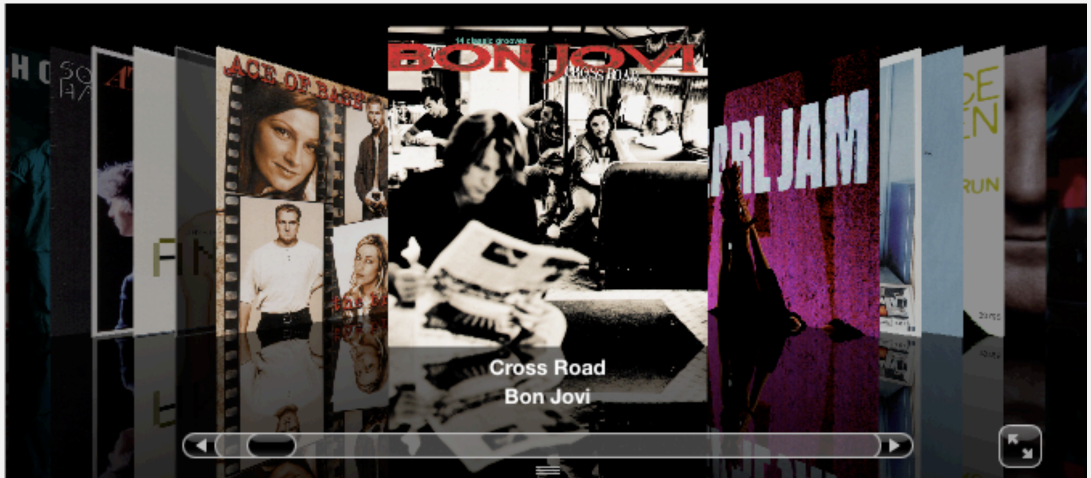
- mergesort
- bottom-up mergesort
- sorting complexity
- **comparators**

Sort by artist name



	Name	Artist	Time	Album
12	Let It Be	The Beatles	4:03	Let It Be
13	Take My Breath Away	BERLIN	4:13	Top Gun – Soundtrack
14	Circle Of Friends	Better Than Ezra	3:27	Empire Records
15	Dancing With Myself	Billy Idol	4:43	Don't Stop
16	Rebel Yell	Billy Idol	4:49	Rebel Yell
17	Piano Man	Billy Joel	5:36	Greatest Hits Vol. 1
18	Pressure	Billy Joel	3:16	Greatest Hits, Vol. II (1978 – 1985) (Disc 2)
19	The Longest Time	Billy Joel	3:36	Greatest Hits, Vol. II (1978 – 1985) (Disc 2)
20	Atomic	Blondie	3:50	Atomic: The Very Best Of Blondie
21	Sunday Girl	Blondie	3:15	Atomic: The Very Best Of Blondie
22	Call Me	Blondie	3:33	Atomic: The Very Best Of Blondie
23	Dreaming	Blondie	3:06	Atomic: The Very Best Of Blondie
24	Hurricane	Bob Dylan	8:32	Desire
25	The Times They Are A-Changin'	Bob Dylan	3:17	Greatest Hits
26	Livin' On A Prayer	Bon Jovi	4:11	Cross Road
27	Beds Of Roses	Bon Jovi	6:35	Cross Road
28	Runaway	Bon Jovi	3:53	Cross Road
29	Rasputin (Extended Mix)	Boney M	5:50	Greatest Hits
30	Have You Ever Seen The Rain	Bonnie Tyler	4:10	Faster Than The Speed Of Night
31	Total Eclipse Of The Heart	Bonnie Tyler	7:02	Faster Than The Speed Of Night
32	Straight From The Heart	Bonnie Tyler	3:41	Faster Than The Speed Of Night
33	Holding Out For A Hero	Bonny Tyler	5:49	Meat Loaf And Friends
34	Dancing In The Dark	Bruce Springsteen	4:05	Born In The U.S.A.
35	Thunder Road	Bruce Springsteen	4:51	Born To Run
36	Born To Run	Bruce Springsteen	4:30	Born To Run
37	Jungleland	Bruce Springsteen	9:34	Born To Run

Sort by song name



	Name	Artist	Time	Album
1	Alive	Pearl Jam	5:41	Ten
2	All Over The World	Pixies	5:27	Bossanova
3	All Through The Night	Cyndi Lauper	4:30	She's So Unusual
4	Allison Road	Gin Blossoms	3:19	New Miserable Experience
5	Ama, Ama, Ama Y Ensancha El ...	Extremoduro	2:34	Deltoya (1992)
6	And We Danced	Hooters	3:50	Nervous Night
7	As I Lay Me Down	Sophie B. Hawkins	4:09	Whaler
8	Atomic	Blondie	3:50	Atomic: The Very Best Of Blondie
9	Automatic Lover	Jay-Jay Johanson	4:19	Antenna
10	Baba O'Riley	The Who	5:01	Who's Better, Who's Best
11	Beautiful Life	Ace Of Base	3:40	The Bridge
12	Beds Of Roses	Bon Jovi	6:35	Cross Road
13	Black	Pearl Jam	5:44	Ten
14	Bleed American	Jimmy Eat World	3:04	Bleed American
15	Borderline	Madonna	4:00	The Immaculate Collection
16	Born To Run	Bruce Springsteen	4:30	Born To Run
17	Both Sides Of The Story	Phil Collins	6:43	Both Sides
18	Bouncing Around The Room	Phish	4:09	A Live One (Disc 1)
19	Boys Don't Cry	The Cure	2:35	Staring At The Sea: The Singles 1979-1985
20	Brat	Green Day	1:43	Insomniac
21	Breakdown	Deerheart	3:40	Deerheart
22	Bring Me To Life (Kevin Roen Mix)	Evanescence Vs. Pa...	9:48	
23	Californication	Red Hot Chili Pepp...	1:40	
24	Call Me	Blondie	3:33	Atomic: The Very Best Of Blondie
25	Can't Get You Out Of My Head	Kylie Minogue	3:50	Fever
26	Celebration	Kool & The Gang	3:45	Time Life Music Sounds Of The Seventies - C

Natural order

Comparable interface: sort uses type's natural order.

```
public class Date implements Comparable<Date>
{
   private final int month, day, year;

   public Date(int m, int d, int y)
   {
      month = m;
      day   = d;
      year  = y;
   }
   …
   public int compareTo(Date that)
   {
      if (this.year  < that.year ) return -1;
      if (this.year  > that.year ) return +1;
      if (this.month < that.month) return -1;
      if (this.month > that.month) return +1;
      if (this.day   < that.day  ) return -1;
      if (this.day   > that.day  ) return +1;
      return 0;
   }
}
```

natural order

Comparable interface: sort uses type's natural order.

Problem 1. May want to use a non-natural order.

Problem 2. Desired data type may not come with a "natural" order.

Ex. Sort strings by:

- Natural order. `Now is the time`
- Case insensitive. `is Now the time`
- Spanish. `café cafetero cuarto churro nube ñoño`
- British phone book. `McKinley Mackintosh`

pre-1994 order for digraphs ch and ll and rr

```
String[] a;
...
Arrays.sort(a);
Arrays.sort(a, String.CASE_INSENSITIVE_ORDER);
Arrays.sort(a, Collator.getInstance(Locale.SPANISH));
```

import `java.text.Collator;`

Solution. Use Java's `Comparator` interface.

```
public interface Comparator<Key>
{
   public int compare(Key v, Key w);
}
```

Remark. The `compare()` method implements a total order like `compareTo()`.

Advantages. Decouples the definition of the data type from the definition of what it means to compare two objects of that type.

- Can add any number of new orders to a data type.
- Can add an order to a library data type with no natural order.

Comparator example

Reverse order. Sort an array of strings in reverse order.

```
public class ReverseOrder implements Comparator<String>
{
   public int compare(String a, String b)
   {
       return b.compareTo(a);
   }
}
```

comparator implementation

```
   ...
   Arrays.sort(a, new ReverseOrder());
   ...
```

client

Sort implementation with comparators

To support comparators in our sort implementations:

- Pass `Comparator` to `sort()` and `less()`.
- Use it in `less()`.

Ex. Insertion sort.

```
public static void sort(Object[] a, Comparator comparator)
{
   int N = a.length;
   for (int i = 0; i < N; i++)
      for (int j = i; j > 0 && less(comparator, a[j], a[j-1]); j--)
         exch(a, j, j-1);
}

private static boolean less(Comparator c, Object v, Object w)
{  return c.compare(v, w) < 0;   }

private static void exch(Object[] a, int i, int j)
{  Object swap = a[i]; a[i] = a[j]; a[j] = swap;  }
```

Generalized compare

Comparators enable multiple sorts of a single array (by different keys).

Ex. Sort students by name or by section.

```
Arrays.sort(students, Student.BY_NAME);
Arrays.sort(students, Student.BY_SECT);
```

sort by name

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	2	A	991-878-4944	308 Blair
Fox	1	A	884-232-5341	11 Dickinson
Furia	3	A	766-093-9873	101 Brown
Gazsi	4	B	665-303-0266	22 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	3	A	232-343-5555	343 Forbes

sort by section

Fox	1	A	884-232-5341	11 Dickinson
Chen	2	A	991-878-4944	308 Blair
Andrews	3	A	664-480-0023	097 Little
Furia	3	A	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	3	A	232-343-5555	343 Forbes
Battle	4	C	874-088-1212	121 Whitman
Gazsi	4	B	665-303-0266	22 Brown

Generalized compare

Ex. Enable sorting students by name or by section.

```
public class Student
{
   public static final Comparator<Student> BY_NAME = new ByName();
   public static final Comparator<Student> BY_SECT = new BySect();

   private final String name;
   private final int section;
   ...
   private static class ByName implements Comparator<Student>
   {
      public int compare(Student a, Student b)
      {  return a.name.compareTo(b.name);   }
   }

   private static class BySect implements Comparator<Student>
   {
      public int compare(Student a, Student b)
      {  return a.section - b.section;   }
   }
}
```

only use this trick if no danger of overflow

Generalized compare problem

A typical application. First, sort by name; then sort by section.

```
Arrays.sort(students, Student.BY_NAME);
```

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	2	A	991-878-4944	308 Blair
Fox	1	A	884-232-5341	11 Dickinson
Furia	3	A	766-093-9873	101 Brown
Gazsi	4	B	665-303-0266	22 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	3	A	232-343-5555	343 Forbes

```
Arrays.sort(students, Student.BY_SECT);
```

Fox	1	A	884-232-5341	11 Dickinson
Chen	2	A	991-878-4944	308 Blair
Kanaga	3	B	898-122-9643	22 Brown
Andrews	3	A	664-480-0023	097 Little
Furia	3	A	766-093-9873	101 Brown
Rohde	3	A	232-343-5555	343 Forbes
Battle	4	C	874-088-1212	121 Whitman
Gazsi	4	B	665-303-0266	22 Brown

@#%&@!!. Students in section 3 no longer in order by name.

A stable sort preserves the relative order of records with equal keys.

Sorting challenge 5

Q. Which sorts are stable?

Insertion sort? Selection sort? Shellsort? Mergesort?

sorted by time	sorted by location (not stable)	sorted by location (stable)
Chicago 09:00:00	Chicago 09:25:52	Chicago 09:00:00
Phoenix 09:00:03	Chicago 09:03:13	Chicago 09:00:59
Houston 09:00:13	Chicago 09:21:05	Chicago 09:03:13
Chicago 09:00:59	Chicago 09:19:46	Chicago 09:19:32
Houston 09:01:10	Chicago 09:19:32	Chicago 09:19:46
Chicago 09:03:13	Chicago 09:00:00	Chicago 09:21:05
Seattle 09:10:11	Chicago 09:35:21	Chicago 09:25:52
Seattle 09:10:25	Chicago 09:00:59	Chicago 09:35:21
Phoenix 09:14:25	Houston 09:01:10	Houston 09:00:13
Chicago 09:19:32	Houston 09:00:13	Houston 09:01:10
Chicago 09:19:46	Phoenix 09:37:44	Phoenix 09:00:03
Chicago 09:21:05	Phoenix 09:00:03	Phoenix 09:14:25
Seattle 09:22:43	Phoenix 09:14:25	Phoenix 09:37:44
Seattle 09:22:54	Seattle 09:10:25	Seattle 09:10:11
Chicago 09:25:52	Seattle 09:36:14	Seattle 09:10:25
Chicago 09:35:21	Seattle 09:22:43	Seattle 09:22:43
Seattle 09:36:14	Seattle 09:10:11	Seattle 09:22:54
Phoenix 09:37:44	Seattle 09:22:54	Seattle 09:36:14

no longer sorted by time

still sorted by time

Stability when sorting on a second key

Sorting challenge 5A

Q. Is insertion sort stable?

```
public class Insertion
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
         for (int j = i; j > 0 && less(a[j], a[j-1]); j--)
            exch(a, j, j-1);
   }
}
```

i	j	0	1	2	3	4
0	0	B_1	A_1	A_2	A_3	B_2
1	0	A_1	B_1	A_2	A_3	B_2
2	1	A_1	A_2	B_1	A_3	B_2
3	2	A_1	A_2	A_3	B_1	B_2
4	4	A_1	A_2	A_3	B_1	B_2
		A_1	A_2	A_3	B_1	B_2

A. Yes, equal elements never more past each other.

Sorting challenge 5B

Q. Is selection sort stable ?

```
public class Selection
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
      {
         int min = i;
         for (int j = i+1; j < N; j++)
            if (less(a[j], a[min]))
               min = j;
         exch(a, i, min);
      }
   }
}
```

i	min	0	1	2
0	2	B_1	B_2	A
1	1	A	B_2	B_1
2	2	A	B_2	B_1
		A	B_2	B_1

A. No, long-distance exchange might move left element to the right of some equal element.

Sorting challenge 5C

Q. Is shellsort stable?

```
public class Shell
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      int h = 1;
      while (h < N/3) h = 3*h + 1;
      while (h >= 1)
      {
         for (int i = h; i < N; i++)
         {
            for (int j = i; j > h && less(a[j], a[j-h]); j -= h)
               exch(a, j, j-h);
         }
         h = h/3;
      }
   }
}
```

h	0	1	2	3	4
	B_1	B_2	B_3	B_4	A_1
4	A_1	B_2	B_3	B_4	B_1
1	A_1	B_2	B_3	B_4	B_1
	A_1	B_2	B_3	B_4	B_1

A. No. Long-distance exchanges.

Sorting challenge 5D

Q. Is mergesort stable?

```
public class Merge
{
   private static Comparable[] aux;
   private static void merge(Comparable[] a, int lo, int mid, int hi)
   {  /* as before */  }

   private static void sort(Comparable[] a, int lo, int hi)
   {
      if (hi <= lo) return;
      int mid = lo + (hi - lo) / 2;
      sort(a, lo, mid);
      sort(a, mid+1, hi);
      merge(a, lo, mid, hi);
   }

   public static void sort(Comparable[] a)
   {
      aux = new Comparable[a.length];
      sort(a, 0, a.length - 1);
   }
}
```

Sorting challenge 5D

Q. Is mergesort stable?

	a[i]															
lo m hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 4, 5)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 6, 6, 7)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 8, 8, 9)	E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
merge(a, 10, 10, 11)	E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
merge(a, 12, 12, 13)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, 14, 14, 15)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
merge(a, 0, 1, 3)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 5, 7)	E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
merge(a, 8, 9, 11)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, 12, 13, 15)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, 0, 3, 7)	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
merge(a, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge(a, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Trace of merge results for bottom-up mergesort

A. Yes, if merge is stable.

Sorting challenge 5D (continued)

Q. Is merge stable?

```
private static void merge(Comparable[] a, int lo, int mid, int hi)
{
   for (int k = lo; k <= hi; k++)
      aux[k] = a[k];

   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)
   {
      if      (i > mid)              a[k] = aux[j++];
      else if (j > hi)               a[k] = aux[i++];
      else if (less(aux[j], aux[i])) a[k] = aux[j++];
      else                           a[k] = aux[i++];
   }
}
```

A. Yes, if implemented carefully (take from left subarray if equal).

Sorting challenge 5 (summary)

Q. Which sorts are stable ?

Yes. Insertion sort, mergesort.

No. Selection sort, shellsort.

Note. Need to carefully check code ("less than" vs "less than or equal").

Postscript: optimizing mergesort (a short history)

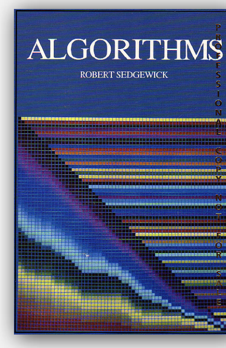
Goal. Remove instructions from the inner loop.

```
private static void merge(Comparable[] a, int lo, int mid, int hi)
{

   for (int k = lo; k <= hi; k++)
      aux[k] = a[k];

   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)
      if      (i > mid)                 a[k] = aux[j++];
      else if (j > hi )                 a[k] = aux[i++];
      else if (less(aux[j], aux[i]))    a[k] = aux[j++];
      else                              a[k] = aux[i++];

}
```

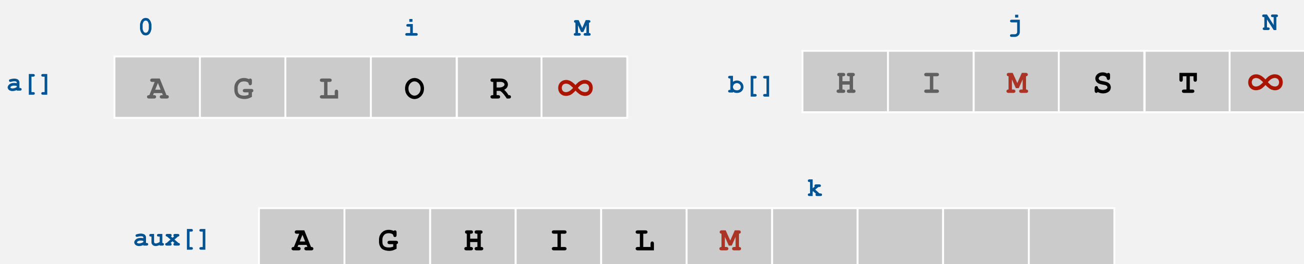
	lo			i	mid			j		hi
aux[]	A	G	L	O	R	H	I	M	S	T
							k			
a[]	A	G	H	I	L	M				

Postscript: optimizing mergesort (a short history)

Idea 1 (1960s). Use sentinels.




```
a[M] := maxint; b[N] := maxint;
for (int i = 0, j = 0, k = 0; k < M+1; k++)
   if (less(aux[j], aux[i])) aux[k] = a[i++];
                              aux[k] = b[j++];
```



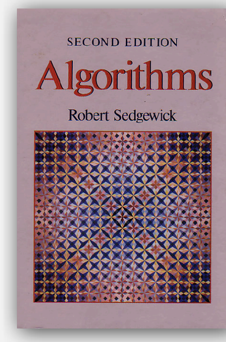
Problem 1. Still need copy.

Problem 2. No good place to put sentinels.

Problem 3. Complicates data-type interface (what is infinity for your type?)

Postscript: Optimizing mergesort (a short history)

Idea 2 (1980s). Reverse copy.



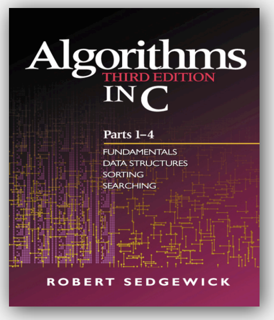

```
private static void merge(Comparable[] a, int lo, int mid, int hi)
{
   for (int i = lo; i <= mid; i++)
      aux[i] = a[i];                                    // copy

   for (int j = mid+1; j <= hi; j++)
      aux[j] = a[hi-j+mid+1];                           // reverse copy

   int i = lo, j = hi;
   for (int k = lo; k <= hi; k++)                       // merge
      if (less(aux[j], aux[i])) a[k] = aux[j--];
      else                      a[k] = aux[i++];
}
```

	lo			i	mid		j			hi
aux[]	A	G	L	O	R	T	S	M	I	H
							k			
a[]	A	G	H	I	L	M				

Problem. Copy still in inner loop.

Idea 3 (1990s). Eliminate copy with recursive argument switch.

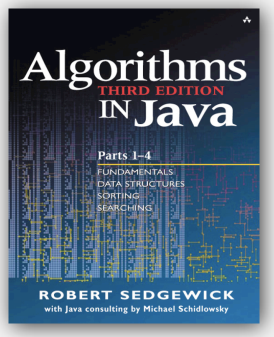
```
int mid = (lo+hi)/2;
mergesortABr(b, a, lo, mid);
mergesortABr(b, a, mid+1, r);
mergeAB(a, lo, b, lo, mid, b, mid+1, hi);
```



Problem. Complex interactions with reverse copy.

Solution. Go back to sentinels.

`Arrays.sort()`



Sorting challenge 6

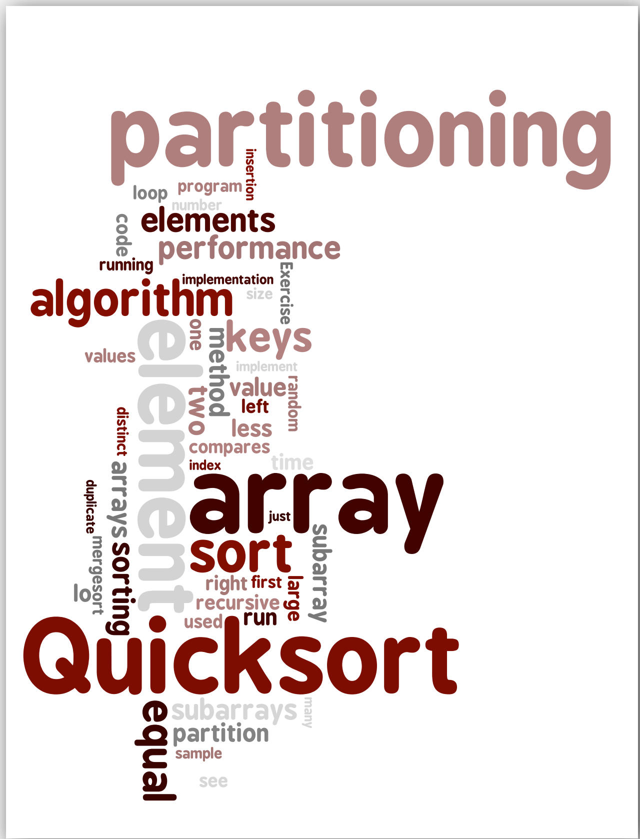
Problem. Choose mergesort for Algs 4th edition.

Recursive argument switch is out (recommended only for pros).

Q. Why not use reverse array copy?

```
private static void merge(Comparable[] a, int lo, int mid, int hi)
{
   for (int i = lo; i <= mid; i++)
      aux[i] = a[i];

   for (int j = mid+1; j <= hi; j++)
      aux[j] = a[hi-j+mid+1];

   int i = lo, j = hi;
   for (int k = lo; k <= hi; k++)
      if (less(aux[j], aux[i])) a[k] = aux[j--];
      else                      a[k] = aux[i++];
}
```

2.3 Quicksort



- quicksort
- selection
- duplicate keys
- system sorts

Two classic sorting algorithms

Critical components in the world's computational infrastructure.

- Full scientific understanding of their properties has enabled us to develop them into practical system sorts.
- Quicksort honored as one of top 10 algorithms of 20th century in science and engineering.

Mergesort. ← last lecture

- Java sort for objects.
- Perl, Python stable sort.

Quicksort. ← this lecture

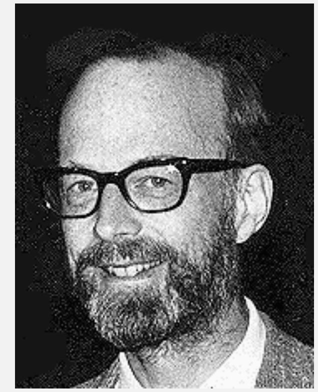
- Java sort for primitive types.
- C qsort, Unix, g++, Visual C++, Python.

- quicksort
- selection
- duplicate keys
- system sorts

Quicksort

Basic plan.

- Shuffle the array.
- Partition so that, for some `j`
  - element `a[j]` is in place
  - no larger element to the left of `j`
  - no smaller element to the right of `j`
- Sort each piece recursively.



Sir Charles Antony Richard Hoare
1980 Turing Award

input	Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
shuffle	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
partition	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
sort left	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
sort right	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
result	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

partitioning element

not greater

not less

Quicksort overview

Quicksort partitioning

Basic plan.

- Scan `i` from left for an item that belongs on the right.
- Scan `j` from right for item item that belongs on the left.
- Exchange `a[i]` and `a[j]`.
- Continue until pointers cross.

v a[i]

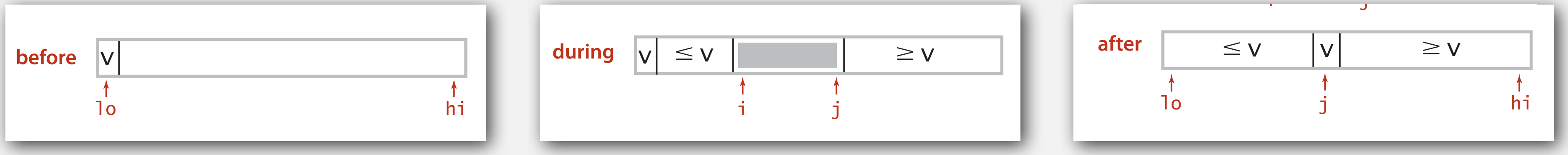
	i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values	-1	15	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
scan left, scan right	1	12	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
exchange	1	12	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
scan left, scan right	3	9	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
exchange	3	9	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
scan left, scan right	5	6	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
exchange	5	6	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
scan left, scan right	6	5	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
final exchange	0	5	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
result			E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S

Partitioning trace (array contents before and after each exchange)

Quicksort: Java code for partitioning

```
private static int partition(Comparable[] a, int lo, int hi)
{
   int i = lo, j = hi+1;
   while (true)
   {
      while (less(a[++i], a[lo]))
         if (i == hi) break;

      while (less(a[lo], a[--j]))
         if (j == lo) break;

      if (i >= j) break;
      exch(a, i, j);
   }

   exch(a, lo, j);
   return j;
}
```

- find item on left to swap
- find item on right to swap
- check if pointers cross
- swap
- swap with partitioning item
- return index of item now known to be in place



Quicksort: Java implementation

```
public class Quick
{
   private static int partition(Comparable[] a, int lo, int hi)
   {  /* see previous slide */  }

   public static void sort(Comparable[] a)
   {
      StdRandom.shuffle(a);
      sort(a, 0, a.length - 1);
   }

   private static void sort(Comparable[] a, int lo, int hi)
   {
      if (hi <= lo) return;
      int j = partition(a, lo, hi);
      sort(a, lo, j-1);
      sort(a, j+1, hi);
  }
}
```

shuffle needed for performance guarantee

Quicksort trace

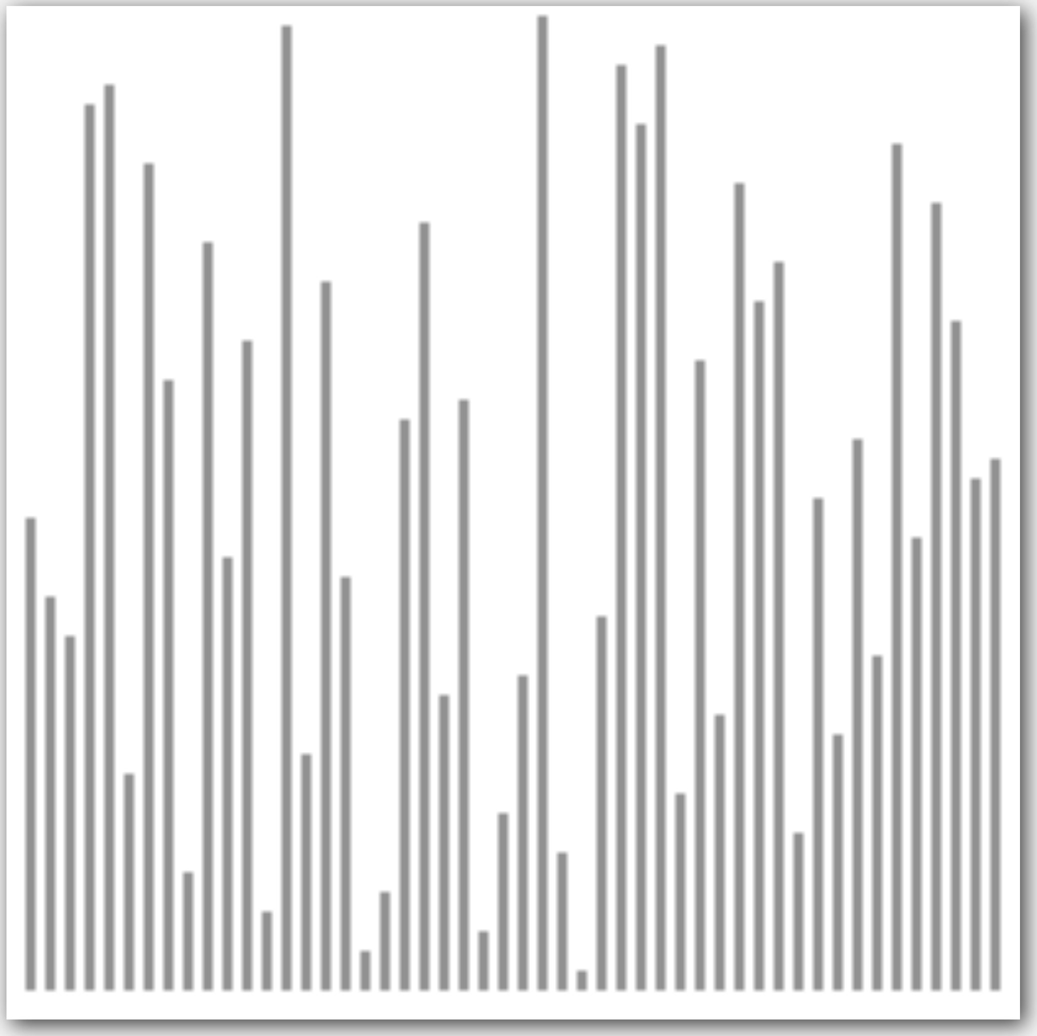
	lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values				Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
random shuffle				K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
	0	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
	0	3	4	E	C	A	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	2	2	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	0	1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	1		1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	4		4	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	6	6	15	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	7	9	15	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	7	7	8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	8		8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	10	13	15	A	C	E	E	I	K	L	M	O	P	S	Q	R	T	U	X
	10	12	12	A	C	E	E	I	K	L	M	O	P	R	Q	S	T	U	X
	10	11	11	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	10		10	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	14	14	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	15		15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
result				A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

no partition for subarrays of size 1

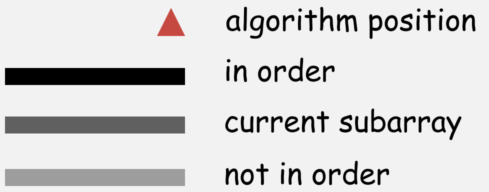
Quicksort trace (array contents after each partition)

Quicksort animation

50 random elements



http://www.sorting-algorithms.com/quick-sort



Quicksort: implementation details

Partitioning in-place. Using a spare array makes partitioning easier (and stable), but is not worth the cost.

Terminating the loop. Testing whether the pointers cross is a bit trickier than it might seem.

Staying in bounds. The `(j == lo)` test is redundant (why?), but the `(i == hi)` test is not.

Preserving randomness. Shuffling is needed for performance guarantee.

Equal keys. When duplicates are present, it is (counter-intuitively) best to stop on elements equal to the partitioning element.

Quicksort: empirical analysis

Running time estimates:

- Home pc executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (N^2)			mergesort (N log N)			quicksort (N log N)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.3 sec	6 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

Lesson 1. Good algorithms are better than supercomputers.

Lesson 2. Great algorithms are better than good ones.

Quicksort: best case analysis

Best case. Number of compares is ~ N lg N.

lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
									a[ ]								
			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
0	7	14	D	A	C	B	F	E	G	H	L	I	K	J	N	M	O
0	3	6	B	A	C	D	F	E	G	H	L	I	K	J	N	M	O
0	1	2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
0		0	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
2		2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
4	5	6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
4		4	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
6		6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
8	11	14	A	B	C	D	E	F	G	H	J	I	K	L	N	M	O
8	9	10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
8		8	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
10		10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
12	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12		12	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Quicksort: worst case analysis

Worst case. Number of compares is $\sim N^2 / 2$.

			a[ ]														
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0	0	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	1	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
2	2	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
3	3	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
4	4	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	5	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
6	6	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
7	7	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
8	8	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
9	9	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
10	10	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
11	11	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12	12	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
13	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Proposition I. The average number of compares C_N to quicksort an array of N elements is ~ 2N ln N (and the number of exchanges is ~ ⅓ N ln N).

Pf. C_N satisfies the recurrence $C_0 = C_1 = 0$ and for N ≥ 2:

$$C_N = \underbrace{(N+1)}_{\text{partitioning}} + \frac{\overbrace{C_0 + C_1 + \ldots + C_{N-1}}^{\text{left}}}{N} + \frac{\overbrace{C_{N-1} + C_{N-2} + \ldots + C_0}^{\text{right}}}{\underbrace{N}_{\text{partitioning probability}}}$$

- Multiply both sides by N and collect terms:

$$NC_N = N(N+1) + 2(C_0 + C_1 + \ldots + C_{N-1})$$

- Subtract this from the same equation for N-1:

$$NC_N - (N-1)C_{N-1} = 2N + 2C_{N-1}$$

- Rearrange terms and divide by N(N+1):

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}$$

Quicksort: average-case analysis

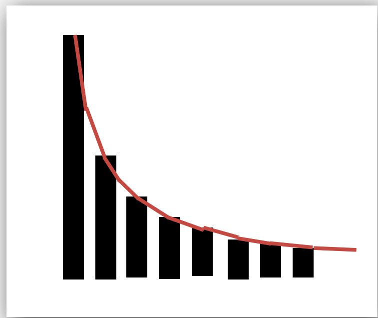
- Repeatedly apply above equation:

$$
\begin{aligned}
\frac{C_N}{N+1} &= \frac{C_{N-1}}{N} + \frac{2}{N+1} \\
&= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\
&= \frac{C_{N-3}}{N-2} + \frac{2}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\
&= \frac{2}{1} + \frac{2}{2} + \frac{2}{3} + \ldots + \frac{2}{N+1}
\end{aligned}
$$

previous equation

- Approximate sum by an integral:

$$
\begin{aligned}
C_N &\sim 2(N+1)\left(1+\frac{1}{2}+\frac{1}{3}+\ldots\frac{1}{N}\right) \\
&\sim 2(N+1)\int_1^N \frac{1}{x}dx
\end{aligned}
$$



- Finally, the desired result:

$$C_N \sim 2(N+1)\ln N \approx 1.39 N \lg N$$

Quicksort: summary of performance characteristics

Worst case. Number of compares is quadratic.

- $N + (N-1) + (N-2) + \ldots + 1 \sim N^2 / 2$.
- More likely that your computer is struck by lightning.

Average case. Number of compares is $\sim 1.39 N \lg N$.

- 39% more compares than mergesort.
- But faster than mergesort in practice because of less data movement.

Random shuffle.

- Probabilistic guarantee against worst case.
- Basis for math model that can be validated with experiments.

Caveat emptor. Many textbook implementations go quadratic if input:

- Is sorted or reverse sorted.
- Has many duplicates (even if randomized!) [stay tuned]

Quicksort: practical improvements

Median of sample.

- Best choice of pivot element = median.
- Estimate true median by taking median of sample.

Insertion sort small subarrays.

- Even quicksort has too much overhead for tiny subarrays.
- Can delay insertion sort until end.

Optimize parameters.

~ 12/7 N ln N compares
~ 12/35 N ln N exchanges

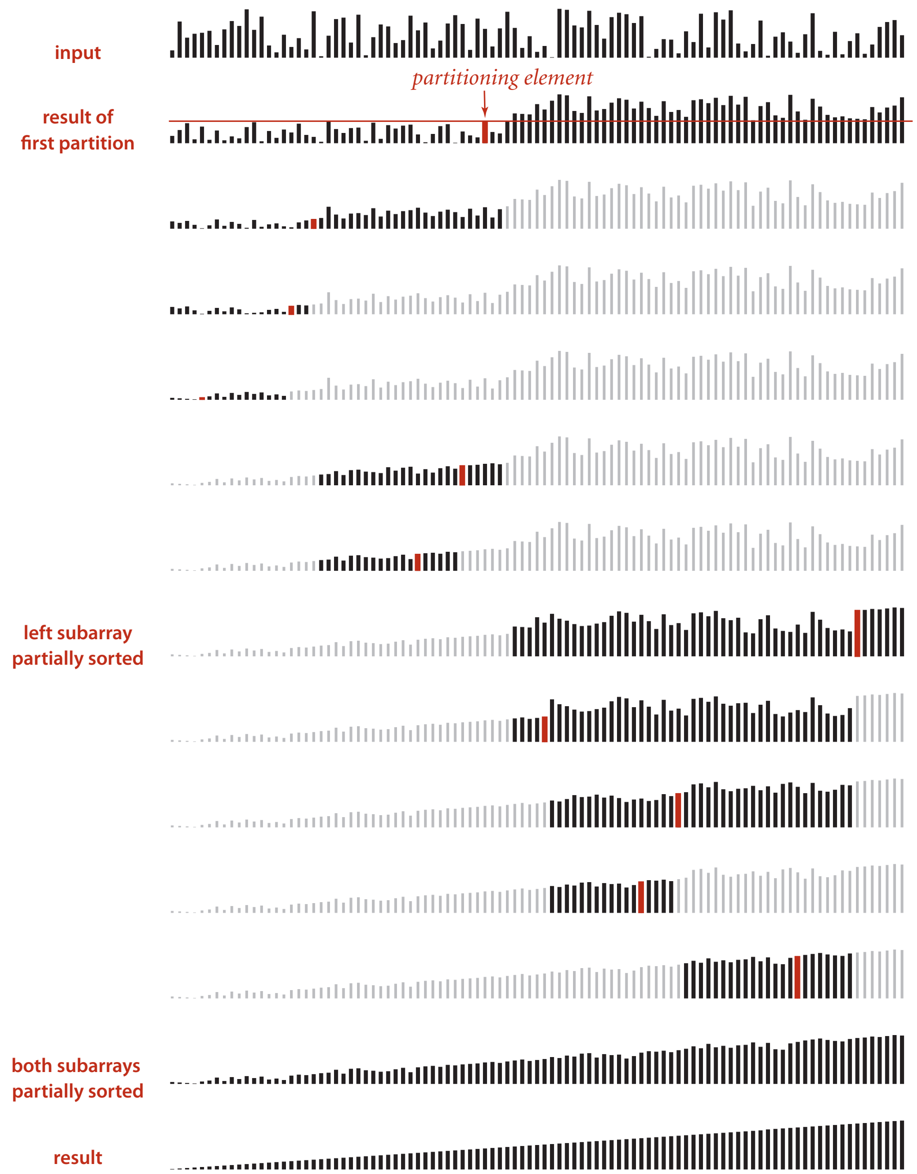
- Median-of-3 random elements.
- Cutoff to insertion sort for ≈ 10 elements.

Non-recursive version.

guarantees O(log N) stack size

- Use explicit stack.
- Always sort smaller half first.

Quicksort with cutoff to insertion sort: visualization



Quicksort with median-of-3 partitioning and cutoff for small subarrays

- quicksort
- **selection**
- duplicate keys
- system sorts

Selection

Goal. Find the k^{th} largest element.

Ex. Min ($k = 0$), max ($k = N-1$), median ($k = N/2$).

Applications.

- Order statistics.
- Find the "top k."

Use theory as a guide.

- Easy $O(N \log N)$ upper bound.
- Easy $O(N)$ upper bound for $k = 1, 2, 3$.
- Easy $\Omega(N)$ lower bound.

Which is true?

- $\Omega(N \log N)$ lower bound? ← is selection as hard as sorting?
- $O(N)$ upper bound? ← is there a linear-time algorithm for all k?

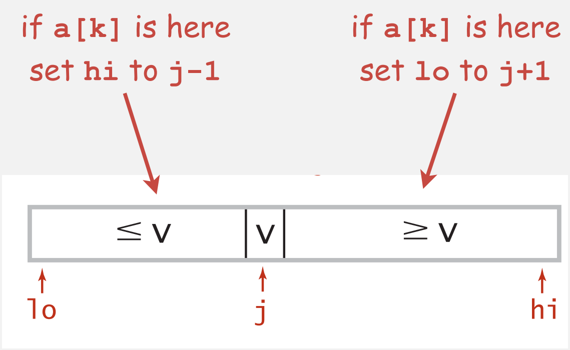
Quick-select

Partition array so that:

- Element `a[j]` is in place.
- No larger element to the left of `j`.
- No smaller element to the right of `j`.

Repeat in one subarray, depending on `j`; finished when `j` equals `k`.

```
public static Comparable select(Comparable[] a, int k)
{
    StdRandom.shuffle(a);
    int lo = 0, hi = a.length - 1;
    while (hi > lo)
    {
       int j = partition(a, lo, hi);
       if      (j < k) lo = j + 1;
       else if (j > k) hi = j - 1;
       else            return a[k];
    }
    return a[k];
}
```



Quick-select: mathematical analysis

Proposition. Quick-select takes linear time on average.

Pf sketch.

- Intuitively, each partitioning step roughly splits array in half: $N + N/2 + N/4 + \ldots + 1 \sim 2N$ compares.
- Formal analysis similar to quicksort analysis yields:

$$C_N = 2N + k \ln(N/k) + (N-k) \ln(N/(N-k))$$

Ex. $(2 + 2 \ln 2) N$ compares to find the median.

Remark. Quick-select uses $\sim N^2/2$ compares in worst case, but as with quicksort, the random shuffle provides a probabilistic guarantee.

Theoretical context for selection

Challenge. Design algorithm whose worst-case running time is linear.

Proposition. [Blum, Floyd, Pratt, Rivest, Tarjan, 1973] There exists a compare-based selection algorithm whose worst-case running time is linear.

Remark. But, algorithm is too complicated to be useful in practice.

Use theory as a guide.

- Still worthwhile to seek practical linear-time (worst-case) algorithm.
- Until one is discovered, use quick-select if you don't need a full sort.

Generic methods

In our `select()` implementation, client needs a cast.

```
Double[] a = new Double[N];
for (int i = 0; i < N; i++)
   a[i] = StdRandom.uniform();
Double median = (Double) Quick.select(a, N/2);
```

unsafe cast required

The compiler also complains.

```
% javac Quick.java
Note: Quick.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Q. How to fix?

Generic methods

Pedantic (safe) version. Compiles cleanly, no cast needed in client.

generic type variable
(value inferred from argument a[])

```
public class QuickPedantic
{
    public  static <Key extends Comparable<Key>> Key select(Key[] a, int k)
    {  /* as before */  }
```

return type matches array type

```
    public  static <Key extends Comparable<Key>> void sort(Key[] a)
    {  /* as before */  }

    private static <Key extends Comparable<Key>> int partition(Key[] a, int lo, int hi)
    {  /* as before */  }

    private static <Key extends Comparable<Key>> boolean less(Key v, Key w)
    {  /* as before */  }

    private static <Key extends Comparable<Key>> void exch(Key[] a, int i, int j)
    {  Key swap = a[i]; a[i] = a[j]; a[j] = swap;  }

}
```

can declare variables of generic type

http://www.cs.princeton.edu/algs4/35applications/QuickPedantic.java.html

Remark. Obnoxious code needed in system sort; not in this course (for brevity).

- quicksort
- selection
- **duplicate keys**
- system sorts

Duplicate keys

Often, purpose of sort is to bring records with duplicate keys together.

- Sort population by age.
- Find collinear points. ← see Assignment 3
- Remove duplicates from mailing list.
- Sort job applicants by college attended.

Typical characteristics of such applications.

- Huge array.
- Small number of key values.

```
Chicago 09:25:52
Chicago 09:03:13
Chicago 09:21:05
Chicago 09:19:46
Chicago 09:19:32
Chicago 09:00:00
Chicago 09:35:21
Chicago 09:00:59
Houston 09:01:10
Houston 09:00:13
Phoenix 09:37:44
Phoenix 09:00:03
Phoenix 09:14:25
Seattle 09:10:25
Seattle 09:36:14
Seattle 09:22:43
Seattle 09:10:11
Seattle 09:22:54
```

↑
key

Duplicate keys

Mergesort with duplicate keys. Always ~ N lg N compares.

Quicksort with duplicate keys.

- Algorithm goes quadratic unless partitioning stops on equal keys!
- 1990s C user found this defect in `qsort()`.

several textbook and system implementations
also have this defect

S T O P O N E Q U A L K E Y S

swap

if we don't stop
on equal keys

if we stop on
equal keys

Duplicate keys: the problem

Mistake. Put all keys equal to the partitioning element on one side.

Consequence. $\sim N^2 / 2$ compares when all keys equal.

`B A A B A B B B C C C`   `A A A A A A A A A A A`

Recommended. Stop scans on keys equal to the partitioning element.

Consequence. $\sim N \lg N$ compares when all keys equal.

`B A A B A B C C B C B`   `A A A A A A A A A A A`

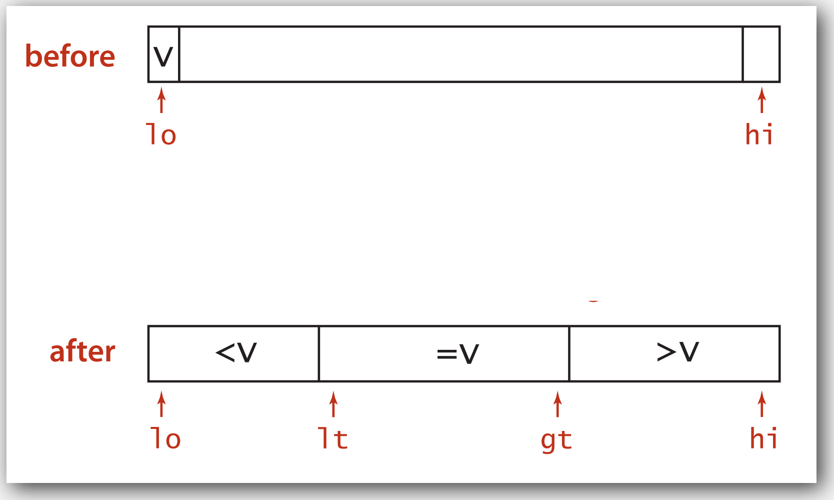
Desirable. Put all keys equal to the partitioning element in place.

`A A A B B B B B C C C`   `A A A A A A A A A A A`

3-way partitioning

Goal. Partition array into 3 parts so that:

- Elements between `lt` and `gt` equal to partition element `v`.
- No larger elements to left of `lt`.
- No smaller elements to right of `gt`.



Dutch national flag problem. [Edsger Dijkstra]

- Conventional wisdom until mid 1990s: not worth doing.
- New approach discovered when fixing mistake in C library `qsort()`.
- Now incorporated into `qsort()` and Java system sort.

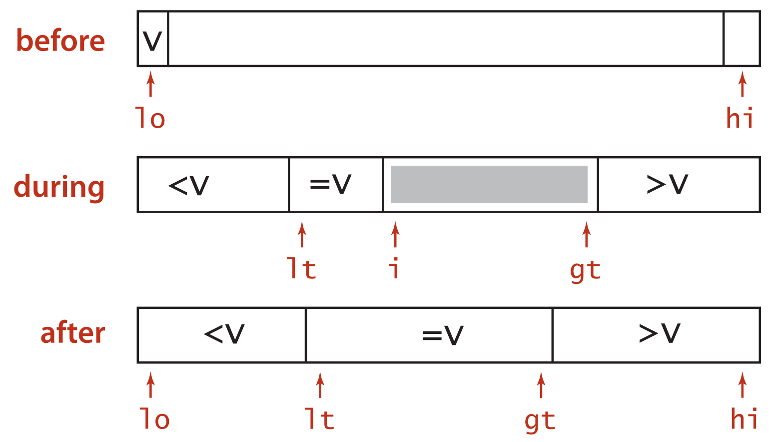
3-way partitioning: Dijkstra's solution

3-way partitioning.

- Let `v` be partitioning element `a[lo]`.
- Scan `i` from left to right.
  - `a[i]` less than `v`: exchange `a[lt]` with `a[i]` and increment both `lt` and `i`
  - `a[i]` greater than `v`: exchange `a[gt]` with `a[i]` and decrement `gt`
  - `a[i]` equal to `v`: increment `i`

All the right properties.

- In-place.
- Not much code.
- Small overhead if no equal keys.



3-way partitioning

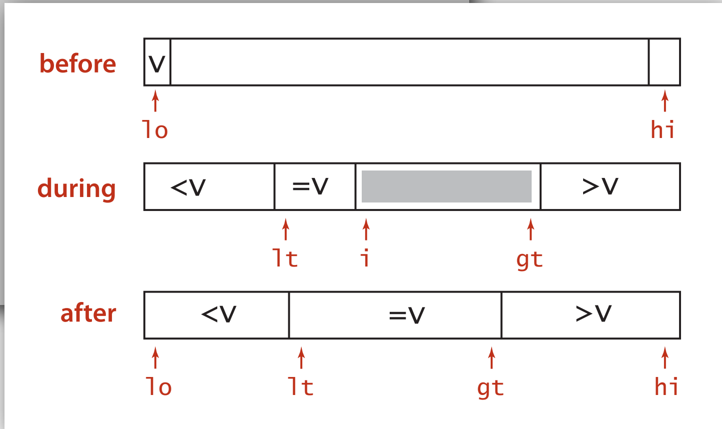
3-way partitioning: trace

lt	i	gt	0	1	2	3	4	5	6	7	8	9	10	11
0	0	11	R	B	W	W	R	W	B	R	R	W	B	R
0	1	11	R	B	W	W	R	W	B	R	R	W	B	R
1	2	11	B	R	W	W	R	W	B	R	R	W	B	R
1	2	10	B	R	R	W	R	W	B	R	R	W	B	W
1	3	10	B	R	R	W	R	W	B	R	R	W	B	W
1	3	9	B	R	R	B	R	W	B	R	R	W	W	W
2	4	9	B	B	R	R	R	W	B	R	R	W	W	W
2	5	9	B	B	R	R	R	W	B	R	R	W	W	W
2	5	8	B	B	R	R	R	W	B	R	R	W	W	W
2	5	7	B	B	R	R	R	R	B	R	W	W	W	W
2	6	7	B	B	R	R	R	R	B	R	W	W	W	W
3	7	7	B	B	B	R	R	R	R	R	W	W	W	W
3	8	7	B	B	B	R	R	R	R	R	W	W	W	W

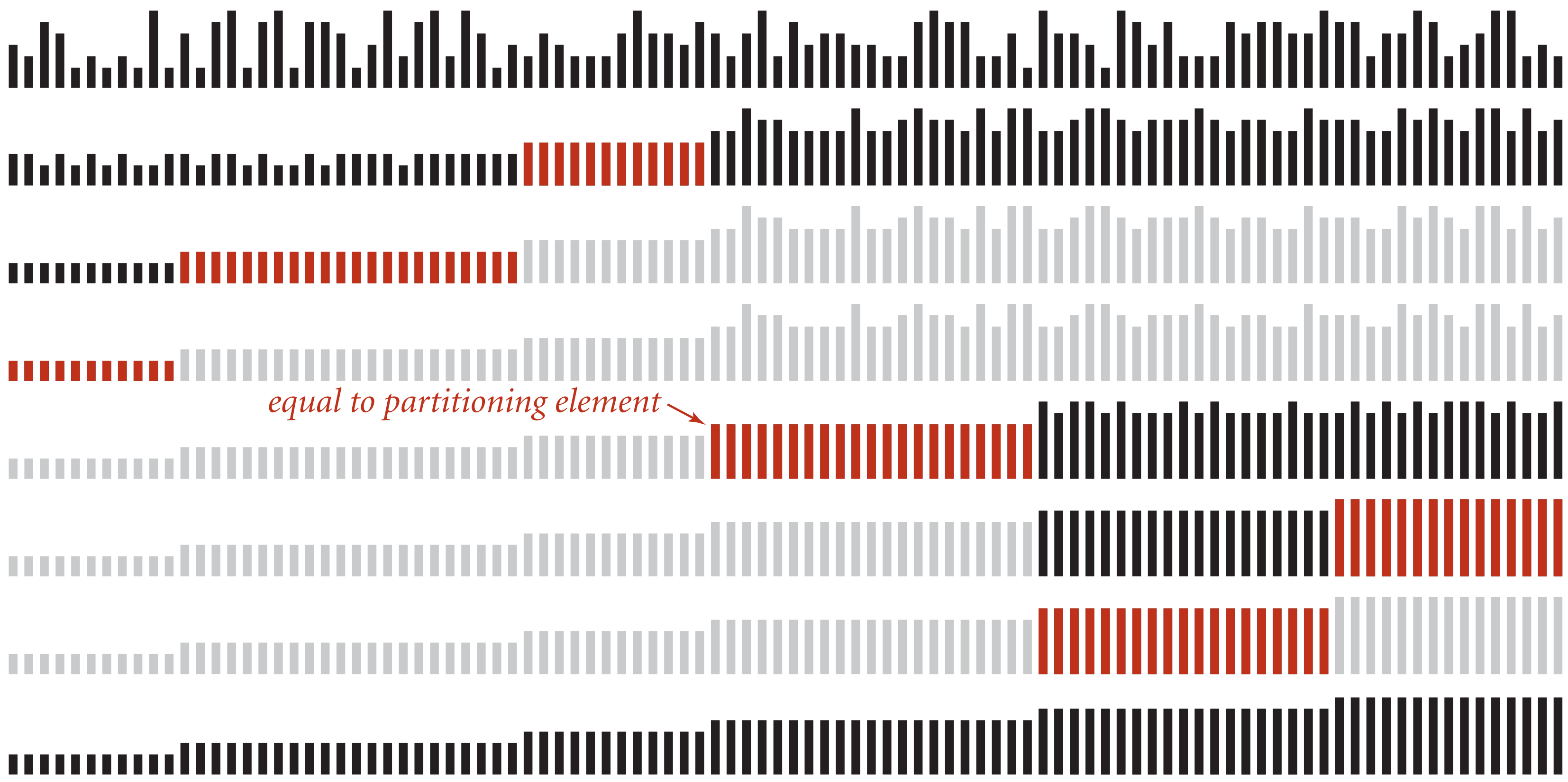
3-way partitioning trace (array contents after each loop iteration)

3-way quicksort: Java implementation

```
private static void sort(Comparable[] a, int lo, int hi)
{
   if (hi <= lo) return;
   int lt = lo, gt = hi;
   Comparable v = a[lo];
   int i = lo;
   while (i <= gt)
   {
      int cmp = a[i].compareTo(v);
      if      (cmp < 0) exch(a, lt++, i++);
      else if (cmp > 0) exch(a, i, gt--);
      else              i++;
   }

   sort(a, lo, lt - 1);
   sort(a, gt + 1, hi);
}
```



3-way quicksort: visual trace



Visual trace of quicksort with 3-way partitioning

Sorting lower bound. If there are n distinct keys and the i^{th} one occurs x_i times, any compare-based sorting algorithm must use at least

$$\lg\left(\frac{N!}{x_1!\; x_2!\; \cdots\; x_n!}\right) \;\sim\; -\sum_{i=1}^{n} x_i \lg \frac{x_i}{N}$$

N lg N when all distinct;
linear when only a constant number of distinct keys

compares in the worst case.

proportional to lower bound

Proposition. [Sedgewick-Bentley, 1997]
Quicksort with 3-way partitioning is entropy-optimal.
Pf. [beyond scope of course]

Bottom line. Randomized quicksort with 3-way partitioning reduces running time from linearithmic to linear in broad class of applications.

- selection
- duplicate keys
- comparators
- **system sorts**

Sorting applications

Sorting algorithms are essential in a broad variety of applications:

- Sort a list of names.
- Organize an MP3 library.
- Display Google PageRank results.
- List RSS news items in reverse chronological order.

obvious applications

- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.

problems become easy once items are in sorted order

- Data compression.
- Computer graphics.
- Computational biology.
- Supply chain management.
- Load balancing on a parallel computer.

. . .

non-obvious applications

Every system needs (and has) a system sort!

Java uses both mergesort and quicksort.

- `Arrays.sort()` sorts array of `Comparable` or any primitive type.
- Uses quicksort for primitive types; mergesort for objects.

```
import java.util.Arrays;

public class StringSort
{
   public static void main(String[] args)
   {
      String[] a = StdIn.readAll().split("\\s+");
      Arrays.sort(a);
      for (int i = 0; i < N; i++)
         StdOut.println(a[i]);
   }
}
```

Q. Why use different algorithms, depending on type?

Java system sort for primitive types

Engineering a sort function. [Bentley-McIlroy, 1993]

- Original motivation: improve `qsort()`.
- Basic algorithm = 3-way quicksort with cutoff to insertion sort.
- Partition on Tukey's ninther: median of the medians of 3 samples, each of 3 elements.

approximate median-of-9

nine evenly spaced elements	R	L	A	P	M	C	G	A	X	Z	K	R	B	R	J	J	E
groups of 3	R	A	M		G	X	K		B	J	E						
medians	M	K	E														
ninther	K																

Why use Tukey's ninther?

- Better partitioning than random shuffle.
- Less costly than random shuffle.

Achilles heel in Bentley-McIlroy implementation (Java system sort)

Based on all this research, Java's system sort is solid, right?

A killer input.

more disastrous consequences in C

- Blows function call stack in Java and crashes program.
- Would take quadratic time if it didn't crash first.

```
% more 250000.txt
0
218750
222662
11
166672
247070
83339
...
```

```
% java IntegerSort < 250000.txt
Exception in thread "main"
java.lang.StackOverflowError
   at java.util.Arrays.sort1(Arrays.java:562)
   at java.util.Arrays.sort1(Arrays.java:606)
   at java.util.Arrays.sort1(Arrays.java:608)
   at java.util.Arrays.sort1(Arrays.java:608)
   at java.util.Arrays.sort1(Arrays.java:608)
   ...
```

250,000 integers
between 0 and 250,000

Java's sorting library crashes, even if
you give it as much stack space as Windows allows

Achilles heel in Bentley-McIlroy implementation (Java system sort)

McIlroy's devious idea. [A Killer Adversary for Quicksort]

- Construct malicious input while running system quicksort, in response to elements compared.
- If `v` is partitioning element, commit to `(v < a[i])` and `(v < a[j])`, but don't commit to `(a[i] < a[j])` or `(a[j] > a[i])` until `a[i]` and `a[j]` are compared.

Consequences.

- Confirms theoretical possibility.
- Algorithmic complexity attack: you enter linear amount of data; server performs quadratic amount of work.

Remark. Attack is not effective if array is shuffled before sort.

Q. Why do you think system sort is deterministic?

System sort: Which algorithm to use?

Many sorting algorithms to choose from:

Internal sorts.

- Insertion sort, selection sort, bubblesort, shaker sort.
- Quicksort, mergesort, heapsort, samplesort, shellsort.
- Solitaire sort, red-black sort, splaysort, Dobosiewicz sort, psort, ...

External sorts. Poly-phase mergesort, cascade-merge, oscillating sort.

Radix sorts. Distribution, MSD, LSD, 3-way radix quicksort.

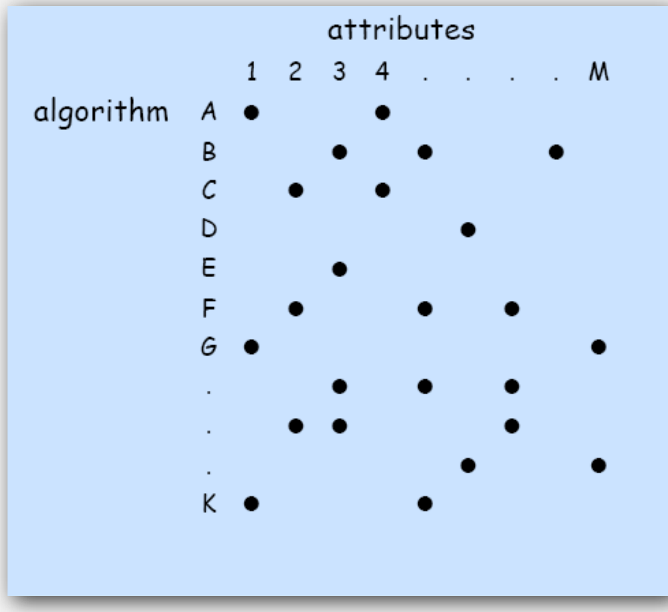
Parallel sorts.

- Bitonic sort, Batcher even-odd sort.
- Smooth sort, cube sort, column sort.
- GPUsort.

System sort: Which algorithm to use?

Applications have diverse attributes.

- Stable?
- Parallel?
- Deterministic?
- Keys all distinct?
- Multiple key types?
- Linked list or arrays?
- Large or small records?
- Is your array randomly ordered?
- Need guaranteed performance?



many more combinations of attributes than algorithms

Elementary sort may be method of choice for some combination.

Cannot cover all combinations of attributes.

Q. Is the system sort good enough?

A. Usually.

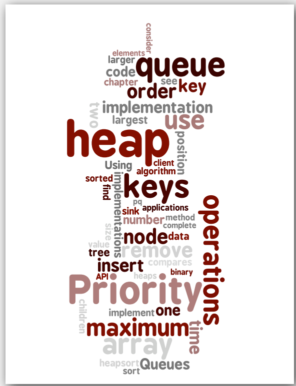
Sorting summary

	inplace?	stable?	worst	average	best	remarks
selection	x		$N^2 / 2$	$N^2 / 2$	$N^2 / 2$	N exchanges
insertion	x	x	$N^2 / 2$	$N^2 / 4$	N	use for small N or partially ordered
shell	x		?	?	N	tight code, subquadratic
quick	x		$N^2 / 2$	$2 N \ln N$	$N \lg N$	$N \log N$ probabilistic guarantee fastest in practice
3-way quick	x		$N^2 / 2$	$2 N \ln N$	N	improves quicksort in presence of duplicate keys
merge		x	$N \lg N$	$N \lg N$	$N \lg N$	$N \log N$ guarantee, stable
???	x	x	$N \lg N$	$N \lg N$	$N \lg N$	holy sorting grail

Which sorting algorithm?

lifo	find	data	data	data	data	hash	data
fifo	fifo	fifo	fifo	exch	fifo	fifo	exch
data	data	find	find	fifo	lifo	data	fifo
type	exch	hash	hash	find	type	link	find
hash	hash	heap	heap	hash	hash	leaf	hash
heap	heap	lifo	lifo	heap	heap	heap	heap
sort	less	link	link	leaf	link	exch	leaf
link	left	list	list	left	sort	node	left
list	leaf	push	push	less	find	lifo	less
push	lifo	root	root	lifo	list	left	lifo
find	push	sort	sort	link	push	find	link
root	root	type	type	list	root	path	list
leaf	list	leaf	leaf	sort	leaf	list	next
tree	tree	left	tree	tree	null	next	node
null	null	node	null	null	path	less	null
path	path	null	path	path	tree	root	path
node	node	path	node	node	exch	sink	push
left	link	tree	left	type	left	swim	root
less	sort	exch	less	root	less	null	sink
exch	type	less	exch	push	node	sort	sort
sink	sink	next	sink	sink	next	type	swap
swim	swim	sink	swim	swim	sink	tree	swim
next	next	swap	next	next	swap	push	tree
swap	swap	swim	swap	swap	swim	swap	type
original	?	?	?	?	?	?	sorted

2.4 Priority Queues



- API
- elementary implementations
- binary heaps
- heapsort
- event-based simulation

Algorithms in Java, 4th Edition · Robert Sedgewick and Kevin Wayne · Copyright © 2009 · January 22, 2010 4:15:59 PM

Priority queue API

data type	delete
stack	last in, first out
queue	first in, first out
priority queue	largest value out

```
public class MaxPQ<Key extends Comparable<Key>>
                 MaxPQ()          create a priority queue
                 MaxPQ(maxN)      create a priority queue of initial capacity maxN
          void   insert(Key v)    insert a key into the priority queue
           Key   max()            return the largest key
           Key   delMax()         return and remove the largest key
       boolean   isEmpty()        is the priority queue empty?
           int   size()           number of entries in the priority queue
```

API for a generic priority queue

operation	argument	return value
insert	P	
insert	Q	
insert	E	
remove max		Q
insert	X	
insert	A	
insert	M	
remove max		X
insert	P	
insert	L	
insert	E	
remove max		P

Priority queue applications

- Event-driven simulation. [customers in a line, colliding particles]
- Numerical computation. [reducing roundoff error]
- Data compression. [Huffman codes]
- Graph searching. [Dijkstra's algorithm, Prim's algorithm]
- Computational number theory. [sum of powers]
- Artificial intelligence. [A* search]
- Statistics. [maintain largest M values in a sequence]
- Operating systems. [load balancing, interrupt handling]
- Discrete optimization. [bin packing, scheduling]
- Spam filtering. [Bayesian spam filter]

Generalizes: stack, queue, randomized queue.

Priority queue client example

Problem. Find the largest M in a stream of N elements.
- Fraud detection: isolate $$ transactions.
- File maintenance: find biggest files or directories.

Constraint. Not enough memory to store N elements.
Solution. Use a min-oriented priority queue.

```
MinPQ<String> pq = new MinPQ<String>();

while(!StdIn.isEmpty())
{
   String s = StdIn.readString();
   pq.insert(s);
   if (pq.size() > M)
      pq.delMin();
}

while (!pq.isEmpty())
   System.out.println(pq.delMin());
```

implementation	time	space
sort	N log N	N
elementary PQ	M N	M
binary heap	N log M	M
best in theory	N	M

cost of finding the largest M in a stream of N items

- API
- **elementary implementations**
- binary heaps
- heapsort
- event-based simulation

Priority queue: unordered and ordered array implementation

operation	argument	return value	size	contents (unordered)	contents (ordered)
insert	P		1	P	P
insert	Q		2	P Q	P Q
insert	E		3	P Q E	E P Q
remove max		Q	2	P E	E P
insert	X		3	P E X	E P X
insert	A		4	P E X A	A E P X
insert	M		5	P E X A M	A E M P X
remove max		X	4	P E M A	A E M P
insert	P		5	P E M A P	A E M P P
insert	L		6	P E M A P L	A E L M P P
insert	E		7	P E M A P L E	A E E L M P P
remove max		P	6	E M A P L E	A E E L M P

A sequence of operations on a priority queue

Priority queue: unordered array implementation

```
public class UnorderedMaxPQ<Key extends Comparable<Key>>
{
   private Key[] pq;   // pq[i] = ith element on pq
   private int N;      // number of elements on pq

   public UnorderedMaxPQ(int capacity)
   {  pq = (Key[]) new Comparable[capacity];  }

   public boolean isEmpty()
   {  return N == 0; }

   public void insert(Key x)
   {  pq[N++] = x;  }

   public Key delMax()
   {
      int max = 0;
      for (int i = 1; i < N; i++)
         if (less(max, i)) max = i;
      exch(max, N-1);
      return pq[--N];
   }
}
```

no generic array creation

`less()` and `exch()` as for sorting

Priority queue elementary implementations

Challenge. Implement all operations efficiently.

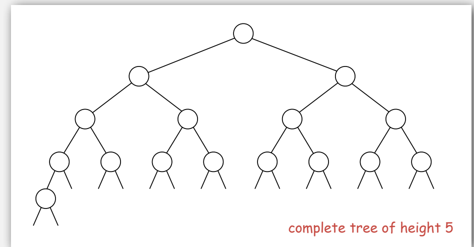
implementation	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1
goal	log N	log N	log N

order-of-growth running time for PQ with N items

Binary tree

Binary tree. Empty or node with links to left and right binary trees.

Complete tree. Perfectly balanced, except for bottom level.



N = 16
⌊lg N⌋ = 4
height = 5

Property. Height of complete tree with N nodes is 1 + ⌊lg N⌋.
Pf. Height only increases when N is exactly a power of 2.

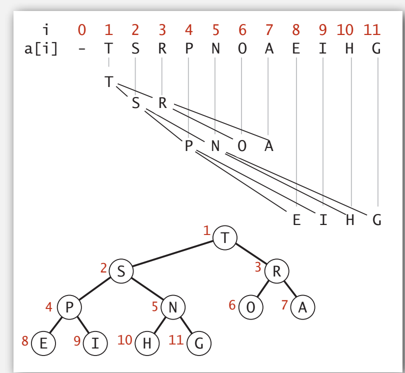
A complete binary tree in nature

Binary heap

Binary heap. Array representation of a heap-ordered complete binary tree.

Heap-ordered binary tree.
- Keys in nodes.
- No smaller than children's keys.

Array representation.
- Take nodes in level order.
- No explicit links needed!



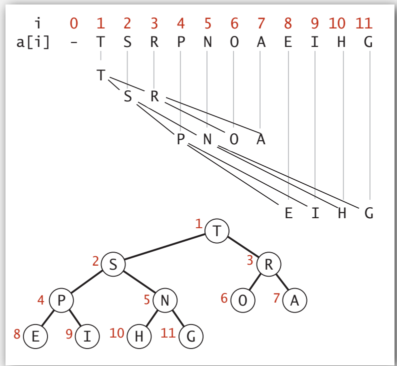
Binary heap properties

Property A. Largest key is `a[1]`, which is root of binary tree.

indices start at 1

Property B. Can use array indices to move through tree.

- Parent of node at `k` is at `k/2`.
- Children of node at `k` are at `2k` and `2k+1`.



Promotion in a heap

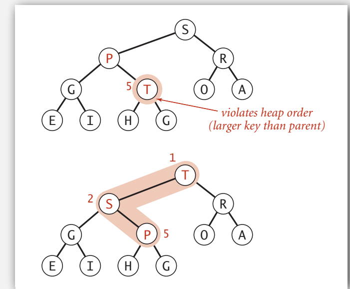
Scenario. Node's key becomes larger key than its parent's key.

To eliminate the violation:

- Exchange key in node with key in parent.
- Repeat until heap order restored.

```
private void swim(int k)
{
   while (k > 1 && less(k/2, k))
   {
      exch(k, k/2);
      k = k/2;
   }
}
```

parent of node at k is at k/2



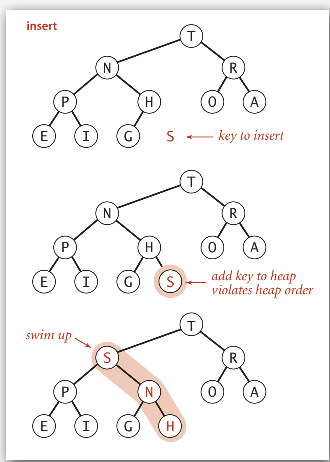
Peter principle. Node promoted to level of incompetence.

Insertion in a heap

Insert. Add node at end, then swim it up.

Running time. At most ~ lg N compares.

```
public void insert(Key x)
{
   pq[++N] = x;
   swim(N);
}
```



Demotion in a heap

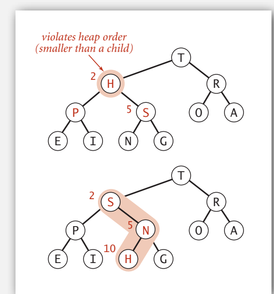
Scenario. Node's key becomes smaller than one (or both) of its children's keys.

To eliminate the violation:

- Exchange key in node with key in larger child.
- Repeat until heap order restored.

```
private void sink(int k)
{
   while (2*k <= N)
   {
      int j = 2*k;
      if (j < N && less(j, j+1)) j++;
      if (!less(k, j)) break;
      exch(k, j);
      k = j;
   }
}
```

children of node
at k are 2k and 2k+1



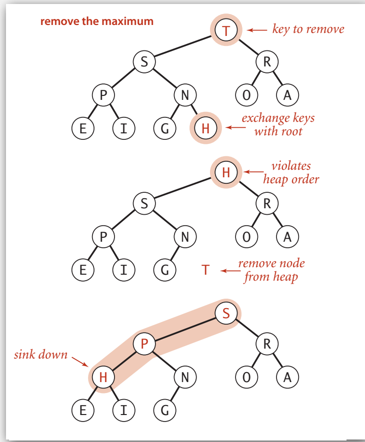
Power struggle. Better subordinate promoted.

Delete the maximum in a heap

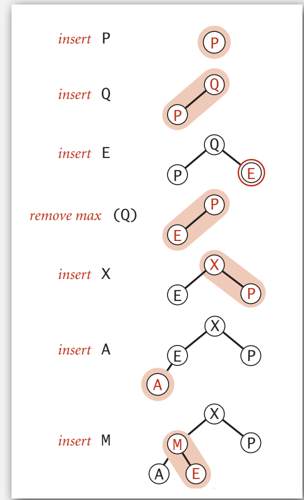
Delete max. Exchange root with node at end, then sink it down.

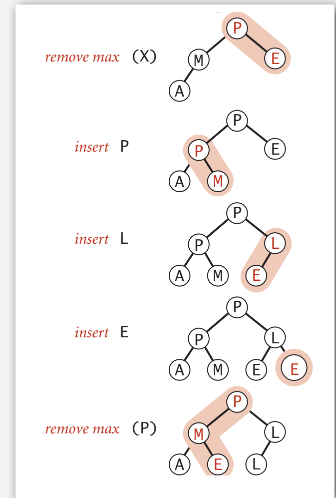
Running time. At most ~ 2 lg N compares.

```
public Key delMax()
{
   Key max = pq[1];
   exch(1, N--);
   sink(1);
   pq[N+1] = null;      ← prevent loitering
   return max;
}
```



Heap operations





Binary heap: Java implementation

```
public class MaxPQ<Key extends Comparable<Key>>
{
   private Key[] pq;
   private int N;

   public MaxPQ(int capacity)
   {  pq = (Key[]) new Comparable[capacity+1];  }

   public boolean isEmpty()
   {   return N == 0;   }
   public void insert(Key key)                              ← PQ ops
   {   /* see previous code */  }
   public Key delMax()
   {   /* see previous code */  }

   private void swim(int k)
   {   /* see previous code */  }                            ← heap helper functions
   private void sink(int k)
   {   /* see previous code */  }

   private boolean less(int i, int j)
   {   return pq[i].compareTo(pq[j]) < 0;  }                  ← array helper functions
   private void exch(int i, int j)
   {   Key t = pq[i]; pq[i] = pq[j]; pq[j] = t;  }
}
```


Priority queues implementation cost summary

implementation	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1
binary heap	log N	log N	1

order-of-growth running time for PQ with N items

Hopeless challenge. Make all operations constant time.

Q. Why hopeless?

Binary heap considerations

Minimum-oriented priority queue.
- Replace `less()` with `greater()`.
- Implement `greater()`.

Dynamic array resizing.
- Add no-arg constructor.
- Apply repeated doubling and shrinking. ← leads to O(log N) amortized time per op

Immutability of keys.
- Assumption: client does not change keys while they're on the PQ.
- Best practice: use immutable keys.

Other operations.
- Remove an arbitrary item.
- Change the priority of an item.

easy to implement with `sink()` and `swim()` [stay tuned]

- API
- elementary implementations
- binary heaps
- **heapsort**
- event-based simulation

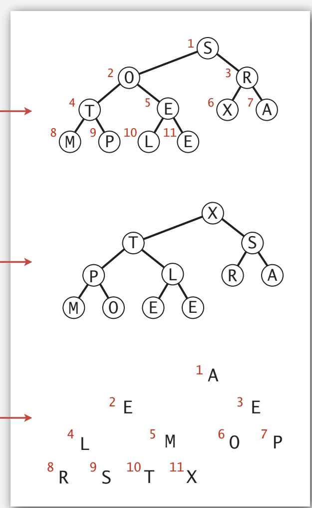
Heapsort

Basic plan for in-place sort.
- Create max-heap with all N keys.
- Repeatedly remove the maximum key.

start with array of keys in arbitrary order →

build a max-heap (in place) →

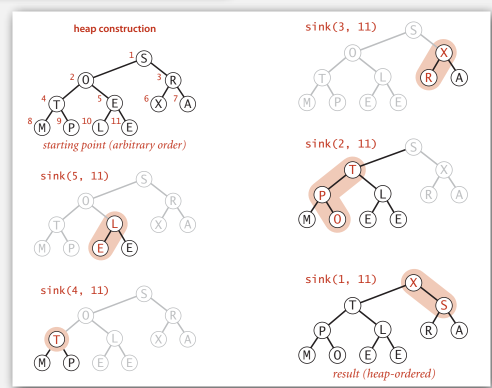
sorted result (in place) →



Heapsort: heap construction

First pass. Build heap using bottom-up method.

```
for (int k = N/2; k >= 1; k--)
   sink(a, k, N);
```

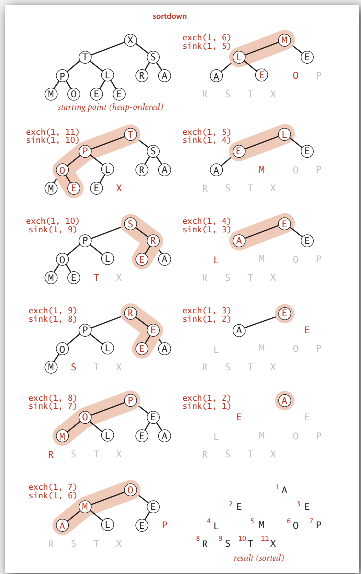


Heapsort: sortdown

Second pass.

- Remove the maximum, one at a time.
- Leave in array, instead of nulling out.

```
while (N > 1)
{
   exch(a, 1, N--);
   sink(a, 1, N);
}
```



Heapsort: Java implementation

```
public class Heap
{
   public static void sort(Comparable[] pq)
   {
      int N = pq.length;
      for (int k = N/2; k >= 1; k--)
         sink(pq, k, N);
      while (N > 1)
      {
         exch(pq, 1, N);
         sink(pq, 1, --N);
      }
   }

   private static void sink(Comparable[] pq, int k, int N)
   {  /* as before */  }

   private static boolean less(Comparable[] pq, int i, int j)
   {  /* as before */  }

   private static void exch(Comparable[] pq, int i, int j)
   {  /* as before */  }

}
```

but use 1-based indexing

Heapsort: trace

N	k	a[i] 0	1	2	3	4	5	6	7	8	9	10	11
initial values			S	O	R	T	E	X	A	M	P	L	E
11	5		S	O	R	T	L	X	A	M	P	E	E
11	4		S	O	R	T	L	X	A	M	P	E	E
11	3		S	O	X	T	L	R	A	M	P	E	E
11	2		S	T	X	P	L	R	A	M	O	E	E
11	1		X	T	S	P	L	R	A	M	O	E	E
heap-ordered			X	T	S	P	L	R	A	M	O	E	E
10	1		T	P	S	O	L	R	A	M	E	E	X
9	1		S	P	R	O	L	E	A	M	E	T	X
8	1		R	P	E	O	L	E	A	M	S	T	X
7	1		P	O	E	M	L	E	A	R	S	T	X
6	1		O	M	E	A	L	E	P	R	S	T	X
5	1		M	L	E	A	E	O	P	R	S	T	X
4	1		L	E	E	A	M	O	P	R	S	T	X
3	1		E	A	E	L	M	O	P	R	S	T	X
2	1		E	A	E	L	M	O	P	R	S	T	X
1	1		A	E	E	L	M	O	P	R	S	T	X
sorted result			A	E	E	L	M	O	P	R	S	T	X

Heapsort trace (array contents just after each sink)

Heapsort: mathematical analysis

Proposition Q. At most 2 N lg N compares and exchanges.

Significance. Sort in N log N worst-case without using extra memory.

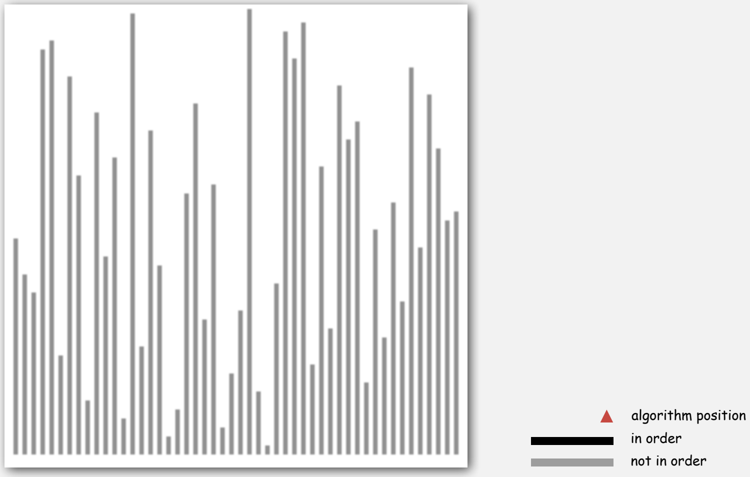
- Mergesort: no, linear extra space. ← in-place merge possible, not practical
- Quicksort: no, quadratic time in worst case. ← N log N worst-case quicksort possible, not practical
- Heapsort: yes!

Bottom line. Heapsort is optimal for both time and space, but:

- Inner loop longer than quicksort's.
- Makes poor use of cache memory.
- Not stable.

Heapsort animation

50 random elements



http://www.sorting-algorithms.com/heap-sort

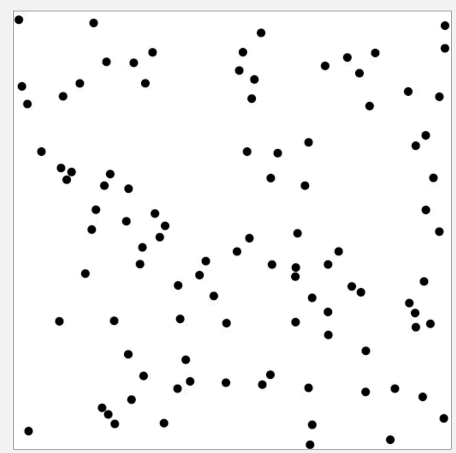
Sorting algorithms: summary

	inplace?	stable?	worst	average	best	remarks
selection	x		$N^2 / 2$	$N^2 / 2$	$N^2 / 2$	N exchanges
insertion	x	x	$N^2 / 2$	$N^2 / 4$	N	use for small N or partially ordered
shell	x		?	?	N	tight code, subquadratic
quick	x		$N^2 / 2$	$2 N \ln N$	$N \lg N$	$N \log N$ probabilistic guarantee fastest in practice
3-way quick	x		$N^2 / 2$	$2 N \ln N$	N	improves quicksort in presence of duplicate keys
merge		x	$N \lg N$	$N \lg N$	$N \lg N$	$N \log N$ guarantee, stable
heap	x		$2 N \lg N$	$2 N \lg N$	$N \lg N$	$N \log N$ guarantee, in-place
???	x	x	$N \lg N$	$N \lg N$	$N \lg N$	holy sorting grail

- API
- elementary implementations
- binary heaps
- heapsort
- **event-based simulation**

Molecular dynamics simulation of hard discs

Goal. Simulate the motion of N moving particles that behave according to the laws of elastic collision.



Molecular dynamics simulation of hard discs

Goal. Simulate the motion of N moving particles that behave according to the laws of elastic collision.

Hard disc model.

- Moving particles interact via elastic collisions with each other and walls.
- Each particle is a disc with known position, velocity, mass, and radius.
- No other forces.

temperature, pressure, diffusion constant

motion of individual atoms and molecules

Significance. Relates macroscopic observables to microscopic dynamics.

- Maxwell-Boltzmann: distribution of speeds as a function of temperature.
- Einstein: explain Brownian motion of pollen grains.

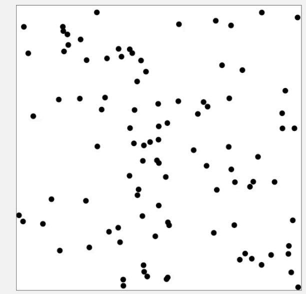
Warmup: bouncing balls

Time-driven simulation. N bouncing balls in the unit square.

```
public class BouncingBalls
{
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      Ball balls[] = new Ball[N];
      for (int i = 0; i < N; i++)
         balls[i] = new Ball();
      while(true)
      {
         StdDraw.clear();
         for (int i = 0; i < N; i++)
         {
            balls[i].move(0.5);
            balls[i].draw();
         }
         StdDraw.show(50);
      }
   }
}
```

main simulation loop

% java BouncingBalls 100



Warmup: bouncing balls

```
public class Ball
{
    private double rx, ry;          // position
    private double vx, vy;          // velocity
    private final double radius;    // radius
    public Ball()
    {  /* initialize position and velocity */  }

    public void move(double dt)
    {
        if ((rx + vx*dt < radius) || (rx + vx*dt > 1.0 - radius)) { vx = -vx; }
        if ((ry + vy*dt < radius) || (ry + vy*dt > 1.0 - radius)) { vy = -vy; }
        rx = rx + vx*dt;
        ry = ry + vy*dt;
    }
    public void draw()
    {  StdDraw.filledCircle(rx, ry, radius);  }
}
```

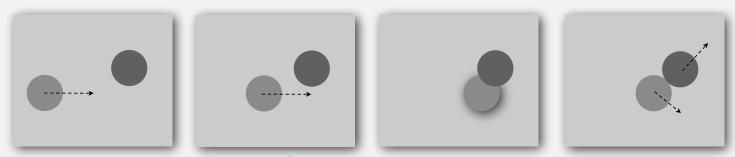
check for collision with walls

Missing. Check for balls colliding with each other.

- Physics problems: when? what effect?
- CS problems: which object does the check? too many checks?

Time-driven simulation

- Discretize time in quanta of size dt.
- Update the position of each particle after every dt units of time, and check for overlaps.
- If overlap, roll back the clock to the time of the collision, update the velocities of the colliding particles, and continue the simulation.

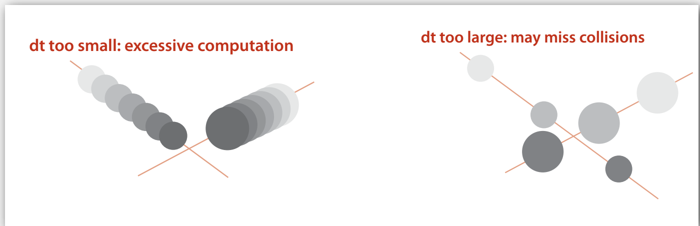


t t + dt t + 2 dt (collision detected) t + Δt (roll back clock)

Time-driven simulation

Main drawbacks.

- ~ $N^2/2$ overlap checks per time quantum.
- Simulation is too slow if dt is very small.
- May miss collisions if dt is too large.
  (if colliding particles fail to overlap when we are looking)



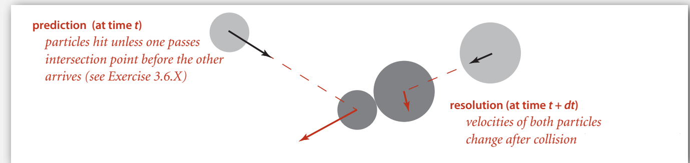
Event-driven simulation

Change state only when something happens.

- Between collisions, particles move in straight-line trajectories.
- Focus only on times when collisions occur.
- Maintain PQ of collision events, prioritized by time.
- Remove the min = get next collision.

Collision prediction. Given position, velocity, and radius of a particle, when will it collide next with a wall or another particle?

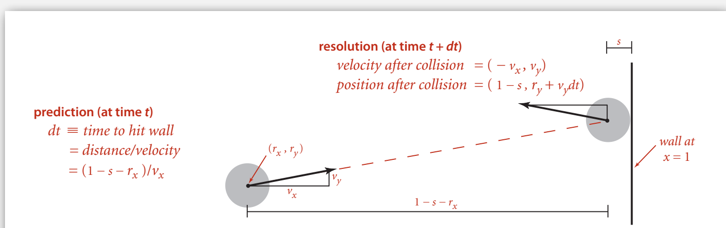
Collision resolution. If collision occurs, update colliding particle(s) according to laws of elastic collisions.



Particle-wall collision

Collision prediction and resolution.

- Particle of radius s at position (rx, ry).
- Particle moving in unit box with velocity (vx, vy).
- Will it collide with a vertical wall? If so, when?

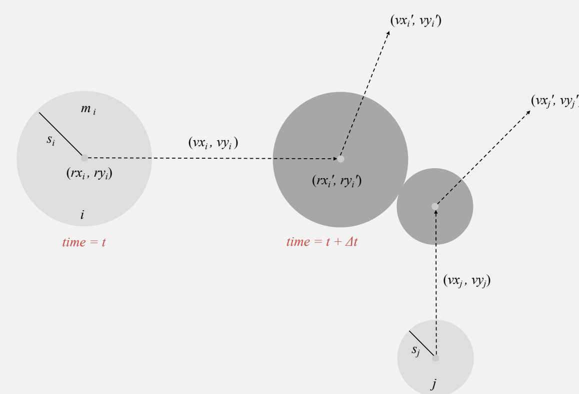


Predicting and resolving a particle-wall collision

Particle-particle collision prediction

Collision prediction.

- Particle i: radius s_i, position (rx_i, ry_i), velocity (vx_i, vy_i).
- Particle j: radius s_j, position (rx_j, ry_j), velocity (vx_j, vy_j).
- Will particles i and j collide? If so, when?



Particle-particle collision prediction

Collision prediction.

- Particle i: radius s_i, position (rx_i, ry_i), velocity (vx_i, vy_i).
- Particle j: radius s_j, position (rx_j, ry_j), velocity (vx_j, vy_j).
- Will particles i and j collide? If so, when?

$$\Delta t = \begin{cases} \infty & \text{if } \Delta v \cdot \Delta r \geq 0 \\ \infty & \text{if } d < 0 \\ -\dfrac{\Delta v \cdot \Delta r + \sqrt{d}}{\Delta v \cdot \Delta v} & \text{otherwise} \end{cases}$$

$$d = (\Delta v \cdot \Delta r)^2 - (\Delta v \cdot \Delta v)\,(\Delta r \cdot \Delta r - \sigma^2) \qquad \sigma = \sigma_i + \sigma_j$$

$$\Delta v = (\Delta vx,\ \Delta vy) = (vx_i - vx_j,\ vy_i - vy_j)$$
$$\Delta r = (\Delta rx,\ \Delta ry) = (rx_i - rx_j,\ ry_i - ry_j)$$

$$\Delta v \cdot \Delta v = (\Delta vx)^2 + (\Delta vy)^2$$
$$\Delta r \cdot \Delta r = (\Delta rx)^2 + (\Delta ry)^2$$
$$\Delta v \cdot \Delta r = (\Delta vx)(\Delta rx) + (\Delta vy)(\Delta ry)$$

Important note: This is high-school physics, so we won't be testing you on it!

Particle-particle collision resolution

Collision resolution. When two particles collide, how does velocity change?

$$\begin{aligned} vx_i' &= vx_i + Jx / m_i \\ vy_i' &= vy_i + Jy / m_i \\ vx_j' &= vx_j - Jx / m_j \\ vy_j' &= vy_j - Jy / m_j \end{aligned}$$

Newton's second law (momentum form)

$$Jx = \frac{J\,\Delta rx}{\sigma},\quad Jy = \frac{J\,\Delta ry}{\sigma},\quad J = \frac{2\, m_i\, m_j\, (\Delta v \cdot \Delta r)}{\sigma(m_i + m_j)}$$

impulse due to normal force
(conservation of energy, conservation of momentum)

Important note: This is high-school physics, so we won't be testing you on it!

Particle data type skeleton

```
public class Particle
{
    private double rx, ry;         // position
    private double vx, vy;         // velocity
    private final double radius;   // radius
    private final double mass;     // mass
    private int count;             // number of collisions

    public Particle(...) { }

    public void move(double dt)  { }
    public void draw()           { }

    public double timeToHit(Particle that)   { }
    public double timeToHitVerticalWall()    { }
    public double timeToHitHorizontalWall()  { }

    public void bounceOff(Particle that)     { }
    public void bounceOffVerticalWall()      { }
    public void bounceOffHorizontalWall()    { }

}
```

predict collision with particle or wall

resolve collision with particle or wall

Particle-particle collision and resolution implementation

```
public double timeToHit(Particle that)
{
   if (this == that) return INFINITY;
   double dx  = that.rx - this.rx, dy  = that.ry - this.ry;
   double dvx = that.vx - this.vx; dvy = that.vy - this.vy;
   double dvdr = dx*dvx + dy*dvy;
   if( dvdr > 0) return INFINITY;
   double dvdv = dvx*dvx + dvy*dvy;
   double drdr = dx*dx + dy*dy;
   double sigma = this.radius + that.radius;
   double d = (dvdr*dvdr) - dvdv * (drdr - sigma*sigma);
   if (d < 0) return INFINITY;
   return -(dvdr + Math.sqrt(d)) / dvdv;
}
```

no collision

```
public void bounceOff(Particle that)
{
   double dx  = that.rx - this.rx, dy  = that.ry - this.ry;
   double dvx = that.vx - this.vx, dvy = that.vy - this.vy;
   double dvdr = dx*dvx + dy*dvy;
   double dist = this.radius + that.radius;
   double J = 2 * this.mass * that.mass * dvdr / ((this.mass + that.mass) * dist);
   double Jx = J * dx / dist;
   double Jy = J * dy / dist;
   this.vx += Jx / this.mass;
   this.vy += Jy / this.mass;
   that.vx -= Jx / that.mass;
   that.vy -= Jy / that.mass;
   this.count++;
   that.count++;
}
```

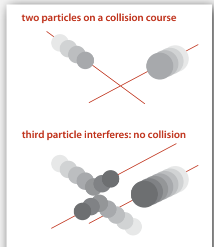
Important note: This is high-school physics, so we won't be testing you on it!

Collision system: event-driven simulation main loop

Initialization.

- Fill PQ with all potential particle-wall collisions.
- Fill PQ with all potential particle-particle collisions.

"potential" since collision may not happen if some other collision intervenes



An invalidated event

Main loop.

- Delete the impending event from PQ (min priority = t).
- If the event has been invalidated, ignore it.
- Advance all particles to time t, on a straight-line trajectory.
- Update the velocities of the colliding particle(s).
- Predict future particle-wall and particle-particle collisions involving the colliding particle(s) and insert events onto PQ.

Event data type

Conventions.

- Neither particle `null` ⇒ particle-particle collision.
- One particle `null` ⇒ particle-wall collision.
- Both particles `null` ⇒ redraw event.

```
private class Event implements Comparable<Event>
{
   private double time;         // time of event
   private Particle a, b;       // particles involved in event
   private int countA, countB;  // collision counts for a and b

   public Event(double t, Particle a, Particle b) { }

   public int compareTo(Event that)
   {  return this.time - that.time;   }

   public boolean isValid()
   {   }
}
```

create event

ordered by time

invalid if intervening collision

Collision system implementation: skeleton

```
public class CollisionSystem
{
   private MinPQ<Event> pq;        // the priority queue
   private double t  = 0.0;        // simulation clock time
   private Particle[] particles;   // the array of particles

   public CollisionSystem(Particle[] particles) { }

   private void predict(Particle a)
   {
      if (a == null) return;
      for (int i = 0; i < N; i++)
      {
         double dt = a.timeToHit(particles[i]);
         pq.insert(new Event(t + dt, a, particles[i]));
      }
      pq.insert(new Event(t + a.timeToHitVerticalWall()  , a, null));
      pq.insert(new Event(t + a.timeToHitHorizontalWall(), null, a));
 }

   private void redraw()  { }

   public void simulate() {  /* see next slide */  }
}
```

add to PQ all particle-wall and particle-particle collisions involving this particle

Collision system implementation: main event-driven simulation loop

```
public void simulate()
{
   pq = new MinPQ<Event>();
   for(int i = 0; i < N; i++) predict(particles[i]);
   pq.insert(new Event(0, null, null));

   while(!pq.isEmpty())
   {
      Event event = pq.delMin();
      if(!event.isValid()) continue;
      Particle a = event.a;
      Particle b = event.b;

      for(int i = 0; i < N; i++)
         particles[i].move(event.time - t);
      t = event.time;

      if      (a != null && b != null) a.bounceOff(b);
      else if (a != null && b == null) a.bounceOffVerticalWall()
      else if (a == null && b != null) b.bounceOffHorizontalWall();
      else if (a == null && b == null) redraw();

      predict(a);
      predict(b);
   }
}
```

initialize PQ with collision events and redraw event

get next event

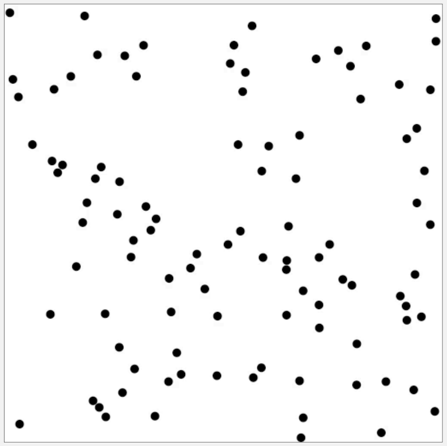
update positions and time

process event

predict new events based on changes

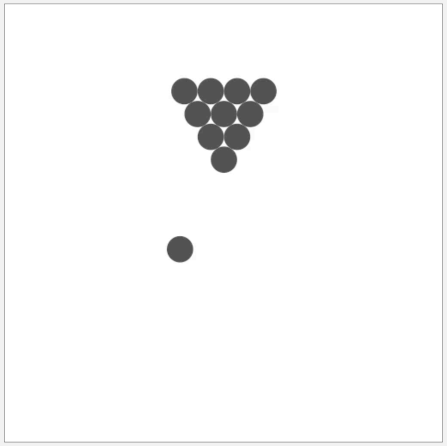
Simulation example 1

% java CollisionSystem 100



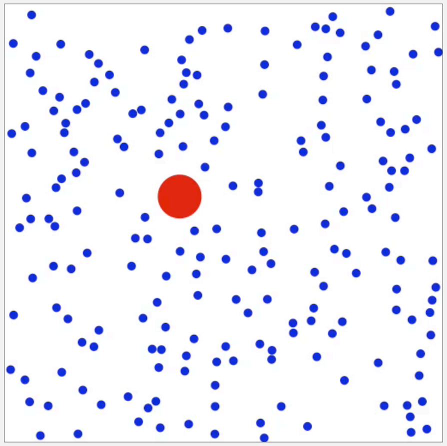
Simulation example 2

% java CollisionSystem < billiards.txt



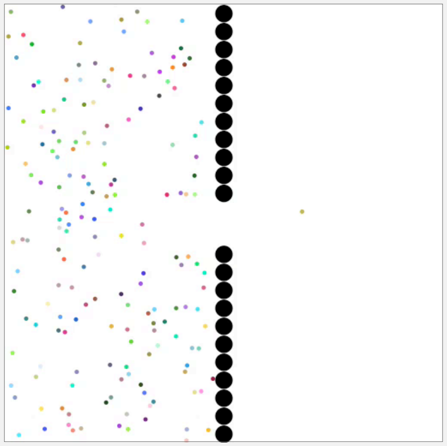
Simulation example 3

% java CollisionSystem < brownian.txt



Simulation example 4

% java CollisionSystem < diffusion.txt



3.1 Symbol Tables

- API
- sequential search
- binary search
- ordered operations

Symbol tables

Key-value pair abstraction.

- Insert a value with specified key.
- Given a key, search for the corresponding value.

Ex. DNS lookup.

- Insert URL with specified IP address.
- Given URL, find corresponding IP address.

URL	IP address
`www.cs.princeton.edu`	`128.112.136.11`
`www.princeton.edu`	`128.112.128.15`
`www.yale.edu`	`130.132.143.21`
`www.harvard.edu`	`128.103.060.55`
`www.simpsons.com`	`209.052.165.60`

key value

Symbol table applications

application	purpose of search	key	value
dictionary	find definition	word	definition
book index	find relevant pages	term	list of page numbers
file share	find song to download	name of song	computer ID
financial account	process transactions	account number	transaction details
web search	find relevant web pages	keyword	list of page names
compiler	find properties of variables	variable name	type and value
routing table	route Internet packets	destination	best route
DNS	find IP address given URL	URL	IP address
reverse DNS	find URL given IP address	IP address	URL
genomics	find markers	DNA string	known positions
file system	find file on disk	filename	location on disk

Symbol table API

Associative array abstraction. Associate one value with each key.

public class ST<Key, Value>

	ST()	*create a symbol table*	
void	put(Key key, Value val)	*put key-value pair into the table (remove* key *from table if value is null)*	a[key] = val;
Value	get(Key key)	*value paired with* key *(*null *if* key *is absent)*	a[key]
void	delete(Key key)	*remove* key *(and its value) from table*	
boolean	contains(Key key)	*is there a value paired with* key*?*	
boolean	isEmpty()	*is the table empty?*	
int	size()	*number of key-value pairs in the table*	
Iterable<Key>	keys()	*all the keys in the table*	

API for a generic basic symbol table

Conventions

- Values are not `null`.
- Method `get()` returns `null` if key not present.
- Method `put()` overwrites old value with new value.

Intended consequences.

- Easy to implement `contains()`.

```
public boolean contains(Key key)
{  return get(key) != null;  }
```

- Can implement lazy version of `delete()`.

```
public void delete(Key key)
{  put(key, null);           }
```

Keys and values

Value type. Any generic type.

Key type: several natural assumptions.

- Assume keys are `Comparable`, use `compareTo()`.
- Assume keys are any generic type, use `equals()` to test equality.
- Assume keys are any generic type, use `equals()` to test equality and `hashCode()` to scramble key.

Best practices. Use immutable types for symbol table keys.

- Immutable in Java: `String`, `Integer`, `Double`, `File`, …
- Mutable in Java: `Date`, `StringBuilder`, `Url`, …

ST test client for traces

Build ST by associating value i with ith string from standard input.

```
public static void main(String[] args)
{
   ST<String, Integer> st = new ST<String, Integer>();
   String[] a = StdIn.readAll().split("\\s+");
   for (int i = 0; i < a.length; i++)
      st.put(a[i], i);
   for (String s : st.keys())
      StdOut.println(s + " " + st.get(s));
}
```

keys	S	E	A	R	C	H	E	X	A	M	P	L	E
values	0	1	2	3	4	5	6	7	8	9	10	11	12

output

A	8
C	4
E	12
H	5
L	9
M	11
P	10
R	3
S	0
X	7

ST test client for analysis

Frequency counter. Read a sequence of strings from standard input and print out one that occurs with highest frequency.

```
% more tinyTale.txt
it was the best of times
it was the worst of times
it was the age of wisdom
it was the age of foolishness
it was the epoch of belief
it was the epoch of incredulity
it was the season of light
it was the season of darkness
it was the spring of hope
it was the winter of despair

% java FrequencyCounter 1 < tinyTale.txt
it 10

% java FrequencyCounter 8 < tale.txt
business 122

% java FrequencyCounter 10 < leipzig1M.txt
government 24763
```

tiny example (60 words, 20 distinct)

real example (135,635 words, 10,769 distinct)

real example (21,191,455 words, 534,580 distinct)

Frequency counter implementation

```
public class FrequencyCounter
{
   public static void main(String[] args)
   {
      int minlen = Integer.parseInt(args[0]);
      ST<String, Integer> st = new ST<String, Integer>();
      while (!StdIn.isEmpty())
      {
         String word = StdIn.readString();
         if (word.length() < minlen) continue;
         if (!st.contains(word)) st.put(word, 1);
         else                    st.put(word, st.get(word) + 1);
      }
      String max = "";
      st.put(max, 0);
      for (String word : st.keys())
         if (st.get(word) > st.get(max))
            max = word;
      StdOut.println(max + " " + st.get(max));
   }
}
```

create ST

ignore short strings

read string and update frequency

print a string with max freq

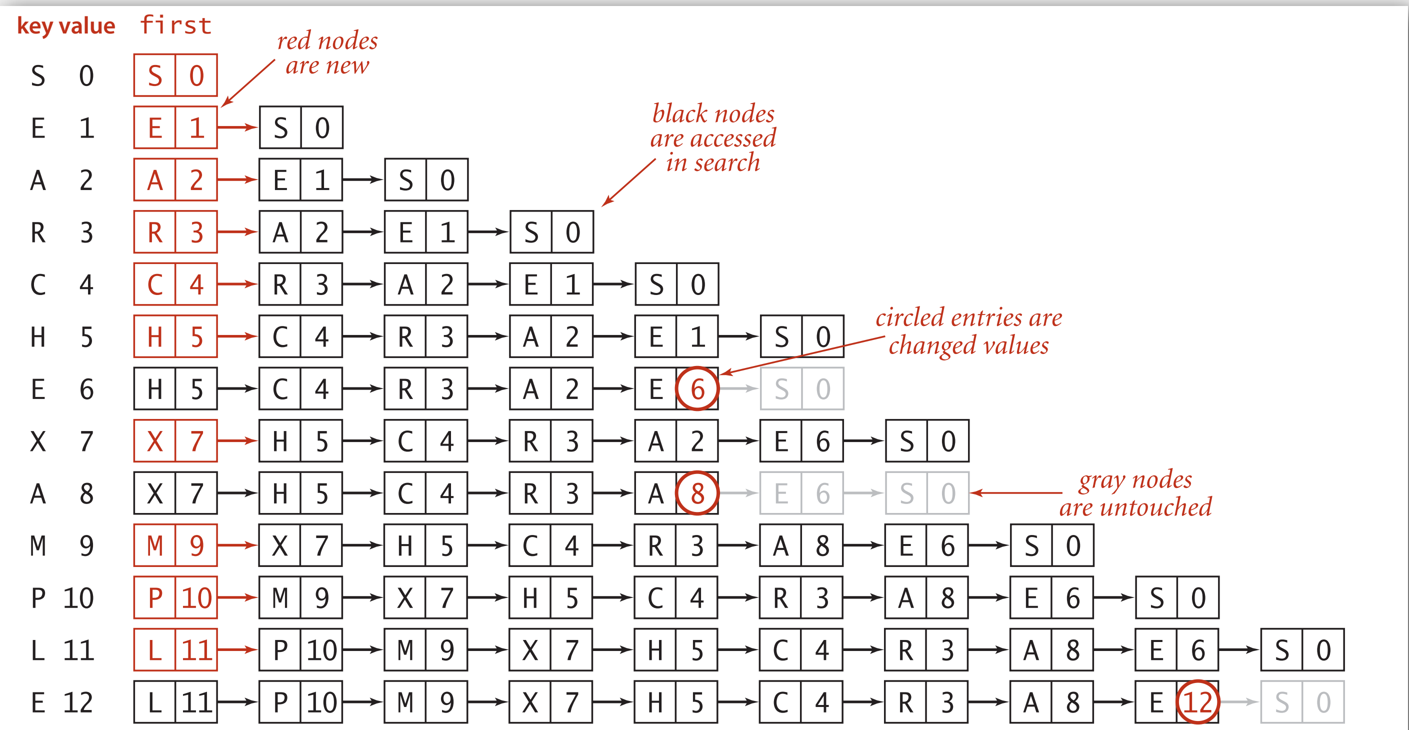
- API
- **sequential search**
- binary search
- ordered operations

Sequential search in a linked list

Data structure. Maintain an (unordered) linked list of key-value pairs.

Search. Scan through all keys until find a match.

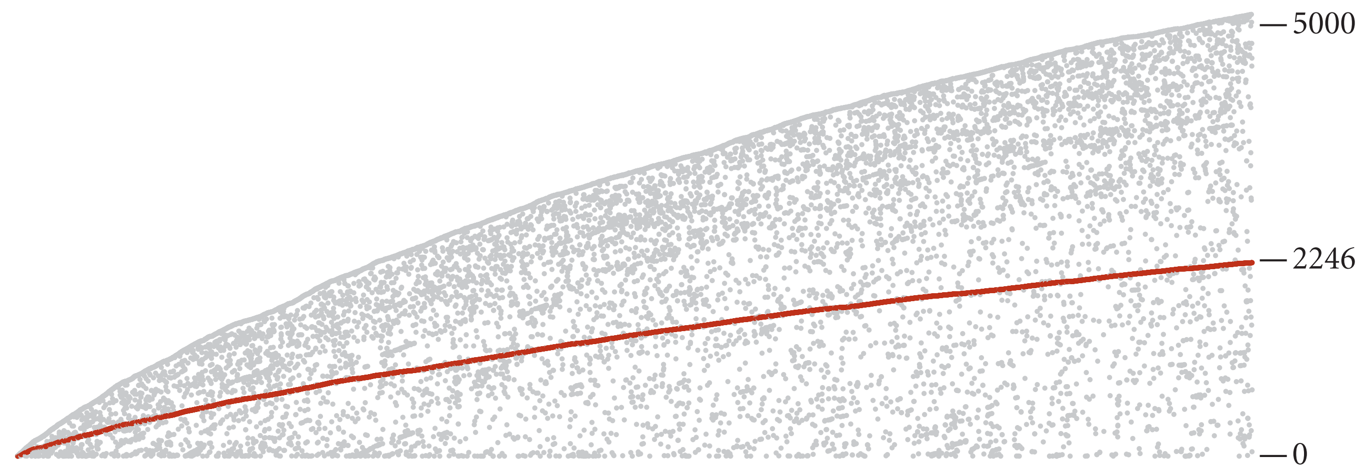
Insert. Scan through all keys until find a match; if no match add to front.



Trace of linked-list ST implementation for standard indexing client

Elementary ST implementations: summary

ST implementation	worst case		average case		ordered iteration?	operations on keys
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	$N / 2$	N	no	`equals()`



Costs for `java FrequencyCounter 8 < tale.txt` **using** `LinkedListST`

Challenge. Efficient implementations of both search and insert.

- API
- sequential search
- **binary search**
- ordered symbol table ops

Binary search

Data structure. Maintain an ordered array of key-value pairs.

Rank helper function. How many keys < k?

keys[]

successful search for P

lo	hi	m	0	1	2	3	4	5	6	7	8	9
0	9	4	A	C	E	H	L	M	P	R	S	X
5	9	7	A	C	E	H	L	M	P	R	S	X
5	6	5	A	C	E	H	L	M	P	R	S	X
6	6	6	A	C	E	H	L	M	P	R	S	X

entries in black are `a[lo..hi]`

entry in red is `a[m]`

loop exits with `keys[m] = P:` *return* 6

unsuccessful search for Q

lo	hi	m	0	1	2	3	4	5	6	7	8	9
0	9	4	A	C	E	H	L	M	P	R	S	X
5	9	7	A	C	E	H	L	M	P	R	S	X
5	6	5	A	C	E	H	L	M	P	R	S	X
7	6	6	A	C	E	H	L	M	P	R	S	X

loop exits with `lo > hi:` *return* 7

Trace of binary search for rank in an ordered array

Binary search: Java implementation

```
public Value get(Key key)
{
   if (isEmpty()) return null;
   int i = rank(key);
   if (i < N && keys[i].compareTo(key) == 0) return vals[i];
   else  return null;
}
```

```
private int rank(Key key)
{
   int lo = 0, hi = N-1;
   while (lo <= hi)
   {
       int mid = lo + (hi - lo) / 2;
       int cmp = key.compareTo(keys[mid]);
       if      (cmp  < 0) hi = mid - 1;
       else if (cmp  > 0) lo = mid + 1;
       else if (cmp == 0) return mid;
   }
   return lo;
}
```

number of keys < key

Binary search: mathematical analysis

Proposition. Binary search uses $\sim \lg N$ compares to search any array of size N.

Def. $T(N) \equiv$ number of compares to binary search in a sorted array of size N.

$$\leq \underbrace{T(N/2)}_{\text{left or right half}} + 1$$

Binary search recurrence. $T(N) \leq T(N/2) + 1$ for $N > 1$, with $T(1) = 1$.

- Not quite right for odd N.
- Same recurrence holds for many algorithms.

Solution. $T(N) \sim \lg N$.

- For simplicity, we'll prove when N is a power of 2.
- True for all N. [see COS 340]

Binary search recurrence

Binary search recurrence. $T(N) \le T(N/2) + 1$ for $N > 1$, with $T(1) = 1$.

Proposition. If N is a power of 2, then $T(N) \le \lg N + 1$.

Pf.

$$
\begin{aligned}
T(N) &\le T(N/2) + 1 && \text{given} \\
&\le T(N/4) + 1 + 1 && \text{apply recurrence to first term} \\
&\le T(N/8) + 1 + 1 + 1 && \text{apply recurrence to first term} \\
&\;\;\cdots \\
&\le T(N/N) + 1 + 1 + \ldots + 1 && \text{stop applying, } T(1) = 1 \\
&= \lg N + 1
\end{aligned}
$$

Binary search: trace of standard indexing client

Problem. To insert, need to shift all greater keys over.

		keys[]											vals[]									
key	value	0	1	2	3	4	5	6	7	8	9	N	0	1	2	3	4	5	6	7	8	9
S	0	S										1	0									
E	1	E	S									2	1	0								
A	2	A	E	S								3	2	1	0							
R	3	A	E	R	S							4	2	1	3	0						
C	4	A	C	E	R	S						5	2	4	1	3	0					
H	5	A	C	E	H	R	S					6	2	4	1	5	3	0				
E	6	A	C	E	H	R	S					6	2	4	6	5	3	0				
X	7	A	C	E	H	R	S	X				7	2	4	6	5	3	0	7			
A	8	A	C	E	H	R	S	X				7	8	4	6	5	3	0	7			
M	9	A	C	E	H	M	R	S	X			8	8	4	6	5	9	3	0	7		
P	10	A	C	E	H	M	P	R	S	X		9	8	4	6	5	9	10	3	0	7	
L	11	A	C	E	H	L	M	P	R	S	X	10	8	4	6	5	11	9	10	3	0	7
E	12	A	C	E	H	L	M	P	R	S	X	10	8	4	12	5	11	9	10	3	0	7
		A	C	E	H	L	M	P	R	S	X		8	4	12	5	11	9	10	3	0	7

entries in red were inserted

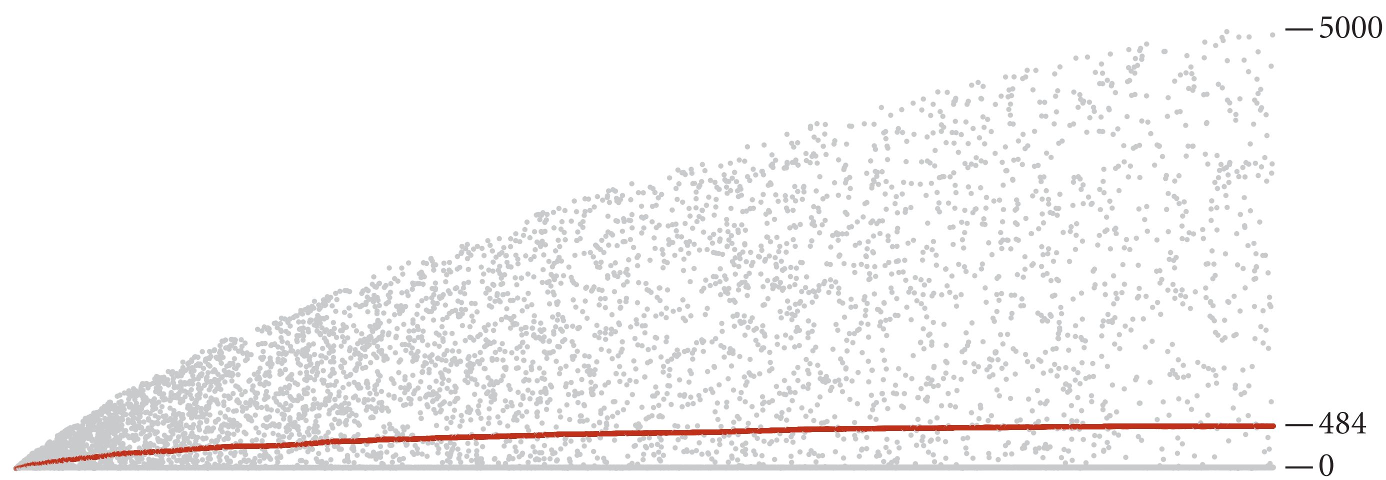
entries in gray did not move

entries in black moved to the right

circled entries are changed values

Elementary ST implementations: summary

ST implementation	worst case		average case		ordered iteration?	operations on keys
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	$N / 2$	N	no	`equals()`
binary search (ordered array)	$\log N$	N	$\log N$	$N / 2$	yes	`compareTo()`



Costs for `java FrequencyCounter 8 < tale.txt` **using** `OrderedArrayST`

Challenge. Efficient implementations of both search and insert.

- API
- sequential search
- binary search
- **ordered operations**

Ordered symbol table API

	keys	*values*
min() →	09:00:00	Chicago
	09:00:03	Phoenix
	09:00:13	Houston
get(09:00:13)	09:00:59	Chicago
	09:01:10	Houston
floor(09:05:00) →	09:03:13	Chicago
	09:10:11	Seattle
select(7) →	09:10:25	Seattle
	09:14:25	Phoenix
	09:19:32	Chicago
	09:19:46	Chicago
keys(09:15:00, 09:25:00) →	09:21:05	Chicago
	09:22:43	Seattle
	09:22:54	Seattle
	09:25:52	Chicago
ceiling(09:30:00) →	09:35:21	Chicago
	09:36:14	Seattle
max() →	09:37:44	Phoenix

size(09:15:00, 09:25:00) *is* 5
rank(09:10:25) *is* 7

Examples of ordered symbol-table operations

Ordered symbol table API

`public class ST<Key extends Comparable<Key>, Value>`

	`ST()`	*create an ordered symbol table*
`void`	`put(Key key, Value val)`	*put key-value pair into the table (remove* `key` *from table if value is null)*
`Value`	`get(Key key)`	*value paired with* `key` *(null if* `key` *is absent)*
`void`	`delete(Key key)`	*remove* `key` *(and its value) from table*
`boolean`	`contains(Key key)`	*is there a value paired with* `key`*?*
`boolean`	`isEmpty()`	*is the table empty?*
`int`	`size()`	*number of key-value pairs*
`Key`	`min()`	*smallest key*
`Key`	`max()`	*largest key*
`Key`	`floor(Key key)`	*largest key less than or equal to* `key`
`Key`	`ceiling(Key key)`	*smallest key greater than or equal to* `key`
`int`	`rank(Key key)`	*number of keys less than* `key`
`Key`	`select(int k)`	*key of rank* `k`
`void`	`deleteMin()`	*delete smallest key*
`void`	`deleteMax()`	*delete largest key*
`int`	`size(Key lo, Key hi)`	*number of keys in* `[lo..hi]`
`Iterable<Key>`	`keys(Key lo, Key hi)`	*keys in* `[lo..hi]`*, in sorted order*
`Iterable<Key>`	`keys()`	*all keys in the table, in sorted order*

API for a generic ordered symbol table

Binary search: ordered symbol table operations summary

	sequential search	binary search
search	N	lg N
insert	1	N
min / max	N	1
floor / ceiling	N	lg N
rank	N	lg N
select	N	1
ordered iteration	N log N	N

worst-case running time of ordered symbol table operations

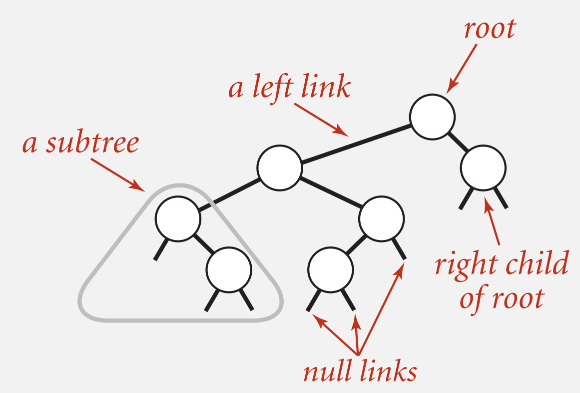
3.2 Binary Search Trees

- BSTs
- ordered operations
- deletion

Definition. A BST is a binary tree in symmetric order.

A binary tree is either:

- Empty.
- Two disjoint binary trees (left and right).

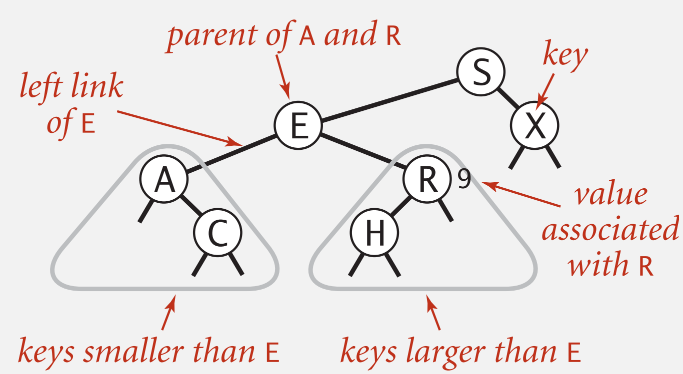


Anatomy of a binary tree

Symmetric order.

Each node has a key, and every node's key is:

- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.



Anatomy of a binary search tree

BST representation in Java

Java definition. A BST is a reference to a root `Node`.

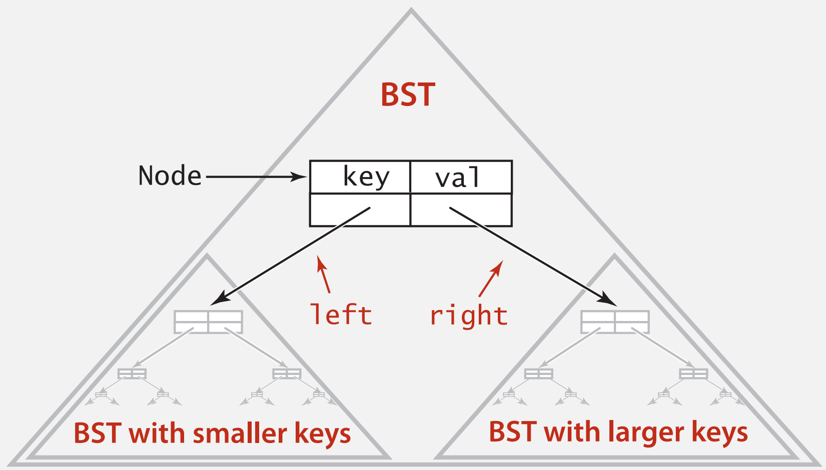
A `Node` is comprised of four fields:

- A `Key` and a `Value`.
- A reference to the left and right subtree.

smaller keys   larger keys

```
private class Node
{
   private Key key;
   private Value val;
   private Node left, right;
   public Node(Key key, Value val)
   {
      this.key = key;
      this.val = val;
   }
}
```

`Key` and `Value` are generic types; `Key` is `Comparable`



Binary search tree

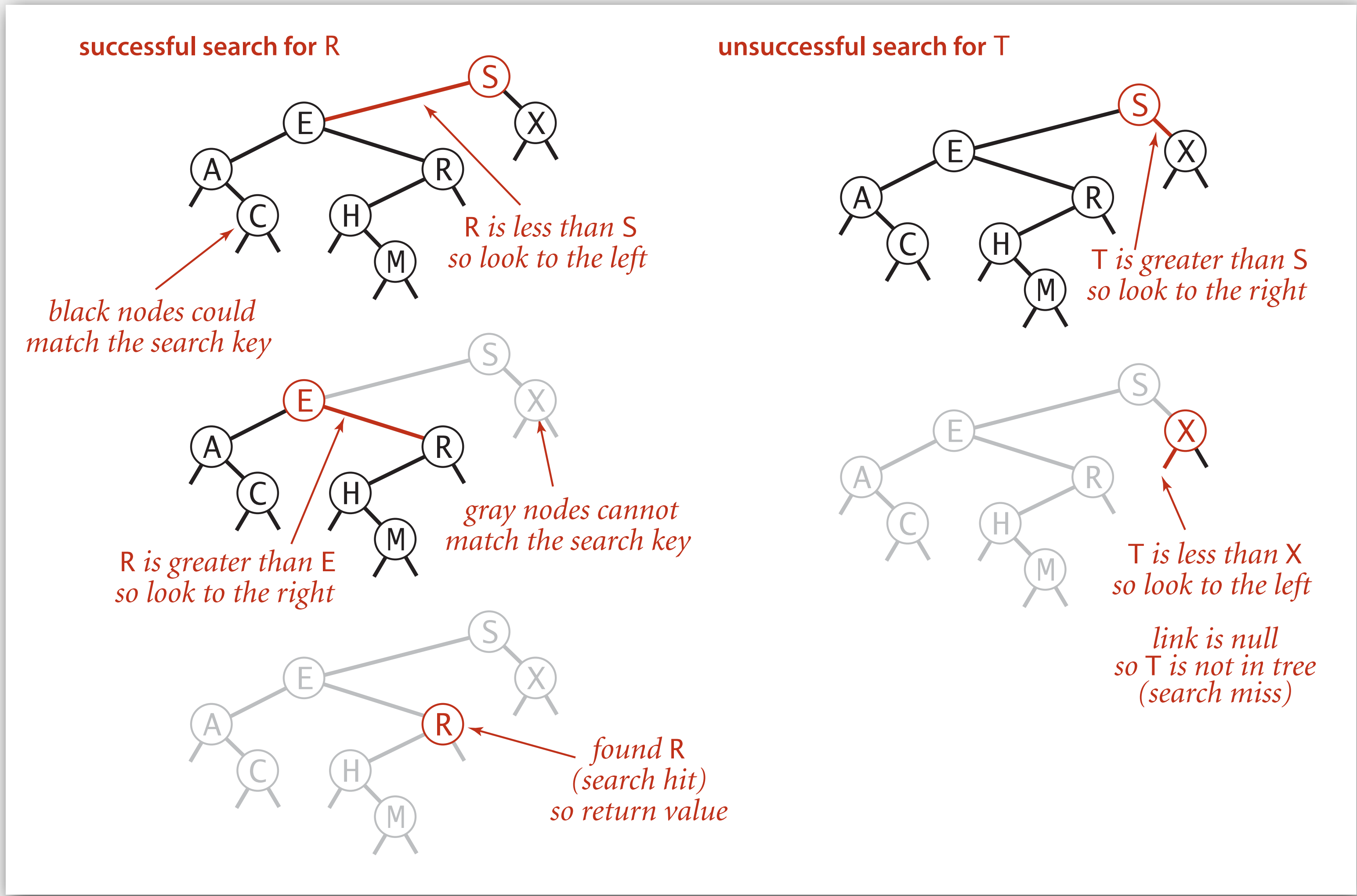
BST implementation (skeleton)

```
public class BST<Key extends Comparable<Key>, Value>
{
   private Node root;

   private class Node
   {  /* see previous slide */  }

   public void put(Key key, Value val)
   {  /* see next slides */  }

   public Value get(Key key)
   {  /* see next slides */  }

   public void delete(Key key)
   {  /* see next slides */  }

   public Iterable<Key> iterator()
   {  /* see next slides */  }

}
```

root of BST

BST search

Get. Return value corresponding to given key, or `null` if no such key.



BST search: Java implementation

Get. Return value corresponding to given key, or `null` if no such key.

```
public Value get(Key key)
{
   Node x = root;
   while (x != null)
   {
      int cmp = key.compareTo(x.key);
      if      (cmp  < 0) x = x.left;
      else if (cmp  > 0) x = x.right;
      else if (cmp == 0) return x.val;
   }
   return null;
}
```

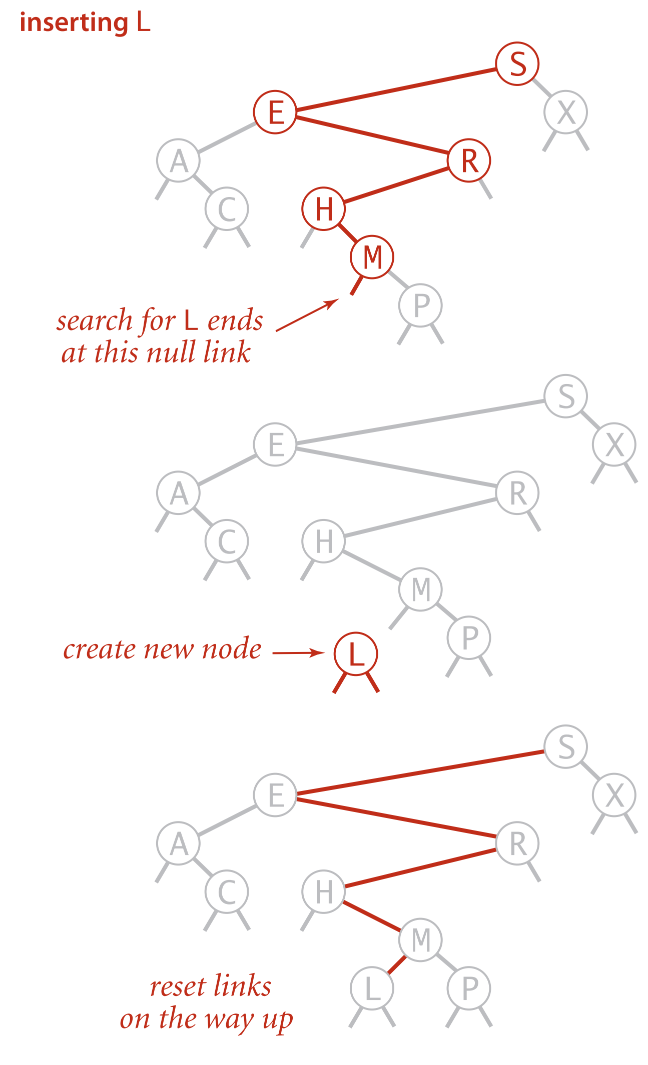
Running time. Proportional to depth of node.

BST insert

Put. Associate value with key.

Search for key, then two cases:

- Key in tree $\Rightarrow$ reset value.
- Key not in tree $\Rightarrow$ add new node.



Insertion into a BST

BST insert: Java implementation

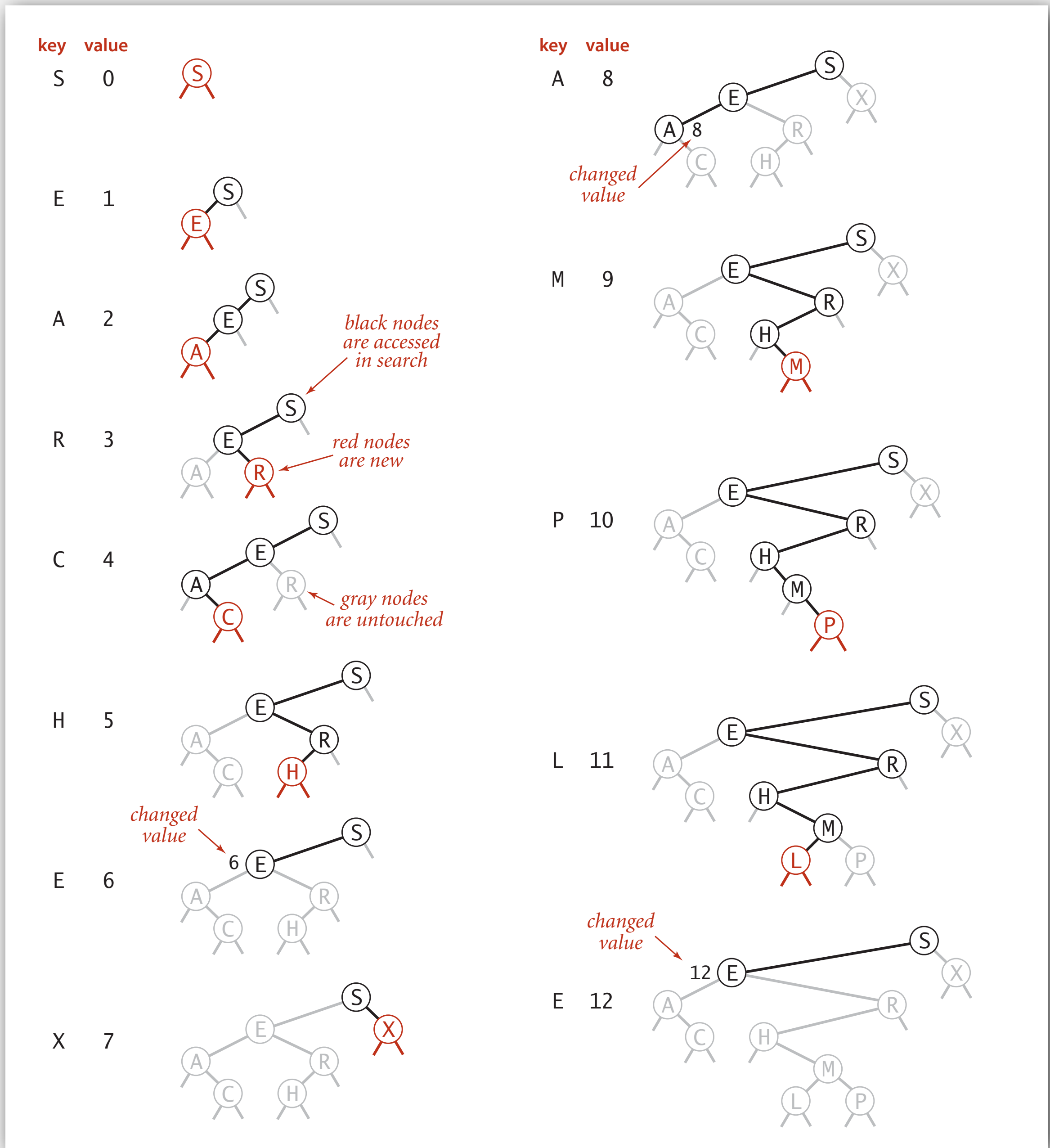
Put. Associate value with key.

concise, but tricky, recursive code; read carefully!

```
public void put(Key key, Value val)
{  root = put(root, key, val);  }

private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val);
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0)
      x.left  = put(x.left,  key, val);
   else if (cmp  > 0)
      x.right = put(x.right, key, val);
   else if (cmp == 0)
      x.val = val;
   return x;
}
```

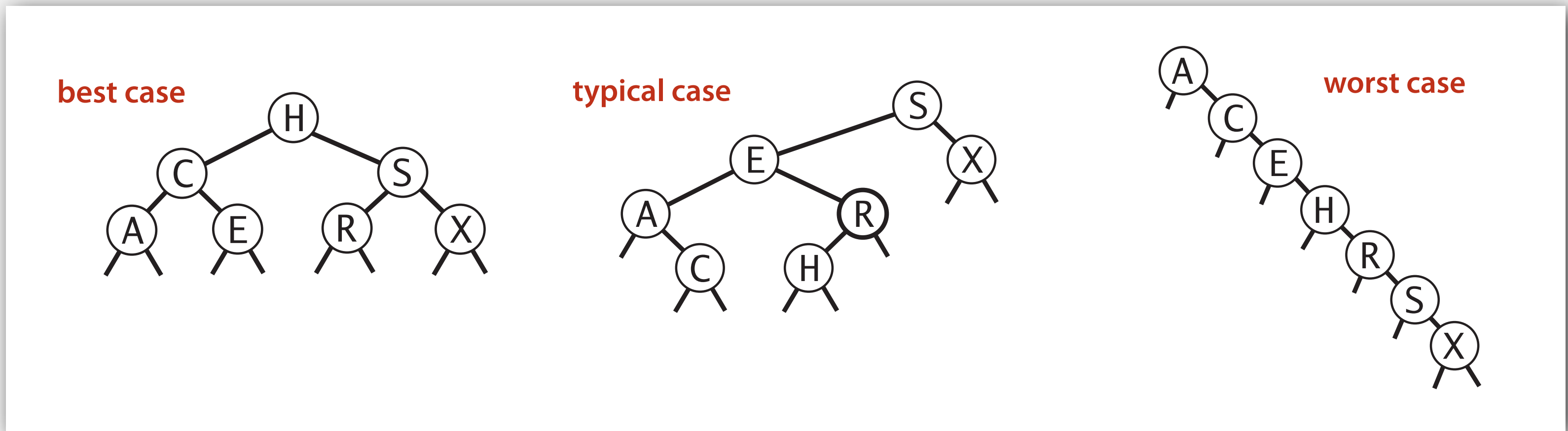
Running time. Proportional to depth of node.

BST trace: standard indexing client



Tree shape

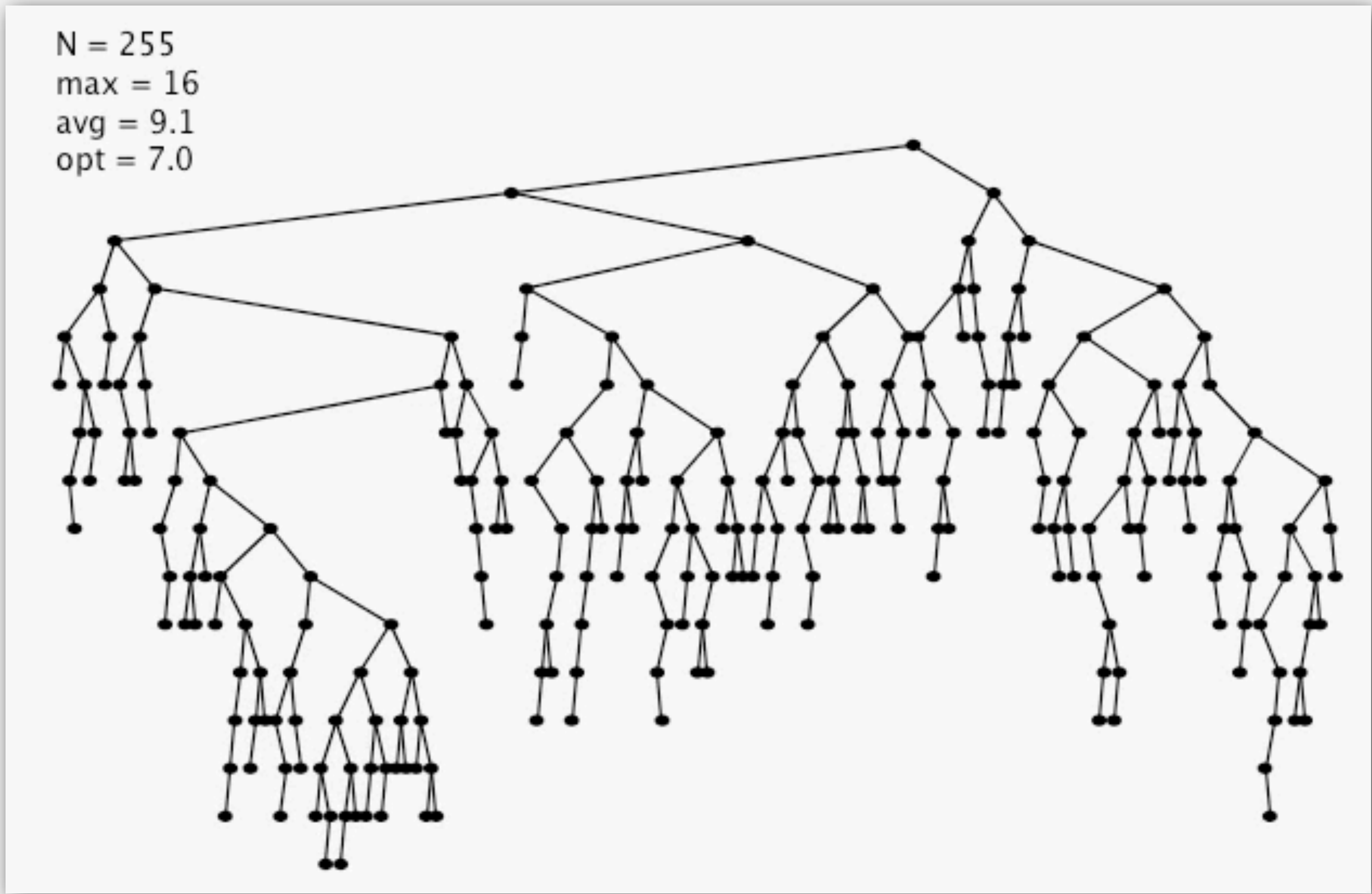
- Many BSTs correspond to same set of keys.
- Cost of search/insert is proportional to depth of node.



Remark. Tree shape depends on order of insertion.

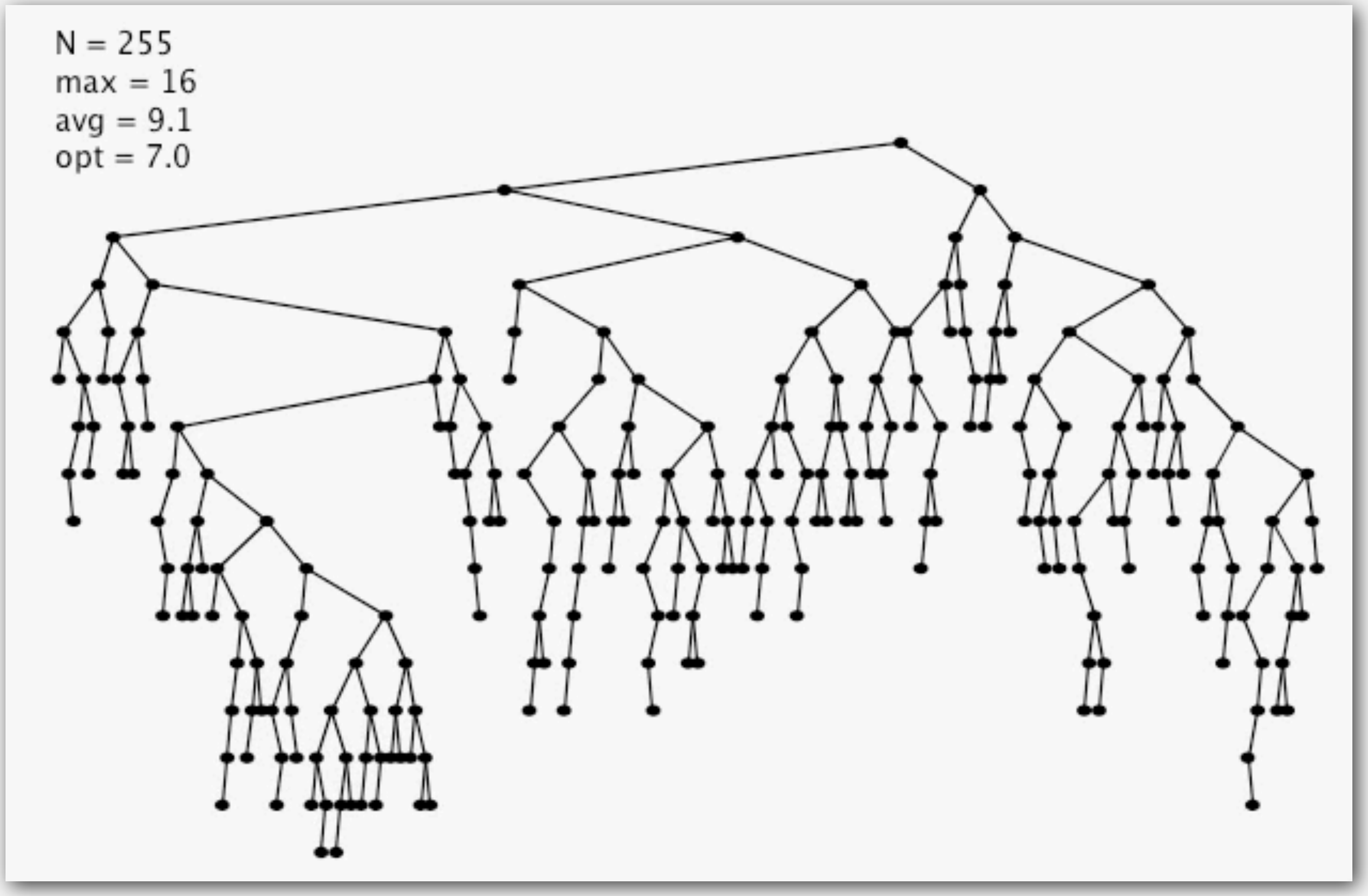
BST insertion: random order

Observation. If keys inserted in random order, tree stays relatively flat.

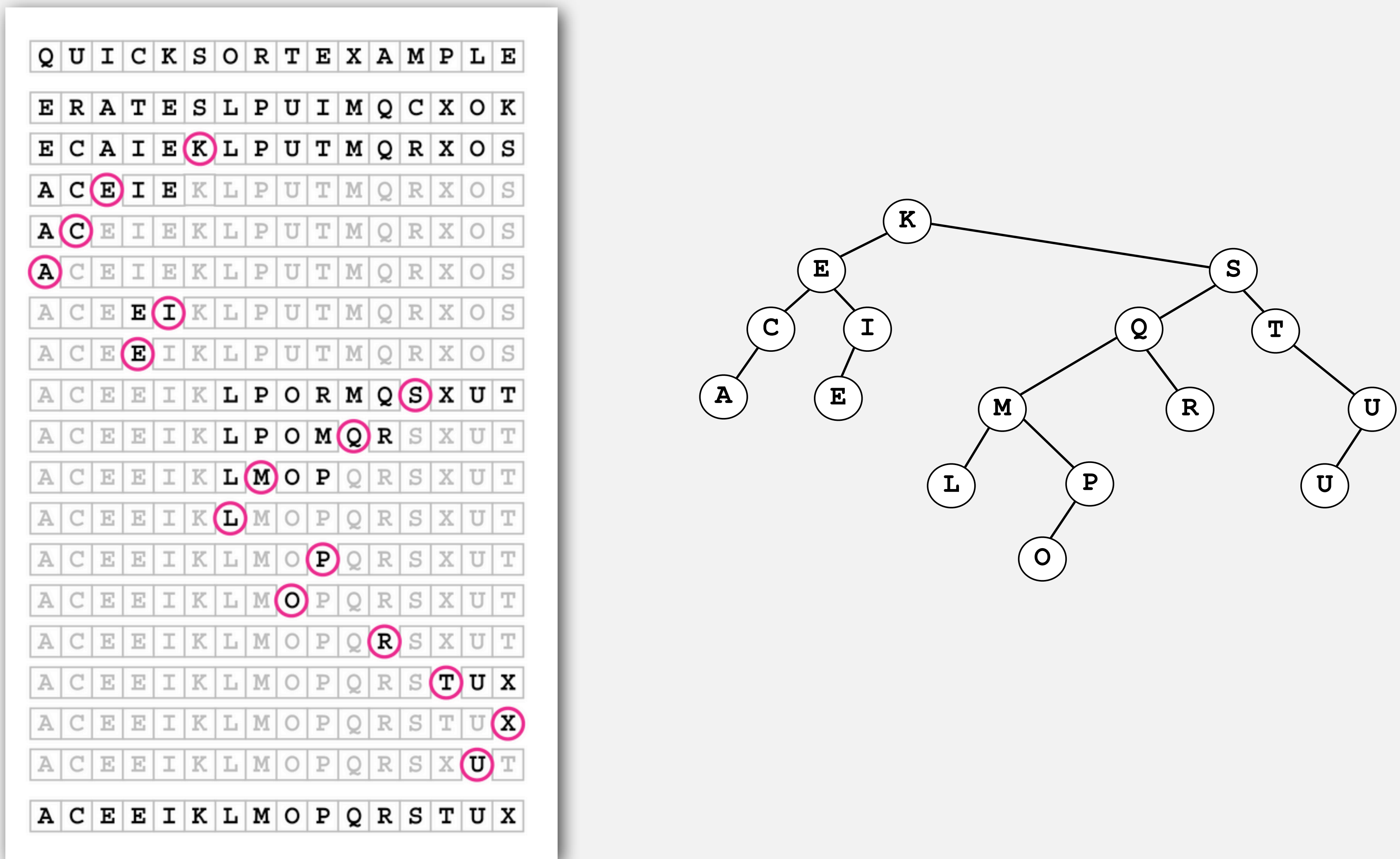


BST insertion: random order visualization

Ex. Insert keys in random order.



Correspondence between BSTs and quicksort partitioning



Remark. Correspondence is 1-1 if no duplicate keys.

BSTs: mathematical analysis

Proposition. If keys are inserted in random order, the expected number of compares for a search/insert is ~ 2 ln N.

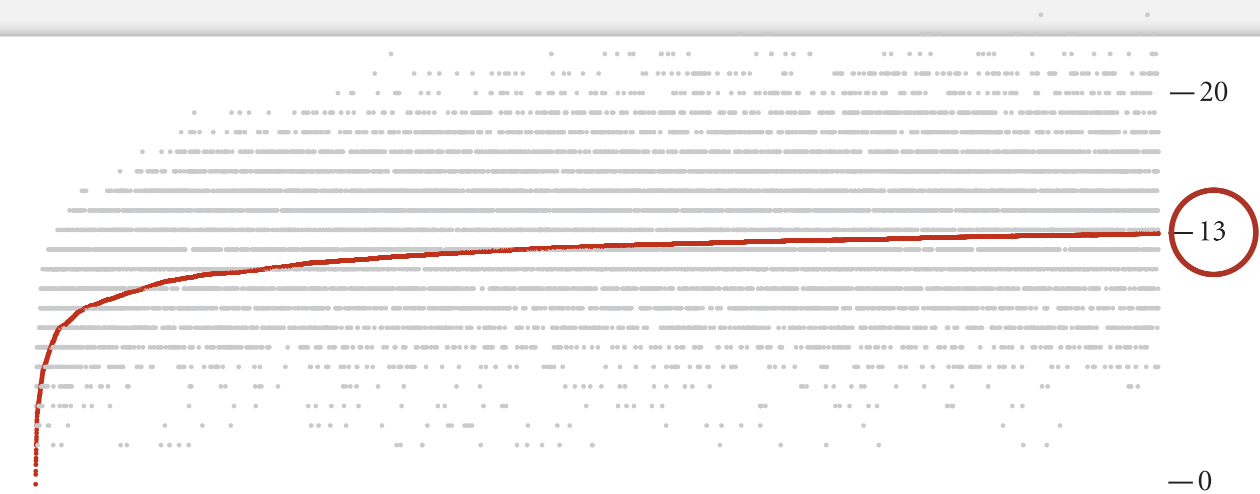
Pf. 1-1 correspondence with quicksort partitioning.

Proposition. [Reed, 2003] If keys are inserted in random order, expected height of tree is ~ 4.311 ln N.

But… Worst-case for search/insert/height is N.
(exponentially small chance when keys are inserted in random order)

ST implementations: summary

implementation	guarantee		average case		ordered ops?	operations on keys
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	N/2	N	no	`equals()`
binary search (ordered array)	lg N	N	lg N	N/2	yes	`compareTo()`
BST	N	N	1.39 lg N	1.39 lg N	?	`compareTo()`



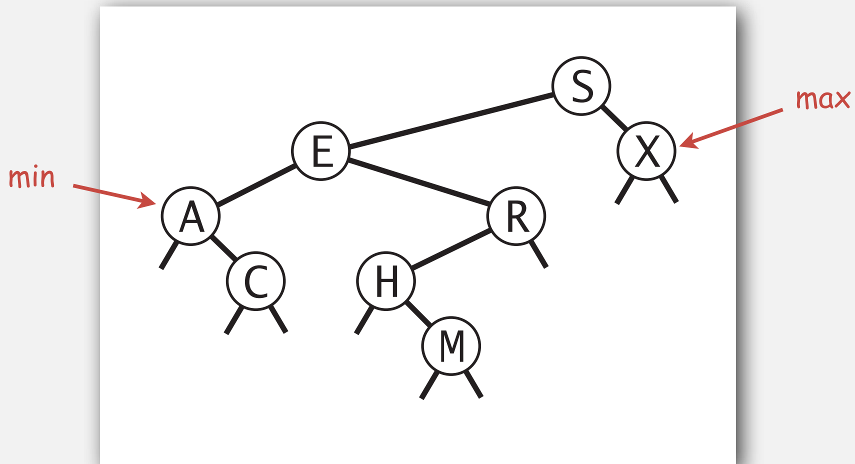
Costs for `java FrequencyCounter 8 < tale.txt` using BST

- BSTs
- ordered operations
- deletion

Minimum and maximum

Minimum. Smallest key in table.

Maximum. Largest key in table.

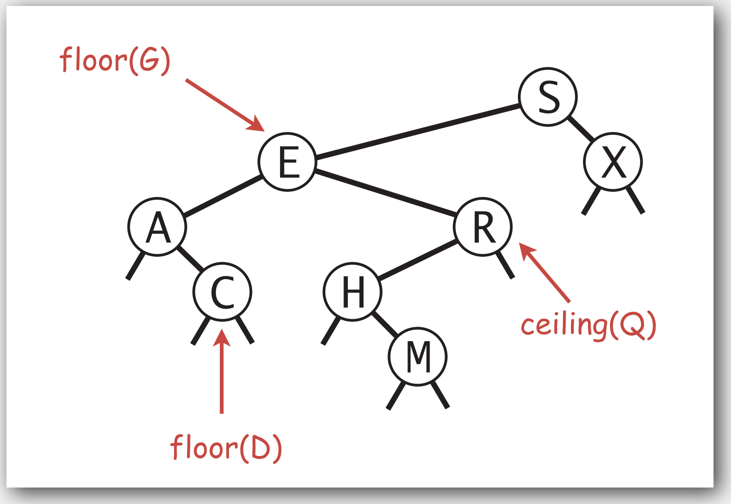


Q. How to find the min / max.

Floor and ceiling

Floor. Largest key ≤ to a given key.

Ceiling. Smallest key ≥ to a given key.



Q. How to find the floor /ceiling.

Computing the floor

Case 1. [k equals the key at root]

The floor of k is k.

Case 2. [k is less than the key at root]

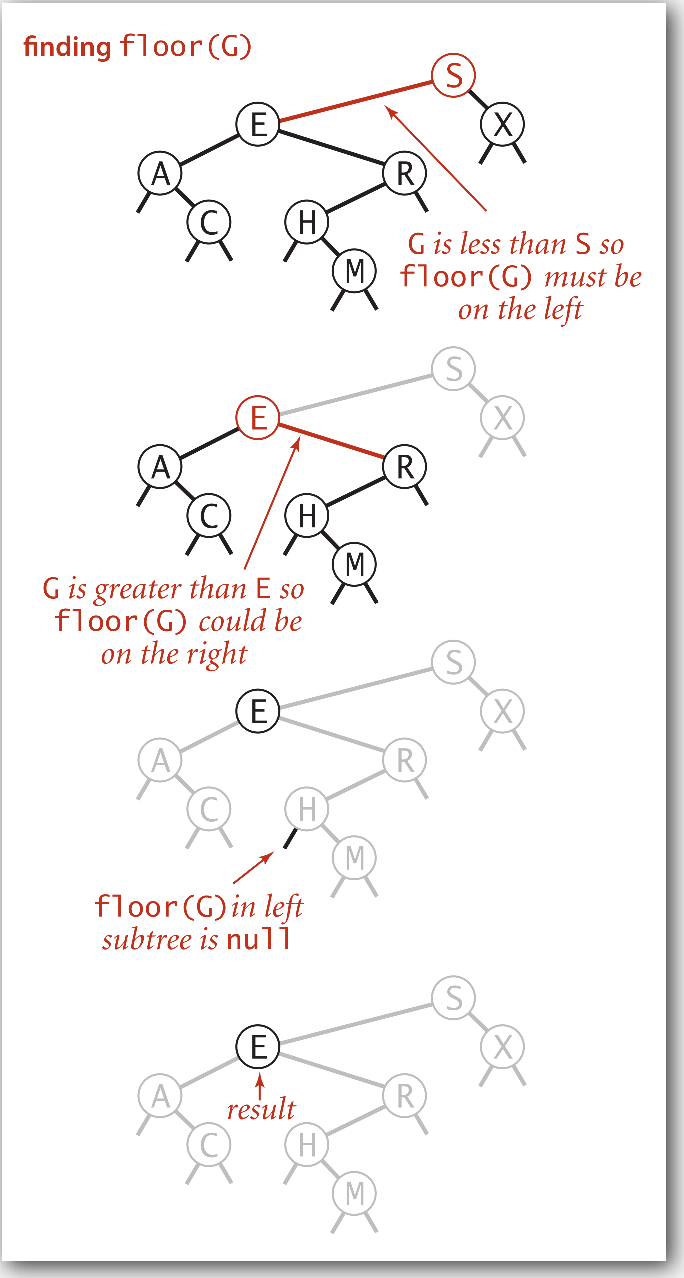
The floor of k is in the left subtree.

Case 3. [k is greater than the key at root]

The floor of k is in the right subtree

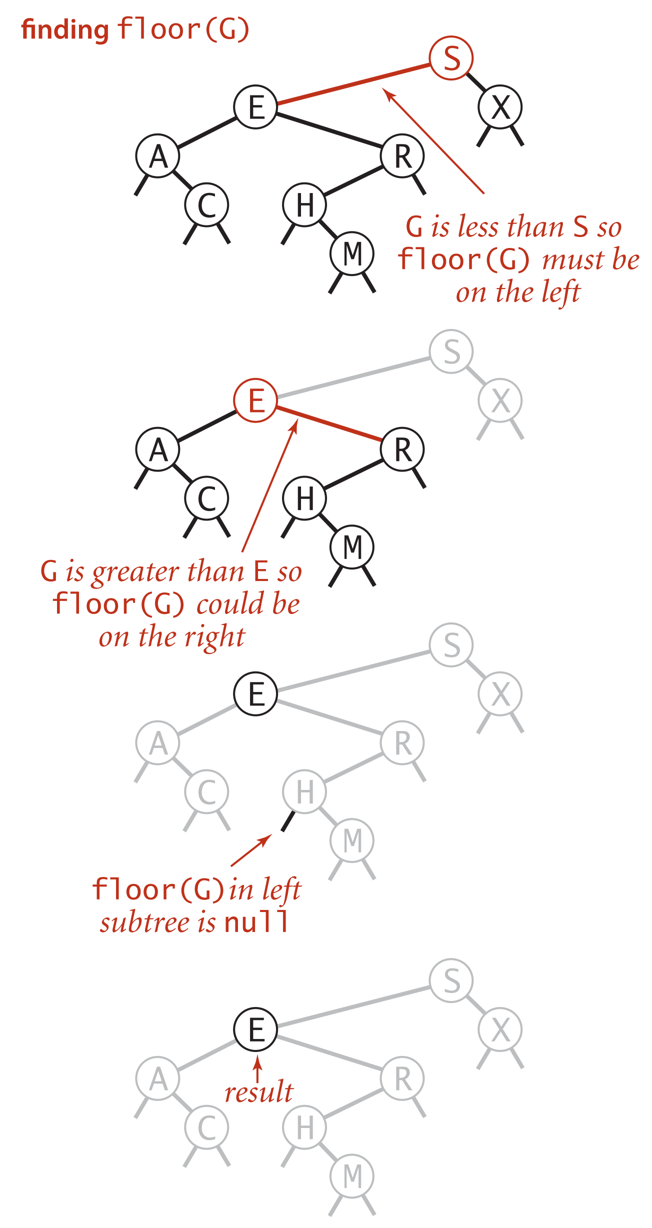
(if there is any key ≤ k in right subtree);

otherwise it is the key in the root.



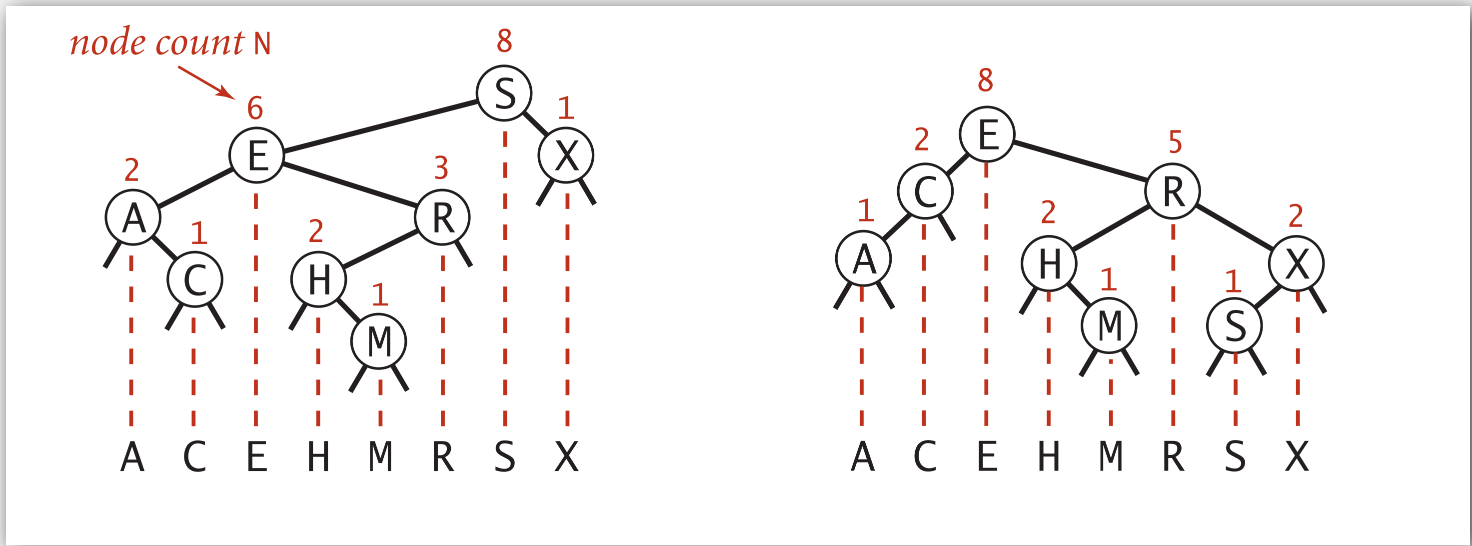
Computing the floor

```
public Key floor(Key key)
{
   Node x = floor(root, key);
   if (x == null) return null;
   return x.key;
}
private Node floor(Node x, Key key)
{
   if (x == null) return null;
   int cmp = key.compareTo(x.key);

   if (cmp == 0) return x;

   if (cmp < 0)  return floor(x.left, key);

   Node t = floor(x.right, key);
   if (t != null) return t;
   else           return x;

}
```



In each node, we store the number of nodes in the subtree rooted at that node.

To implement `size()`, return the count at the root.



Remark. This facilitates efficient implementation of `rank()` and `select()`.

BST implementation: subtree counts

```
private class Node
{
   private Key key;
   private Value val;
   private Node left;
   private Node right;
   private int N;
}
```

nodes in subtree

```
public int size()
{  return size(root);  }
```

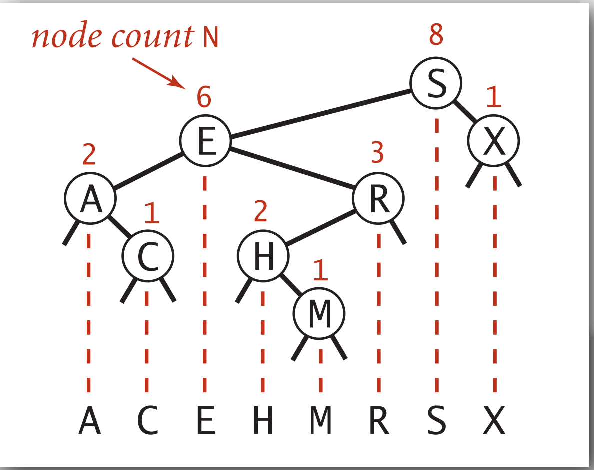
```
private int size(Node x)
{
   if (x == null) return 0;
   return x.N;
}
```

```
private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val);
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0) x.left  = put(x.left,  key, val);
   else if (cmp  > 0) x.right = put(x.right, key, val);
   else if (cmp == 0) x.val = val;
   x.N = 1 + size(x.left) + size(x.right);
   return x;
}
```

Rank

Rank. How many keys < k?

Easy recursive algorithm (4 cases!)

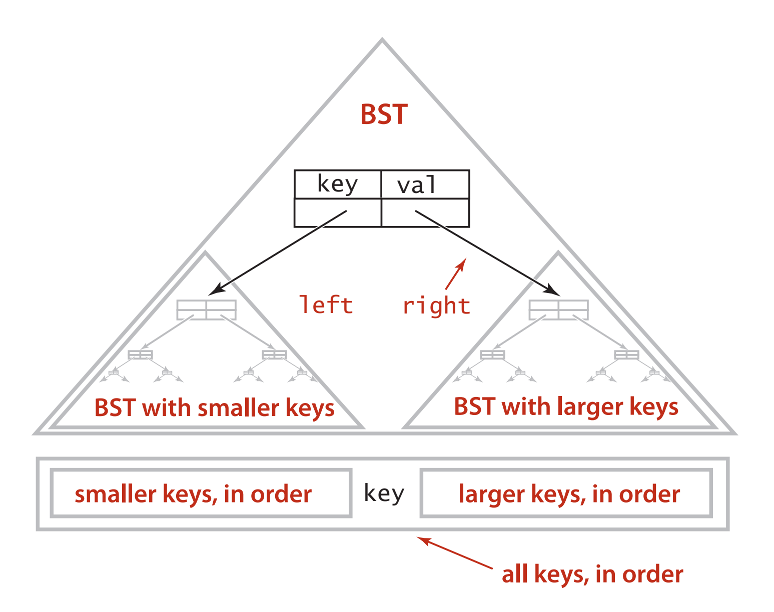



```
public int rank(Key key)
{  return rank(key, root);  }

private int rank(Key key, Node x)
{
   if (x == null) return 0;
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) return rank(key, x.left);
   else if (cmp > 0) return 1 + size(x.left) + rank(key, x.right);
   else              return size(x.left);
}
```

Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.

```
public Iterable<Key> keys()
{
    Queue<Key> q = new Queue<Key>();
    inorder(root, queue);
    return q;
}

private void inorder(Node x, Queue<Key> q)
{
   if (x == null) return;
   inorder(x.left, q);
   q.enqueue(x.key);
   inorder(x.right, q);
}
```

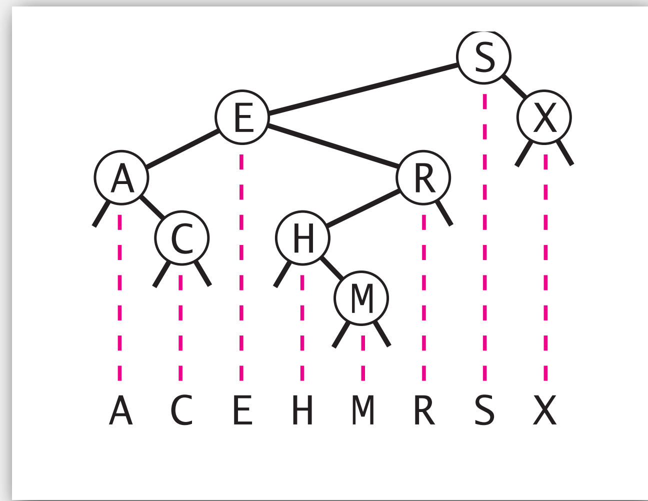


Property. Inorder traversal of a BST yields keys in ascending order.

Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.

recursive calls	queue	function call stack
inorder(S)		S
inorder(E)		S E
inorder(A)		S E A
enqueue A	A	
inorder(C)		S E A C
enqueue C	C	
enqueue E	E	
inorder(R)		S E R
inorder(H)		S E R H
enqueue H	H	
inorder(M)		S E R H M
enqueue M	M	
print R	R	
enqueue S	S	
inorder(X)		S X
enqueue X	X	



BST: ordered symbol table operations summary

	sequential search	binary search	BST
search	N	lg N	h
insert	1	N	h
min / max	N	1	h
floor / ceiling	N	lg N	h
rank	N	lg N	h
select	N	1	h
ordered iteration	N log N	N	N

h = height of BST
(proportional to log N
if keys inserted in random order)

worst-case running time of ordered symbol table operations

- BSTs
- ordered operations
- **deletion**

ST implementations: summary

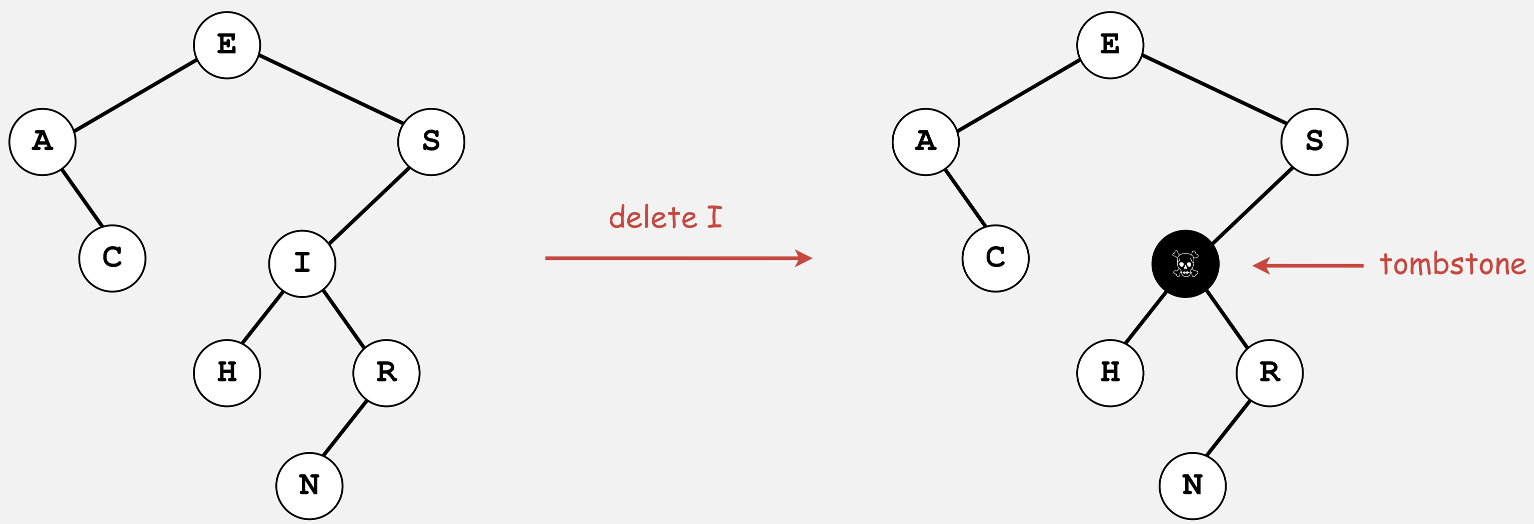
implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.39 lg N	1.39 lg N	???	yes	`compareTo()`

Next. Deletion in BSTs.

BST deletion: lazy approach

To remove a node with a given key:

- Set its value to `null`.
- Leave key in tree to guide searches (but don't consider it equal to search key).



Cost. O(log N') per insert, search, and delete (if keys in random order), where N' is the number of key-value pairs ever inserted in the BST.

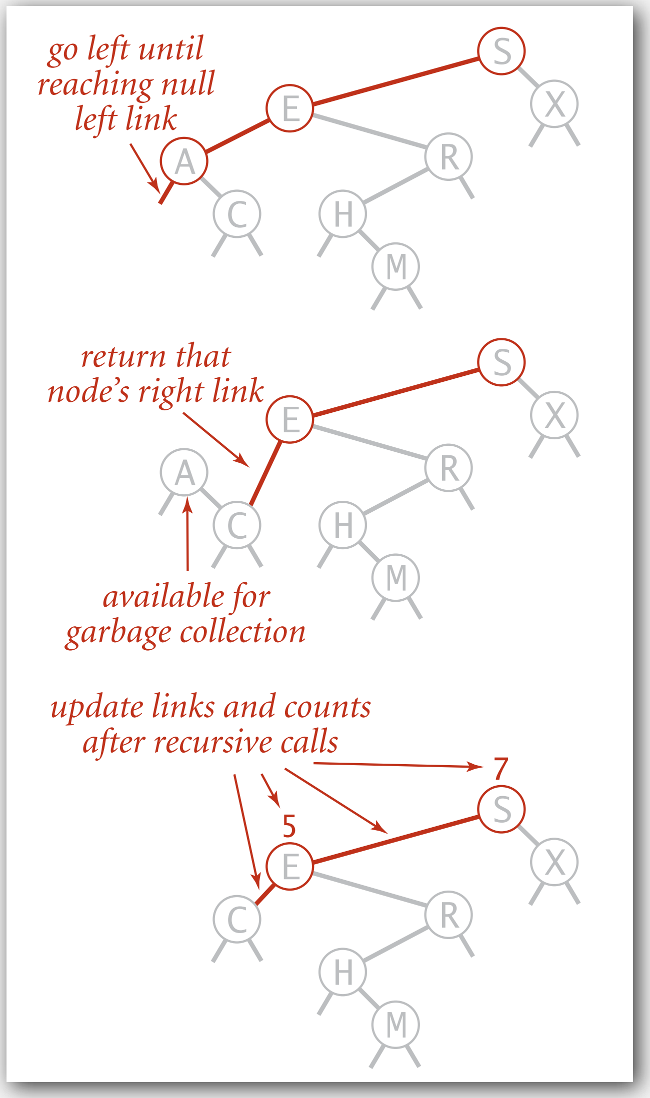
Unsatisfactory solution. Tombstone overload.

Deleting the minimum

To delete the minimum key:

- Go left until finding a node with a null left link.
- Replace that node by its right link.
- Update subtree counts.

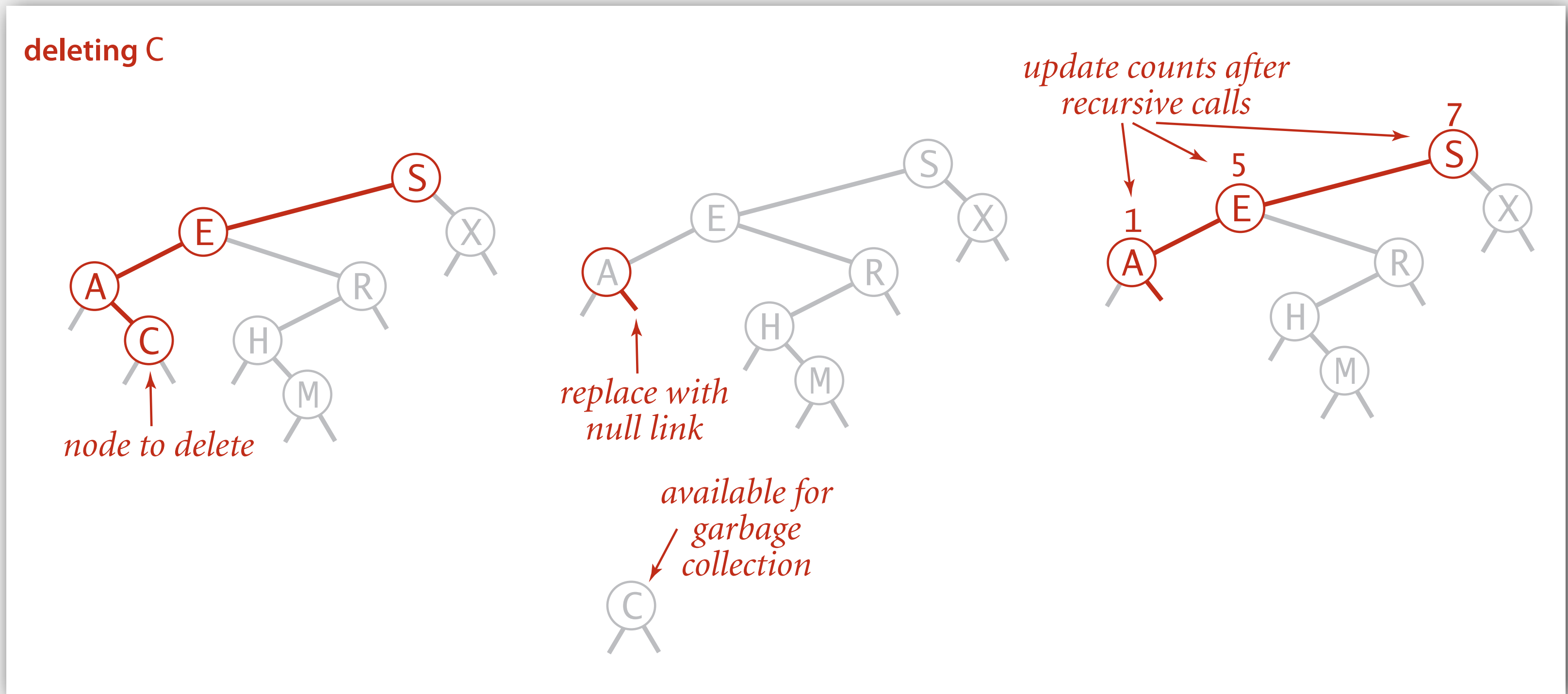
```
public void deleteMin()
{  root = deleteMin(root);  }

private Node deleteMin(Node x)
{
   if (x.left == null) return x.right;
   x.left = deleteMin(x.left);
   x.N = 1 + size(x.left) + size(x.right);
   return x;
}
```



Hibbard deletion

To delete a node with key k: search for node t containing key k.

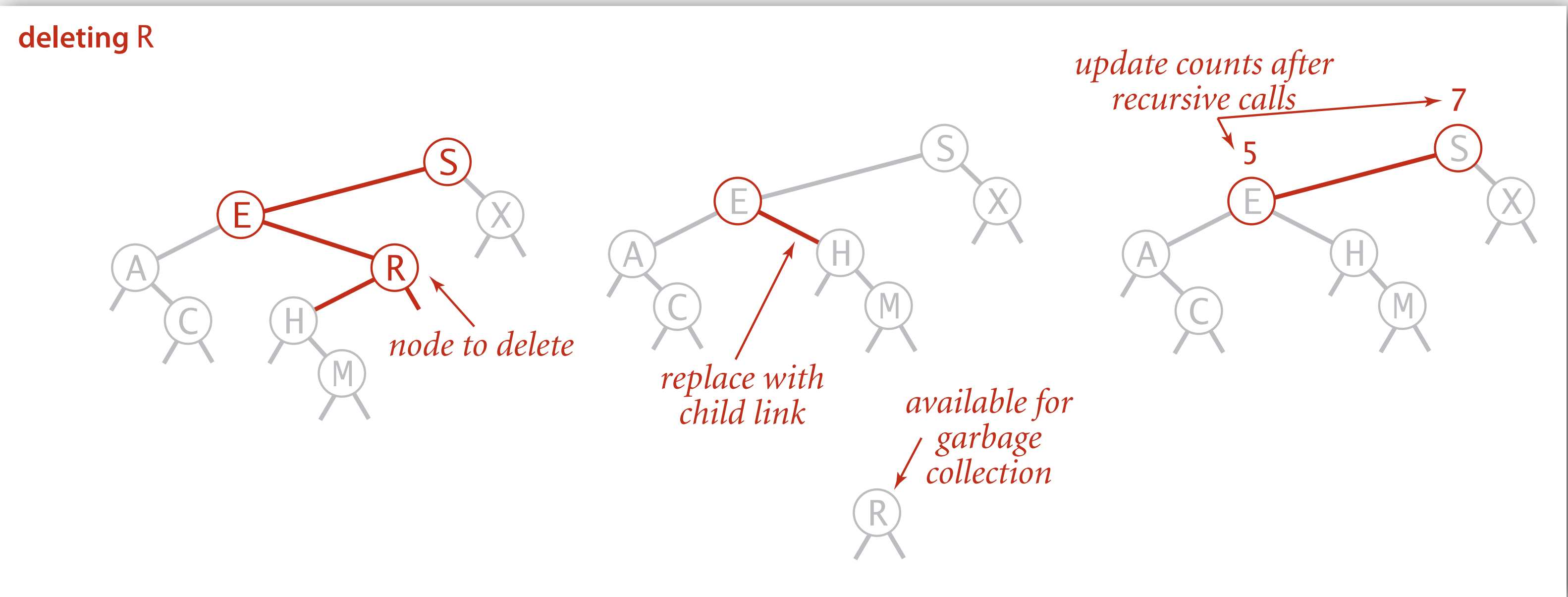
Case 0. [0 children] Delete t by setting parent link to null.



Hibbard deletion

To delete a node with key k: search for node t containing key k.

Case 1. [1 child] Delete t by replacing parent link.

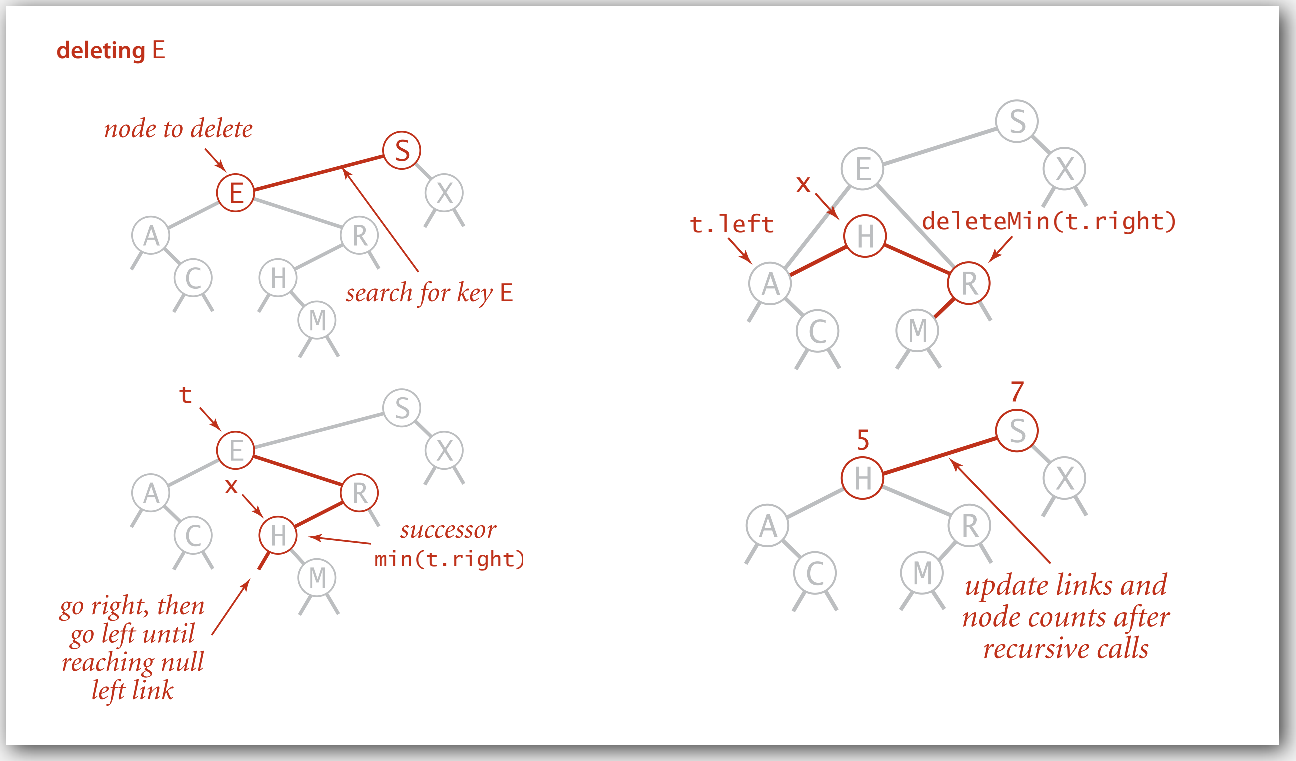


Hibbard deletion

To delete a node with key k: search for node t containing key k.

Case 2. [2 children]

- Find successor x of t. ← x has no left child
- Delete the minimum in t's right subtree. ← but don't garbage collect x
- Put x in t's spot. ← still a BST



Hibbard deletion: Java implementation

```
public void delete(Key key)
{  root = delete(root, key);  }

private Node delete(Node x, Key key) {
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) x.left  = delete(x.left,  key);
   else if (cmp > 0) x.right = delete(x.right, key);
   else {
      if (x.right == null) return x.left;

      Node t = x;
      x = min(t.right);
      x.right = deleteMin(t.right);
      x.left = t.left;
   }
   x.N = size(x.left) + size(x.right) + 1;
   return x;
}
```

search for key

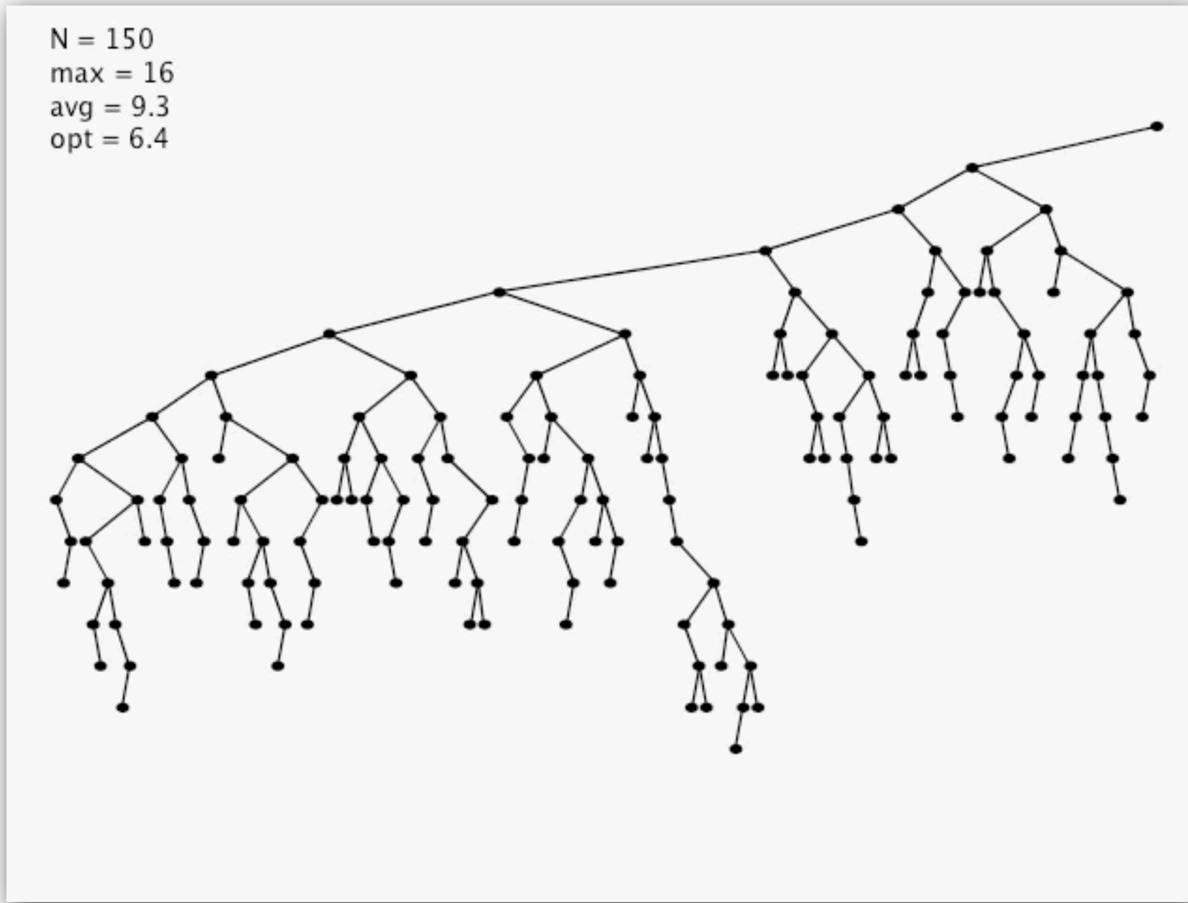
no right child

replace with successor

update subtree counts

Hibbard deletion: analysis

Unsatisfactory solution. Not symmetric.



Surprising consequence. Trees not random (!) $\Rightarrow$ sqrt(N) per op.

Longstanding open problem. Simple and efficient delete for BSTs.

ST implementations: summary

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.39 lg N	1.39 lg N	√N	yes	`compareTo()`

other operations also become √N
if deletions allowed

Next lecture. Guarantee logarithmic performance for all operations.

3.3 Balanced Trees

- 2-3 trees
- red-black trees
- B-trees

Symbol table review

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes	`compareTo()`
Goal	log N	log N	log N	log N	log N	log N	yes	`compareTo()`

Challenge. Guarantee performance.

This lecture. 2-3 trees, left-leaning red-black trees, B-trees.

introduced to the world in COS 226, Fall 2007

- **2-3 trees**
- red-black trees
- B-trees

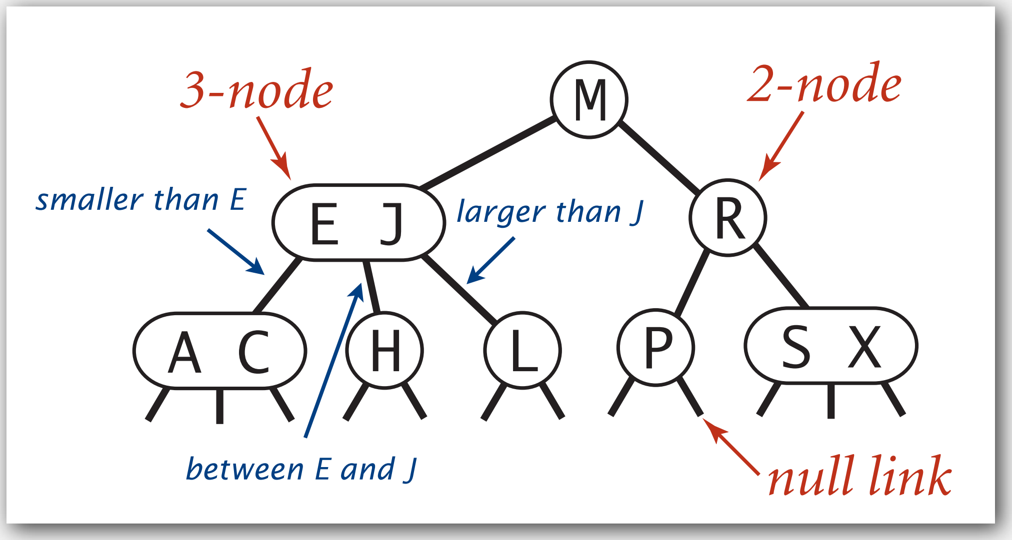
2-3 tree

Allow 1 or 2 keys per node.

- 2-node: one key, two children.
- 3-node: two keys, three children.

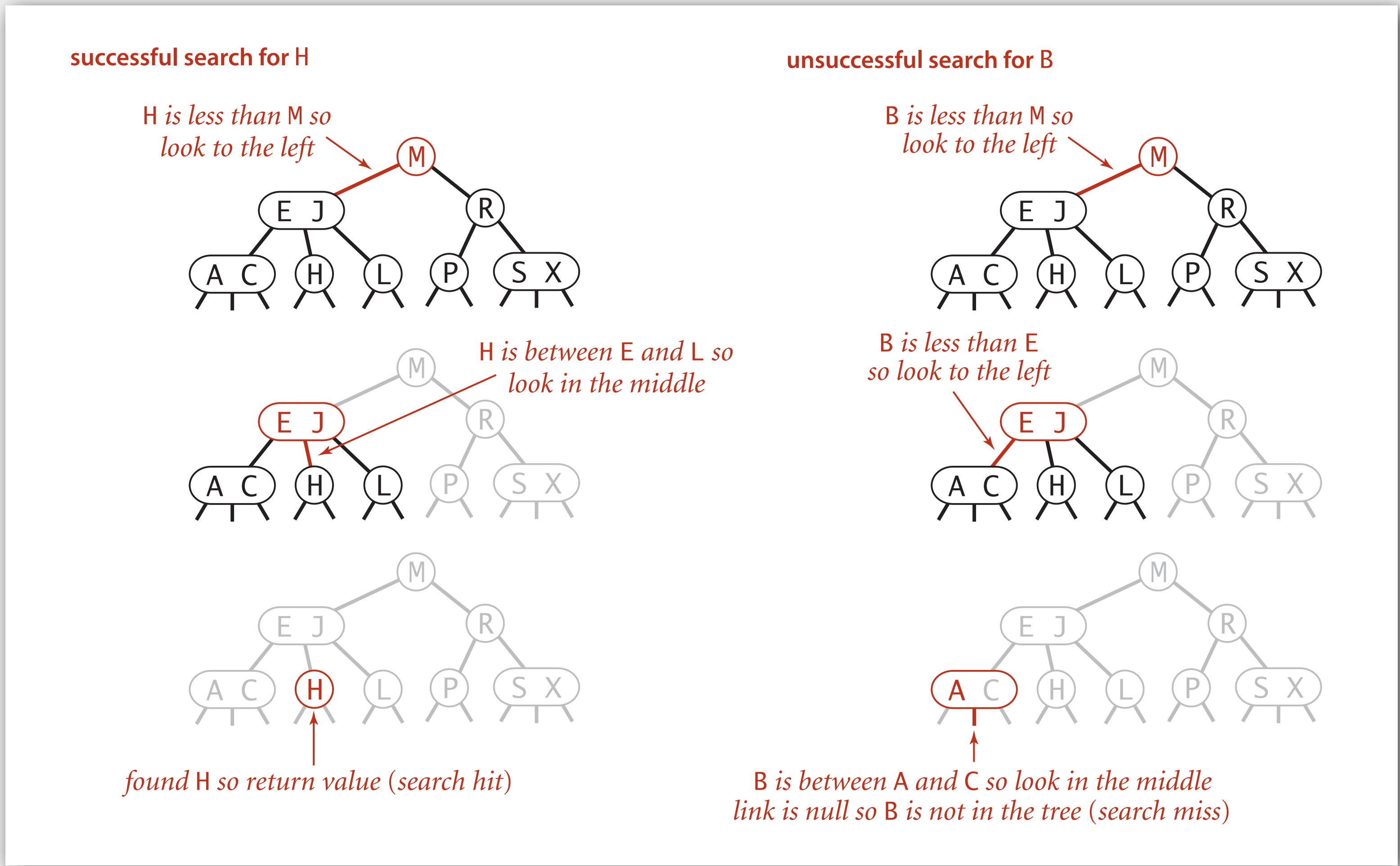
Symmetric order. Inorder traversal yields keys in ascending order.

Perfect balance. Every path from root to null link has same length.



Search in a 2-3 tree

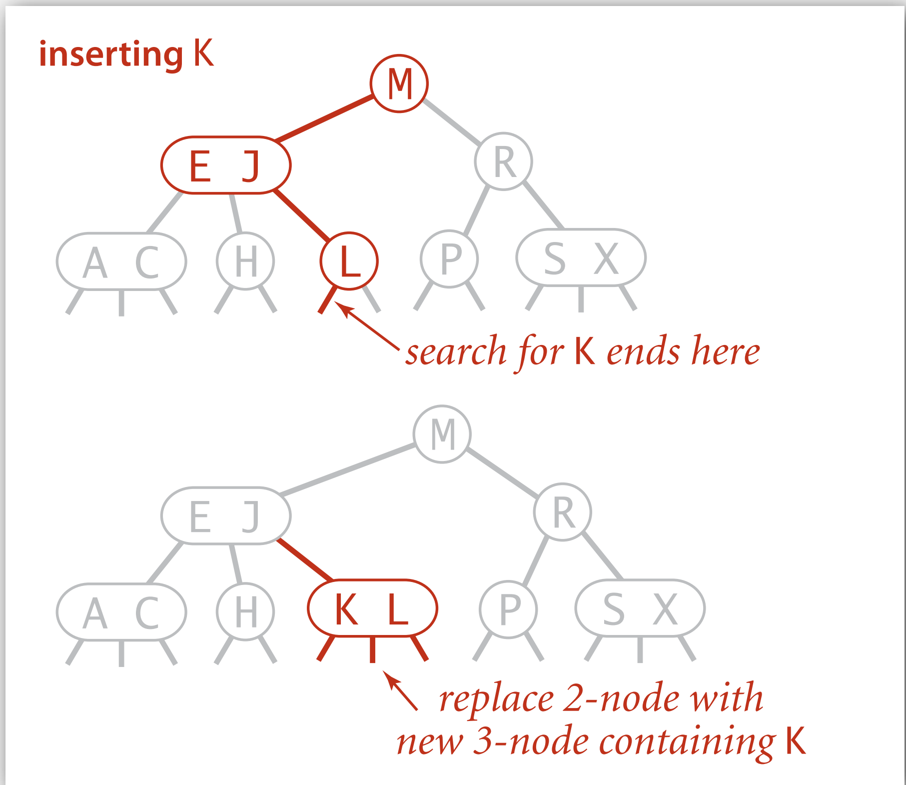
- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).



Insertion in a 2-3 tree

Case 1. Insert into a 2-node at bottom.

- Search for key, as usual.
- Replace 2-node with 3-node.

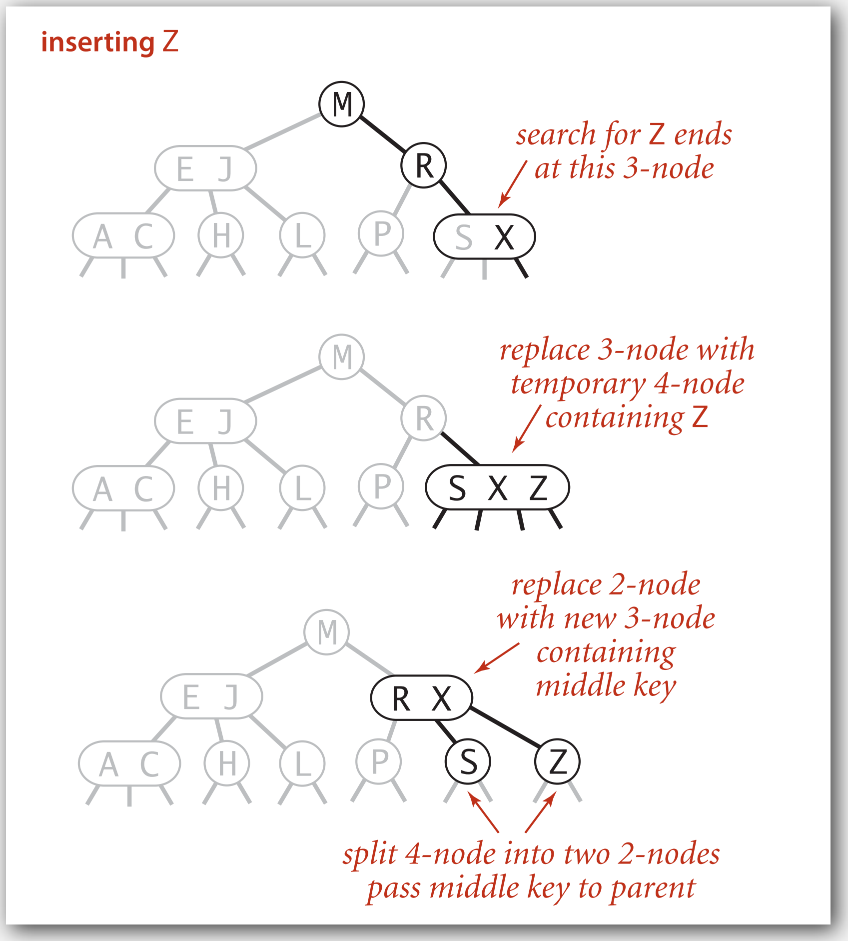


Insertion in a 2-3 tree

Case 2. Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.

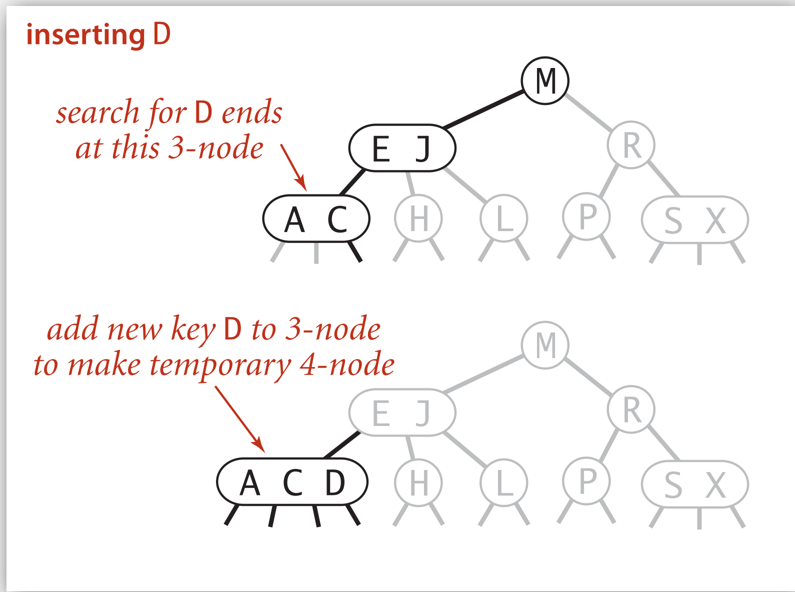
why middle key?

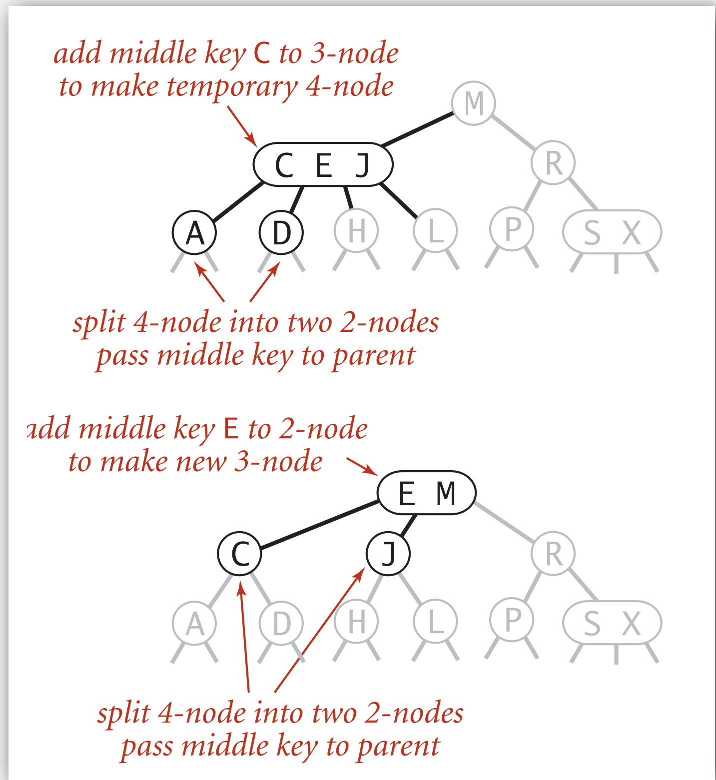


Insertion in a 2-3 tree

Case 2. Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.

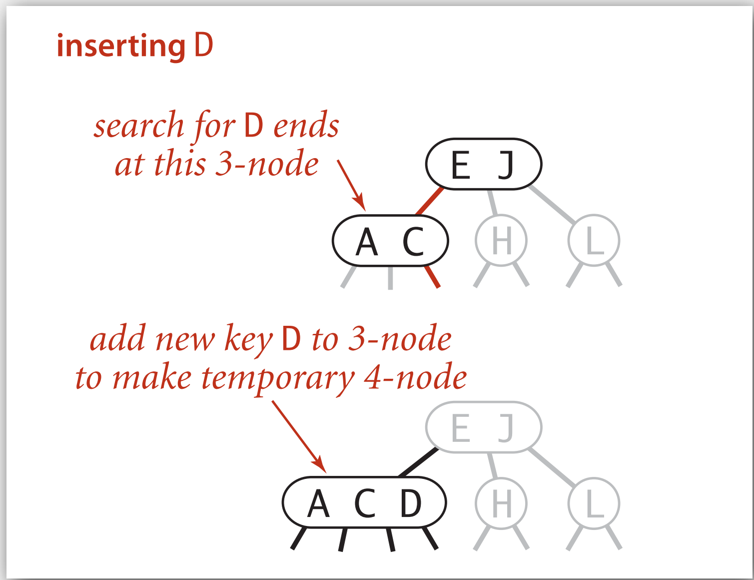


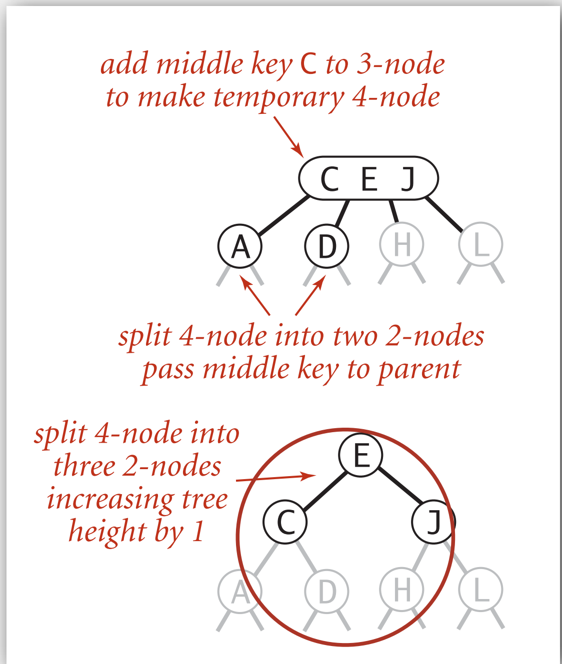


Insertion in a 2-3 tree

Case 2. Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.
- If you reach the root and it's a 4-node, split it into three 2-nodes.

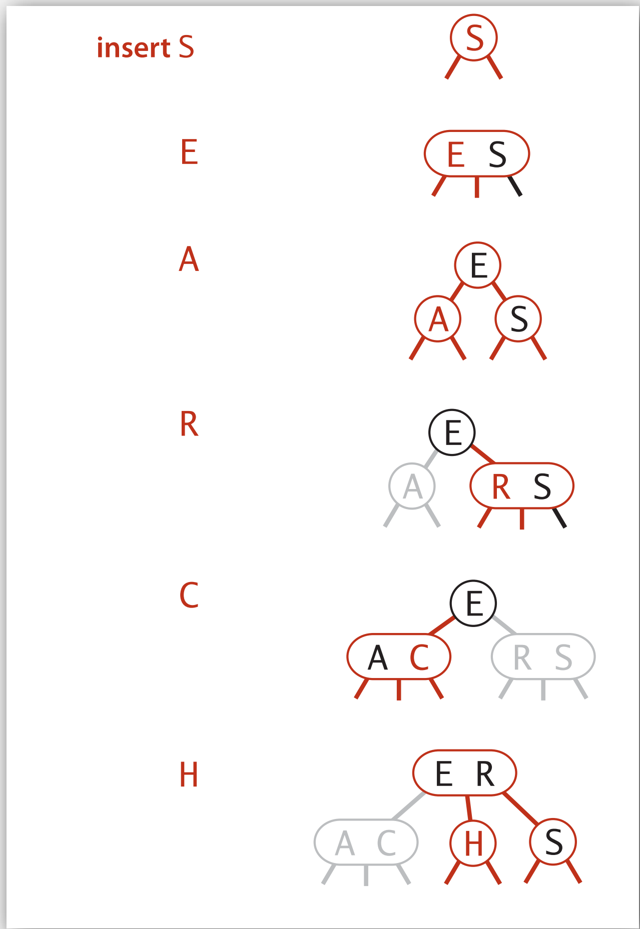


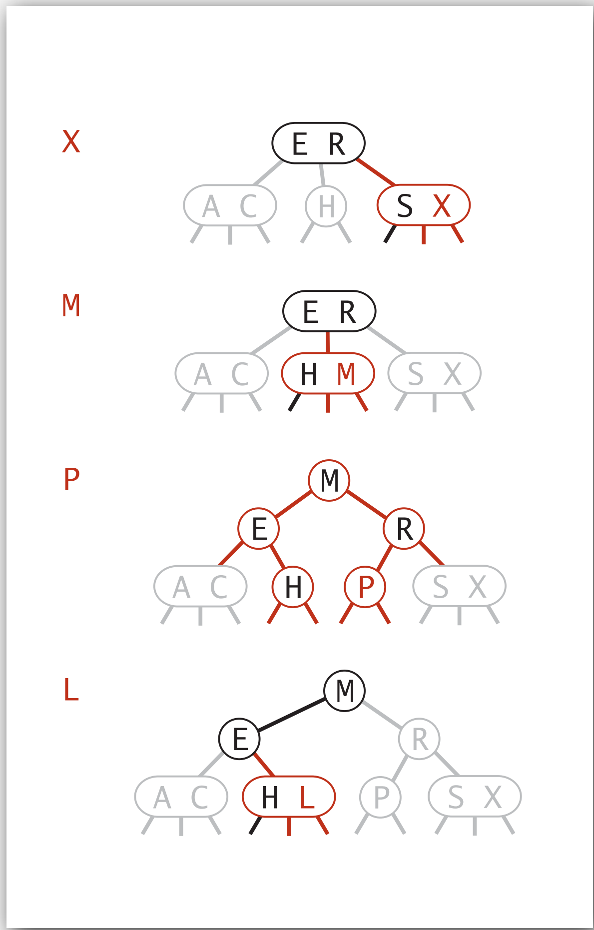


Remark. Splitting the root increases height by 1.

2-3 tree construction trace

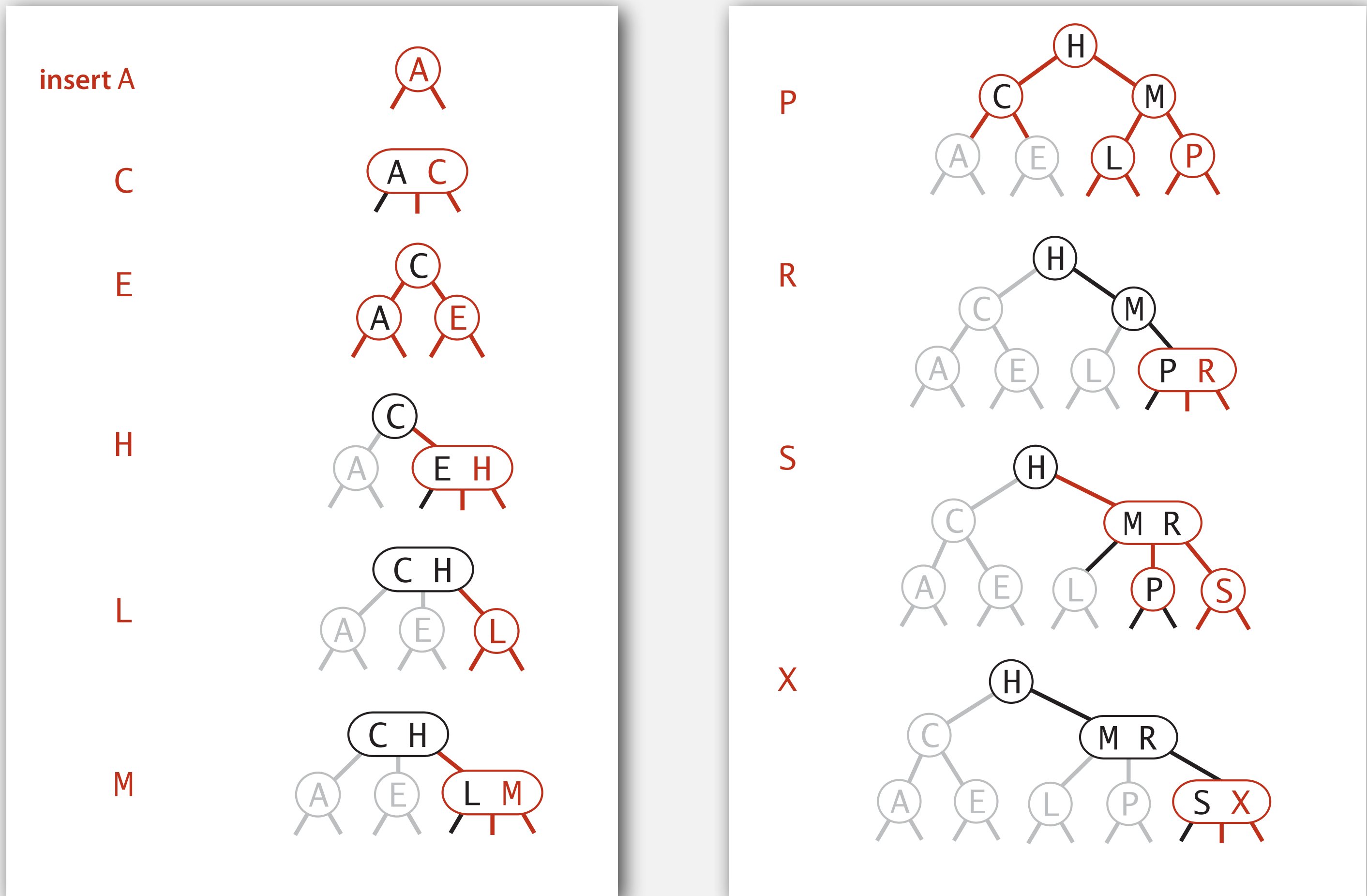
Standard indexing client.





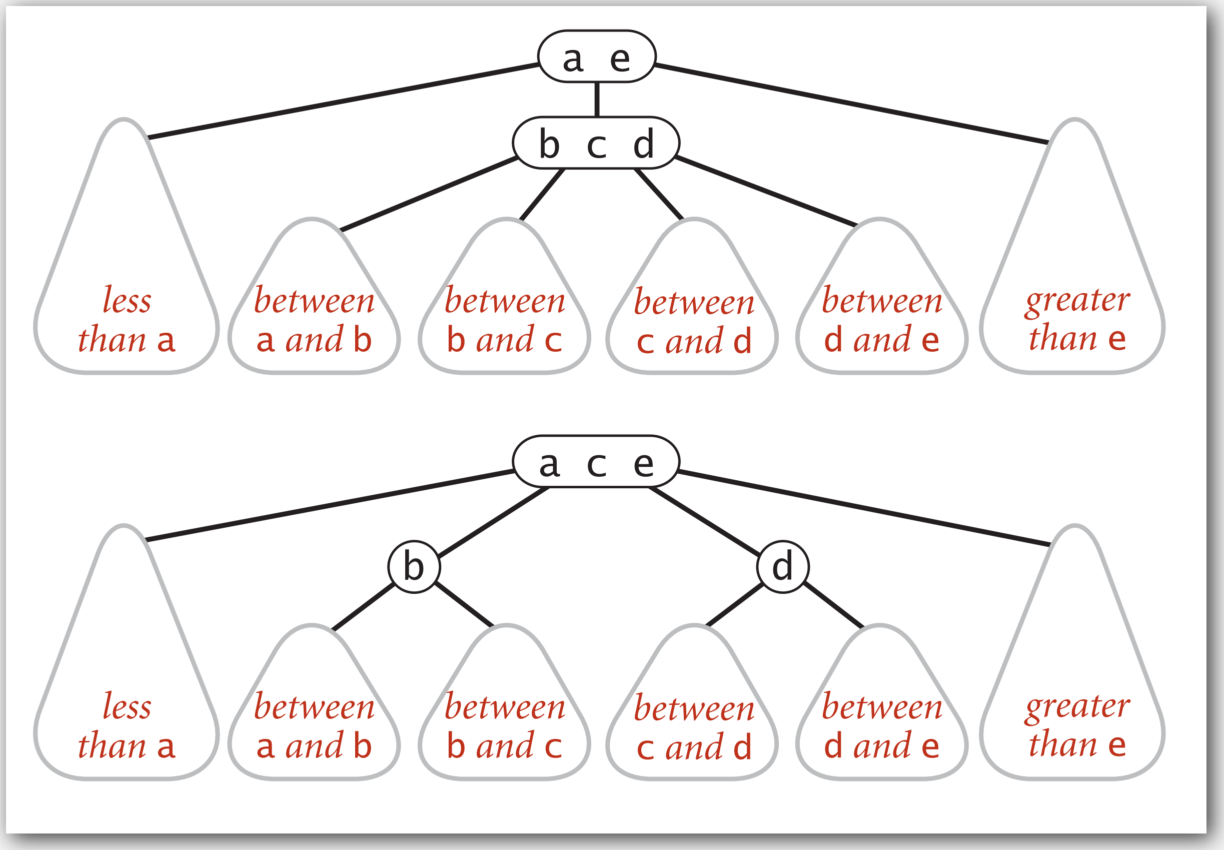
2-3 tree construction trace

The same keys inserted in ascending order.



Local transformations in a 2-3 tree

Splitting a 4-node is a local transformation: constant number of operations.

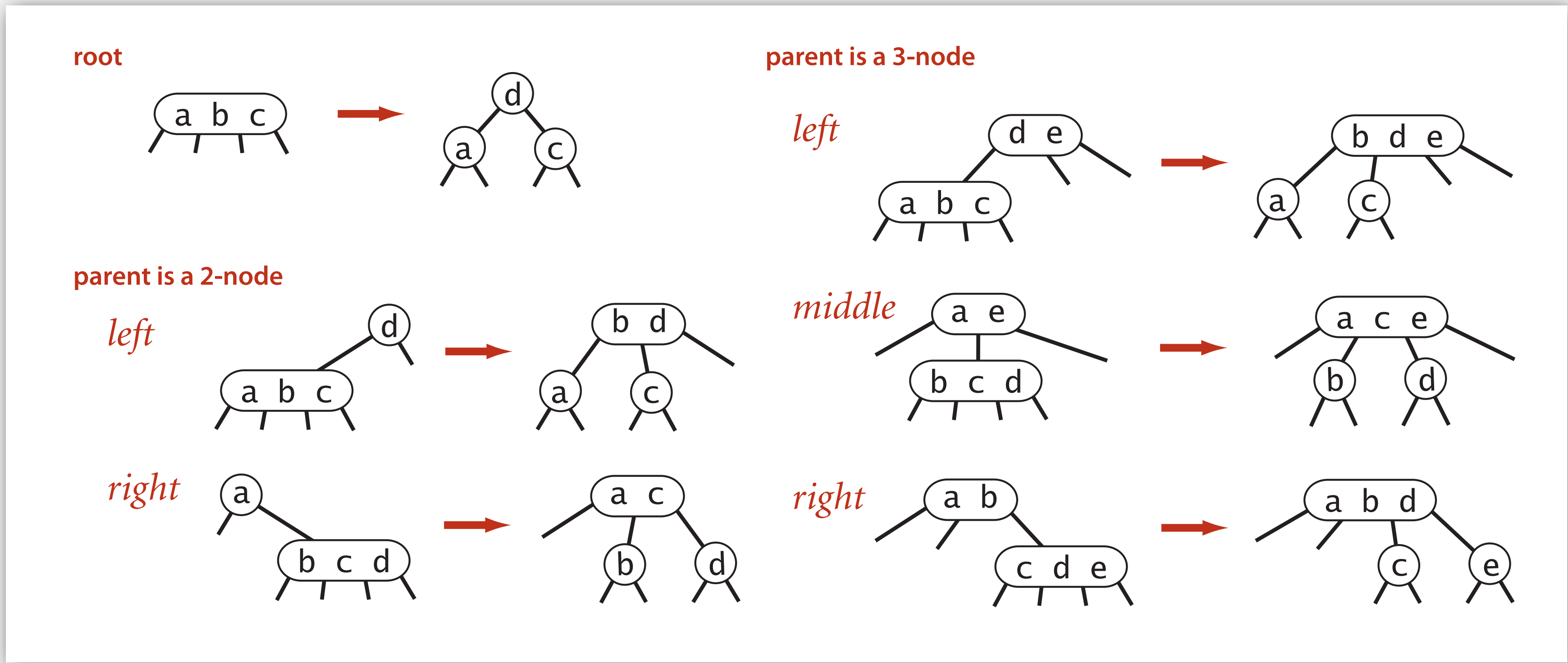


Global properties in a 2-3 tree

Invariant. Symmetric order.

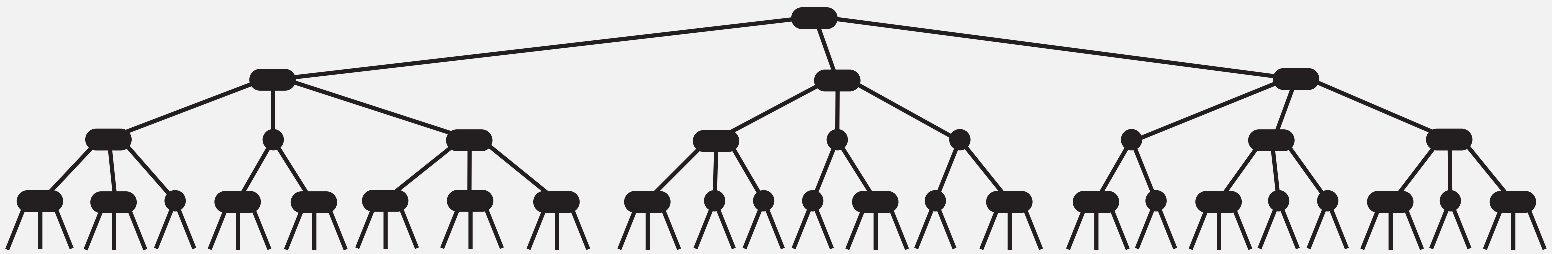
Invariant. Perfect balance.

Pf. Each transformation maintains order and balance.



2-3 tree: performance

Perfect balance. Every path from root to null link has same length.

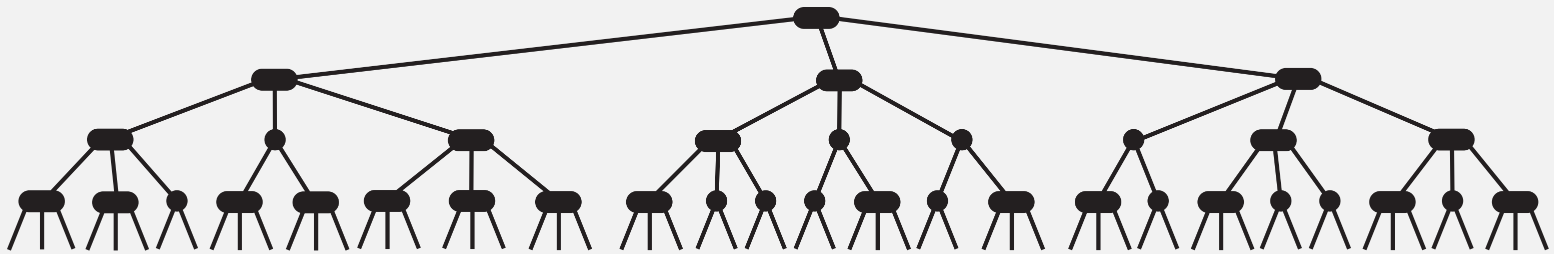


Tree height.

- Worst case:
- Best case:

2-3 tree: performance

Perfect balance. Every path from root to null link has same length.



Tree height.

- Worst case: lg N. [all 2-nodes]
- Best case: $\log_3 N \approx .631 \lg N$. [all 3-nodes]
- Between 12 and 20 for a million nodes.
- Between 18 and 30 for a billion nodes.

Guaranteed logarithmic performance for search and insert.

ST implementations: summary

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes	`compareTo()`
2-3 tree	c lg N	c lg N	c lg N	c lg N	c lg N	c lg N	yes	`compareTo()`

constants depend upon implementation

2-3 tree: implementation?

Direct implementation is complicated, because:

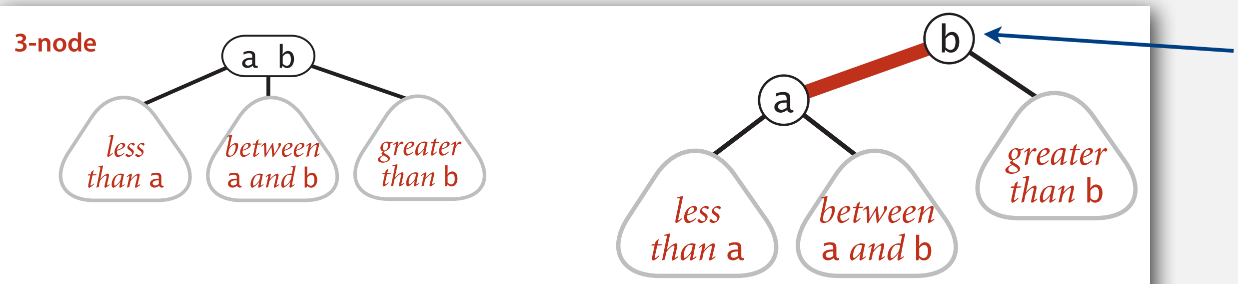
- Maintaining multiple node types is cumbersome.
- Need multiple compares to move down tree.
- Need to move back up the tree to split 4-nodes.
- Large number of cases for splitting.

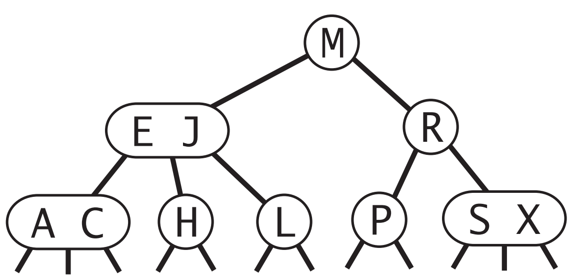
Bottom line. Could do it, but there's a better way.

- 2-3-4 trees
- red-black trees
- B-trees

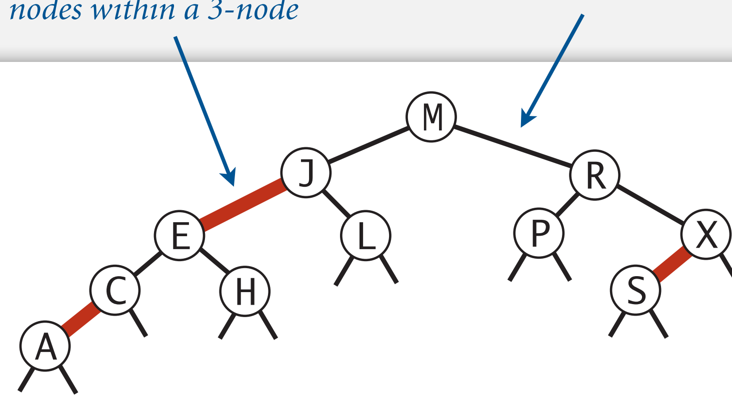
Left-leaning red-black trees (Guibas-Sedgewick 1979 and Sedgewick 2007)

1. Represent 2–3 tree as a BST.
2. Use "internal" left-leaning links as "glue" for 3–nodes.





2-3 tree



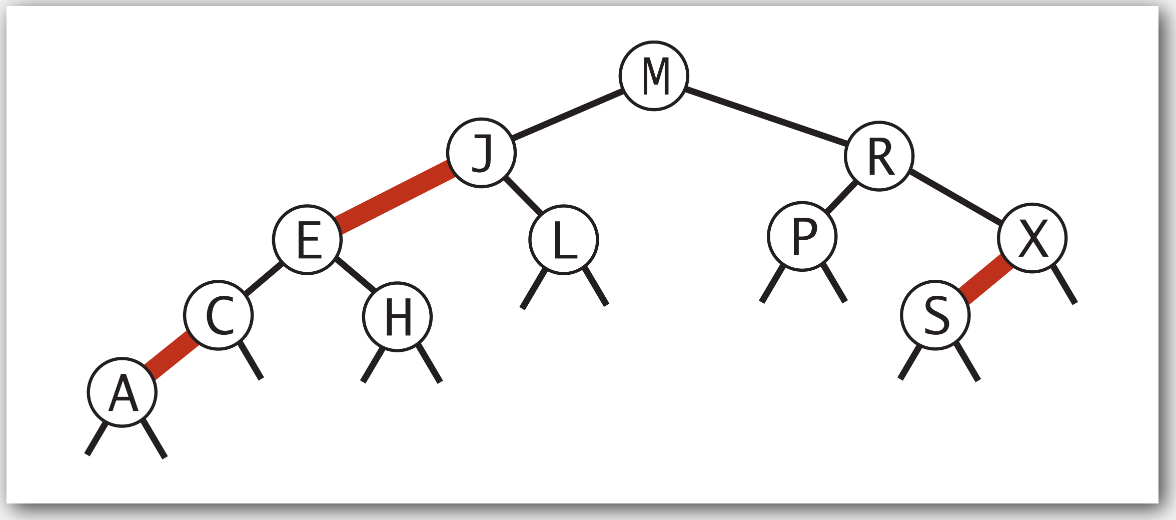
red-black tree

An equivalent definition

A BST such that:

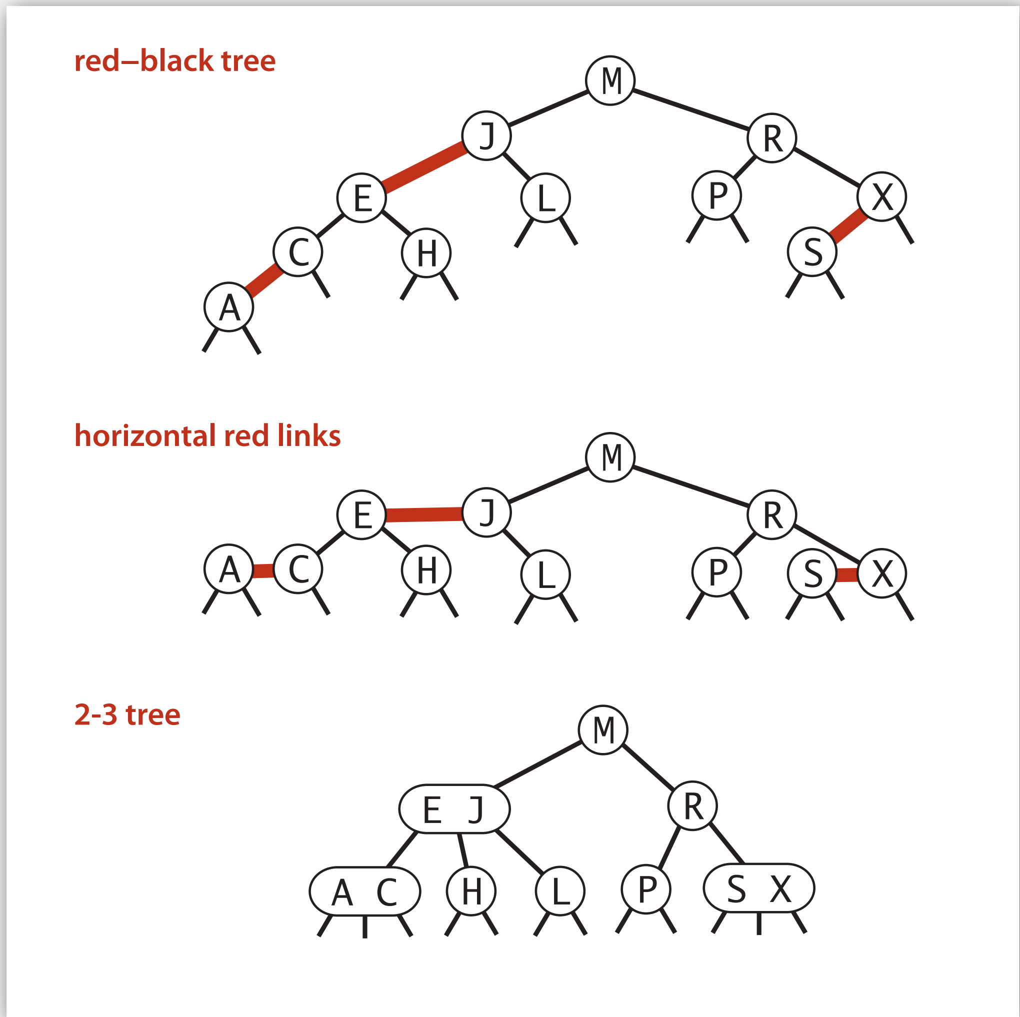
- No node has two red links connected to it.
- Every path from root to null link has the same number of black links.
- Red links lean left.

"perfect black balance"



Left-leaning red-black trees: 1-1 correspondence with 2-3 trees

Key property. 1–1 correspondence between 2–3 and LLRB.

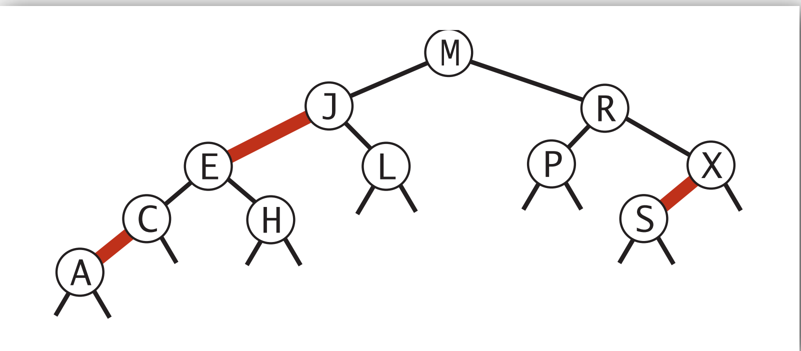


Search implementation for red-black trees

Observation. Search is the same as for elementary BST (ignore color).

but runs faster because of better balance

```
public Val get(Key key)
{
   Node x = root;
   while (x != null)
   {
      int cmp = key.compareTo(x.key);
      if       (cmp  < 0) x = x.left;
      else if (cmp  > 0) x = x.right;
      else if (cmp == 0) return x.val;
   }
   return null;
}
```



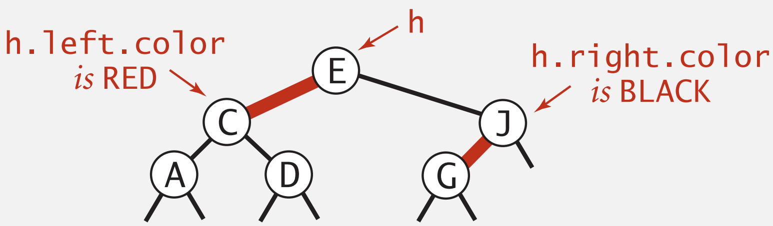
Remark. Many other ops (e.g., ceiling, selection, iteration) are also identical.

Red-black tree representation

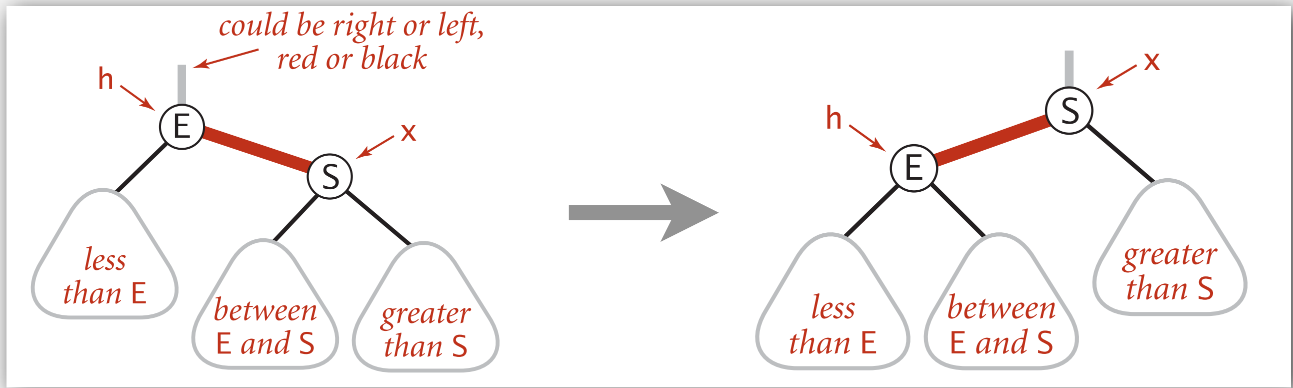
Each node is pointed to by precisely one link (from its parent) ⇒ can encode color of links in nodes.

```
private static final boolean RED   = true;
private static final boolean BLACK = false;

private class Node
{
   Key key;
   Value val;
   Node left, right;
   boolean color;   // color of parent link
}

private boolean isRed(Node x)
{
   if (x == null) return false;
   return x.color == RED;
}
```

null links are black



Left rotation. Orient a (temporarily) right-leaning red link to lean left.

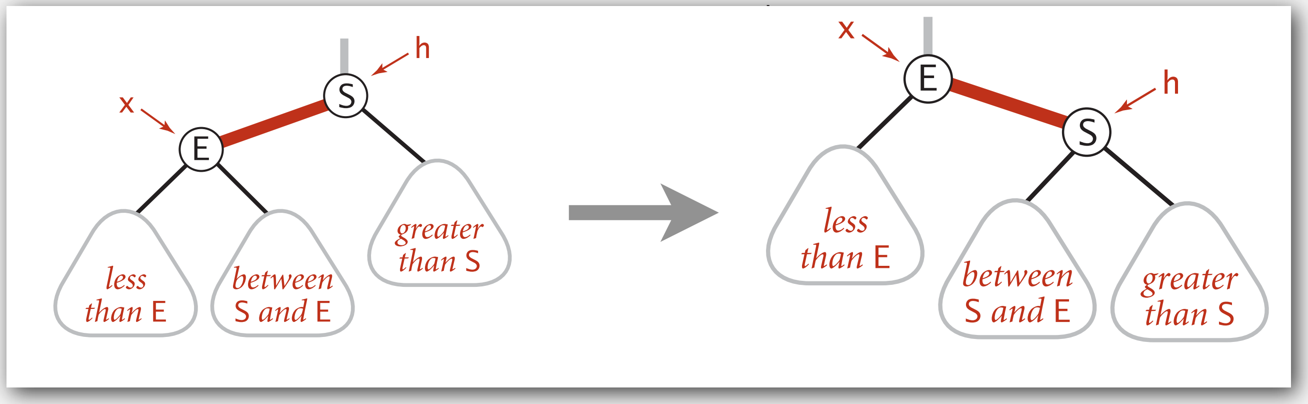



```
private Node rotateLeft(Node h)
{
   assert (h != null) && isRed(h.right);
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black tree operations

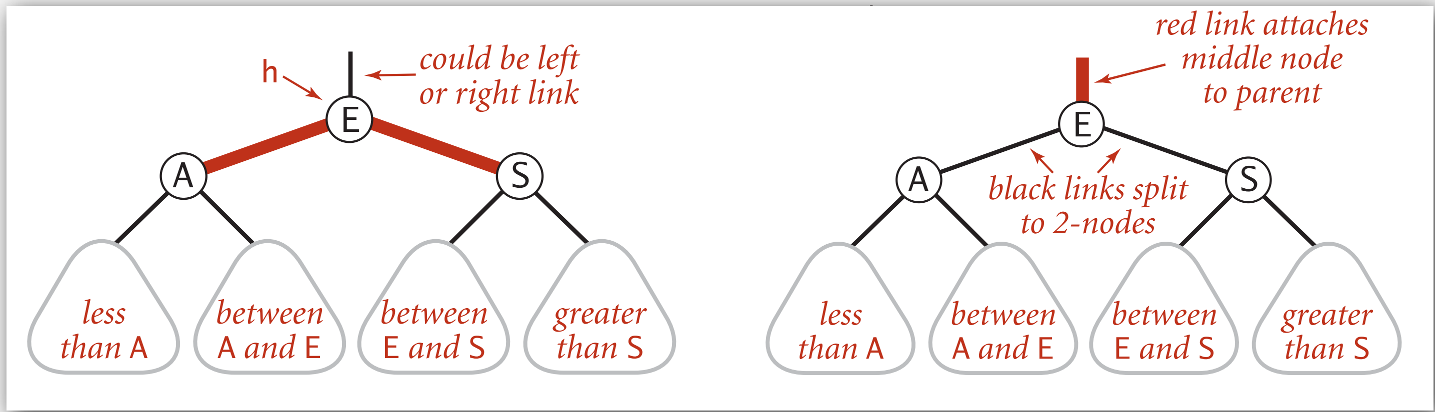
Right rotation. Orient a left-leaning red link to (temporarily) lean right.




```
private Node rotateRight(Node h)
{
   assert (h != null) && isRed(h.left);
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Color flip. Recolor to split a (temporary) 4-node.

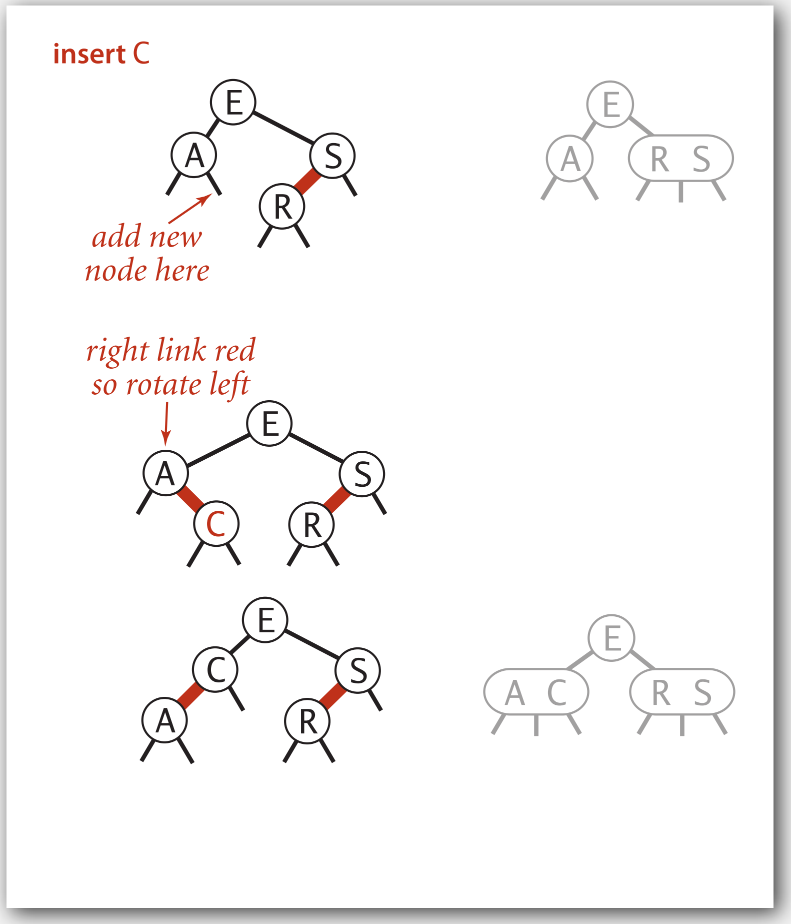



```
private void flipColors(Node h)
{
   assert !isRed(h) && isRed(h.left) && isRed(h.right);

   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.

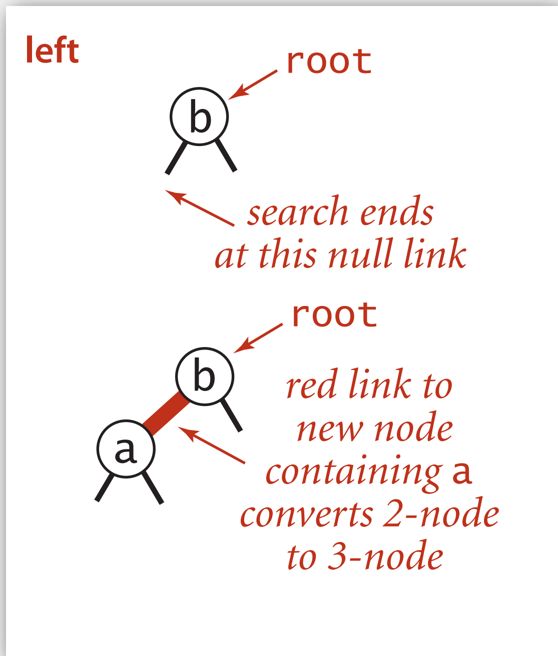
Insertion in a LLRB tree: overview

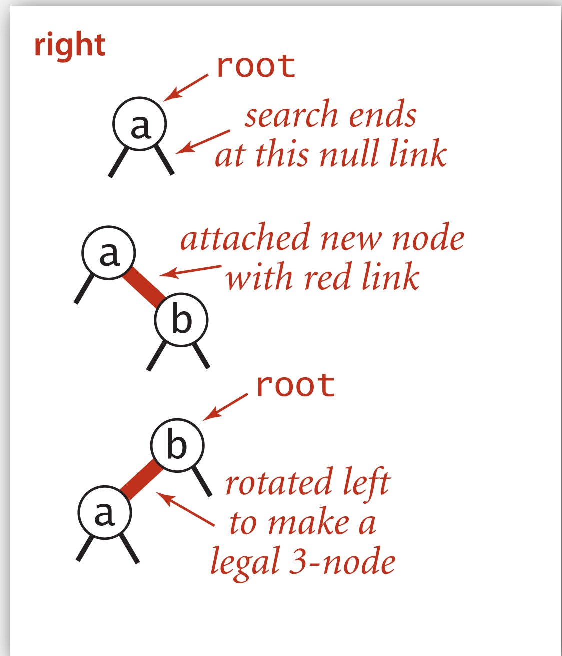
Basic strategy. Maintain 1-1 correspondence with 2-3 trees by applying elementary red-black tree operations



Insertion in a LLRB tree

Warmup 1. Insert into a tree with exactly 1 node.

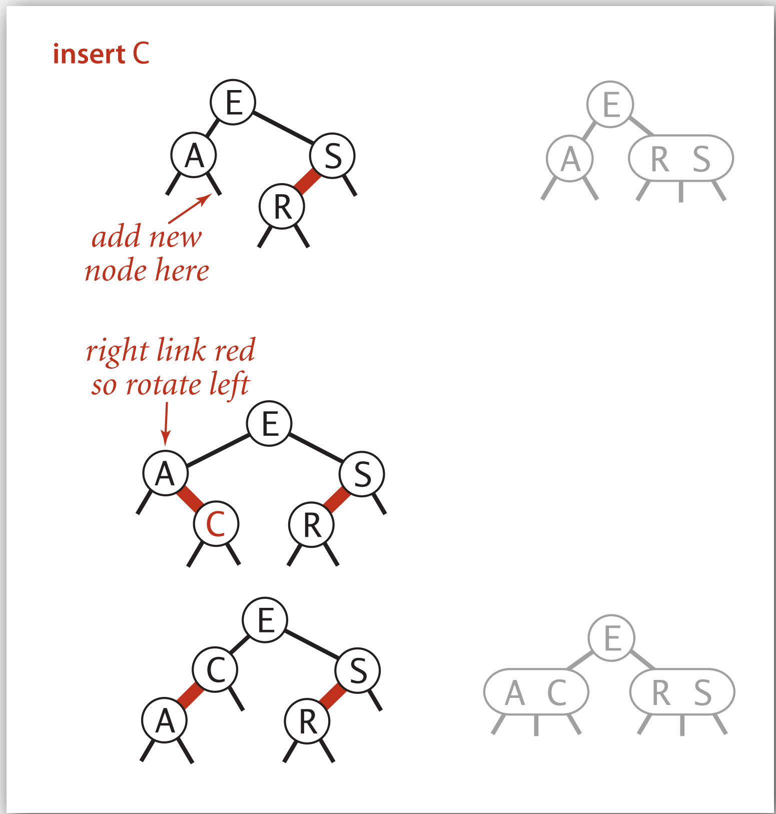




Insertion in a LLRB tree

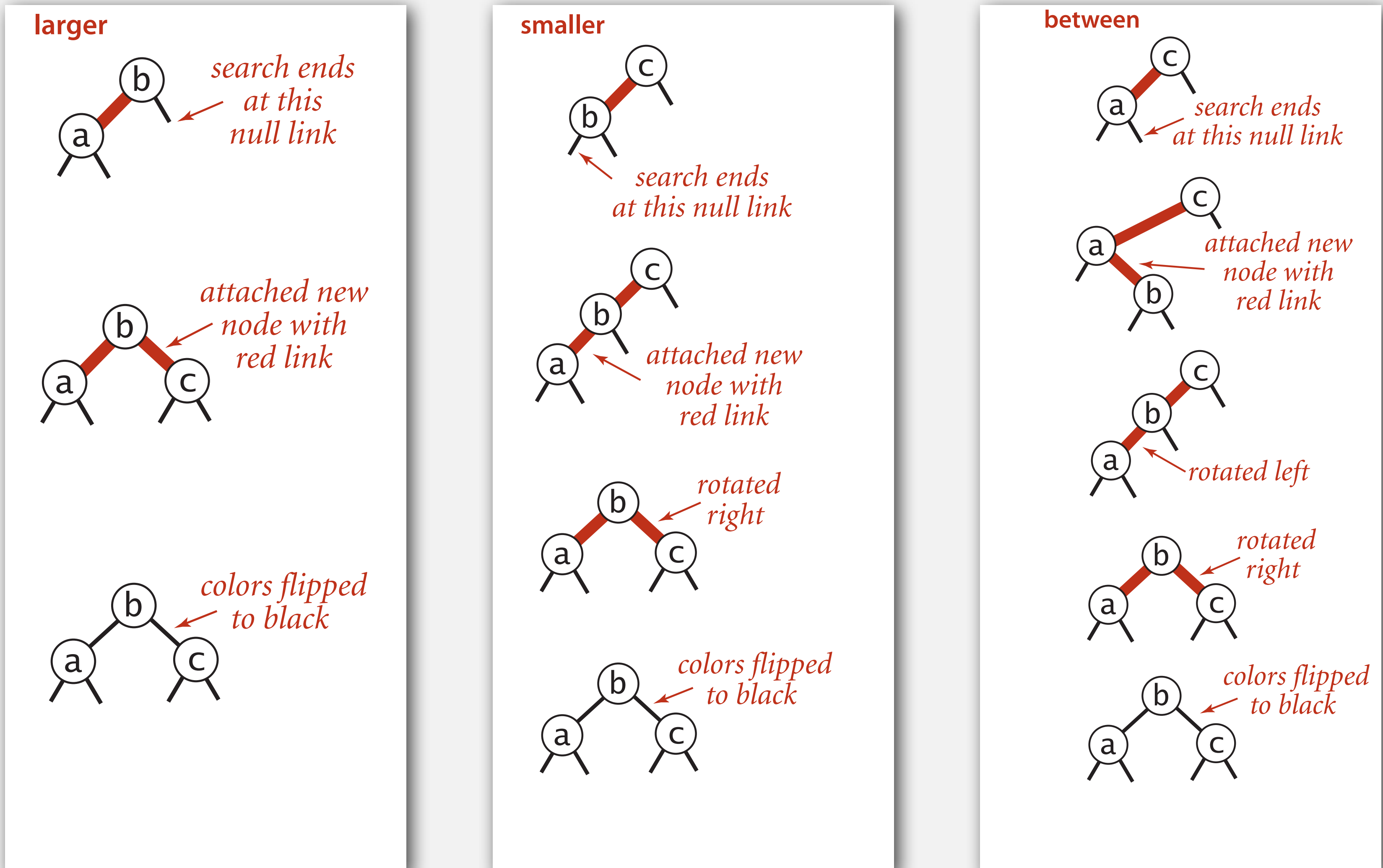
Case 1. Insert into a 2-node at the bottom.

- Do standard BST insert; color new link red.
- If new red link is a right link, rotate left.



Insertion in a LLRB tree

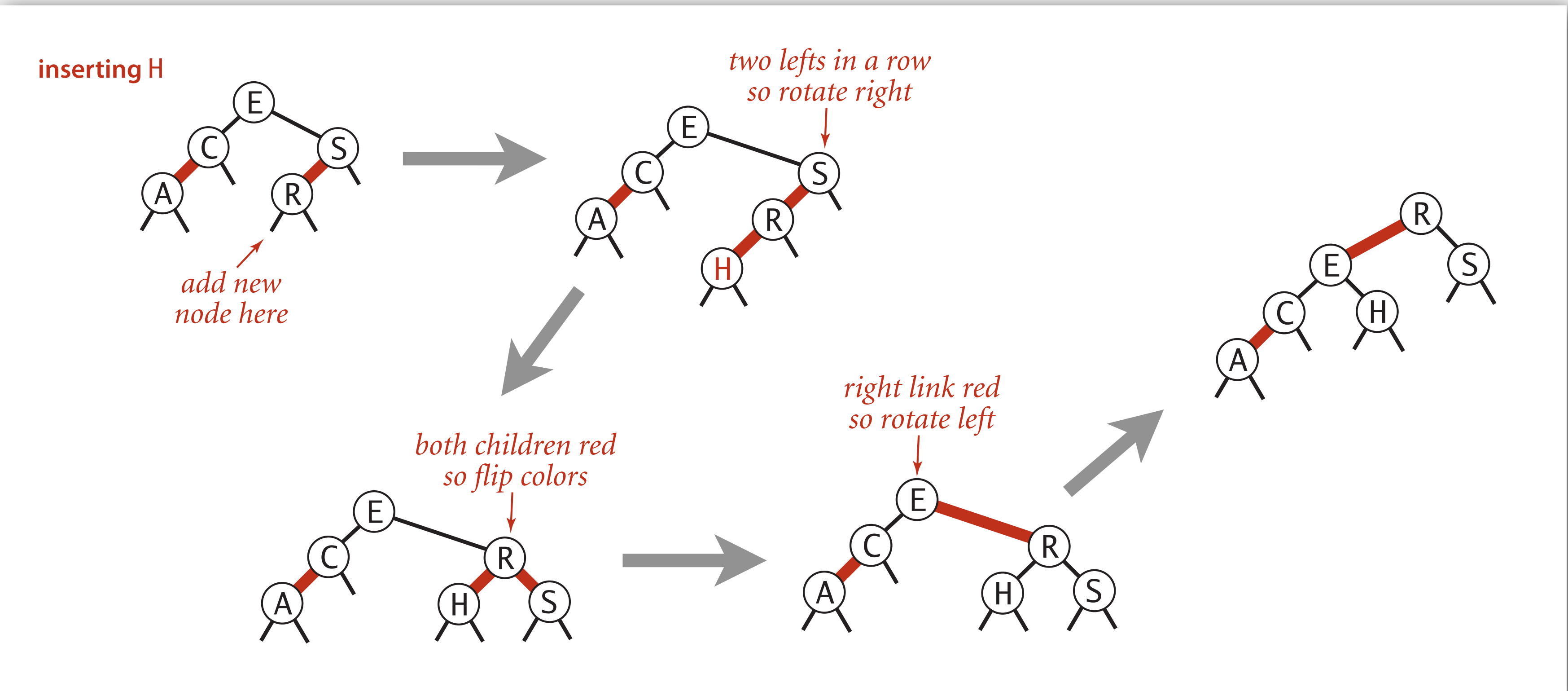
Warmup 2. Insert into a tree with exactly 2 nodes.



Insertion in a LLRB tree

Case 2. Insert into a 3-node at the bottom.

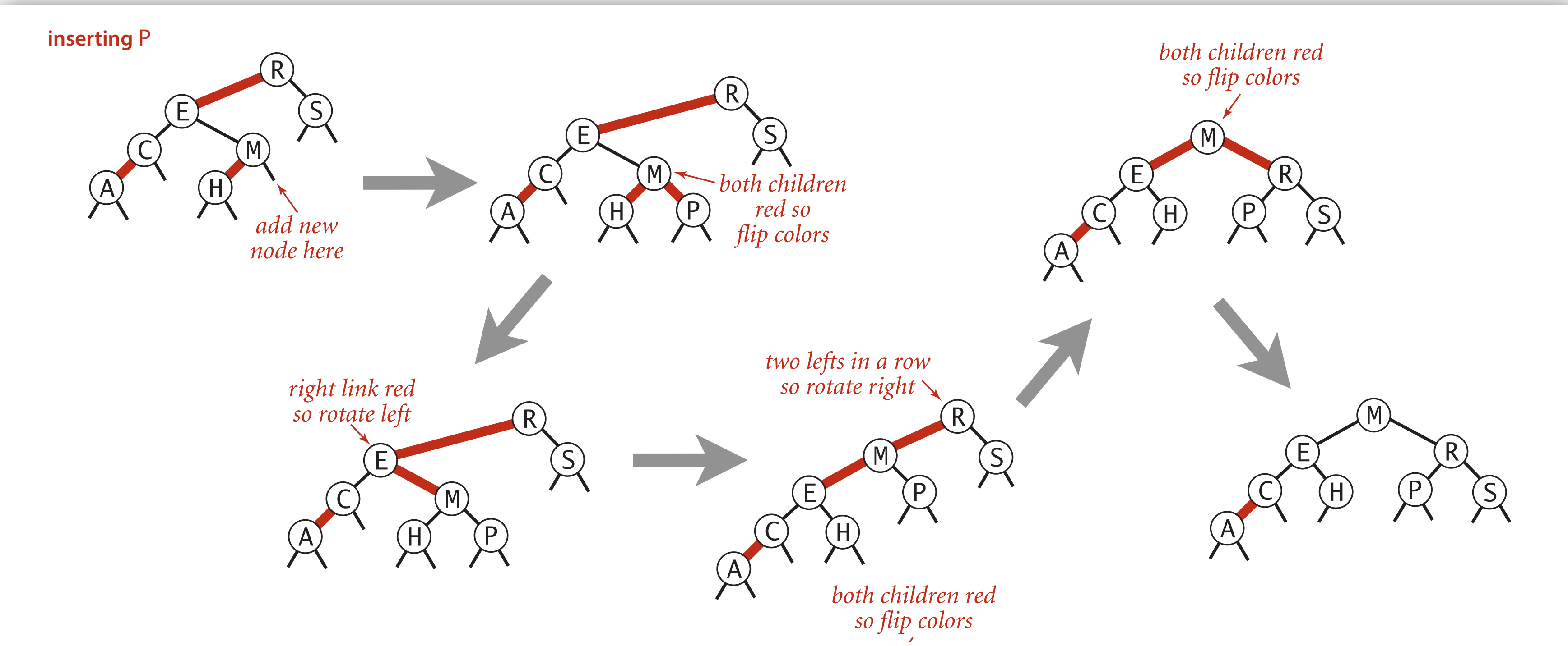
- Do standard BST insert; color new link red.
- Rotate to balance the 4-node (if needed).
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed).



Insertion in a LLRB tree: passing red links up the tree

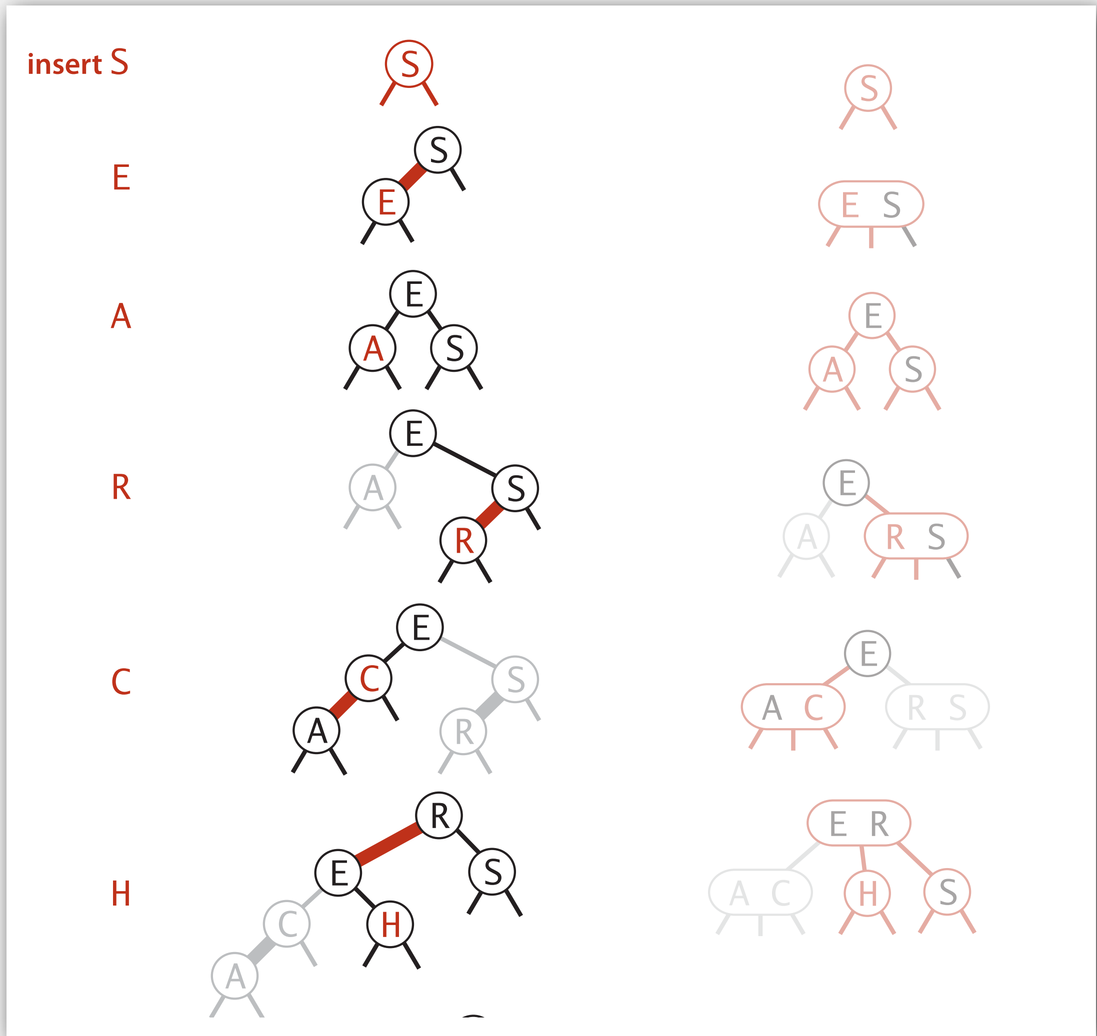
Case 2. Insert into a 3-node at the bottom.

- Do standard BST insert; color new link red.
- Rotate to balance the 4-node (if needed).
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed).
- Repeat Case 1 or Case 2 up the tree (if needed).



LLRB tree construction trace

Standard indexing client.

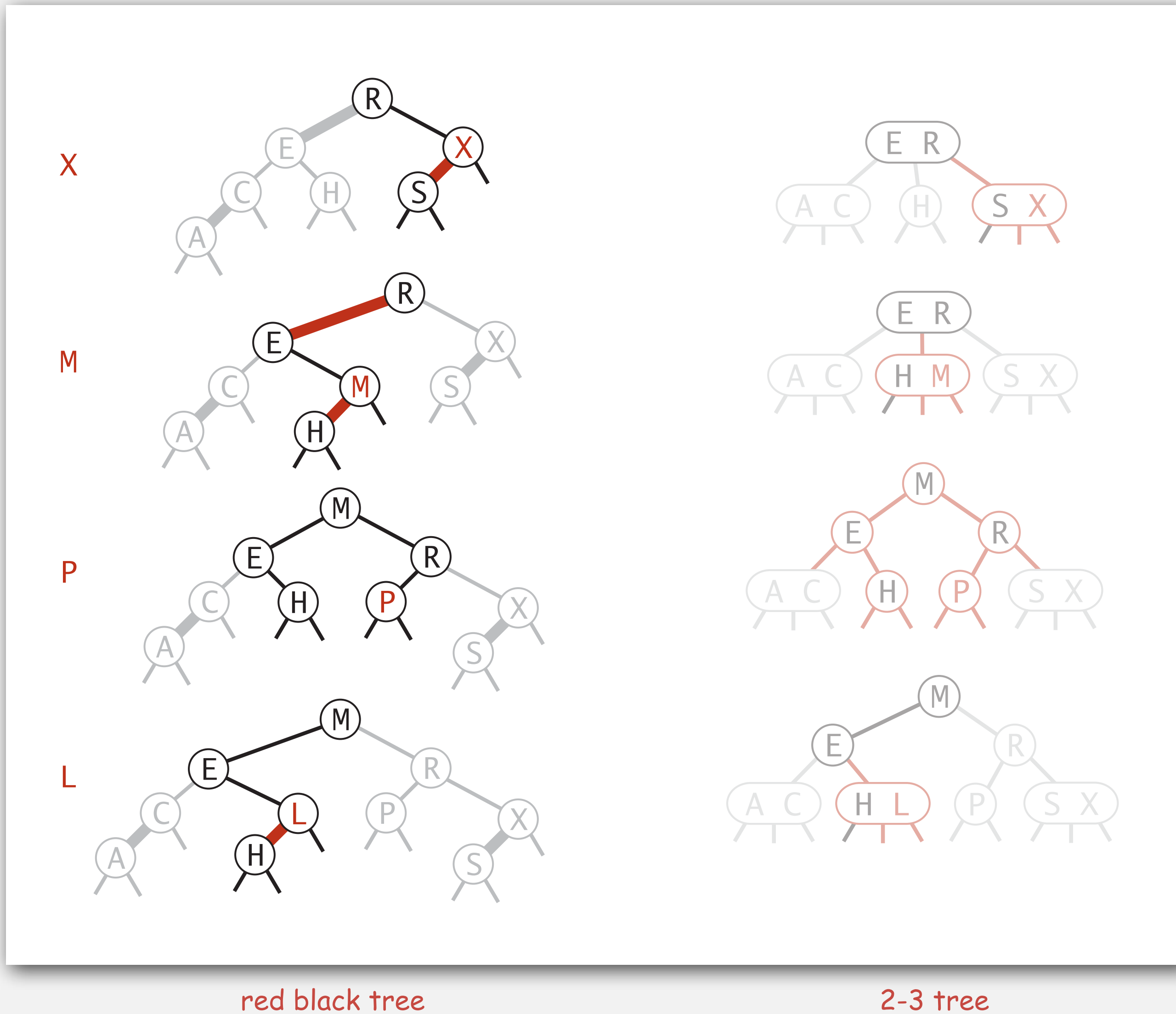


red black tree

2-3 tree

LLRB tree construction trace

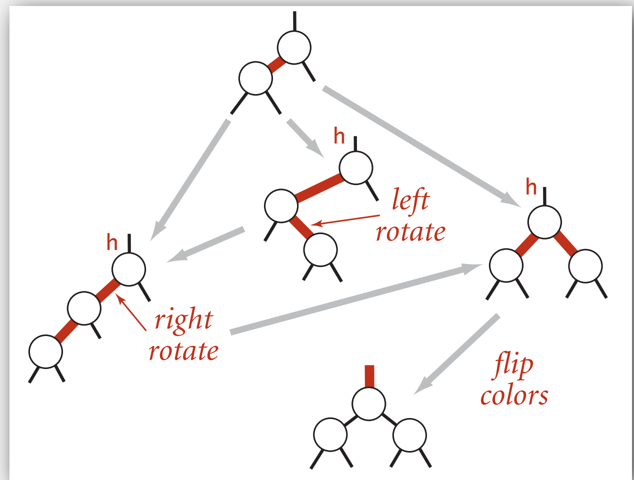
Standard indexing client (continued).



Insertion in a LLRB tree: Java implementation

Same code for both cases.

- Right child red, left child black: rotate left.
- Left child, left-left grandchild red: rotate right.
- Both children red: flip colors.




```
private Node put(Node h, Key key, Value val)
{
   if (h == null) return new Node(key, val, RED);
   int cmp = key.compareTo(h.key);
   if      (cmp < 0) h.left  = put(h.left,  key, val);
   else if (cmp > 0) h.right = put(h.right, key, val);
   else h.val = val;

   if (isRed(h.right) && !isRed(h.left))      h = rotateLeft(h);
   if (isRed(h.left)  && isRed(h.left.left)) h = rotateRight(h);
   if (isRed(h.left)  && isRed(h.right))      h = flipColors(h);

   return h;
}
```

insert at bottom

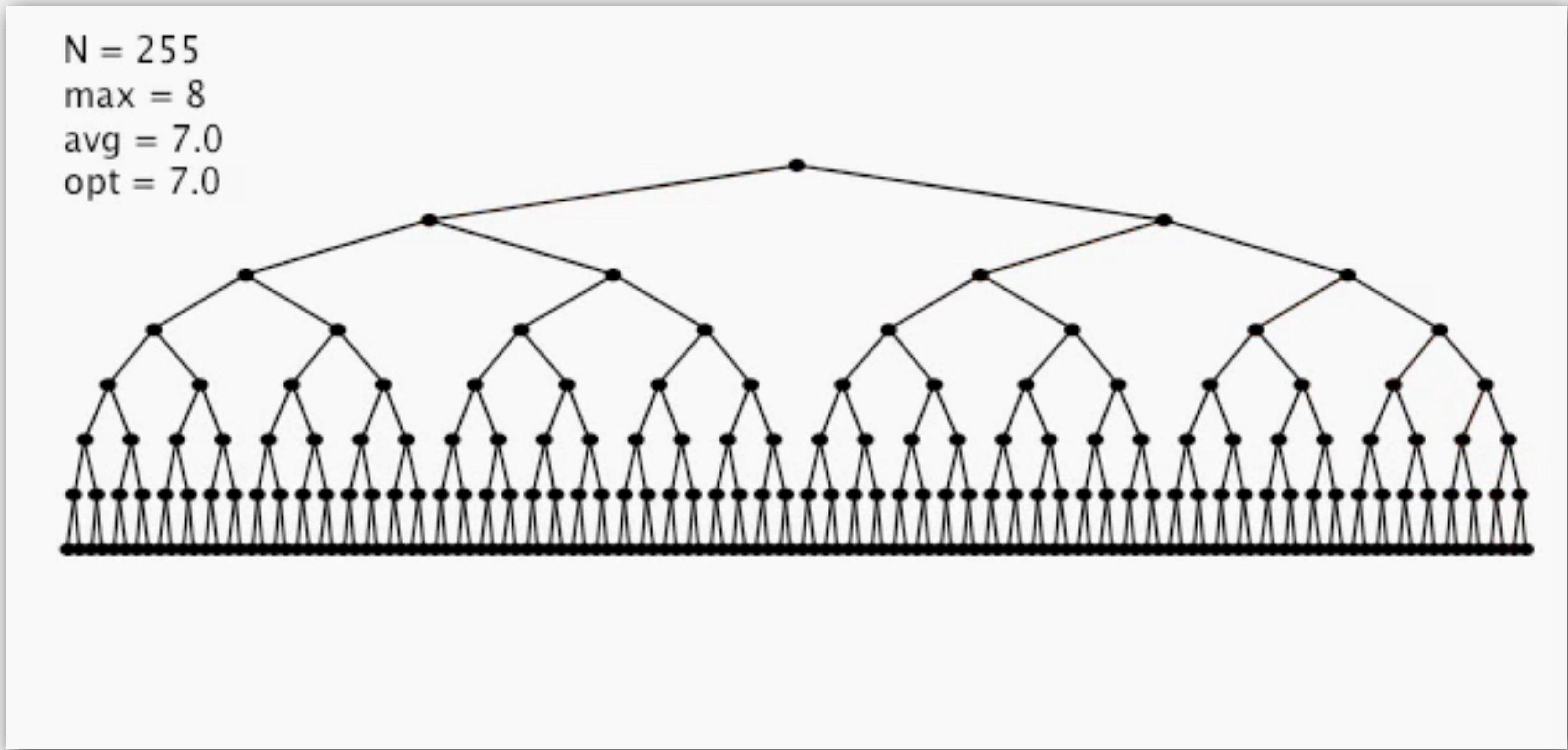
lean left

balance 4-node

split 4-node

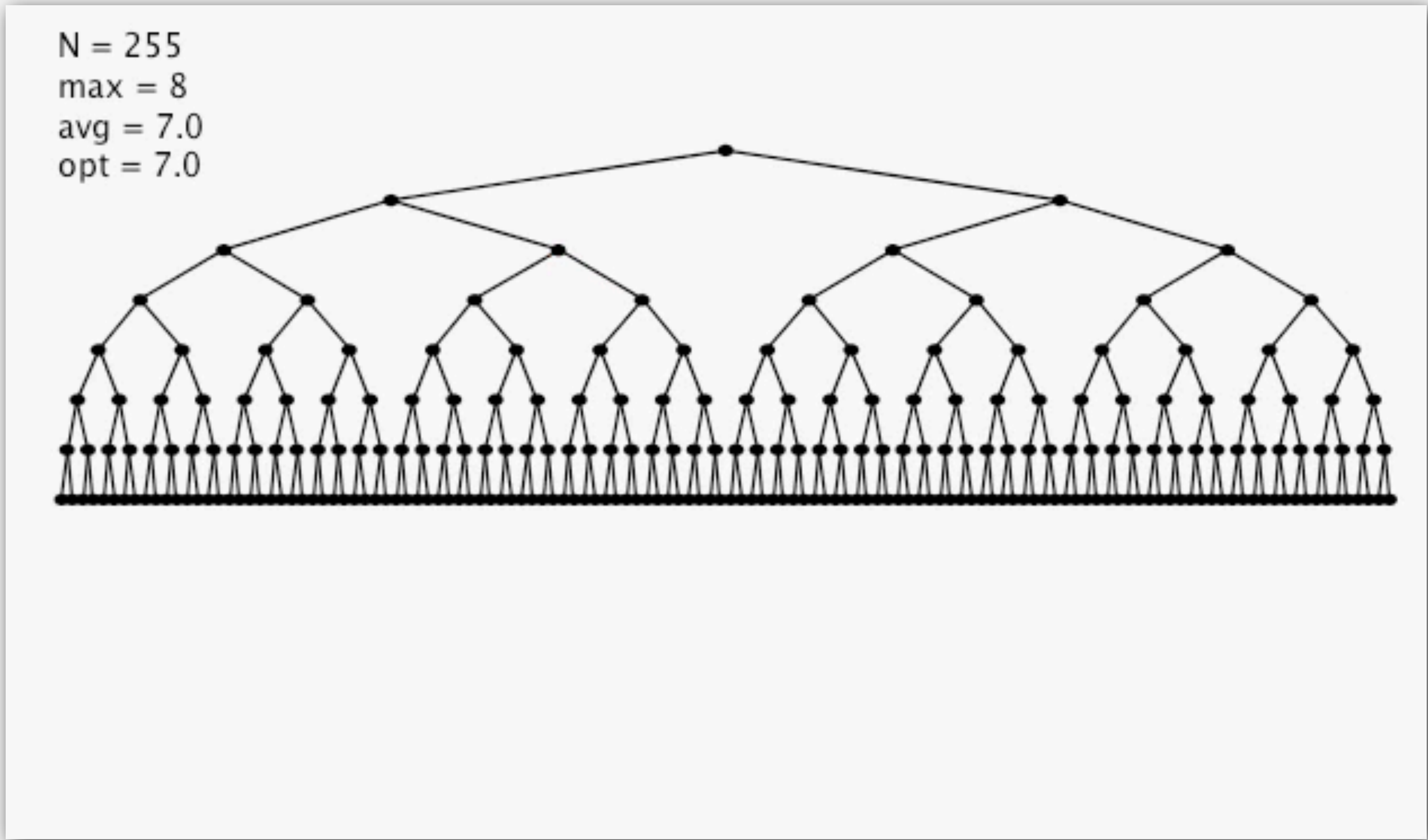
only a few extra lines of code
to provide near-perfect balance

Insertion in a LLRB tree: visualization



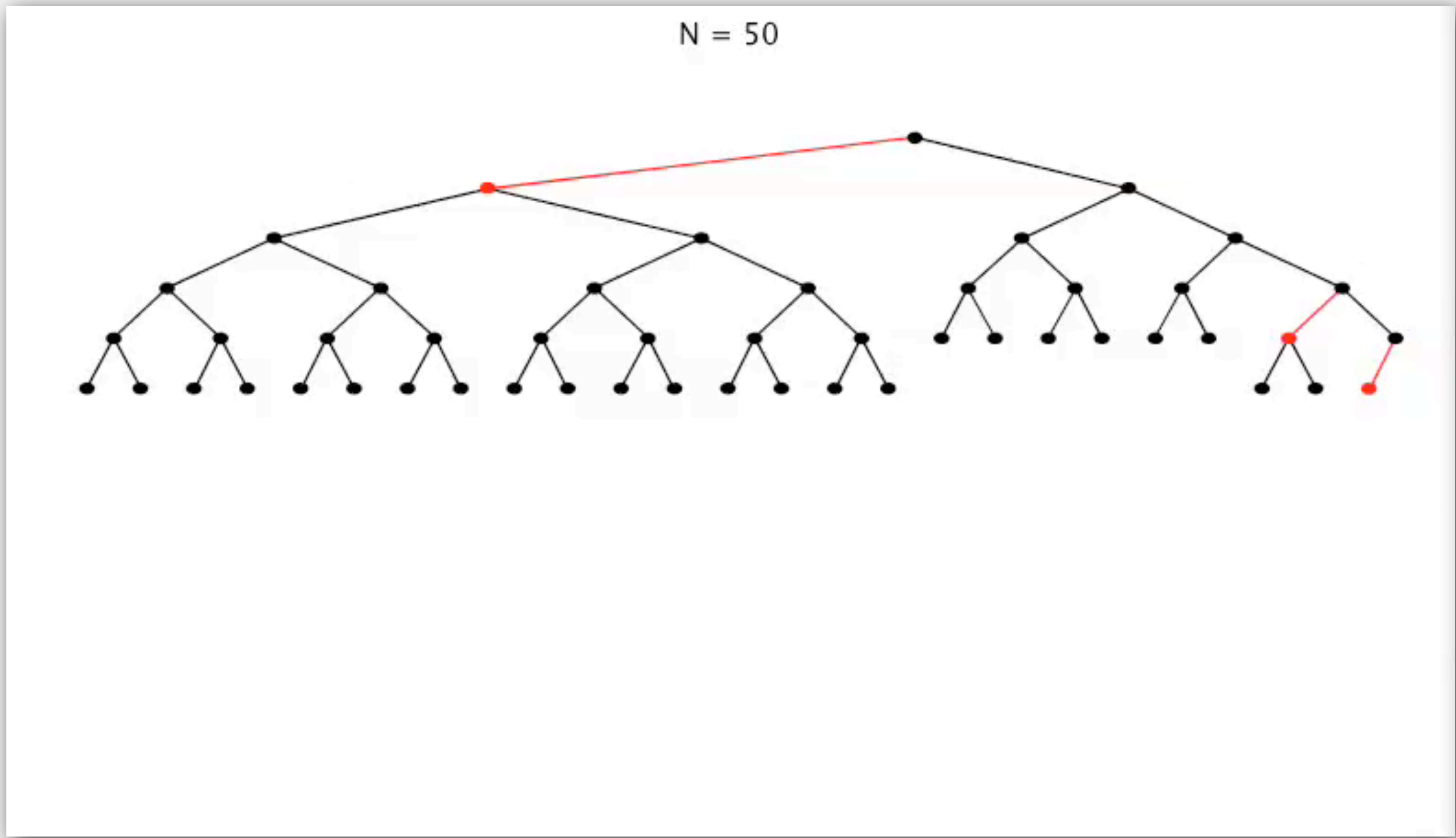
255 insertions in ascending order

Insertion in a LLRB tree: visualization



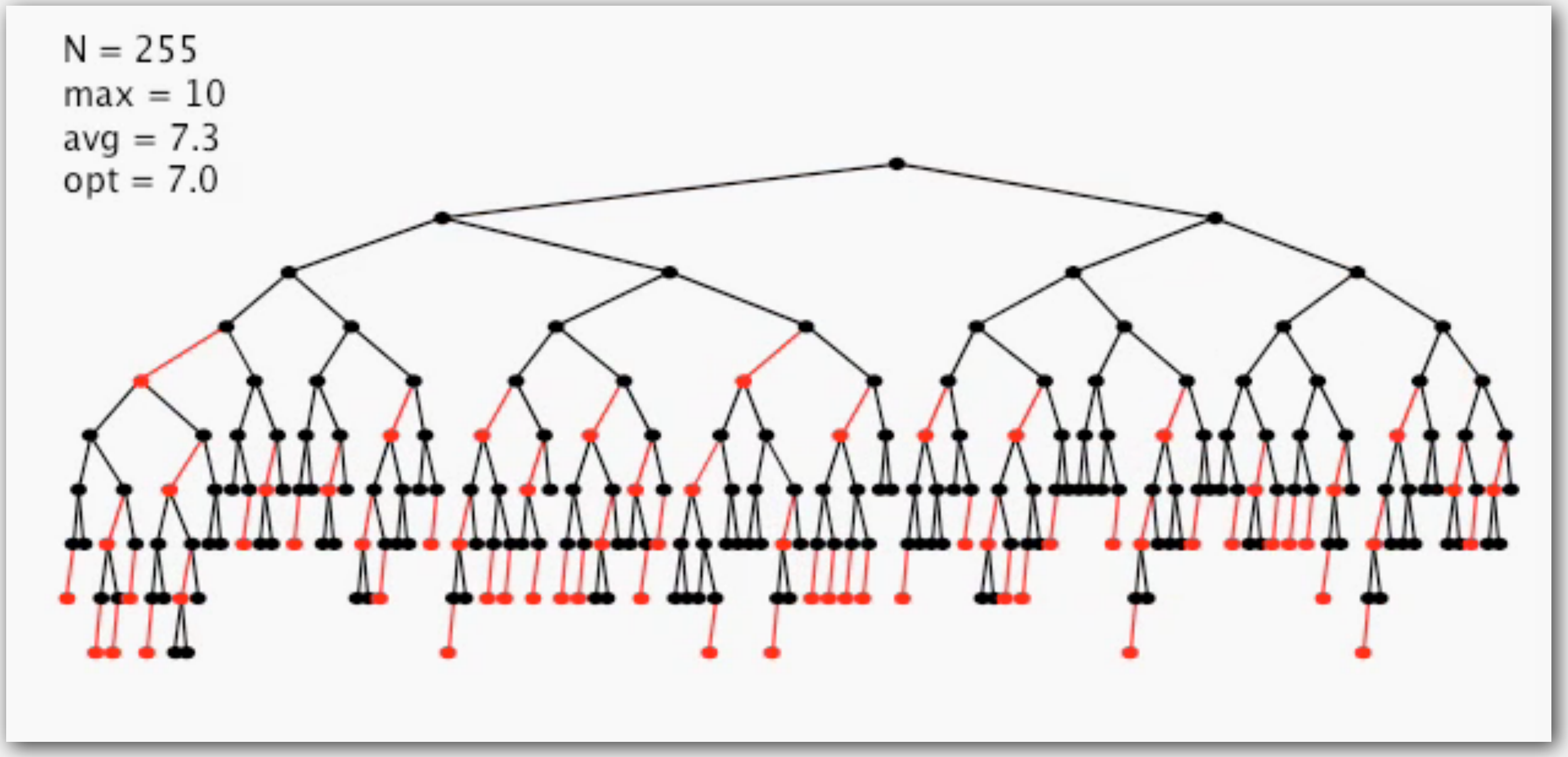
255 insertions in descending order

Insertion in a LLRB tree: visualization



50 random insertions

Insertion in a LLRB tree: visualization



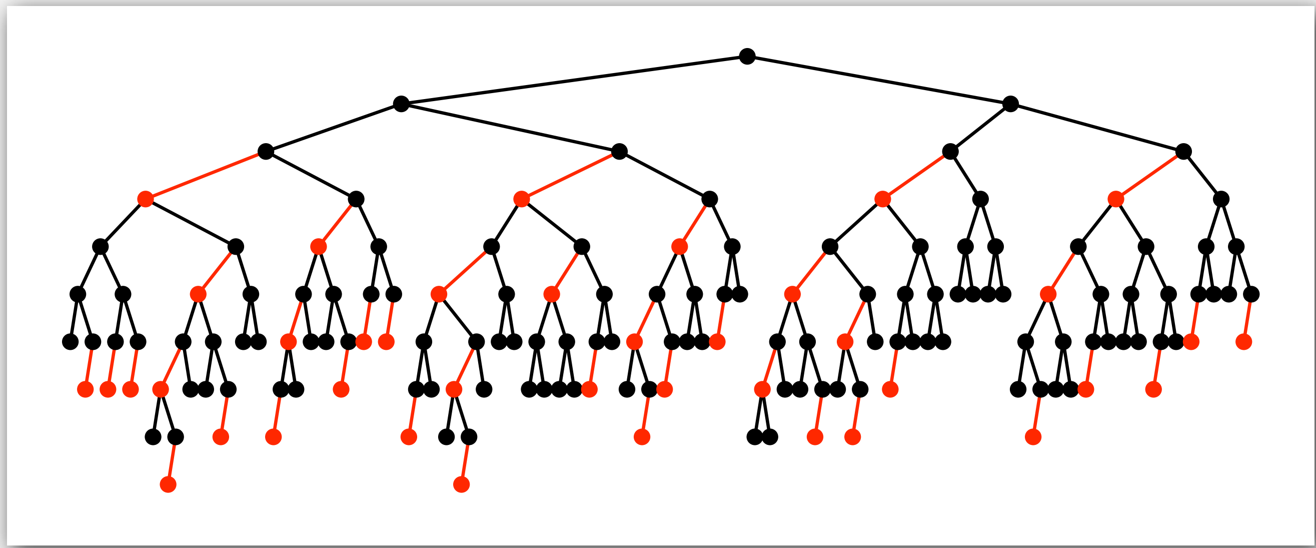
255 random insertions

Balance in LLRB trees

Proposition. Height of tree is ≤ 2 lg N in the worst case.

Pf.

- Every path from root to null link has same number of black links.
- Never two red links in-a-row.

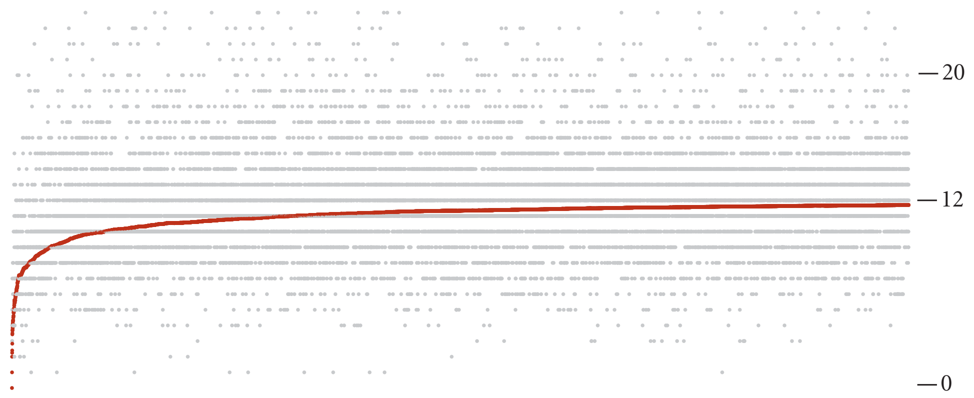


Property. Height of tree is ~ 1.00 lg N in typical applications.

ST implementations: summary

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes	`compareTo()`
2-3 tree	c lg N	c lg N	c lg N	c lg N	c lg N	c lg N	yes	`compareTo()`
red-black tree	2 lg N	2 lg N	2 lg N	1.00 lg N *	1.00 lg N *	1.00 lg N *	yes	`compareTo()`

* exact value of coefficient unknown but extremely close to 1



Costs for `java FrequencyCounter 8 < tale.txt` using `RedBlackBST`

Why left-leaning trees?

old code (that students had to learn in the past)

```
private Node put(Node x, Key key, Value val, boolean sw)
{
   if (x == null)
      return new Node(key, value, RED);
   int cmp = key.compareTo(x.key);

   if (isRed(x.left) && isRed(x.right))
   {
      x.color = RED;
      x.left.color  = BLACK;
      x.right.color = BLACK;
   }
   if (cmp < 0)
   {
      x.left = put(x.left, key, val, false);
      if (isRed(x) && isRed(x.left) && sw)
         x = rotateRight(x);
      if (isRed(x.left) && isRed(x.left.left))
      {
         x = rotateRight(x);
         x.color = BLACK; x.right.color = RED;
      }
   }
   else if (cmp > 0)
   {
      x.right = put(x.right, key, val, true);
      if (isRed(h) && isRed(x.right) && !sw)
         x = rotateLeft(x);
      if (isRed(h.right) && isRed(h.right.right))
      {
         x = rotateLeft(x);
         x.color = BLACK; x.left.color = RED;
      }
   }
   else x.val = val;
   return x;
}
```

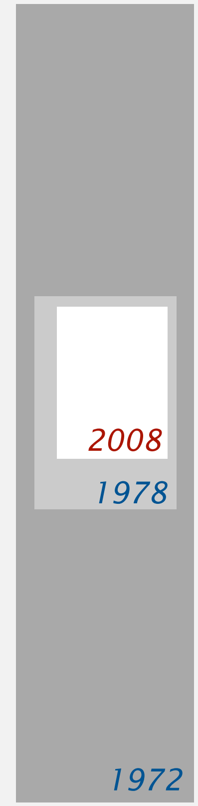
extremely tricky

new code (that you have to learn)

```
public Node put(Node h, Key key, Value val)
{
   if (h == null)
      return new Node(key, val, RED);
   int cmp = kery.compareTo(h.key);
   if (cmp < 0)
      h.left  = put(h.left,  key, val);
   else if (cmp > 0)
      h.right = put(h.right, key, val);
   else h.val = val;

   if (isRed(h.right) && !isRed(h.left))
      h = rotateLeft(h);
   if (isRed(h.left) && isRed(h.left.left))
      h = rotateRight(h);
   if (isRed(h.left) && isRed(h.right))
      h = flipColors(h);

  return h;
}
```

straightforward
(if you've paid attention)

Why left-leaning trees?



Simplified code.

- Left-leaning restriction reduces number of cases.
- Short inner loop.

Same ideas simplify implementation of other operations.

- Delete min/max.
- Arbitrary delete.

Improves widely-used algorithms.

- AVL trees, 2-3 trees, 2-3-4 trees.
- Red-black trees.

Bottom line. Left-leaning red-black trees are the simplest balanced BST to implement and the fastest in practice.

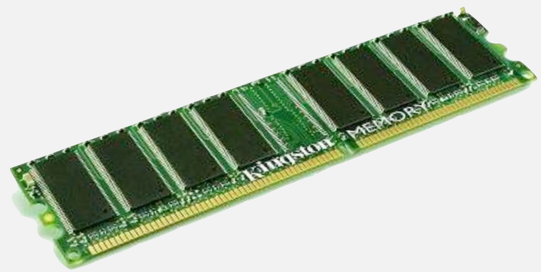
- 2-3-4 trees
- red-black trees
- B-trees

File system model

Page. Contiguous block of data (e.g., a file or 4096-byte chunk).

Probe. First access to a page (e.g., from disk to memory).

slow



fast

Model. Time required for a probe is much larger than time to access data within a page.

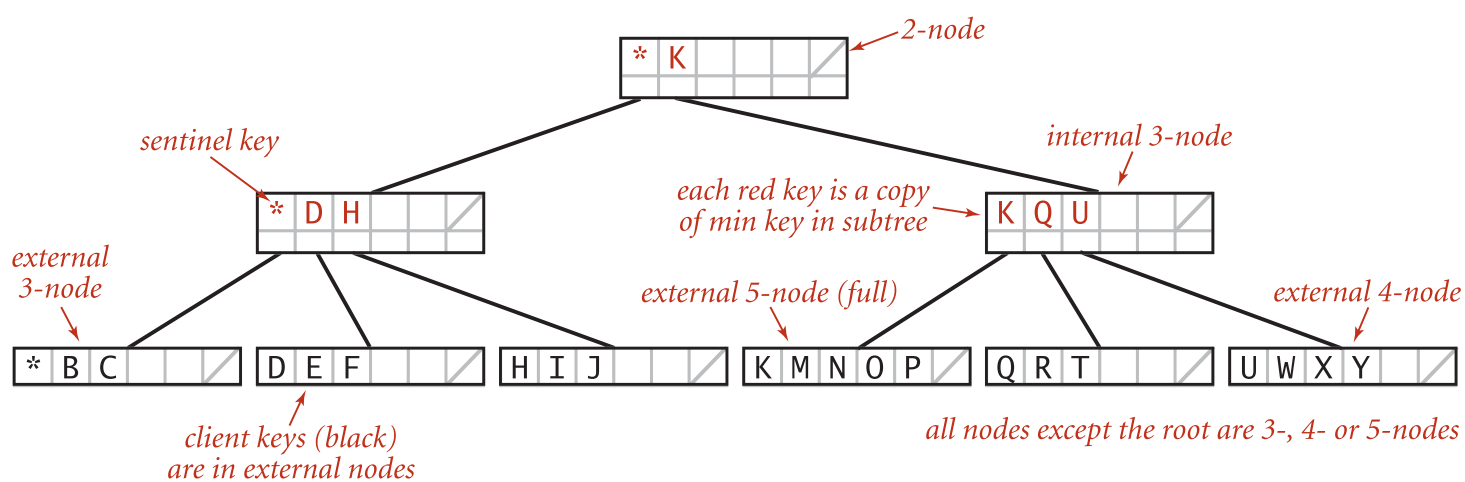
Goal. Access data using minimum number of probes.

B-trees (Bayer-McCreight, 1972)

B-tree. Generalize 2-3 trees by allowing up to M-1 key-link pairs per node.

- At least 2 key-link pairs at root.
- At least M/2 key-link pairs in other nodes.
- External nodes contain client keys.
- Internal nodes contain copies of keys to guide search.

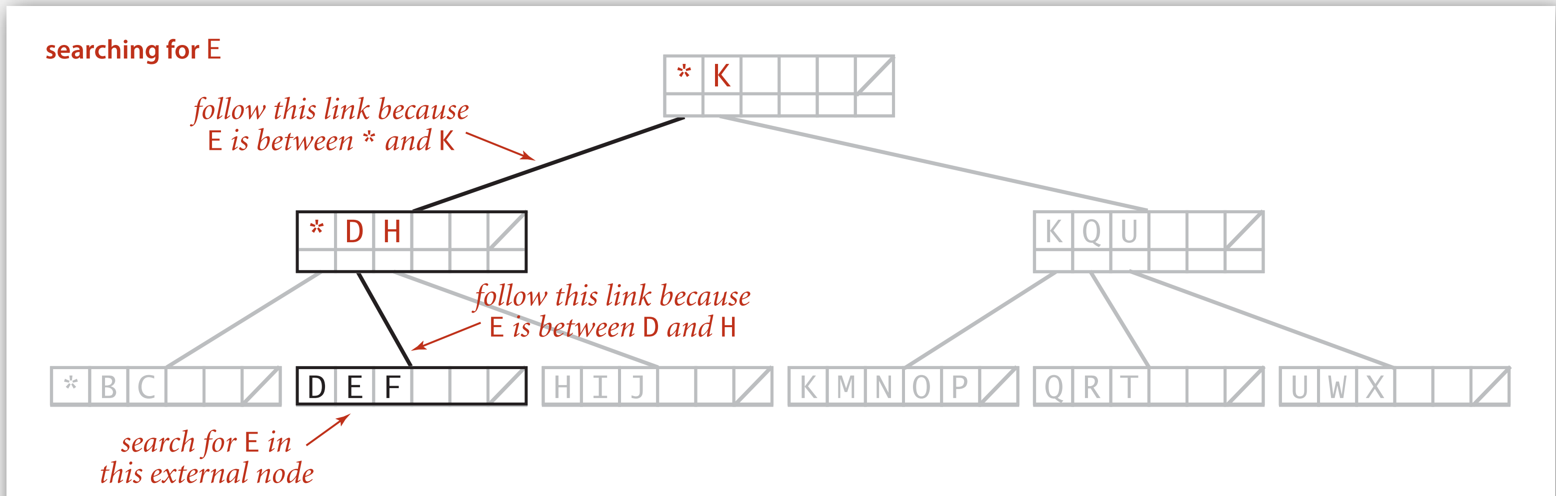
choose M as large as possible so that M links fit in a page, e.g., M = 1000



Anatomy of a B-tree set (M = 6)

Searching in a B-tree

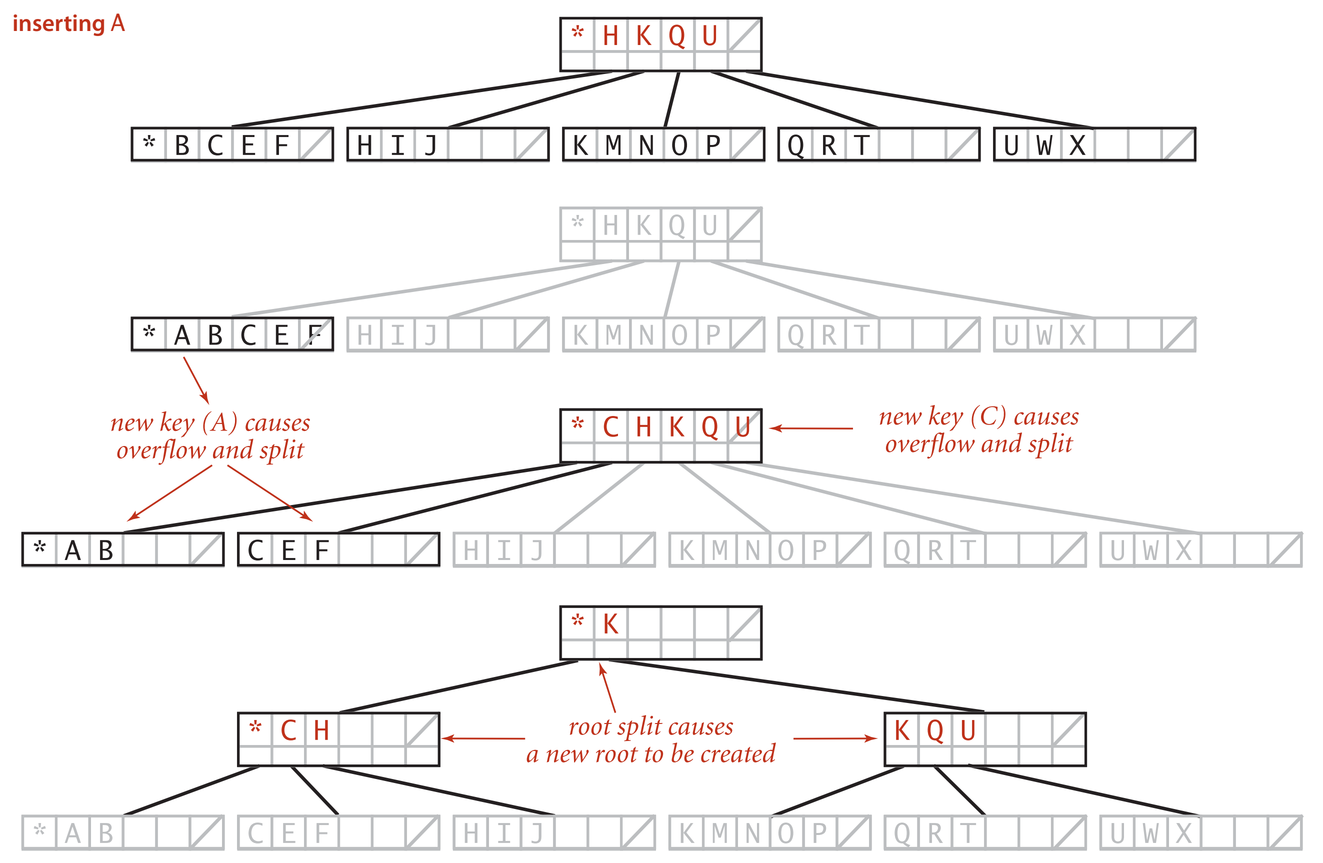
- Start at root.
- Find interval for search key and take corresponding link.
- Search terminates in external node.



Searching in a B-tree set (M = 6)

Insertion in a B-tree

- Search for new key.
- Insert at bottom.
- Split nodes with M key-link pairs on the way up the tree.



Inserting a new key into a B-tree set

Balance in B-tree

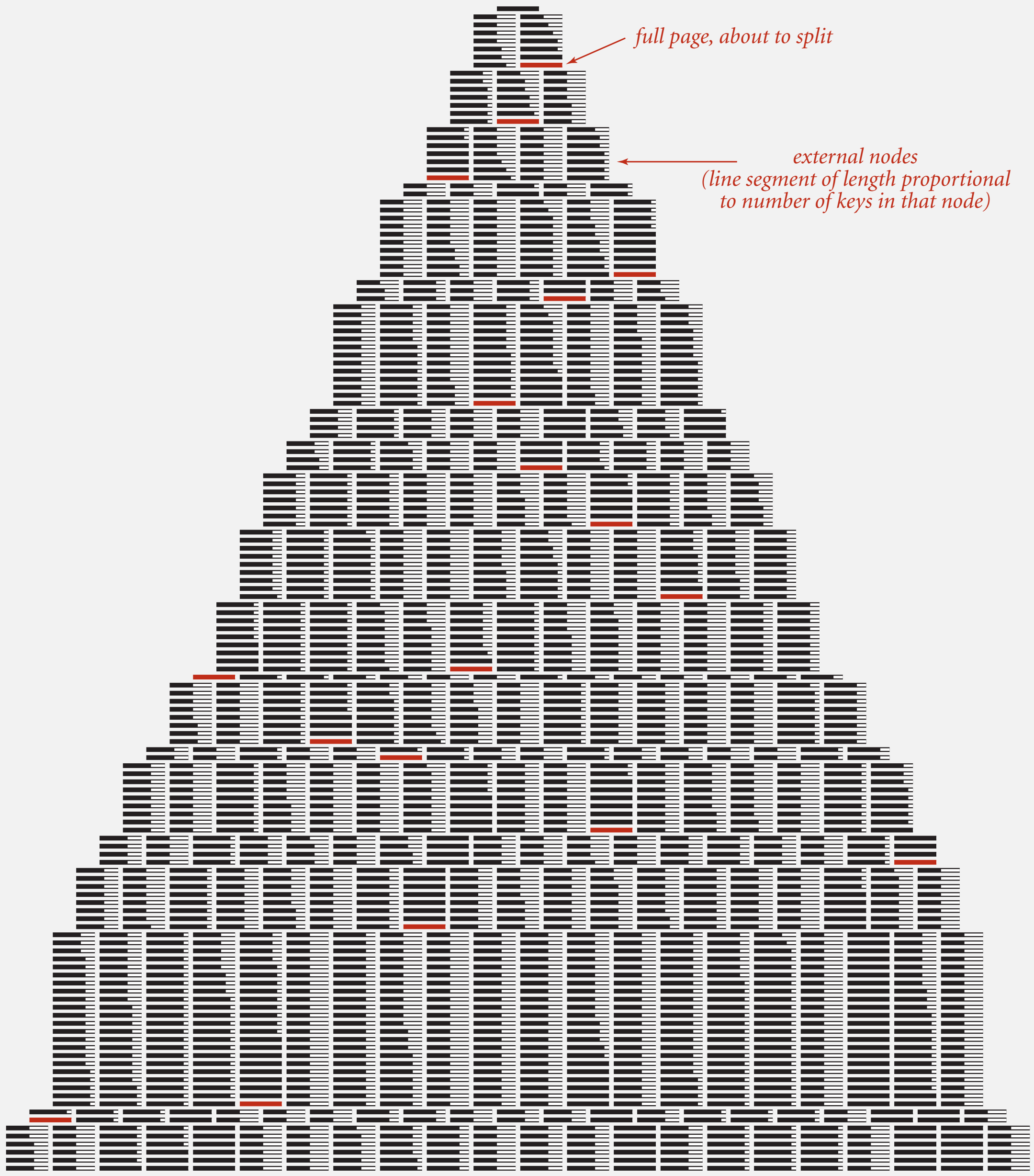
Proposition. A search or an insertion in a B-tree of order M with N keys requires between $\log_{M-1}N$ and $\log_{M/2}N$ probes.

Pf. All internal nodes (besides root) have between M/2 and M-1 links.

In practice. Number of probes is at most 4. ← M = 1000; N = 62 billion $\log_{M/2} N \leq 4$

Optimization. Always keep root page in memory.

Building a large B tree



Balanced trees in the wild

Red-black trees are widely used as system symbol tables.

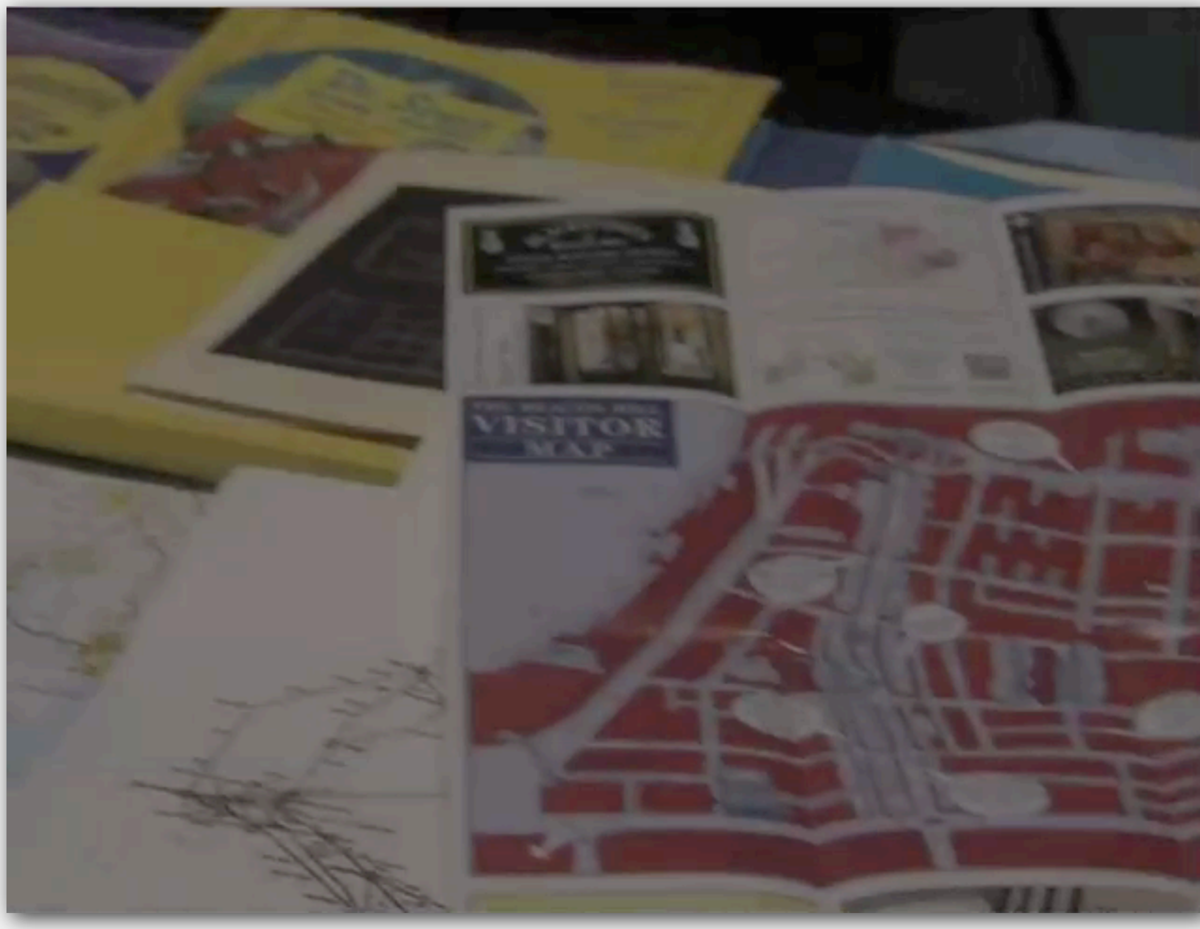
- Java: `java.util.TreeMap`, `java.util.TreeSet`.
- C++ STL: map, multimap, multiset.
- Linux kernel: completely fair scheduler, `linux/rbtree.h`.

B-tree variants. B+ tree, B*tree, B# tree, …

B-trees (and variants) are widely used for file systems and databases.

- Windows: HPFS.
- Mac: HFS, HFS+.
- Linux: ReiserFS, XFS, Ext3FS, JFS.
- Databases: ORACLE, DB2, INGRES, SQL, PostgreSQL.

Red-black trees in the wild



Common sense. Sixth sense.
Together they're the
FBI's newest team.

Red-black trees in the wild

ACT FOUR

FADE IN:

48 INT. FBI HQ - NIGHT 48

Antonio is at THE COMPUTER as Jess explains herself to Nicole and Pollock. The CONFERENCE TABLE is covered with OPEN REFERENCE BOOKS, TOURIST GUIDES, MAPS and REAMS OF PRINTOUTS.

JESS
It was the red door again.

POLLOCK
I thought the red door was the storage container.

JESS
But it wasn't red anymore. It was black.

ANTONIO
So red turning to black means... what?

POLLOCK
Budget deficits? Red ink, black ink?

NICOLE
Yes. I'm sure that's what it is. But maybe we should come up with a couple other options, just in case.

Antonio refers to his COMPUTER SCREEN, which is filled with mathematical equations.

ANTONIO
It could be an algorithm from a binary search tree. A red-black tree tracks every simple path from a node to a descendant leaf with the same number of black nodes.

JESS
Does that help you with girls?

Nicole is tapping away at a computer keyboard. She finds something.

3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- applications

Optimize judiciously

> *" More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason— including blind stupidity. "* — William A. Wulf

> *" We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. "* — Donald E. Knuth

> *" We follow two rules in the matter of optimization:*
> *Rule 1: Don't do it.*
> *Rule 2 (for experts only). Don't do it yet - that is, not until you have a perfectly clear and unoptimized solution. "* — M. A. Jackson

Reference: Effective Java by Joshua Bloch

ST implementations: summary

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	`compareTo()`
red-black tree	2 lg N	2 lg N	2 lg N	1.00 lg N	1.00 lg N	1.00 lg N	yes	`compareTo()`

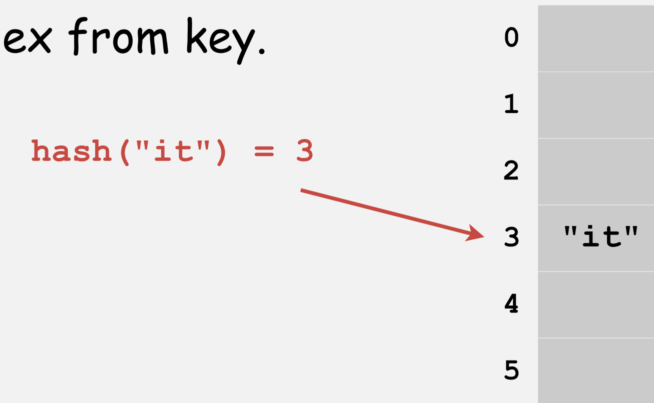
Q. Can we do better?

A. Yes, but with different access to the data.

Hashing: basic plan

Save items in a key-indexed table (index is a function of the key).

Hash function. Method for computing array index from key.



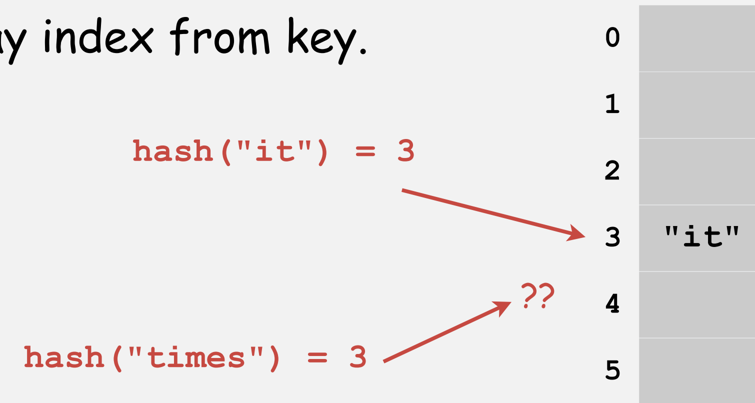
Issues.

- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.

Hashing: basic plan

Save items in a key-indexed table (index is a function of the key).

Hash function. Method for computing array index from key.



Issues.

- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.
- Collision resolution: Algorithm and data structure to handle two keys that hash to the same array index.

Classic space-time tradeoff.

- No space limitation: trivial hash function with key as index.
- No time limitation: trivial collision resolution with sequential search.
- Limitations on both time and space: hashing (the real world).

- hash functions
- separate chaining
- linear probing
- applications

Equality test

Needed because hash methods do not use `compareTo()`.

All Java classes inherit a method `equals()`.

Java requirements. For any references `x`, `y` and `z`:

- Reflexive: `x.equals(x)` is `true`.
- Symmetric: `x.equals(y)` iff `y.equals(x)`.
- Transitive: if `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`.
- Non-null: `x.equals(null)` is `false`.

equivalence relation

do `x` and `y` refer to the same object?

Default implementation. `(x == y)`

Customized implementations. `Integer`, `Double`, `String`, `File`, `URL`, `Date`, ...

User-defined implementations. Some care needed.

Implementing equals for user-defined types

Seems easy

```
public        class Record
{
   private final String name;
   private final long val;
   ...

   public boolean equals(Record y)
   {






      Record that =           y;
      return (this.val == that.val) &&
             (this.name.equals(that.name));
   }
}
```

check that all significant fields are the same

Implementing equals for user-defined types

Seems easy, but requires some care.

```
public final class Record
{
   private final String name;
   private final long val;
   ...

   public boolean equals(Object y)
   {
      if (y == this) return true;

      if (y == null) return false;

      if (y.getClass() != this.getClass())
         return false;

      Record that = (Record) y;
      return (this.val == that.val) &&
             (this.name.equals(that.name));
   }
}
```

no safe way to use **equals()** with inheritance

must be **Object**.
Why? Experts still debate.

optimize for true object equality

check for **null**

objects must be in the same class

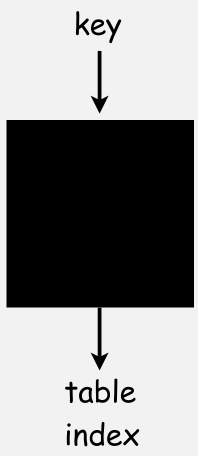
check that all significant fields are the same

Computing the hash function

Idealistic goal. Scramble the keys uniformly to produce a table index.

- Efficiently computable.
- Each table index equally likely for each key.

thoroughly researched problem,
still problematic in practical applications



Ex 1. Phone numbers.

- Bad: first three digits.
- Better: last three digits.

Ex 2. Social Security numbers.

573 = California, 574 = Alaska
(assigned in chronological order within geographic region)

- Bad: first three digits.
- Better: last three digits.

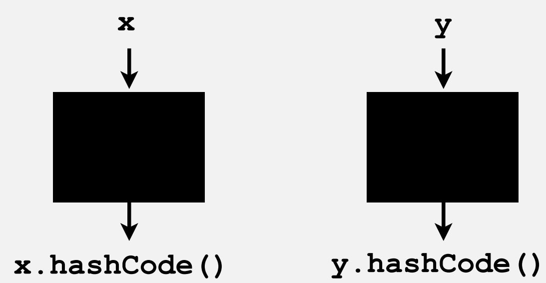
Practical challenge. Need different approach for each key type.

Java's hash code conventions

All Java classes inherit a method `hashCode()`, which returns a 32-bit `int`.

Requirement. If `x.equals(y)`, then `(x.hashCode() == y.hashCode())`.

Highly desirable. If `!x.equals(y)`, then `(x.hashCode() != y.hashCode())`.



Default implementation. Memory address of `x`.

Customized implementations. `Integer`, `Double`, `String`, `File`, `URL`, `Date`, …

User-defined types. Users are on their own.

Implementing hash code: integers and doubles

```
public final class Integer
{
   private final int value;
   ...

   public int hashCode()
   {  return value;  }
}
```

```
public final class Double
{
   private final double value;
   ...

   public int hashCode()
   {
      long bits = doubleToLongBits(value);
      return (int) (bits ^ (bits >>> 32));
   }
}
```

convert to IEEE 64-bit representation;
xor most significant 32-bits
with least significant 32-bits

Implementing hash code: strings

```
public final class String
{
   private final char[] s;
   ...

   public int hashCode()
   {
      int hash = 0;
      for (int i = 0; i < length(); i++)
         hash = s[i] + (31 * hash);
      return hash;
   }
}
```

i^{th} character of s

char	Unicode
…	…
'a'	97
'b'	98
'c'	99
…	…

- Horner's method to hash string of length L: L multiplies/adds.
- Equivalent to $h = 31^{L-1} \cdot s^0 + \ldots + 31^2 \cdot s^{L-3} + 31^1 \cdot s^{L-2} + 31^0 \cdot s^{L-1}$.

Ex.

```
String s = "call";
int code = s.hashCode();
```

$3045982 = 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0$

$= 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99)))$

A poor hash code

Ex. Strings (in Java 1.1).

- For long strings: only examine 8-9 evenly spaced characters.
- Benefit: saves time in performing arithmetic.

```
public int hashCode()
{
   int hash = 0;
   int skip = Math.max(1, length() / 8);
   for (int i = 0; i < length(); i += skip)
      hash = s[i] + (37 * hash);
   return hash;
}
```

- Downside: great potential for bad collision patterns.

```
http://www.cs.princeton.edu/introcs/13loop/Hello.java
http://www.cs.princeton.edu/introcs/13loop/Hello.class
http://www.cs.princeton.edu/introcs/13loop/Hello.html
http://www.cs.princeton.edu/introcs/13loop/index.html
http://www.cs.princeton.edu/introcs/12type/index.html
```

Implementing hash code: user-defined types

```
public final class Record
{
   private String name;
   private int id;
   private double value;

   public Record(String name, int id, double value)
   {  /* as before */  }

   ...

   public boolean equals(Object y)
   {  /* as before */  }

   public int hashCode()
   {
      int hash = 17;
      hash = 31*hash + name.hashCode();
      hash = 31*hash + id;
      hash = 31*hash + Double.valueOf(value).hashCode();
      return hash;
   }
}
```

nonzero constant

typically a small prime

Hash code design

"Standard" recipe for user-defined types.

- Combine each significant field using the 31x + y rule.
- If field is a primitive type, use built-in hash code.
- If field is an array, apply to each element.
- If field is an object, apply rule recursively.

In practice. Recipe works reasonably well; used in Java libraries.
In theory. Need a theorem for each type to ensure reliability.

Basic rule. Need to use the whole key to compute hash code; consult an expert for state-of-the-art hash codes.

Modular hashing

Hash code. An `int` between -2^{31} and $2^{31}-1$.

Hash function. An `int` between `0` and `M-1` (for use as array index).

typically a prime or power of 2

```
private int hash(Key key)
{   return key.hashCode() % M;   }
```

bug

```
private int hash(Key key)
{   return Math.abs(key.hashCode()) % M;   }
```

1-in-a-billion bug

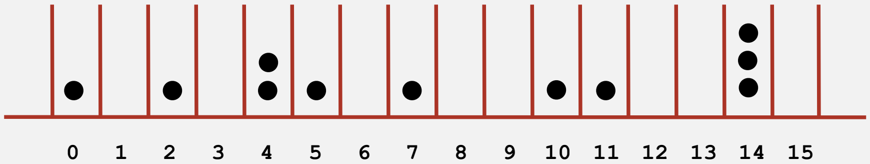
```
private int hash(Key key)
{   return (key.hashCode() & 0x7fffffff) % M;   }
```

correct

Uniform hashing assumption

Assumption J (uniform hashing hashing assumption).
Each key is equally likely to hash to an integer between 0 and M-1.

Bins and balls. Throw balls uniformly at random into M bins.



Birthday problem. Expect two balls in the same bin after $\sim \sqrt{\pi M / 2}$ tosses.

Coupon collector. Expect every bin has ≥ 1 ball after $\sim M \ln M$ tosses.

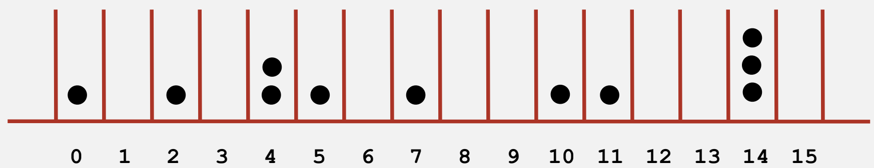
Load balancing. After M tosses, expect most loaded bin has
$\Theta(\log M / \log \log M)$ balls.

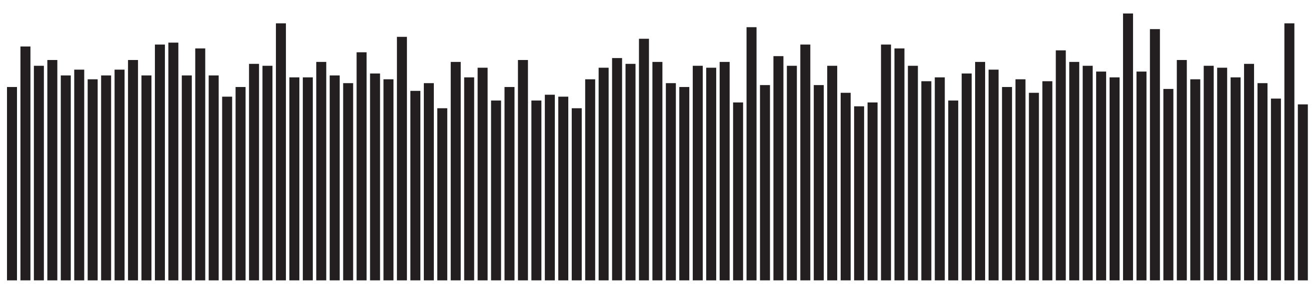
Uniform hashing assumption

Assumption J (uniform hashing hashing assumption).

Each key is equally likely to hash to an integer between 0 and M-1.

Bins and balls. Throw balls uniformly at random into M bins.





Hash value frequencies for words in Tale of Two Cities (M = 97)

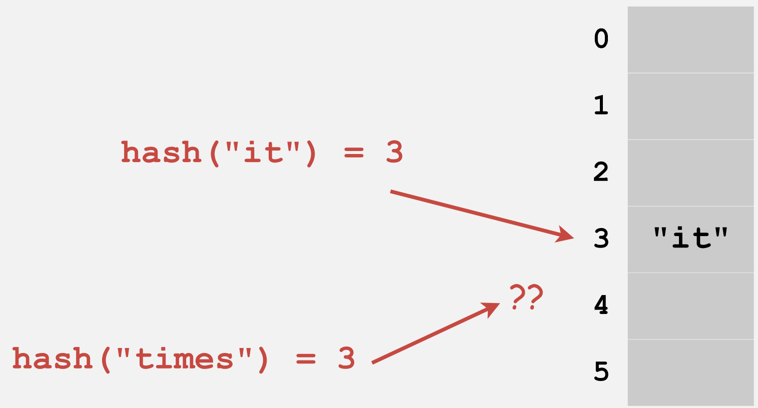
Java's `String` data uniformly distribute the keys of Tale of Two Cities

Collisions

Collision. Two distinct keys hashing to same index.

- Birthday problem ⇒ can't avoid collisions unless you have a ridiculous amount (quadratic) of memory.
- Coupon collector + load balancing ⇒ collisions will be evenly distributed.

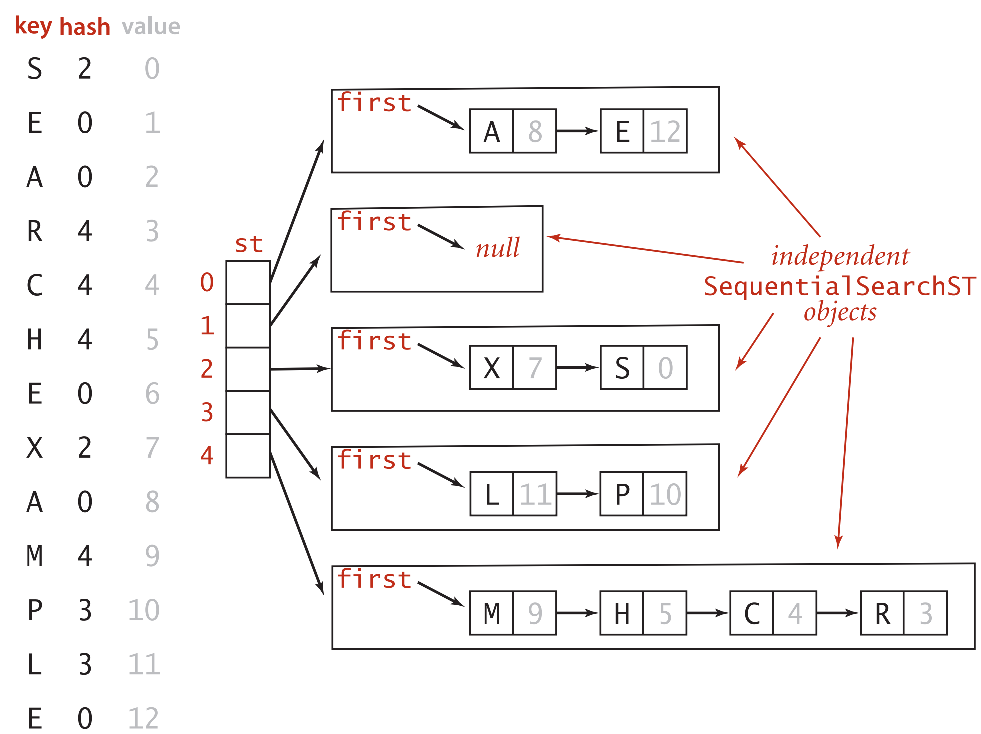
Challenge. Deal with collisions efficiently.



Separate chaining ST

Use an array of $M < N$ linked lists. [H. P. Luhn, IBM 1953]

- Hash: map key to integer i between 0 and M-1.
- Insert: put at front of i^{th} chain (if not already there).
- Search: only need to search i^{th} chain.



Hashing with separate chaining for standard indexing client

Separate chaining ST: Java implementation

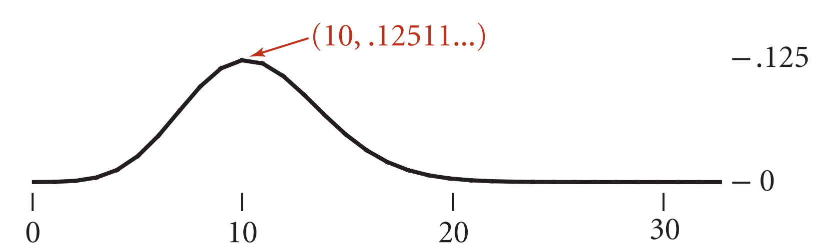
```
public class SeparateChainingHashST<Key, Value>
{
   private int N;      // number of key-value pairs
   private int M;      // hash table size
   private SequentialSearchST<Key, Value> [] st; // array of STs

   public SeparateChainingHashST()
   {  this(997);  }

   public SeparateChainingHashST(int M)
   {
      this.M = M;
      st = (SequentialSearchST<Key, Value>[]) new SequentialSearchST[M];
      for (int i = 0; i < M; i++)
         st[i] = new SequentialSearchST<Key, Value>();
   }
   private int hash(Key key)
   {  return (key.hashCode() & 0x7fffffff) % M;  }

   public Value get(Key key)
   {  return st[hash(key)].get(key);  }

   public void put(Key key, Value val)
   {  st[hash(key)].put(key, val);  }
}
```

array doubling code omitted

Analysis of separate chaining

Proposition K. Under uniform hashing assumption, probability that the number of keys in a list is within a constant factor of N/M is extremely close to 1.

Pf sketch. Distribution of list size obeys a binomial distribution.



Binomial distribution $(N = 10^4,\ M = 10^3,\ \alpha = 10)$

equals() and hashCode()

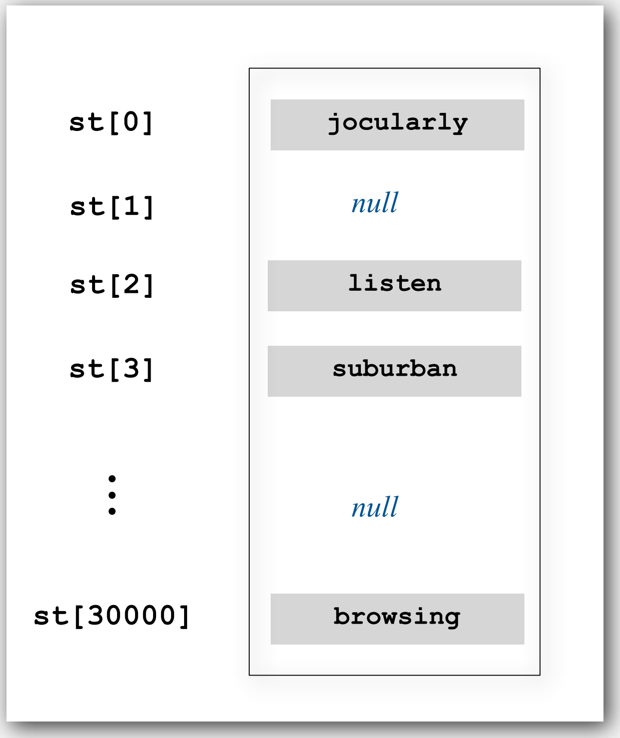
Consequence. Number of probes for search/insert is proportional to N/M.

- M too large $\Rightarrow$ too many empty chains.
- M too small $\Rightarrow$ chains too long.
- Typical choice: M ~ N/5 $\Rightarrow$ constant-time ops.

M times faster than sequential search

- hash functions
- separate chaining
- **linear probing**
- applications

Open addressing. [Amdahl-Boehme-Rocherster-Samuel, IBM 1953]
When a new key collides, find next empty slot, and put it there.



linear probing (M = 30001, N = 15000)

Linear probing

Use an array of size M > N.

- Hash: map key to integer i between 0 and M-1.
- Insert: put at table index i if free; if not try i+1, i+2, etc.
- Search: search table index i; if occupied but no match, try i+1, i+2, etc.

-	-	-	S	H	-	-	A	C	E	R	-	-
0	1	2	3	4	5	6	7	8	9	10	11	12

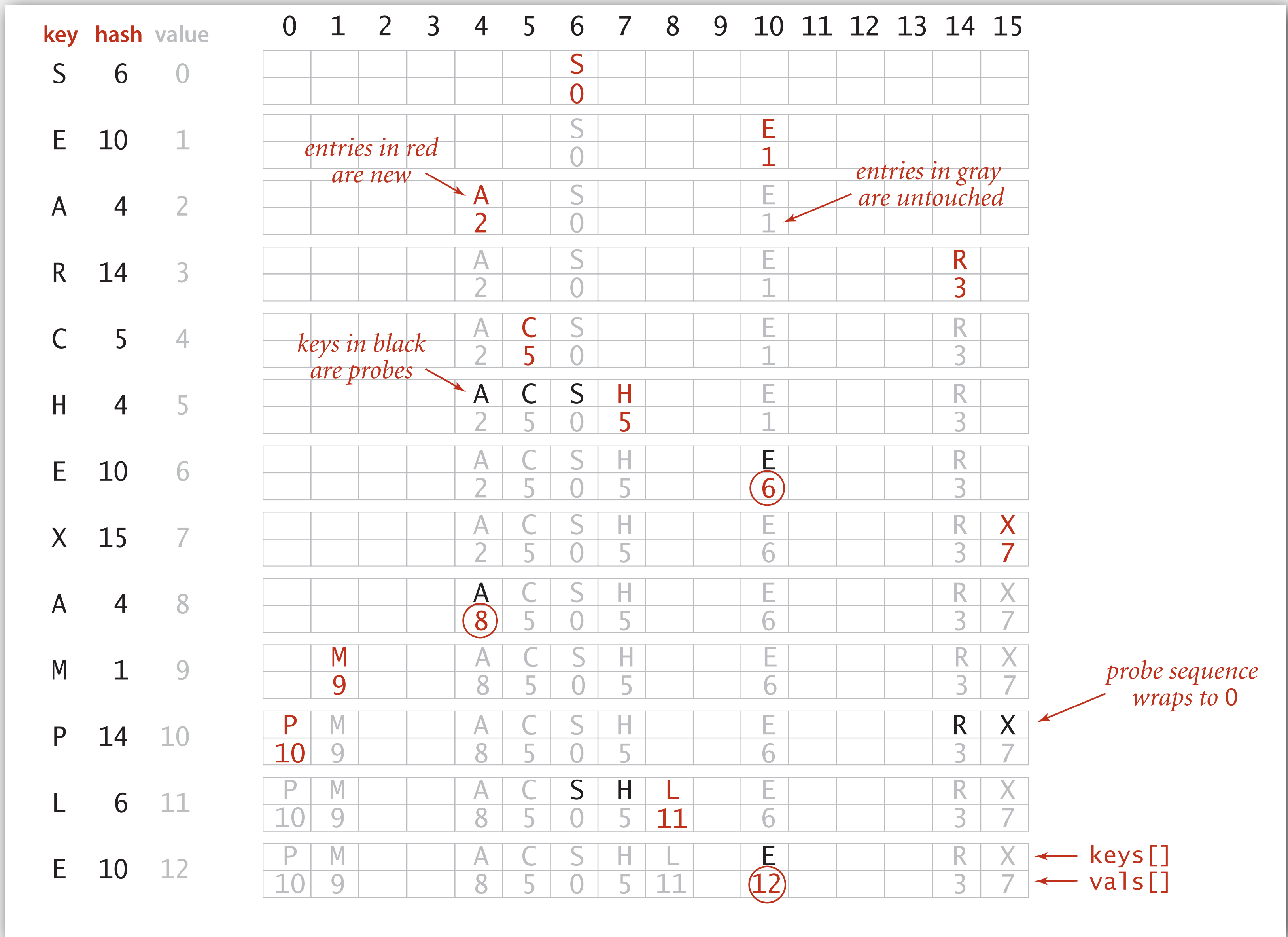
-	-	-	S	H	-	-	A	C	E	R	I	-
0	1	2	3	4	5	6	7	8	9	10	11	12

insert I
hash(I) = 11

-	-	-	S	H	-	-	A	C	E	R	I	N
0	1	2	3	4	5	6	7	8	9	10	11	12

insert N
hash(N) = 8

Linear probing: trace of standard indexing client



Linear probing ST implementation

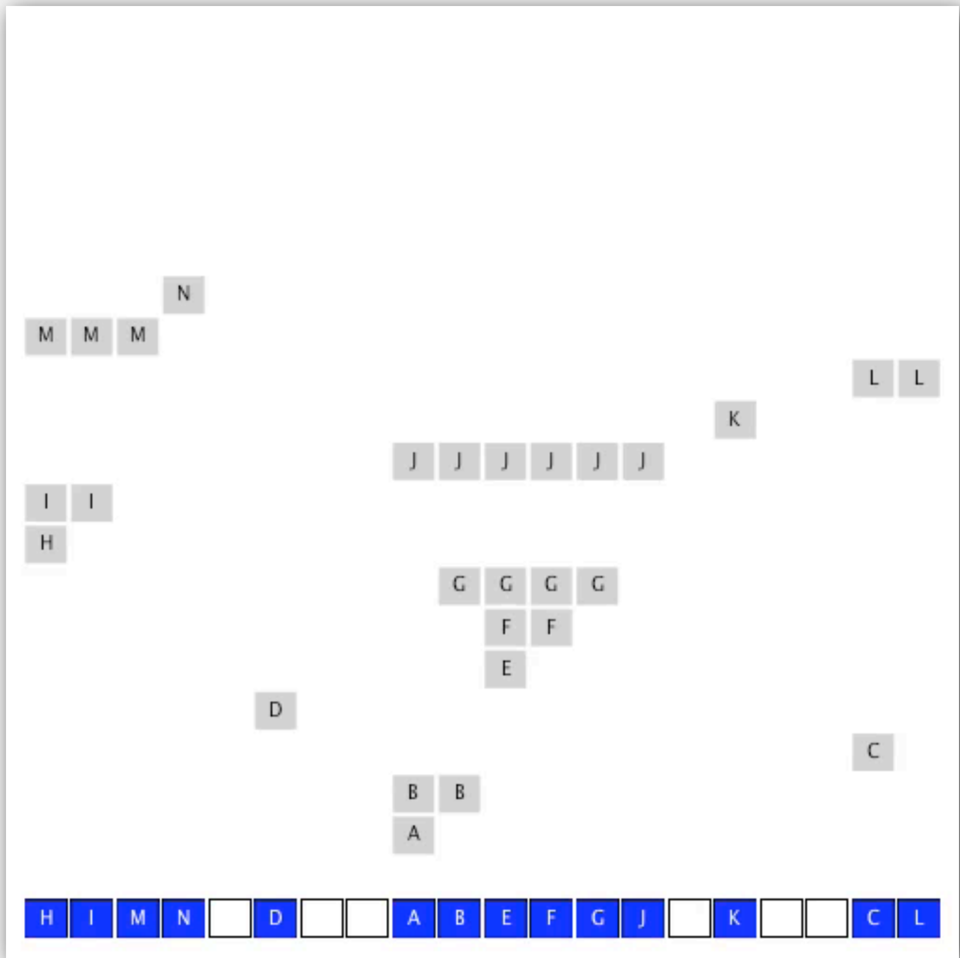
```
public class LinearProbingHashST<Key, Value>
{
   private int M = 30001;
   private Value[] vals = (Value[]) new Object[M];
   private Key[]   keys = (Key[])   new Object[M];

   private int hash(Key key) {  /* as before */  }

   public void put(Key key, Value val)
   {
      int i;
      for (i = hash(key); keys[i] != null; i = (i+1) % M)
         if (keys[i].equals(key))
             break;
      keys[i] = key;
      vals[i] = val;
   }

   public Value get(Key key)
   {
      for (int i = hash(key); keys[i] != null; i = (i+1) % M)
         if (key.equals(keys[i]))
             return vals[i];
      return null;
   }
}
```

array doubling
code omitted

Clustering

Cluster. A contiguous block of items.

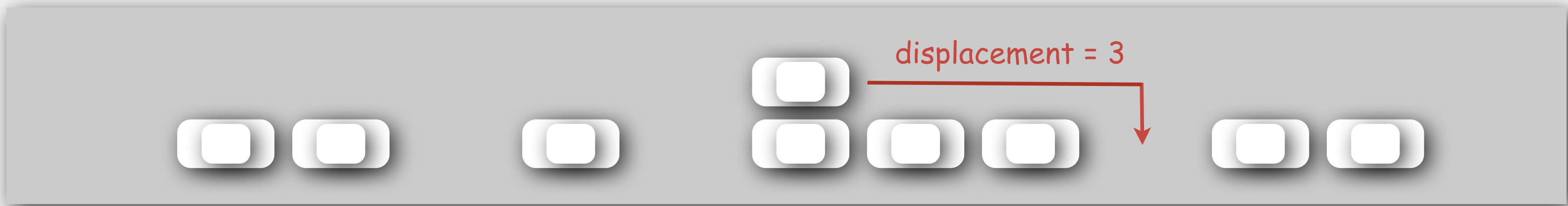
Observation. New keys likely to hash into middle of big clusters.



Knuth's parking problem

Model. Cars arrive at one-way street with M parking spaces.
Each desires a random space i: if space i is taken, try $i+1$, $i+2$, …

Q. What is mean displacement of a car?



Empty. With $M/2$ cars, mean displacement is $\sim 3/2$.
Full. With M cars, mean displacement is $\sim \sqrt{\pi M / 8}$

Analysis of linear probing

Proposition M. Under uniform hashing assumption, the average number of probes in a hash table of size M that contains N = α M keys is:

$$\sim \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \qquad \sim \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

search hit   search miss / insert

Pf. [Knuth 1962] A landmark in analysis of algorithms.

Parameters.

- M too large $\Rightarrow$ too many empty array entries.
- M too small $\Rightarrow$ search time blows up.
- Typical choice: α = N/M ~ $\frac{1}{2}$.

probes for search hit is about 3/2
probes for search miss is about 5/2

ST implementations: summary

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	`compareTo()`
red-black tree	2 lg N	2 lg N	2 lg N	1.00 lg N	1.00 lg N	1.00 lg N	yes	`compareTo()`
hashing	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	`equals()`

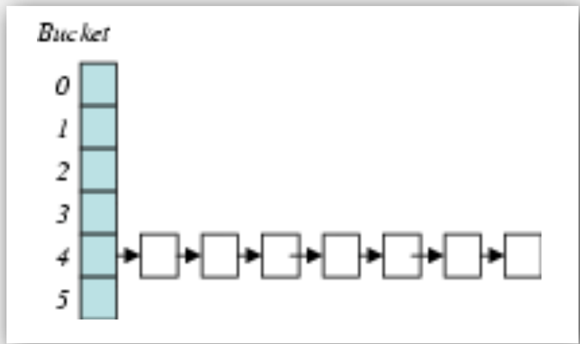
* under uniform hashing assumption

Algorithmic complexity attacks

Q. Is the uniform hashing assumption important in practice?

A. Obvious situations: aircraft control, nuclear reactor, pacemaker.

A. Surprising situations: denial-of-service attacks.



malicious adversary learns your hash function (e.g., by reading Java API) and causes a big pile-up in single slot that grinds performance to a halt

Real-world exploits. [Crosby-Wallach 2003]

- Bro server: send carefully chosen packets to DOS the server, using less bandwidth than a dial-up modem.
- Perl 5.8.0: insert carefully chosen strings into associative array.
- Linux 2.4.20 kernel: save files with carefully chosen names.

Algorithmic complexity attack on Java

Goal. Find family of strings with the same hash code.

Solution. The base-31 hash code is part of Java's string API.

key	hashCode()
"Aa"	2112
"BB"	2112

key	hashCode()
"AaAaAaAa"	-540425984
"AaAaAaBB"	-540425984
"AaAaBBAa"	-540425984
"AaAaBBBB"	-540425984
"AaBBAaAa"	-540425984
"AaBBAaBB"	-540425984
"AaBBBBAa"	-540425984
"AaBBBBBB"	-540425984

key	hashCode()
"BBAaAaAa"	-540425984
"BBAaAaBB"	-540425984
"BBAaBBAa"	-540425984
"BBAaBBBB"	-540425984
"BBBBAaAa"	-540425984
"BBBBAaBB"	-540425984
"BBBBBBAa"	-540425984
"BBBBBBBB"	-540425984

2^N strings of length $2N$ that hash to same value!

Diversion: one-way hash functions

One-way hash function. Hard to find a key that will hash to a desired value, or to find two keys that hash to same value.

Ex. $\underbrace{\text{MD4, MD5, SHA-0, SHA-1,}}_{\text{known to be insecure}}$ SHA-2, WHIRLPOOL, RIPEMD-160.

```
String password = args[0];
MessageDigest sha1 = MessageDigest.getInstance("SHA1");
byte[] bytes = sha1.digest(password);

/* prints bytes as hex string */
```

Applications. Digital fingerprint, message digest, storing passwords.
Caveat. Too expensive for use in ST implementations.

Separate chaining vs. linear probing

Separate chaining.

- Easier to implement delete.
- Performance degrades gracefully.
- Clustering less sensitive to poorly-designed hash function.

Linear probing.

- Less wasted space.
- Better cache performance.

Hashing: variations on the theme

Many improved versions have been studied.

Two-probe hashing. (separate chaining variant)

- Hash to two positions, put key in shorter of the two chains.
- Reduces average length of the longest chain to log log N.

Double hashing. (linear probing variant)

- Use linear probing, but skip a variable amount, not just 1 each time.
- Effectively eliminates clustering.
- Can allow table to become nearly full.

Hashing vs. balanced trees

Hashing.

- Simpler to code.
- No effective alternative for unordered keys.
- Faster for simple keys (a few arithmetic ops versus log N compares).
- Better system support in Java for strings (e.g., cached hash code).

Balanced trees.

- Stronger performance guarantee.
- Support for ordered ST operations.
- Easier to implement `compareTo()` correctly than `equals()` and `hashCode()`.

Java system includes both.

- Red-black trees: `java.util.TreeMap`, `java.util.TreeSet`.
- Hashing: `java.util.HashMap`, `java.util.IdentityHashMap`.

3.5 Symbol Tables Applications

- sets
- dictionary clients
- indexing clients
- sparse vectors

- sets
- dictionary clients
- indexing clients
- sparse vectors

Set API

Mathematical set. A collection of distinct keys.

`public class SET<Key extends Comparable<Key>>`		
	`SET()`	*create an empty set*
`void`	`add(Key key)`	*add the key to the set*
`boolean`	`contains(Key key)`	*is the key in the set?*
`void`	`remove(Key key)`	*remove the key from the set*
`int`	`size()`	*return the number of keys in the set*
`Iterator<Key>`	`iterator()`	*iterator through keys in the set*

Q. How to implement?

Exception filter

- Read in a list of words from one file.
- Print out all words from standard input that are { in, not in } the list.

```
% more list.txt
was it the of

% java WhiteList list.txt < tinyTale.txt
it was the of it was the of
it was the of it was the of
it was the of it was the of
it was the of it was the of
it was the of it was the of

% java BlackList list.txt < tinyTale.txt
best times worst times
age wisdom age foolishness
epoch belief epoch incredulity
season light season darkness
spring hope winter despair
```

list of exceptional words

Exception filter applications

- Read in a list of words from one file.
- Print out all words from standard input that are { in, not in } the list.

application	purpose	key	in list
spell checker	identify misspelled words	word	dictionary words
browser	mark visited pages	URL	visited pages
parental controls	block sites	URL	bad sites
chess	detect draw	board	positions
spam filter	eliminate spam	IP address	spam addresses
credit cards	check for stolen cards	number	stolen cards

Exception filter: Java implementation

- Read in a list of words from one file.
- Print out all words from standard input that are { in, not in } the list.

```
public class WhiteList
{
   public static void main(String[] args)
   {
      SET<String> set = new SET<String>();

      In in = new In(args[0]);
      while (!in.isEmpty())
         set.add(in.readString());

      while (!StdIn.isEmpty())
      {
         String word = StdIn.readString();
         if (set.contains(word))
            StdOut.println(word);
      }
   }
}
```

create empty set of strings

read in whitelist

print words in list

Exception filter: Java implementation

- Read in a list of words from one file.
- Print out all words from standard input that are { in, not in } the list.

```
public class BlackList
{
   public static void main(String[] args)
   {
      SET<String> set = new SET<String>();

      In in = new In(args[0]);
      while (!in.isEmpty())
         set.add(in.readString());

      while (!StdIn.isEmpty())
      {
         String word = StdIn.readString();
         if (!set.contains(word))
            StdOut.println(word);
      }
   }
}
```

create empty set of strings

read in blacklist

print words not in list

- sets
- **dictionary clients**
- indexing clients
- sparse vectors

Dictionary lookup

Command-line arguments.

- A comma-separated value (CSV) file.
- Key field.
- Value field.

Ex 1. DNS lookup.

URL is key IP is value

```
% java LookupCSV ip.csv 0 1
adobe.com
192.150.18.60
www.princeton.edu
128.112.128.15
ebay.edu
Not found
```

IP is key URL is value

```
% java LookupCSV ip.csv 1 0
128.112.128.15
www.princeton.edu
999.999.999.99
Not found
```

```
% more ip.csv
www.princeton.edu,128.112.128.15
www.cs.princeton.edu,128.112.136.35
www.math.princeton.edu,128.112.18.11
www.cs.harvard.edu,140.247.50.127
www.harvard.edu,128.103.60.24
www.yale.edu,130.132.51.8
www.econ.yale.edu,128.36.236.74
www.cs.yale.edu,128.36.229.30
espn.com,199.181.135.201
yahoo.com,66.94.234.13
msn.com,207.68.172.246
google.com,64.233.167.99
baidu.com,202.108.22.33
yahoo.co.jp,202.93.91.141
sina.com.cn,202.108.33.32
ebay.com,66.135.192.87
adobe.com,192.150.18.60
163.com,220.181.29.154
passport.net,65.54.179.226
tom.com,61.135.158.237
nate.com,203.226.253.11
cnn.com,64.236.16.20
daum.net,211.115.77.211
blogger.com,66.102.15.100
fastclick.com,205.180.86.4
wikipedia.org,66.230.200.100
rakuten.co.jp,202.72.51.22
...
```

Dictionary lookup

Command-line arguments.

- A comma-separated value (CSV) file.
- Key field.
- Value field.

Ex 2. Amino acids.

codon is key　name is value

```
% java Lookup amino.csv 0 3
ACT
Threonine
TAG
Stop
CAT
Histidine
```

```
% more amino.csv
TTT,Phe,F,Phenylalanine
TTC,Phe,F,Phenylalanine
TTA,Leu,L,Leucine
TTG,Leu,L,Leucine
TCT,Ser,S,Serine
TCC,Ser,S,Serine
TCA,Ser,S,Serine
TCG,Ser,S,Serine
TAT,Tyr,Y,Tyrosine
TAC,Tyr,Y,Tyrosine
TAA,Stop,Stop,Stop
TAG,Stop,Stop,Stop
TGT,Cys,C,Cysteine
TGC,Cys,C,Cysteine
TGA,Stop,Stop,Stop
TGG,Trp,W,Tryptophan
CTT,Leu,L,Leucine
CTC,Leu,L,Leucine
CTA,Leu,L,Leucine
CTG,Leu,L,Leucine
CCT,Pro,P,Proline
CCC,Pro,P,Proline
CCA,Pro,P,Proline
CCG,Pro,P,Proline
CAT,His,H,Histidine
CAC,His,H,Histidine
CAA,Gln,Q,Glutamine
CAG,Gln,Q,Glutamine
CGT,Arg,R,Arginine
CGC,Arg,R,Arginine
...
```

Dictionary lookup

Command-line arguments.

- A comma-separated value (CSV) file.
- Key field.
- Value field.

Ex 3. Class list.

login is key

first name is value

```
% java Lookup classlist.csv 4 1
eberl
Ethan
nwebb
Natalie
```

login is key

precept is value

```
% java Lookup classlist.csv 4 3
dpan
P01
```

```
% more classlist.csv
13,Berl,Ethan Michael,P01,eberl
11,Bourque,Alexander Joseph,P01,abourque
12,Cao,Phillips Minghua,P01,pcao
11,Chehoud,Christel,P01,cchehoud
10,Douglas,Malia Morioka,P01,malia
12,Haddock,Sara Lynn,P01,shaddock
12,Hantman,Nicole Samantha,P01,nhantman
11,Hesterberg,Adam Classen,P01,ahesterb
13,Hwang,Roland Lee,P01,rhwang
13,Hyde,Gregory Thomas,P01,ghyde
13,Kim,Hyunmoon,P01,hktwo
11,Kleinfeld,Ivan Maximillian,P01,ikleinfe
12,Korac,Damjan,P01,dkorac
11,MacDonald,Graham David,P01,gmacdona
10,Michal,Brian Thomas,P01,bmichal
12,Nam,Seung Hyeon,P01,seungnam
11,Nastasescu,Maria Monica,P01,mnastase
11,Pan,Di,P01,dpan
12,Partridge,Brenton Alan,P01,bpartrid
13,Rilee,Alexander,P01,arilee
13,Roopakalu,Ajay,P01,aroopaka
11,Sheng,Ben C,P01,bsheng
12,Webb,Natalie Sue,P01,nwebb
...
```

Dictionary lookup: Java implementation

```
public class LookupCSV
{
   public static void main(String[] args)
   {
      In in = new In(args[0]);
      int keyField = Integer.parseInt(args[1]);
      int valField = Integer.parseInt(args[2]);

      ST<String, String> st = new ST<String, String>();
      while (!in.isEmpty())
      {
         String line = in.readLine();
         String[] tokens = database[i].split(",");
         String key = tokens[keyField];
         String val = tokens[valField];
         st.put(key, val);
      }

      while (!StdIn.isEmpty())
      {
         String s = StdIn.readString();
         if (!st.contains(s)) StdOut.println("Not found");
         else                 StdOut.println(st.get(s));
      }
   }
}
```

process input file

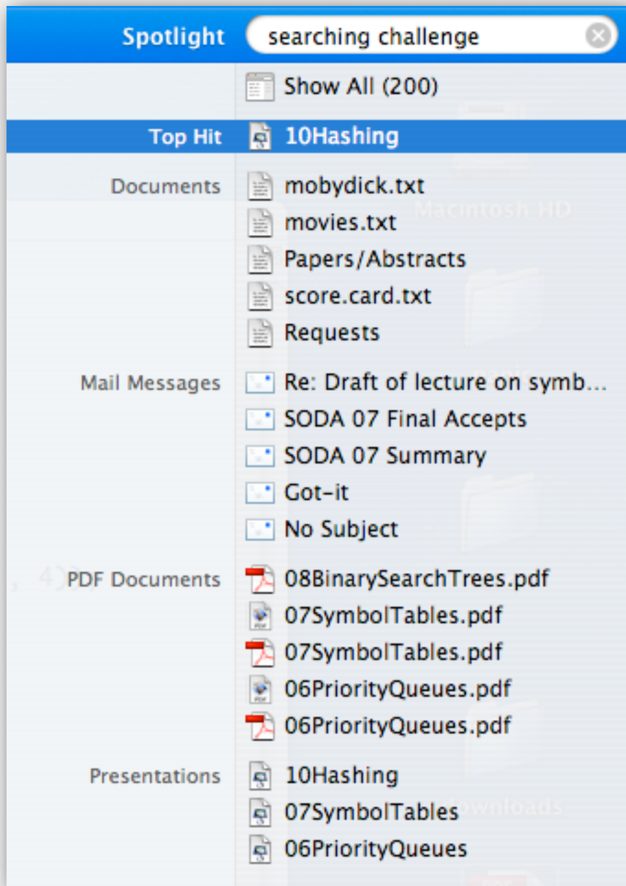
build symbol table

process lookups
with standard I/O

- sets
- dictionary clients
- indexing clients
- sparse vectors

File indexing

Goal. Index a PC (or the web).



File indexing

Goal. Given a list of files specified as command-line arguments, create an index so that can efficiently find all files containing a given query string.

```
% ls *.txt
aesop.txt magna.txt moby.txt
sawyer.txt tale.txt

% java FileIndex *.txt
freedom
magna.txt moby.txt tale.txt

whale
moby.txt

lamb
sawyer.txt aesop.txt
```

```
% ls *.java

% java FileIndex *.java
BlackList.java Concordance.java
DeDup.java FileIndex.java ST.java
SET.java WhiteList.java

import
FileIndex.java SET.java ST.java

Comparator
null
```

Solution. Key = query string; value = set of files containing that string.

File indexing

```
public class FileIndex
{
   public static void main(String[] args)
   {
      ST<String, SET<File>> st = new ST<String, SET<File>>();

      for (String filename : args) {
         File file = new File(filename);
         In in = new In(file);
         while !(in.isEmpty())
         {
            String word = in.readString();
            if (!st.contains(word))
               st.put(s, new SET<File>());
            SET<File> set = st.get(key);
            set.add(file);
         }
      }

      while (!StdIn.isEmpty())
      {
         String query = StdIn.readString();
         StdOut.println(st.get(query));
      }
   }
}
```

symbol table

list of file names from command line

for each word in file, add file to corresponding set

process queries

Book index

Goal. Index for an e-book.

Index

727

Concordance

Goal. Preprocess a text corpus to support concordance queries: given a word, find all occurrences with their immediate contexts.

```
% java Concordance tale.txt
cities
tongues of the two *cities* that were blended in

majesty
their turnkeys and the *majesty* of the law fired
me treason against the *majesty* of the people in
of his most gracious *majesty* king george the third

princeton
no matches
```

Concordance

```
public class Concordance
{
   public static void main(String[] args)
   {
      In in = new In(args[0]);
      String[] words = StdIn.readAll().split("\\s+");
      ST<String, SET<Integer>> st = new ST<String, SET<Integer>>();
      for (int i = 0; i < words.length; i++)
      {
         String s = words[i];
         if (!st.contains(s))
            st.put(s, new SET<Integer>());
         SET<Integer> pages = st.get(s);
         set.put(i);
      }

      while (!StdIn.isEmpty())
      {
         String query = StdIn.readString();
         SET<Integer> set = st.get(query);
         for (int k : set)
            // print words[k-5] to words[k+5]
      }
   }
}
```

read text and build index

process queries and print concordances

- sets
- dictionary clients
- indexing clients
- **sparse vectors**

Matrix-vector multiplication (standard implementation)

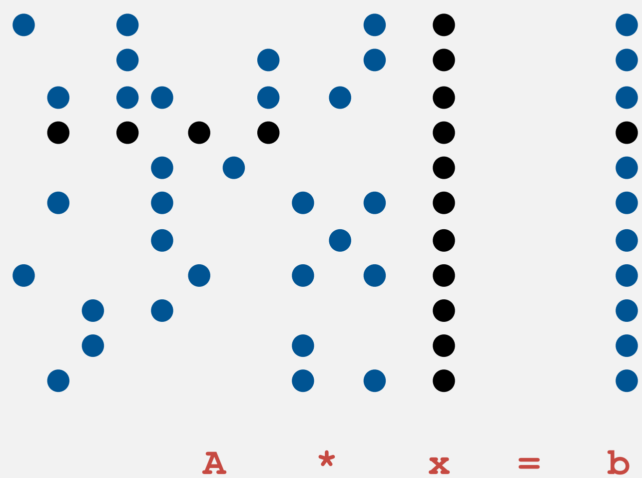
$$
\begin{array}{ccc}
\texttt{a[][]} & \texttt{x[]} & & \texttt{b[]} \\
\begin{bmatrix} 0 & .90 & 0 & 0 & 0 \\ 0 & 0 & .36 & .36 & .18 \\ 0 & 0 & 0 & .90 & 0 \\ .90 & 0 & 0 & 0 & 0 \\ .47 & 0 & .47 & 0 & 0 \end{bmatrix} &
\begin{bmatrix} .05 \\ .04 \\ .36 \\ .37 \\ .19 \end{bmatrix} & = &
\begin{bmatrix} .036 \\ .297 \\ .333 \\ .045 \\ .1927 \end{bmatrix}
\end{array}
$$

```
...
double[][] a = new double[N][N];
double[] x = new double[N];
double[] b = new double[N];
...
// initialize a[][] and x[]
...
for (int i = 0; i < N; i++)
{
   sum = 0.0;
   for (int j = 0; j < N; j++)
      sum += a[i][j]*x[j];
   b[i] = sum;
}
```

nested loops
N^2 running time

Problem. Sparse matrix-vector multiplication.

Assumptions. Matrix dimension is 10,000; average nonzeros per row ~ 10.



Vector representations

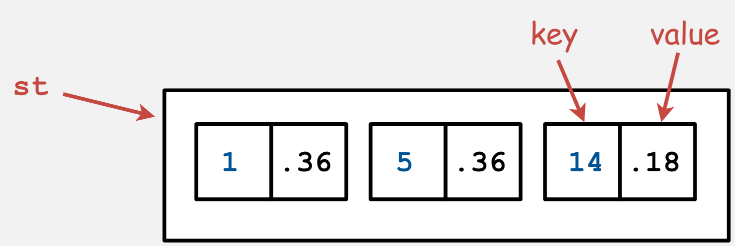
1D array (standard) representation.

- Constant time access to elements.
- Space proportional to N.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	.36	0	0	0	.36	0	0	0	0	0	0	0	0	.18	0	0	0	0	0

Symbol table representation.

- key = index, value = entry
- Efficient iterator.
- Space proportional to number of nonzeros.



Sparse vector data type

```
public class SparseVector
{
   private HashST<Integer, Double> v;

   public SparseVector()
   {  v = new HashST<Integer, Double>();  }

   public void put(int i, double x)
   {  v.put(i, x);  }

   public double get(int i)
   {
      if (!v.contains(i)) return 0.0;
      else return v.get(i);
   }

   public Iterable<Integer> indices()
   {  return v.keys();  }

   public double dot(double[] that)
   {
       double sum = 0.0;
       for (int i : indices())
           sum += that[i]*this.get(i);
       return sum;
   }
}
```

`HashST` because order not important

empty ST represents all 0s vector

`a[i] = value`

`return a[i]`

dot product is constant time for sparse vectors

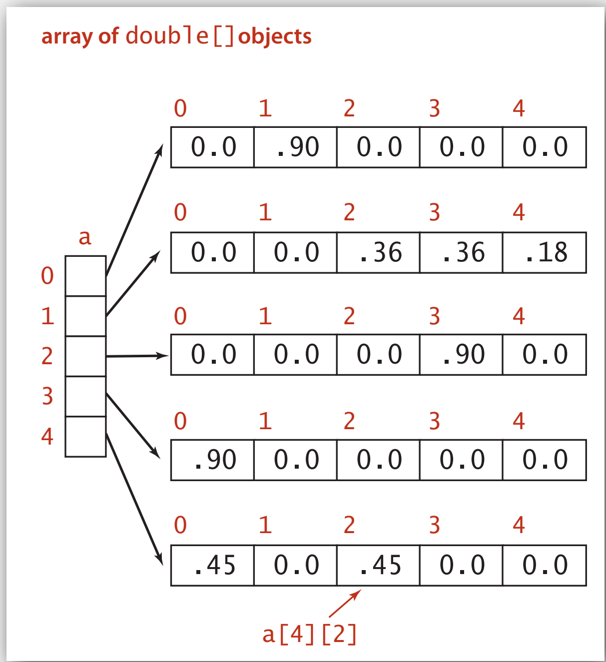
Matrix representations

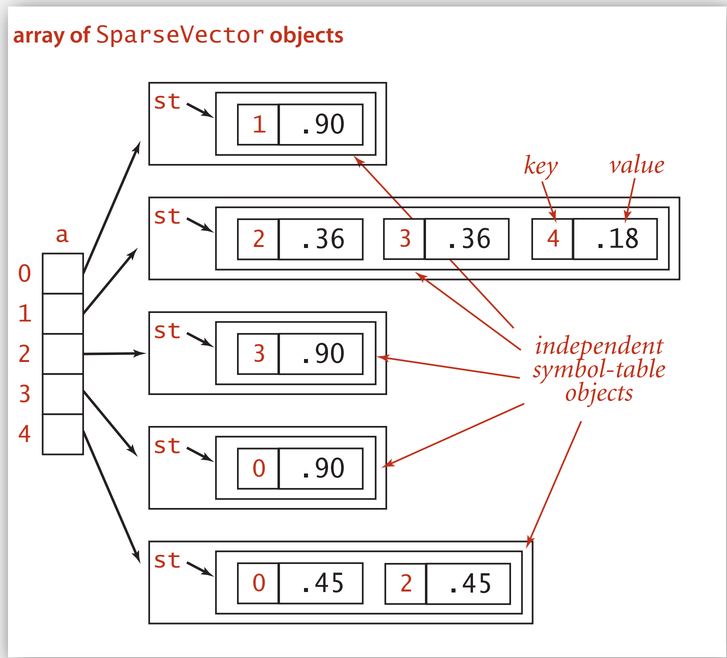
2D array (standard) representation: Each row of matrix is an array.

- Constant time access to elements.
- Space proportional to N^2.

Sparse representation: Each row of matrix is a sparse vector.

- Efficient access to elements.
- Space proportional to number of nonzeros (plus N).





Sparse matrix-vector multiplication

$$\begin{array}{cccc} \texttt{a[][]} & \texttt{x[]} & & \texttt{b[]} \\ \begin{bmatrix} 0 & .90 & 0 & 0 & 0 \\ 0 & 0 & .36 & .36 & .18 \\ 0 & 0 & 0 & .90 & 0 \\ .90 & 0 & 0 & 0 & 0 \\ .47 & 0 & .47 & 0 & 0 \end{bmatrix} & \begin{bmatrix} .05 \\ .04 \\ .36 \\ .37 \\ .19 \end{bmatrix} & = & \begin{bmatrix} .036 \\ .297 \\ .333 \\ .045 \\ .1927 \end{bmatrix} \end{array}$$

```
..
SparseVector[] a;
a = new SparseVector[N];
double[] x = new double[N];
double[] b = new double[N];
...
// Initialize a[] and x[]
...
for (int i = 0; i < N; i++)
   b[i] = a[i].dot(x);
```

one loop
linear running time
for sparse matrix

Searching challenge 2A:

Problem. IP lookups in a web monitoring device.

Assumption A. Billions of lookups, millions of distinct addresses.

Which searching method to use?

1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.

Searching challenge 2A

Problem. IP lookups in a web monitoring device.

Assumption A. Billions of lookups, millions of distinct addresses.

Which searching method to use?

1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow. ✓
4) Doesn't matter much, all fast enough.

total cost of insertions is $c*1000000^2$ = c*1,000,000,000,000 (way too much)

Searching challenge 2B

Problem. IP lookups in a web monitoring device.

Assumption B. Billions of lookups, thousands of distinct addresses.

Which searching method to use?

1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.

Searching challenge 2B

Problem. IP lookups in a web monitoring device.

Assumption B. Billions of lookups, thousands of distinct addresses.

Which searching method to use?

1) Sequential search in a linked list.
2) ✓ Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.

total cost of insertions is $c_1*1000^2 = c_1*1000000$ and dominated by $c_2*1000000000$ cost of lookups

Searching challenge 4

Problem. Spell checking for a book.

Assumptions. Dictionary has 25,000 words; book has 100,000+ words.

Which searching method to use?

1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.

Searching challenge 4

Problem. Spell checking for a book.

Assumptions. Dictionary has 25,000 words; book has 100,000+ words.

Which searching method to use?

1) Sequential search in a linked list.
✓ 2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.

easy to presort dictionary total cost of lookups is optimal c_2*1,500,000

Searching challenge 1A

Problem. Maintain symbol table of song names for an iPod.

Assumption A. Hundreds of songs.

Which searching method to use?

1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.

Searching challenge 1A

Problem. Maintain symbol table of song names for an iPod.

Assumption A. Hundreds of songs.

Which searching method to use?

1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough. ← $100^2 = 10{,}000$

Searching challenge 1B

Problem. Maintain symbol table of song names for an iPod.

Assumption B. Thousands of songs.

Which searching method to use?

1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.

Searching challenge 1B

Problem. Maintain symbol table of song names for an iPod.

Assumption B. Thousands of songs.

Which searching method to use?

1) Sequential search in a linked list.
2) Binary search in an ordered array. ← maybe, but 1000^2 = 1,000,000 so user might wait for complete rebuild of index
3) ✓ Need better method, all too slow.
4) Doesn't matter much, all fast enough.

Searching challenge 3

Problem. Frequency counts in “Tale of Two Cities.”

Assumptions. Book has 135,000+ words; about 10,000 distinct words.

Which searching method to use?

1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn’t matter much, all fast enough.

Searching challenge 3

Problem. Frequency counts in "Tale of Two Cities."

Assumptions. Book has 135,000+ words; about 10,000 distinct words.

Which searching method to use?

1) Sequential search in a linked list. ← total cost of searches: c_2*1,350,000,000
2) Binary search in an ordered array. ← maybe, but total cost of insertions is c_1*100,000,000
3) ✓ Need better method, all too slow.
4) Doesn't matter much, all fast enough.

Searching challenge 3 (revisited):

Problem. Frequency counts in "Tale of Two Cities"

Assumptions. Book has 135,000+ words; about 10,000 distinct words.

Which searching method to use?

1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.
5) ✓ BSTs.

insertion cost < 10000 * 1.38 * lg 10000 < .2 million

lookup cost < 135000 * 1.38 * lg 10000 < 2.5 million

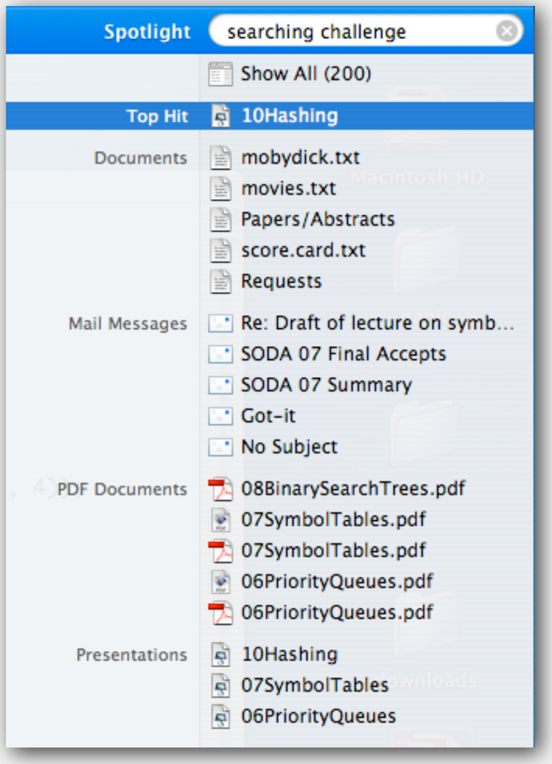
Searching challenge 5

Problem. Index for a PC or the web.

Assumptions. 1 billion++ words to index.

Which searching method to use?

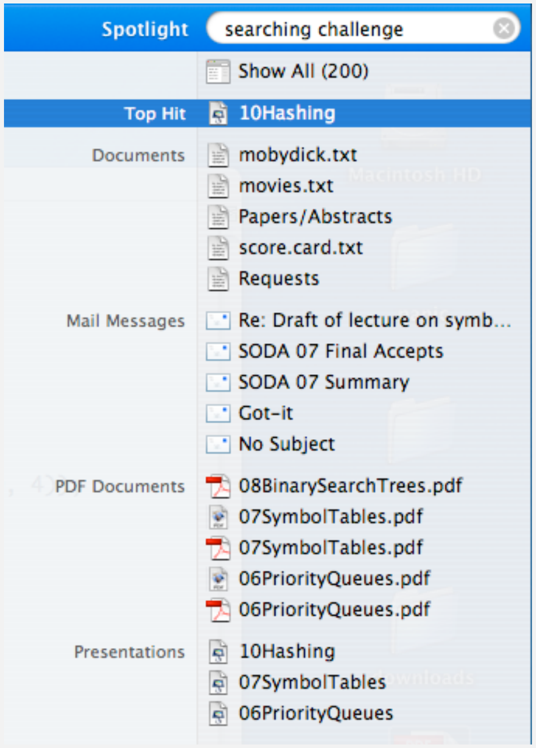
- Hashing
- Red-black-trees
- Doesn't matter much.



Searching challenge 5

Problem. Index for a PC or the web.

Assumptions. 1 billion++ words to index.



Which searching method to use?

✓ • Hashing
• Red-black-trees ← too much space
• Doesn't matter much.

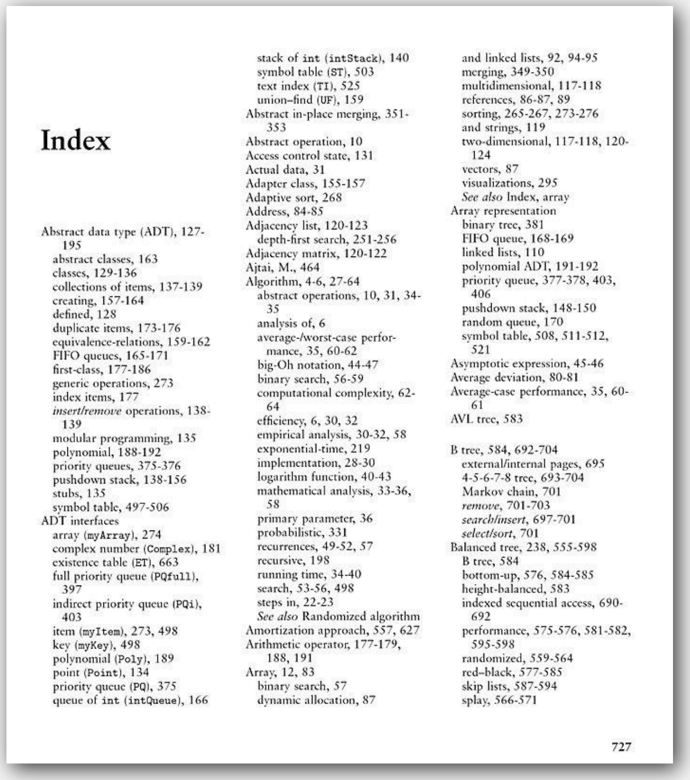
Solution. Symbol table with:
• Key = query string.
• Value = set of pointers to files.

sort the (relatively few) search hits

Searching challenge 6

Problem. Index for an e-book.

Assumptions. Book has 100,000+ words.



Index

727

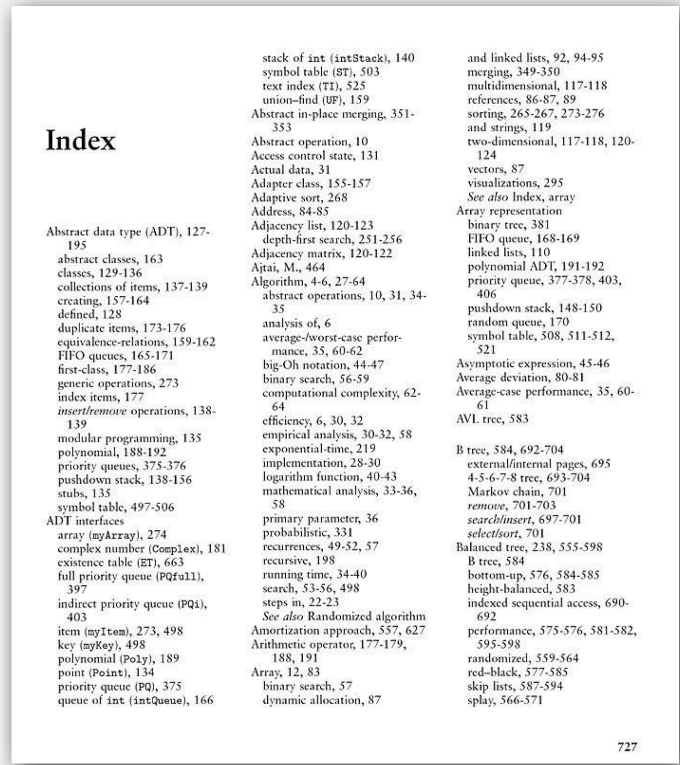
Which searching method to use?

1. Hashing
2. Red-black-tree
3. Doesn't matter much.

Searching challenge 6

Problem. Index for an e-book.

Assumptions. Book has 100,000+ words.



Index

727

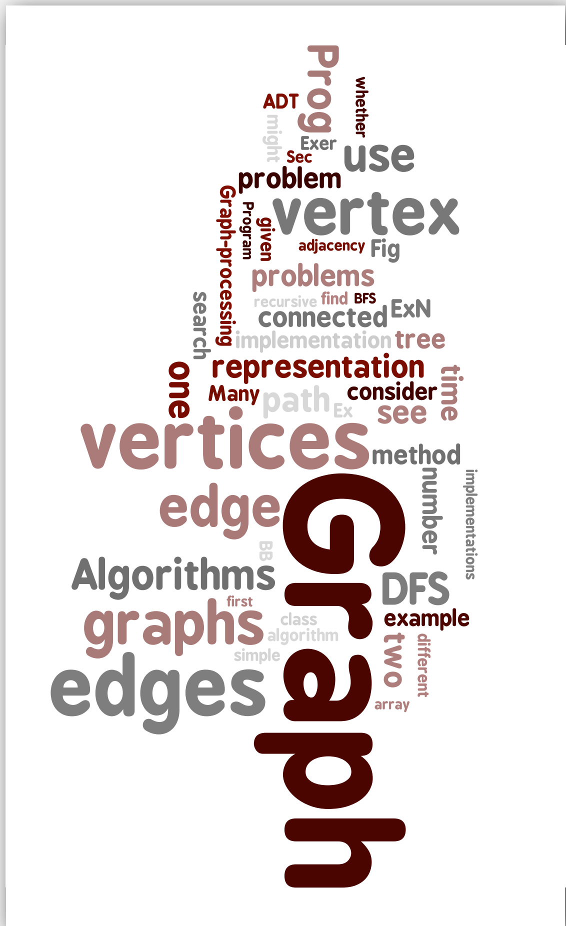
Which searching method to use?

1. Hashing
2. ✓ Red-black-tree ← need ordered iteration
3. Doesn't matter much.

Solution. Symbol table with:

- Key = index term.
- Value = ordered set of pages on which term appears.

4.1 Undirected Graphs



- graph API
- maze exploration
- depth-first search
- breadth-first search
- connected components
- challenges

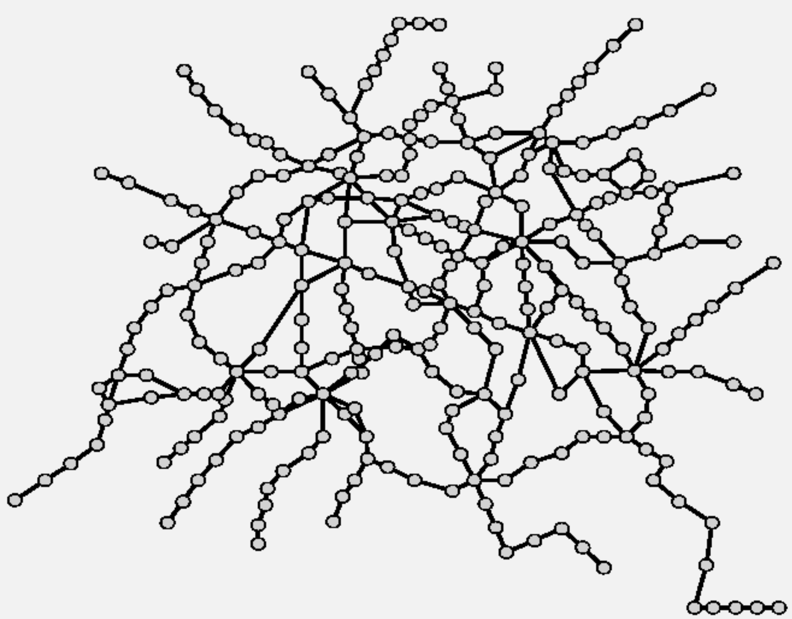
References: Algorithms in Java (Part 5), 3rd edition, Chapters 17 and 18

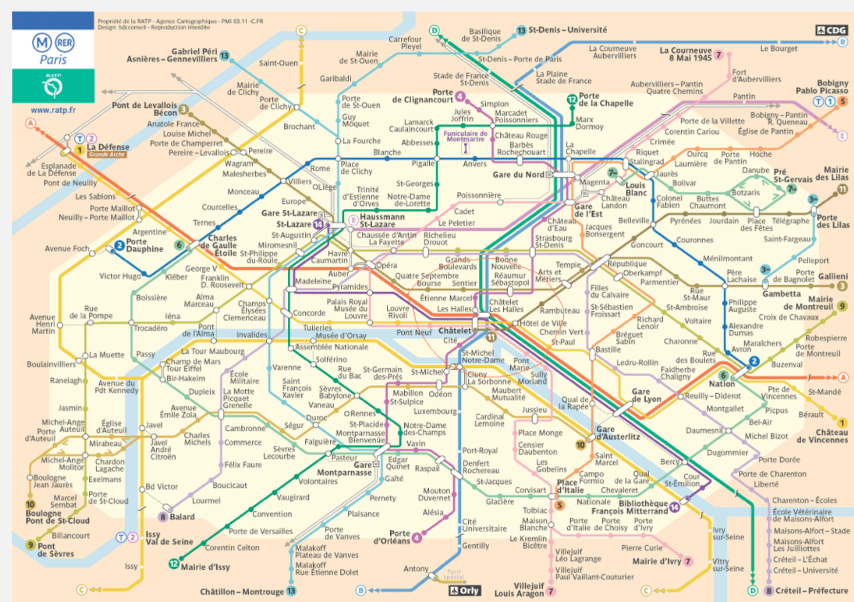
Undirected graphs

Graph. Set of vertices connected pairwise by edges.

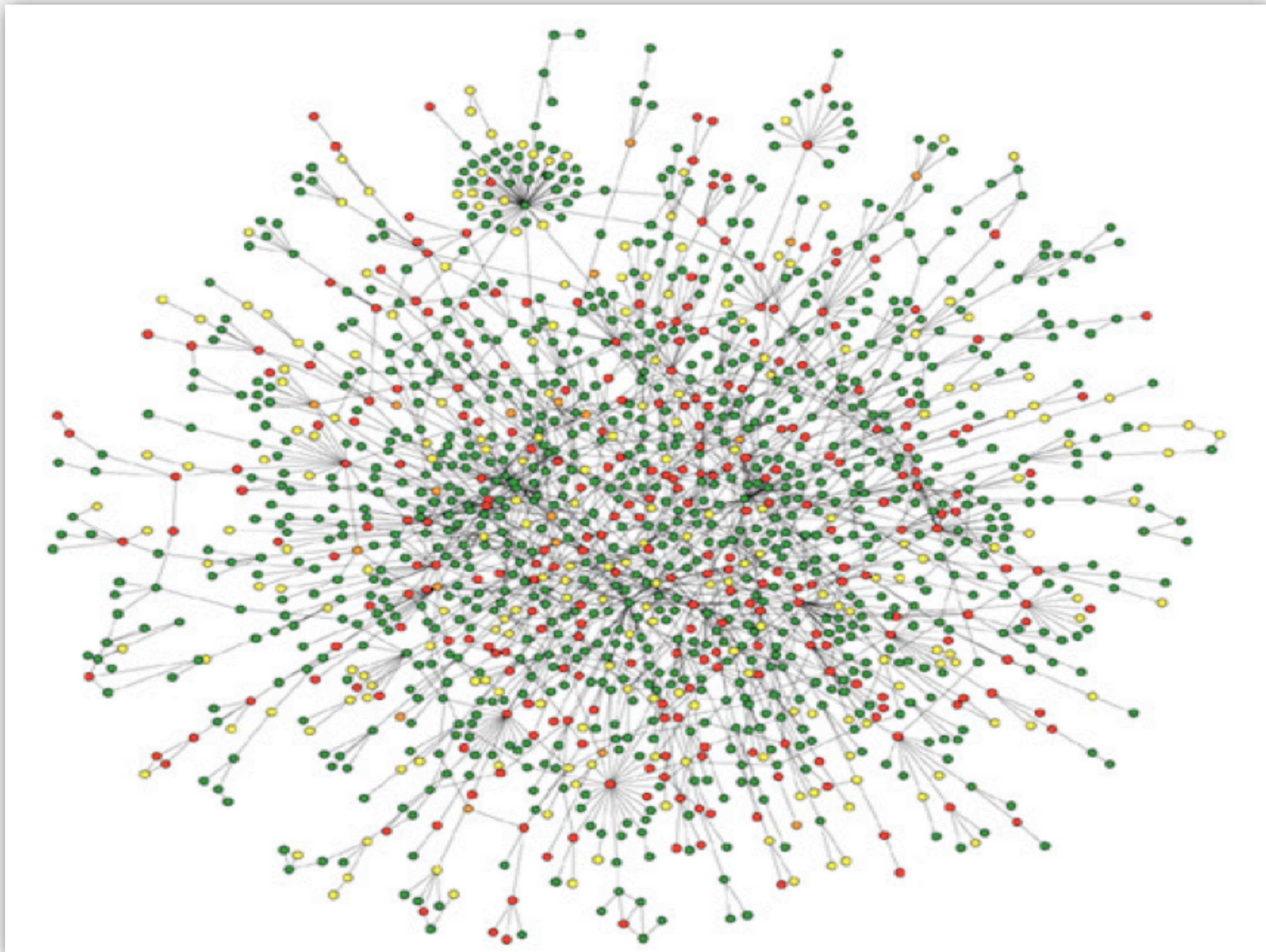
Why study graph algorithms?

- Interesting and broadly useful abstraction.
- Challenging branch of computer science and discrete math.
- Hundreds of graph algorithms known.
- Thousands of practical applications.



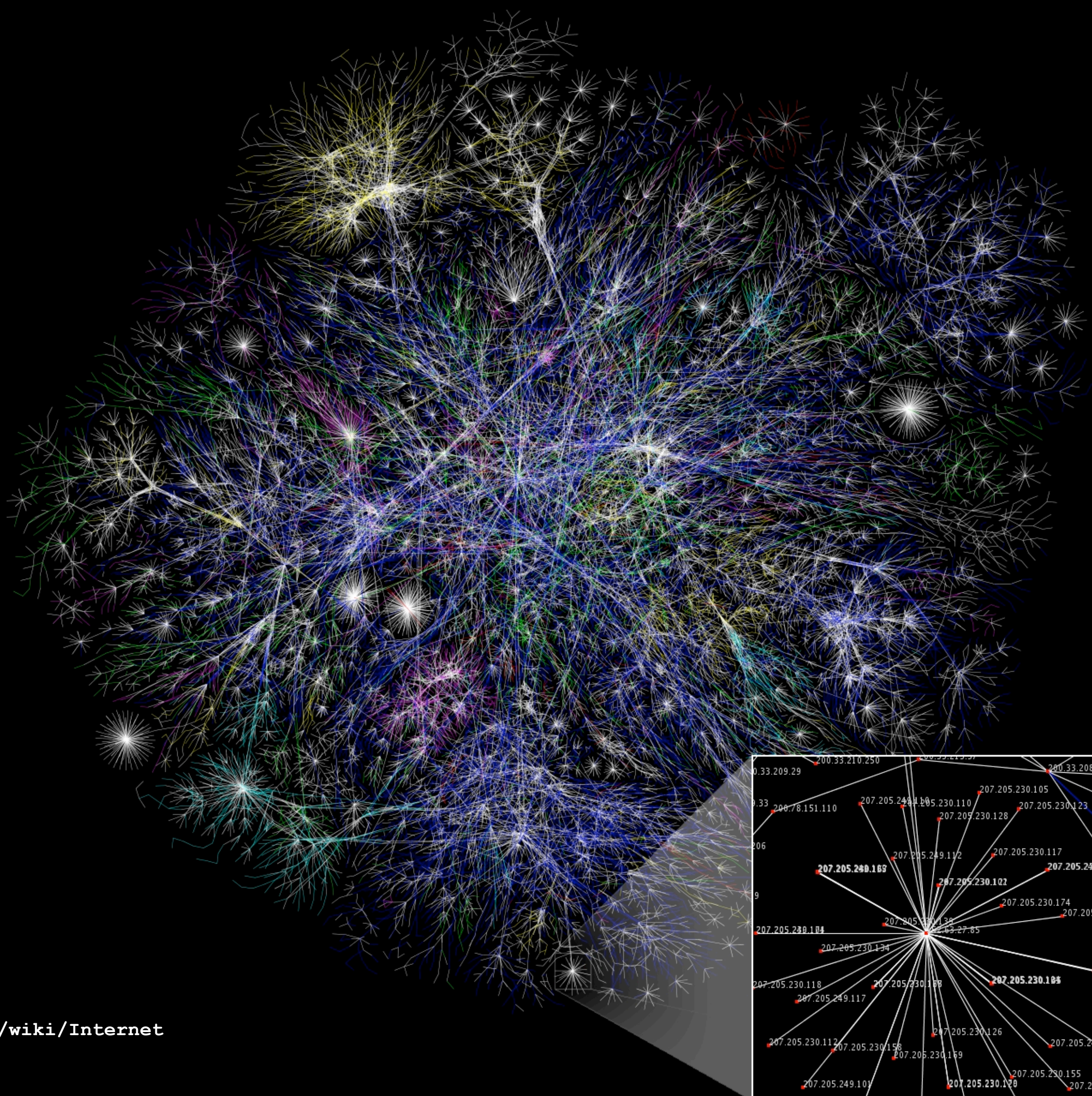


Protein interaction network



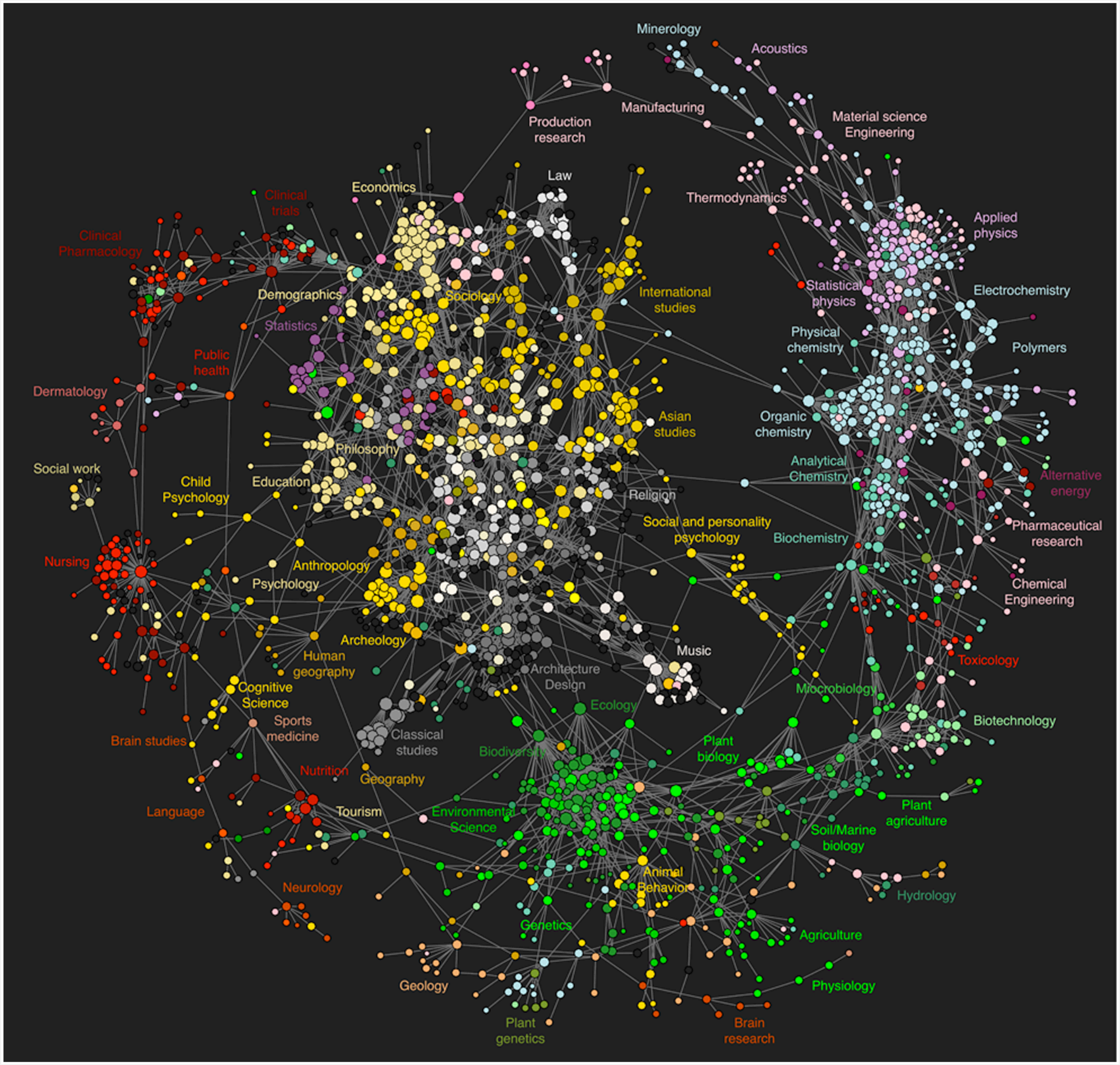
Reference: Jeong et al, Nature Review | Genetics

The Internet as mapped by the Opte Project



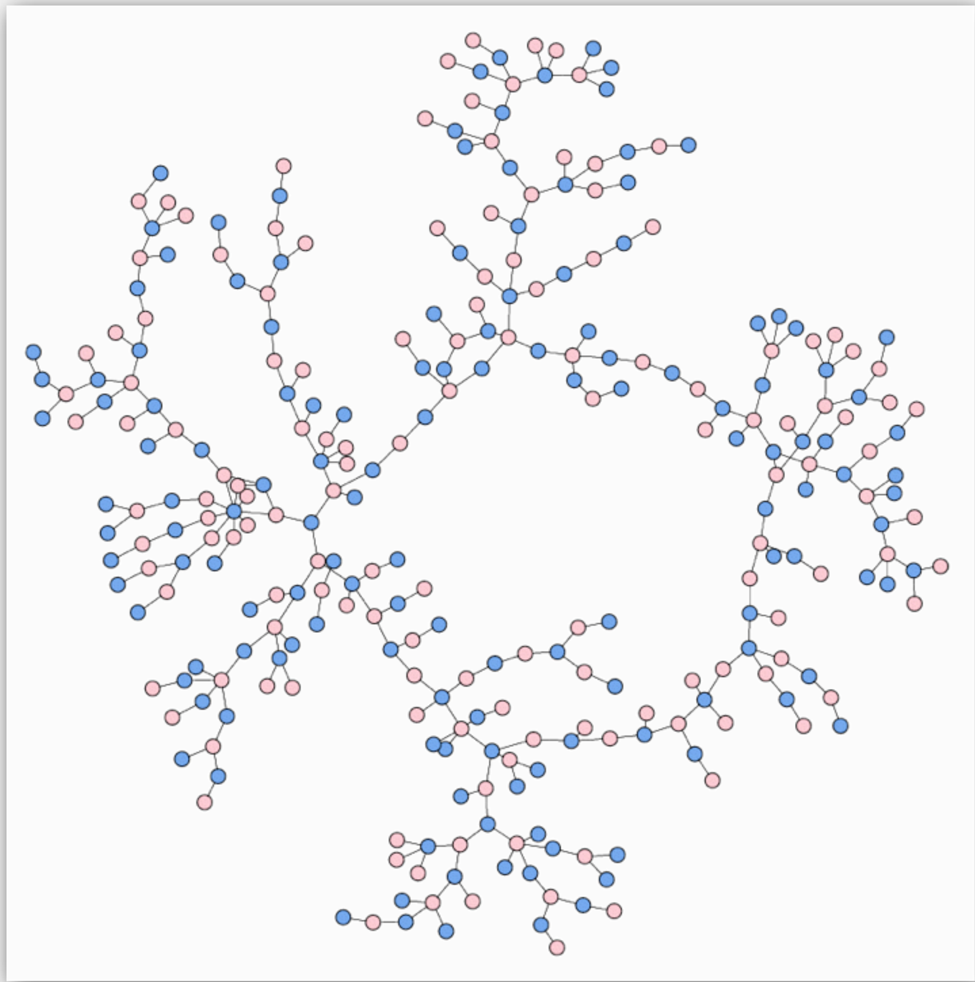
http://en.wikipedia.org/wiki/Internet

Map of science clickstreams



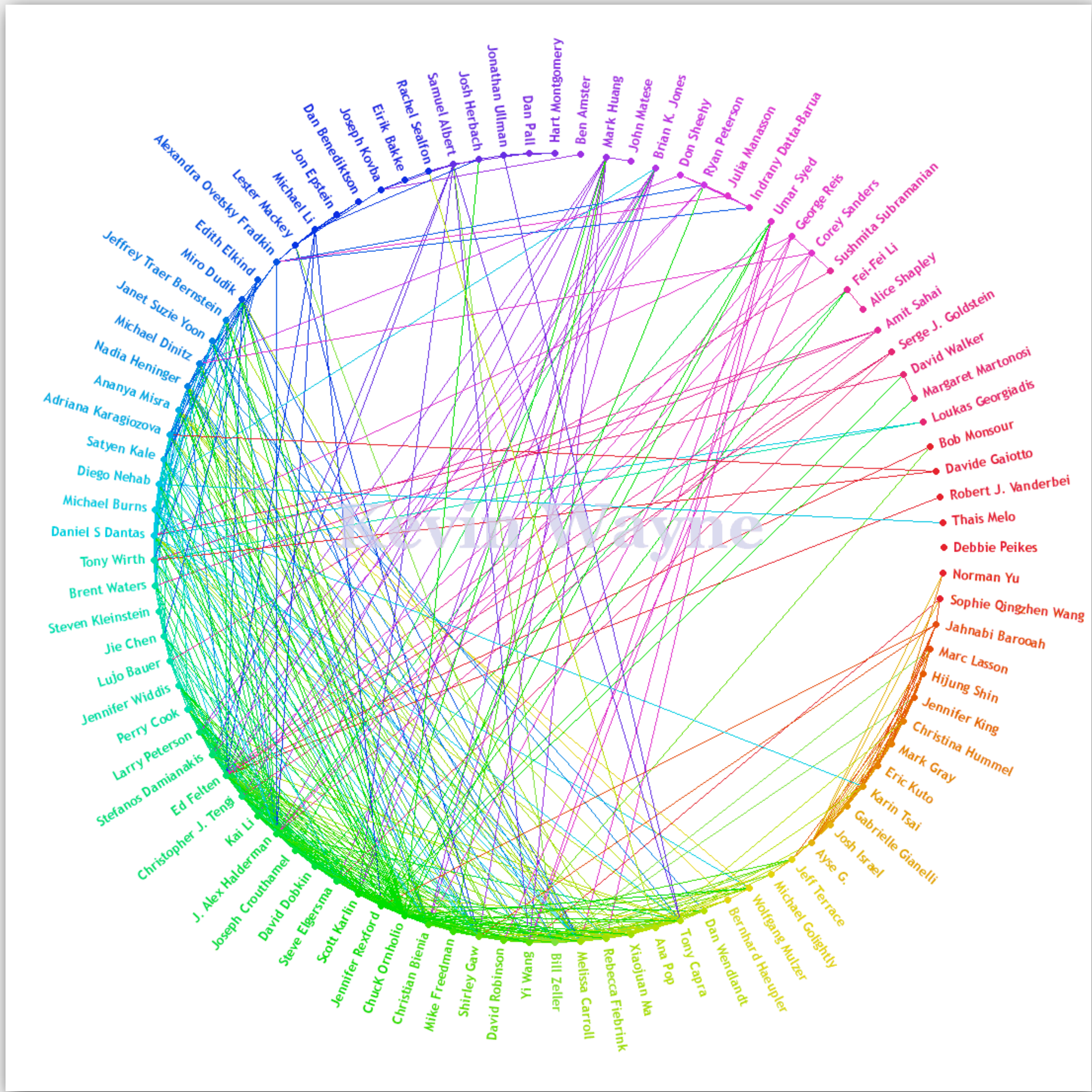
http://www.plosone.org/article/info:doi/10.1371/journal.pone.0004803

High-school dating

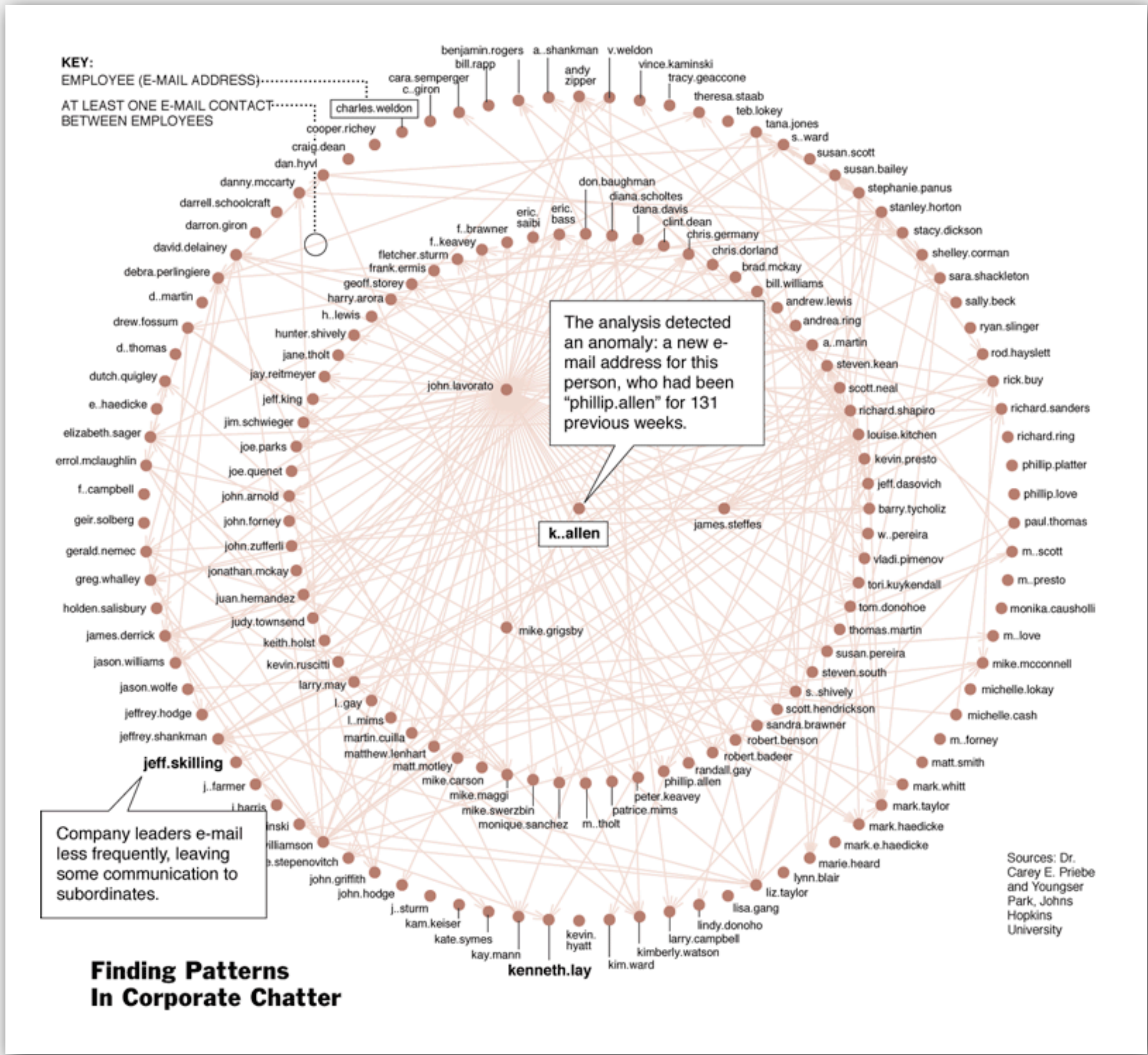


Reference: Bearman, Moody and Stovel, 2004
image by Mark Newman

Kevin's facebook friends (Princeton network)



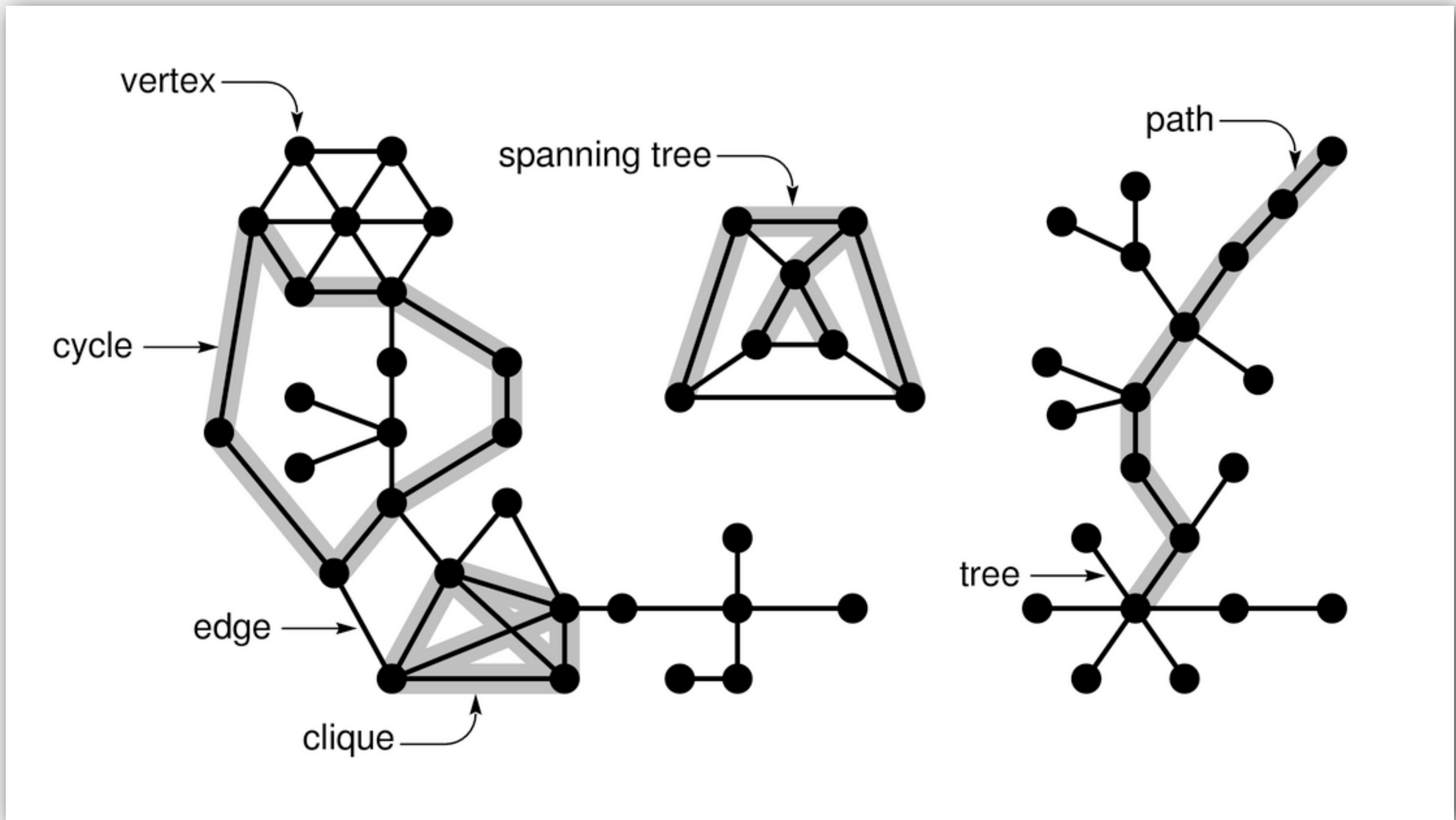
One week of Enron emails



Graph applications

graph	vertex	edge
communication	telephone, computer	fiber optic cable
circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transactions
transportation	street intersection, airport	highway, airway route
internet	class C network	connection
game	board position	legal move
social relationship	person, actor	friendship, movie cast
neural network	neuron	synapse
protein network	protein	protein-protein interaction
chemical compound	molecule	bond

Graph terminology



Some graph-processing problems

Path. Is there a path between s and t?

Shortest path. What is the shortest path between s and t?

Cycle. Is there a cycle in the graph?

Euler tour. Is there a cycle that uses each edge exactly once?

Hamilton tour. Is there a cycle that uses each vertex exactly once?

Connectivity. Is there a way to connect all of the vertices?

MST. What is the best way to connect all of the vertices?

Biconnectivity. Is there a vertex whose removal disconnects the graph?

Planarity. Can you draw the graph in the plane with no crossing edges?

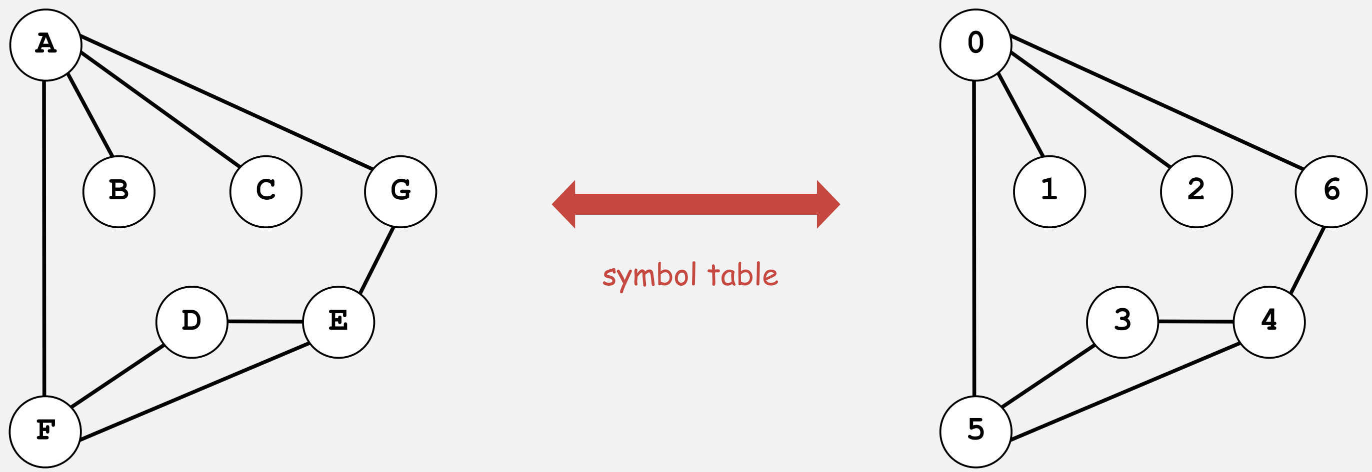
Graph isomorphism. Do two adjacency matrices represent the same graph?

Challenge. Which of these problems are easy? difficult? intractable?

- graph API
- maze exploration
- depth-first search
- breadth-first search
- connected components
- challenges

Vertex representation.

- This lecture: use integers between 0 and V-1.
- Real world: convert between names and integers with symbol table.



Issues. Parallel edges, self-loops.

Graph API

`public class Graph`		*graph data type*
	`Graph(int V)`	*create an empty graph with V vertices*
	`Graph(In in)`	*create a graph from input stream*
`void`	`addEdge(int v, int w)`	*add an edge v-w*
`Iterable<Integer>`	`adj(int v)`	*return an iterator over the neighbors of v*
`int`	`V()`	*return number of vertices*

```
In in = new In();
Graph G = new Graph(in);
```

read graph from standard input

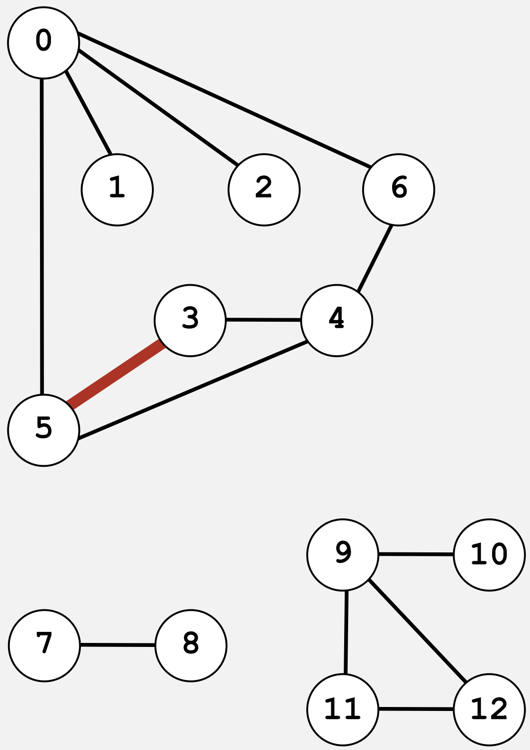
```
for (int v = 0; v < G.V(); v++)
   for (int w : G.adj(v))
      /* process edge v-w */
```

process both v-w and w-v

```
% more tiny.txt
7
0 1
0 2
0 5
0 6
3 4
3 5
4 6
```

Set of edges representation

Maintain a list of the edges (linked list or array).

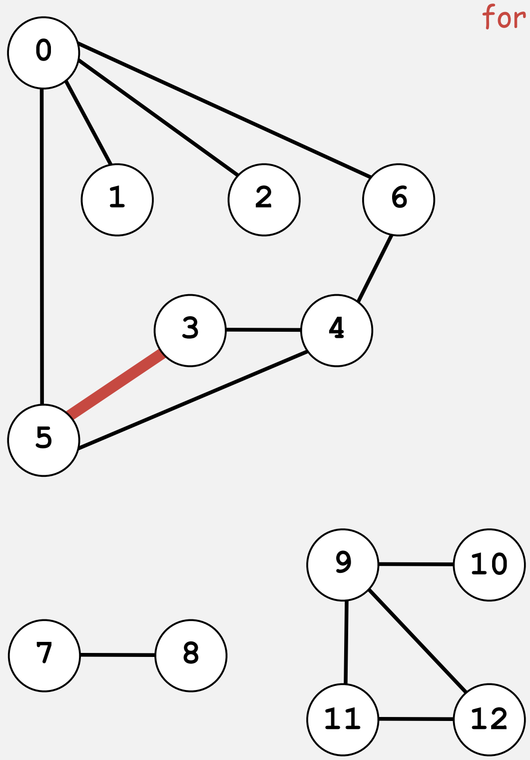


0	1
0	2
0	5
0	6
3	4
3	5
4	6
7	8
9	10
9	11
9	12

Adjacency-matrix representation

Maintain a two-dimensional V-by-V boolean array;

for each edge v-w in graph: `adj[v][w] = adj[w][v] = true`.



two entries
for each edge

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	0	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

Adjacency-matrix representation: Java implementation

```
public class Graph
{
   private final int V;
   private final boolean[][] adj;

   public Graph(int V)
   {
      this.V = V;
      adj = new boolean[V][V];
   }

   public void addEdge(int v, int w)
   {
      adj[v][w] = true;
      adj[w][v] = true;
   }

   public Iterable<Integer> adj(int v)
   {  return new AdjIterator(v);   }
}
```

adjacency matrix

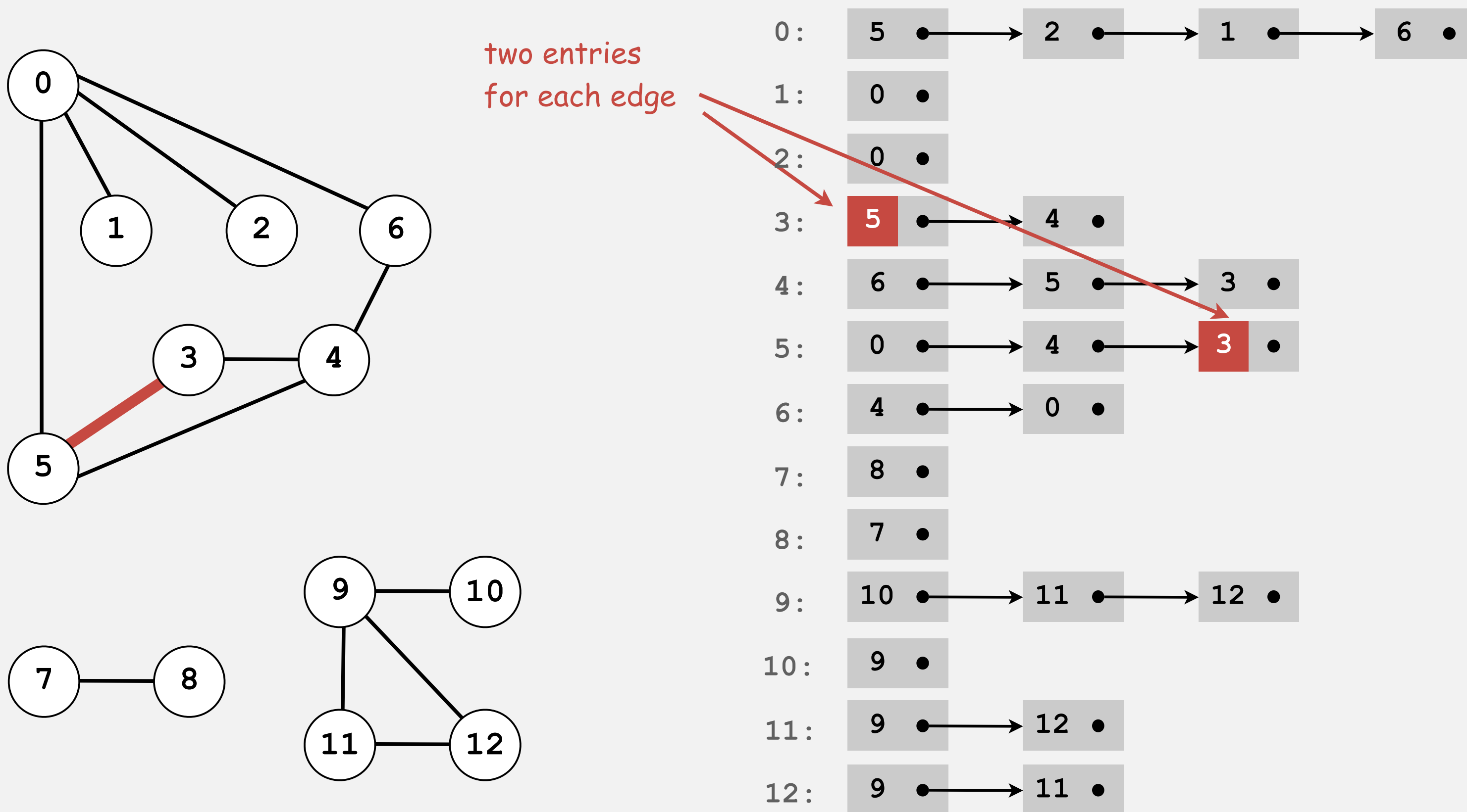
create empty graph
with V vertices

add edge v-w
(no parallel edges)

iterator for v's neighbors
(code for `AdjIterator` omitted)

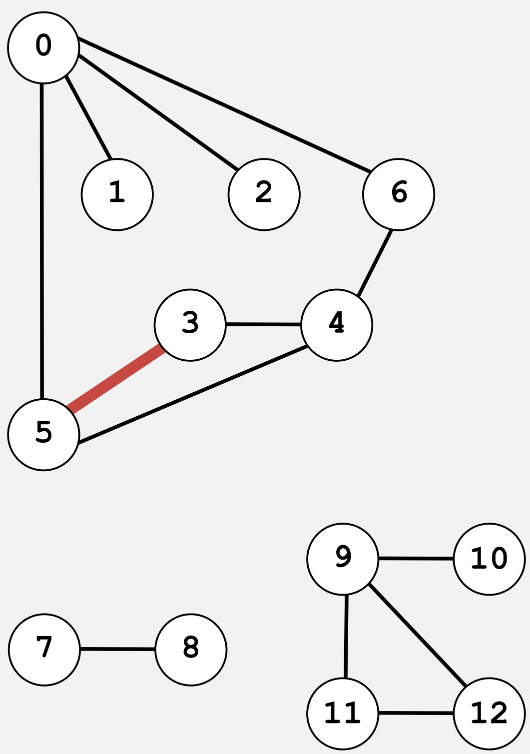
Adjacency-list representation

Maintain vertex-indexed array of lists (implementation omitted).



Adjacency-set graph representation

Maintain vertex-indexed array of sets.



0:	{ 1 2 5 6 }
1:	{ 0 }
2:	{ 0 }
3:	{ 4, 5 }
4:	{ 3, 5, 6 }
5:	{ 0, 3, 4 }
6:	{ 0, 4 }
7:	{ 8 }
8:	{ 7 }
9:	{ 10, 11, 12 }
10:	{ 9 }
11:	{ 9, 12 }
12:	{ 9, 11 }

two entries for each edge

Adjacency-set representation: Java implementation

```
public class Graph
{
   private final int V;
   private final SET<Integer>[] adj;

   public Graph(int V)
   {
      this.V = V;
      adj = (SET<Integer>[]) new SET[V];
      for (int v = 0; v < V; v++)
         adj[v] = new SET<Integer>();
   }

   public void addEdge(int v, int w)
   {
      adj[v].add(w);
      adj[w].add(v);
   }

   public Iterable<Integer> adj(int v)
   {  return adj[v];  }
}
```

adjacency sets

create empty graph with V vertices

add edge v-w (no parallel edges)

iterator for v's neighbors

Graph representations

In practice. Use adjacency-set (or adjacency-list) representation.

- Algorithms based on iterating over edges incident to v.
- Real-world graphs tend to be "sparse."

huge number of vertices,
small average vertex degree

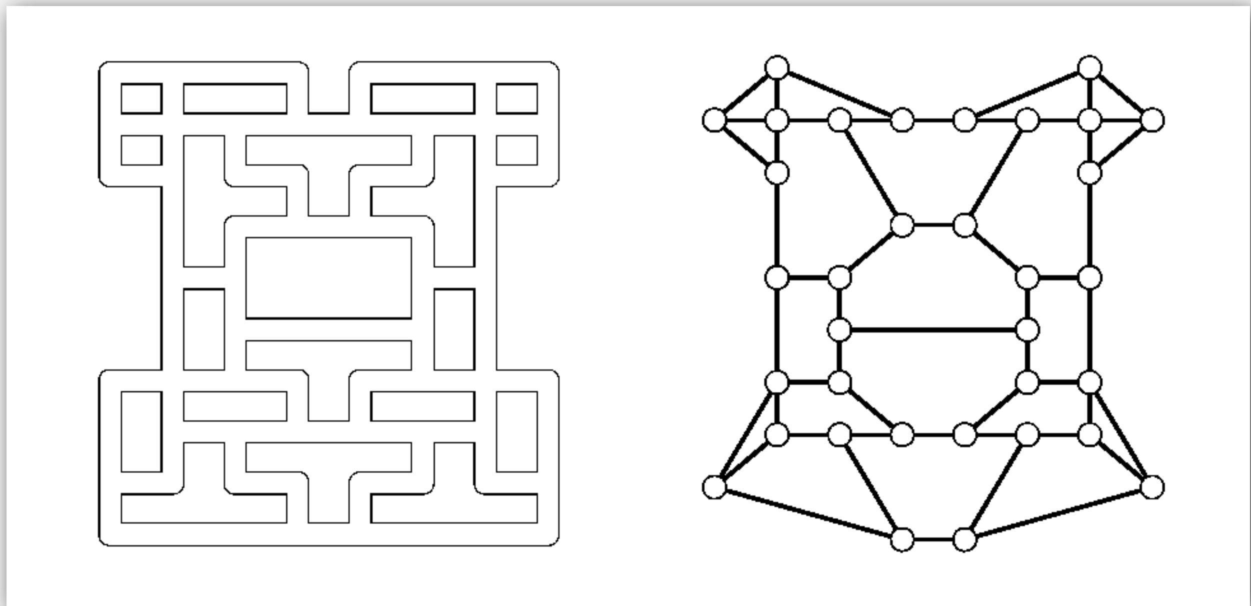
representation	space	insert edge	edge between v and w?	iterate over edges incident to v?
list of edges	E	E	E	E
adjacency matrix	V^2	1	1	V
adjacency list	E + V	degree(v)	degree(v)	degree(v)
adjacency set	E + V	log (degree(v))	log (degree(v))	degree(v)

- graph API
- **maze exploration**
- depth-first search
- breadth-first search
- connected components
- challenges

Maze exploration

Maze graphs.

- Vertex = intersection.
- Edge = passage.



Goal. Explore every passage in the maze.

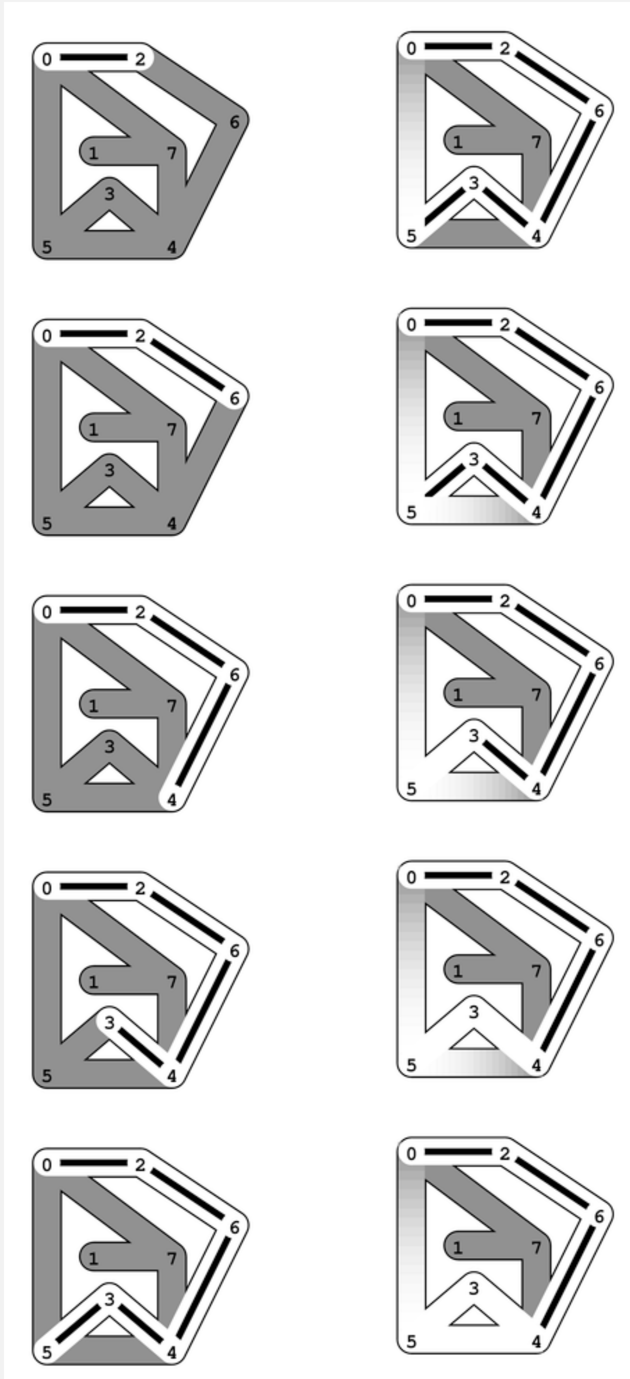
Trémaux maze exploration

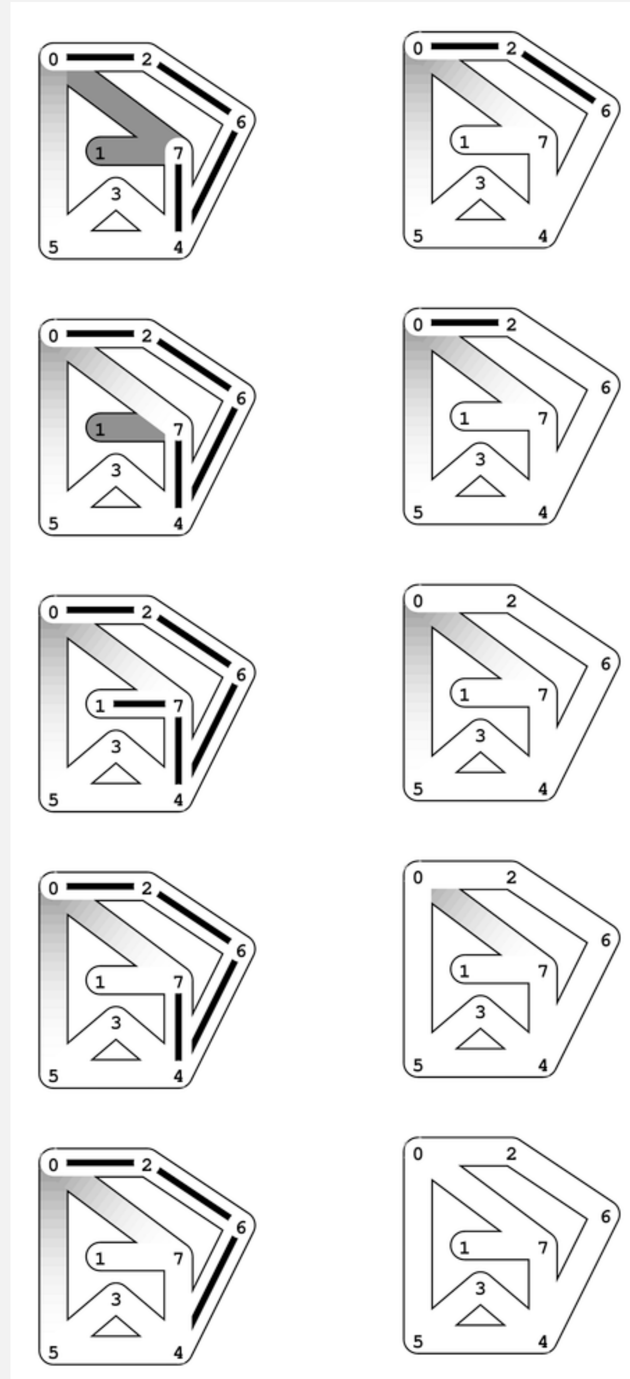
Algorithm.

- Unroll a ball of string behind you.
- Mark each visited intersection by turning on a light.
- Mark each visited passage by opening a door.

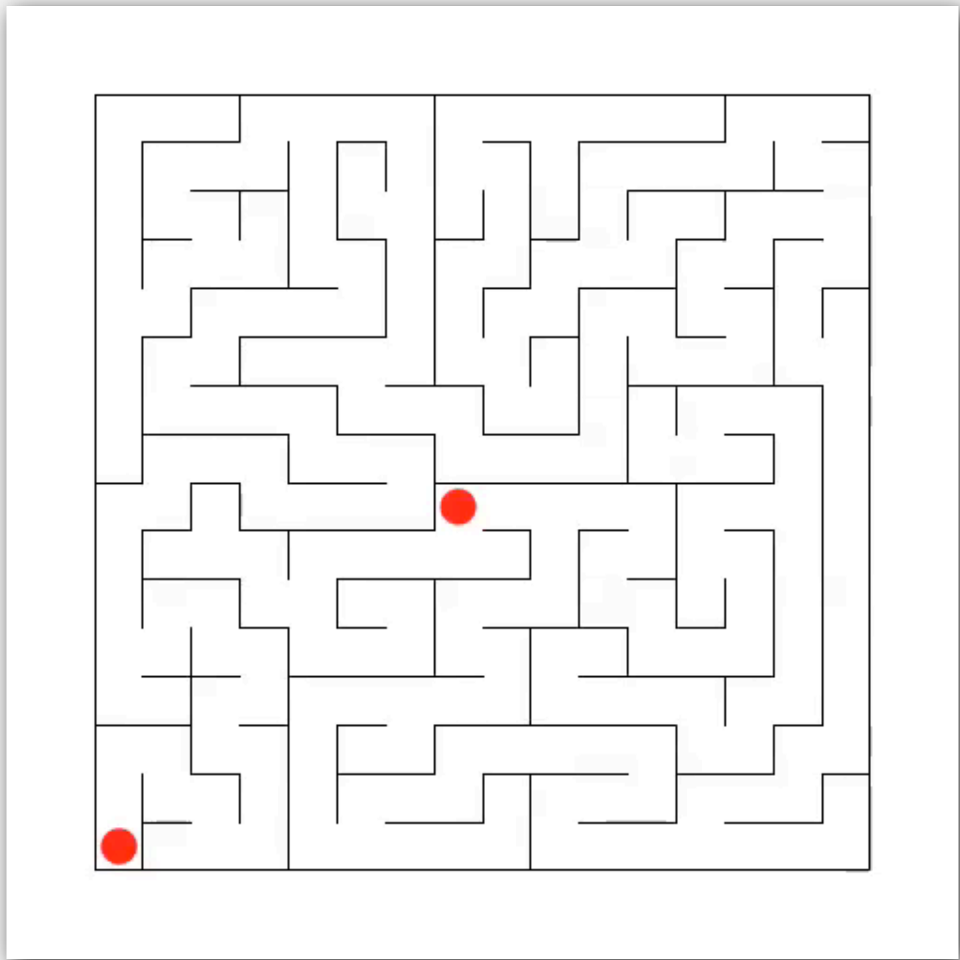
First use? Theseus entered labyrinth to kill the monstrous Minotaur; Ariadne held ball of string.

Claude Shannon (with Theseus mouse)

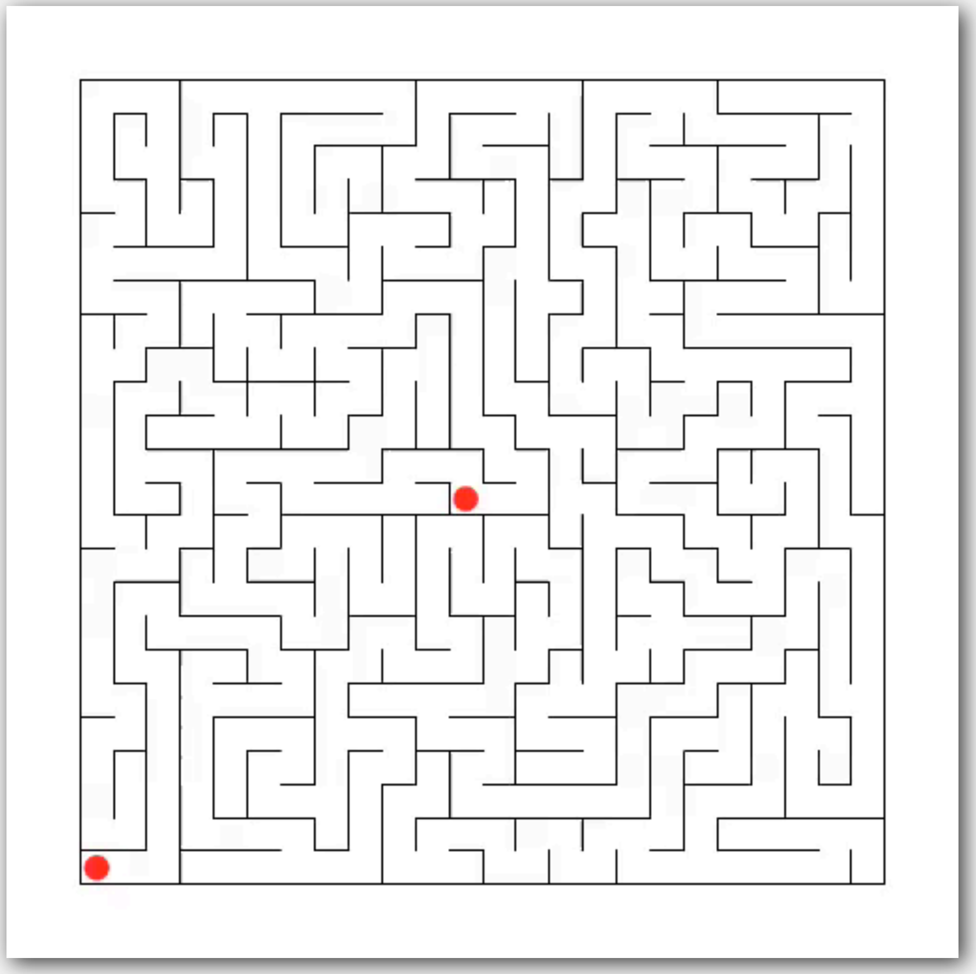




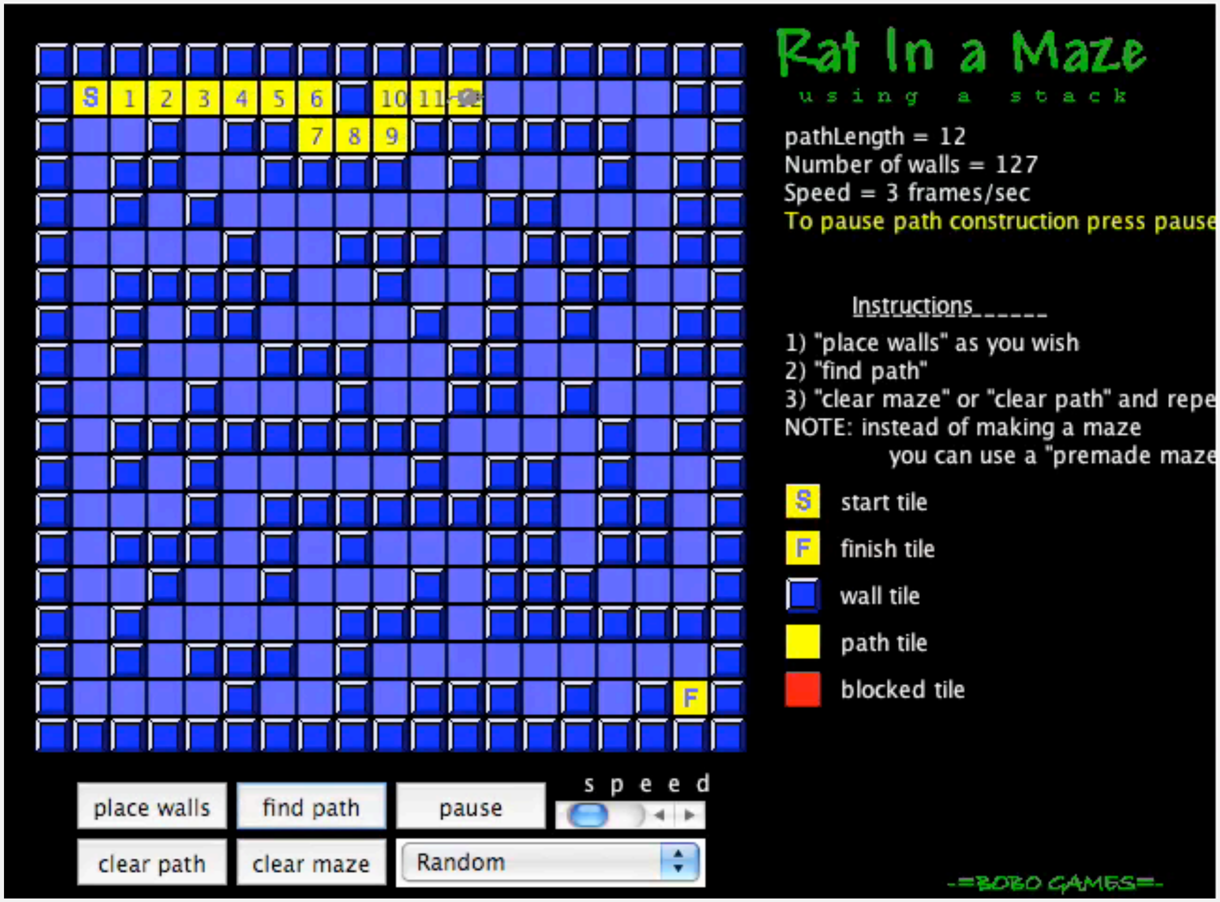
Maze exploration



Maze exploration



Rat in a maze


Rat In a Maze
using a stack
S
1
2
3
4
5
6
7
8
9
10
11
F
pathLength = 12
Number of walls = 127
Speed = 3 frames/sec
To pause path construction press pause
Instructions
1) "place walls" as you wish
2) "find path"
3) "clear maze" or "clear path" and repe
NOTE: instead of making a maze
you can use a "premade maze
start tile
finish tile
wall tile
path tile
blocked tile
place walls
find path
pause
speed
clear path
clear maze
Random
-=BOBO GAMES=-

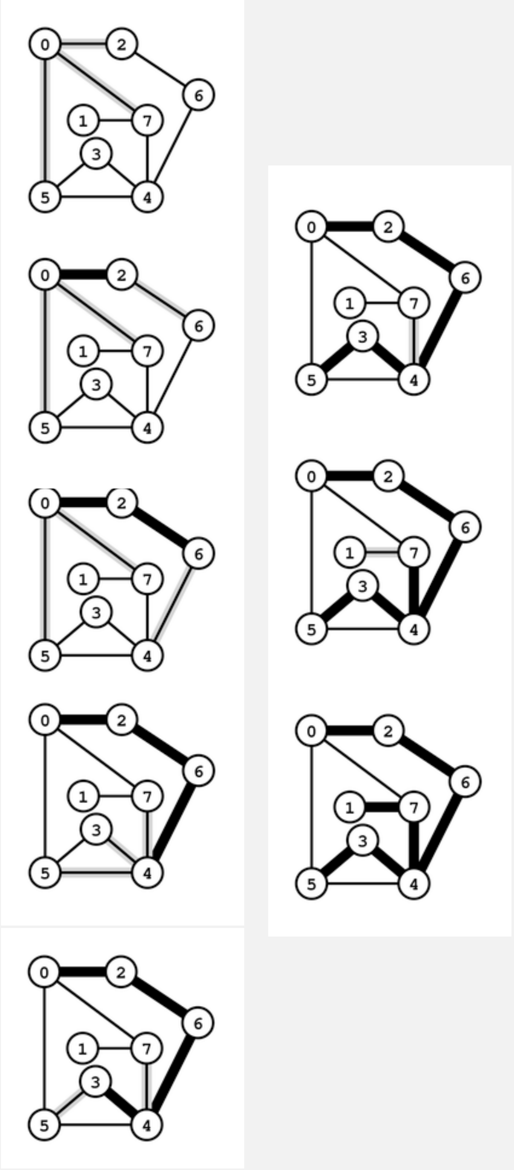
Depth-first search

Goal. Systematically search through a graph.

Idea. Mimic maze exploration.

DFS (to visit a vertex s)

Mark s as visited.
Recursively visit all unmarked vertices v adjacent to s.



Running time.

- O(E) since each edge examined at most twice.
- Usually less than V in real-world graphs.

- Typical applications.
- Find all vertices connected to a given s.
- Find a path from s to t.

Design pattern for graph processing

Design goal. Decouple graph data type from graph processing.

```
// print all vertices connected to s
In in = new In(args[0]);
Graph G = new Graph(in);
int s = 0;
DFSearcher dfs = new DFSearcher(G, s);
for (int v = 0; v < G.V(); v++)
   if (dfs.isConnected(v))
      StdOut.println(v);
```

Typical client program.

- Create a `Graph`.
- Pass the `Graph` to a graph-processing routine, e.g., `DFSearcher`.
- Query the graph-processing routine for information.

Depth-first search (connectivity)

```
public class DFSearcher
{
   private boolean[] marked;

   public DFSearcher(Graph G, int s)
   {
      marked = new boolean[G.V()];
      dfs(G, s);
   }

   private void dfs(Graph G, int v)
   {
      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w]) dfs(G, w);
   }

   public boolean isConnected(int v)
   {  return marked[v];  }
}
```

true if connected to s

constructor marks vertices connected to s

recursive DFS does the work

client can ask whether any vertex is connected to s

Photoshop "magic wand"

Graph-processing challenge 1

Problem. Flood fill.

Assumptions. Picture has millions to billions of pixels.

How difficult?

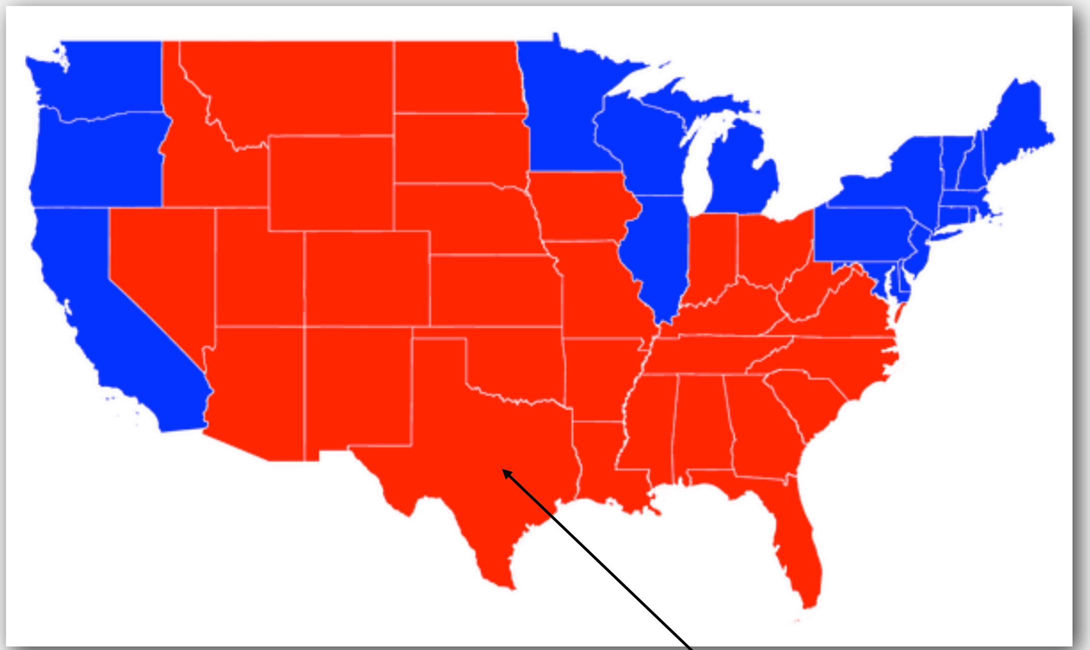
- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

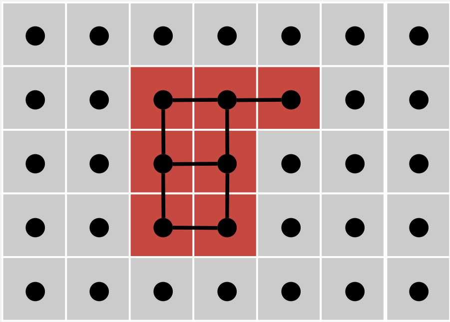
Connectivity application: flood fill

Change color of entire blob of neighboring red pixels to blue.

Build a grid graph.

- Vertex: pixel.
- Edge: between two adjacent red pixels.
- Blob: all pixels connected to given pixel.





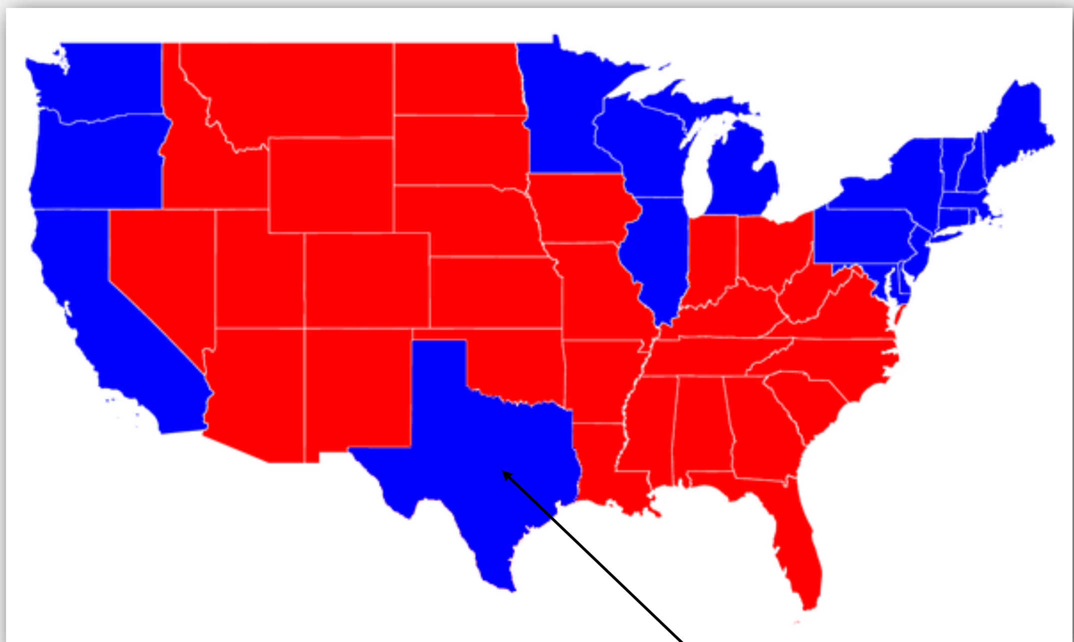
recolor red blob to blue

Connectivity application: flood fill

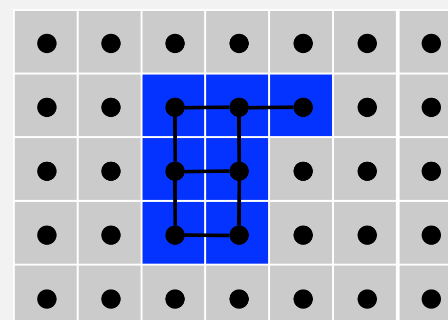
Change color of entire blob of neighboring red pixels to blue.

Build a grid graph.

- Vertex: pixel.
- Edge: between two adjacent red pixels.
- Blob: all pixels connected to given pixel.



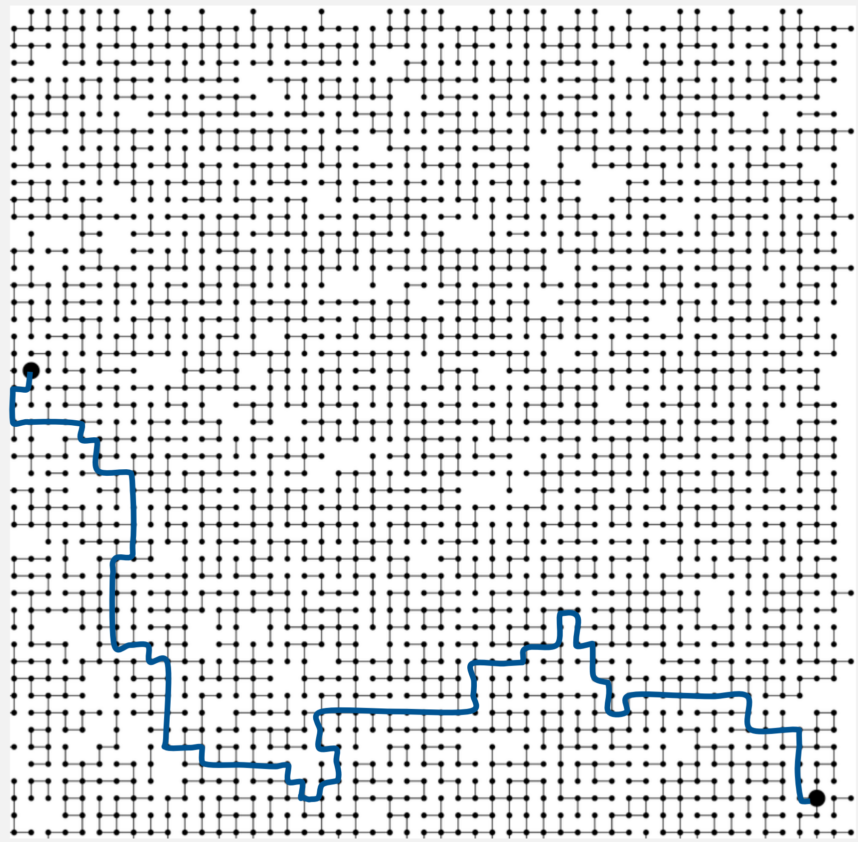
recolor red blob to blue



Graph-processing challenge 2

Problem. Find a path from s to t ?

Assumption. Any path will do.



How difficult?

- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.

Paths in graphs: union find vs. DFS

Goal. Is there a path from s to t?

method	preprocessing time	query time	space
union-find	V + E log* V	log* V †	V
DFS	E + V	1	E + V

† amortized

If so, find one.

- Union-find: not much help (run DFS on connected subgraph).
- DFS: easy (see next slides).

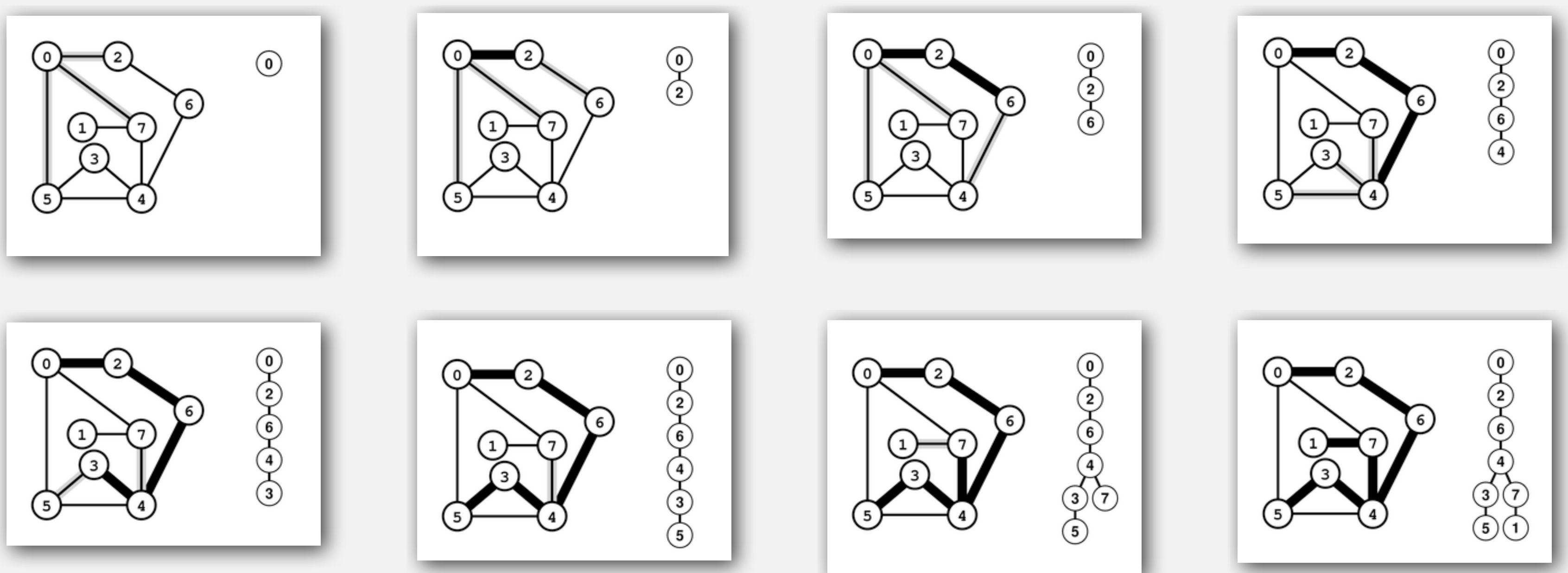
Union-find advantage. Can intermix queries and edge insertions.

DFS advantage. Can recover path itself in time proportional to its length.

Keeping track of paths with DFS

DFS tree. Upon visiting a vertex `v` for the first time, remember that you came from `pred[v]` (parent-link representation).

Retrace path. To find path between `s` and `v`, follow `pred[]` back from `v`.



Depth-first-search (pathfinding)

```
public class DFSearcher
{
   private int[] pred;
   ...
   public DFSearcher(Graph G, int s)
   {
      ...
      pred = new int[G.V()];
      for (int v = 0; v < G.V(); v++)
         pred[v] = -1;
      ...
   }
   private void dfs(Graph G, int v)
   {
      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w])
         {
            pred[w] = v;
            dfs(G, w);
         }
   }

   public Iterable<Integer> path(int v)
   {  /* see next slide */  }
}
```

add instance variable for parent-link representation of DFS tree

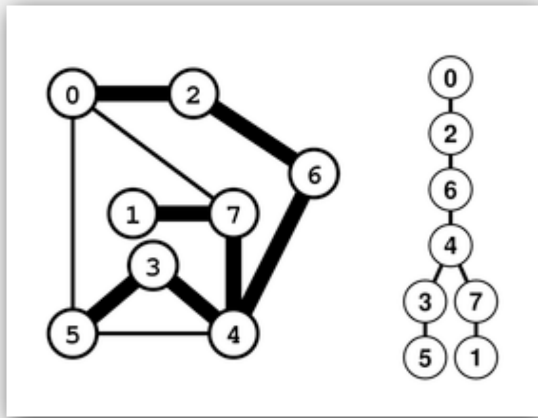
initialize it in the constructor

set parent link

add method for client to iterate through path

Depth-first-search (pathfinding iterator)

```
public Iterable<Integer> path(int v)
{
   Stack<Integer> path = new Stack<Integer>();
   while (v != -1 && marked[v])
   {
      path.push(v);
      v = pred[v];
   }
   return path;
}
```



DFS summary

Enables direct solution of simple graph problems.

✓ • Find path from s to t.
- Connected components (stay tuned).
- Euler tour (see book).
- Cycle detection (simple exercise).
- Bipartiteness checking (see book).

Basis for solving more difficult graph problems.

- Biconnected components (see book).
- Planarity testing (beyond scope).

- graph API
- maze exploration
- depth-first search
- **breadth-first search**
- connected components
- challenge

Breadth-first search

Depth-first search. Put unvisited vertices on a stack.

Breadth-first search. Put unvisited vertices on a queue.

Shortest path. Find path from s to t that uses fewest number of edges.

BFS (from source vertex s)

Put s onto a FIFO queue.

Repeat until the queue is empty:

- *remove the least recently added vertex v*
- *add each of v's unvisited neighbors to the queue, and mark them as visited.*

Property. BFS examines vertices in increasing distance from s.

Breadth-first search scaffolding

```
public class BFSearcher
{
   private int[] dist;

   public BFSearcher(Graph G, int s)
   {
      dist = new int[G.V()];
      for (int v = 0; v < G.V(); v++)
         dist[v] = G.V() + 1;
      dist[s] = 0;

      bfs(G, s);
   }

   public int distance(int v)
   {  return dist[v];  }

   private void bfs(Graph G, int s)
   {  /* See next slide */  }

}
```

distances from s

initialize distances

compute distances

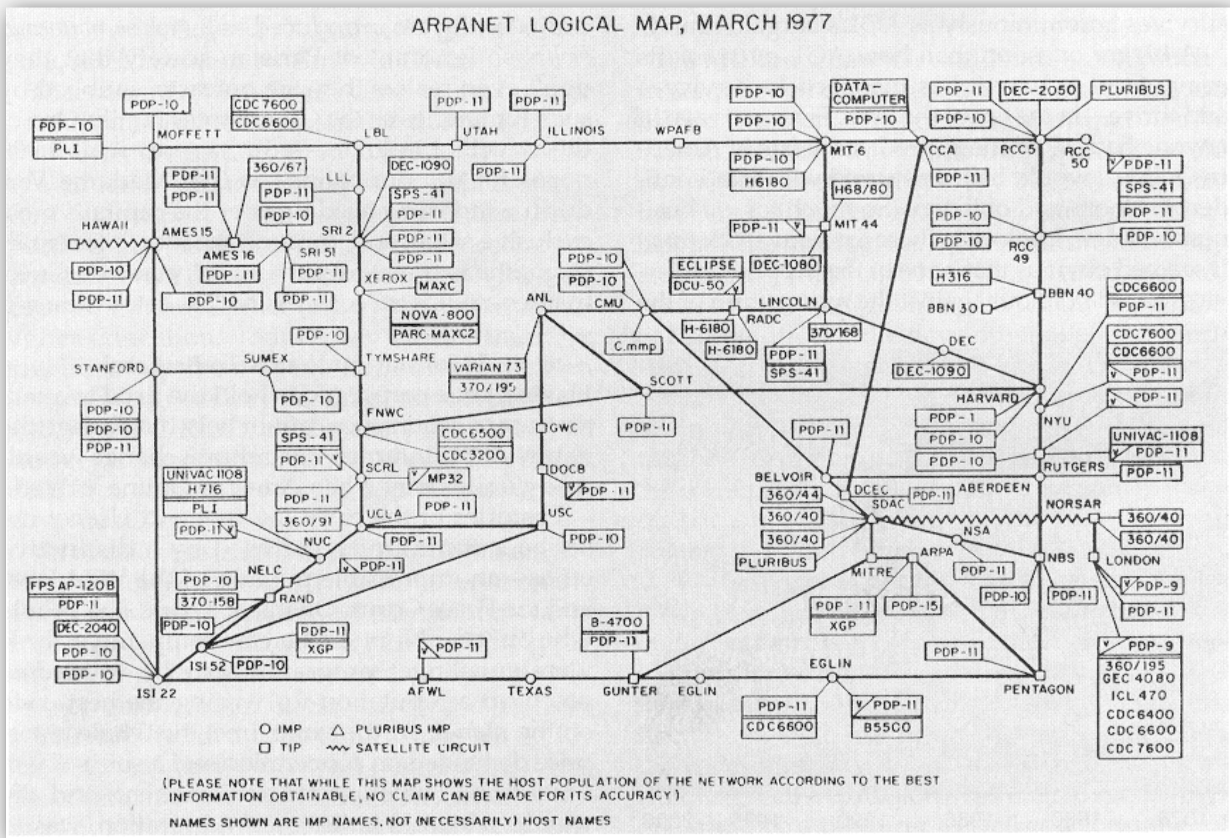
answer client query

Breadth-first search (compute shortest-path distances)

```
private void bfs(Graph G, int s)
{
   Queue<Integer> q = new Queue<Integer>();
   q.enqueue(s);
   while (!q.isEmpty())
   {
      int v = q.dequeue();
      for (int w : G.adj(v))
      {
         if (dist[w] > G.V())
         {
            q.enqueue(w);
            dist[w] = dist[v] + 1;
         }
      }
   }
}
```

BFS application

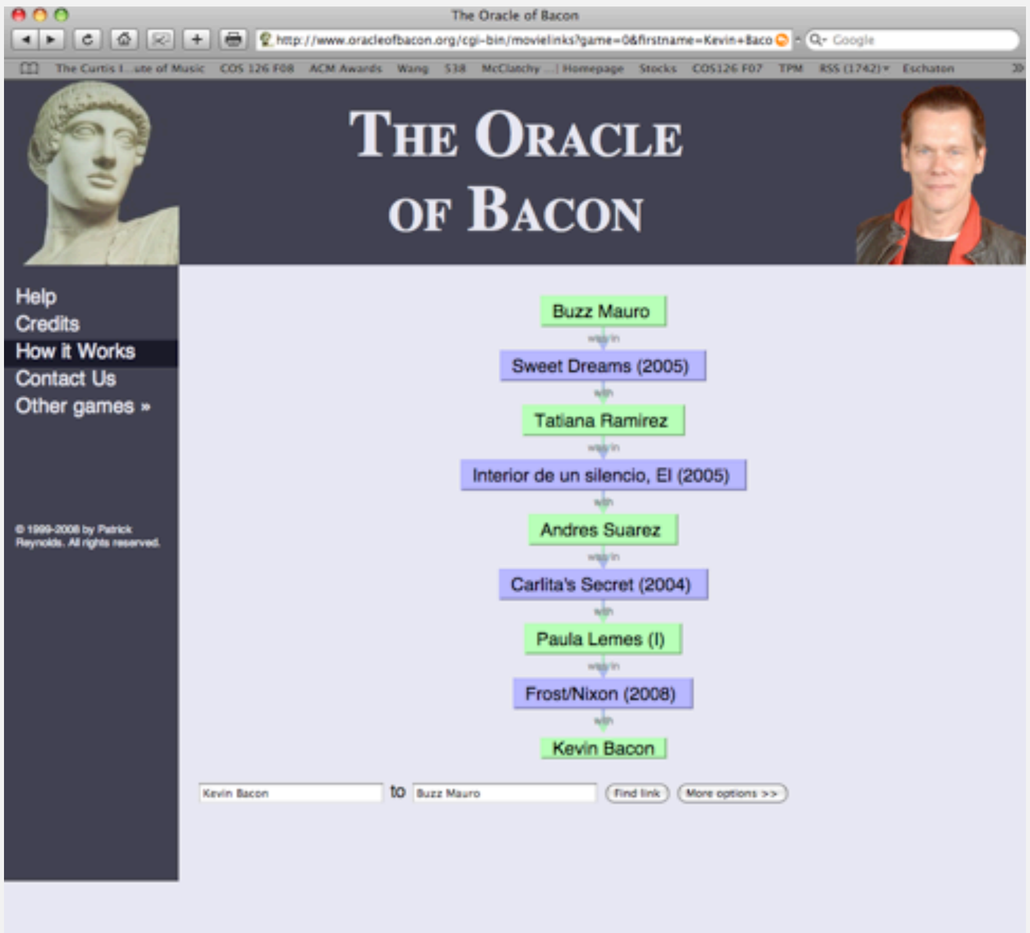
- Facebook.
- Kevin Bacon numbers.
- Fewest number of hops in a communication network.



ARPANET

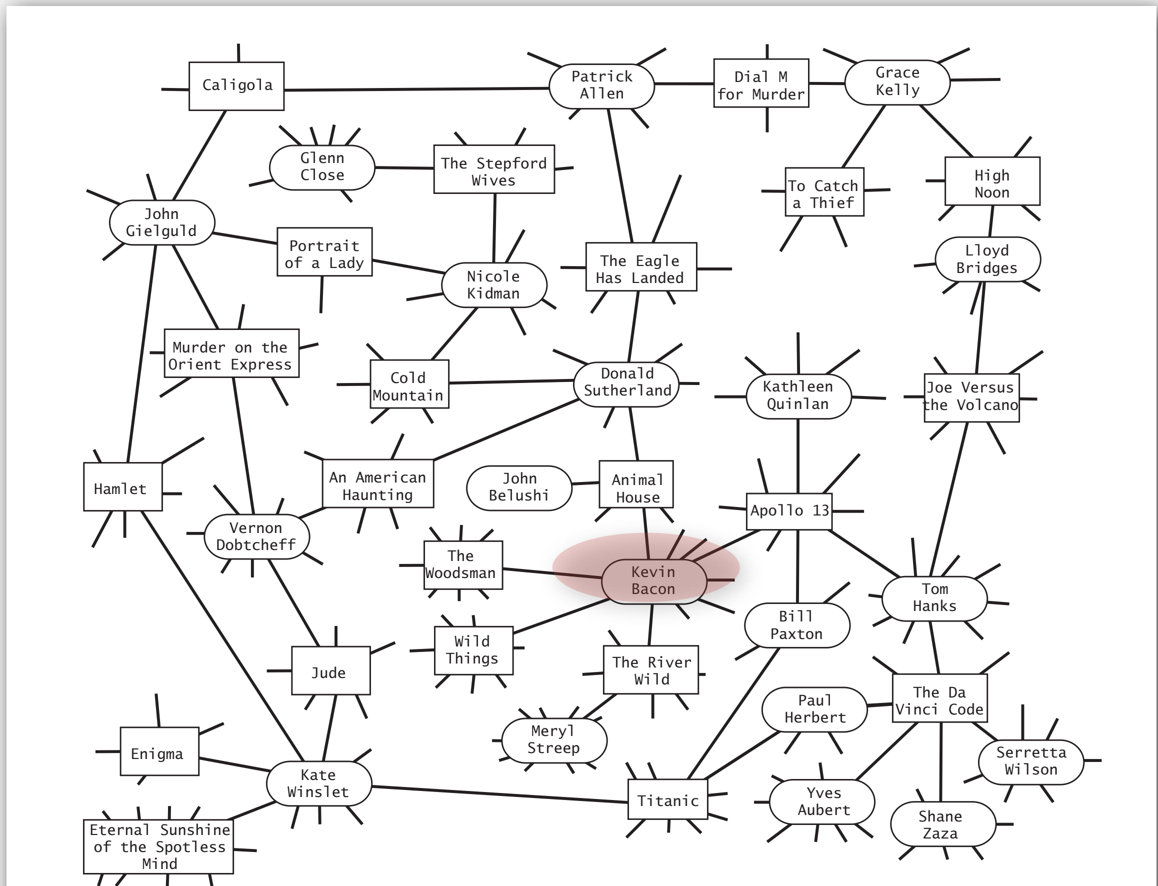
BFS application

- Facebook.
- Kevin Bacon numbers.
- Fewest number of hops in a communication network.



Kevin Bacon graph

- Include vertex for each performer and movie.
- Connect movie to all performers that appear in movie.
- Compute shortest path from s = Kevin Bacon.



A tiny portion of the movie-performer relationship graph

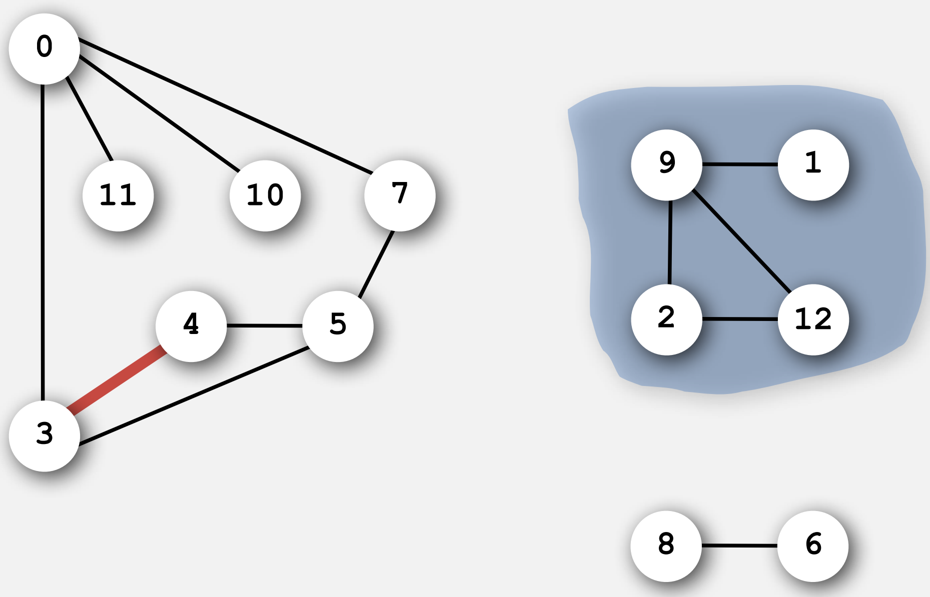
- graph API
- maze exploration
- depth-first search
- breadth-first search
- **connected components**
- challenge

Connectivity queries

Def. Vertices v and w are connected if there is a path between them.

Def. A connected component is a maximal set of connected vertices.

Goal. Preprocess graph to answer queries: is `v` connected to `w`?
in constant time



Vertex	Component
0	0
1	1
2	1
3	0
4	0
5	0
6	2
7	0
8	2
9	1
10	0
11	0
12	1

Union-Find? Not quite.

Connected components

Goal. Partition vertices into connected components.

Connected components

Initialize all vertices v as unmarked.

For each unmarked vertex v, run DFS to identify all vertices discovered as part of the same component.

---

preprocess time	query time	extra space
E + V	1	V

Depth-first search for connected components

```
public class CCFinder
{
   private final static int UNMARKED = -1;
   private int components;
   private int[] cc;

   public CCFinder(Graph G)
   {  /* see next slide */   }

   public int connected(int v, int w)
   {  return cc[v] == cc[w];   }

}
```

component labels

constant-time connectivity query

Depth-first search for connected components

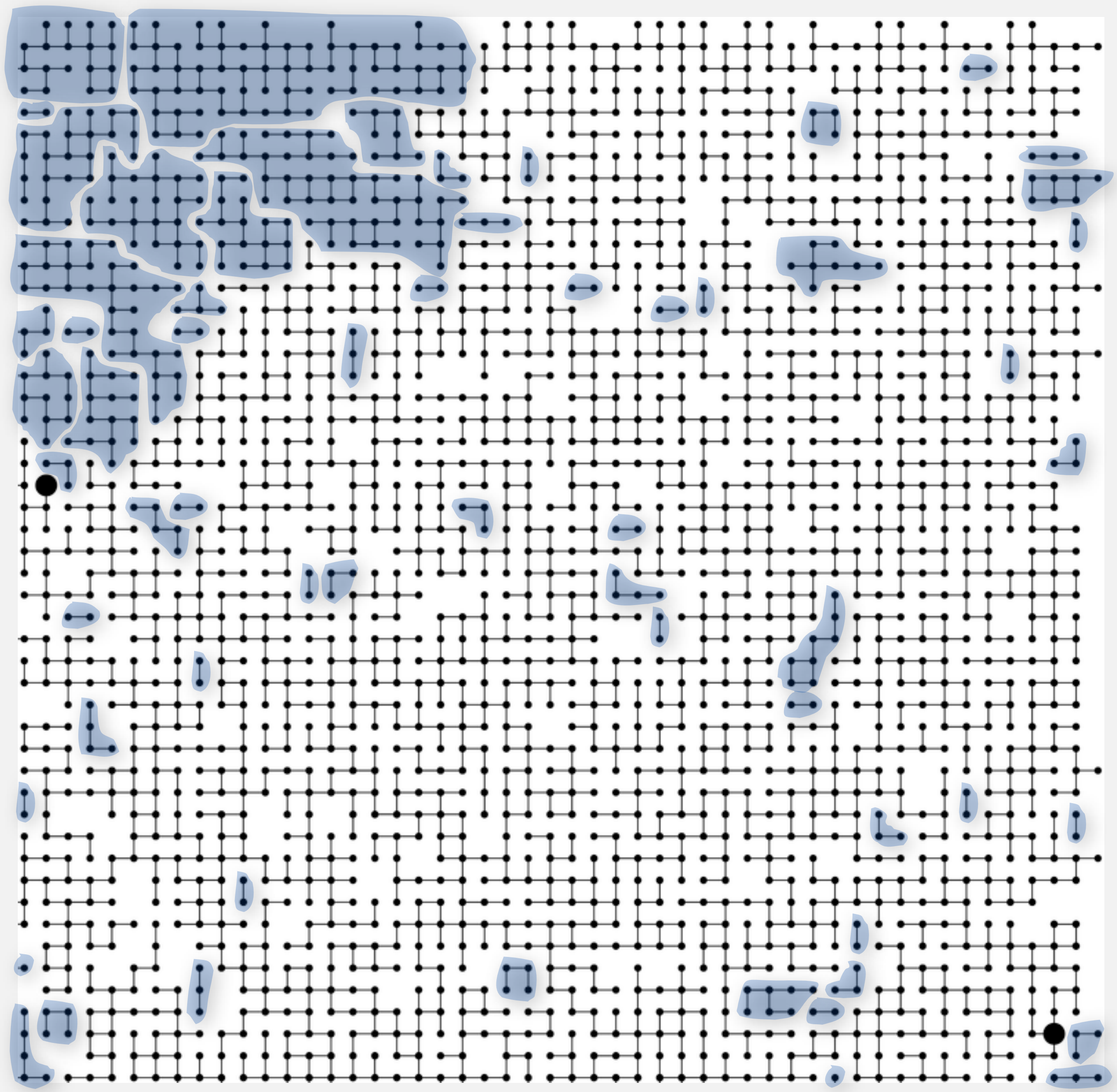
```
public CCFinder(Graph G)
{
   cc = new int[G.V()];
   for (int v = 0; v < G.V(); v++)
      cc[v] = UNMARKED;
   for (int v = 0; v < G.V(); v++)
      if (cc[v] == UNMARKED)
      {
         dfs(G, v);
         components++;
      }
}
```

DFS for each component

```
private void dfs(Graph G, int v)
{
   cc[v] = components;
   for (int w : G.adj(v))
      if (cc[w] == UNMARKED) dfs(G, w);
}
```

standard DFS

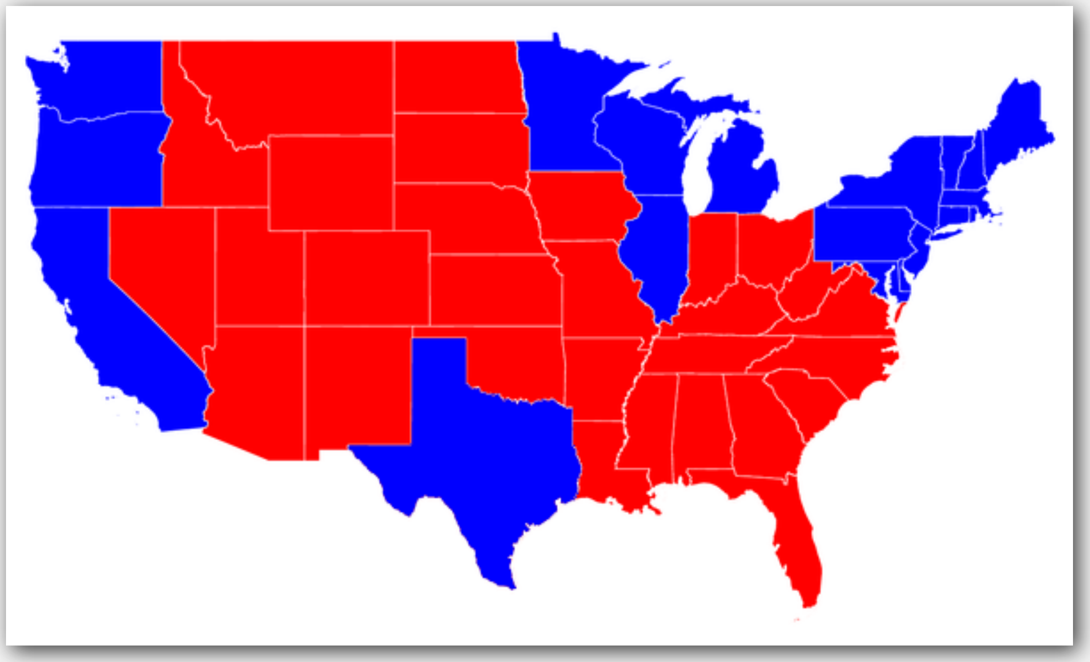
Connected components



63 components

Connected components application: image processing

Goal. Read in a 2D color image and find regions of connected pixels that have the same color.



assuming contiguous states

Input. Scanned image.

Output. Number of red and blue states.

Connected components application: image processing

Goal. Read in a 2D color image and find regions of connected pixels that have the same color.

Efficient algorithm.

- Create grid graph.
- Connect each pixel to neighboring pixel if same color.
- Find connected components in resulting graph.

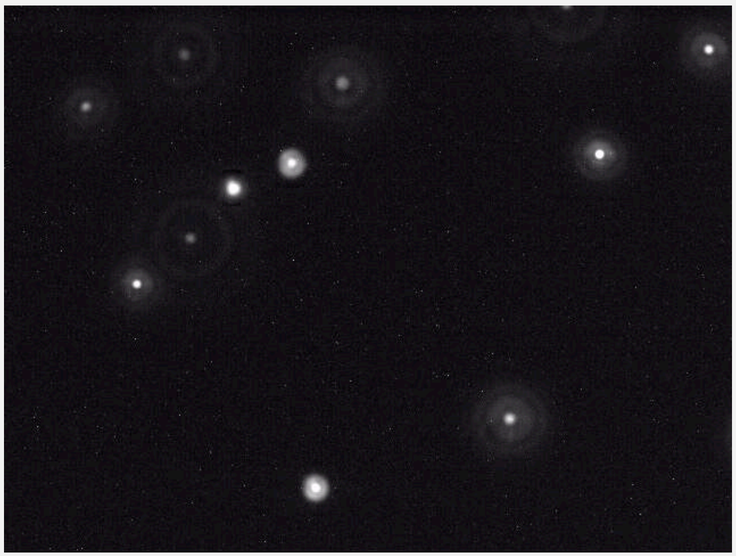
0	1	1	1	1	1	6	6	8	9	9	11
0	0	0	1	6	6	6	8	8	11	9	11
3	0	0	1	6	6	4	8	11	11	11	11
3	0	0	1	1	6	2	11	11	11	11	11
10	10	10	10	1	1	2	11	11	11	11	11
7	7	2	2	2	2	2	11	11	11	11	11
7	7	5	5	5	2	2	11	11	11	11	11

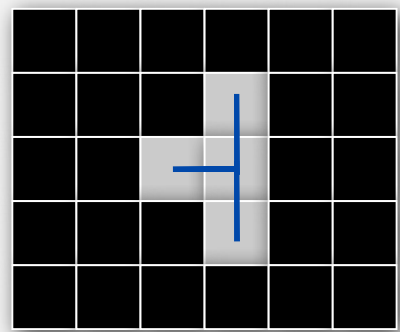
Connected components application: particle detection

Particle detection. Given grayscale image of particles, identify "blobs."

- Vertex: pixel.
- Edge: between two adjacent pixels with grayscale value ≥ 70.
- Blob: connected component of 20-30 pixels.

black = 0
white = 255





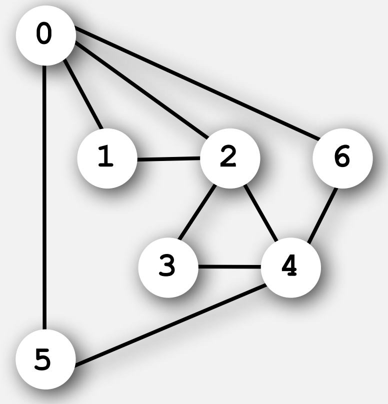
Particle tracking. Track moving particles over time.

- graph API
- maze exploration
- depth-first search
- breadth-first search
- connected components
- **challenges**

Graph-processing challenge 3

Problem. Find a cycle that uses every edge.

Assumption. Need to use each edge exactly once.



0-1-2-3-4-2-0-6-4-5-0

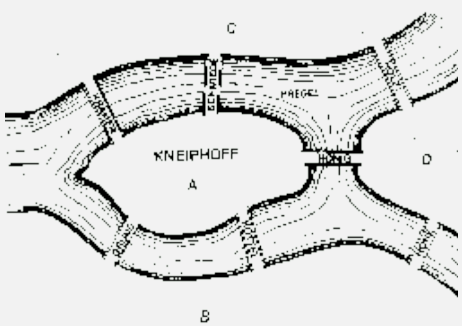
```
0-1
0-2
0-5
0-6
1-2
2-3
2-4
3-4
4-5
4-6
```

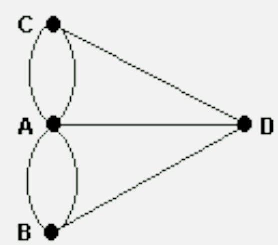
How difficult?

- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

The Seven Bridges of Königsberg. [Leonhard Euler 1736]

> *" ... in Königsberg in Prussia, there is an island A, called the Kneiphof; the river which surrounds it is divided into two branches ... and these branches are crossed by seven bridges. Concerning these bridges, it was asked whether anyone could arrange a route in such a way that he could cross each bridge once and only once."*





Euler tour. Is there a cyclic path that uses each edge exactly once?

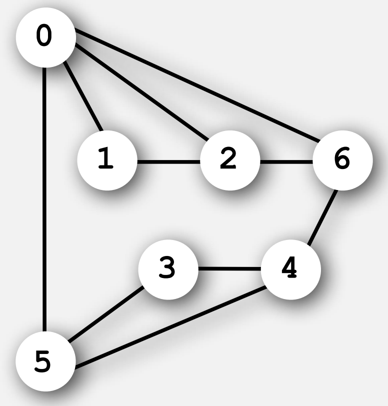
Answer. Yes iff connected and all vertices have even degree.

To find path. DFS-based algorithm (see Algs in Java).

Graph-processing challenge 4

Problem. Find a cycle that visits every vertex.

Assumption. Need to visit each vertex exactly once.



How difficult?

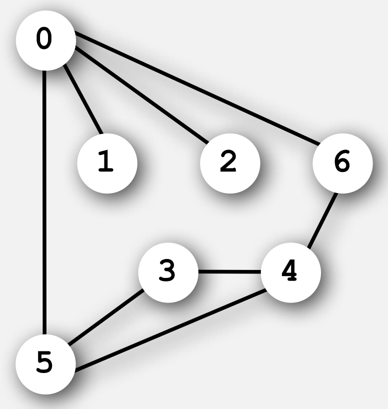
- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Graph-processing challenge 5

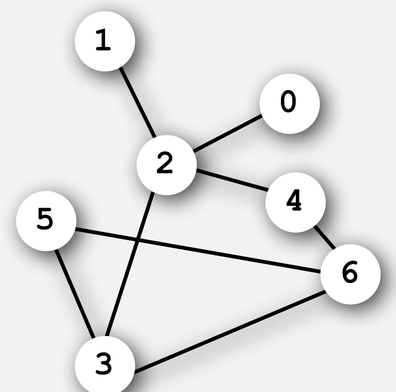
Problem. Are two graphs identical except for vertex names?

How difficult?

- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.



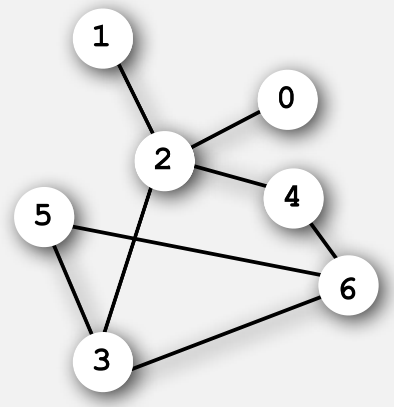
0-1
0-6
0-2
4-3
5-3
5-4
0-5
6-4



2-1
2-4
2-0
6-5
5-3
3-6
2-3
6-4

Graph-processing challenge 6

Problem. Lay out a graph in the plane without crossing edges?

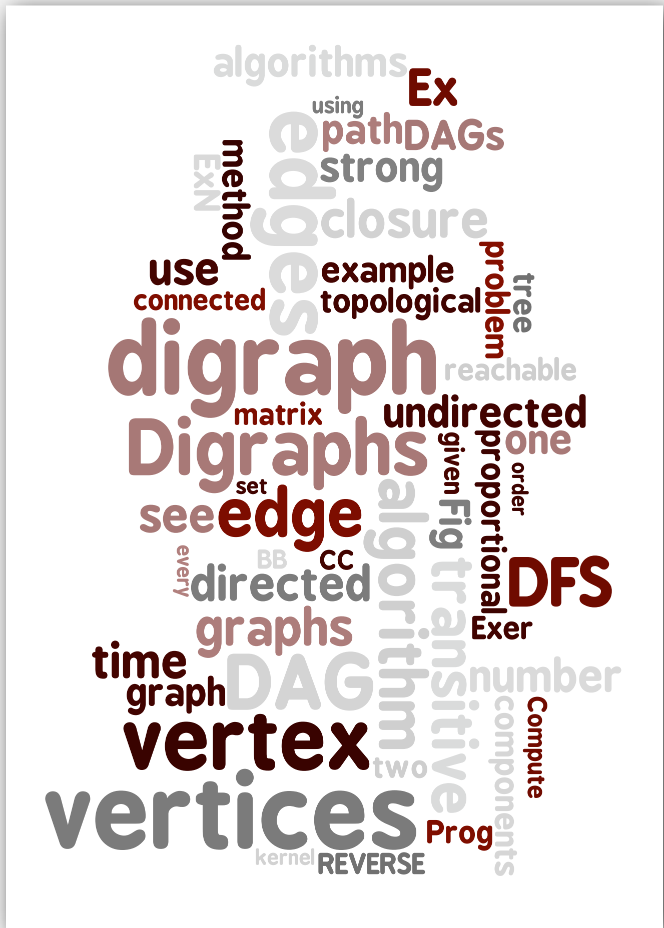


0-2
1-2
2-3
2-4
3-5
3-6
4-6
5-6

How difficult?

- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

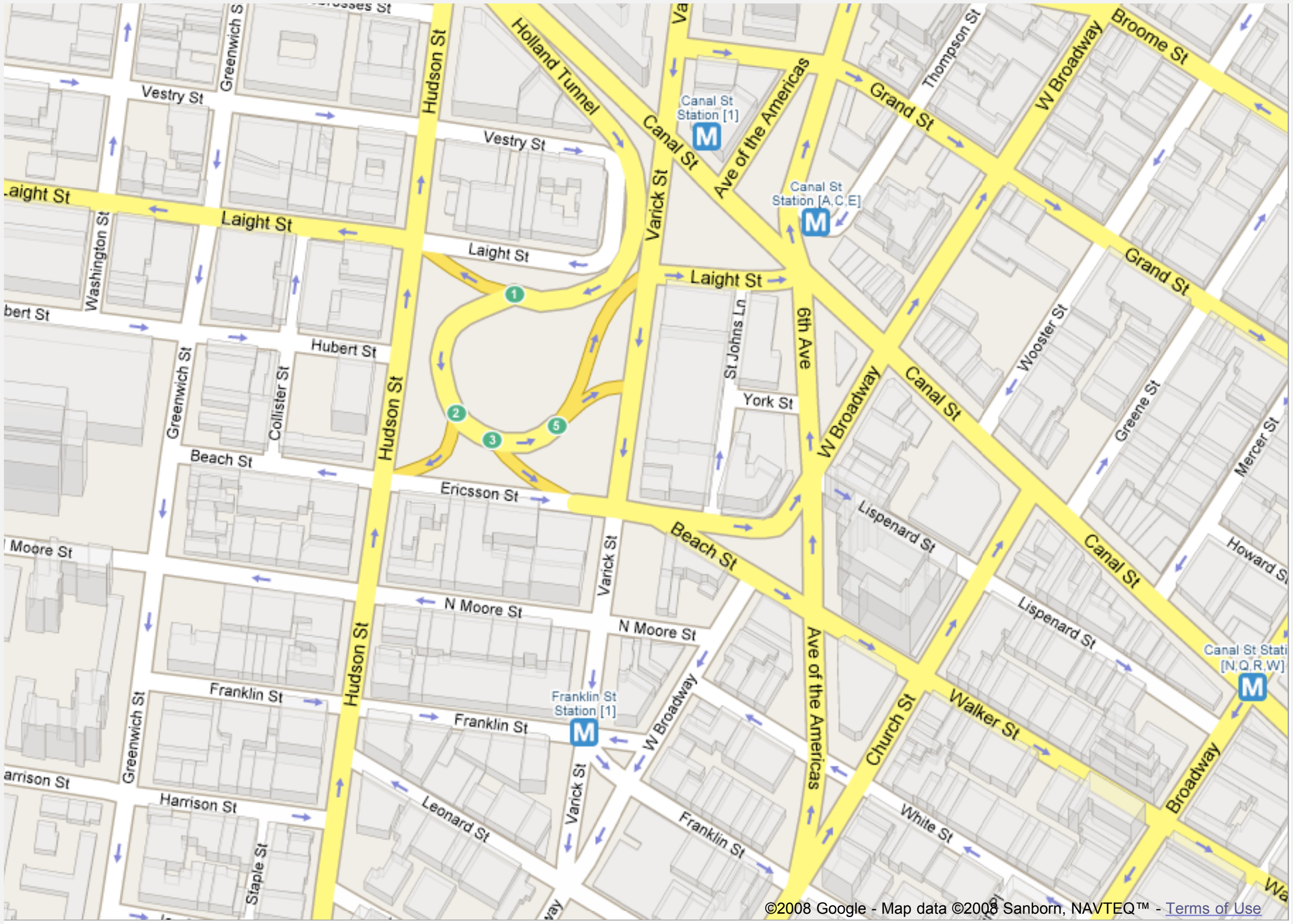
4.2 Directed Graphs



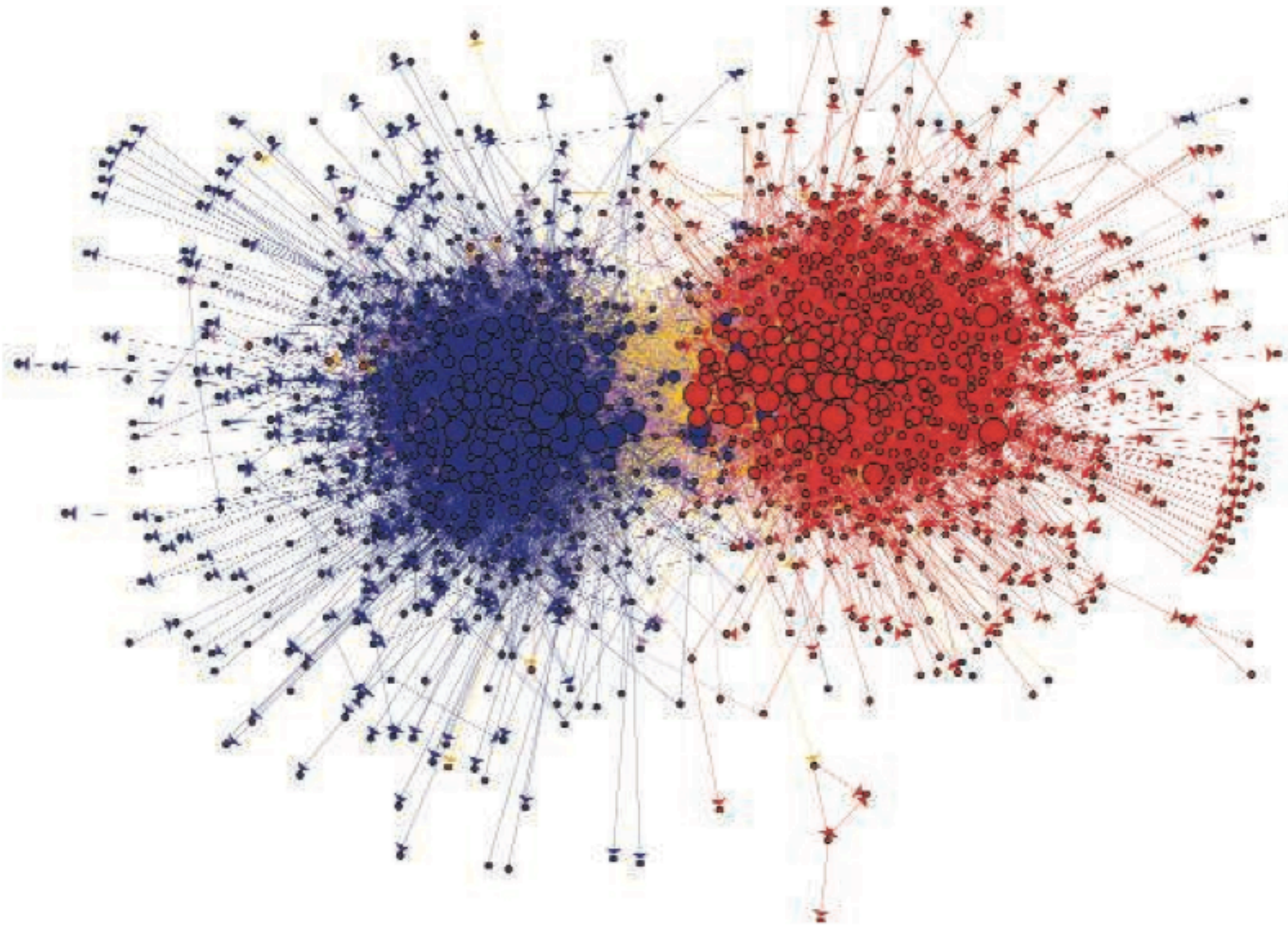
- digraph API
- digraph search
- transitive closure
- topological sort
- strong components

References: Algorithms in Java, 3rd edition, Chapter 19

Digraph. Set of vertices connected pairwise by oriented edges.



Link structure of political blogs

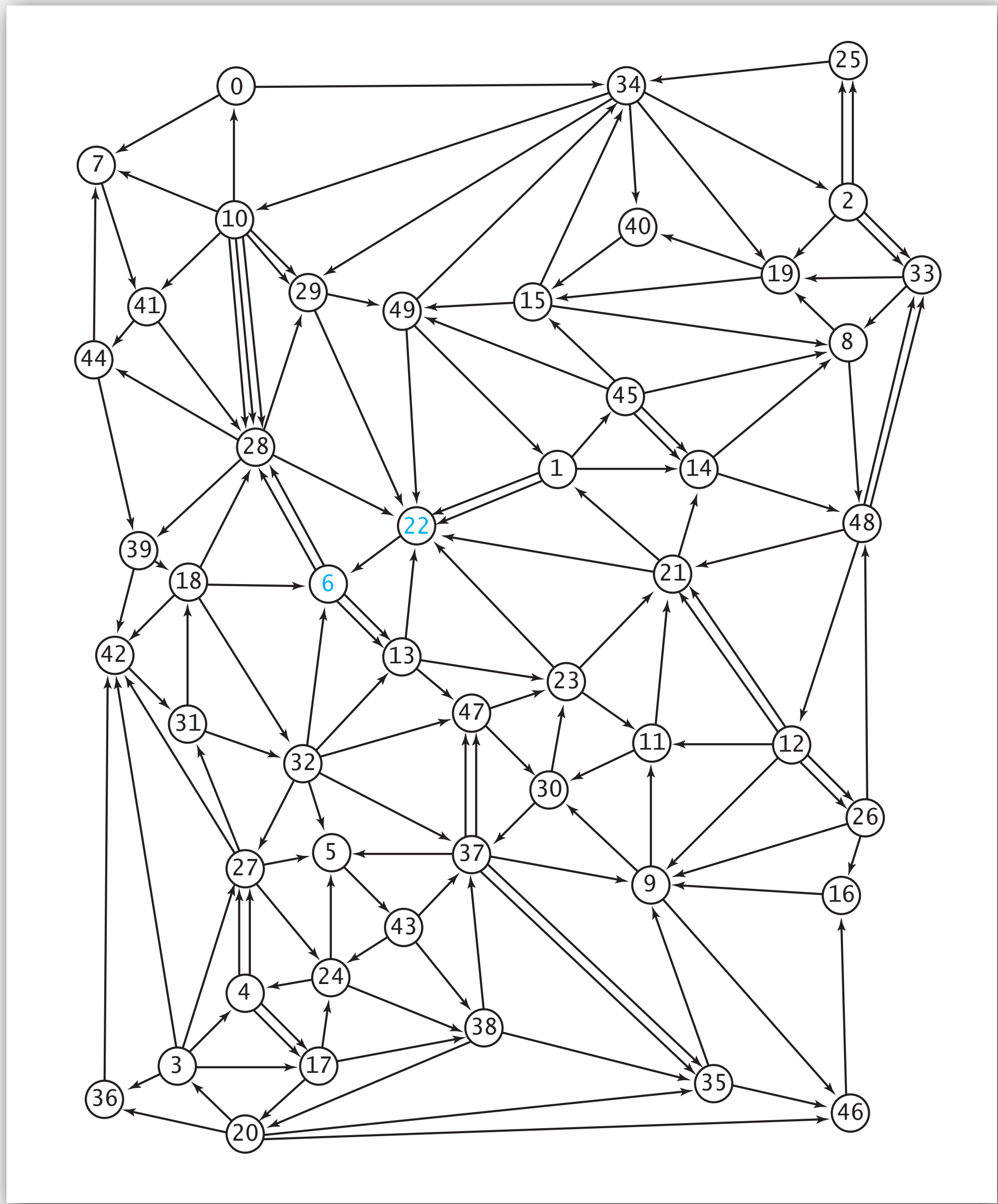


Data from the blogosphere. Shown is a link structure within a community of political blogs (from 2004), where red nodes indicate conservative blogs, and blue liberal. Orange links go from liberal to conservative, and purple ones from conservative to liberal. The size of each blog reflects the number of other blogs that link to it. [Reproduced from (*8*) with permission from the Association for Computing Machinery]

Web graph

Vertex = web page.

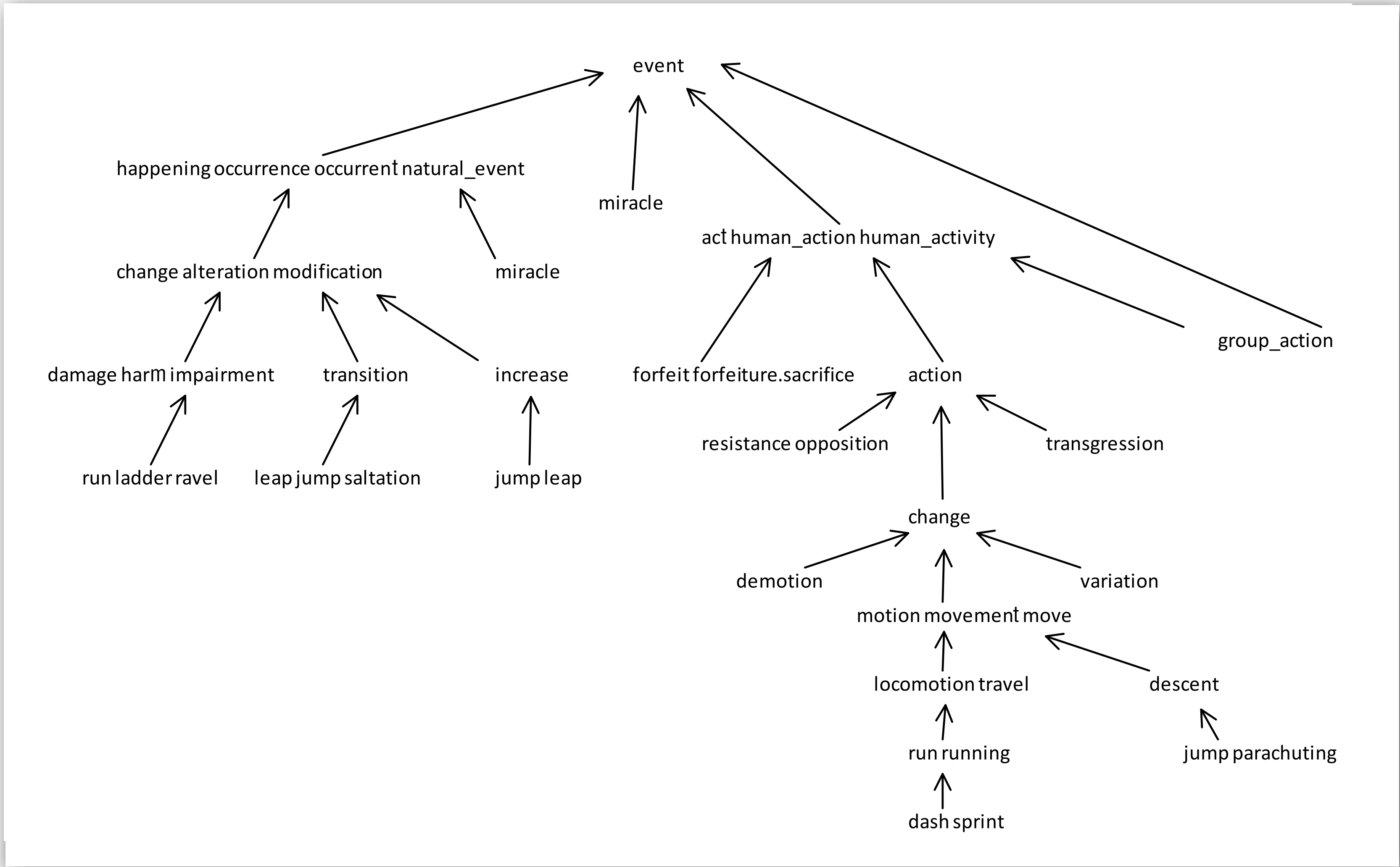
Edge = hyperlink.



WordNet graph

Vertex = synset.

Edge = hypernym relationship.



Digraph applications

graph	vertex	edge
transportation	street intersection	one-way street
web	web page	hyperlink
food web	species	predator-prey relationship
WordNet	synset	hypernym
scheduling	task	precedence constraint
financial	stock, currency	transaction
cell phone	person	placed call
infectious disease	person	infection
game	board position	legal move
citation	journal article	citation
object graph	object	pointer
inheritance hierarchy	class	inherits from
control flow	code block	jump

Some digraph problems

Path. Is there a directed path from s to t?

Shortest path. What is the shortest directed path from s and t?

Strong connectivity. Are all vertices mutually reachable?

Transitive closure. For which vertices v and w is there a path from v to w?

Topological sort. Can you draw the digraph so that all edges point from left to right?

Precedence scheduling. Given a set of tasks with precedence constraints, how can we best complete them all?

PageRank. What is the importance of a web page?

- digraph API
- digraph search
- topological sort
- transitive closure
- strong components

Digraph API

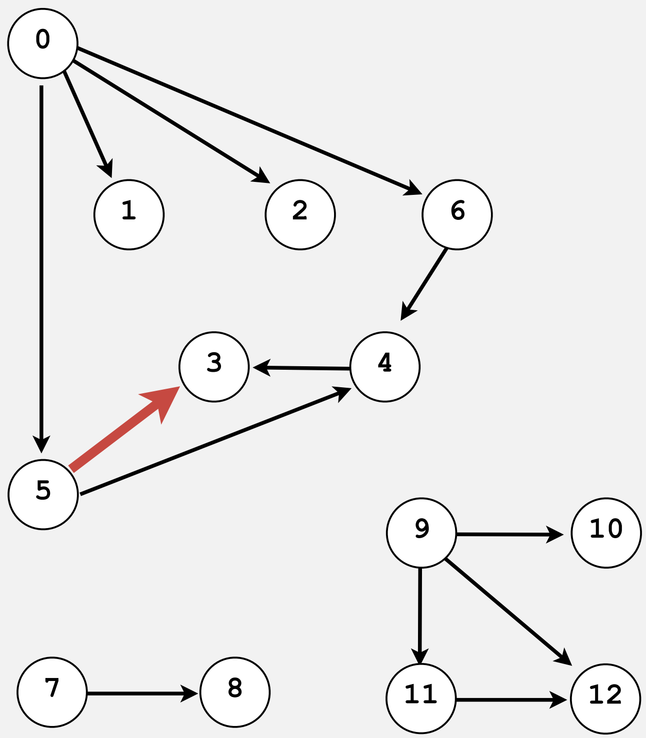
`public class Digraph`		*digraph data type*
	`Digraph(int V)`	*create an empty digraph with V vertices*
	`Digraph(In in)`	*create a digraph from input stream*
`void`	`addEdge(int v, int w)`	*add an edge from v to w*
`Iterable<Integer>`	`adj(int v)`	*return an iterator over the neighbors of v*
`int`	`V()`	*return number of vertices*

```
In in = new In();
Digraph G = new Digraph(in);
```

```
for (int v = 0; v < G.V(); v++)
   for (int w : G.adj(v))
      /*  process edge v→w  */
```

Set of edges representation

Store a list of the edges (linked list or array).

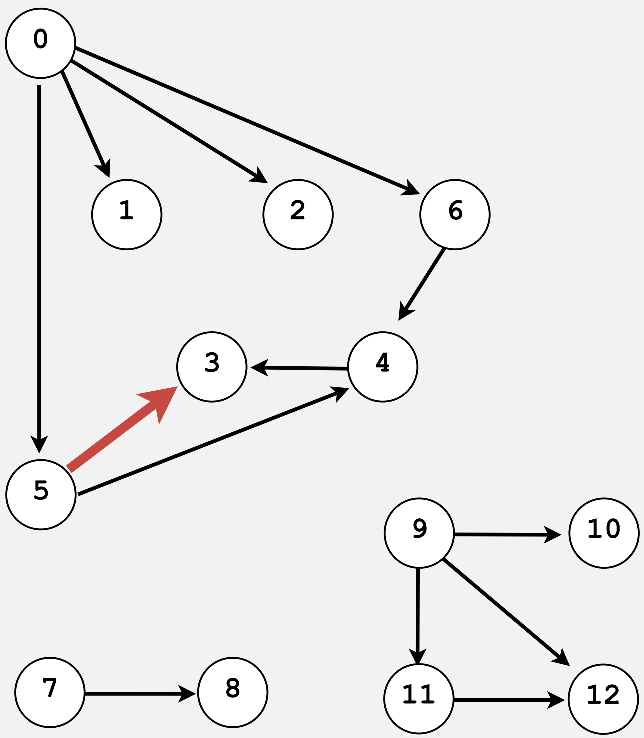


0	1
0	2
0	5
0	6
4	3
5	3
5	4
6	4
7	8
9	10
9	11
9	12
11	12

Adjacency-matrix representation

Maintain a two-dimensional V-by-V boolean array;

for each edge $v \to w$ in the digraph: `adj[v][w] = true`.

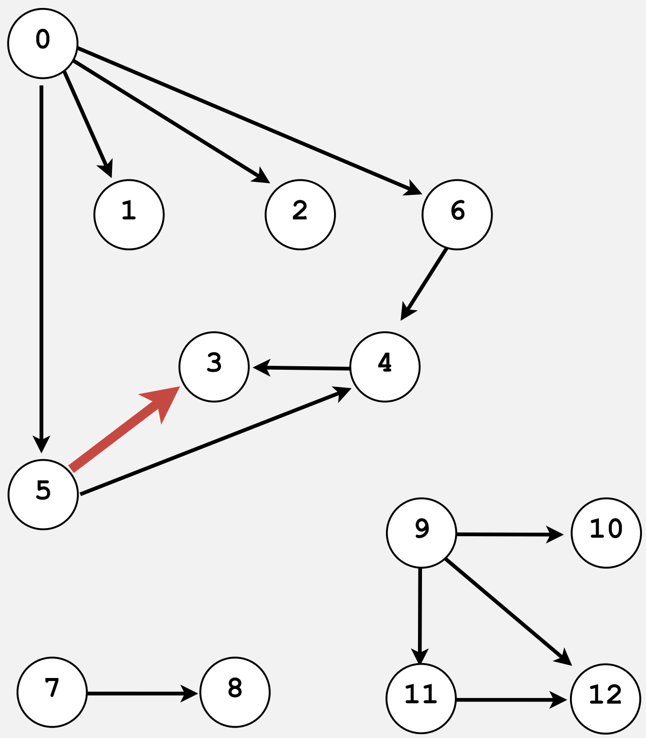


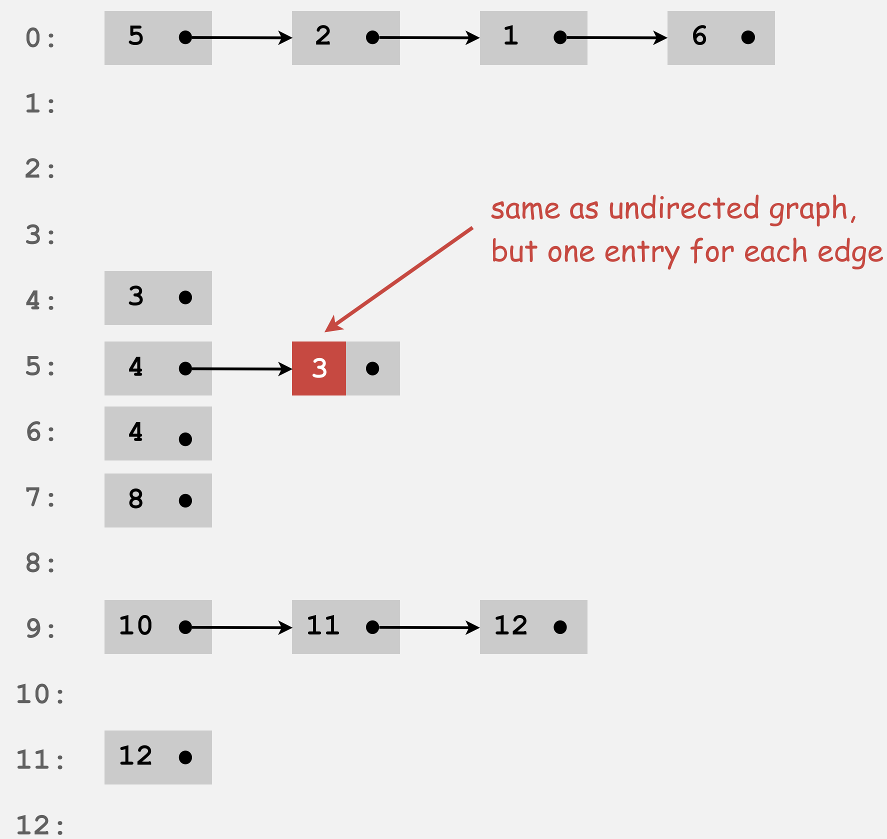
to

from	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	1	0	0	0	0	0	0	0	0	0
5	0	0	0	1	1	0	0	0	0	0	0	0	0
6	0	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	1
12	0	0	0	0	0	0	0	0	0	0	0	0	0

Adjacency-list representation

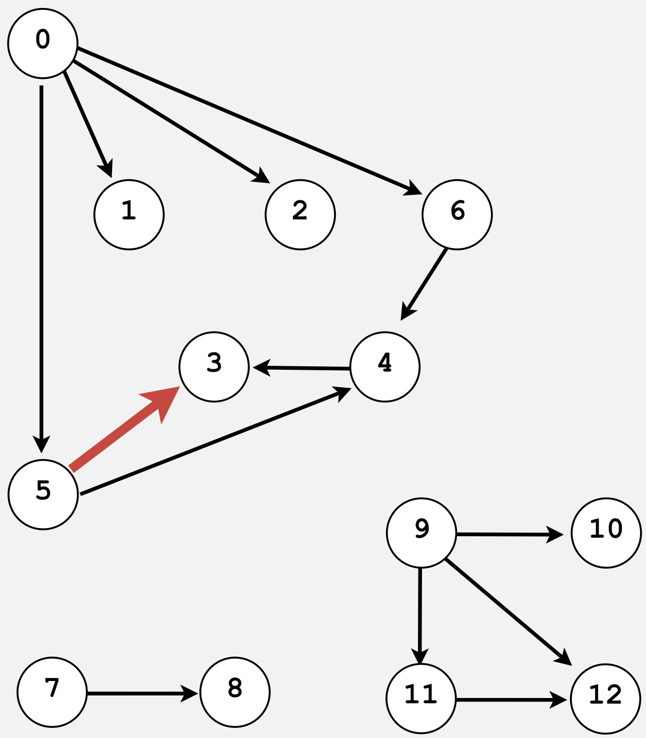
Maintain vertex-indexed array of lists.





Adjacency-set representation

Maintain vertex-indexed array of sets.



0:	{ 1 2 5 6 }
1:	{ }
2:	{ }
3:	{ }
4:	{ 3 }
5:	{ 3, 4 }
6:	{ 4 }
7:	{ 8 }
8:	{ }
9:	{ 10, 11, 12 }
10:	{ }
11:	{ 12 }
12:	{ }

same as undirected graph,
but one entry for each edge

Adjacency-set representation: Java implementation

Same as `Graph`, but only insert one copy of each edge.

```
public class Digraph
{
   private final int V;
   private final SET<Integer>[] adj;

   public Digraph(int V)
   {
      this.V = V;
      adj = (SET<Integer>[]) new SET[V];
      for (int v = 0; v < V; v++)
         adj[v] = new SET<Integer>();
   }

   public void addEdge(int v, int w)
   {  adj[v].add(w);  }

   public Iterable<Integer> adj(int v)
   {  return adj[v];  }
}
```

adjacency sets

create empty graph with V vertices

add edge from v to w (no parallel edges)

iterator for v's neighbors

Digraph representations

In practice. Use adjacency-set (or adjacency-list) representation.

- Algorithms all based on iterating over edges incident to v.
- Real-world digraphs tend to be sparse.

huge number of vertices,
small average vertex degree

representation	space	insert edge from v to w	edge from v to w?	iterate over edges leaving v?
list of edges	E	E	E	E
adjacency matrix	V^2	1	1	V
adjacency list	E + V	outdegree(v)	outdegree(v)	outdegree(v)
adjacency set	E + V	log (outdegree(v))	log (outdegree(v))	outdegree(v)

Typical digraph application: Google's PageRank algorithm

Goal. Determine which pages on web are important.

Solution. Ignore keywords and content, focus on hyperlink structure.

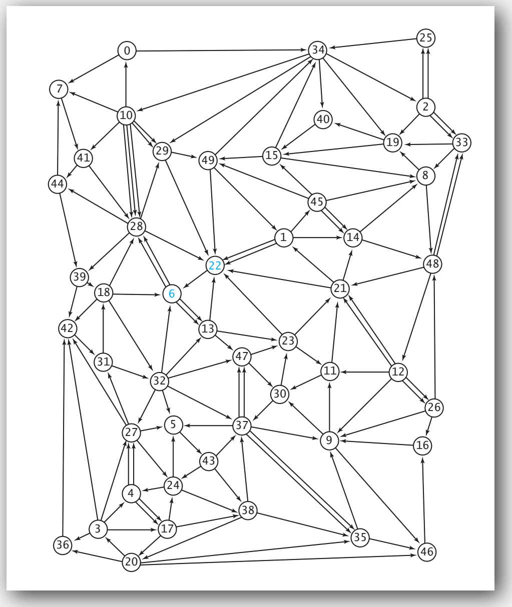
Random surfer model.

- Start at random page.
- With probability 0.85, randomly select a hyperlink to visit next; with probability 0.15, randomly select any page.
- PageRank = proportion of time random surfer spends on each page.

Solution 1. Simulate random surfer for a long time.

Solution 2. Compute ranks directly until they converge.

Solution 3. Compute eigenvalues of adjacency matrix!

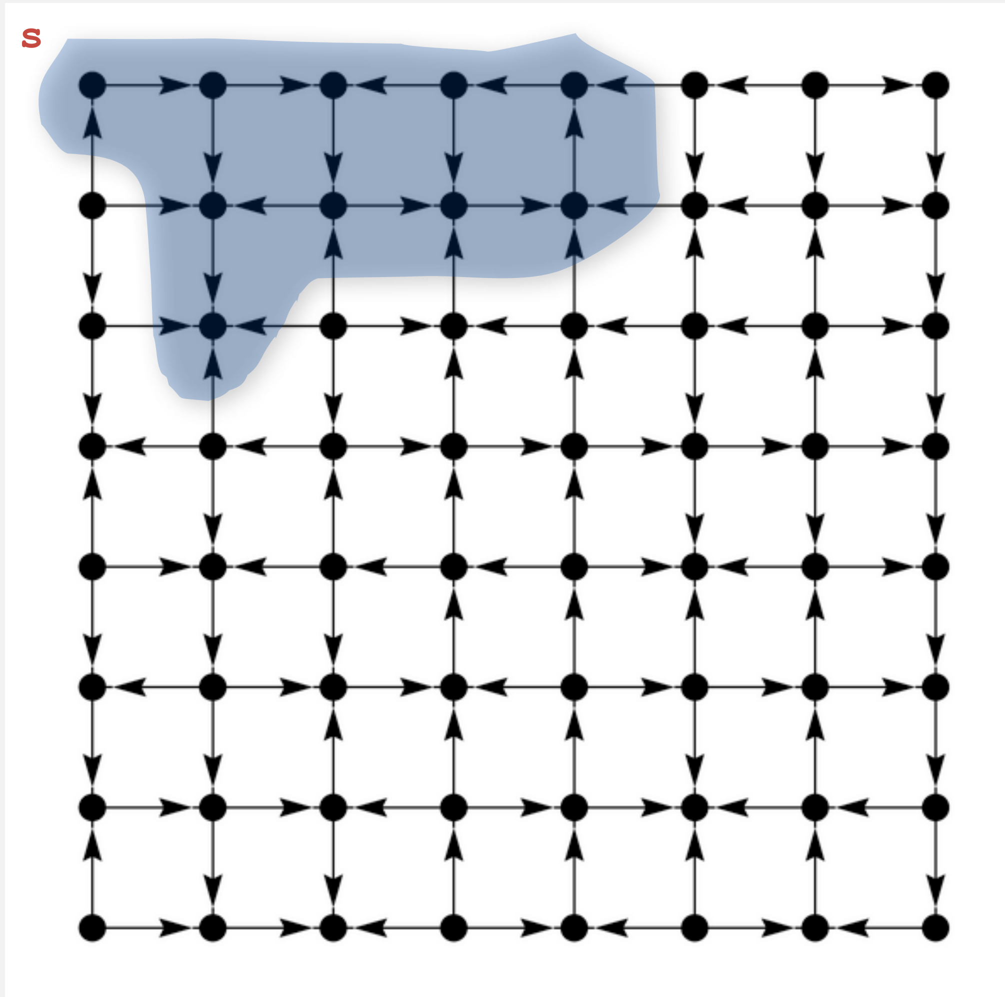


None feasible without sparse digraph representation.

- digraph API
- **digraph search**
- transitive closure
- topological sort
- strong components

Reachability

Problem. Find all vertices reachable from s along a directed path.



Depth-first search in digraphs

Same method as for undirected graphs.

Every undirected graph is a digraph.

- Happens to have edges in both directions.
- DFS is a digraph algorithm.

DFS (to visit a vertex v)

Mark v as visited.
Recursively visit all unmarked vertices w adjacent to v.

Depth-first search (single-source reachability)

Identical to undirected version (substitute `Digraph` for `Graph`).

```
public class DFSearcher
{
   private boolean[] marked;

   public DFSearcher(Digraph G, int s)
   {
      marked = new boolean[G.V()];
      dfs(G, s);
   }

   private void dfs(Digraph G, int v)
   {
      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w]) dfs(G, w);
   }

   public boolean isReachable(int v)
   {  return marked[v];   }
}
```

true if connected to s

constructor marks vertices connected to s

recursive DFS does the work

client can ask whether any vertex is reachable from s

Reachability application: program control-flow analysis

Every program is a digraph.

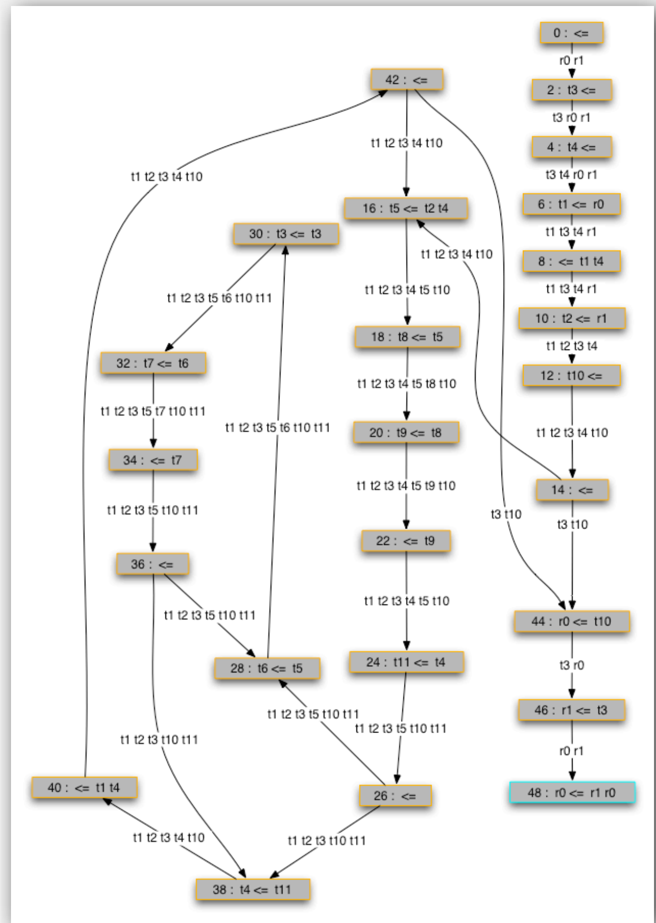
- Vertex = basic block of instructions (straight-line program).
- Edge = jump.

Dead code elimination.

Find (and remove) unreachable code.

Infinite loop detection.

Determine whether exit is unreachable.



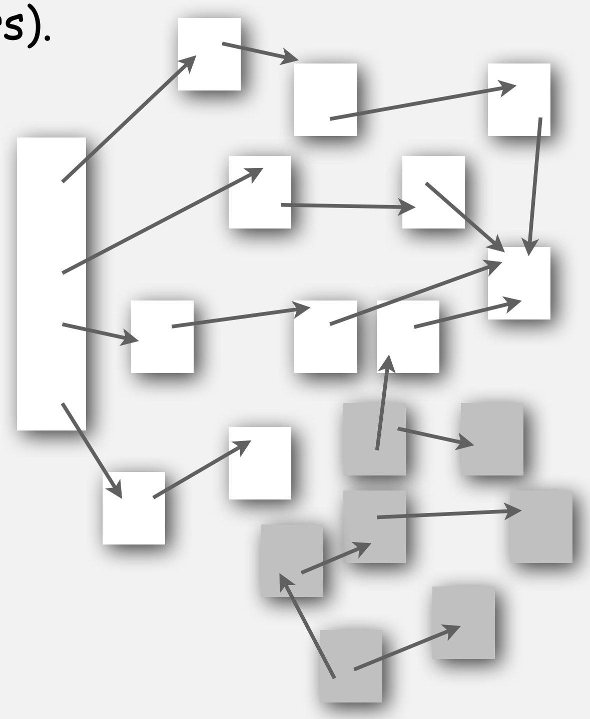
Reachability application: mark-sweep garbage collector

Every data structure is a digraph.

- Vertex = object.
- Edge = reference.

Roots. Objects known to be directly accessible by program (e.g., stack).

Reachable objects. Objects indirectly accessible by program (starting at a root and following a chain of pointers).

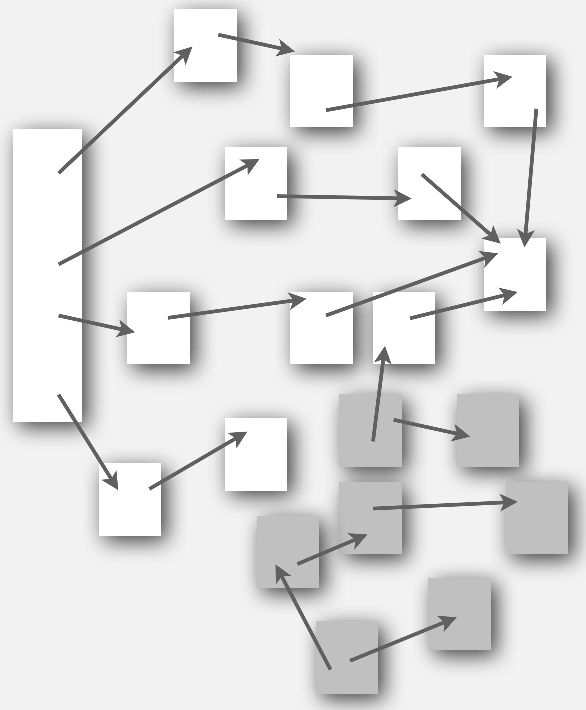


Reachability application: mark-sweep garbage collector

Mark-sweep algorithm. [McCarthy, 1960]

- Mark: mark all reachable objects.
- Sweep: if object is unmarked, it is garbage, so add to free list.

Memory cost. Uses 1 extra mark bit per object, plus DFS stack.



Depth-first search (DFS)

DFS enables direct solution of simple digraph problems.

✓ • Reachability.
- Cycle detection.
- Topological sort.
- Transitive closure.

Basis for solving difficult digraph problems.
- Directed Euler path.
- Strong connected components.

Breadth-first search in digraphs

Every undirected graph is a digraph.

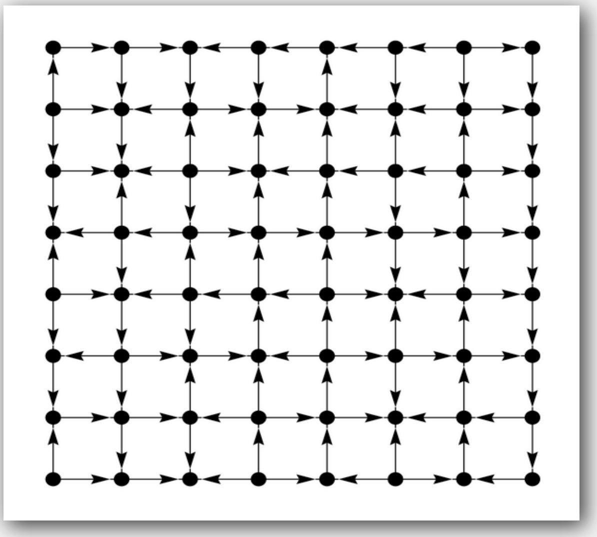
- Happens to have edges in both directions.
- BFS is a digraph algorithm.

BFS (from source vertex s)

Put s onto a FIFO queue.

Repeat until the queue is empty:

- *remove the least recently added vertex v*
- *add each of v's unvisited neighbors to the queue and mark them as visited.*



Property. Visits vertices in increasing distance from s.

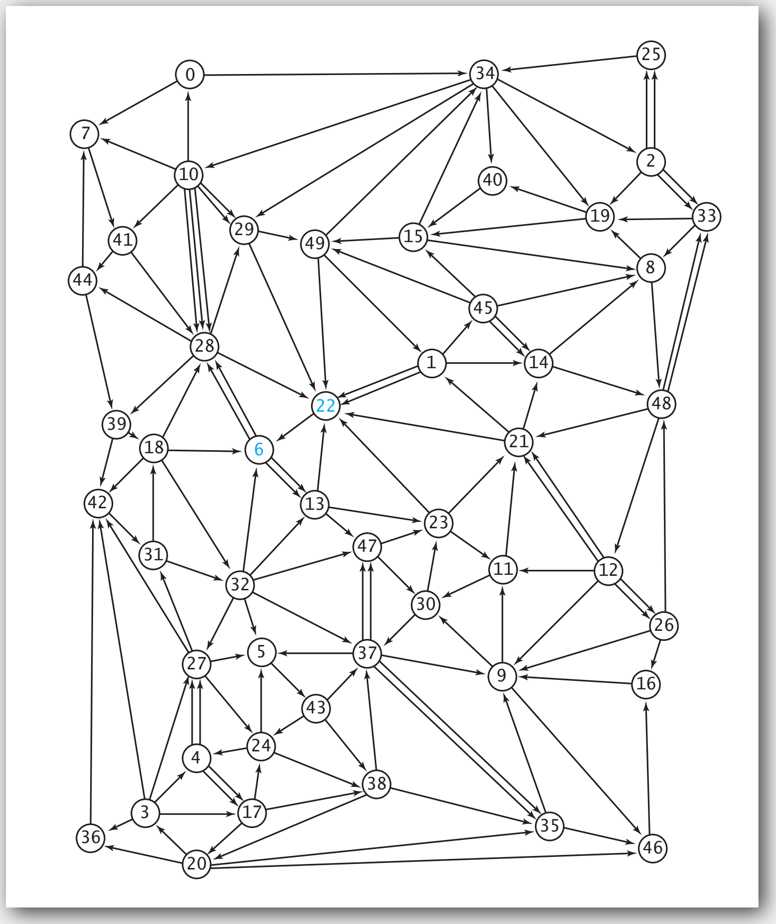
Digraph BFS application: web crawler

Goal. Crawl web, starting from some root web page, say **`www.princeton.edu`**.

Solution. BFS with implicit graph.

BFS.

- Start at some root web page.
- Maintain a **Queue** of websites to explore.
- Maintain a **SET** of discovered websites.
- Dequeue the next website and enqueue websites to which it links (provided you haven't done so before).



Q. Why not use DFS?

Web crawler: BFS-based Java implementation

```
Queue<String> q = new Queue<String>();
SET<String> visited = new SET<String>();

String s = "http://www.princeton.edu";
q.enqueue(s);
visited.add(s);

while (!q.isEmpty())
{
   String v = q.dequeue();
   StdOut.println(v);
   In in = new In(v);
   String input = in.readAll();

   String regexp = "http://(\\w+\\.)*(\\w+)";
   Pattern pattern = Pattern.compile(regexp);
   Matcher matcher = pattern.matcher(input);
   while (matcher.find())
   {
      String w = matcher.group();
      if (!visited.contains(w))
      {
         visited.add(w);
         q.enqueue(w);
      }
   }
}
```

queue of websites to crawl

set of visited websites

start crawling from website s

read in raw html for next website in queue

use regular expression to find all URLs in website of form `http://xxx.yyy.zzz`

if unvisited, mark as visited and put on queue

- digraph API
- digraph search
- **transitive closure**
- topological sort
- strong components

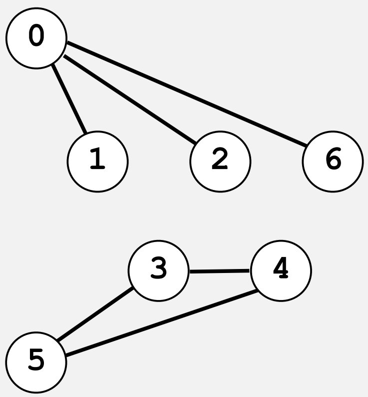
Graph-processing challenge (revisited)

Problem. Is there an undirected path between `v` and `w` ?

Goals. Linear preprocessing time, constant query time.

How difficult?

- Any COS 126 student could do it.
- ✓ Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.




```
0-1
0-6
0-2
3-4
3-5
4-5
```

Digraph-processing challenge 1

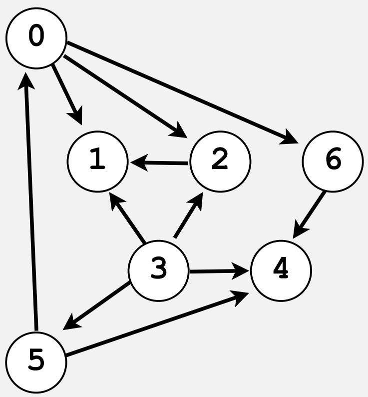
Problem. Is there a directed path from `v` to `w` ?

Goals. Linear preprocessing time, constant query time.

How difficult?

- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- ✓ Impossible.

can't do better than V^2
(reduction from boolean matrix multiplication)

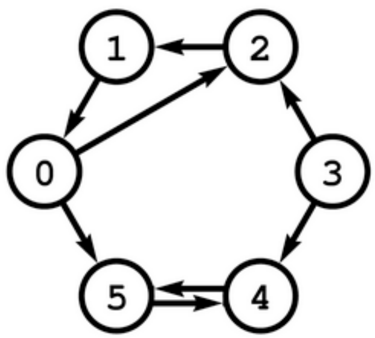


0→1
0→6
0→2
3→4
3→2
5→4
5→0
3→5
2→1
6→4
3→1

Transitive closure

Def. The transitive closure of a digraph G is another digraph with a directed edge from `v` to `w` if there is a directed path from `v` to `w` in G.

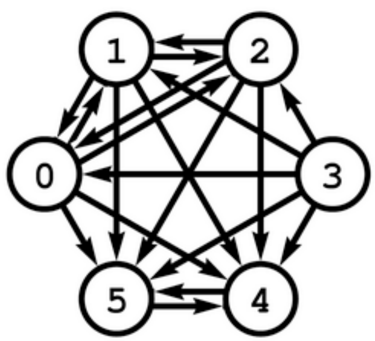
digraph G



	0	1	2	3	4	5
0	1	0	1	0	0	1
1	1	1	0	0	0	0
2	0	1	1	0	0	0
3	0	0	1	1	1	0
4	0	0	0	0	1	1
5	0	0	0	0	1	1

digraph G is usually sparse

transitive closure TC(G)



	0	1	2	3	4	5
0	1	1	1	0	1	1
1	1	1	1	0	1	1
2	1	1	1	0	1	1
3	1	1	1	1	1	1
4	0	0	0	0	1	1
5	0	0	0	0	1	1

TC(G) is usually dense

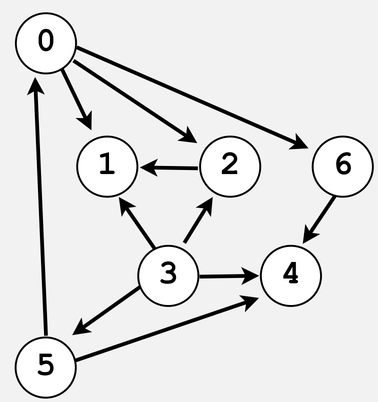
Digraph-processing challenge 1 (revised)

Problem. Is there a directed path from `v` to `w` ?

Goals. ~ V^2 preprocessing time, constant query time.

How difficult?

- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- ✓ No one knows. ← open research problem
- Impossible.



0→1
0→6
0→2
3→4
3→2
5→4
5→0
3→5
2→1
6→4
3→1

Digraph-processing challenge 1 (revised again)

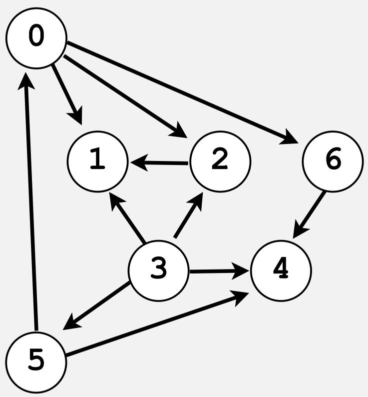
Problem. Is there a directed path from `v` to `w` ?

Goals. ~ V E preprocessing time, ~ V^2 space, constant query time.

How difficult?

- Any COS 126 student could do it.
- ✓ Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Use DFS once for each vertex
to compute rows of transitive closure



0→1
0→6
0→2
3→4
3→2
5→4
5→0
3→5
2→1
6→4
3→1

Transitive closure: Java implementation

Use an array of `DFSearcher` objects, one for each row of transitive closure.

```
public class TransitiveClosure
{
   private DFSearcher[] tc;

   public TransitiveClosure(Digraph G)
   {
      tc = new DFSearcher[G.V()];
      for (int v = 0; v < G.V(); v++)
         tc[v] = new DFSearcher(G, v);
   }

   public boolean reachable(int v, int w)
   {  return tc[v].isReachable(w);   }
}
```

array of `DFSearcher` objects

initialize array

is there a directed path from v to w ?

- digraph API
- digraph search
- transitive closure
- **topological sort**
- strong components

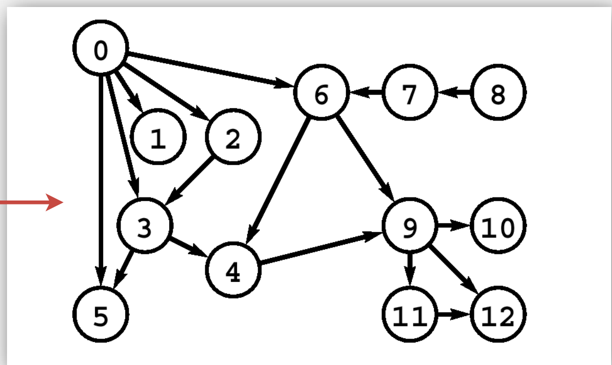
Digraph application: scheduling

Scheduling. Given a set of tasks to be completed with precedence constraints, in what order should we schedule the tasks?

Graph model.

- Create a vertex `v` for each task.
- Create an edge `v→w` if task `v` must precede task `w`.

precedence constraint graph



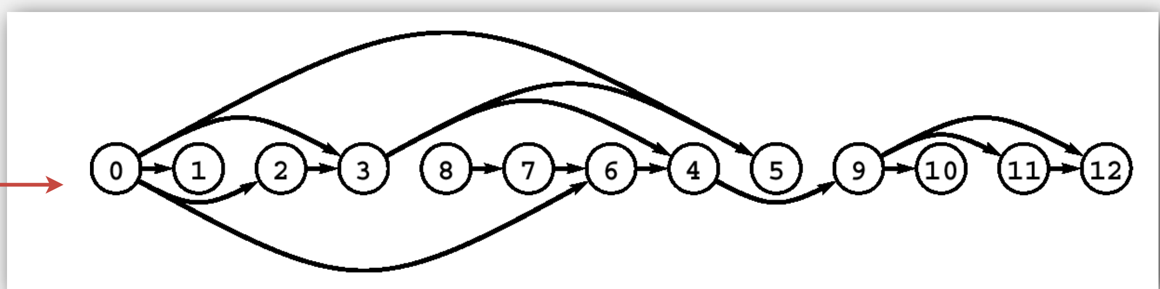
tasks

0. read programming assignment
1. download files
2. write code
3. attend precept

…

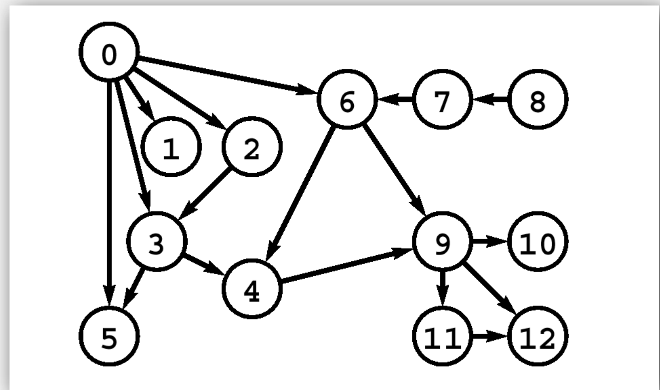
12. sleep

feasible schedule

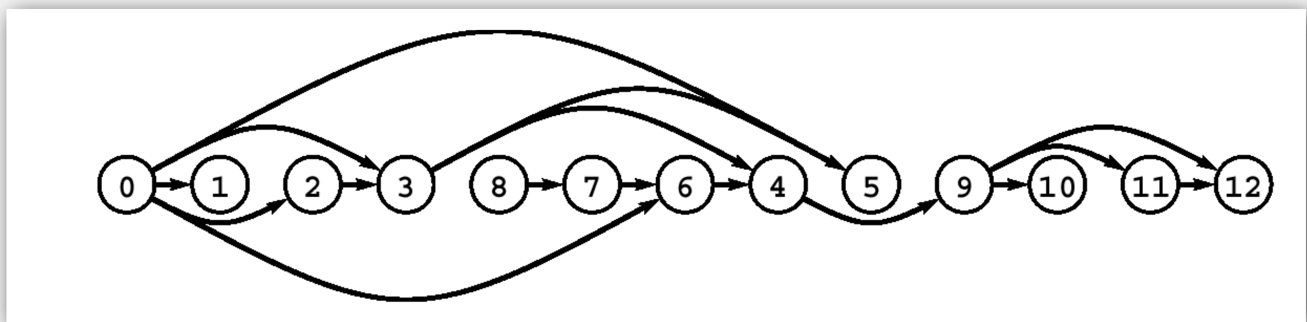


Topological sort

DAG. Directed acyclic graph.



Topological sort. Redraw DAG so all edges point left to right.



Fact. Digraph is a DAG iff no directed cycle.

Digraph-processing challenge 2a

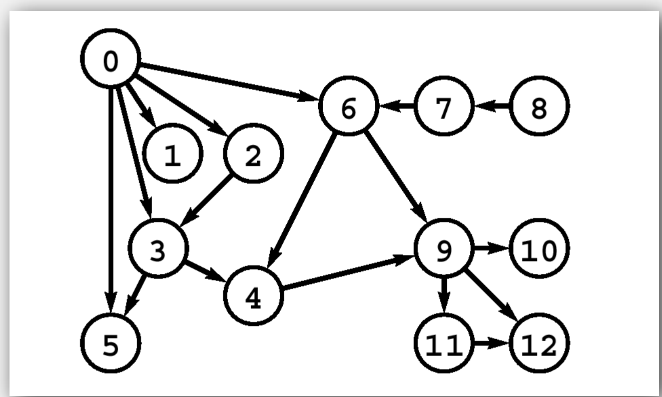
Problem. Check that a digraph is a DAG; if so, find a topological order.

Goal. Linear time.

How difficult?

- Any COS 126 student could do it.
- ✓ Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

use DFS



0 1 2 3 8 7 6 4 5 9 10 11 12

0→1
0→6
0→2
0→5
2→3
4→9
6→4
6→9
7→6
8→7
9→10
9→11
9→12
11→12

Topological sort in a DAG: Java implementation

```
public class TopologicalSorter
{
   private boolean[] marked;
   private Stack<Integer> sorted;

   public TopologicalSorter(Digraph G)
   {
      marked = new boolean[G.V()];
      sorted = new Stack<Integer>();
      for (int v = 0; v < G.V(); v++)
         if (!marked[v]) tsort(G, v);
   }

   private void tsort(Digraph G, int v)
   {
      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w]) tsort(G, w);
      sorted.push(v);
   }

   public Iterable<Integer> order()
   {  return sorted;  }
}
```

vertices in topological order

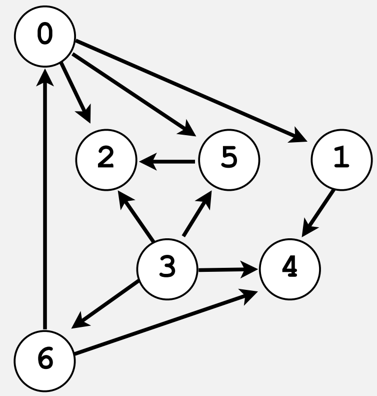
reverse DFS postorder

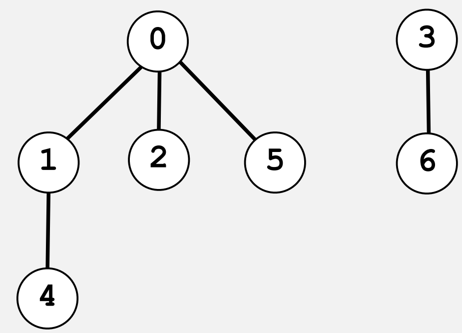
Topological sort in a DAG: trace

Visit means *call* `tsort()` and *leave* means *return from* `tsort()`.

	marked[]	sorted
visit 0:	1 0 0 0 0 0 0	-
visit 1:	1 1 0 0 0 0 0	-
visit 4:	1 1 0 0 1 0 0	-
leave 4:	1 1 0 0 1 0 0	4
leave 1:	1 1 0 0 1 0 0	4 1
visit 2:	1 1 1 0 1 0 0	4 1
leave 2:	1 1 1 0 1 0 0	4 1 2
visit 5:	1 1 1 0 1 1 0	4 1 2
check 2:	1 1 1 0 1 1 0	4 1 2
leave 5:	1 1 1 0 1 1 0	4 1 2 5
leave 0:	1 1 1 0 1 1 0	4 1 2 5 0
check 1:	1 1 1 0 1 1 0	4 1 2 5 0
check 2:	1 1 1 0 1 1 0	4 1 2 5 0
visit 3:	1 1 1 1 1 1 0	4 1 2 5 0
check 2:	1 1 1 1 1 1 0	4 1 2 5 0
check 4:	1 1 1 1 1 1 0	4 1 2 5 0
check 5:	1 1 1 1 1 1 0	4 1 2 5 0
visit 6:	1 1 1 1 1 1 1	4 1 2 5 0
leave 6:	1 1 1 1 1 1 1	4 1 2 5 0 6
leave 3:	1 1 1 1 1 1 1	4 1 2 5 0 6 3
check 4:	1 1 1 1 1 1 0	4 1 2 5 0 6 3
check 5:	1 1 1 1 1 1 0	4 1 2 5 0 6 3
check 6:	1 1 1 1 1 1 0	4 1 2 5 0 6 3

0→1
0→6
0→2
3→4
3→2
5→4
5→0
3→5
2→1
6→4
3→1





3 6 0 5 2 1 4

Topological sort in a DAG: correctness proof

Proposition. If digraph is a DAG, algorithm yields a topological order.

Pf.

- Algorithm terminates in O(E + V) time since it's just a version of DFS.
- Consider any edge `v→w`. When `tsort(G, v)` is called,
  - Case 1: `tsort(G, w)` has already been called and returned. Thus, `w` will appear after `v` in topological order.
  - Case 2: `tsort(G, w)` has not yet been called, so it will get called directly or indirectly by `tsort(G, v)` and it will finish before `tsort(G, v)`. Thus, `w` will appear after `v` in topological order.
  - Case 3: `tsort(G, w)` has already been called, but not returned. Then the function call stack contains a directed path from `w` to `v`. Combining this path with the edge `v→w` yields a directed cycle, contradicting DAG.

Digraph-processing challenge 2b

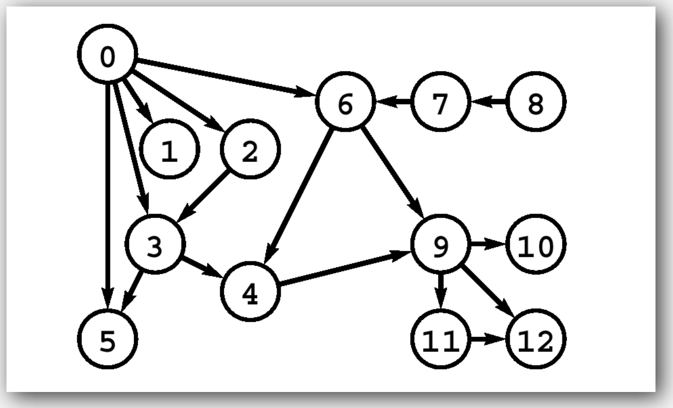
Problem. Given a digraph, is there a directed cycle?

Goal. Linear time.

run DFS-based topological sort algorithm;
if it yields a topological sort, no directed cycle
(can modify code to find cycle)

How difficult?

- Any COS 126 student could do it.
- ✓ Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.



0 1 2 3 8 7 6 4 5 9 10 11 12

0→1
0→6
0→2
0→5
2→3
4→9
6→4
6→9
7→6
8→7
9→10
9→11
9→12
11→12

Topological sort and cycle detection applications

- Causalities.
- Email loops.
- Compilation units.
- Class inheritance.
- Course prerequisites.
- Deadlocking detection.
- Precedence scheduling.
- Temporal dependencies.
- Pipeline of computing jobs.
- Check for symbolic link loop.
- Evaluate formula in spreadsheet.

Cycle detection application: cyclic inheritance

The Java compiler does cycle detection.

```
public class A extends B
{
   ...
}
```

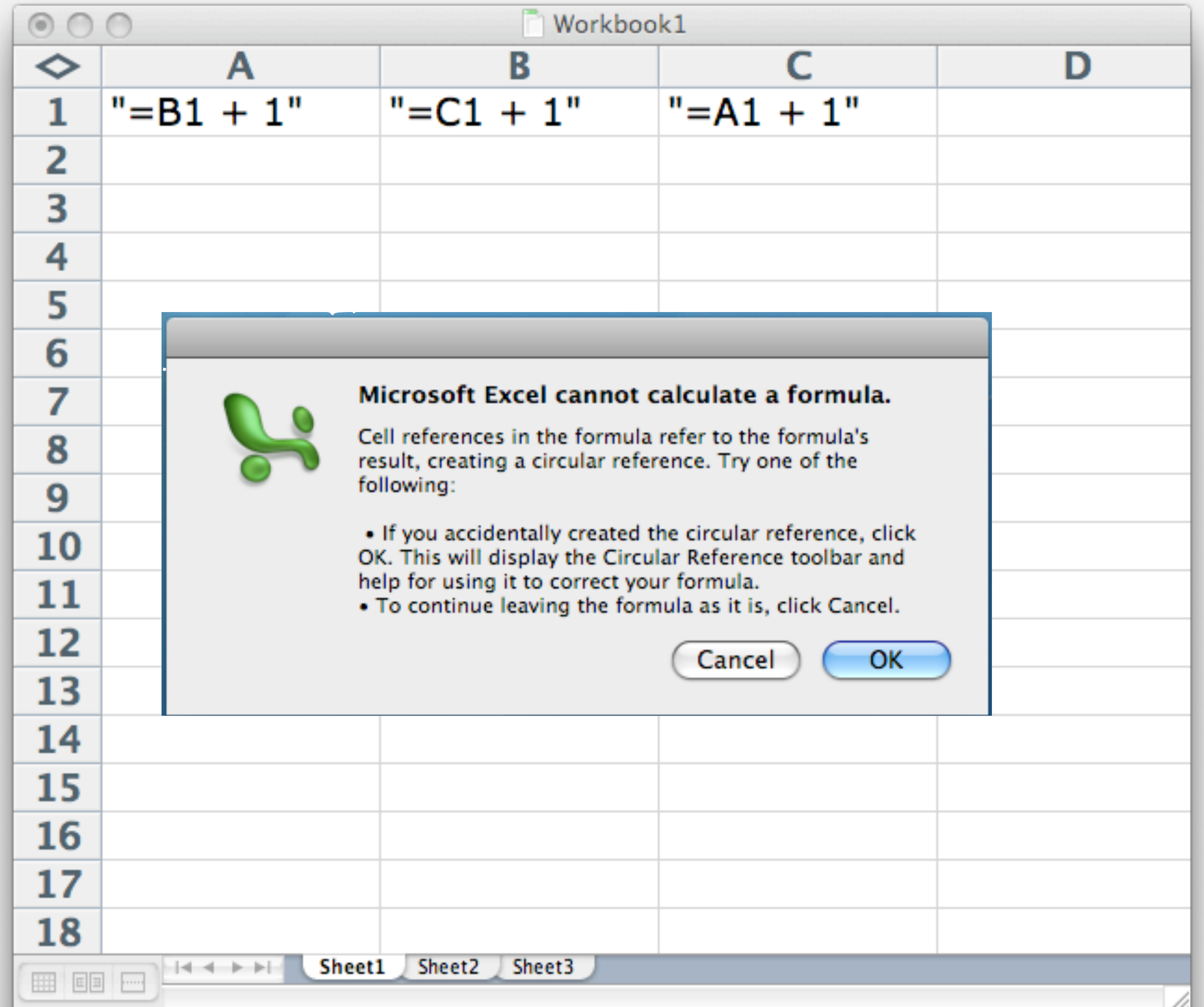
```
public class B extends C
{
   ...
}
```

```
public class C extends A
{
   ...
}
```

```
% javac A.java
A.java:1: cyclic inheritance
involving A
public class A extends B { }
       ^
1 error
```

Cycle detection application: spreadsheet recalculation

Microsoft Excel does cycle detection (and has a circular reference toolbar!)



Cycle detection application: symbolic links

The Linux file system does not do cycle detection.

```
% ln -s a.txt b.txt
% ln -s b.txt c.txt
% ln -s c.txt a.txt

% more a.txt
a.txt: Too many levels of symbolic links
```

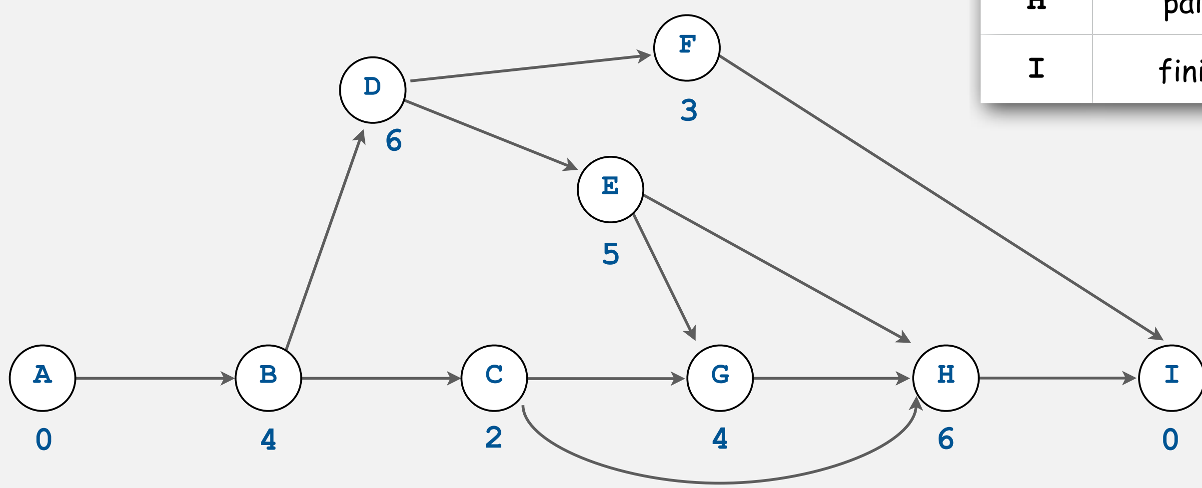
Topological sort application: precedence scheduling

Precedence scheduling.

- Task `v` takes `time[v]` units of time.
- Can work on jobs in parallel.
- Precedence constraints: must finish task `v` before beginning task `w`.
- Goal: finish each task as soon as possible.

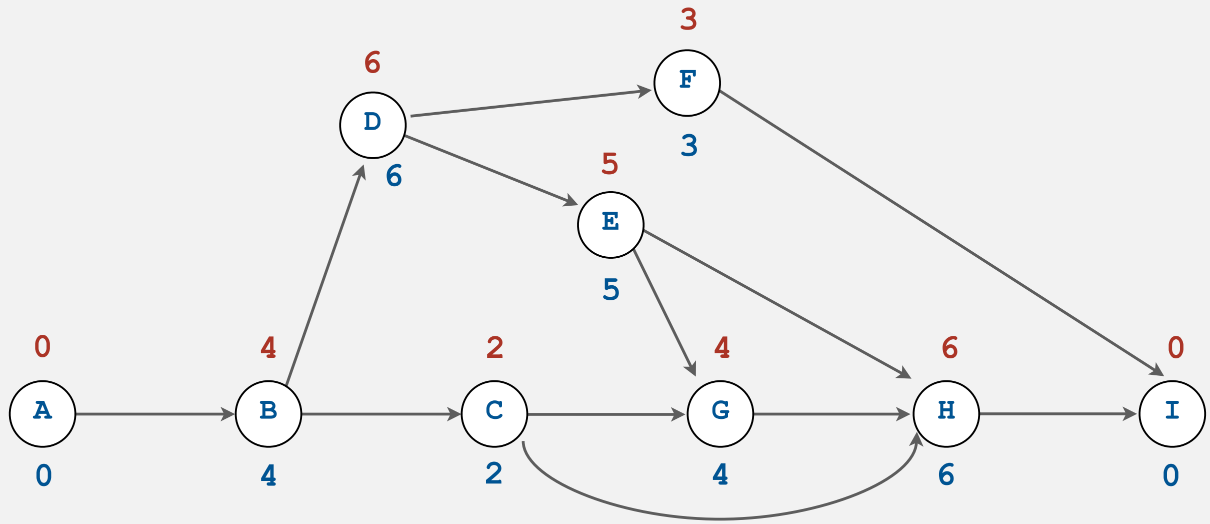
index	task	time	prereqs
A	begin	0	-
B	framing	4	A
C	roofing	2	B
D	siding	6	B
E	windows	5	D
F	plumbing	3	D
G	electricity	4	C, E
H	paint	6	C, E
I	finish	0	F, H

Ex.



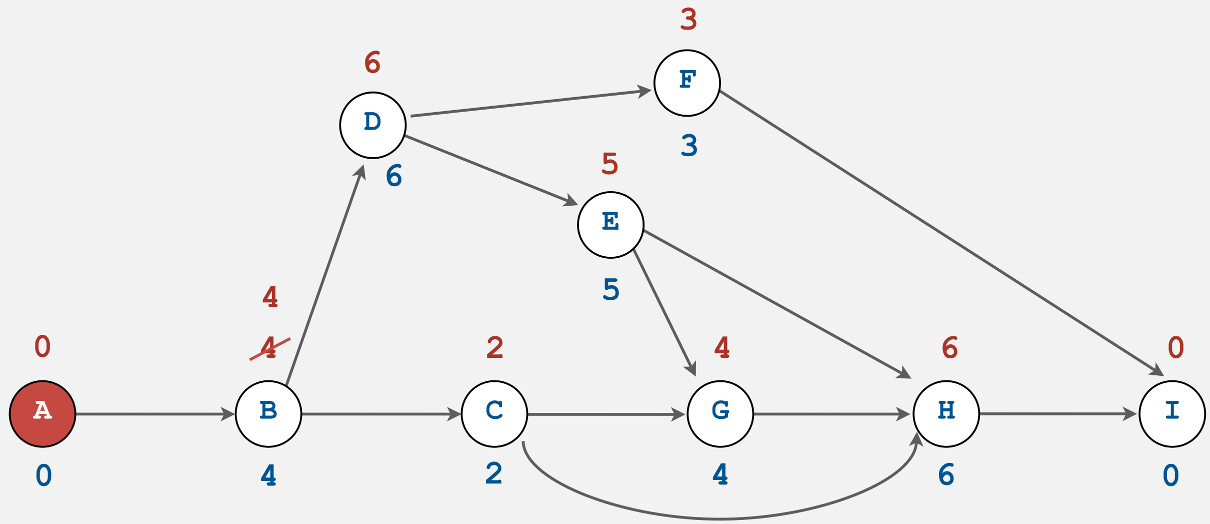
PERT/CPM algorithm.

- Compute topological order of vertices.
- Initialize `fin[v] = time[v]` for all vertices `v`.
- Consider vertices `v` in topologically sorted order.
  - for each edge `v→w`, set `fin[w] = max(fin[w], fin[v] + time[w])`



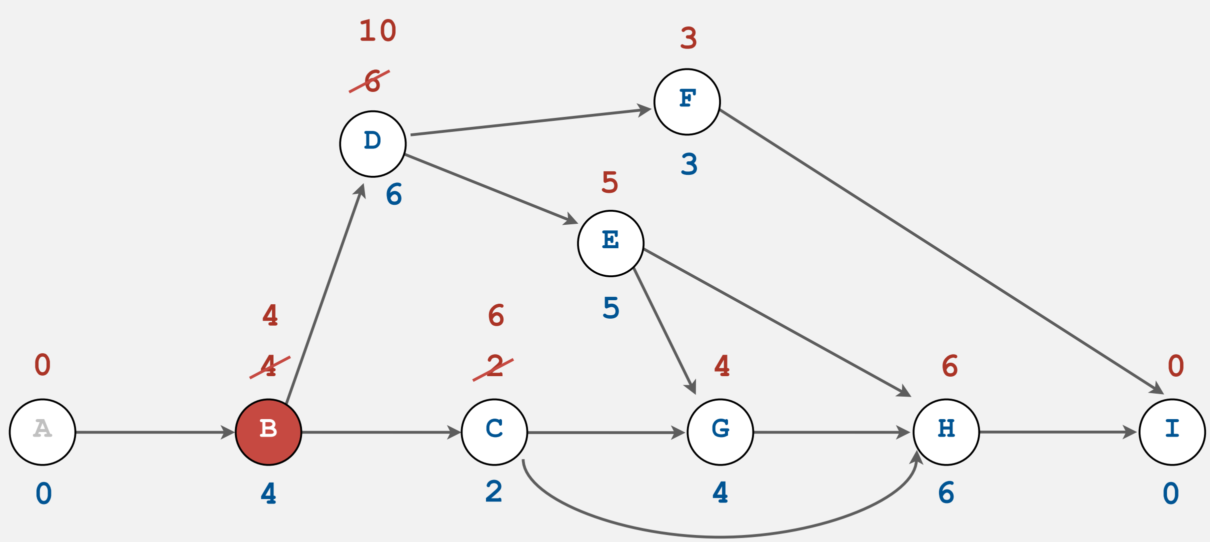
PERT/CPM algorithm.

- Compute topological order of vertices.
- Initialize **fin[v] = time[v]** for all vertices **v**.
- Consider vertices **v** in topologically sorted order.
  - for each edge **v→w**, set **fin[w] = max(fin[w], fin[v] + time[w])**



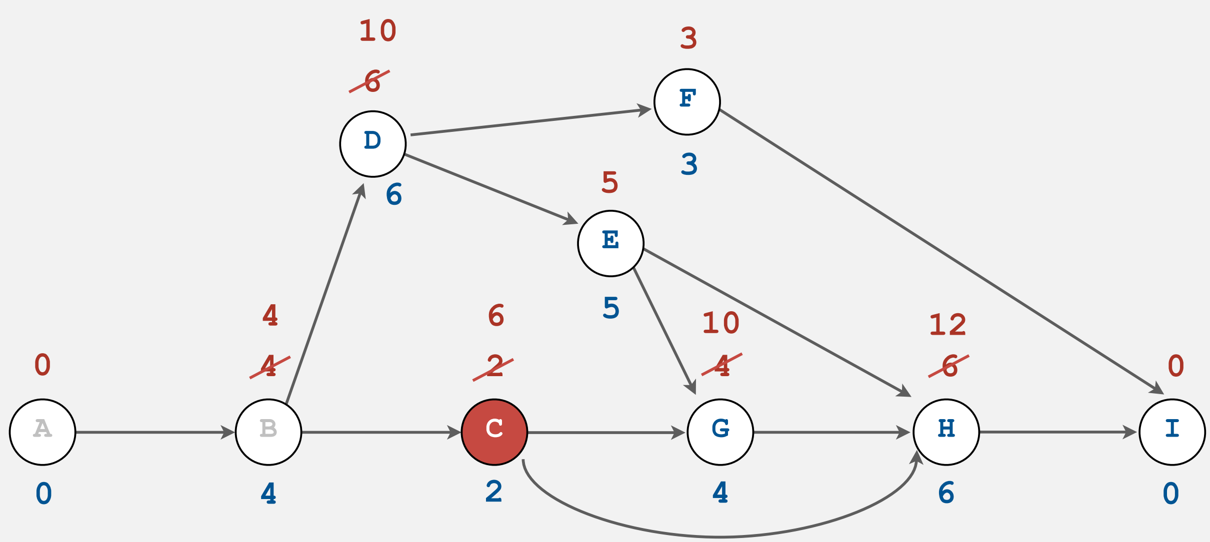
PERT/CPM algorithm.

- Compute topological order of vertices.
- Initialize `fin[v] = time[v]` for all vertices `v`.
- Consider vertices `v` in topologically sorted order.
  - for each edge `v→w`, set `fin[w] = max(fin[w], fin[v] + time[w])`



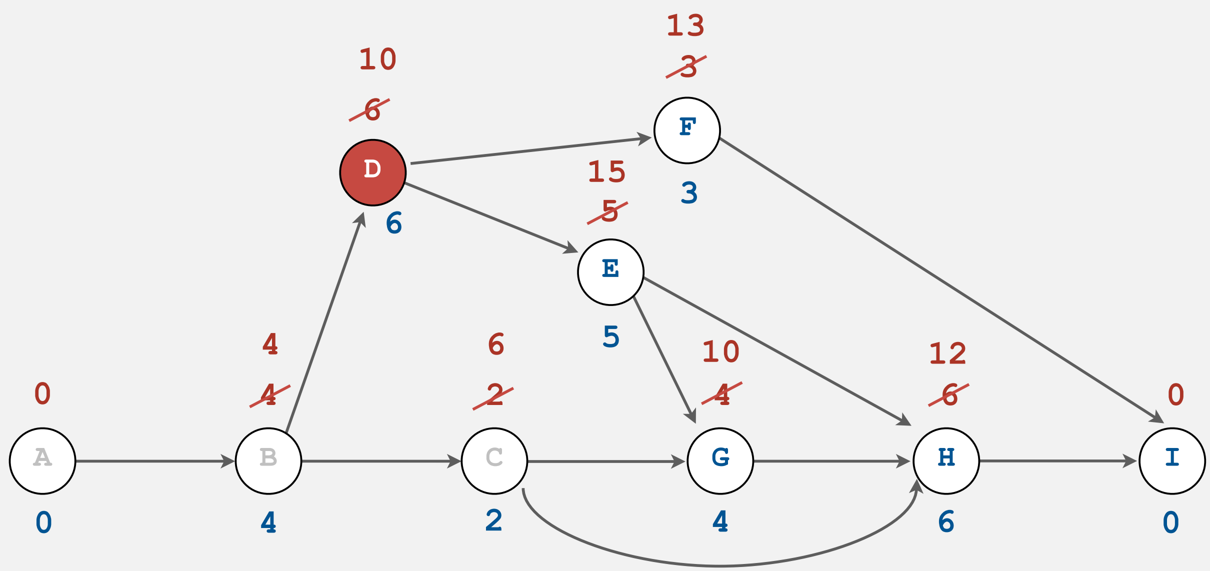
PERT/CPM algorithm.

- Compute topological order of vertices.
- Initialize `fin[v] = time[v]` for all vertices `v`.
- Consider vertices `v` in topologically sorted order.
  - for each edge `v`→`w`, set `fin[w] = max(fin[w], fin[v] + time[w])`



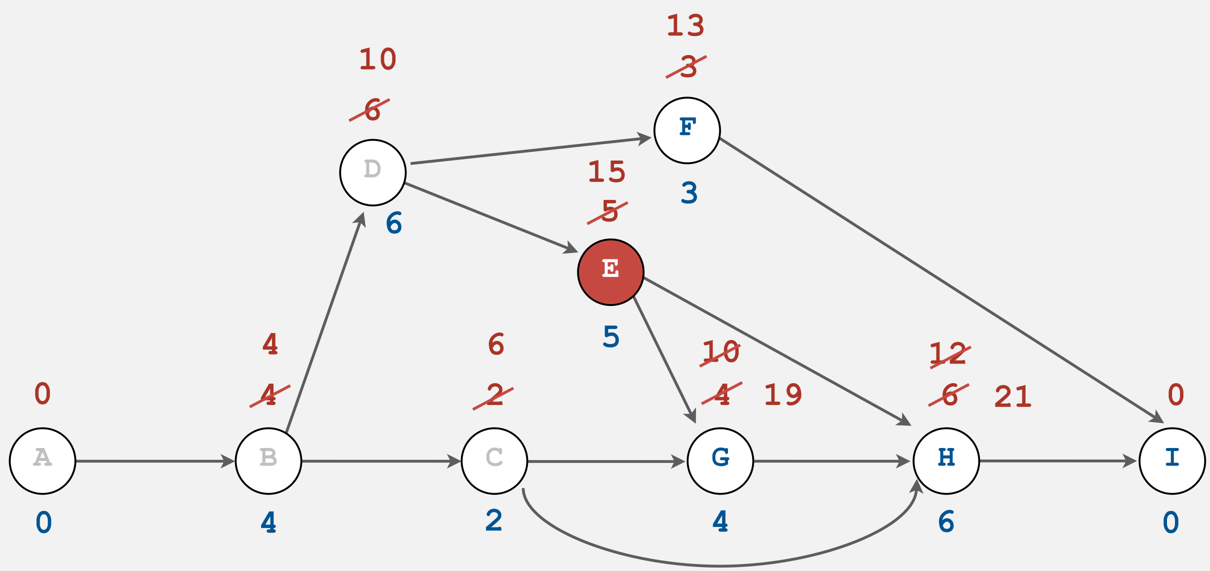
PERT/CPM algorithm.

- Compute topological order of vertices.
- Initialize `fin[v] = time[v]` for all vertices `v`.
- Consider vertices `v` in topologically sorted order.
  - for each edge `v→w`, set `fin[w] = max(fin[w], fin[v] + time[w])`



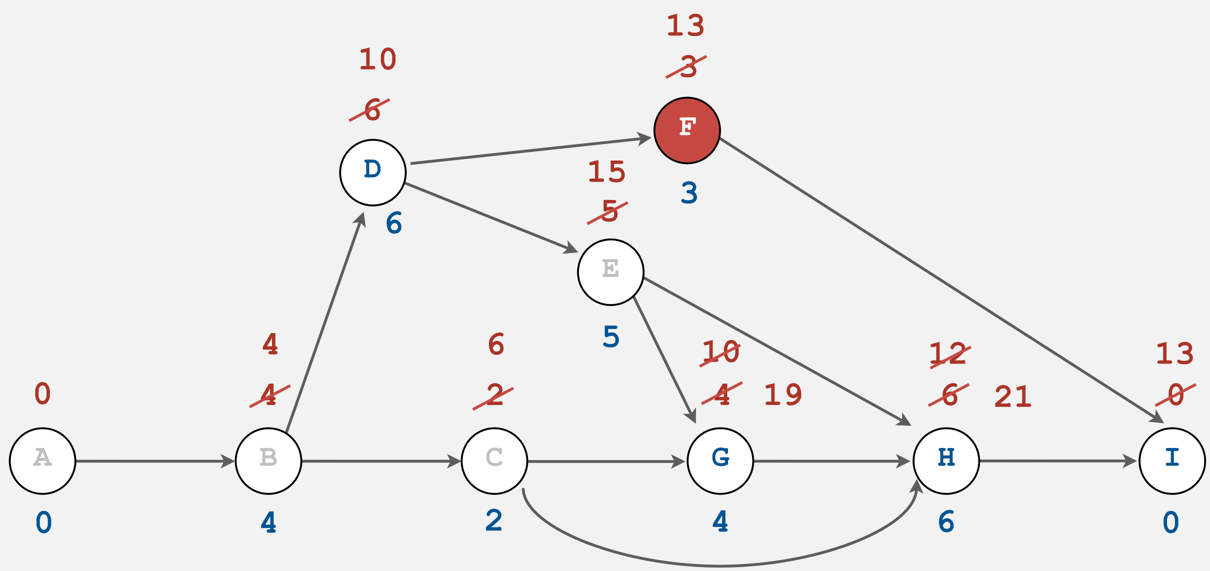
PERT/CPM algorithm.

- Compute topological order of vertices.
- Initialize `fin[v] = time[v]` for all vertices `v`.
- Consider vertices `v` in topologically sorted order.
  - for each edge `v`→`w`, set `fin[w] = max(fin[w], fin[v] + time[w])`



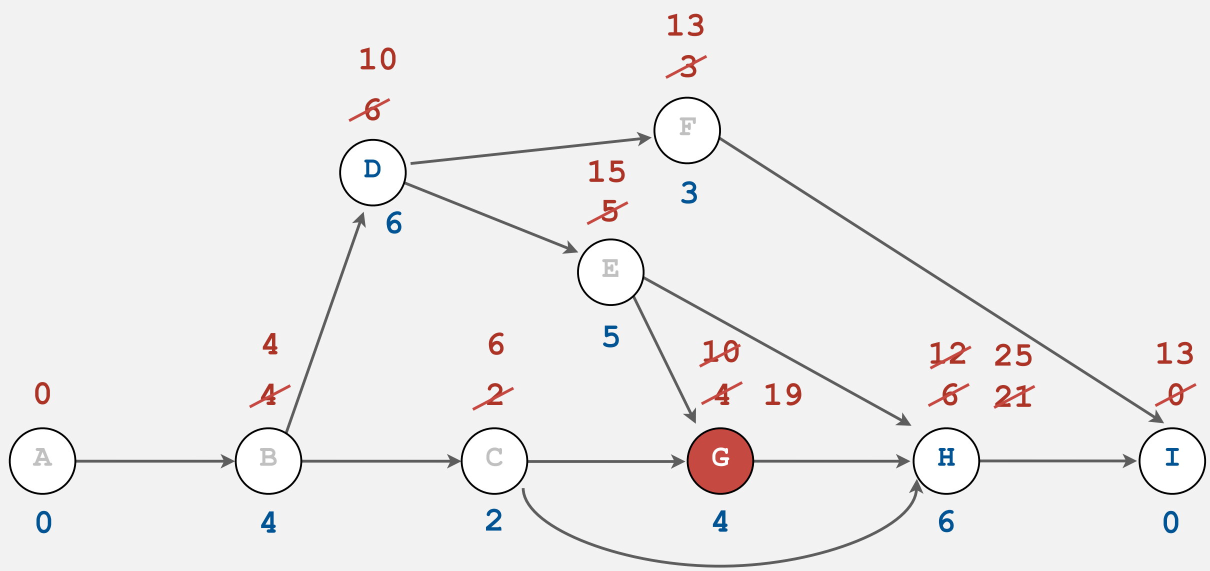
PERT/CPM algorithm.

- Compute topological order of vertices.
- Initialize `fin[v] = time[v]` for all vertices `v`.
- Consider vertices `v` in topologically sorted order.
  - for each edge `v→w`, set `fin[w] = max(fin[w], fin[v] + time[w])`



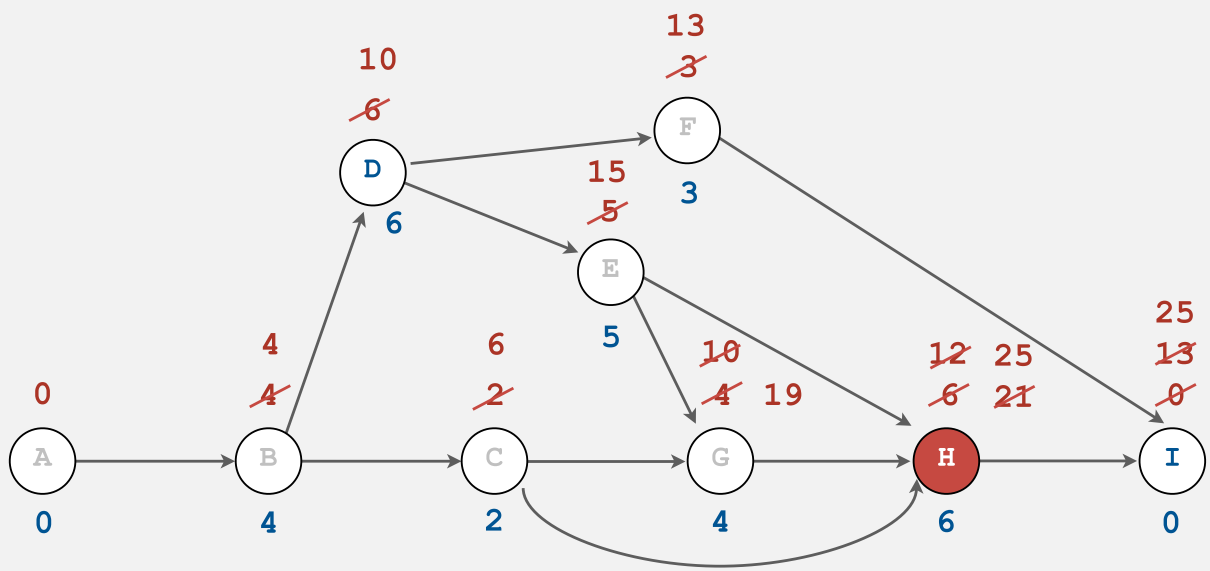
PERT/CPM algorithm.

- Compute topological order of vertices.
- Initialize `fin[v] = time[v]` for all vertices `v`.
- Consider vertices `v` in topologically sorted order.
  - for each edge `v→w`, set `fin[w] = max(fin[w], fin[v] + time[w])`



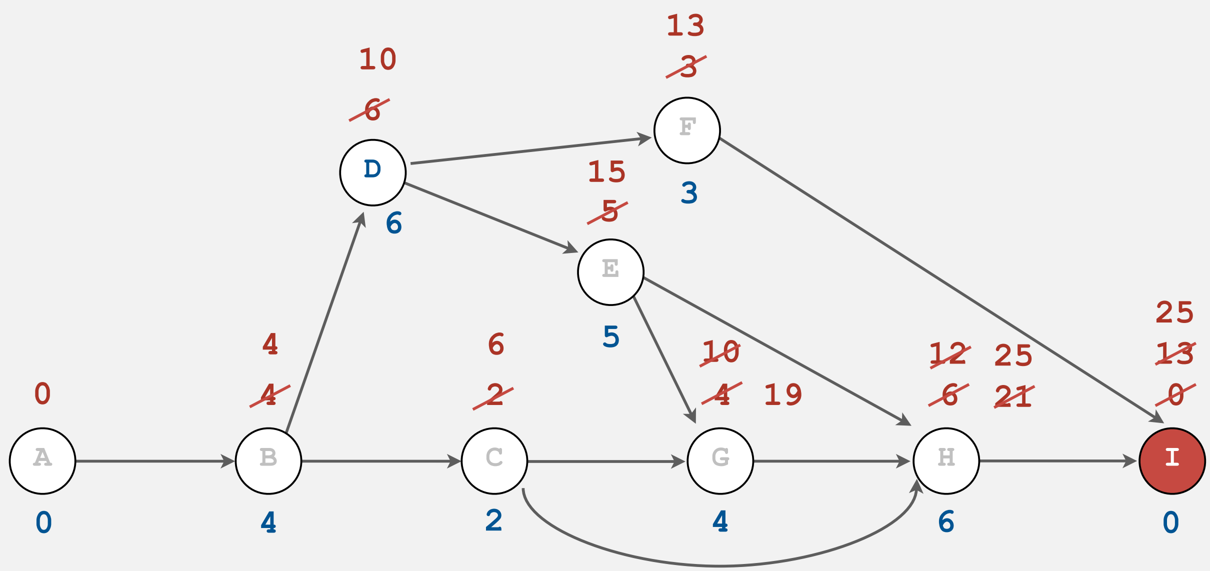
PERT/CPM algorithm.

- Compute topological order of vertices.
- Initialize `fin[v] = time[v]` for all vertices `v`.
- Consider vertices `v` in topologically sorted order.
  - for each edge `v→w`, set `fin[w] = max(fin[w], fin[v] + time[w])`



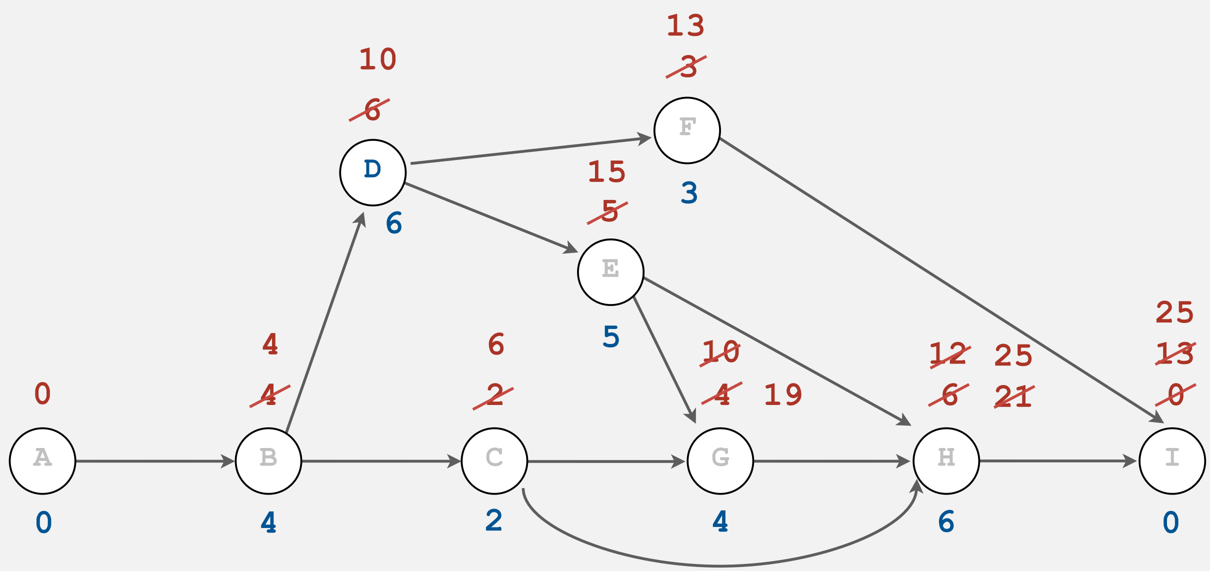
PERT/CPM algorithm.

- Compute topological order of vertices.
- Initialize `fin[v] = time[v]` for all vertices `v`.
- Consider vertices `v` in topologically sorted order.
  - for each edge `v→w`, set `fin[w] = max(fin[w], fin[v] + time[w])`



PERT/CPM algorithm.

- Compute topological order of vertices.
- Initialize `fin[v] = time[v]` for all vertices `v`.
- Consider vertices `v` in topologically sorted order.
  - for each edge `v`→`w`, set `fin[w] = max(fin[w], fin[v] + time[w])`



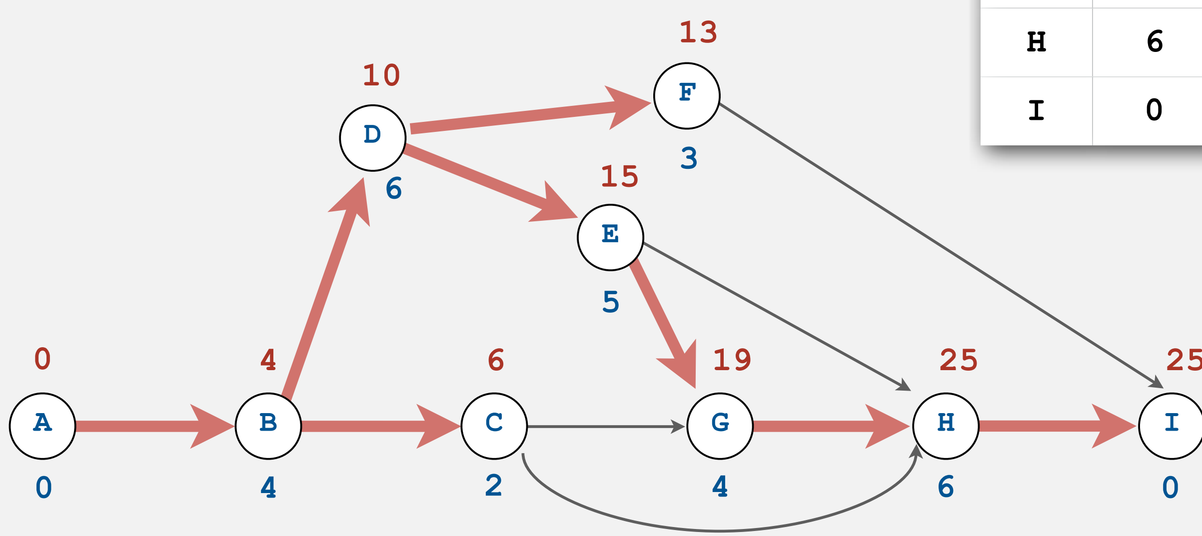
Program Evaluation and Review Technique / Critical Path Method

Critical path. Longest path from source to sink.

To compute:

- Remember vertex that set value (parent-link).
- Work backwards from sink.

index	time	prereqs	finish
A	0	-	0
B	4	A	4
C	2	B	6
D	6	B	10
E	5	D	15
F	3	D	13
G	4	C, E	19
H	6	C, E	25
I	0	F, H	25



PERT/CPM: Java implementation

G = DAG of precedence constraints

```
double[] fin = new double[G.V()];
for (int v = 0; v < G.V(); v++)
   fin[v] = time[v];
```

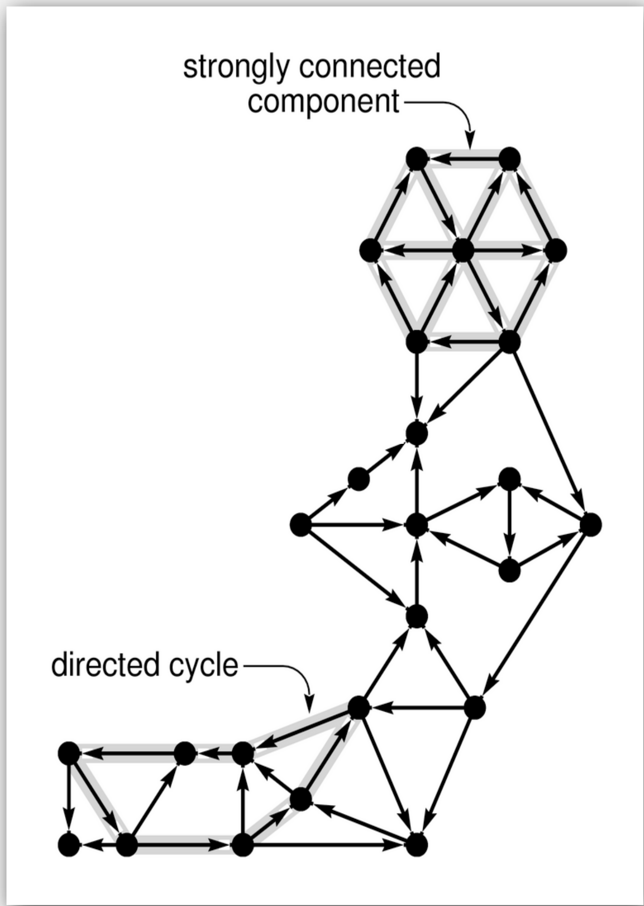
`fin[v]` = finishing time of task v

```
TopologicalSorter ts = new TopologicalSorter(G);
for (int v : ts.order())
   for (int w : G.adj(v))
      fin[w] = Math.max(fin[w], fin[v] + time[w]);
```

apply updates to vertices
in topological order

Def. Vertices v and w are strongly connected if there is a directed path from v to w and one from w to v.

Def. A strong component is a maximal subset of strongly connected vertices.



Digraph-processing challenge 3

Problem. Are v and w strongly connected?

Goal. Linear preprocessing time, constant query time.

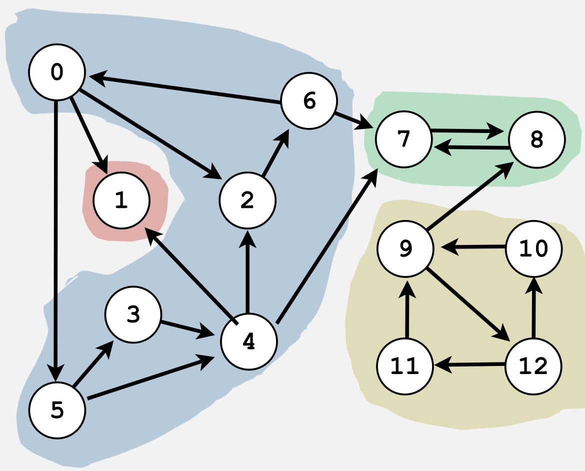
implementation: use DFS twice to find strong components (see textbook)

How difficult?

- Any COS 126 student could do it.
- ✓ Need to be a typical diligent COS 226 student.
- ✓ Hire an expert (or a COS 423 student).
- Intractable.
- No one knows.
- Impossible.

correctness proof

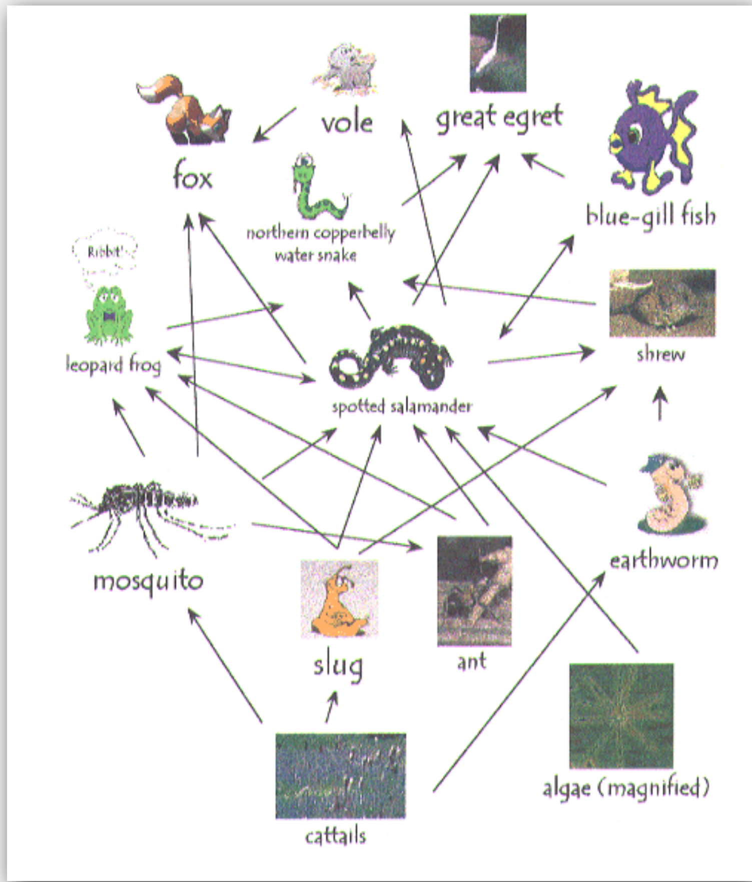
5 strong components



Ecological food web graph

Vertex = species.

Edge: from producer to consumer.

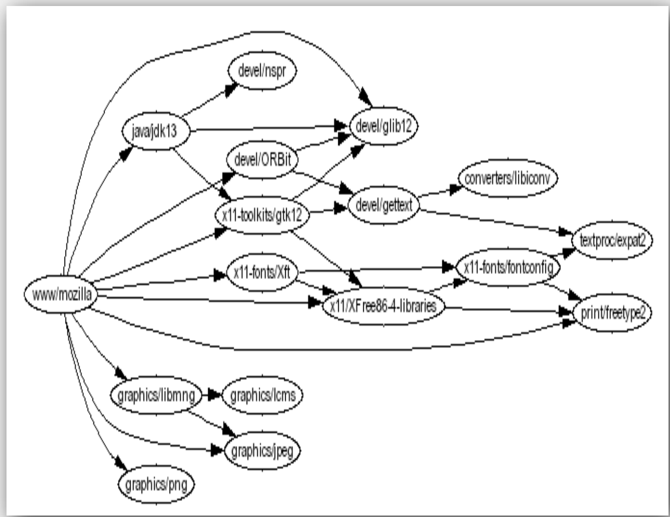


Strong component. Subset of species with common energy flow.

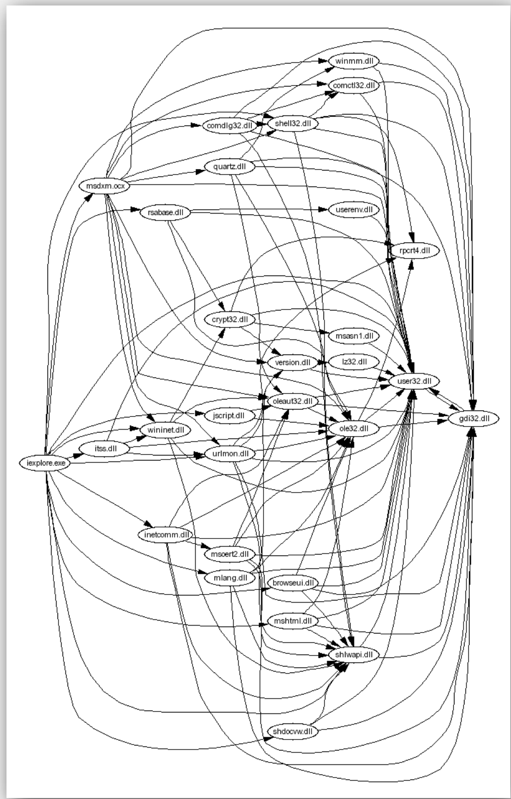
Software module dependency graph

Vertex = software module.

Edge: from module to dependency.



Firefox



Internet explorer

Strong component. Subset of mutually interacting modules.

Approach 1. Package strong components together.

Approach 2. Use to improve design!

Strong components algorithms: brief history

1960s: Core OR problem.

- Widely studied; some practical algorithms.
- Complexity not understood.

1972: linear-time DFS algorithm (Tarjan).

- Classic algorithm.
- Level of difficulty: CS226++.
- Demonstrated broad applicability and importance of DFS.

1980s: easy two-pass linear-time algorithm (Kosaraju).

- Forgot notes for teaching algorithms class; developed alg in order to teach it!
- Later found in Russian scientific literature (1972).

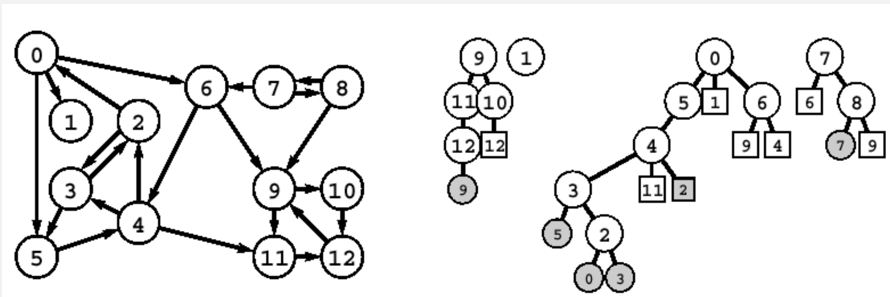
1990s: more easy linear-time algorithms (Gabow, Mehlhorn).

- Gabow: fixed old OR algorithm.
- Mehlhorn: needed one-pass algorithm for LEDA.

Simple (but mysterious) algorithm for computing strong components

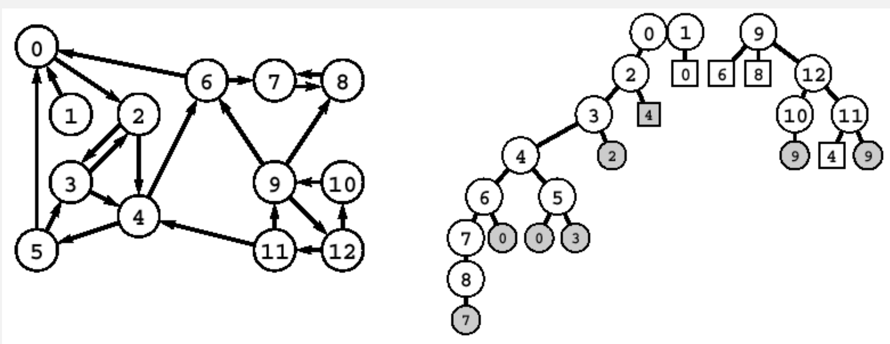
- Run DFS on G^R and compute postorder.
- Run DFS on G, considering vertices in reverse postorder.

G



	0	1	2	3	4	5	6	7	8	9	10	11	12
post	8	7	6	5	4	3	2	0	1	11	10	12	9

G^R

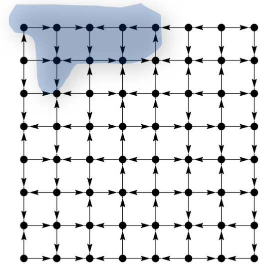
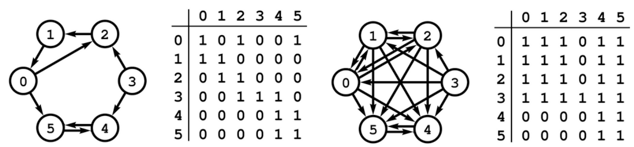
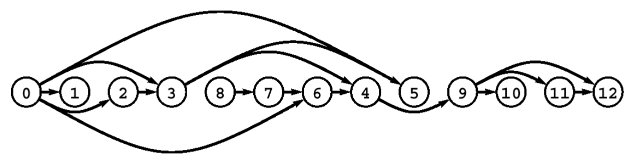
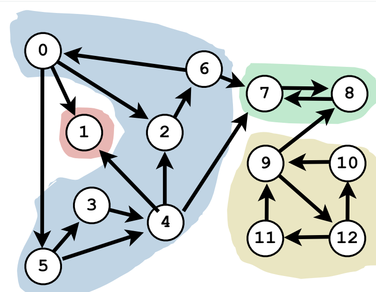


	0	1	2	3	4	5	6	7	8	9	10	11	12
sc	2	1	2	2	2	2	2	3	3	0	0	0	0

Proposition. Trees in second DFS are strong components. (!)

Pf. [see COS 423]

Digraph-processing summary: algorithms of the day

single-source reachability		DFS
transitive closure		DFS (from each vertex)
topological sort (DAG)		DFS
strong components		Kosaraju DFS (twice)

4.3 Minimum Spanning Trees

- weighted graph API
- cycles and cuts
- Kruskal's algorithm
- Prim's algorithm
- advanced topics

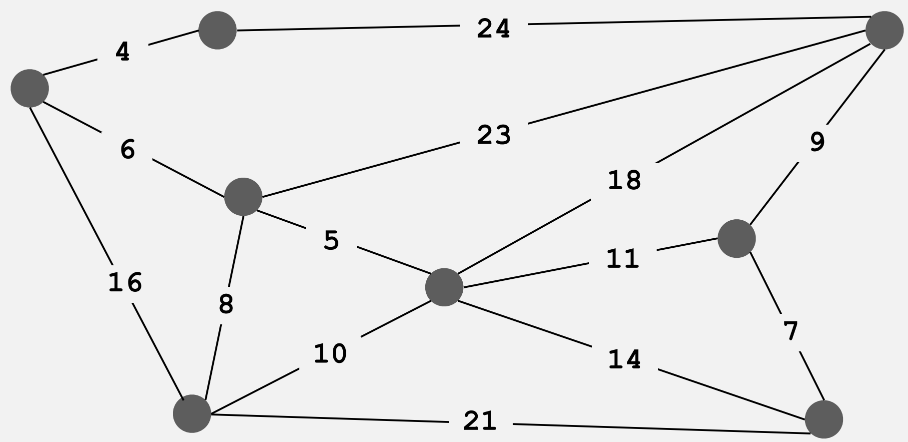
Reference: Algorithms in Java, 3rd edition, Part 5, Chapter 20

Minimum spanning tree

Given. Undirected graph G with positive edge weights (connected).

Def. A spanning tree of G is a subgraph T that is connected and acyclic.

Goal. Find a min weight spanning tree.



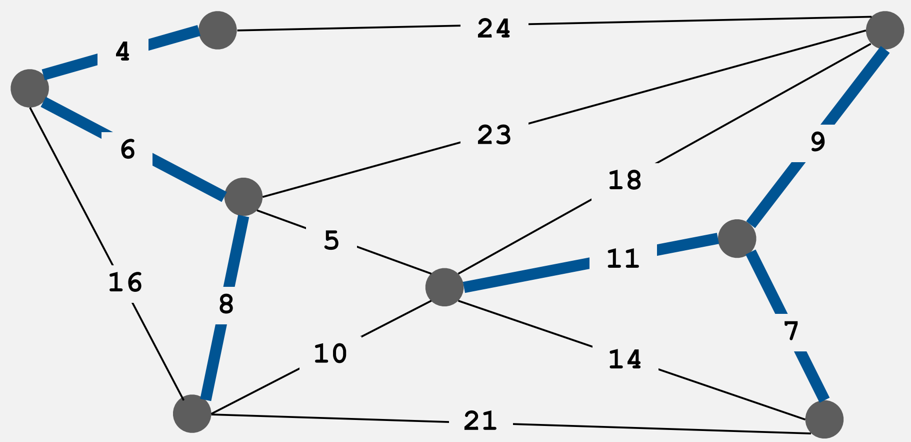
graph G

Minimum spanning tree

Given. Undirected graph G with positive edge weights (connected).

Def. A spanning tree of G is a subgraph T that is connected and acyclic.

Goal. Find a min weight spanning tree.



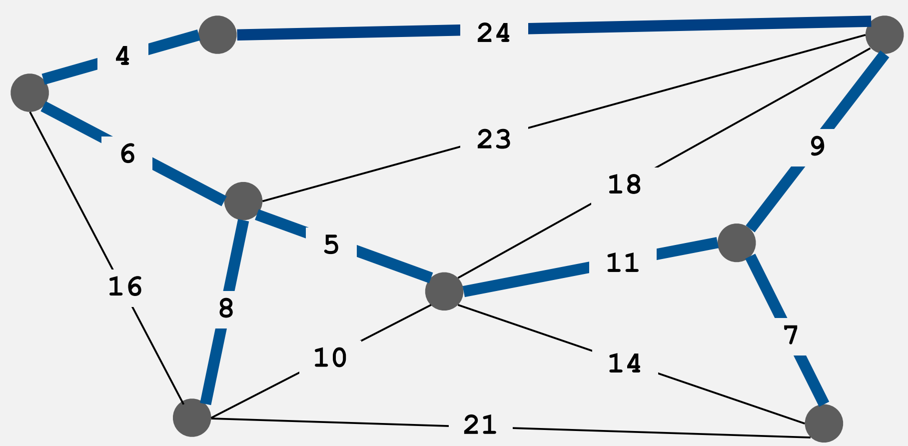
not connected

Minimum spanning tree

Given. Undirected graph G with positive edge weights (connected).

Def. A spanning tree of G is a subgraph T that is connected and acyclic.

Goal. Find a min weight spanning tree.



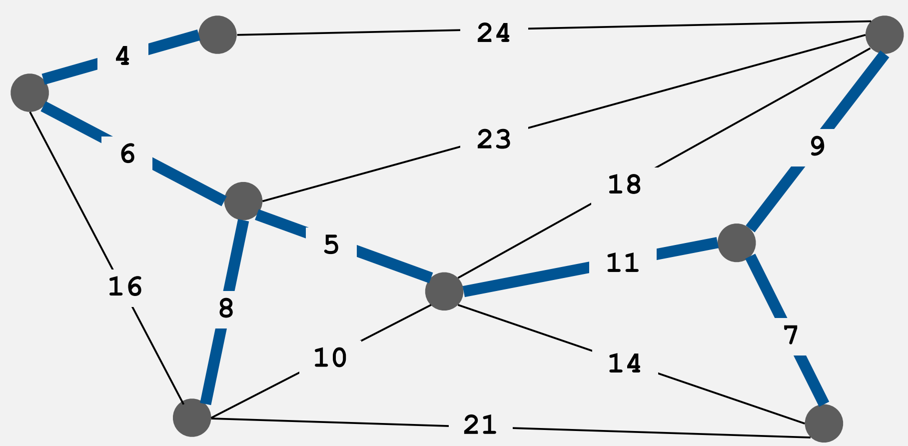
not acyclic

Minimum spanning tree

Given. Undirected graph G with positive edge weights (connected).

Def. A spanning tree of G is a subgraph T that is connected and acyclic.

Goal. Find a min weight spanning tree.



spanning tree T: cost = 50 = 4 + 6 + 8 + 5 + 11 + 9 + 7

Brute force. Try all spanning trees.

Applications

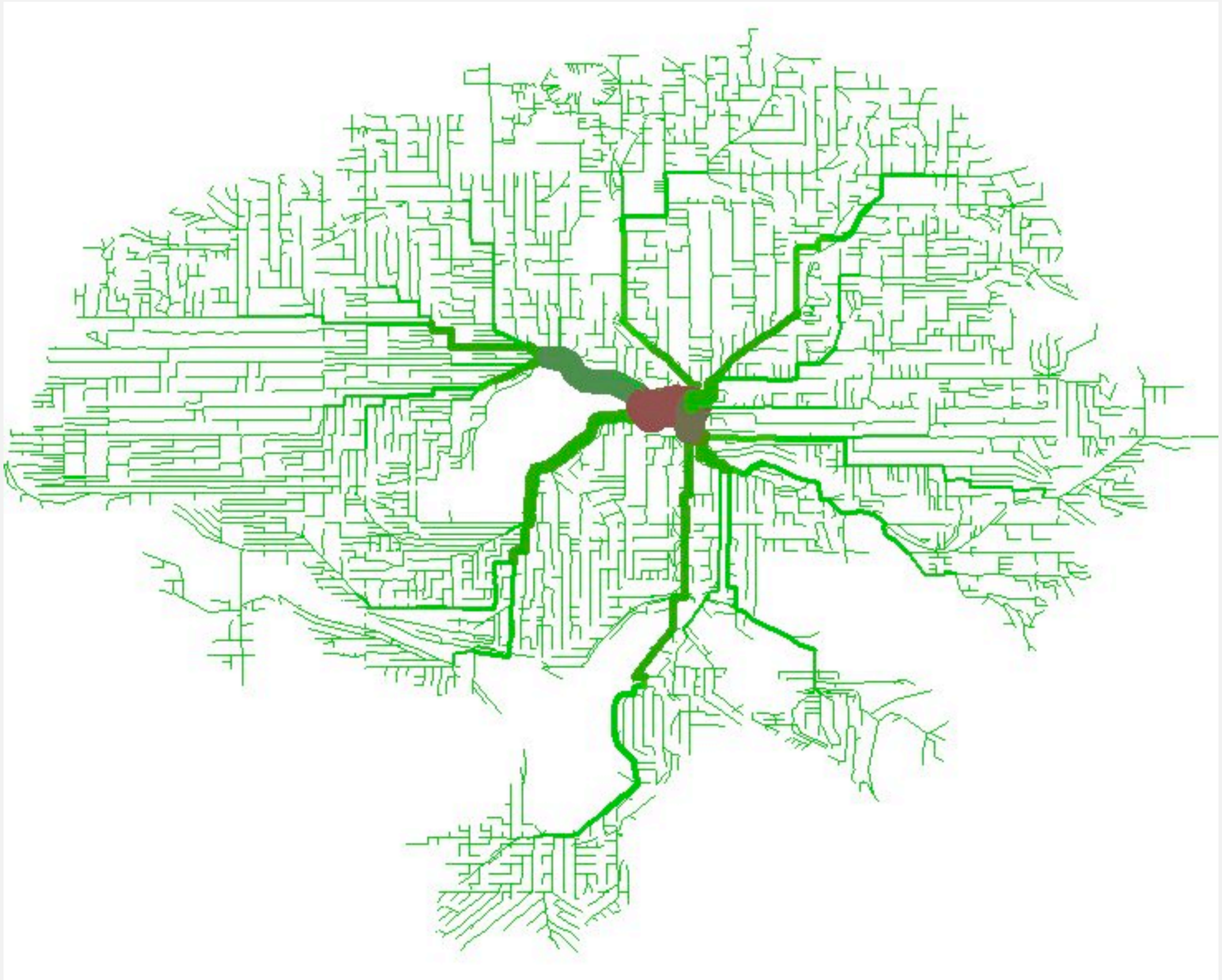
MST is fundamental problem with diverse applications.

- Cluster analysis.
- Max bottleneck paths.
- Real-time face verification.
- LDPC codes for error correction.
- Image registration with Renyi entropy.
- Find road networks in satellite and aerial imagery.
- Reducing data storage in sequencing amino acids in a protein.
- Model locality of particle interactions in turbulent fluid flows.
- Autoconfig protocol for Ethernet bridging to avoid cycles in a network.
- Network design (communication, electrical, hydraulic, cable, computer, road).
- Approximation algorithms for NP-hard problems (e.g., TSP, Steiner tree).

http://www.ics.uci.edu/~eppstein/gina/mst.html

Network design

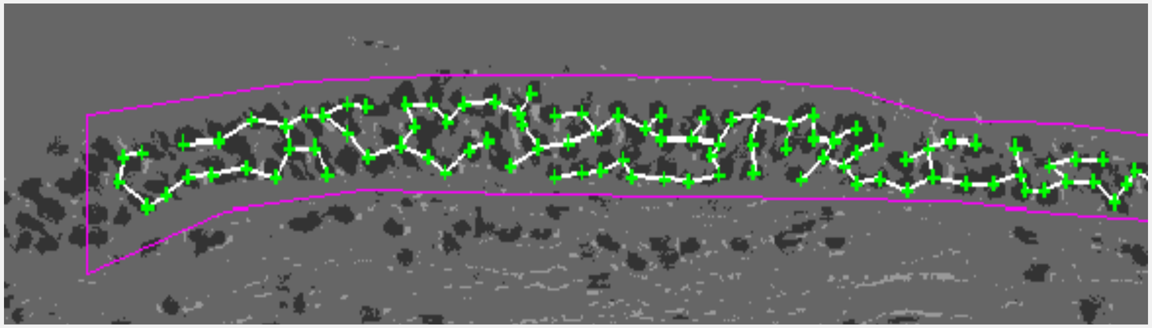
MST of bicycle routes in North Seattle

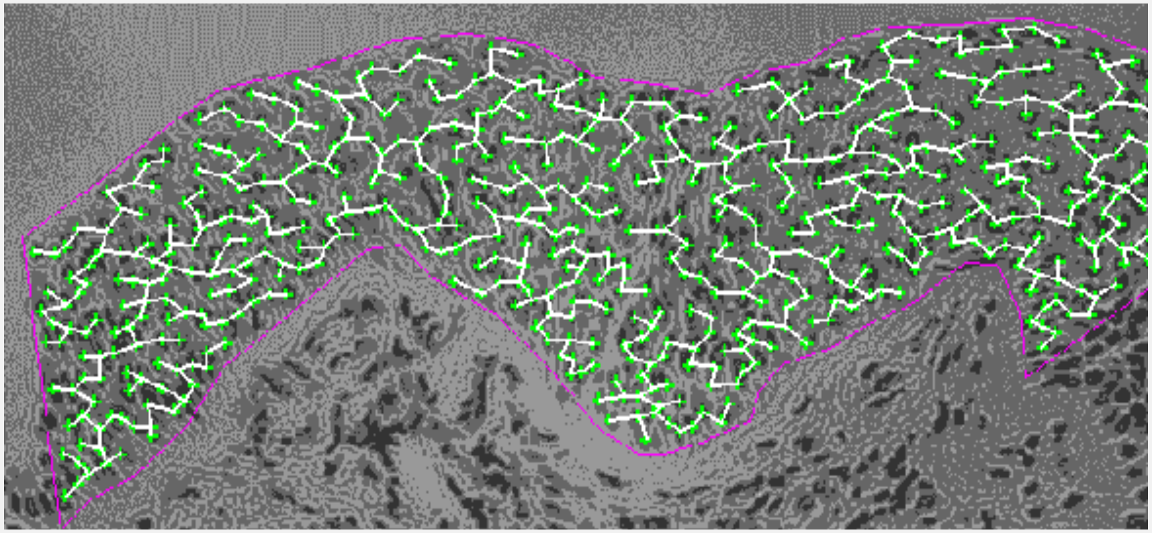


http://www.flickr.com/photos/ewedistrict/21980840

Medical image processing

MST describes arrangement of nuclei in the epithelium for cancer research

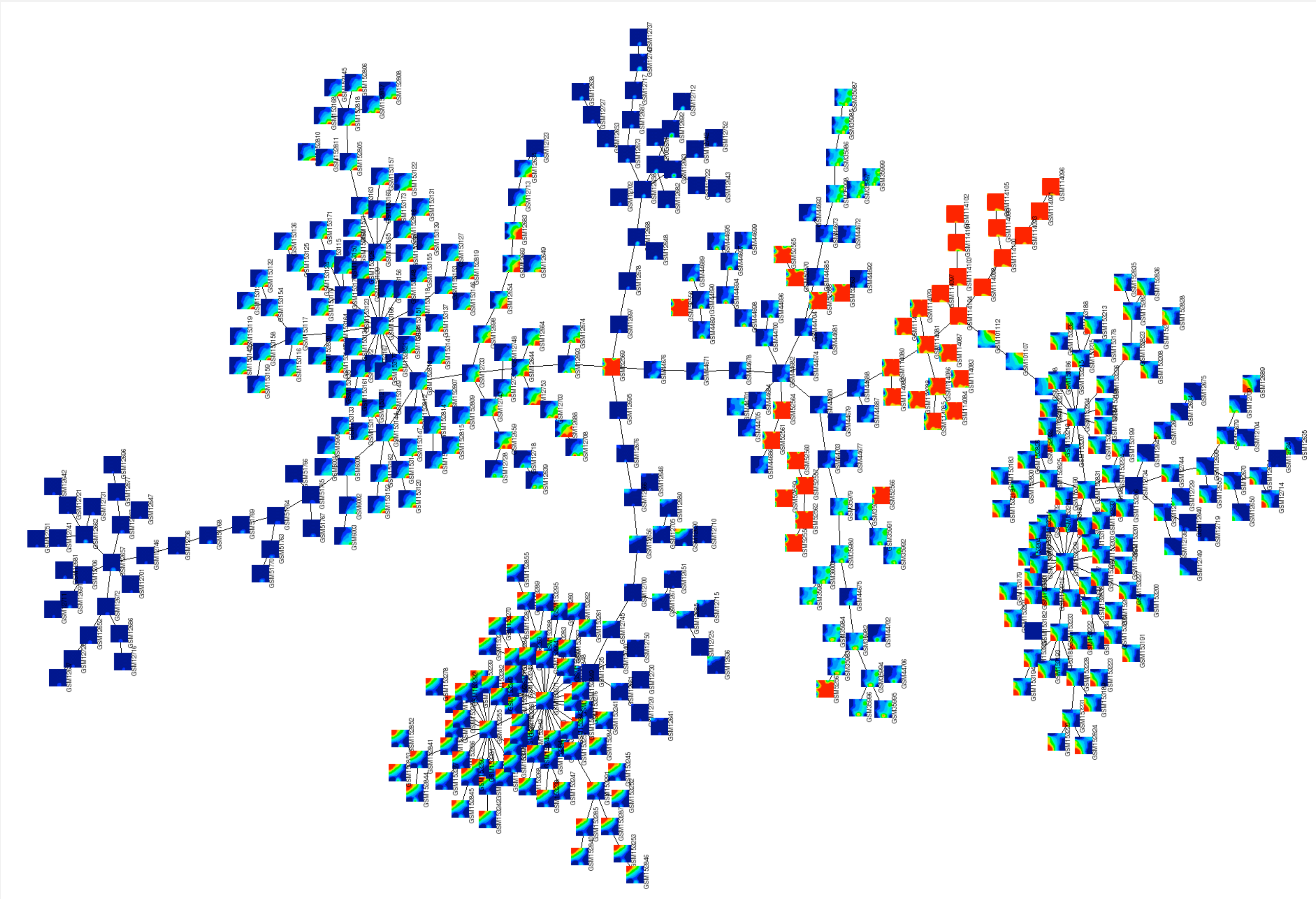




http://www.bccrc.ca/ci/ta01_archlevel.html

Genetic research

MST of tissue relationships measured by gene expression correlation coefficient



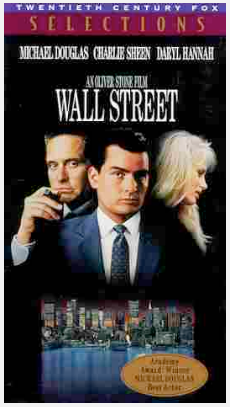
http://riodb.ibase.aist.go.jp/CELLPEDIA

Two greedy algorithms

Kruskal's algorithm. Consider edges in ascending order of weight. Add to T the next edge unless doing so would create a cycle.

Prim's algorithm. Start with any vertex s and greedily grow a tree T from s. At each step, add to T the edge of min weight with exactly one endpoint in T.

> *" Greed is good. Greed is right. Greed works. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit. " — Gordon Gecko*



Proposition. Both greedy algorithms compute MST.

- weighted graph API
- cycles and cuts
- Kruskal's algorithm
- Prim's algorithm
- advanced topics

Edge API

Edge abstraction needed for weighted edges.

public class Edge implements Comparable<Edge>		
	`Edge(int v, int w, double weight)`	*create a weighted edge v-w*
`int`	`either()`	*either endpoint*
`int`	`other(int v)`	*the endpoint that's not v*
`double`	`weight()`	*the weight*
`Comparator<Edge>`	`ByWeight()`	*compare by edge weight*

Weighted graph API

`public class WeightedGraph`		
	`WeightedGraph(int V)`	*create an empty graph with V vertices*
	`WeightedGraph(In in)`	*create a graph from input stream*
`void`	`addEdge(Edge e)`	*add edge e*
`void`	`removeEdge(Edge e)`	*delete edge e*
`Iterable<Edge>`	`adj(int v)`	*return an iterator over edges incident to v*
`int`	`V()`	*return number of vertices*

Conventions.

- Allow self-loops.
- Allow parallel edges (provided they have different weights).

Weighted graph API

	`public class WeightedGraph`	
	`WeightedGraph(int V)`	*create an empty graph with V vertices*
	`WeightedGraph(In in)`	*create a graph from input stream*
`void`	`addEdge(Edge e)`	*add edge e*
`void`	`removeEdge(Edge e)`	*delete edge e*
`Iterable<Edge>`	`adj(int v)`	*return an iterator over edges incident to v*
`int`	`V()`	*return number of vertices*

```
for (int v = 0; v < G.V(); v++)
{
   for (Edge e : G.adj(v))
   {
      int w = e.other(v);
      // process edge v-w
   }
}
```

iterate through all edges
(once in each direction)

Weighted graph: adjacency-set implementation

```
public class WeightedGraph
{
   private final int V;
   private final SET<Edge>[] adj;

   public WeightedGraph(int V)
   {
      this.V = V;
      adj = (SET<Edge>[]) new SET[V];
      for (int v = 0; v < V; v++)
         adj[v] = new SET<Edge>();
   }

   public void addEdge(Edge e)
   {
      int v = e.either(), w = e.other(v);
      adj[v].add(e);
      adj[w].add(e);
   }

   public Iterable<Edge> adj(int v)
   {  return adj[v];  }
}
```

same as **`Graph`**, but adjacency sets of **`Edges`** instead of integers

constructor

add edge to both adjacency sets

Weighted edge: Java implementation

```
public class Edge implements Comparable<Edge>
{
   private final int v, w;
   private final double weight;

   public Edge(int v, int w, double weight)
   {
      this.v = Math.min(v, w);
      this.w = Math.max(v, w);
      this.weight = weight;
   }

   public int either()
   {  return v;  }

   public int other(int vertex)
   {
      if (vertex == v) return w;
      else return v;
   }

   public int weight()
   {  return weight;  }

   // See next slide for compare methods.
}
```

constructor

either endpoint

other endpoint

weight of edge

Weighted edge: Java implementation (cont)

```
public static class ByWeight implements Comparator<Edge>
{
   public int compare(Edge e, Edge f)
   {
      if (e.weight < f.weight) return -1;
      if (e.weight > f.weight) return +1;
      return 0;
   }
}
```

order edges by weight
(for sorting in Kruskal)

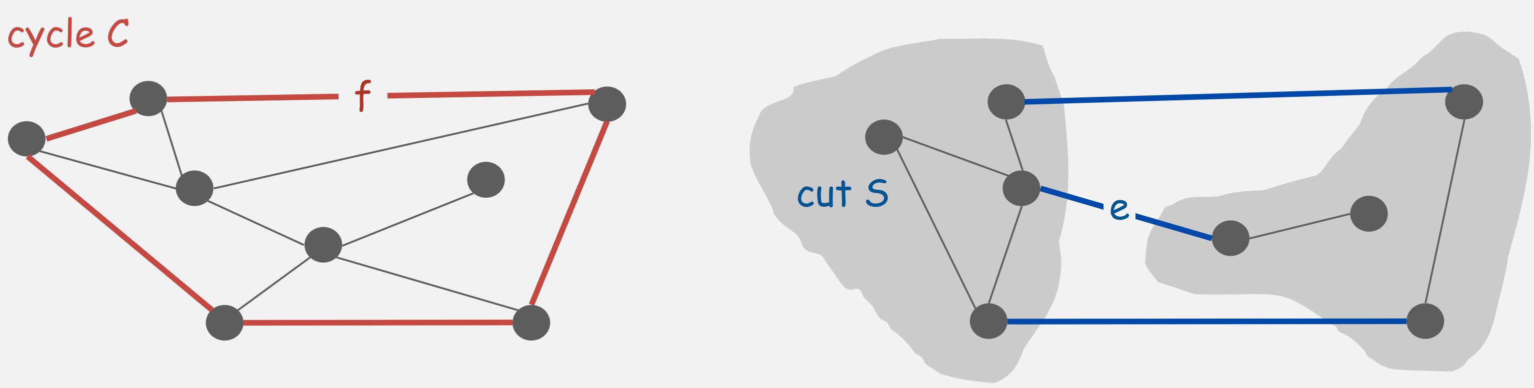
```
public int compareTo(Edge that)
{
    if (this.v < that.v) return -1;
    if (this.v > that.v) return +1;
    if (this.w < that.w) return -1;
    if (this.w > that.w) return +1;
    if (this.weight < that.weight) return -1;
    if (this.weight > that.weight) return +1;
    return 0;
}
```

lexicographic order,
breaking ties by weight
(for use in a symbol table)

Simplifying assumption. All edge weights w_e are distinct.

Cycle property. Let C be any cycle, and let f be the max weight edge belonging to C. Then the MST T* does not contain f.

Cut property. Let S be any subset of vertices, and let e be the min weight edge with exactly one endpoint in S. Then the MST contains e.



f is not in the MST T*

e is in the MST T*

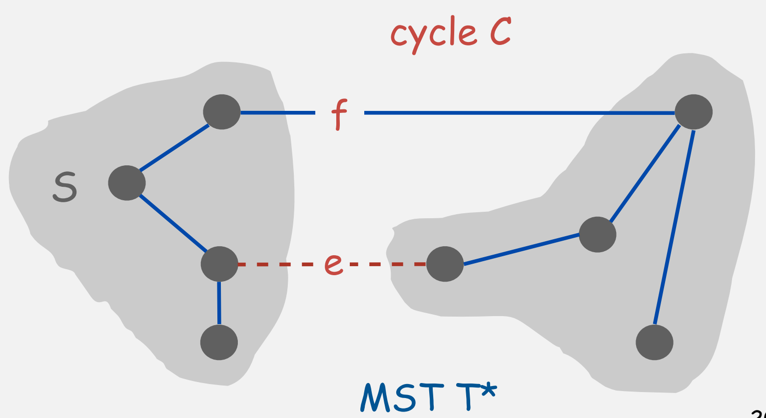
Cycle property: correctness proof

Simplifying assumption. All edge weights w_e are distinct.

Cycle property. Let C be any cycle, and let f be the max weight edge belonging to C. Then the MST T* does not contain f.

Pf. [by contradiction]

- Suppose f belongs to T*. Let's see what happens.
- Deleting f from T* disconnects T*. Let S be one side of the cut.
- Some other edge in C, say e, has exactly one endpoint in S.
- $T = T^* \cup \{ e \} - \{ f \}$ is also a spanning tree.
- Since $w_e < w_f$, weight(T) < weight(T*).
- Contradicts minimality of T*. ▪



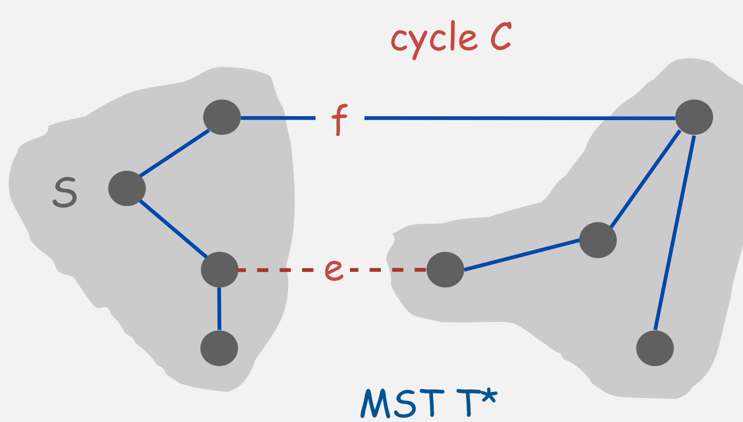
Cut property: correctness proof

Simplifying assumption. All edge weights w_e are distinct.

Cut property. Let S be any subset of vertices, and let e be the min weight edge with exactly one endpoint in S. Then the MST T^* contains e.

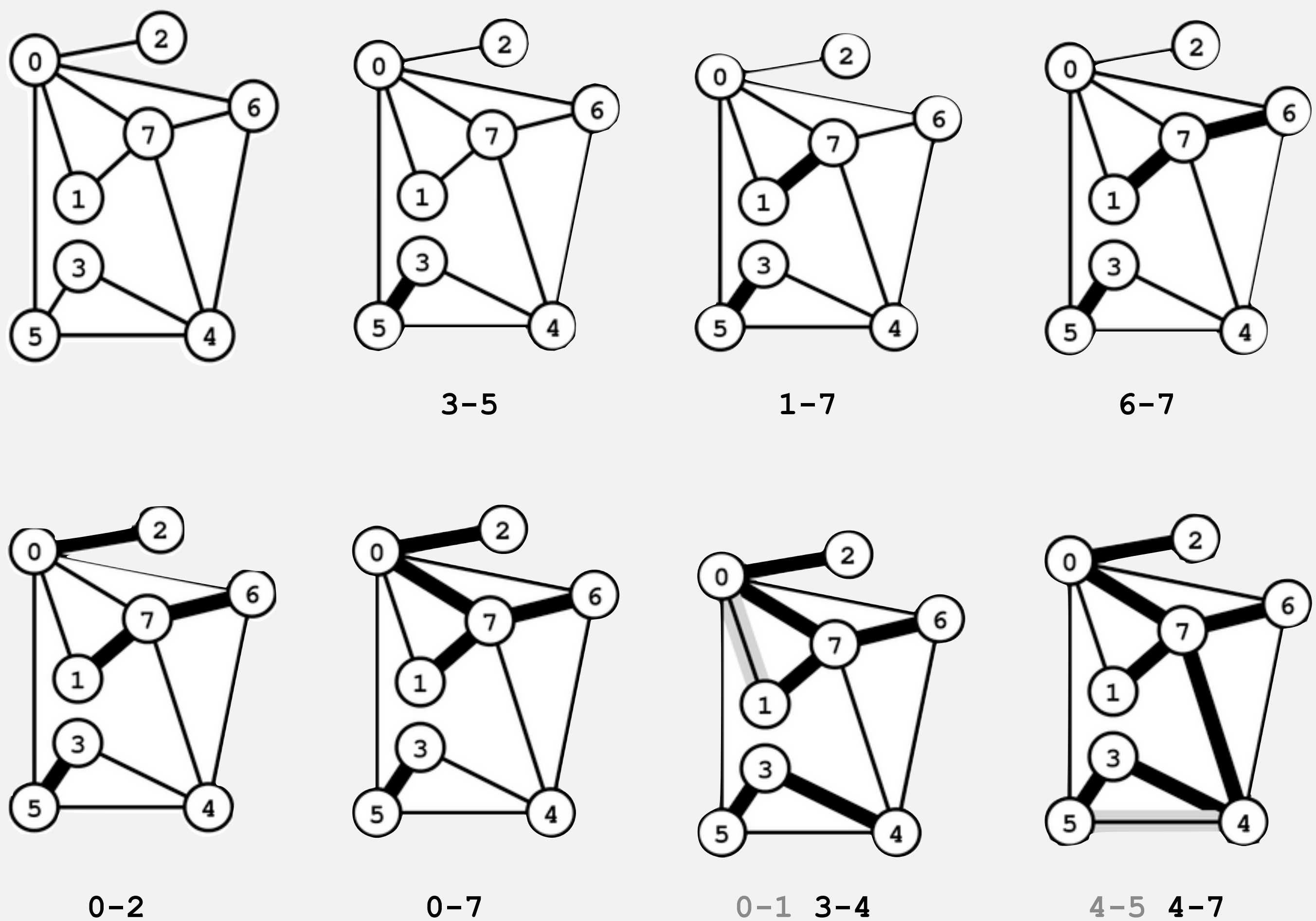
Pf. [by contradiction]

- Suppose e does not belong to T^*. Let's see what happens.
- Adding e to T^* creates a cycle C in T^*.
- Some other edge in C, say f, has exactly one endpoint in S.
- $T = T^* \cup \{ e \} - \{ f \}$ is also a spanning tree.
- Since $w_e < w_f$, weight(T) < weight(T^*).
- Contradicts minimality of T^*. ▪



Kruskal's algorithm

Kruskal's algorithm. [Kruskal 1956] Consider edges in ascending order of weight.
Add to T the next edge unless doing so would create a cycle.

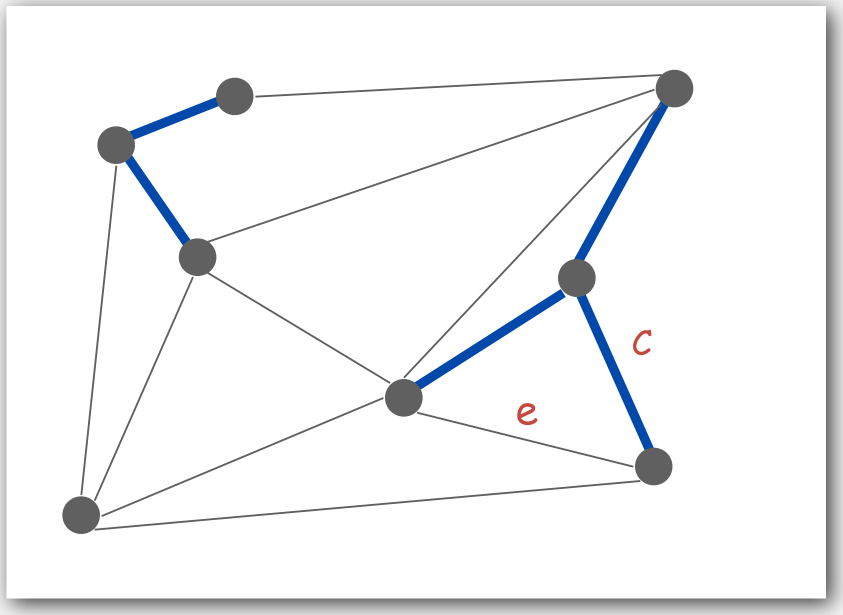


3-5	0.18
1-7	0.21
6-7	0.25
0-2	0.29
0-7	0.31
0-1	0.32
3-4	0.34
4-5	0.40
4-7	0.46
0-6	0.51
4-6	0.51
0-5	0.60

Proposition. Kruskal's algorithm computes the MST.

Pf. [Case 1] Suppose that adding e to T creates a cycle C.

- Edge e is the max weight edge in C. ← why max weight?
- Edge e is not in the MST (cycle property).



Kruskal's algorithm: correctness proof

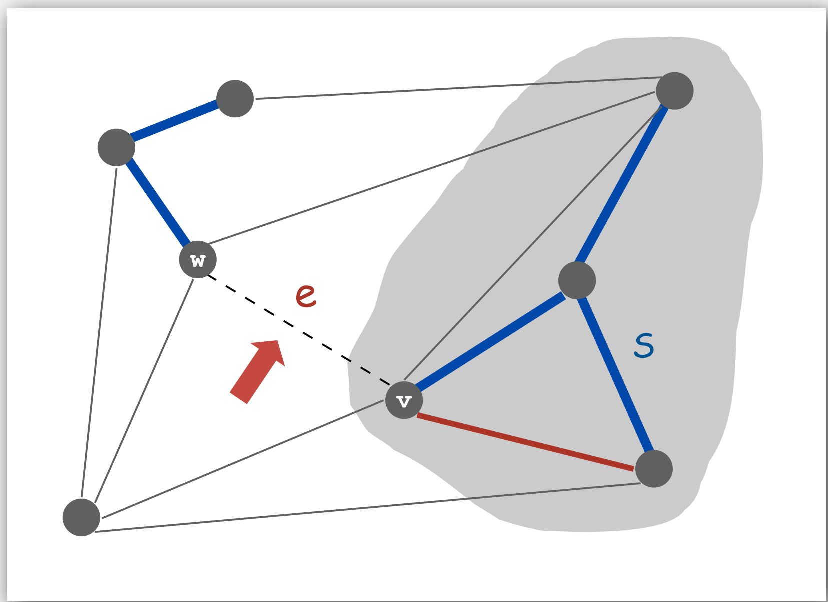
Proposition. Kruskal's algorithm computes the MST.

Pf. [Case 2] Suppose that adding e = v-w to T does not create a cycle.

- Let S be the vertices in v's connected component.
- Vertex w is not in S.
- Edge e is the min weight edge with exactly one endpoint in S.

why min weight?

- Edge e is in the MST (cut property). ▪

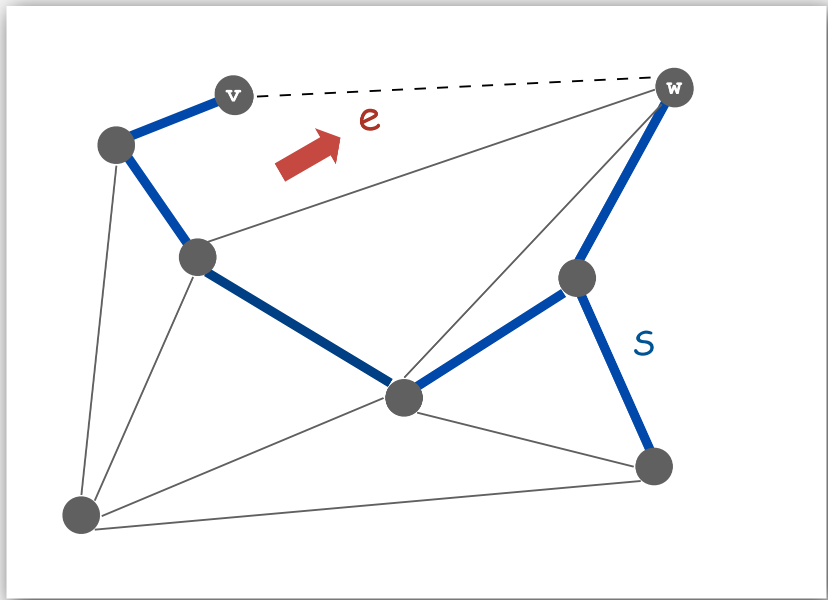


Kruskal implementation challenge

Problem. Check if adding an edge v-w to T creates a cycle.

How difficult?

- O(E + V) time.
- O(V) time. ← run DFS from v, check if w is reachable (T has at most V-1 edges)
- O(log V) time.
- O(log* V) time. ← use the union-find data structure !
- Constant time.

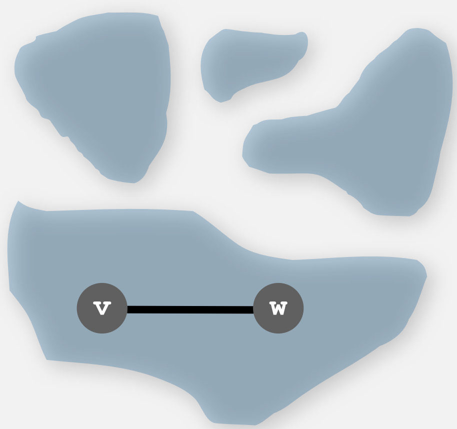


Kruskal's algorithm implementation

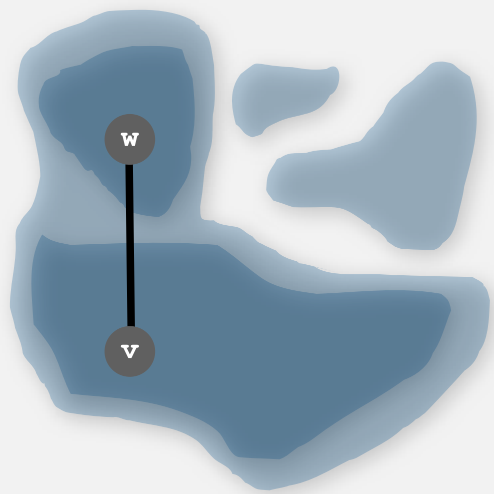
Problem. Check if adding an edge v-w to T creates a cycle.

Efficient solution. Use the union-find data structure.

- Maintain a set for each connected component in T.
- If v and w are in same component, then adding v-w creates a cycle.
- To add v-w to T, merge sets containing v and w.



Case 1: adding v-w creates a cycle



Case 2: add v-w to T and merge sets

Kruskal's algorithm: Java implementation

```
public class Kruskal
{
   private SET<Edge> mst = new SET<Edge>();

   public Kruskal(WeightedGraph G)
   {
      Edge[] edges = G.edges();
      Arrays.sort(edges, new Edge.ByWeight());

      UnionFind uf = new UnionFind(G.V());
      for (Edge e : edges)
      {
         int v = e.either(), w = e.other(v);
         if (!uf.find(v, w))
         {
            uf.unite(v, w);
            mst.add(e);
         }
      }
   }

   public Iterable<Edge> mst()
   {  return mst;   }
}
```

get all edges in graph

sort edges by weight

greedily add edges to MST

Kruskal's algorithm running time

Proposition. Kruskal's algorithm computes MST in O(E log E) time.

Pf.

operation	frequency	time per op
sort	1	E log E
union	V	log* V †
find	E	log* V †

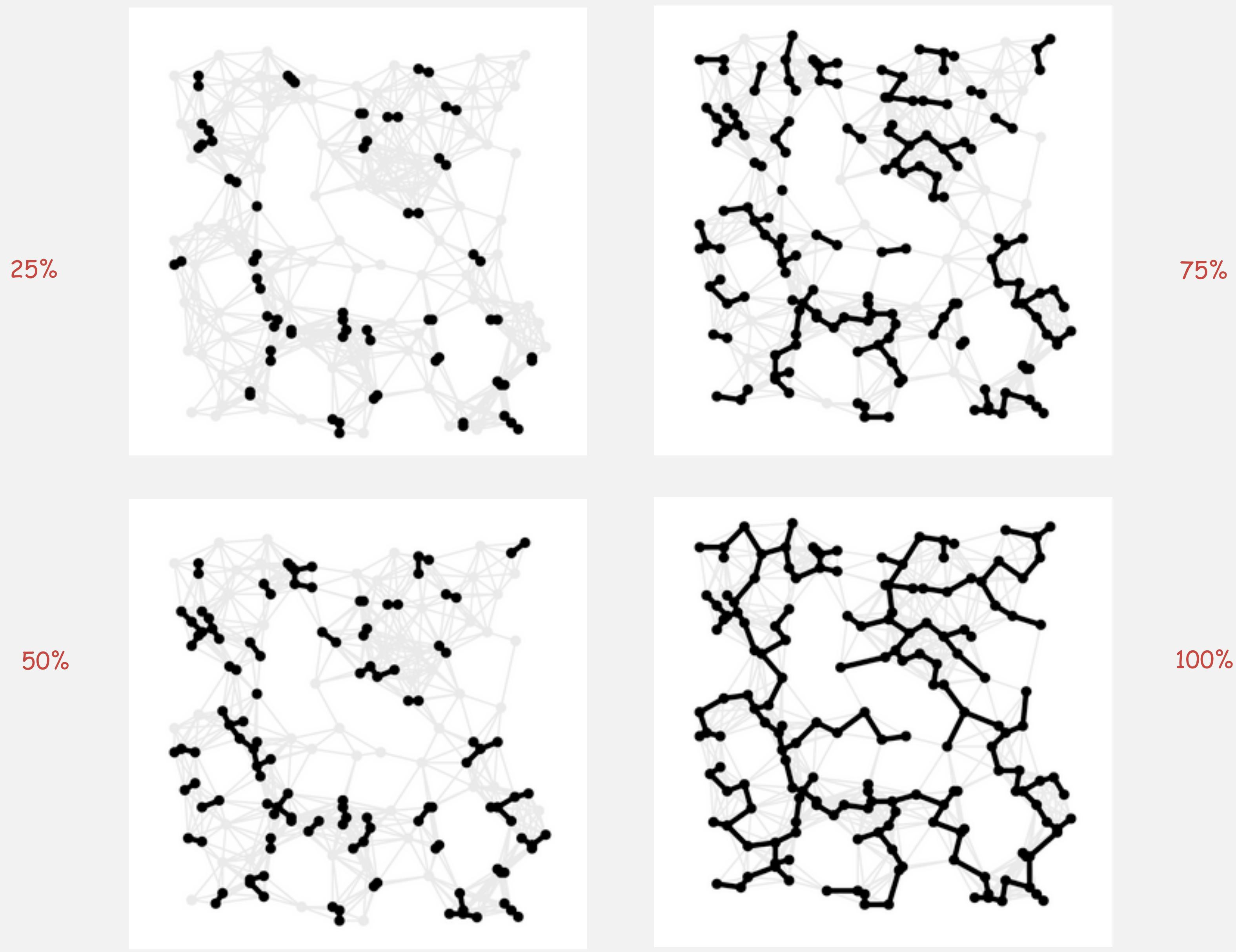
† amortized bound using weighted quick union with path compression

Improvements.

- Stop as soon as there are V-1 edges.
- If edges are already sorted, time is proportional to E log* V.

recall: log* V ≤ 5 in this universe

Kruskal's algorithm example

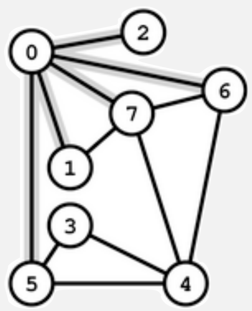


- weighted graph API
- cycles and cuts
- Kruskal's algorithm
- **Prim's algorithm**
- advanced topics

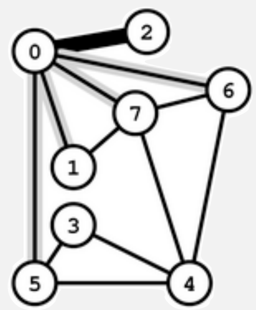
Prim's algorithm example

Prim's algorithm. [Jarník 1930, Dijkstra 1957, Prim 1959]

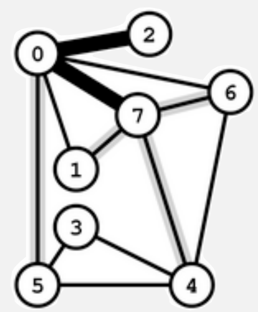
Start with vertex 0 and greedily grow tree T. At each step, add to T the edge of min weight with exactly one endpoint in T.



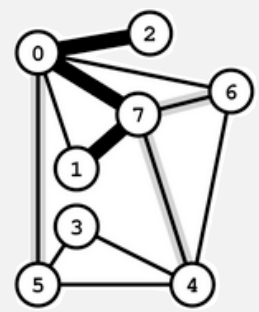
0-2 0-7 0-1 0-6 0-5



0-7 0-1 0-6 0-5

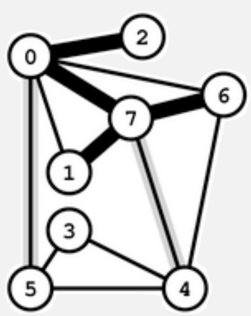


7-1 7-6 0-1 7-4 0-6 0-5

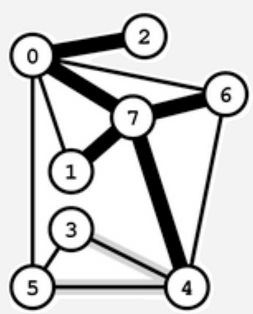


7-6 7-4 0-6 0-5

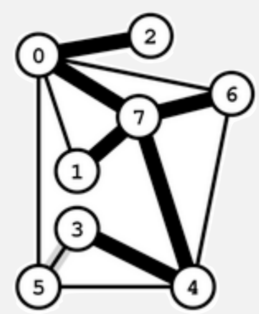
edges with exactly one endpoint in T, sorted by weight



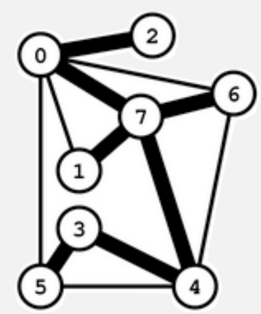
7-4 6-4 0-5



4-3 4-5 0-5



3-5 4-5 0-5



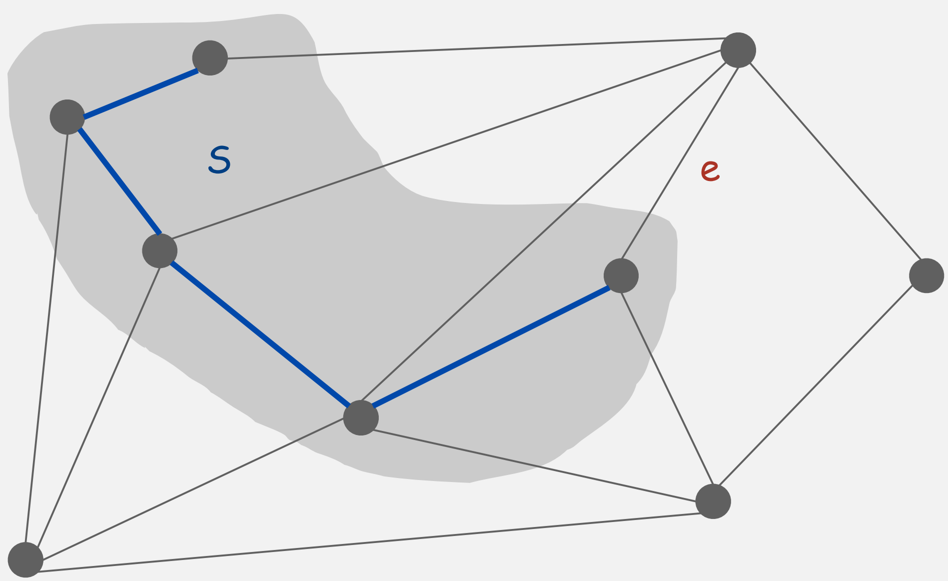
0-1	0.32
0-2	0.29
0-5	0.60
0-6	0.51
0-7	0.31
1-7	0.21
3-4	0.34
3-5	0.18
4-5	0.40
4-6	0.51
4-7	0.46
6-7	0.25

Prim's algorithm correctness proof

Proposition. Prim's algorithm computes the MST.

Pf.

- Let S be the subset of vertices in current tree T.
- Prim adds the min weight edge e with exactly one endpoint in S.
- Edge e is in the MST (cut property). ▪

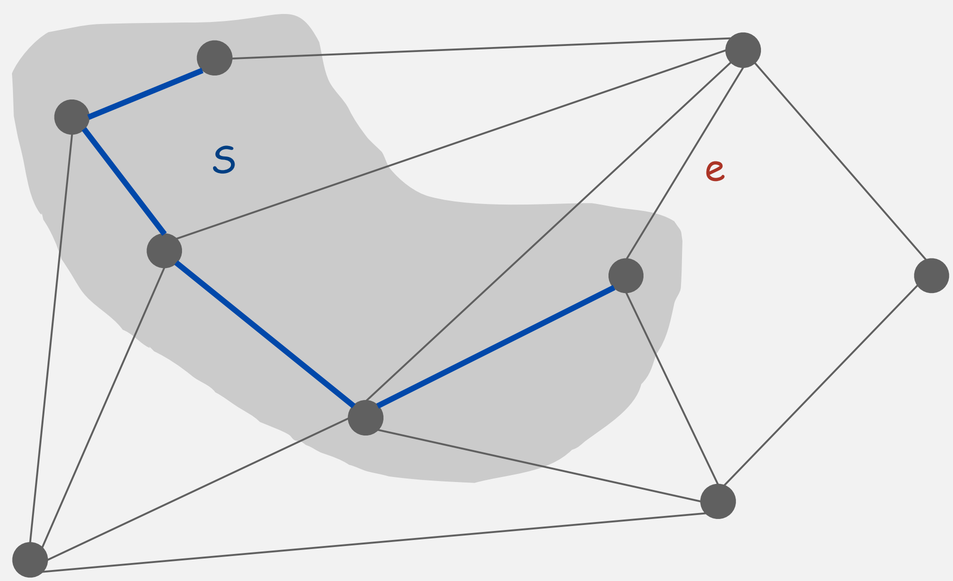


Prim implementation challenge

Problem. Find min weight edge with exactly one endpoint in S.

How difficult?

- O(E) time. ← try all edges
- O(V) time.
- O(log E) time. ← use a priority queue !
- O(log* E) time.
- Constant time.

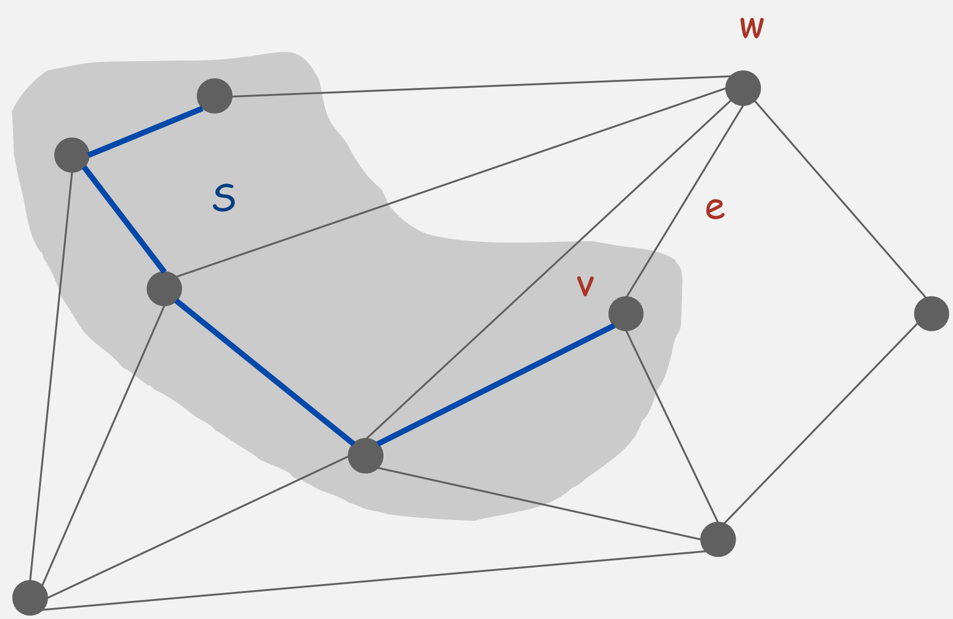


Prim's algorithm implementation (lazy)

Problem. Find min weight edge with exactly one endpoint in S.

Efficient solution. Maintain a PQ of edges with (at least) one endpoint in S.

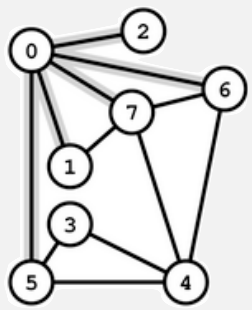
- Delete min to determine next edge e = v-w to add to T.
- Disregard if both v and w are in S.
- Let w be vertex not in S:
  - add to PQ any edge incident to w (assuming other endpoint not in S)
  - add w to S



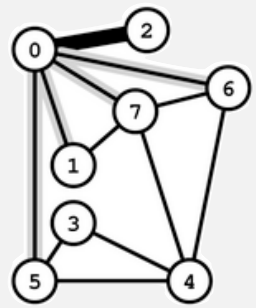
Prim's algorithm example: lazy implementation

Use PQ: key = edge.

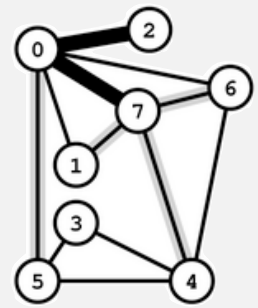
(lazy version leaves some obsolete entries on the PQ)



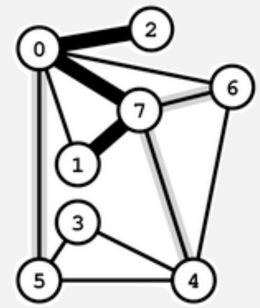
0-2 0-7 0-1
0-6 0-5



0-7 0-1 0-6 0-5

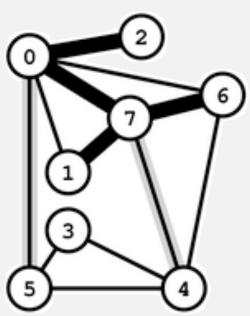


7-1 7-6 0-1
7-4 0-6 0-5

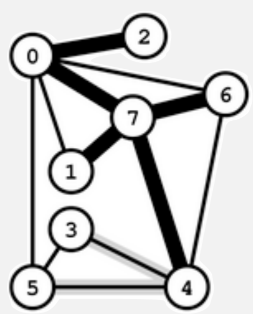


7-6 0-1 7-4
0-6 0-5

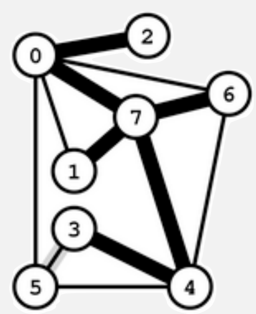
black = PQ edge with exactly one endpoint in S, sorted by weight
gray = PQ edge with both endpoints in S (obsolete)



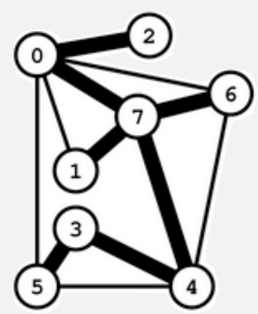
0-1 7-4 0-6
6-4 0-5



4-3 4-5 0-6
6-4 0-5



3-5 4-5 0-6
6-4 0-5



4-5 0-6 6-4 0-5

0-1	0.32
0-2	0.29
0-5	0.60
0-6	0.51
0-7	0.31
1-7	0.21
3-4	0.34
3-5	0.18
4-5	0.40
4-6	0.51
4-7	0.46
6-7	0.25

Lazy implementation of Prim's algorithm

```
public class LazyPrim
{
   private boolean[] scanned;     // vertices in MST
   private Queue<Edge> mst;       // edges in the MST
   private MinPQ<Edge> pq         // the priority queue of edges

    public LazyPrim(WeightedGraph G)
    {
        scanned = new boolean[G.V()];
        mst = new Queue<Edge>();
        pq = new MinPQ<Edge>(Edge.ByWeight());
        prim(G, 0);
    }

    public Iterable<Edge> mst()
    { return mst; }

    // See next slide for prim() implementation.
}
```

comparator by edge weight
(instead of by lexicographic order)

Lazy implementation of Prim's algorithm

```
private void scan(WeightedGraph G, int v)
{
   scanned[v] = true;
   for (Edge e : G.adj(v))
      if (!scanned[e.other(v)])
         pq.insert(e);
}
```

for each edge v-w, add to PQ if w not already in S

```
private void prim(WeightedGraph G, int s)
{
   scan(G, s);
   while (!pq.isEmpty())
   {
      Edge e = pq.delMin();
      int v = e.either(), w = e.other(v);
      if (scanned[v] && scanned[w]) continue;
      mst.enqueue(e);
      if (!scanned[v]) scan(G, v);
      if (!scanned[w]) scan(G, w);
   }
}
```

repeatedly delete the min weight edge v-w from PQ

ignore if both endpoints in S

add e to MST and scan v and w

Prim's algorithm running time

Proposition. Prim's algorithm computes MST in $O(E \log E)$ time.

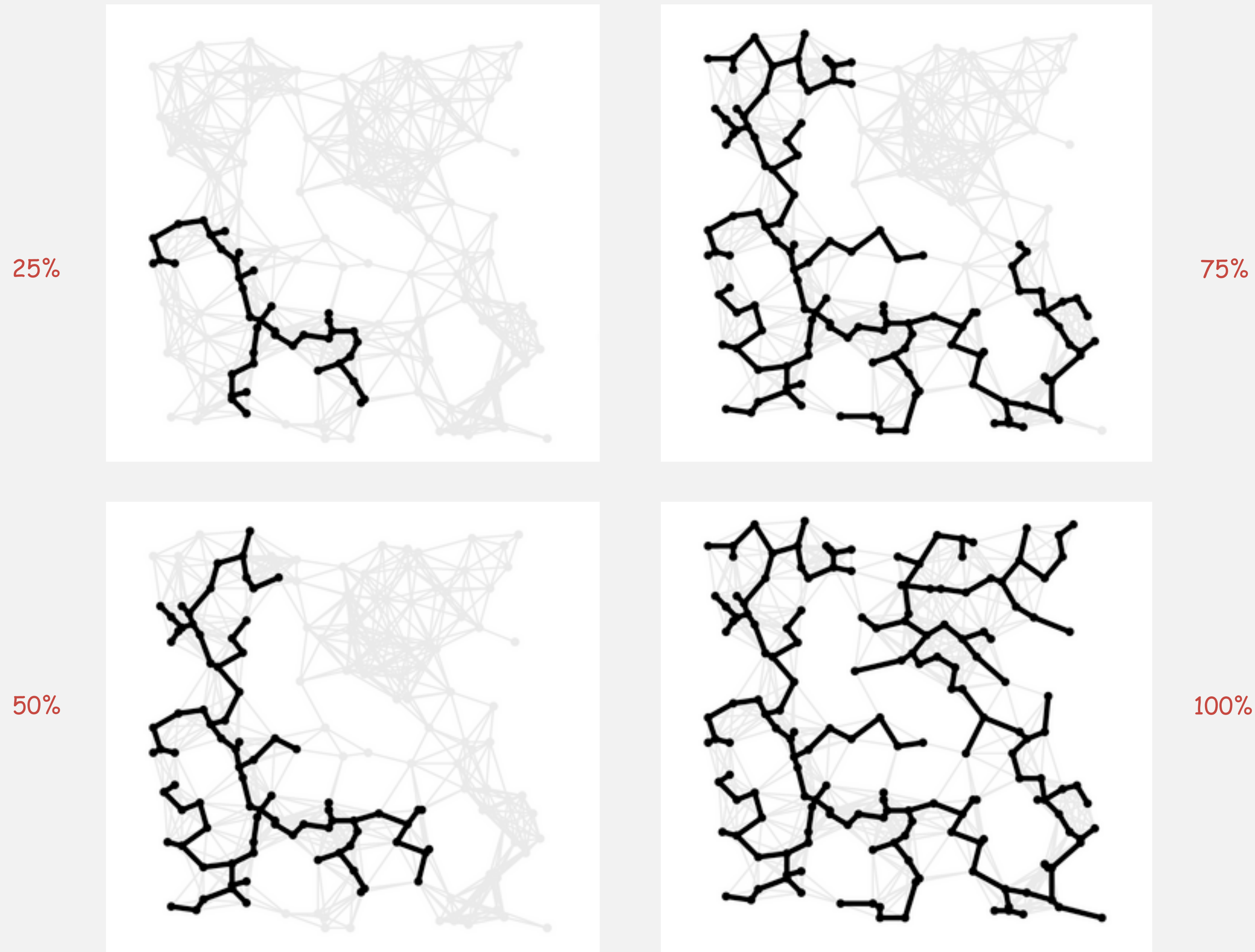
Pf.

operation	frequency	time per op
delete min	E	E log E
insert	E	E log E

Improvements.

- Stop when MST has V-1 edges.
- Eagerly eliminate obsolete edges from PQ.
- Maintain on PQ at most one edge incident to each vertex v not in T
  $\Rightarrow$ at most V edges on PQ.
- Use fancier priority queue: best in theory yields $O(E + V \log V)$.

Prim's algorithm example



Removing the distinct edge weight assumption

Simplifying assumption. All edge weights are distinct.

Approach 1. Introduce tie-breaking rule for `compare()` in `ByWeight`.

```
public int compare(Edge e, Edge f)
{
   if (e.weight < f.weight) return -1;
   if (e.weight > f.weight) return +1;
   if (e.v < f.v) return -1;
   if (e.v > f.v) return +1;
   if (e.w < f.w) return -1;
   if (e.w > f.w) return +1;
   return 0;
}
```

`return e.compareTo(f);`

Approach 2. Prim and Kruskal still find MST if equal weights!
(only our proof of correctness fails)

- weighted graph API
- cycles and cuts
- Kruskal's algorithm
- Prim's algorithm
- **advanced topics**

Does a linear-time MST algorithm exist?

deterministic compare-based MST algorithms

year	worst case	discovered by
1975	E log log V	Yao
1976	E log log V	Cheriton-Tarjan
1984	E log* V, E + V log V	Fredman-Tarjan
1986	E log (log* V)	Gabow-Galil-Spencer-Tarjan
1997	E α(V) log α(V)	Chazelle
2000	E α(V)	Chazelle
2002	optimal	Pettie-Ramachandran
20xx	E	???

Remark. Linear-time randomized MST algorithm (Karger-Klein-Tarjan 1995).

Euclidean MST

Given N points in the plane, find MST connecting them, where the distances between point pairs are their Euclidean distances.

Brute force. Compute ~ $N^2/2$ distances and run Prim's algorithm.

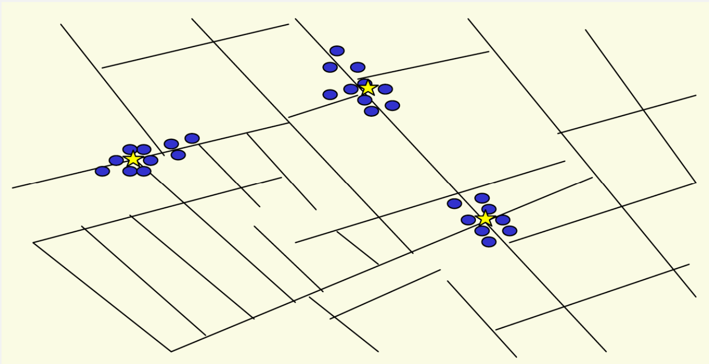
Ingenuity. Exploit geometry and do it in ~ $c N \lg N$.

Scientific application: clustering

k-clustering. Divide a set of objects classify into k coherent groups.

Distance function. Numeric value specifying "closeness" of two objects.

Goal. Divide into clusters so that objects in different clusters are far apart.



outbreak of cholera deaths in London in 1850s (Nina Mishra)

Applications.

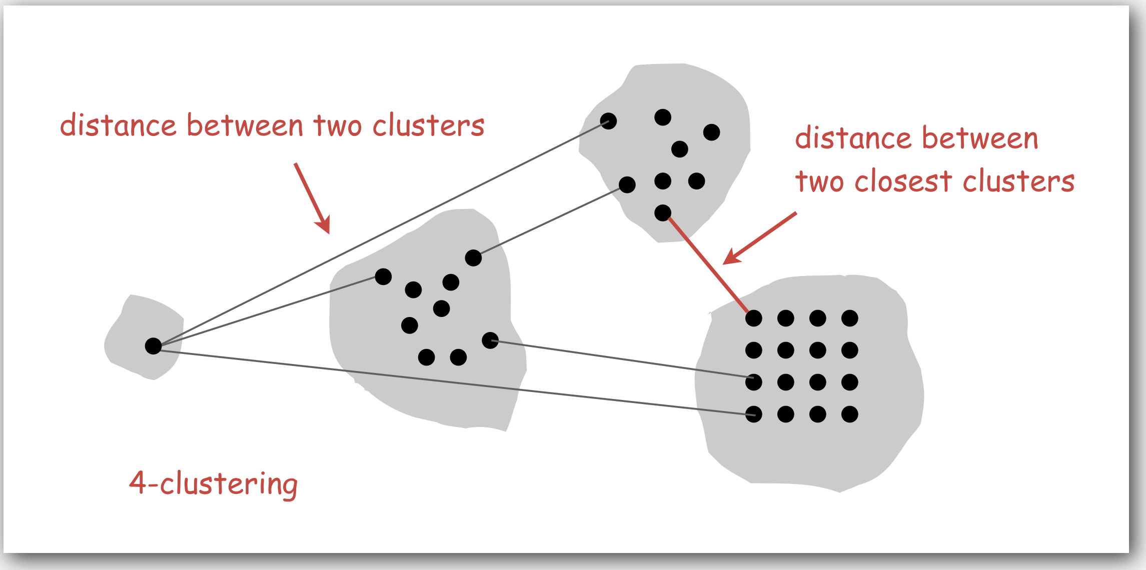
- Routing in mobile ad hoc networks.
- Document categorization for web search.
- Similarity searching in medical image databases.
- Skycat: cluster 10^9 sky objects into stars, quasars, galaxies.

Single-link clustering

k-clustering. Divide a set of objects classify into k coherent groups.
Distance function. Numeric value specifying "closeness" of two objects.

Single link. Distance between two clusters equals the distance between the two closest objects (one in each cluster).

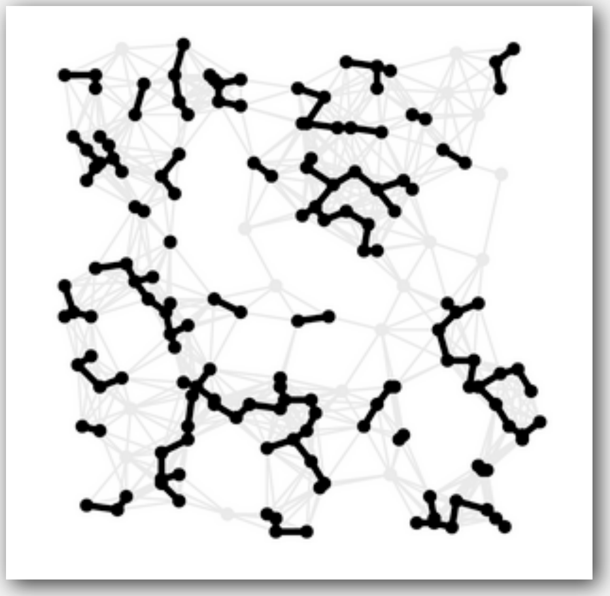
Single-link clustering. Given an integer k, find a k-clustering that maximizes the distance between two closest clusters.



Single-link clustering algorithm

"Well-known" algorithm for single-link clustering:

- Form V clusters of one object each.
- Find the closest pair of objects such that each object is in a different cluster, and merge the two clusters.
- Repeat until there are exactly k clusters.



Observation. This is Kruskal's algorithm (stop when k connected components).

Alternate solution. Run Prim's algorithm and delete k-1 max weight edges.

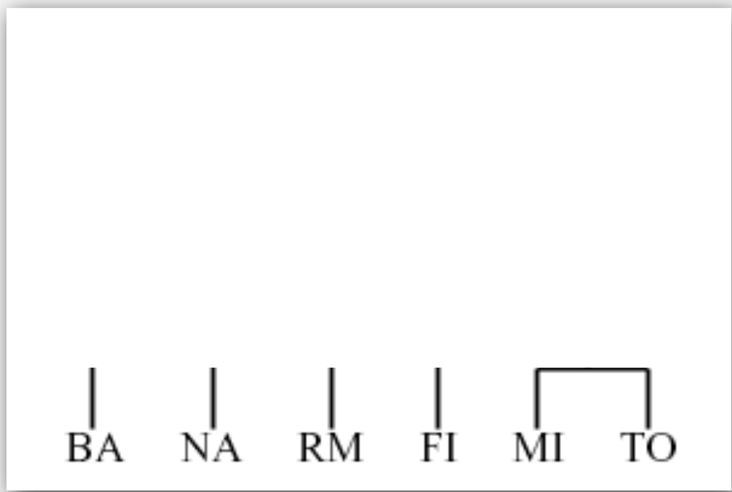
Dendrogram

Dendrogram. Tree diagram that illustrates arrangement of clusters.

http://home.dei.polimi.it/matteucc/Clustering/tutorial_html/hierarchical.html

Dendrogram

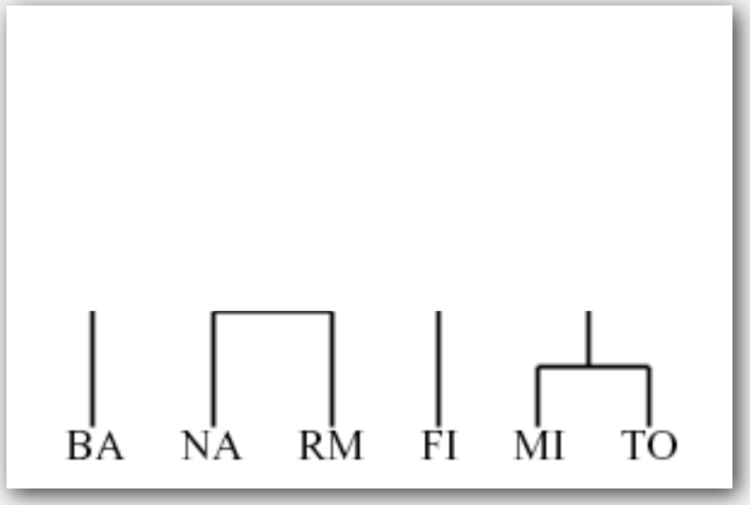
Dendrogram. Tree diagram that illustrates arrangement of clusters.



http://home.dei.polimi.it/matteucc/Clustering/tutorial_html/hierarchical.html

Dendrogram

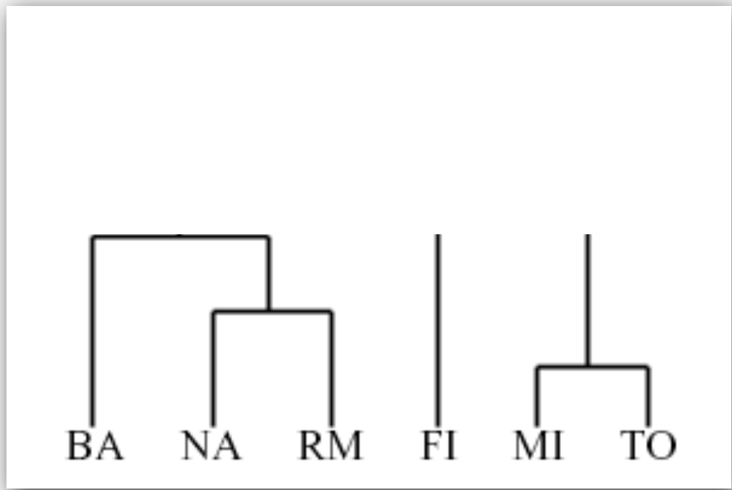
Dendrogram. Tree diagram that illustrates arrangement of clusters.



http://home.dei.polimi.it/matteucc/Clustering/tutorial_html/hierarchical.html

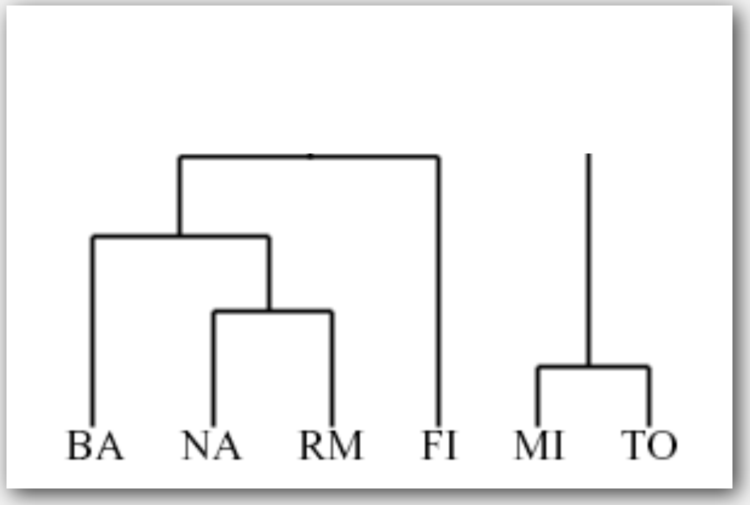
Dendrogram

Dendrogram. Tree diagram that illustrates arrangement of clusters.



http://home.dei.polimi.it/matteucc/Clustering/tutorial_html/hierarchical.html

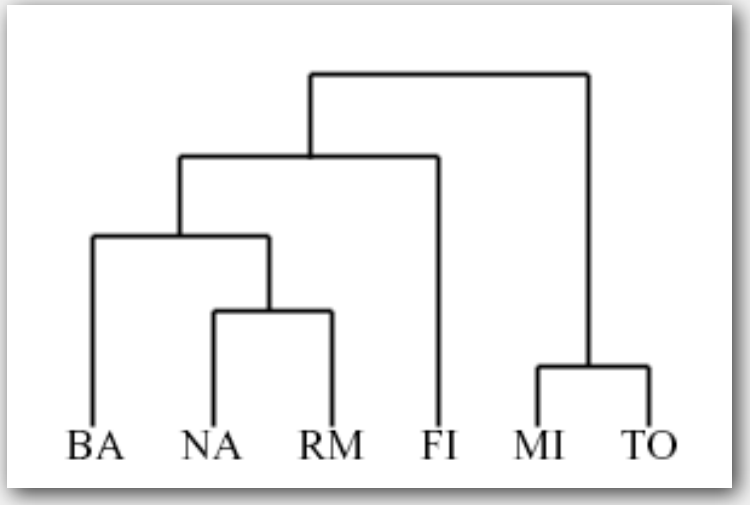
Dendrogram. Tree diagram that illustrates arrangement of clusters.



http://home.dei.polimi.it/matteucc/Clustering/tutorial_html/hierarchical.html

Dendrogram

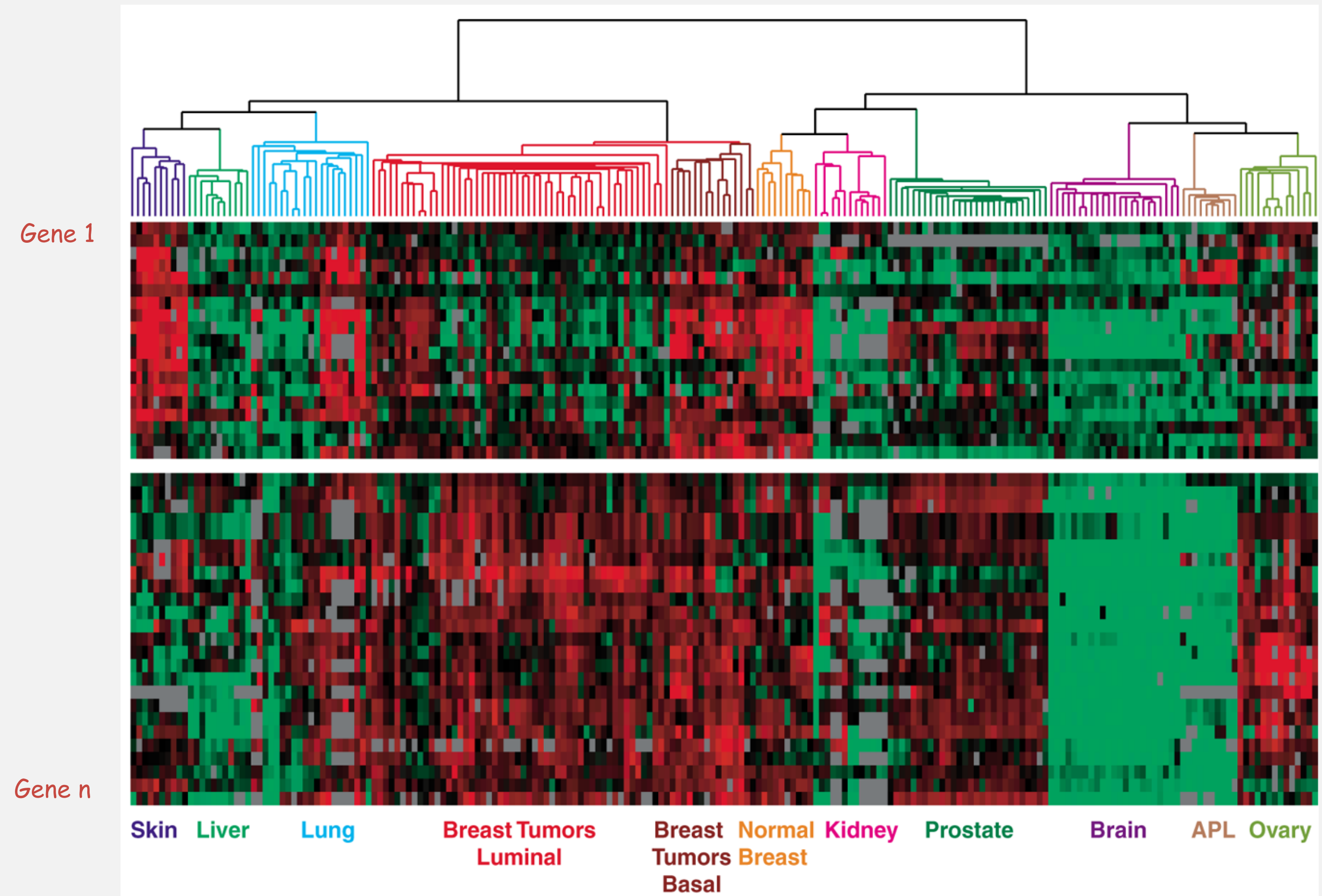
Dendrogram. Tree diagram that illustrates arrangement of clusters.



http://home.dei.polimi.it/matteucc/Clustering/tutorial_html/hierarchical.html

Dendrogram of cancers in human

Tumors in similar tissues cluster together.



Reference: Botstein & Brown group

gene expressed

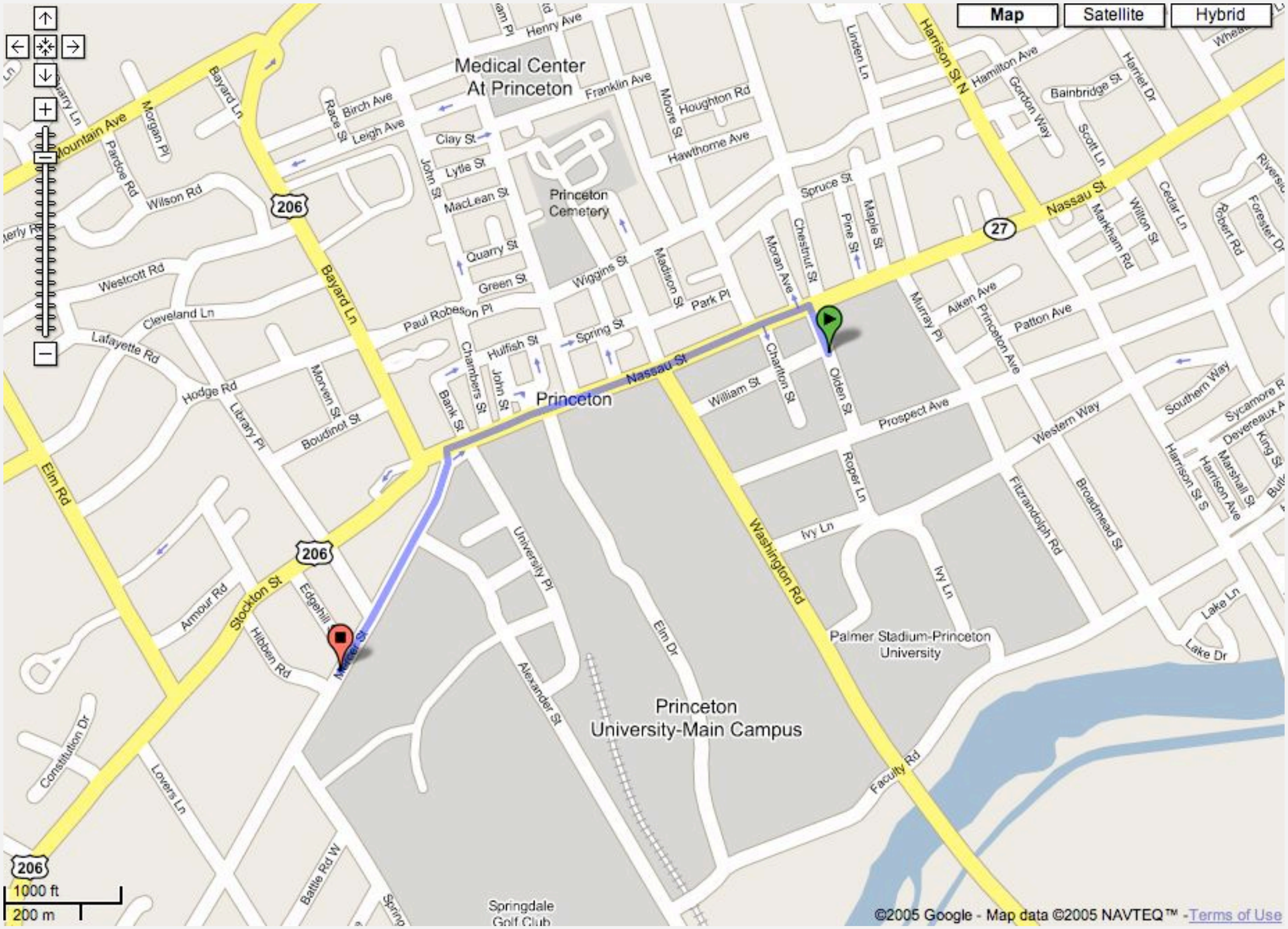
gene not expressed

4.4 Shortest Paths

- Dijkstra's algorithm
- implementation
- negative weights

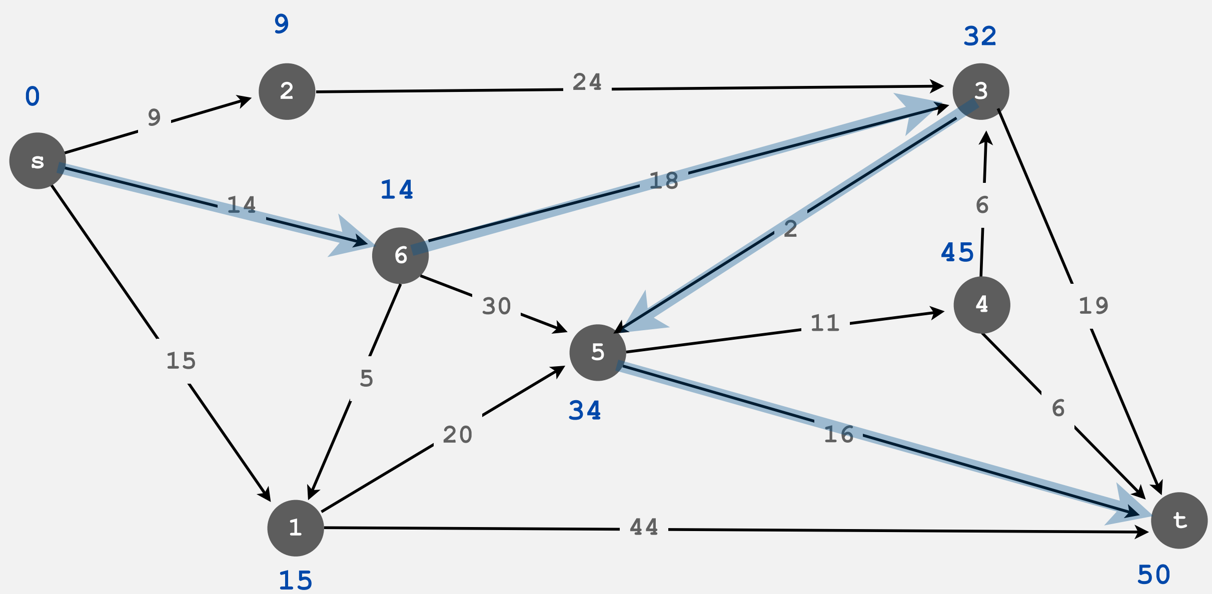
References: Algorithms in Java, 3rd edition, Chapter 21

Google maps


Map
Satellite
Hybrid
Medical Center At Princeton
Princeton Cemetery
Princeton
Nassau St
Bayard Ln
Washington Rd
Stockton St
Mercer St
Palmer Stadium-Princeton University
Princeton University-Main Campus
Springdale Golf Club
206
27
1000 ft
200 m
©2005 Google - Map data ©2005 NAVTEQ™ - Terms of Use

Shortest paths in a weighted digraph

Given a weighted digraph `G`, find the shortest directed path from `s` to `t`.



shortest path: s→6→3→5→t

cost: 14 + 18 + 2 + 16 = 50

Shortest path versions

Which vertices?

- From one vertex to another.
- From one vertex to every other.
- Between all pairs of vertices.

Restrictions on edge weights?

- Nonnegative weights.
- Arbitrary weights.
- Euclidean weights.

Early history of shortest paths algorithms

Shimbel (1955). Information networks.

Ford (1956). RAND, economics of transportation.

Leyzorek, Gray, Johnson, Ladew, Meaker, Petry, Seitz (1957).
Combat Development Dept. of the Army Electronic Proving Ground.

Dantzig (1958). Simplex method for linear programming.

Bellman (1958). Dynamic programming.

Moore (1959). Routing long-distance telephone calls for Bell Labs.

Dijkstra (1959). Simpler and faster version of Ford's algorithm.

Shortest path applications

- Maps.
- Robot navigation.
- Texture mapping.
- Typesetting in TeX.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Telemarketer operator scheduling.
- Subroutine in advanced algorithms.
- Routing of telecommunications messages.
- Approximating piecewise linear functions.
- Network routing protocols (OSPF, BGP, RIP).
- Exploiting arbitrage opportunities in currency exchange.
- Optimal truck routing through given traffic congestion pattern.

Reference: Network Flows: Theory, Algorithms, and Applications, R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, Prentice Hall, 1993.

- Dijkstra's algorithm
- implementation
- negative weights

Edsger W. Dijkstra: select quote

“ The question of whether computers can think is like the question of whether submarines can swim. ”

“ Do only what only you can do. ”

“ In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind. ”

“ The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence. ”

“ APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums. ”

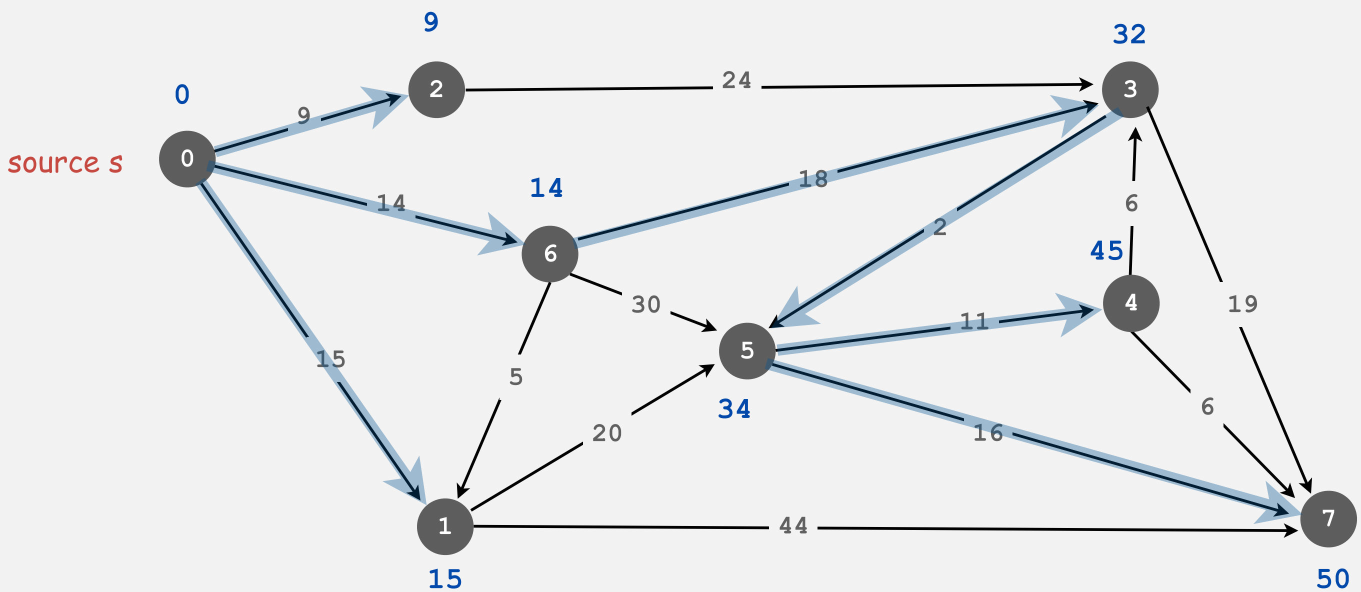
Edger Dijkstra
Turing award 1972

Single-source shortest-paths

Input. Weighted digraph G, source vertex s.

Goal. Find shortest path from s to every other vertex.

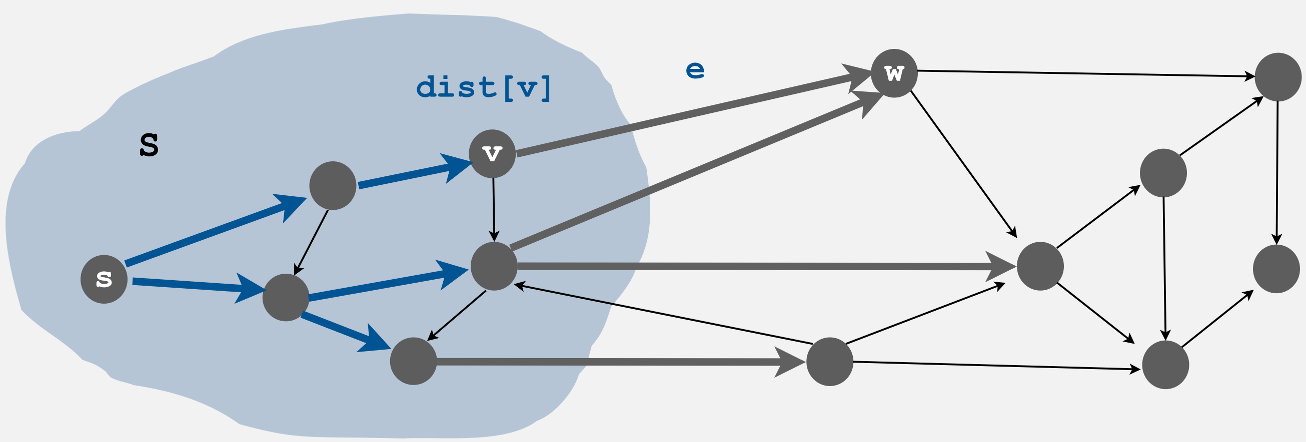
Observation. Use parent-link representation to store shortest path tree.



	0	1	2	3	4	5	6	7
dist[v]	0	15	9	32	45	34	14	50
pred[v]	–	0→1	0→2	6→3	5→4	3→5	0→6	5→7

Dijkstra's algorithm

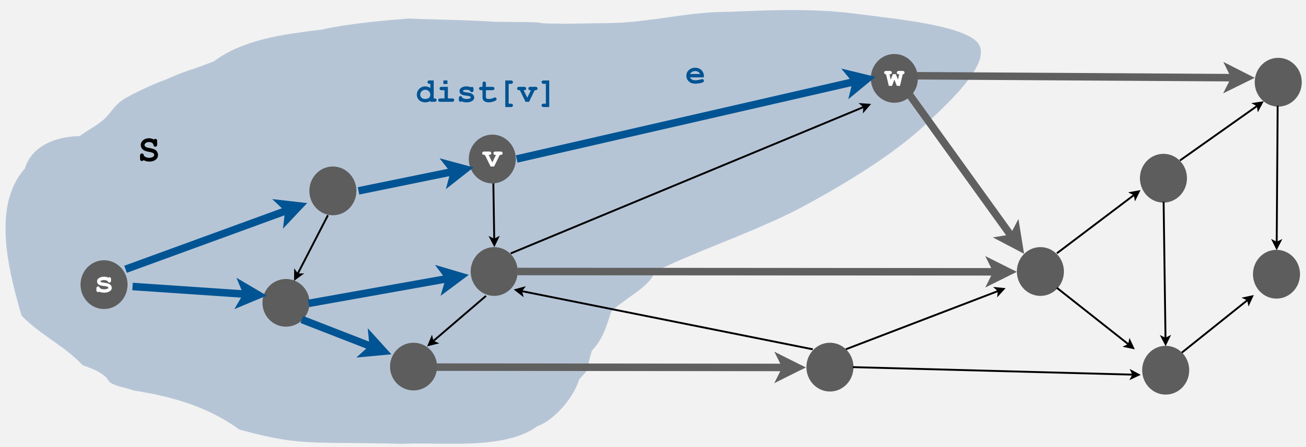
- Initialize S to `s`, `dist[s]` to `0`.
- Repeat until S contains all vertices connected to `s`:
  - find edge `e` with `v` in S and `w` not in S that minimizes `dist[v] + e.weight()`.



Dijkstra's algorithm

- Initialize S to `s`, `dist[s]` to `0`.
- Repeat until S contains all vertices connected to `s`:
  - find edge `e` with `v` in S and `w` not in S that minimizes `dist[v] + e.weight()`.
  - set `dist[w] = dist[v] + e.weight()` and `pred[w] = e`
  - add `w` to S

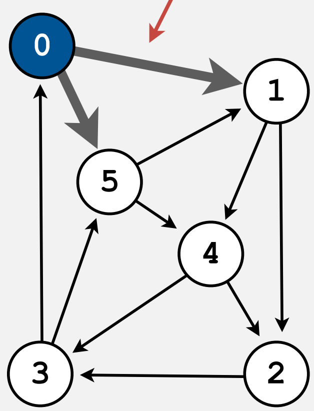
```
dist[w] = dist[v] + e.weight();
pred[w] = e;
```



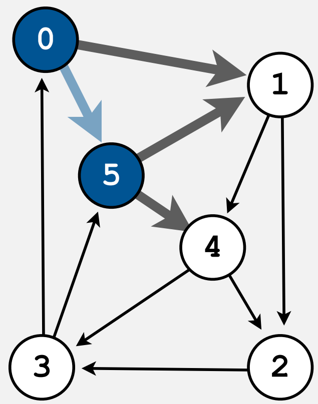
Dijkstra's algorithm example

edge with v in S and w not in S

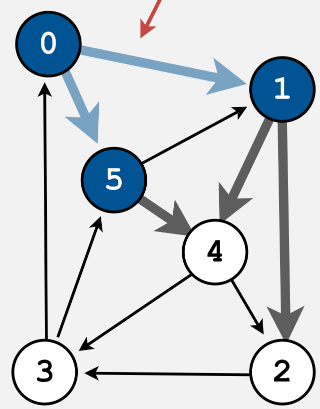
edge in shortest path tree



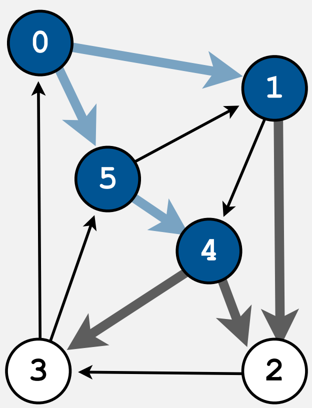
0→5 (.29)
0→1 (.41)



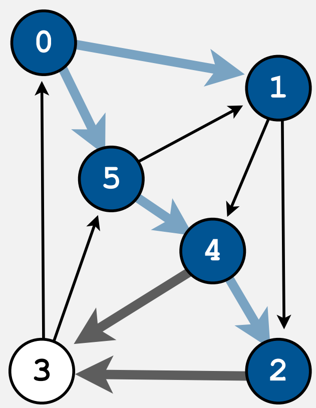
0→1 (.41)
5→4 (.50 = .29 + .21)
5→1 (.58 = .29 + .29)



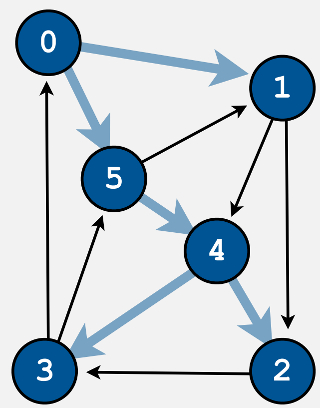
5→4 (.50)
1→4 (.73 = .41 + .32)
1→2 (.92 = .41 + .51)



4→2 (.82 = .50 + .32)
4→3 (.86 = .50 + .36)
1→2 (.92)



4→3 (0.86)
2→3 (1.32 = .82 + .50)



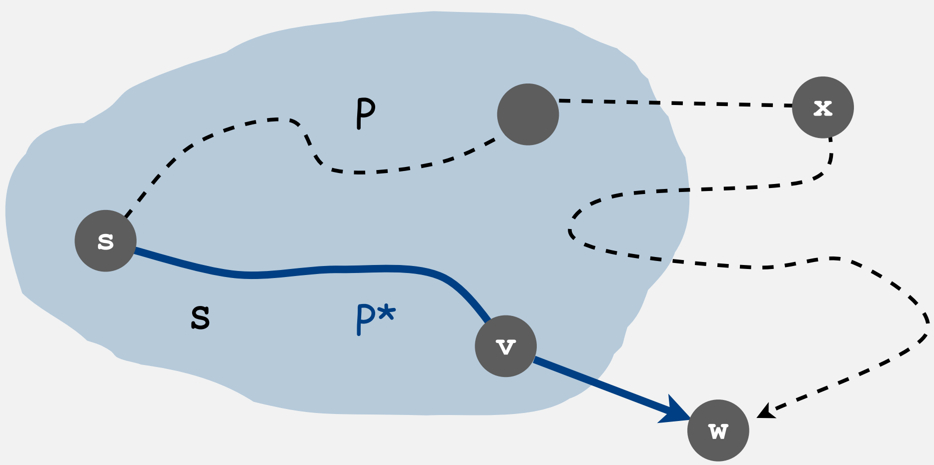
0→1	.41
0→5	.29
1→2	.51
1→4	.32
2→3	.50
3→0	.45
3→5	.38
4→2	.32
4→3	.36
5→1	.29
5→4	.21

Dijkstra's algorithm: correctness proof

Invariant. For `v` in S, `dist[v]` is the length of the shortest path from `s` to `v`.

Pf. (by induction on |S|)

- Let `w` be next vertex added to S.
- Let P* be the `s` → `w` path through `v`.
- Consider any other `s` → `w` path P, and let `x` be first node on path outside S.
- P is already as long as P* as soon as it reaches `x` by greedy choice.
- Thus, `dist[w]` is the length of the shortest path from `s` to `w`.



Shortest path trees

Remark. Dijkstra examines vertices in increasing distance from source.

25%

50%

75%

100%

- Dijkstra's algorithm
- **implementation**
- negative weights

Weighted directed graph API

public class DirectedEdge implements Comparable<DirectedEdge>		
	`DirectedEdge(int v, int w, double weight)`	*create a weighted edge v→w*
`int`	`from()`	*vertex v*
`int`	`to()`	*vertex w*
`double`	`weight()`	*the weight*

public class WeightedDigraph		*weighted digraph data type*
	`WeightedDigraph(int V)`	*create an empty digraph with V vertices*
	`WeightedDigraph(In in)`	*create a digraph from input stream*
`void`	`addEdge(DirectedEdge e)`	*add a weighted edge from v to w*
`Iterable<DirectedEdge>`	`adj(int v)`	*return an iterator over edges leaving v*
`int`	`V()`	*return number of vertices*

Weighted digraph: adjacency-set implementation in Java

```java
public class WeightedDigraph
{
   private final int V;
   private final SET<Edge>[] adj;

   public WeightedDigraph(int V)
   {
      this.V = V;
      adj = (SET<DirectedEdge>[]) new SET[V];
      for (int v = 0; v < V; v++)
         adj[v] = new SET<DirectedEdge>();
   }

   public void addEdge(DirectedEdge e)
   {
      int v = e.from();
      adj[v].add(e);
   }

   public Iterable<DirectedEdge> adj(int v)
   {  return adj[v];  }

   public int V()
   {  return V;  }
}
```

same as weighted undirected graph, but only add edge to v's adjacency set

Weighted directed edge: implementation in Java

```
public class DirectedEdge implements Comparable<DirectedEdge>
{
   private final int v, w;
   private final double weight;

   public DirectedEdge(int v, int w, double weight)
   {
      this.v = v;
      this.w = w;
      this.weight = weight;
   }

   public int from()    {  return v;        }
   public int to()      {  return w;        }
   public int weight()  {  return weight;   }

   public int compareTo(DirectedEdge that)
   {
      if (this.v < that.v) return -1;
      if (this.v > that.v) return +1;
      if (this.w < that.w) return -1;
      if (this.w > that.w) return +1;
      if (this.weight < that.weight) return -1;
      if (this.weight > that.weight) return +1;
      return 0;
    }
}
```

same as `Edge`, except `from()` and `to()` replace `either()` and `other()`

for use in a symbol table (allow parallel edges with different weights)

Shortest path data type

Design pattern.

- `Dijkstra` class is a `WeightedDigraph` client.
- Client query methods return distance and path iterator.

public class Dijkstra	
`Dijkstra(WeightedDigraph G, int s)`	*shortest path from s in graph G*
`double distanceTo(int v)`	*length of shortest path from s to v*
`Iterable <DirectedEdge> path(int v)`	*shortest path from s to v*

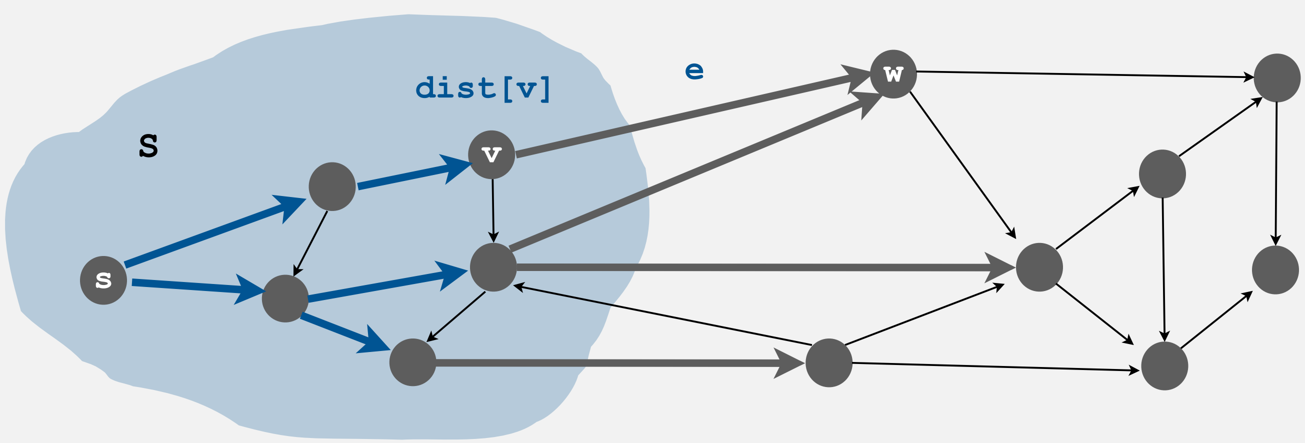
```
In in = new In("network.txt");
WeightedDigraph G = new WeightedDigraph(in);
int s = 0, t = G.V() - 1;
Dijktra dijkstra = new Dijkstra(G, s);
StdOut.println("distance = " + dijkstra.distanceTo(t));
for (DirectedEdge e : dijkstra.path(t))
    StdOut.println(e);
```

Dijkstra implementation challenge

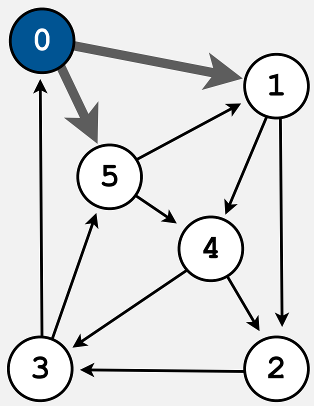
Find edge `e` with `v` in S and `w` not in S that minimizes `dist[v] + e.weight()`.

How difficult?

- Intractable.
- O(E) time. ← try all edges
- O(V) time.
- O(log E) time. ← Dijkstra with a binary heap
- O(log* E) time.
- Constant time.

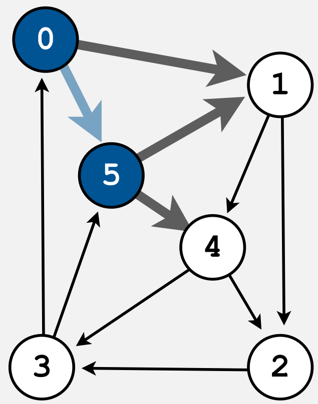


Lazy Dijkstra's algorithm example

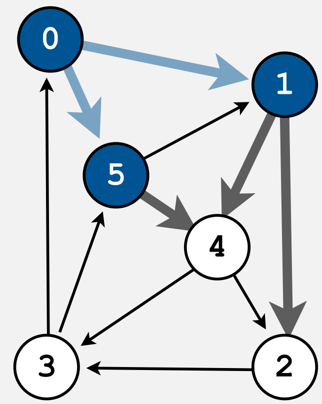


0→5 (.29)
0→1 (.41)

priority queue

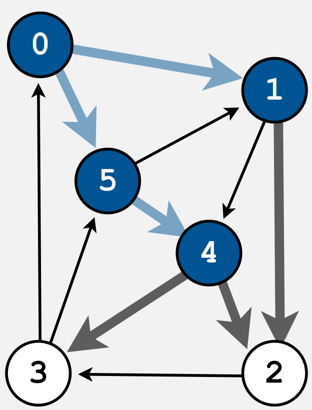


0→1 (.41)
5→4 (.50 = .29 + .21)
5→1 (.58 = .29 + .29)

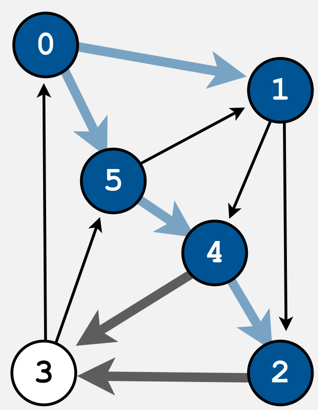


5→4 (.50)
1→4 (.73 = .41 + .32)
1→2 (.92 = .41 + .51)

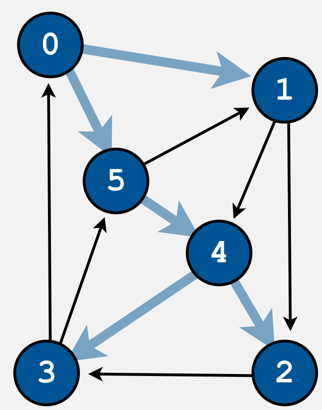
0→1	.41
0→5	.29
1→2	.51
1→4	.32
2→3	.50
3→0	.45
3→5	.38
4→2	.32
4→3	.36
5→1	.29
5→4	.21



1→4 (.73)
4→2 (.82 = .50 + .32)
4→3 (.86 = .50 + .36)
1→2 (.92)



4→3 (0.86)
1→2 (.92)
2→3 (1.32 = .82 + .50)



1→2 (.92)
2→3 (1.32)

Lazy implementation of Dijkstra's algorithm

```
public class LazyDijkstra
{
   private boolean[] scanned;
   private double[] dist;
   private DirectedEdge[] pred;
   private MinPQ<DirectedEdge> pq;

   private class ByDistanceFromSource implements Comparator<DirectedEdge>
   {
      public int compare(DirectedEdge e, DirectedEdge f) {
         double dist1 = dist[e.from()] + e.weight();
         double dist2 = dist[f.from()] + f.weight();
         if      (dist1 < dist2) return -1;
         else if (dist1 > dist2) return +1;
         else                    return  0;
      }
   }

   public LazyDijkstra(WeightedDigraph G, int s) {
      scanned = new boolean[G.V()];
      pred = new DirectedEdge[G.V()];
      dist = new double[G.V()];
      pq = new MinPQ<DirectedEdge>(new ByDistanceFromSource());
      dijkstra(G, s);
   }
```

compare edges in pq by `dist[v] + e.weight()`

Lazy implementation of Dijkstra's algorithm

```
private void dijkstra(WeightedDigraph G, int s)
{
   scan(G, s);
   while (!pq.isEmpty()) {
      DirectedEdge e = pq.delMin();
      int v = e.from(), w = e.to();
      if (scanned[w]) continue;
         pred[w] = e;
         dist[w] = dist[v] + e.weight();
         scan(G, w);
   }
}
```

both endpoints in S

found shortest path to w

```
private void scan(WeightedDigraph G, int v) {
   scanned[v] = true;
   for (DirectedEdge e : G.adj(v))
      if (!scanned[e.to()]) pq.insert(e);
}
```

add all edges v->w to pq, provided w not already in S

Dijkstra's algorithm running time

Proposition. Dijkstra's algorithm computes shortest paths in $O(E \log E)$ time.

Pf.

operation	frequency	time per op
delete min	E	log E
insert	E	log E

Improvements.

- Eagerly eliminate obsolete edges from PQ.
- Maintain on PQ at most one edge incident to each vertex v not in T
  $\Rightarrow$ at most V edges on PQ.
- Use fancier priority queue: best in theory yields $O(E + V \log V)$.

Priority-first search

Insight. All of our graph-search methods are the same algorithm!

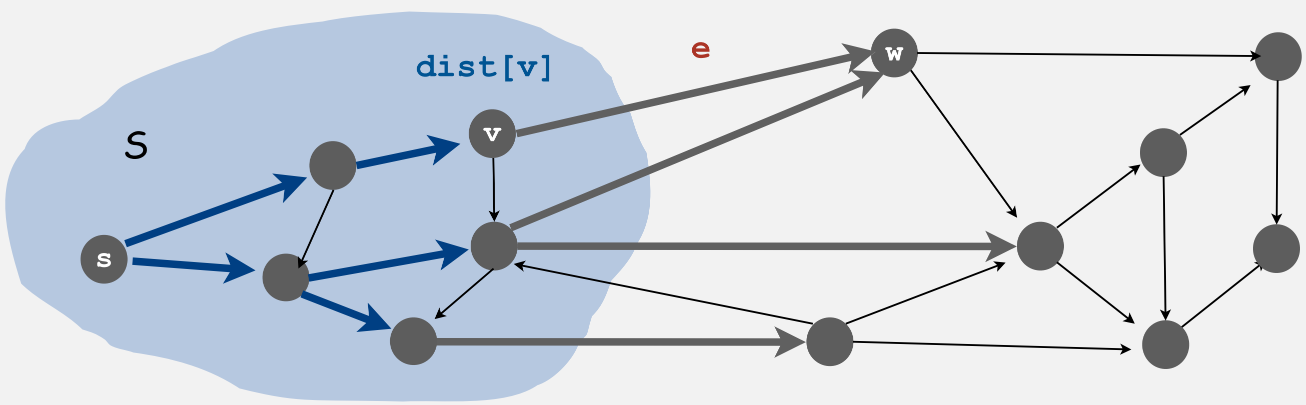
- Maintain a set of explored vertices S.
- Grow S by exploring edges with exactly one endpoint leaving S.

DFS. Take edge from vertex which was discovered most recently.

BFS. Take edge from vertex which was discovered least recently.

Prim. Take edge of minimum weight.

Dijkstra. Take edge to vertex that is closest to s.



Challenge. Express this insight in reusable Java code.

- Dijkstra's algorithm
- implementation
- **negative weights**

Currency conversion

Problem. Given currencies and exchange rates, what is best way to convert one ounce of gold to US dollars?

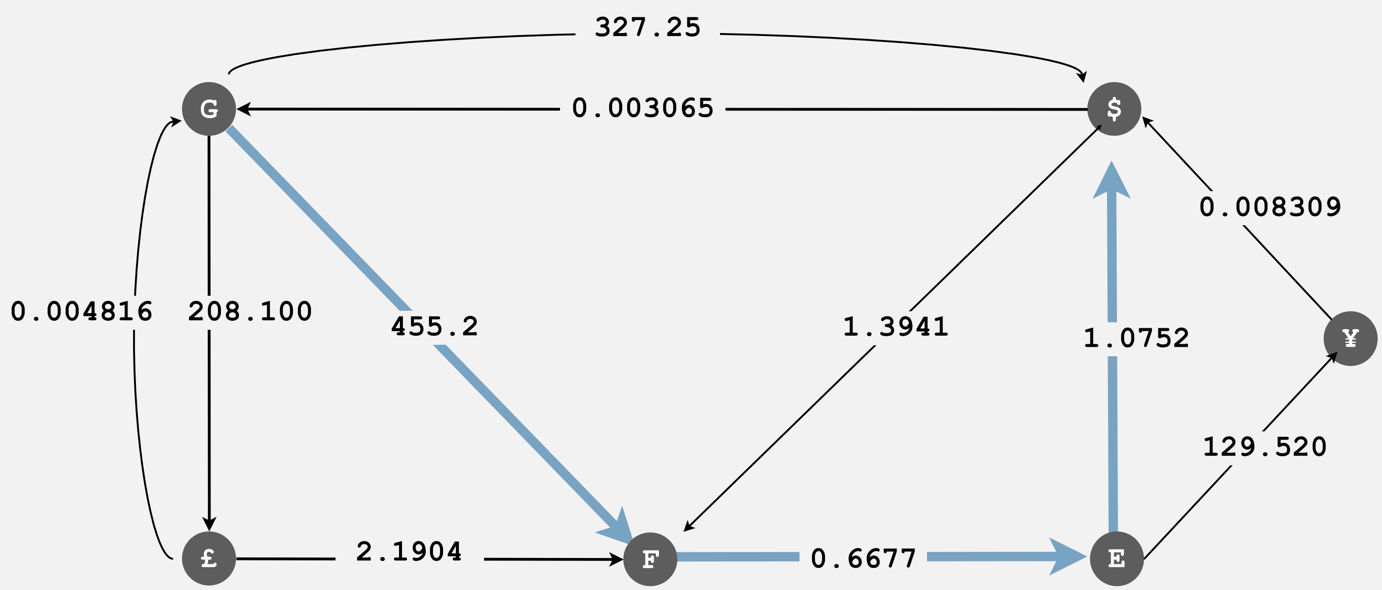
- 1 oz. gold ⇒ \$327.25.
- 1 oz. gold ⇒ £208.10 ⇒ \$327.00. [208.10 × 1.5714]
- 1 oz. gold ⇒ 455.2 Francs ⇒ 304.39 Euros ⇒ \$327.28. [455.2 × .6677 × 1.0752]

currency	£	Euro	¥	Franc	\$	Gold
UK pound	1.0000	0.6853	0.005290	0.4569	0.6368	208.100
Euro	1.45999	1.0000	0.007721	0.6677	0.9303	304.028
Japanese Yen	189.50	129.520	1.0000	85.4694	120.400	39346.7
Swiss Franc	2.1904	1.4978	0.01574	1.0000	1.3941	455.200
US dollar	1.5714	1.0752	0.008309	0.7182	1.0000	327.250
Gold (oz.)	0.004816	0.003295	0.0000255	0.002201	0.003065	1.0000

Currency conversion

Graph formulation.

- Vertex = currency.
- Edge = transaction, with weight equal to exchange rate.
- Find path that maximizes product of weights.

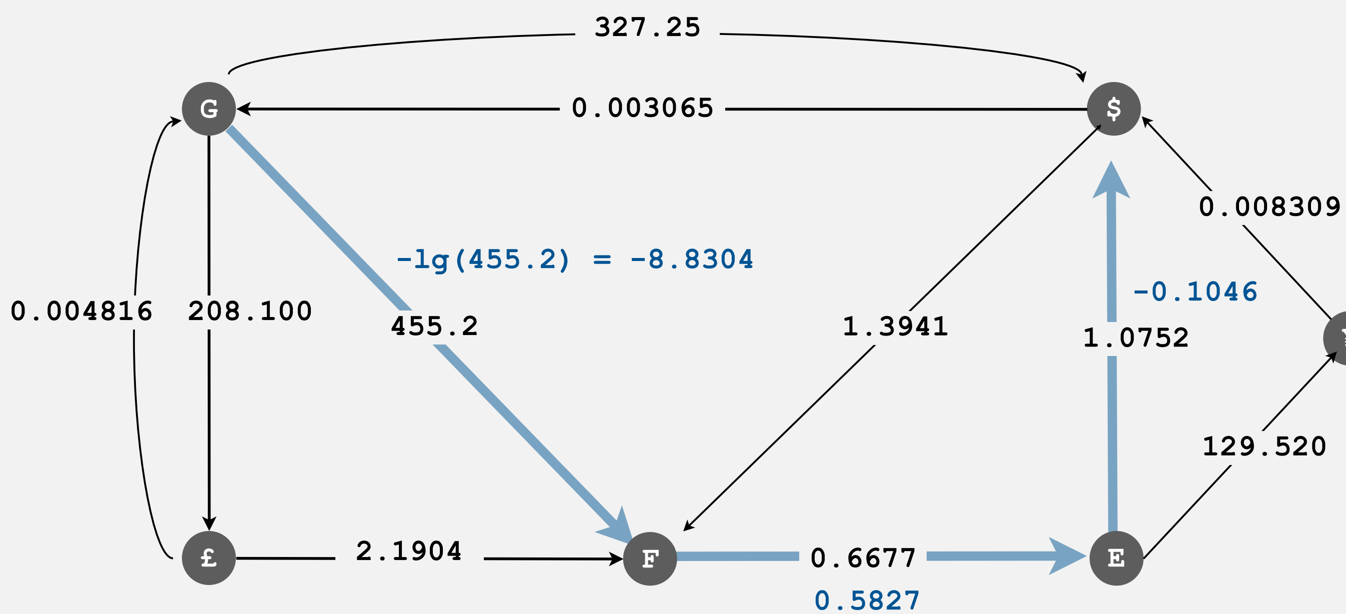


Challenge. Express as a shortest path problem.

Currency conversion

Reduce to shortest path problem by taking logs.

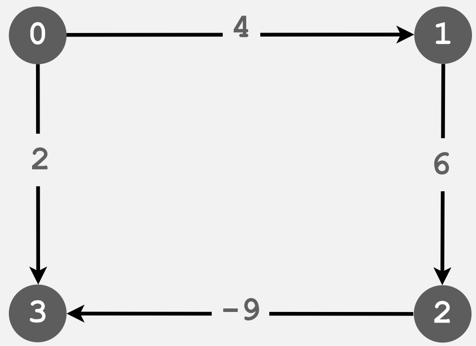
- Let weight of edge $v \to w$ be - lg (exchange rate from currency v to w).
- Multiplication turns to addition.
- Shortest path with given weights corresponds to best exchange sequence.



Challenge. Solve shortest path problem with negative weights.

Shortest paths with negative weights: failed attempts

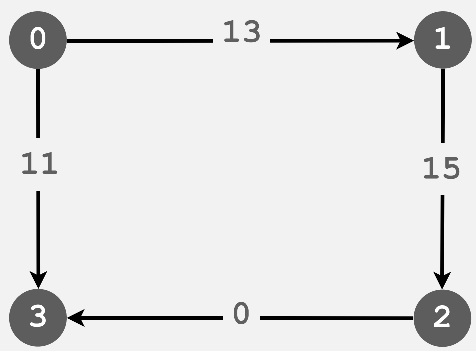
Dijkstra. Doesn't work with negative edge weights.



Dijkstra selects vertex 3 immediately after 0.

But shortest path from 0 to 3 is 0→1→2→3.

Re-weighting. Add a constant to every edge weight also doesn't work.

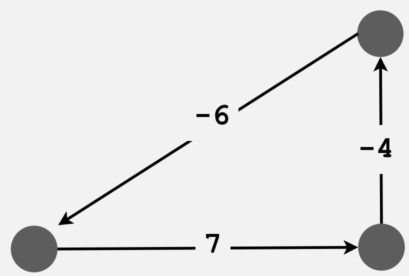


Adding 9 to each edge changes the shortest path because it adds 9 to each edge; wrong thing to do for paths with many edges.

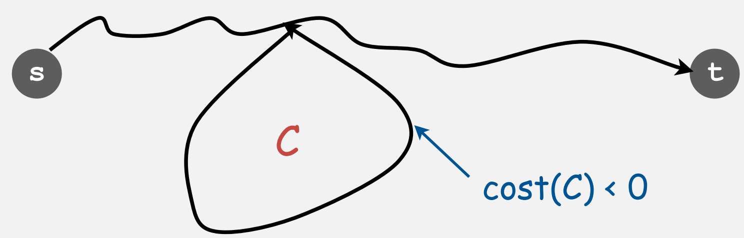
Bad news. Need a different algorithm.

Negative cycles

Def. A negative cycle is a directed cycle whose sum of edge weights is negative.



Observations. If negative cycle C is on a path from `s` to `t`, then shortest path can be made arbitrarily negative by spinning around cycle.

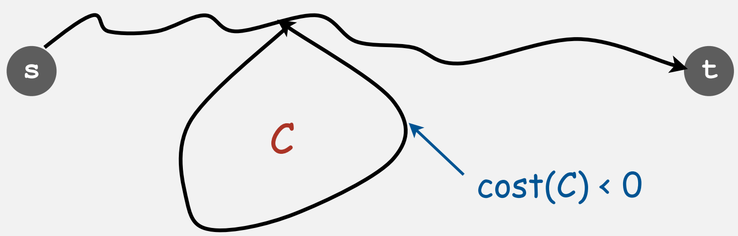


Worse news. Need a different problem.

Shortest paths with negative weights

Problem 1. Does a given digraph contain a negative cycle?

Problem 2. Find the shortest simple path from `s` to `t`.



Bad news. Problem 2 is intractable.

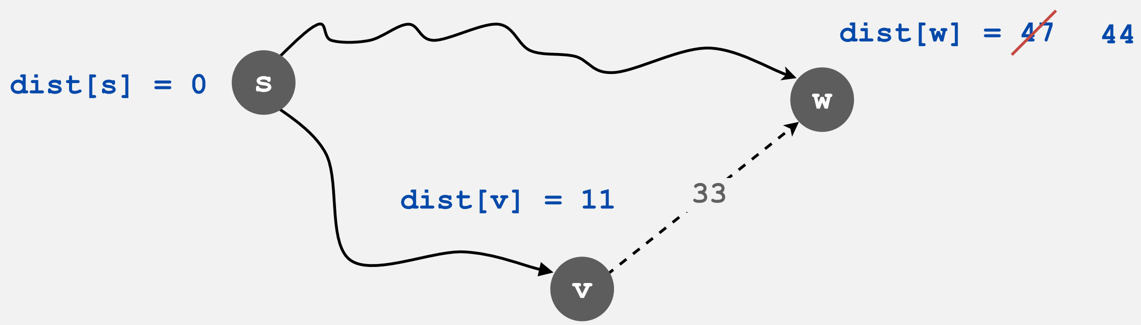
Good news. Can solve problem 1 in O(VE) steps;

if no negative cycles, can solve problem 2 with same algorithm!

Edge relaxation

Relax edge `e` from `v` to `w`.

- `dist[v]` is length of some path from `s` to `v`.
- `dist[w]` is length of some path from `s` to `w`.
- If `v→w` gives a shorter path to `w` through `v`, update `dist[w]` and `pred[w]`.




```
int v = e.from(), w = e.to();
if (dist[w] > dist[v] + e.weight())
{
    dist[w] = dist[v] + e.weight();
    pred[w] = e;
}
```

Shortest paths with negative weights: dynamic programming algorithm

A simple solution that works!

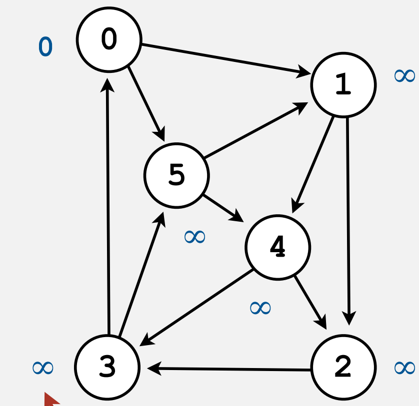
- Initialize `dist[v] = ∞`, `dist[s]= 0`.
- Repeat v times: relax each edge `e`.

```
for (int i = 1; i <= G.V(); i++)
   for (int v = 0; v < G.V(); v++)
      for (DirectedEdge e : G.adj(v))
      {
         int w = e.to();
         if (dist[w] > dist[v] + e.weight())
         {
             dist[w] = dist[v] + e.weight())
             pred[w] = e;
         }
      }
```

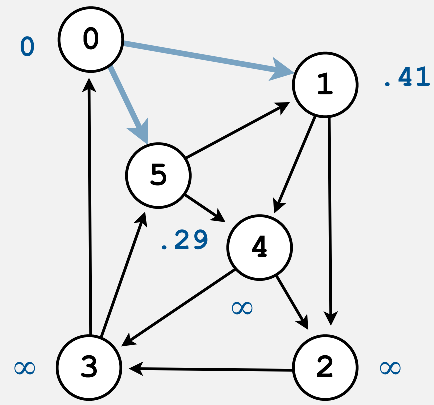
phase i

relax edge v-w

Dynamic programming algorithm trace

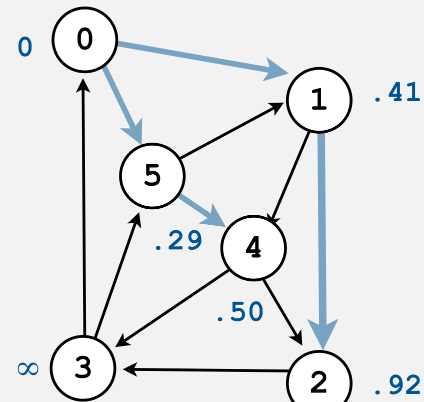


dist[v]



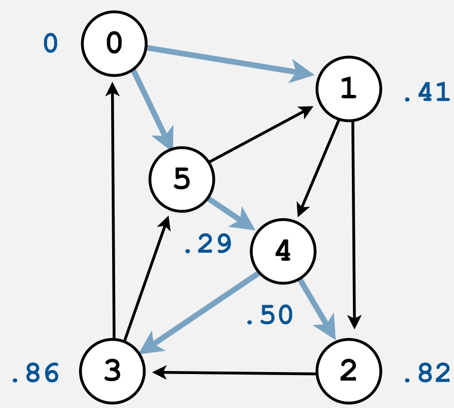
0→1 (.41 = 0 + .41)
0→5 (.50 = 0 + .50)

relaxed edges that update dist[]

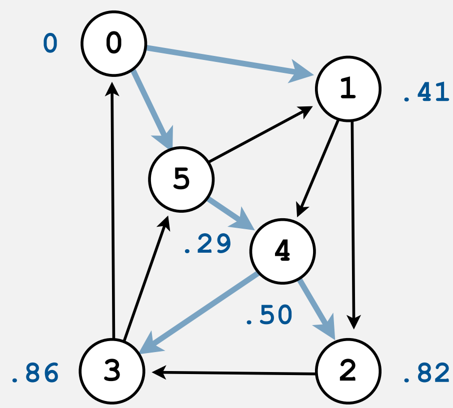


1→2 (.92 = .41 + .51)
1→4 (.73 = .41 + .32)
5→4 (.50 = .29 + .21)

0→1	.41
0→5	.29
1→2	.51
1→4	.32
2→3	.50
3→0	.45
3→5	.38
4→2	.32
4→3	.36
5→1	.29
5→4	.21



2→3 (1.33 = .83 + .50)
4→3 (.86 = .50 + .36)
4→2 (.82 = .50 + .32)



can stop early since
no entries in dist[] updated

Dynamic programming algorithm: analysis

Running time. Proportional to E V.

Invariant. At end of phase `i`, `dist[v]` ≤ length of any path from `s` to `v` using at most `i` edges.

Proposition. If there are no negative cycles, upon termination `dist[v]` is the length of the shortest path from from `s` to `v`.

and `pred[]` gives the shortest paths

Observation. If `dist[v]` doesn't change during phase `i`, no need to relax any edge leaving `v` in phase `i+1`.

FIFO implementation. Maintain queue of vertices whose distance changed.

be careful to keep at most one copy of each vertex on queue

Running time.

- Proportional to EV in worst case.
- Much faster than that in practice.

Single source shortest paths implementation: cost summary

	algorithm	worst case	typical case
nonnegative costs	Dijkstra (binary heap)	$E \log E$	E
no negative cycles	dynamic programming	$E V$	$E V$
	Bellman-Ford	$E V$	E

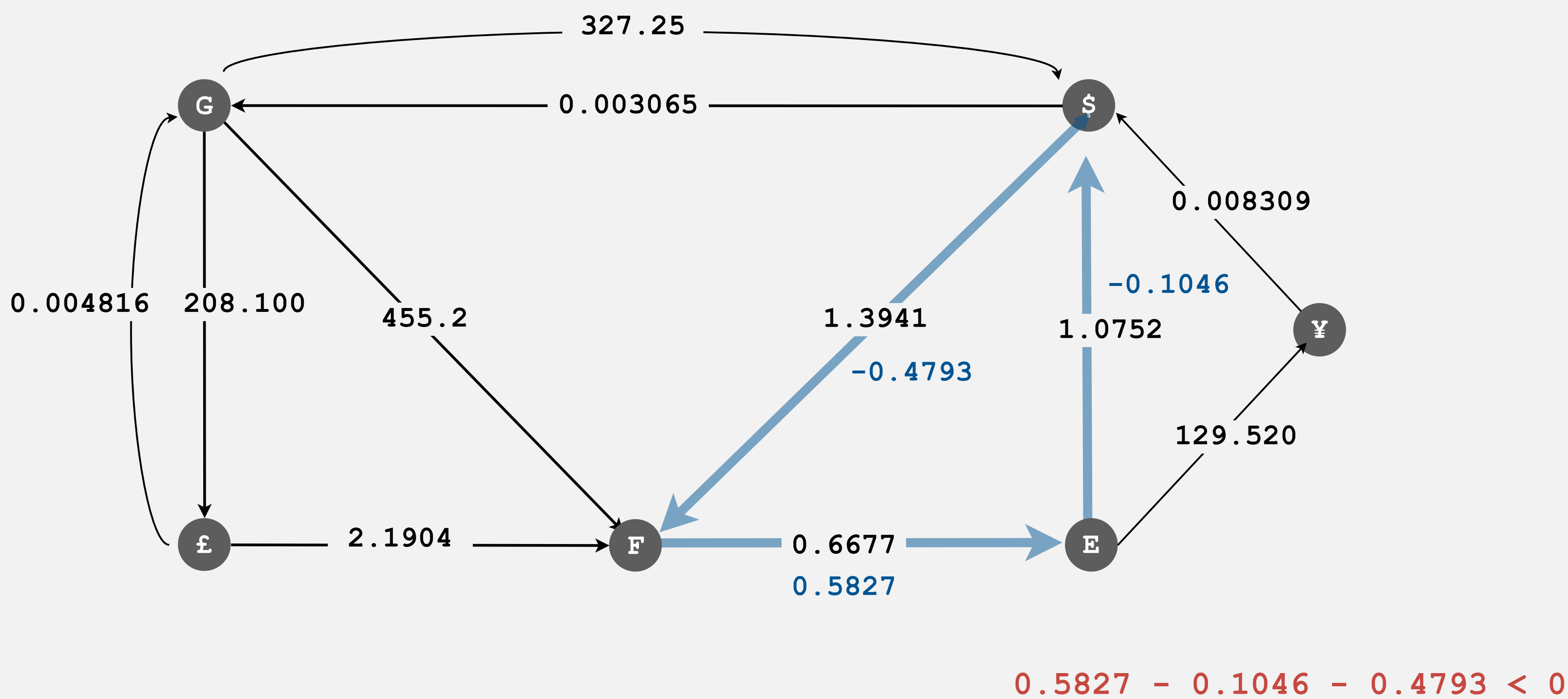
Remark 1. Negative weights makes the problem harder.

Remark 2. Negative cycles makes the problem intractable.

Shortest paths application: arbitrage

Is there an arbitrage opportunity in currency graph?

- Ex: \$1 $\Rightarrow$ 1.3941 Francs $\Rightarrow$ 0.9308 Euros $\Rightarrow$ \$1.00084.
- Is there a negative cost cycle?

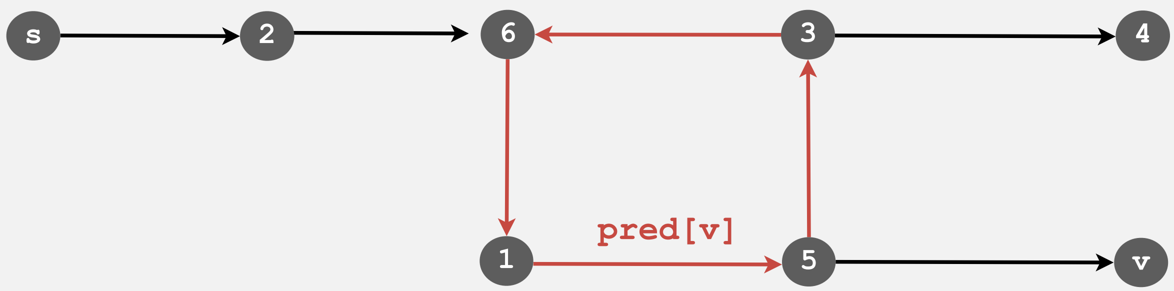


Remark. Fastest algorithm is valuable!

Negative cycle detection

If there is a negative cycle reachable from s.

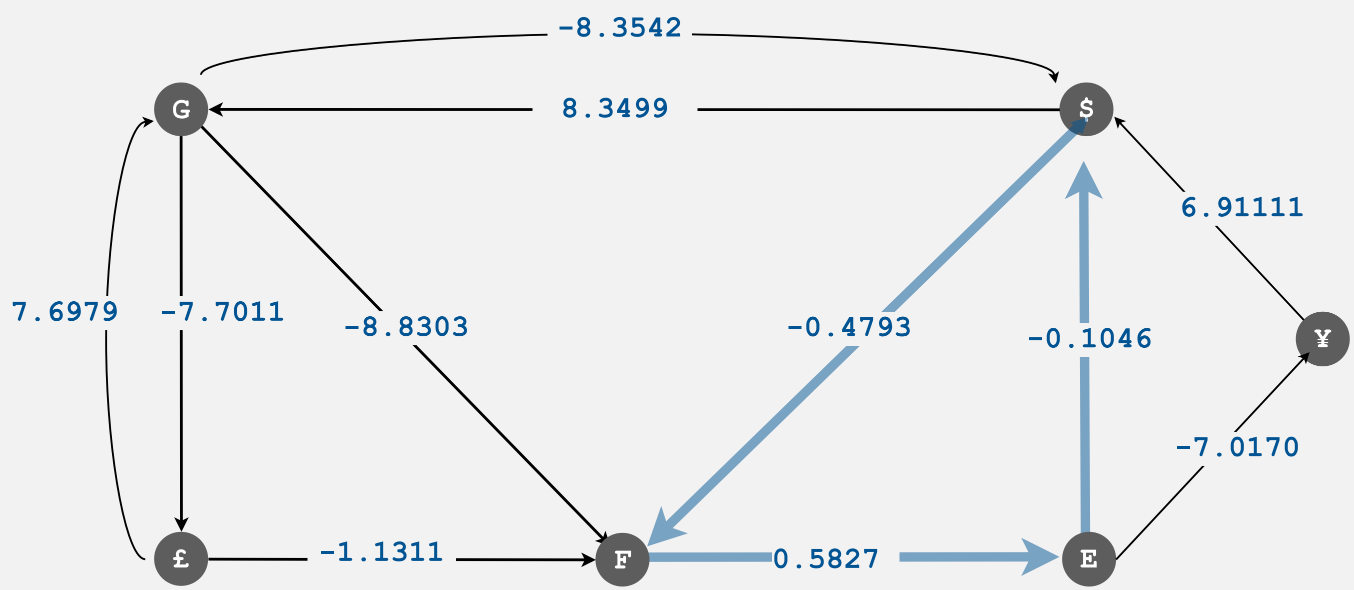
Bellman-Ford-Moore gets stuck in loop, updating vertices in cycle.



Proposition. If any vertex `v` is updated in phase v, there exists a negative cycle, and we can trace back `pred[v]` to find it.

Negative cycle detection

Goal. Identify a negative cycle (reachable from any vertex).



Solution. Initialize Bellman-Ford by setting `dist[v] = 0` for all vertices `v`.

Shortest paths summary

Dijkstra's algorithm.

- Nearly linear-time when weights are nonnegative.

Priority-first search.

- Generalization of Dijkstra's algorithm.
- Encompasses DFS, BFS, and Prim.
- Enables easy solution to many graph-processing problems.

Negative weights.

- Arise in applications.
- If negative cycles, problem is intractable (!)
- If no negative cycles, solvable via classic algorithms.

Shortest-paths is a broadly useful problem-solving model.

5. Strings

String processing

String. Sequence of characters.

Important fundamental abstraction.

- Java programs.
- Natural languages.
- Genomic sequences.
- …

> " *The digital information that underlies biochemistry, cell biology, and development can be represented by a simple string of G's, A's, T's and C's. This string is the root data structure of an organism's biology.* " — *M. V. Olson*

The char data type

C char data type. Typically an 8-bit integer.

- Supports 7-bit ASCII.
- Need more bits to represent certain characters.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	“	#	$	%	&	‘	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Hexadecimal to ASCII conversion table

Java char data type. A 16-bit unsigned integer.

- Supports original 16-bit Unicode.
- Awkwardly supports 21-bit Unicode 3.0.

The String data type

Character extraction. Get the i^{th} character.

Substring extraction. Get a contiguous sequence of characters from a string.

String concatenation. Append one character to end of another string.

s	t	r	i	n	g	s
0	1	2	3	4	5	6

```
String s = "strings";              // s = "strings"
char   c = s.charAt(2);            // c = 'r'
String t = s.substring(2, 6);      // t = "ring"
String u = t + c;                  // u = "ringr"
```

Implementing strings in Java

Java strings are immutable ⇒ two strings can share underlying `char[]` array.

```
public final class String implements Comparable<String>
{
   private char[] value;   // characters
   private int offset;     // index of first char in array
   private int count;      // length of string
   private int hash;       // cache of hashCode()

   private String(int offset, int count, char[] value)
   {
      this.offset = offset;
      this.count  = count;
      this.value  = value;
   }

   public String substring(int from, int to)
   {  return new String(offset + from, to - from, value);  }

   public char charAt(int index)
   {  return value[index + offset];  }
   …
}
```

java.lang.String

constant time

Implementing strings in Java

```
public String concat(String that)
{
   char[] buffer = new char[this.length() + that.length()];
   for (int i = 0; i < this.length(); i++)
       buffer[i] = this.value[i];
   for (int j = 0; j < that.length(); j++)
       buffer[this.length() + j] = that.value[j];
   return new String(0, this.length() + that.length(), buffer);
}
```

Memory. 40 + 2N bytes for a virgin `String` of length N.

use `byte[]` or `char[]` instead of `String` to save space

operation	guarantee	extra space
`charAt()`	1	1
`substring()`	1	1
`concat()`	N	N

String vs. StringBuilder

String. [immutable] Constant substring, linear concatenation.

StringBuilder. [mutable] Linear substring, constant (amortized) append.

Ex. Reverse a String.

```
public static String reverse(String s)
{
   String rev = "";
   for (int i = s.length() - 1; i >= 0; i--)
      rev += s.charAt(i);
   return rev;
}
```

quadratic time

```
public static String reverse(String s)
{
   StringBuilder rev = new StringBuilder();
   for (int i = s.length() - 1; i >= 0; i--)
      rev.append(s.charAt(i));
   return rev.toString();
}
```

linear time

String challenge: array of suffixes

Challenge. How to efficiently form array of suffixes?

input string

a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

suffixes

0	a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
1	a	c	a	a	g	t	t	t	a	c	a	a	g	c	
2	c	a	a	g	t	t	t	a	c	a	a	g	c		
3	a	a	g	t	t	t	a	c	a	a	g	c			
4	a	g	t	t	t	a	c	a	a	g	c				
5	g	t	t	t	a	c	a	a	g	c					
6	t	t	t	a	c	a	a	g	c						
7	t	t	a	c	a	a	g	c							
8	t	a	c	a	a	g	c								
9	a	c	a	a	g	c									
10	c	a	a	g	c										
11	a	a	g	c											
12	a	g	c												
13	g	c													
14	c														

String challenge: array of suffixes

Challenge. How to efficiently form array of suffixes?

A.

```
public static String[] suffixes(String s)
{
   int N = s.length();
   String[] suffixes = new String[N];
   for (int i = 0; i < N; i++)
      suffixes[i] = s.substring(i, N);
   return suffixes;
}
```

linear time and space

B.

```
public static String[] suffixes(String s)
{
   int N = s.length();
   StringBuilder sb = new StringBuilder(s);
   String[] suffixes = new String[N];
   for (int i = 0; i < N; i++)
      suffixes[i] = sb.substring(i, N);
   return suffixes;
}
```

quadratic time and space!

Alphabets

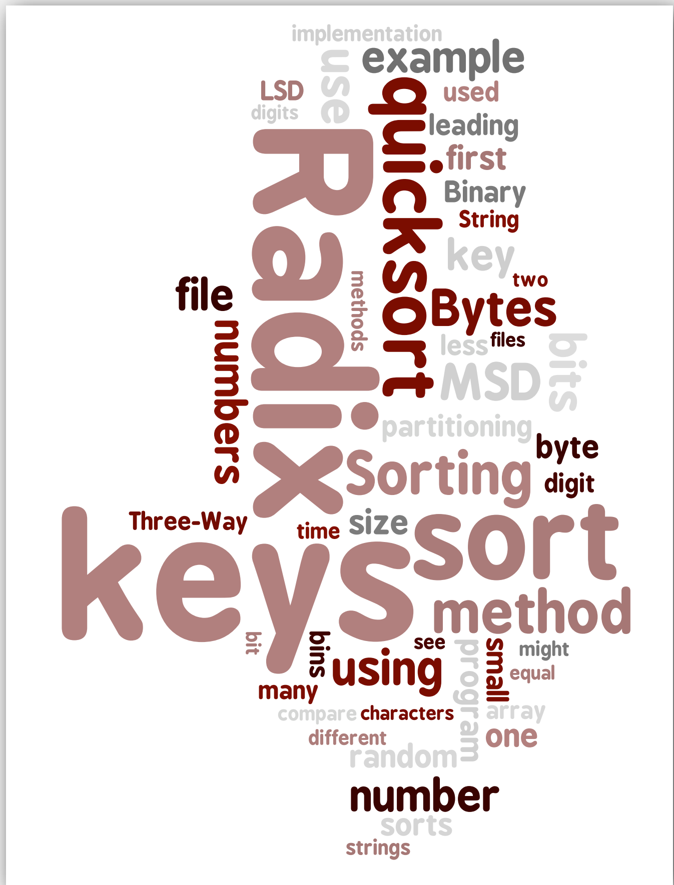
Digital key. Sequence of digits over fixed alphabet.

Radix. Number of digits R in alphabet.

name	R()	lgR()	characters
BINARY	2	1	01
OCTAL	8	3	01234567
DECIMAL	10	4	0123456789
HEXADECIMAL	16	4	0123456789ABCDEF
DNA	4	2	ACTG
LOWERCASE	26	5	abcdefghijklmnopqrstuvwxyz
UPPERCASE	26	5	ABCDEFGHIJKLMNOPQRSTUVWXYZ
PROTEIN	20	5	ACDEFGHIKLMNPQRSTVWY
BASE64	64	6	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdef ghijklmnopqrstuvwxyz0123456789+/
ASCII	128	7	*ASCII characters*
EXTENDED_ASCII	256	8	*extended ASCII characters*
UNICODE16	65536	16	*Unicode characters*

Standard alphabets

6.1 Sorting Strings



- key-indexed counting
- LSD string sort
- MSD string sort
- 3-way string quicksort
- suffix arrays

Review: summary of the performance of sorting algorithms

Frequency of operations = key compares.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$N^2/2$	$N^2/4$	no	yes	`compareTo()`
mergesort	N lg N	N lg N	N	yes	`compareTo()`
quicksort	1.39 N lg N *	1.39 N lg N	c lg N	no	`compareTo()`
heapsort	2 N lg N	2 N lg N	no	no	`compareTo()`

* probabilistic

Lower bound. ~ N lg N compares are required by any compare-based algorithm.

Q. Can we do better (despite the lower bound)?

A. Yes, if we don't depend on compares.

- key-indexed counting
- LSD string sort
- MSD string sort
- 3-way radix quicksort
- longest repeated substring

Key-indexed counting: assumptions about keys

Assumption. Keys are integers between 0 and R-1.

Implication. Can use key as an array index.

Applications.

- Sort string by first letter.
- Sort class roster by section.
- Sort phone numbers by area code.
- Subroutine in a sorting algorithm.

Remark. Keys may have associated data ⇒ can't just count up number of keys of each value.

input name	section	sorted result (by section)	
Anderson	2	Harris	1
Brown	3	Martin	1
Davis	3	Moore	1
Garcia	4	Anderson	2
Harris	1	Martinez	2
Jackson	3	Miller	2
Johnson	4	Robinson	2
Jones	3	White	2
Martin	1	Brown	3
Martinez	2	Davis	3
Miller	2	Jackson	3
Moore	1	Jones	3
Robinson	2	Taylor	3
Smith	4	Williams	3
Taylor	3	Garcia	4
Thomas	4	Johnson	4
Thompson	4	Smith	4
White	2	Thomas	4
Williams	3	Thompson	4
Wilson	4	Wilson	4

keys are small integers

Key-indexed counting

Goal. Sort an array `a[]` of `N` integers between `0` and `R-1`.

- Count frequencies of each letter using key as index.

count frequencies

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

offset by 1 [stay tuned]

r	count[r]
a	0
b	2
c	3
d	1
e	2
f	1
-	3

Key-indexed counting

Goal. Sort an array `a[]` of `N` integers between `0` and `R-1`.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
-
-

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

compute cumulates

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	0
b	2
c	5
d	6
e	8
f	9
-	12

6 keys < d, 8 keys < e
so d's go in `a[6]` and `a[7]`

Key-indexed counting

Goal. Sort an array `a[]` of `N` integers between `0` and `R-1`.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move records.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move records

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	0
b	2
c	5
d	6
e	8
f	9
-	12

i	aux[i]
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	

Key-indexed counting

Goal. Sort an array `a[]` of `N` integers between `0` and `R-1`.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move records.
-

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move records

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	0
b	2
c	5
d	7
e	8
f	9
-	12

i	aux[i]
0	
1	
2	
3	
4	
5	
6	d
7	
8	
9	
10	
11	

Key-indexed counting

Goal. Sort an array `a[]` of `N` integers between `0` and `R-1`.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move records.
-

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move records

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	1
b	2
c	5
d	7
e	8
f	9
-	12

i	aux[i]
0	a
1	
2	
3	
4	
5	
6	d
7	
8	
9	
10	
11	

Key-indexed counting

Goal. Sort an array `a[]` of `N` integers between `0` and `R-1`.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move records.
-

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move records

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	1
b	2
c	6
d	7
e	8
f	9
-	12

i	aux[i]
0	a
1	
2	
3	
4	
5	c
6	d
7	
8	
9	
10	
11	

Key-indexed counting

Goal. Sort an array `a[]` of `N` integers between `0` and `R-1`.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move records.
-

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move records

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	1
b	2
c	6
d	7
e	8
f	10
-	12

i	aux[i]
0	a
1	
2	
3	
4	
5	c
6	d
7	
8	
9	f
10	
11	

Key-indexed counting

Goal. Sort an array `a[]` of `N` integers between `0` and `R-1`.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move records.
-

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move records

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	1
b	2
c	6
d	7
e	8
f	11
-	12

i	aux[i]
0	a
1	
2	
3	
4	
5	c
6	d
7	
8	
9	f
10	f
11	

Key-indexed counting

Goal. Sort an array `a[]` of `N` integers between `0` and `R-1`.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move records.
-

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move records

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	1
b	3
c	6
d	7
e	8
f	11
-	12

i	aux[i]
0	a
1	
2	b
3	
4	
5	c
6	d
7	
8	
9	f
10	f
11	

Key-indexed counting

Goal. Sort an array `a[]` of `N` integers between `0` and `R-1`.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move records.
-

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move records

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	1
b	3
c	6
d	8
e	8
f	11
-	12

i	aux[i]
0	a
1	
2	b
3	
4	
5	c
6	d
7	d
8	
9	f
10	f
11	

Key-indexed counting

Goal. Sort an array `a[]` of `N` integers between `0` and `R-1`.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move records.
-

move records

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	1
b	4
c	6
d	8
e	8
f	11
-	12

i	aux[i]
0	a
1	
2	b
3	b
4	
5	c
6	d
7	d
8	
9	f
10	f
11	

Key-indexed counting

Goal. Sort an array `a[]` of `N` integers between `0` and `R-1`.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move records.
-

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move records

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	1
b	4
c	6
d	8
e	8
f	12
-	12

i	aux[i]
0	a
1	
2	b
3	b
4	
5	c
6	d
7	d
8	
9	f
10	f
11	f

Key-indexed counting

Goal. Sort an array `a[]` of `N` integers between `0` and `R-1`.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move records.
-

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move records

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	1
b	5
c	6
d	8
e	8
f	12
-	12

i	aux[i]
0	a
1	
2	b
3	b
4	b
5	c
6	d
7	d
8	
9	f
10	f
11	f

Key-indexed counting

Goal. Sort an array `a[]` of `N` integers between `0` and `R-1`.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move records.
-

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move records

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	1
b	5
c	6
d	8
e	9
f	12
-	12

i	aux[i]
0	a
1	
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

Key-indexed counting

Goal. Sort an array `a[]` of `N` integers between `0` and `R-1`.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move records.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move records

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	2
b	5
c	6
d	8
e	9
f	12
-	12

i	aux[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

Key-indexed counting

Goal. Sort an array `a[]` of `N` integers between `0` and `R-1`.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move records.
-

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move records

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	2
b	5
c	6
d	8
e	9
f	12
-	12

i	aux[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

Key-indexed counting

Goal. Sort an array `a[]` of `N` integers between `0` and `R-1`.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move records.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

copy back

i	a[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

r	count[r]
a	2
b	5
c	6
d	8
e	9
f	12
-	12

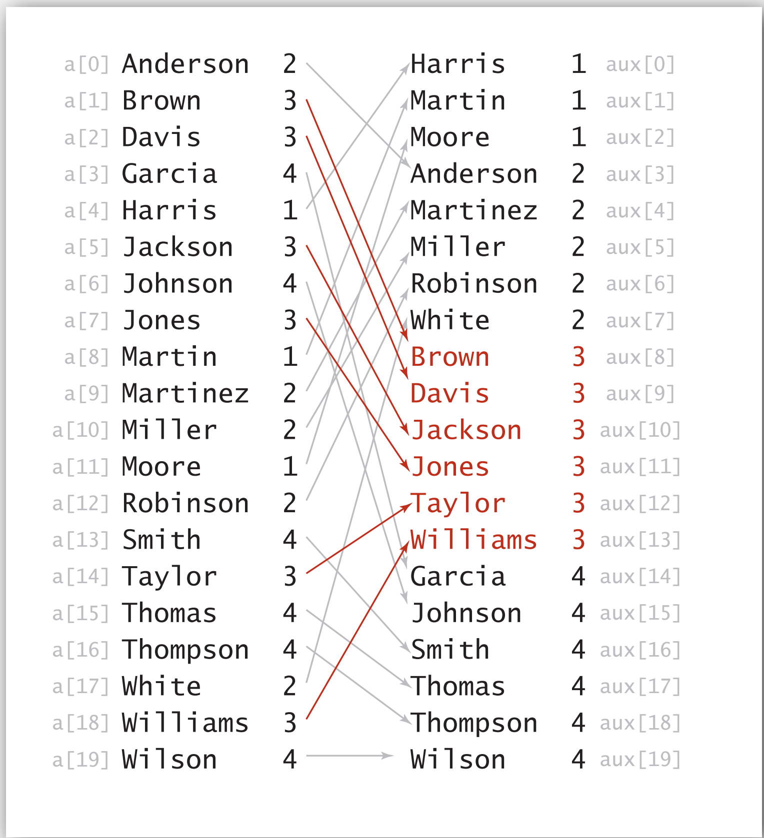
i	aux[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

Key-indexed counting: analysis

Proposition. Key-indexed counting takes time proportional to N + R to sort N records whose keys are integers between 0 and R-1.

Proposition. Key-indexed counting uses extra space proportional to N + R.

Stable? Yes!



- key-indexed counting
- **LSD string sort**
- MSD string sort
- 3-way string quicksort
- suffix arrays

Least-significant-digit-first radix sort

LSD string sort.

- Consider characters from right to left.
- Stably sort using d^{th} character as the key (using key-indexed counting).

		sort key (3rd char)	sort key (2nd char)	sort key (1st char)
0	d a b	d a b	d a b	a c e
1	a d d	c a b	c a b	a d d
2	c a b	e b b	f a d	b a d
3	f a d	a d d	b a d	b e d
4	f e e	f a d	d a d	b e e
5	b a d	b a d	e b b	c a b
6	d a d	d a d	a c e	d a b
7	b e e	f e d	a d d	d a d
8	f e d	b e d	f e d	e b b
9	b e d	f e e	b e d	f a d
10	e b b	b e e	f e e	f e d
11	a c e	a c e	b e e	f e e

sort must be stable
(arrows do not cross)

LSD string sort: correctness proof

Proposition. LSD sorts fixed-length strings in ascending order.

Pf. [thinking about the future]

- If the characters not yet examined differ, it doesn't matter what we do now.
- If the characters not yet examined agree, stability ensures later pass won't affect order.

0	d	a	b
1	c	a	b
2	f	a	d
3	b	a	d
4	d	a	d
5	e	b	b
6	a	c	e
7	a	d	d
8	f	e	d
9	b	e	d
10	f	e	e
11	b	e	e

in order by previous passes

sort key

0	a	c	e
1	a	d	d
2	b	a	d
3	b	e	d
4	b	e	e
5	c	a	b
6	d	a	b
7	d	a	d
8	e	b	b
9	f	a	d
10	f	e	d
11	f	e	e

LSD string sort: Java implementation

```
public class LSD
{
   public static void sort(String[] a, int W)
   {
      int R = 256
      int N = a.length;
      String[] aux = new String[N];
      for (int d = W-1; d >= 0; d--)
      {
         int[] count = new int[R+1];
         for (int i = 0; i < N; i++)
            count[a[i].charAt(d) + 1]++;
         for (int r = 0; r < R; r++)
            count[r+1] += count[r];
         for (int i = 0; i < N; i++)
            aux[count[a[i].charAt(d)]++] = a[i];
         for (int i = 0; i < N; i++)
            a[i] = aux[i];
      }
   }
}
```

fixed-length W strings

radix R

do key-indexed counting
for each digit from right to left

key-indexed counting

LSD string sort: example

Input	d = 6	d = 5	d = 4	d = 3	d= 2	d= 1	d = 0	Output
4PGC938	2IYE230	3CIO720	2IYE230	2RLA629	1ICK750	3ATW723	1ICK750	1ICK750
2IYE230	3CIO720	3CIO720	4JZY524	2RLA629	1ICK750	3CIO720	1ICK750	1ICK750
3CIO720	1ICK750	3ATW723	2RLA629	4PGC938	4PGC938	3CIO720	1OHV845	1OHV845
1ICK750	1ICK750	4JZY524	2RLA629	2IYE230	1OHV845	1ICK750	1OHV845	1OHV845
1OHV845	3CIO720	2RLA629	3CIO720	1ICK750	1OHV845	1ICK750	1OHV845	1OHV845
4JZY524	3ATW723	2RLA629	3CIO720	1ICK750	1OHV845	2IYE230	2IYE230	2IYE230
1ICK750	4JZY524	2IYE230	3ATW723	3CIO720	3CIO720	4JZY524	2RLA629	2RLA629
3CIO720	1OHV845	4PGC938	1ICK750	3CIO720	3CIO720	1OHV845	2RLA629	2RLA629
1OHV845	1OHV845	1OHV845	1ICK750	1OHV845	2RLA629	1OHV845	3ATW723	3ATW723
1OHV845	1OHV845	1OHV845	1OHV845	1OHV845	2RLA629	1OHV845	3CIO720	3CIO720
2RLA629	4PGC938	1OHV845	1OHV845	1OHV845	3ATW723	4PGC938	3CIO720	3CIO720
2RLA629	2RLA629	1ICK750	1OHV845	3ATW723	2IYE230	2RLA629	4JZY524	4JZY524
3ATW723	2RLA629	1ICK750	4PGC938	4JZY524	4JZY524	2RLA629	4PGC938	4PGC938

Summary of the performance of sorting algorithms

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$N^2 / 2$	$N^2 / 4$	1	yes	`compareTo()`
mergesort	N lg N	N lg N	N	yes	`compareTo()`
quicksort	1.39 N lg N *	1.39 N lg N	c lg N	no	`compareTo()`
heapsort	2 N lg N	2 N lg N	1	no	`compareTo()`
LSD †	2 W N	2 W N	N + R	yes	`charAt()`

* probabilistic
† fixed-length W keys

Sorting challenge 1

Problem. Sort a huge commercial database on a fixed-length key field.

Ex. Account number, date, SS number, ...

Which sorting method to use?

- Insertion sort.
- Mergesort.
- Quicksort.
- Heapsort.
- ✓ LSD string sort.

256 (or 65536) counters;
Fixed-length strings sort in W passes.

B14-99-8765
756-12-AD46
CX6-92-0112
332-WX-9877
375-99-QWAX
CV2-59-0221
[illegible]87-SS-0321
[illegible]
KJ-0[illegible]12388
715-YT-013C
MJ0-PP-983F
908-KK-33TY
BBN-63-23RE
48G-BM-912D
982-ER-9P1B
WBL-37-PB81
810-F4-J87Q
LE9-N8-XX76
908-KK-33TY
B14-99-8765
CX6-92-0112
CV2-59-0221
332-WX-23SQ
332-6A-9877

Sorting challenge 2a

Problem. Sort 1 million 32-bit integers.

Ex. Google interview or presidential interview.

Which sorting method to use?

- Insertion sort.
- Mergesort.
- Quicksort.
- Heapsort.
- LSD string sort.

LSD string sort: a moment in history (1960s)

card punch

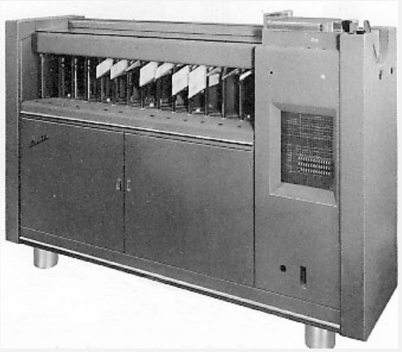
punched cards

card reader

mainframe

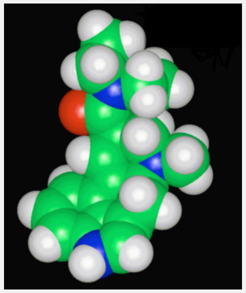
line printer

To sort a card deck
- start on right column
- put cards into hopper
- machine distributes into bins
- pick up cards (stable)
- move left one column
- continue until sorted



card sorter

not related to sorting



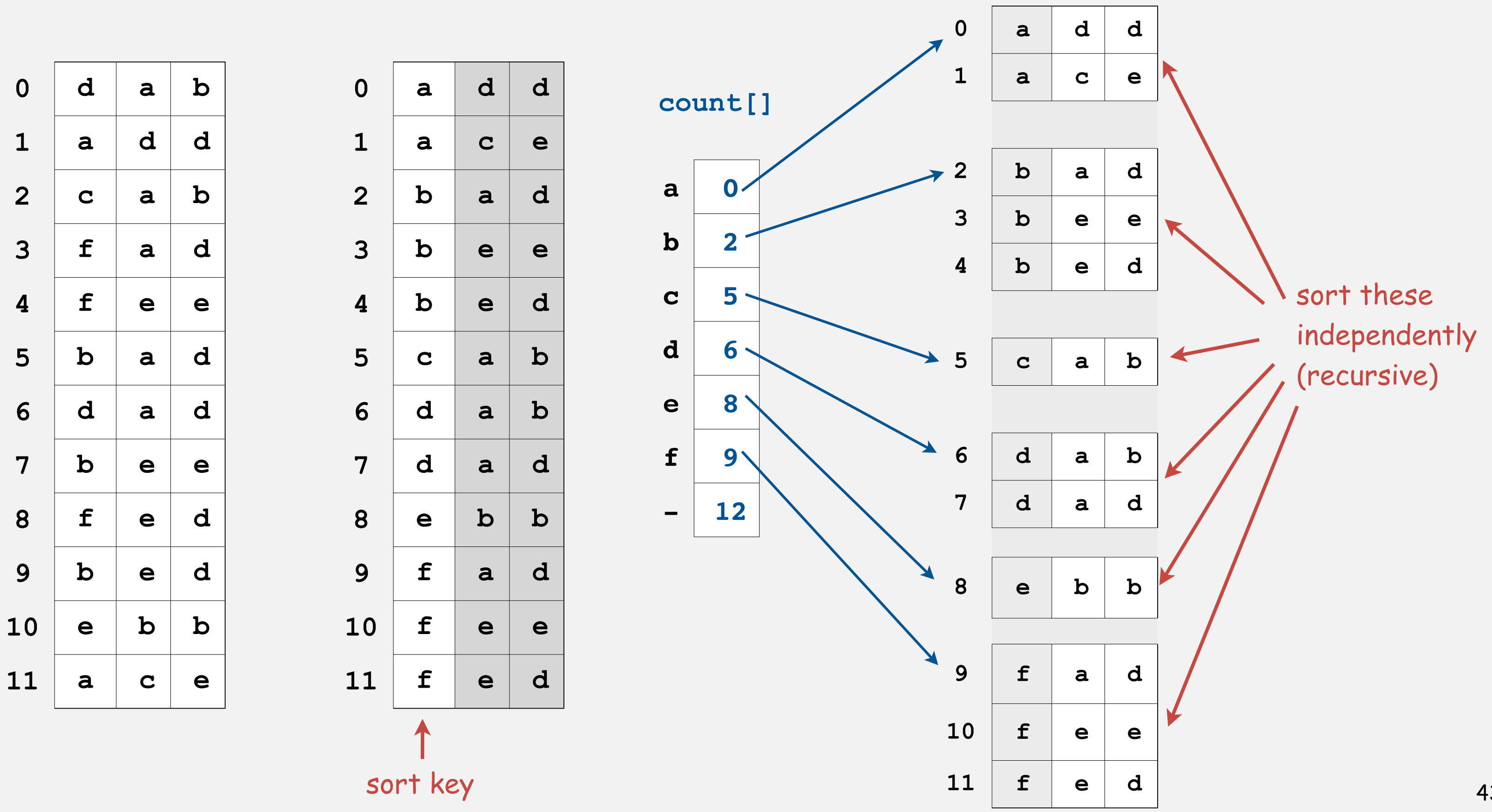
Lysergic Acid Diethylamide
(Lucy in the Sky with Diamonds)

- key-indexed counting
- LSD string sort
- **MSD string sort**
- 3-way string quicksort
- suffix arrays

Most-significant-digit-first string sort

MSD string sort.

- Partition file into R pieces according to first character (use key-indexed counting).
- Recursively sort all strings that start with each character (key-indexed counts delineate subarrays to sort).



0	d	a	b
1	a	d	d
2	c	a	b
3	f	a	d
4	f	e	e
5	b	a	d
6	d	a	d
7	b	e	e
8	f	e	d
9	b	e	d
10	e	b	b
11	a	c	e

0	a	d	d
1	a	c	e
2	b	a	d
3	b	e	e
4	b	e	d
5	c	a	b
6	d	a	b
7	d	a	d
8	e	b	b
9	f	a	d
10	f	e	e
11	f	e	d

MSD string sort: top level trace

use key-indexed counting on first character | **recursively sort subarrays**

i	a[i]
0	she
1	sells
2	seashells
3	by
4	the
5	sea
6	shore
7	the
8	shells
9	she
10	sells
11	are
12	surely
13	seashells

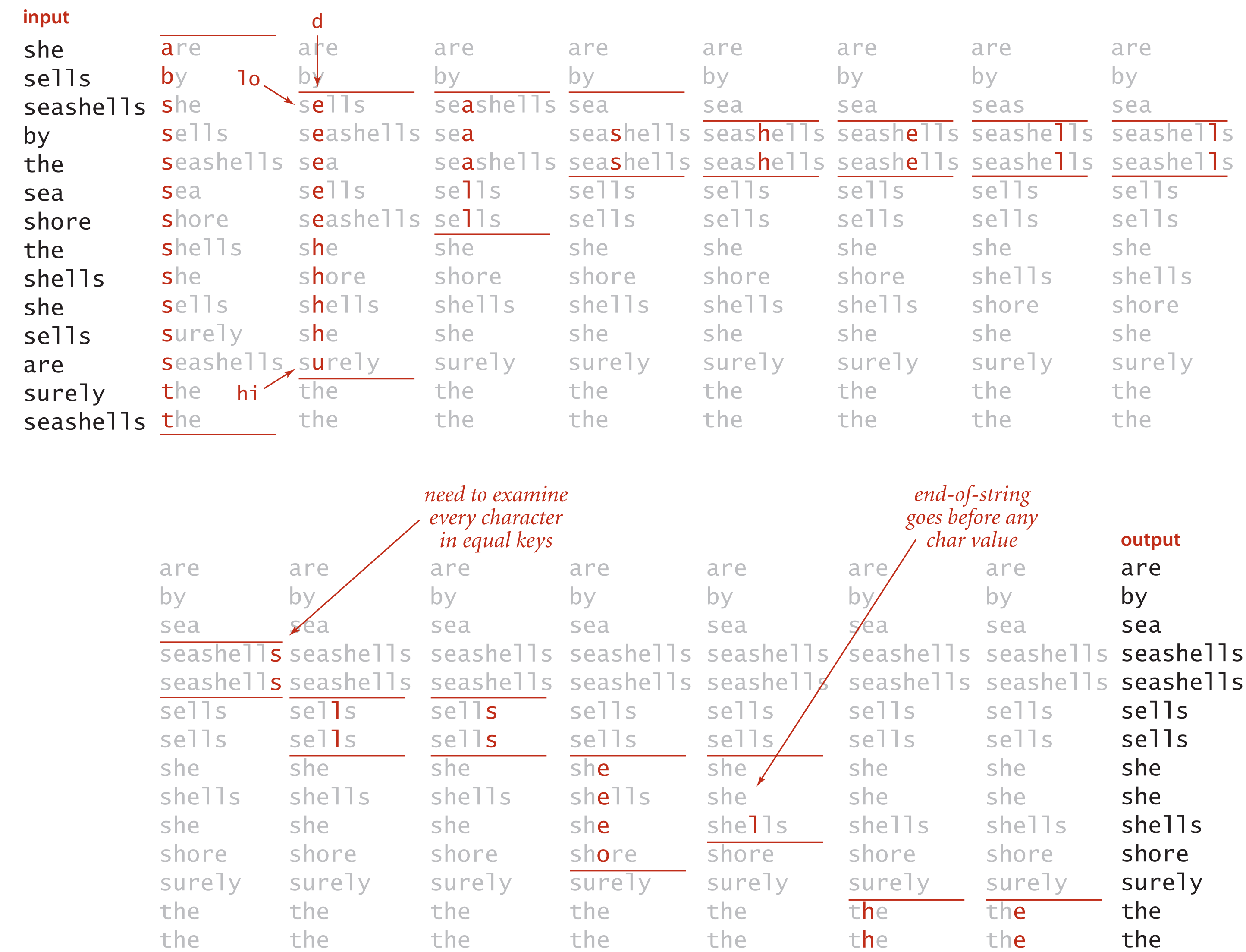
r	char	*count frequencies*	*transform counts to indices*	*indices at completion of distribute phase*	*recursively sort subarrays*
0		0	0	0	sort(a, 0, 0);
1	a	0	0	1	sort(a, 1, 1);
2	b	1	1	2	sort(a, 2, 1);
3	c	1	2	2	sort(a, 2, 1);
4	d	0	2	2	sort(a, 2, 1);
5	e	0	2	2	sort(a, 2, 1);
6	f	0	2	2	sort(a, 2, 1);
7	g	0	2	2	sort(a, 2, 1);
8	h	0	2	2	sort(a, 2, 1);
9	i	0	2	2	sort(a, 2, 1);
10	j	0	2	2	sort(a, 2, 1);
11	k	0	2	2	sort(a, 2, 1);
12	l	0	2	2	sort(a, 2, 1);
13	m	0	2	2	sort(a, 2, 1);
14	n	0	2	2	sort(a, 2, 1);
15	o	0	2	2	sort(a, 2, 1);
16	p	0	2	2	sort(a, 2, 1);
17	q	0	2	2	sort(a, 2, 1);
18	r	0	2	2	sort(a, 2, 11);
19	s	0	2	12	sort(a, 12, 13);
20	t	10	12	14	sort(a, 14, 13);
21	u	2	14	14	sort(a, 14, 13);
22	v	0	14	14	sort(a, 14, 13);
23	w	0	14	14	sort(a, 14, 13);
24	x	0	14	14	sort(a, 14, 13);
25	y	0	14	14	sort(a, 14, 13);
26	z	0	14	14	sort(a, 14, 13);
27		0	14	14	sort(a, 14, 13);

start of s subarray

1 + end of s subarray

i	*distribute and copy back*	sorted
0	are	are
1	by	by
2	she	sea
3	sells	seashells
4	seashells	seashells
5	sea	sells
6	shore	sells
7	shells	she
8	she	she
9	sells	shells
10	surely	shore
11	seashells	surely
12	the	the
13	the	the

MSD string sort: example



Trace of recursive calls for MSD string sort (no cutoff for small subarrays, subarrays of size 0 and 1 omitted)

Variable-length strings

Treat strings as if they had an extra char at end (smaller than any char).

0	s	e	a	-1						
1	s	e	a	s	h	e	l	l	s	-1
2	s	e	l	l	s	-1				
3	s	h	e	-1						
4	s	h	e	-1						
5	s	h	e	l	l	s	-1			
6	s	h	o	r	e	-1				
7	s	u	r	e	l	y	-1			

she before shells

```
private static int charAt(String s, int d)
{
   if (d < s.length()) return s.charAt(d);
   else return -1;
}
```

C strings. Have extra char `'\0'` at end $\Rightarrow$ no extra work needed.

MSD string sort: Java implementation

```
public static void sort(String[] a)
{
  aux = new String[a.length];
  sort(a, aux, 0, a.length, 0);
}

private static void sort(String[] a, String[] aux, int lo, int hi, int d)
{
   if (hi <= lo) return;
   int[] count = new int[R+2];
   for (int i = lo; i <= hi; i++)
      count[charAt(a[i], d) + 2]++;
   for (int r = 0; r < R+1; r++)
      count[r+1] += count[r];
   for (int i = lo; i <= hi; i++)
      aux[count[charAt(a[i], d) + 1]++] = a[i];
   for (int i = lo; i <= hi; i++)
      a[i] = aux[i - lo];

   for (int r = 0; r < R; r++)
      sort(a, aux, lo + count[r], lo + count[r+1] - 1, d+1);
}
```

can recycle `aux[]`
but not `count[]`

key-indexed counting

recursively sort subarrays

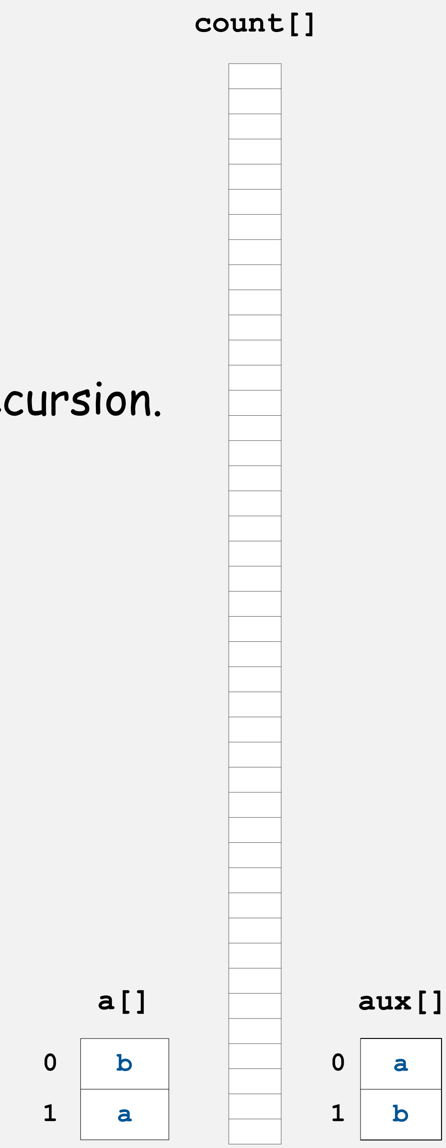
MSD string sort: potential for disastrous performance

Observation 1. Much too slow for small subarrays.

- The `count[]` array must be re-initialized.
- ASCII (256 counts): 100x slower than copy pass for N = 2.
- Unicode (65536 counts): 32,000x slower for N = 2.

Observation 2. Huge number of small subarrays because of recursion.

Solution. Cutoff to insertion sort for small N.



Cutoff to insertion sort

Solution. Cutoff to insertion sort for small N.

- Insertion sort, but start at d^{th} character.
- Implement `less()` so that it compares starting at d^{th} character.

```
public static void sort(String[] a, int lo, int hi, int d)
{
   for (int i = lo; i <= hi; i++)
      for (int j = i; j > lo && less(a[j], a[j-1], d); j--)
         exch(a, j, j-1);
}
```

```
private static boolean less(String v, String w, int d)
{  return v.substring(d).compareTo(w.substring(d)) < 0;  }
```

in Java, forming and comparing substrings is faster than directly comparing chars with `charAt()` !

MSD string sort: performance

Number of characters examined.

- MSD examines just enough characters to sort the keys.
- Number of characters examined depends on keys.
- Can be sublinear!

Random (sublinear)	Non-random with duplicates (nearly linear)	Worst case (linear)
1EIO402	are	1DNB377
1HYL490	by	1DNB377
1ROZ572	sea	1DNB377
2HXE734	seashells	1DNB377
2IYE230	seashells	1DNB377
2XOR846	sells	1DNB377
3CDB573	sells	1DNB377
3CVP720	she	1DNB377
3IGJ319	she	1DNB377
3KNA382	shells	1DNB377
3TAV879	shore	1DNB377
4CQP781	surely	1DNB377
4QGI284	the	1DNB377
4YHV229	the	1DNB377

Characters examined by MSD string sort

Summary of the performance of sorting algorithms

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$N^2 /2$	$N^2 /4$	1	yes	`compareTo()`
mergesort	N lg N	N lg N	N	yes	`compareTo()`
quicksort	1.39 N lg N *	1.39 N lg N	c lg N	no	`compareTo()`
heapsort	2 N lg N	2 N lg N	1	no	`compareTo()`
LSD †	2 N W	2 N W	N + R	yes	`charAt()`
MSD ‡	2 N W	$N \log_R N$	N + D R	yes	`charAt()`

stack depth D = length of longest prefix match

* probabilistic
† fixed-length W keys
‡ average-length W keys

MSD string sort vs. quicksort for strings

Disadvantages of MSD string sort.

- Accesses memory "randomly" (cache inefficient).
- Inner loop has a lot of instructions.
- Extra space for `count[]`.
- Extra space for `aux[]`.

Disadvantage of quicksort.

- Linearithmic number of string compares (not linear).
- Has to rescan long keys for compares.

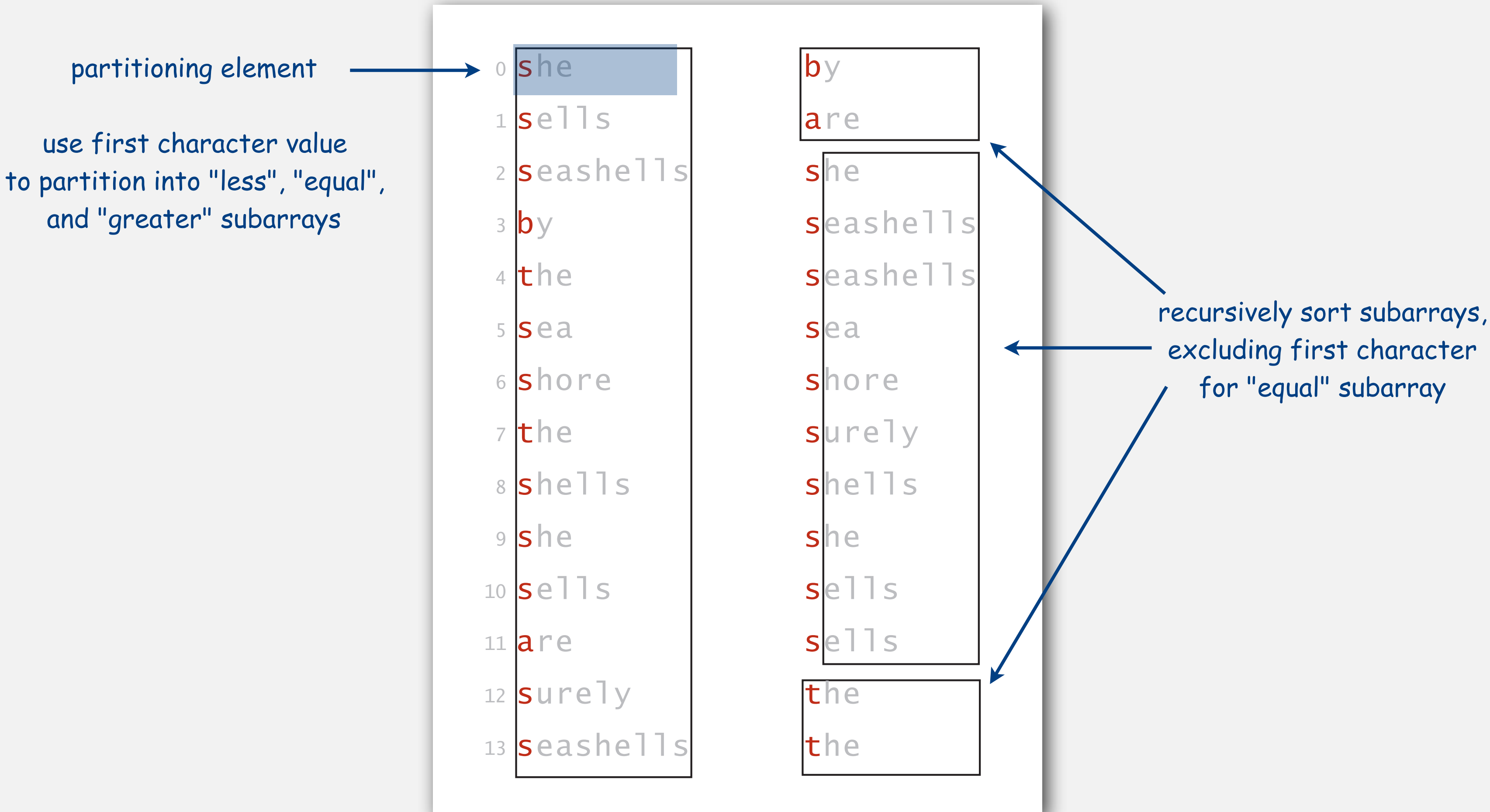
  [but stay tuned]

- key-indexed counting
- LSD string sort
- MSD string sort
- **3-way string quicksort**
- suffix arrays

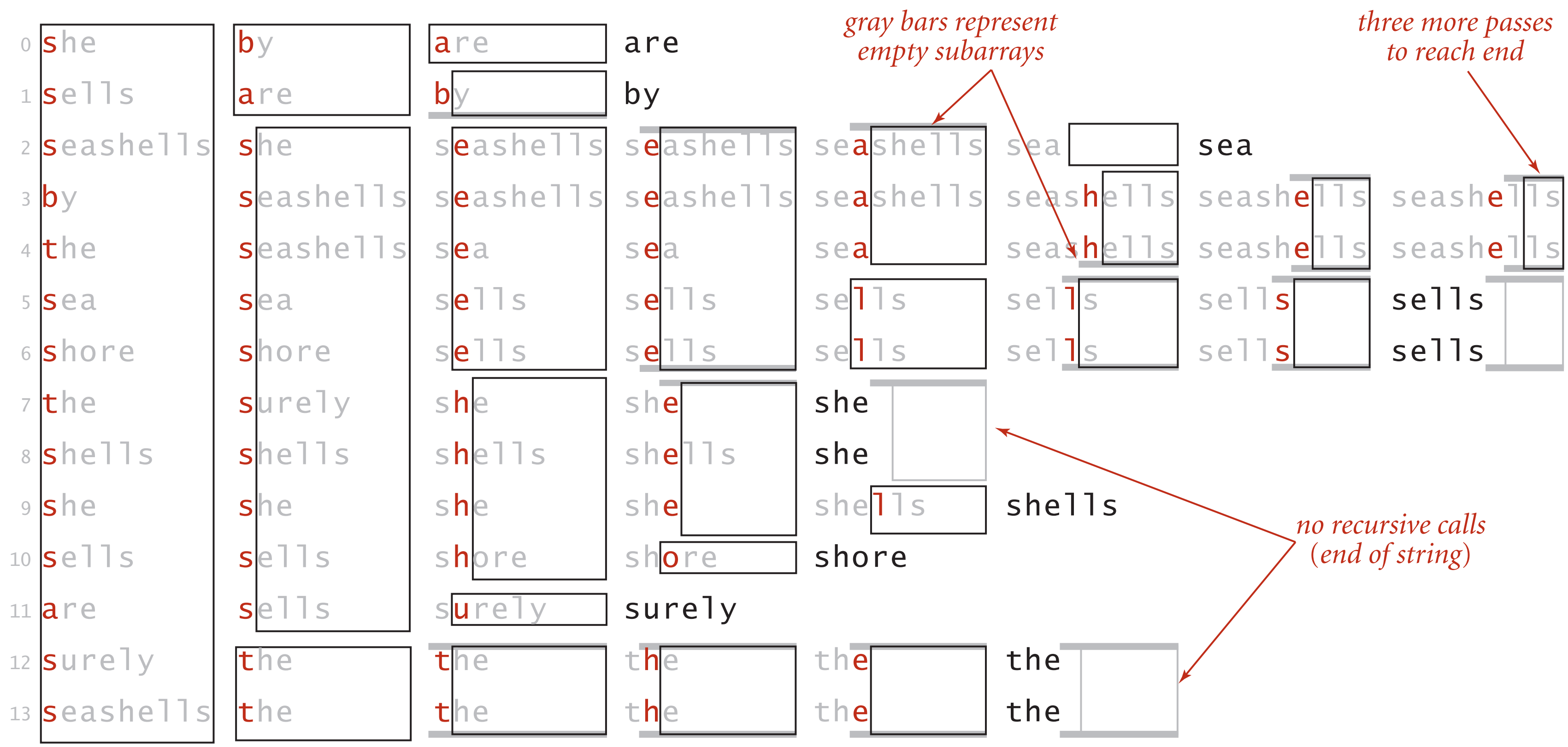
3-way string quicksort (Bentley and Sedgewick, 1997)

Overview. Do 3-way partitioning on the d^{th} character.

- Cheaper than R-way partitioning of MSD string sort.
- Need not examine again characters equal to the partitioning char.



3-way string quicksort: trace of recursive calls



Trace of recursive calls for 3-way string quicksort (no cutoff for small subarrays)

3-way string quicksort: Java implementation

```
private static void sort(String[] a)
{  sort(a, 0, a.length - 1, 0);  }

private static void sort(String[] a, int lo, int hi, int d)
{
   int lt = lo, gt = hi;
   int v = charAt(a[lo], d);
   int i = lo + 1;
   while (i <= gt)
   {
      int t = charAt(a[i], d);
      if      (t < v) exch(a, lt++, i++);
      else if (t > v) exch(a, i, gt--);
      else            i++;
   }

   sort(a, lo, lt-1, d);
   if (v >= 0) sort(a, lt, gt, d+1);
   sort(a, gt+1, hi, d);
}
```

3-way partitioning, using d^{th} character

sort 3 pieces recursively

3-way string quicksort vs. standard quicksort

Standard quicksort.

- Uses 2N ln N string compares on average.
- Costly for long keys that differ only at the end (and this is a common case!)

3-way string quicksort.

- Uses 2 N ln N character compares on average for random strings.
- Avoids recomparing initial parts of the string.
- Adapts to data: uses just "enough" characters to resolve order.
- Sublinear when strings are long.

Proposition. 3-way string quicksort is optimal (to within a constant factor); no sorting algorithm can (asymptotically) examine fewer chars.

Pf. Ties cost to entropy. Beyond scope of 226.

3-way string quicksort vs. MSD string sort

MSD string sort.

- Has a long inner loop.
- Is cache-inefficient.
- Too much overhead reinitializing `count[]` and `aux[]`.

3-way string quicksort.

- Has a short inner loop.
- Is cache-friendly.
- Is in-place.

library call numbers

```
WUS-------10706-----7---10
WUS-------12692-----4---27
WLSOC------2542----30
LTK--6015-P-63-1988
LDS---361-H-4
...
```

Bottom line. 3-way string quicksort is the method of choice for sorting strings.

Summary of the performance of sorting algorithms

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$N^2 /2$	$N^2 /4$	1	yes	`compareTo()`
mergesort	N lg N	N lg N	N	yes	`compareTo()`
quicksort	1.39 N lg N *	1.39 N lg N	c lg N	no	`compareTo()`
heapsort	2 N lg N	2 N lg N	1	no	`compareTo()`
LSD †	2 N W	2 N W	N + R	yes	`charAt()`
MSD ‡	2 N W	$N \log_R N$	N + D R	yes	`charAt()`
3-way string quicksort	1.39 W N lg N *	1.39 N lg N	log N + W	no	`charAt()`

* probabilistic
† fixed-length W keys
‡ average-length W keys

- key-indexed counting
- LSD string sort
- MSD string sort
- 3-way radix quicksort
- **suffix arrays**

Warmup: longest common prefix

LCP. Given two strings, find the longest substring that is a prefix of both.

p	r	e	f	e	t	c	h
0	1	2	3	4	5	6	7
p	r	e	f	i	x		

```
public static String lcp(String s, String t)
{
   int n = Math.min(s.length(), t.length());
   for (int i = 0; i < n; i++)
   {
      if (s.charAt(i) != t.charAt(i))
         return s.substring(0, i);
   }
   return s.substring(0, n);
}
```

Running time. Linear-time in length of prefix match.

Space. Constant extra space.

Longest repeated substring

LRS. Given a string of N characters, find the longest repeated substring.

Ex.

```
a a c a a g t t t a c a a g c a t g a t g c t g t a c t a
g g a g a g t t a t a c t g g t c g t c a a a c c t g a a
c c t a a t c c t t g t g t g t a c a c a c a c t a c t a
c t g t c g t c g t c a t a t a t c g a g a t c a t c g a
a c c g g a a g g c c g g a c a a g g c g g g g g g t a t
a g a t a g a t a g a c c c c t a g a t a c a c a t a c a
t a g a t c t a g c t a g c t a g c t c a t c g a t a c a
c a c t c t c a c a c t c a a g a g t t a t a c t g g t c
a a c a c a c t a c t a c g a c a g a c g a c c a a c c a
g a c a g a a a a a a a a c t c t a t a t c t a t a a a a
```

Applications. Bioinformatics, cryptanalysis, data compression, ...

Longest repeated substring: a musical application

Visualize repetitions in music. `http://www.bewitched.com`

Mary Had a Little Lamb

Bach's Goldberg Variations

LRS. Given a string of N characters, find the longest repeated substring.

Brute force algorithm.

- Try all indices `i` and `j` for start of possible match.
- Compute longest common prefix (LCP) for each pair.

Analysis. Running time ≤ $M N^2$, where M is length of longest match.

Longest repeated substring: a sorting solution

input string

a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

form suffixes

0	a a c a a g t t t a c a a g c
1	a c a a g t t t a c a a g c
2	c a a g t t t a c a a g c
3	a a g t t t a c a a g c
4	a g t t t a c a a g c
5	g t t t a c a a g c
6	t t t a c a a g c
7	t t a c a a g c
8	t a c a a g c
9	a c a a g c
10	c a a g c
11	a a g c
12	a g c
13	g c
14	c

sort suffixes to bring repeated substrings together

0	a a c a a g t t t a c a a g c
11	a a g c
3	a a g t t t a c a a g c
9	a c a a g c
1	a c a a g t t t a c a a g c
12	a g c
4	a g t t t a c a a g c
14	c
10	c a a g c
2	c a a g t t t a c a a g c
13	g c
5	g t t t a c a a g c
8	t a c a a g c
7	t t a c a a g c
6	t t t a c a a g c

compute longest prefix between adjacent suffixes

a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Longest repeated substring: Java implementation

```
public String lrs(String s)
{
   int N = s.length();

   String[] suffixes = new String[N];
   for (int i = 0; i < N; i++)
      suffixes[i] = s.substring(i, N);

   Arrays.sort(suffixes);

   String lrs = "";
   for (int i = 0; i < N-1; i++)
   {
      String x = lcp(suffixes[i], suffixes[i+1]);
      if (x.length() > lrs.length()) lrs = x;
   }
   return lrs;
}
```

create suffixes (linear time and space)

sort suffixes

find LCP between suffixes that are adjacent after sorting

```
% java LRS < mobydick.txt
,- Such a funny, sporty, gamy, jesty, joky, hoky-poky lad, is the Ocean, oh! Th
```

Sorting challenge

Problem. Five scientists A, B, C, D, and E are looking for long repeated substring in a genome with over 1 billion nucleotides.

- A has a grad student do it by hand.
- B uses brute force (check all pairs).
- C uses suffix sorting solution with insertion sort.
- D uses suffix sorting solution with LSD string sort.
- ✓ E uses suffix sorting solution with 3-way string quicksort.

only if LRS is not long (!)

Q. Which one is more likely to lead to a cure cancer?

Longest repeated substring: empirical analysis

input file	characters	brute	suffix sort	length of LRS
`LRS.java`	2,162	0.6 sec	0.14 sec	73
`amendments.txt`	18,369	37 sec	0.25 sec	216
`aesop.txt`	191,945	1.2 hours	1.0 sec	58
`mobydick.txt`	1.2 million	43 hours †	7.6 sec	79
`chromosome11.txt`	7.1 million	2 months †	61 sec	12,567
`pi.txt`	10 million	4 months †	84 sec	14

† estimated

Suffix sorting: worst-case input

Longest repeated substring not long. Hard to beat 3-way string quicksort.

Longest repeated substring very long.

- Radix sorts are quadratic in the length of the longest match.
- Ex: two copies of Aesop's fables.

```
% more abcdefgh2.txt
abcdefgh
abcdefghabcdefgh
bcdefgh
bcdefghabcdefgh
cdefgh
cdefghabcdefgh
defgh
efghabcdefgh
efgh
fghabcdefgh
fgh
ghabcdefgh
fh
habcdefgh
h
```

	time to suffix sort (seconds)	
algorithm	mobydick.txt	aesopaesop.txt
brute-force	36,000 †	4000 †
quicksort	9.5	167
LSD	not fixed length	not fixed length
MSD	395	out of memory
MSD with cutoff	6.8	162
3-way string quicksort	2.8	400

† estimated

Suffix sorting challenge

Problem. Suffix sort an arbitrary string of length N.

Q. What is worst-case running time of best algorithm for problem?

- Quadratic.
- ✓ Linearithmic. ← Manber's algorithm
- ✓ Linear. ← suffix trees (see COS 423)
- Nobody knows.

Suffix sorting in linearithmic time

Manber's MSD algorithm.

- Phase 0: sort on first character using key-indexed counting sort.
- Phase i: given array of suffixes sorted on first 2^{i-1} characters, create array of suffixes sorted on first 2^i characters.

Worst-case running time. N log N.

- Finishes after lg N phases.
- Can perform a phase in linear time. (!) [stay tuned]

Linearithmic suffix sort example: phase 0

original suffixes

0	b a b a a a a b c b a b a a a a a 0
1	a b a a a a b c b a b a a a a a 0
2	b a a a a b c b a b a a a a a 0
3	a a a a b c b a b a a a a a 0
4	a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0
6	a b c b a b a a a a a 0
7	b c b a b a a a a a 0
8	c b a b a a a a a 0
9	b a b a a a a a 0
10	a b a a a a a 0
11	b a a a a a 0
12	a a a a a 0
13	a a a a 0
14	a a a 0
15	a a 0
16	a 0
17	0

key-indexed counting sort (first character)

17	0	
1	a	b a a a a b c b a b a a a a a 0
16	a	0
3	a	a a a b c b a b a a a a a 0
4	a	a a b c b a b a a a a a 0
5	a	a b c b a b a a a a a 0
6	a	b c b a b a a a a a 0
15	a	a 0
14	a	a a 0
13	a	a a a 0
12	a	a a a a 0
10	a	b a a a a a 0
0	b	a b a a a a b c b a b a a a a a 0
9	b	a b a a a a a 0
11	b	a a a a a 0
7	b	c b a b a a a a a 0
2	b	a a a a b c b a b a a a a a 0
8	c	b a b a a a a a 0

sorted

Linearithmic suffix sort example: phase 1

original suffixes

0	b a b a a a a b c b a b a a a a a 0
1	a b a a a a b c b a b a a a a a 0
2	b a a a a b c b a b a a a a a 0
3	a a a a b c b a b a a a a a 0
4	a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0
6	a b c b a b a a a a a 0
7	b c b a b a a a a a 0
8	c b a b a a a a a 0
9	b a b a a a a a 0
10	a b a a a a a 0
11	b a a a a a 0
12	a a a a a 0
13	a a a a 0
14	a a a 0
15	a a 0
16	a 0
17	0

index sort (first two characters)

17	0	
16	a 0	
12	a a	a a a 0
3	a a	a a b c b a b a a a a a 0
4	a a	a b c b a b a a a a a 0
5	a a	b c b a b a a a a a 0
13	a a	a a 0
15	a a	0
14	a a	a 0
6	a b	c b a b a a a a a 0
1	a b	a a a a b c b a b a a a a a 0
10	a b	a a a a a 0
0	b a	b a a a a b c b a b a a a a a 0
9	b a	b a a a a a 0
11	b a	a a a a 0
2	b a	a a a a b c b a b a a a a a 0
7	b c	b a b a a a a a 0
8	c b	a b a a a a a 0

↑ *sorted*

Linearithmic suffix sort example: phase 2

original suffixes

0	b a b a a a a b c b a b a a a a a 0
1	a b a a a a b c b a b a a a a a 0
2	b a a a a b c b a b a a a a a 0
3	a a a a b c b a b a a a a a 0
4	a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0
6	a b c b a b a a a a a 0
7	b c b a b a a a a a 0
8	c b a b a a a a a 0
9	b a b a a a a a 0
10	a b a a a a a 0
11	b a a a a a 0
12	a a a a a 0
13	a a a a 0
14	a a a 0
15	a a 0
16	a 0
17	0

index sort (first four characters)

17	0	
16	a 0	
15	a a 0	
14	a a a 0	
3	a a a a	b c b a b a a a a a 0
12	a a a a	a 0
13	a a a a	0
4	a a a b	c b a b a a a a a 0
5	a a b c	b a b a a a a a 0
1	a b a a	a a b c b a b a a a a a 0
10	a b a a	a a a 0
6	a b c b	a b a a a a a 0
2	b a a a	a b c b a b a a a a a 0 a 0
11	b a a a	a a 0
0	b a b a	a a a b c b a b a a a a a 0
9	b a b a	a a a a 0
7	b c b a	b a a a a a 0
8	c b a b	a a a a a 0

sorted

Linearithmic suffix sort example: phase 3

	original suffixes		*index sort (first eight characters)*
0	b a b a a a a b c b a b a a a a a 0	17	0
1	a b a a a a b c b a b a a a a a 0	16	a 0
2	b a a a a b c b a b a a a a a 0	15	a a 0
3	a a a a b c b a b a a a a a 0	14	a a a 0
4	a a a b c b a b a a a a a 0	13	a a a a 0
5	a a b c b a b a a a a a 0	12	a a a a a 0
6	a b c b a b a a a a a 0	3	a a a a b c b a b a a a a a 0
7	b c b a b a a a a a 0	4	a a a b c b a b a a a a a 0
8	c b a b a a a a a 0	5	a a b c b a b a a a a a 0
9	b a b a a a a a 0	10	a b a a a a a 0
10	a b a a a a a 0	1	a b a a a a b c b a b a a a a a 0
11	b a a a a a 0	6	a b c b a b a a a a a 0
12	a a a a a 0	11	b a a a a a 0
13	a a a a 0	2	b a a a a b c b a b a a a a a 0 a 0
14	a a a 0	9	b a b a a a a a 0
15	a a 0	0	b a b a a a a b c b a b a a a a a 0
16	a 0	7	b c b a b a a a a a 0
17	0	8	c b a b a a a a a 0

sorted

FINISHED! (no equal keys)

Achieve constant-time string compare by indexing into inverse

	original suffixes		*index sort (first four characters)*		*inverse*
0	b a b a a a a b c b a b a a a a a 0	17	0	0	14
1	a b a a a a b c b a b a a a a a 0	16	a 0	1	9
2	b a a a a b c b a b a a a a a 0	15	a a 0	2	12
3	a a a a b c b a b a a a a a 0	14	a a a 0	3	4
4	a a a b c b a b a a a a a 0	3	a a a a b c b a b a a a a a 0	4	$\boxed{7}$
5	a a b c b a b a a a a a 0	12	a a a a a 0	5	8
6	a b c b a b a a a a a 0	13	$\boxed{\texttt{a a a a}}$ 0	6	11
7	b c b a b a a a a a 0	4	$\boxed{\texttt{a a a b}}$ c b a b a a a a a 0	7	16
8	c b a b a a a a a 0	5	a a b c b a b a a a a a 0	8	17
9	b a b a a a a a 0	1	a b a a a a b c b a b a a a a a 0	9	15
10	a b a a a a a 0	10	a b a a a a a 0	10	10
11	b a a a a a 0	6	a b c b a b a a a a a 0	11	13
12	a a a a a 0	2	b a a a a b c b a b a a a a a 0 a 0	12	5
13	a a a a 0	11	b a a a a a 0	13	$\boxed{6}$
14	a a a 0	0	b a b a $\boxed{\texttt{a a a b}}$ c b a b a a a a a 0	14	3
15	a a 0	9	b a b a $\boxed{\texttt{a a a a}}$ 0	15	2
16	a 0	7	b c b a b a a a a a 0	16	1
17	0	8	c b a b a a a a a 0	17	0

0 + 4 = 4

9 + 4 = 13

suffixes$_4$[13] ≤ suffixes$_4$[4] (*because* **inverse[13] < inverse[4]**)

so **suffixes$_8$[9] ≤ suffixes$_8$[0]**

Suffix sort: experimental results

algorithm	time to suffix sort (seconds) mobydick.txt	aesopaesop.txt
brute-force	36.000 †	4000 †
quicksort	9.5	167
LSD	not fixed length	not fixed length
MSD	395	out of memory
MSD with cutoff	6.8	162
3-way string quicksort	2.8	400
Manber MSD	17	8.5

† estimated

String sorting summary

We can develop linear-time sorts.

- Compares not necessary for string keys.
- Use digits to index an array.

We can develop sublinear-time sorts.

- Should measure amount of data in keys, not number of keys.
- Not all of the data has to be examined.

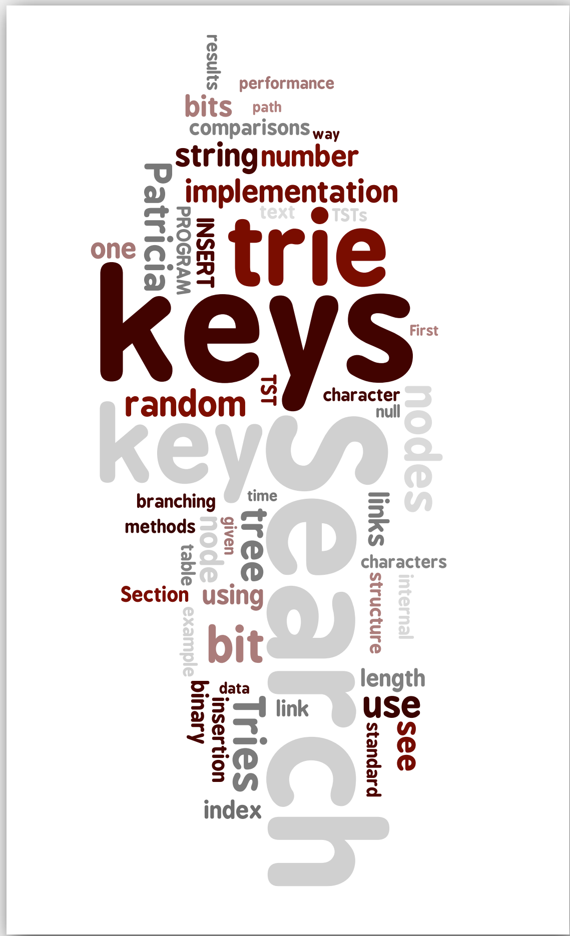
3-way string quicksort is asymptotically optimal.

- 1.39 N lg N chars for random data.

Long strings are rarely random in practice.

- Goal is often to learn the structure!
- May need specialized algorithms.

5.2 Tries



- tries
- TSTs
- applications

Review: summary of the performance of symbol-table implementations

Frequency of operations.

implementation	typical case			ordered operations	operations on keys
	search	insert	delete		
red-black BST	1.00 lg N	1.00 lg N	1.00 lg N	yes	`compareTo()`
hashing	1 †	1 †	1 †	no	`equals()` `hashcode()`

† under uniform hashing assumption

Q. Can we do better?

A. Yes, if we can avoid examining the entire key, as with string sorting.

String symbol table basic API

String symbol table. Symbol table specialized to string keys.

`public class StringST<Value>`	*string symbol table type*
`StringST()`	*create an empty symbol table*
`void put(String key, Value val)`	*put key-value pair into the symbol table*
`Value get(String key)`	*return value paired with given key*
`boolean contains(String key)`	*is there a value paired with the given key?*

Goal. As fast as hashing, more flexible than binary search trees.

String symbol table implementations cost summary

	character accesses (typical case)				dedup	
implementation	search hit	search miss	insert	space (links)	`moby.txt`	`actors.txt`
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4N$	1.40	97.4
hashing	L	L	L	$4N$ to $16N$	0.76	40.6

Parameters
- N = number of strings
- L = length of string
- R = radix

file	size	words	distinct
`moby.txt`	1.2 MB	210 K	32 K
`actors.txt`	82 MB	11.4 M	900 K

Challenge. Efficient performance for string keys.

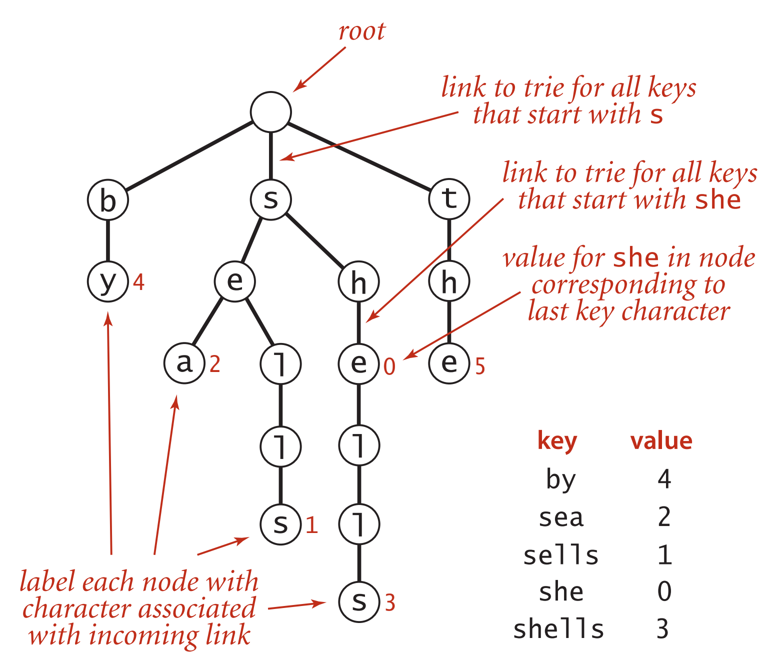
- tries
- TSTs
- string symbol table API

Tries

Tries. [from re**trie**val, but pronounced "try"]

- Store characters and values in nodes (not keys).
- Each node has R children, one for each possible character.

Ex. `she sells sea shells by the`



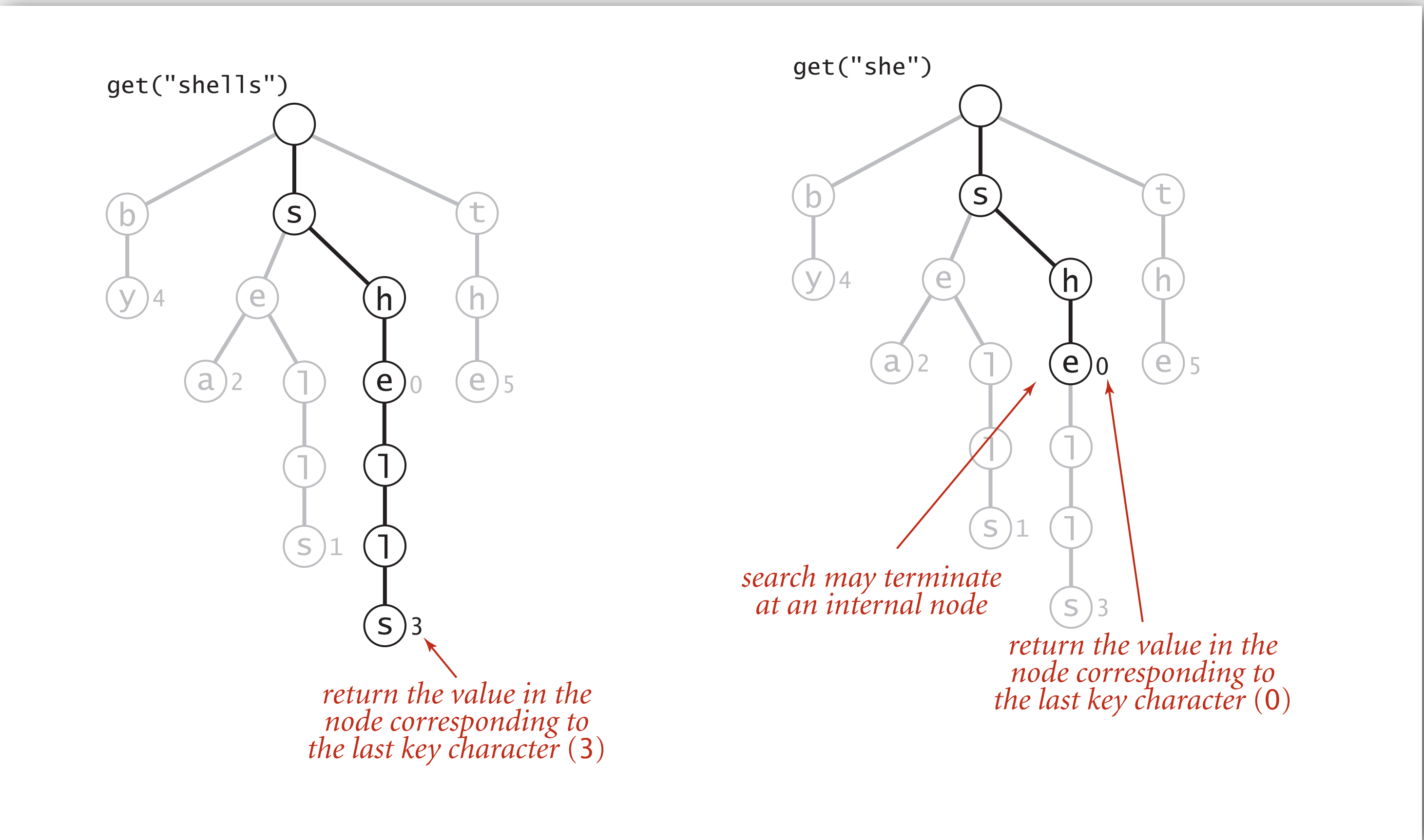
key	value
by	4
sea	2
sells	1
she	0
shells	3
the	5

Anatomy of a trie

Search in a trie

Follow links corresponding to each character in the key.

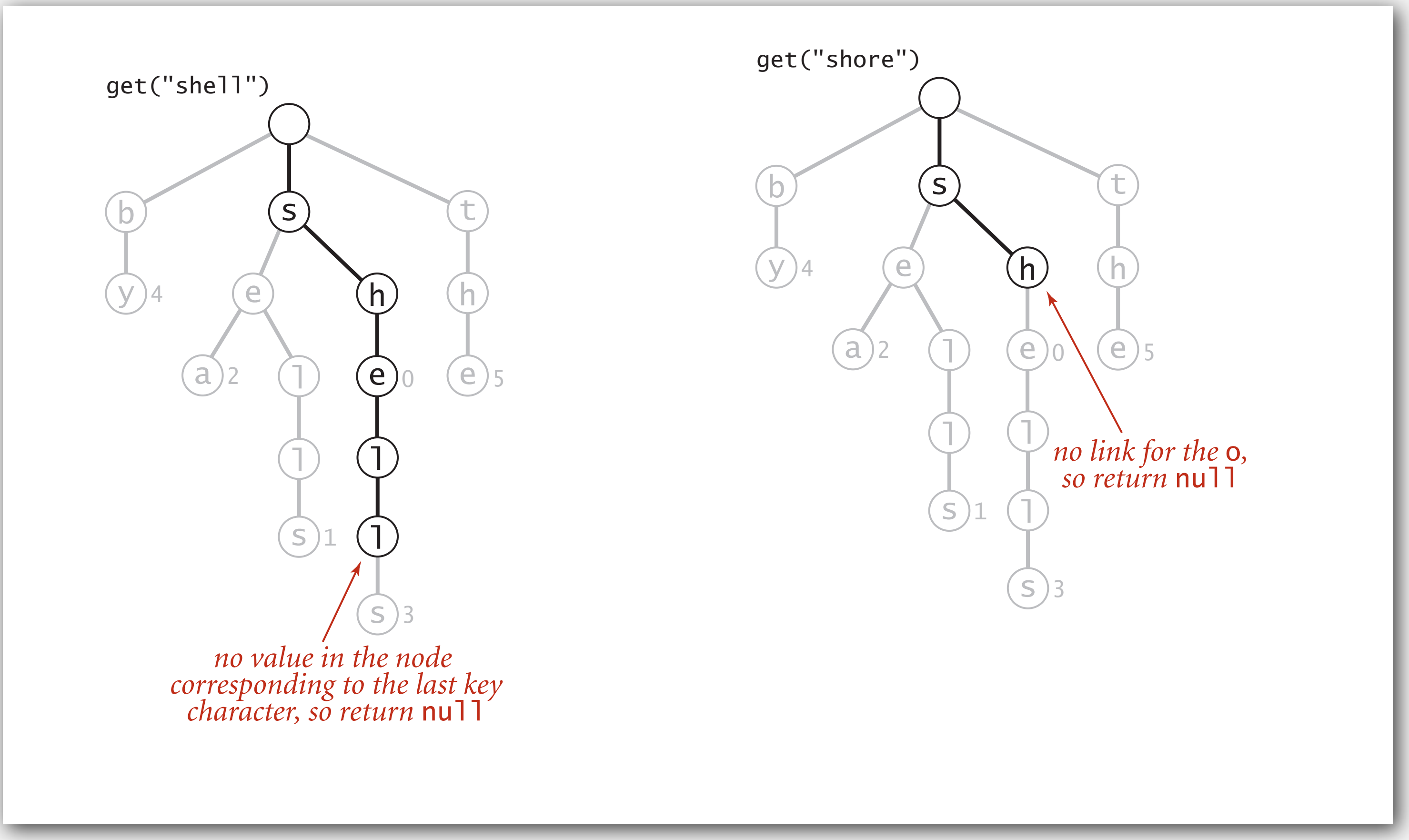
- Search hit: node where search ends has a non-null value.
- Search miss: reach a null link or node where search ends has null value.



Search in a trie

Follow links corresponding to each character in the key.

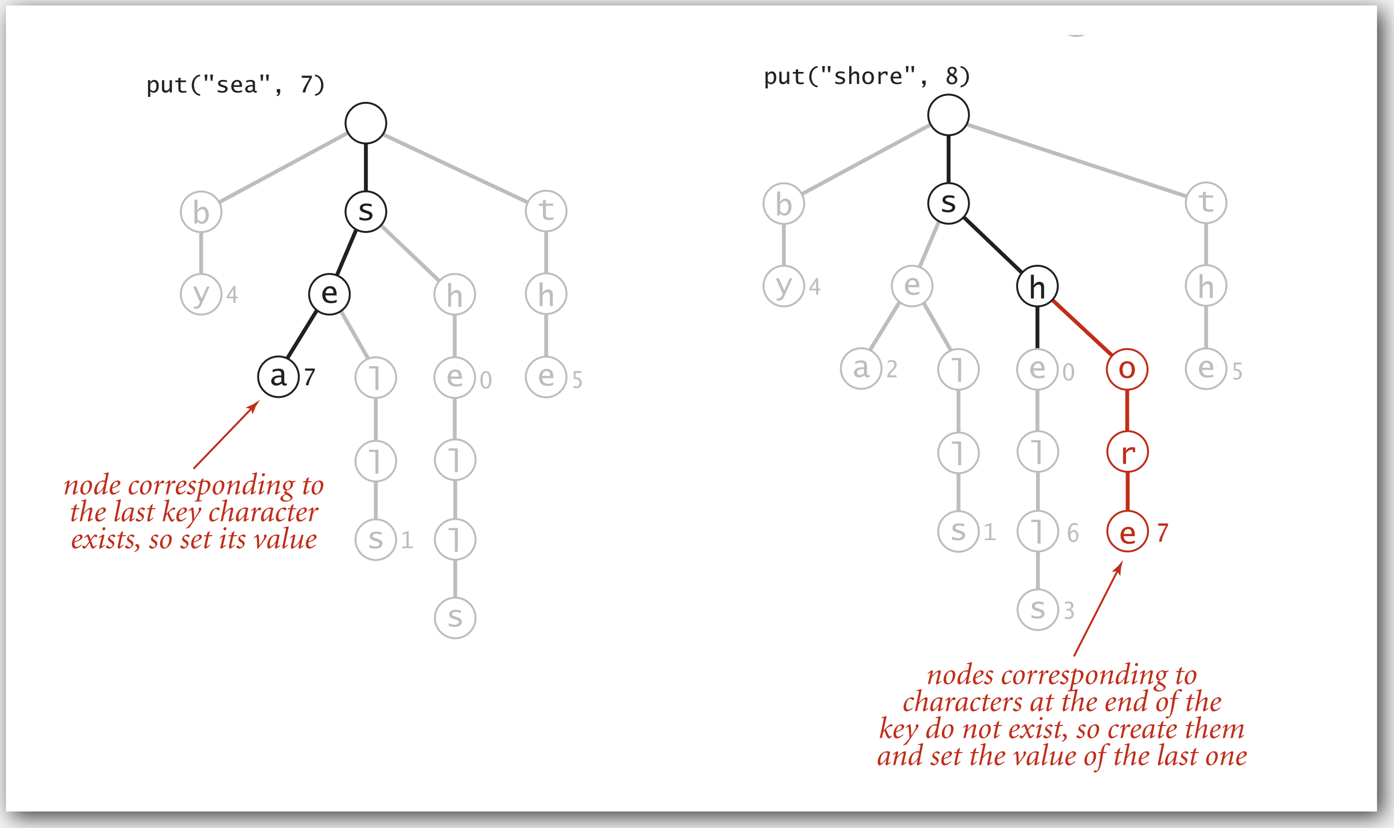
- Search hit: node where search ends has a non-null value.
- Search miss: reach a null link or node where search ends has null value.



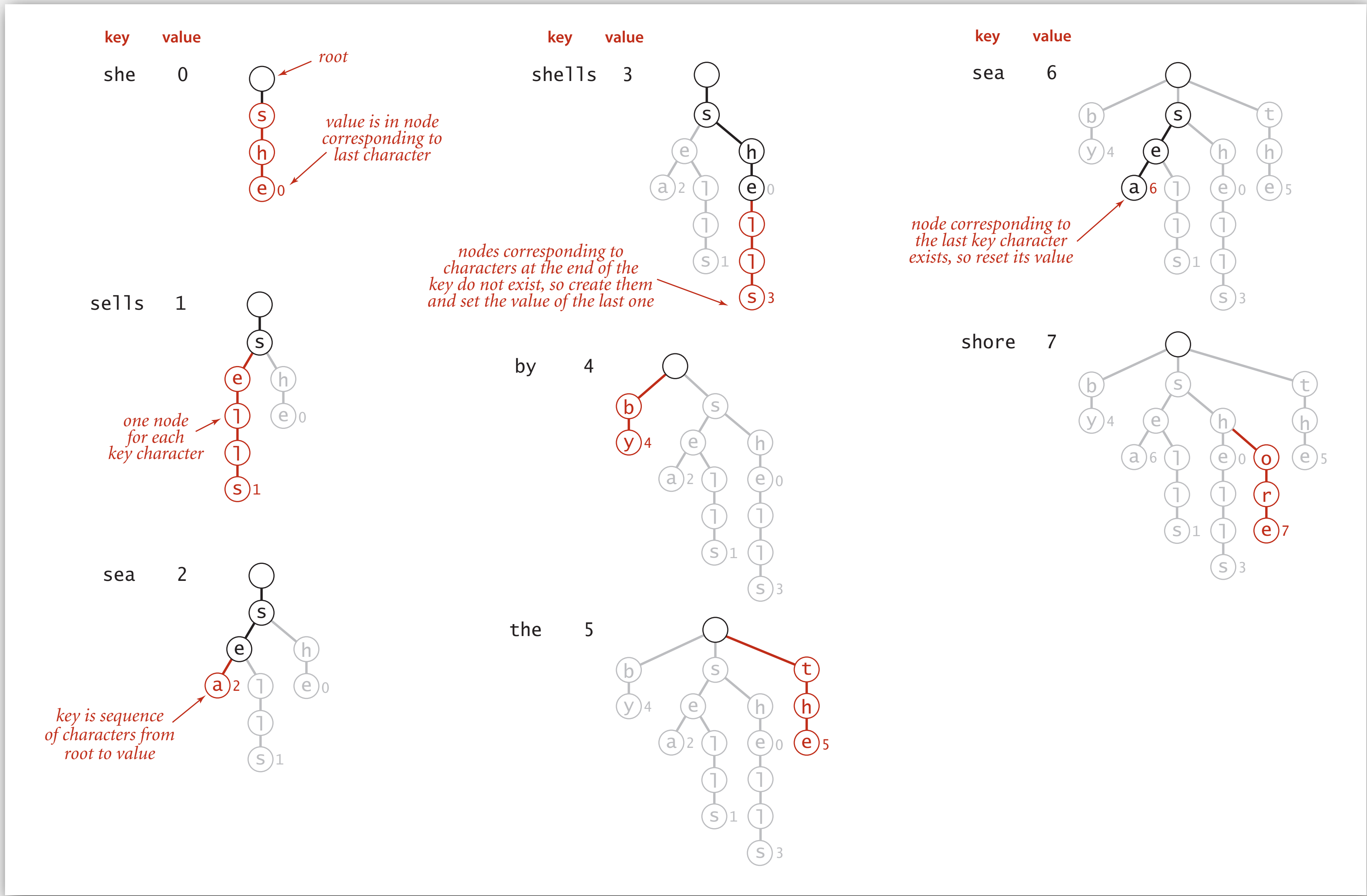
Insertion into a trie

Follow links corresponding to each character in the key.

- Encounter a null link: create new node.
- Encounter the last character of the key: set value in that node.



Trie construction example

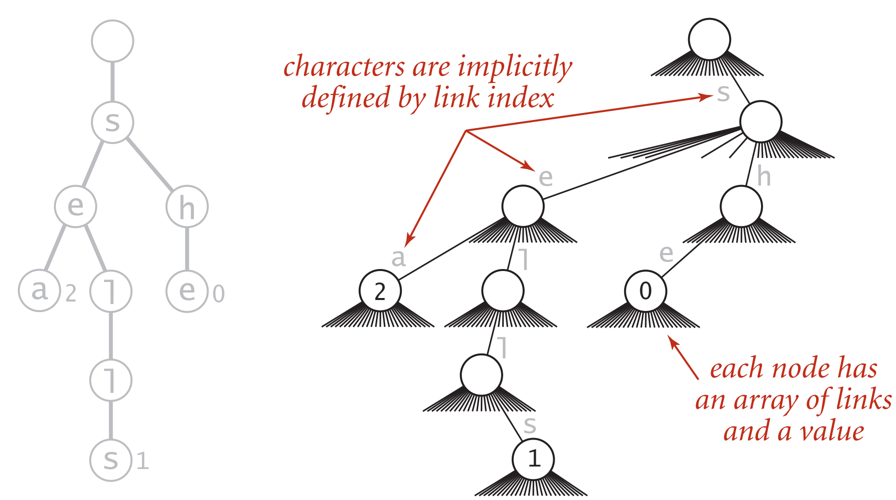


Trie representation: Java implementation

Node. A value, plus references to R nodes.

```
private static class Node
{
   private Object value;
   private Node[] next = new Node[R];
}
```

use `Object` instead of `Value` since no generic array creation in Java



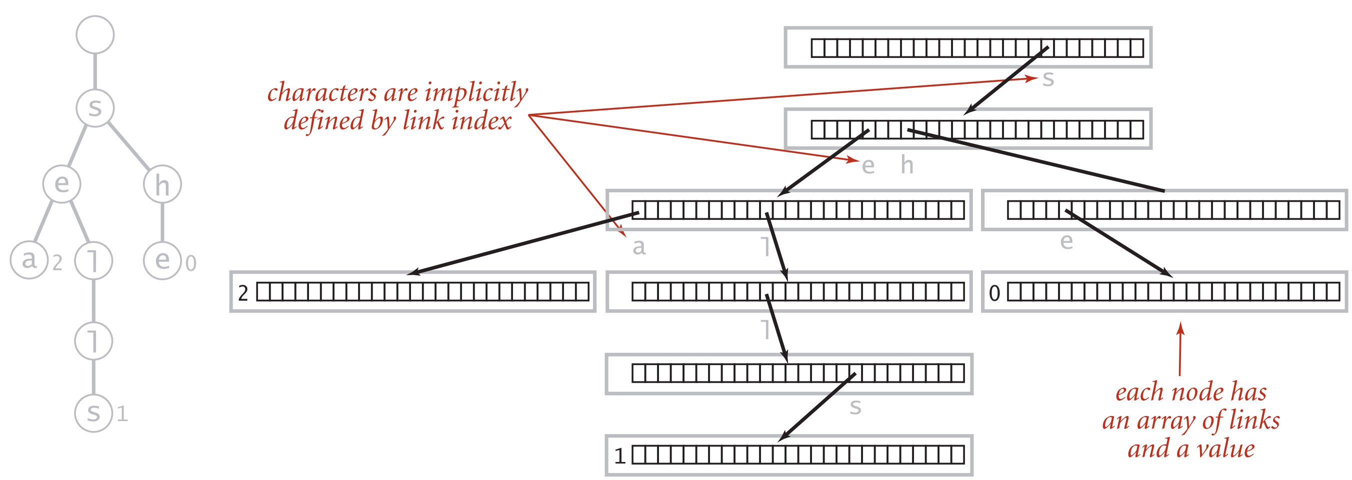
Trie representation

Trie representation: Java implementation

Node. A value, plus references to R nodes.

```
private static class Node
{
   private Object value;
   private Node[] next = new Node[R];
}
```

use `Object` instead of `Value` since no generic array creation in Java



Trie representation (R = 26)

R-way trie: Java implementation

```
public class TrieST<Value>
{
   private static final int R = 256;        extended ASCII
   private Node root;

   private static class Node
   {  /* see previous slide */   }

   public void put(String key, Value val)
   {  root = put(root, key, val, 0);   }

   private Node put(Node x, String key, Value val, int d)
   {
      if (x == null) x = new Node();
      if (d == key.length()) { x.val = val; return x; }
      char c = key.charAt(d);
      x.next[c] = put(x.next[c], key, val, d+1);
      return x;
   }

}
```

R-way trie: Java implementation (continued)

```
public boolean contains(String key)
{  return get(key) != null;  }

public Value get(String key)
{
   Node x = get(root, key, 0);
   if (x == null) return null;
   return (Value) x.val;
}

private Node get(Node x, String key, int d)
{
   if (x == null) return null;
   if (d == key.length()) return x;
   char c = key.charAt(d);
   return get(x.next[c], key, d+1);
}
```

Trie performance

Search miss.

- Could have mismatch on first character.
- Typical case: examine only a few characters.

Search hit. Need to examine all L characters for equality.

Space. R null links at each leaf.
(but sublinear space possible if many short strings share common prefixes)

Bottom line. Fast search hit, sublinear-time search miss, wasted space.

String symbol table implementations cost summary

	character accesses (typical case)				dedup	
implementation	search hit	search miss	insert	space (links)	`moby.txt`	`actors.txt`
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4N$	1.40	97.4
hashing	L	L	L	$4N$ to $16N$	0.76	40.6
R-way trie	L	$\log_R N$	L	$(R+1) N$	1.12	out of memory

R-way trie.

- Method of choice for small R.
- Too much memory for large R.

Challenge. Use less memory, e.g., 65,536-way trie for Unicode!

Digression: out of memory?

> *" 640 K ought to be enough for anybody. "*
> *— attributed to Bill Gates, 1981*
> *(commenting on the amount of RAM in personal computers)*

> *" 64 MB of RAM may limit performance of some Windows XP features; therefore, 128 MB or higher is recommended for best performance. "* *— Windows XP manual, 2002*

> *" 64 bit is coming to desktops, there is no doubt about that. But apart from Photoshop, I can't think of desktop applications where you would need more than 4GB of physical memory, which is what you have to have in order to benefit from this technology. Right now, it is costly. "* *— Bill Gates, 2003*

Digression: out of memory?

A short (approximate) history.

machine	year	address bits	addressable memory	typical actual memory	cost
PDP-8	1960s	12	6 KB	6 KB	$16K
PDP-10	1970s	18	256 KB	256 KB	$1M
IBM S/360	1970s	24	4 MB	512 KB	$1M
VAX	1980s	32	4 GB	1 MB	$1M
Pentium	1990s	32	4 GB	1 GB	$1K
Xeon	2000s	64	enough	4 GB	$100
??	future	128+	enough	enough	$1

" 512-bit words ought to be enough for anybody. "
— RS, 1995

A modest proposal

Number of atoms in the universe (estimated). $\leq 2^{266}$.

Age of universe (estimated). 14 billion years ~ 2^{59} seconds $\leq 2^{89}$ nanoseconds.

Q. How many bits address every atom that ever existed?

A. Use a unique 512-bit address for every atom at every time quantum.

266 bits	89 bits	157 bits
atom	time	cushion for whatever

Ex. Use 256-way trie to map atom to location.

- Represent atom as 64 8-bit chars (512 bits).
- 256-way trie wastes 255/256 actual memory.
- Need better use of memory.

- tries
- **TSTs**
- string symbol table API

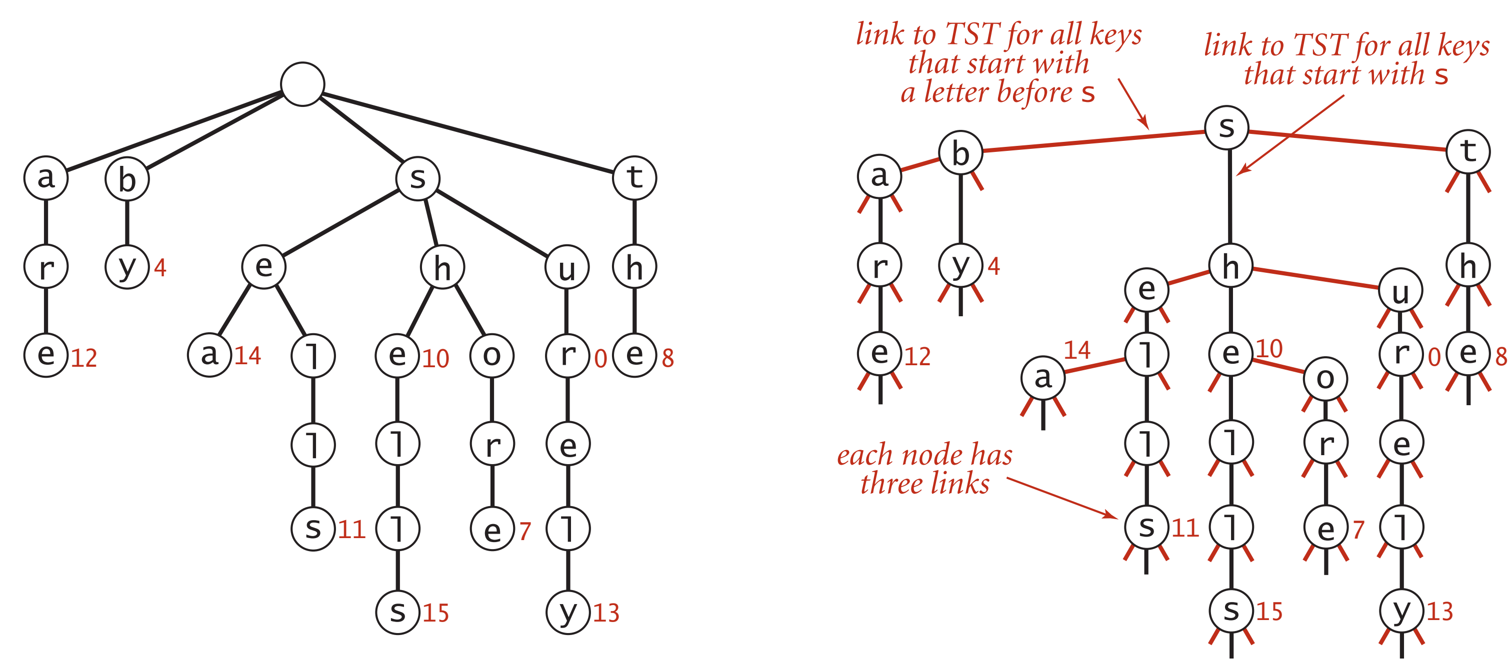
Ternary search tries

TST. [Bentley-Sedgewick, 1997]

- Store characters and values in nodes (not keys).
- Each node has three children: smaller (left), equal (middle), larger (right).

TST. [Bentley-Sedgewick, 1997]

- Store characters and values in nodes (not keys).
- Each node has three children: smaller (left), equal (middle), larger (right).



TST representation of a trie

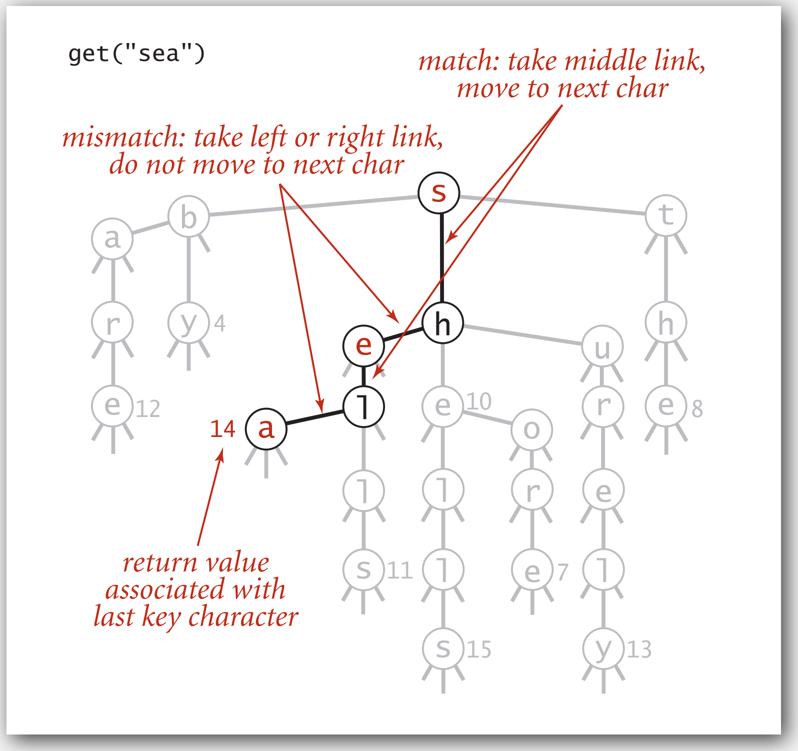
Search in a TST

Follow links corresponding to each character in the key.

- If less, take left link; if greater, take right link.
- If equal, take the middle link and move to the next key character.

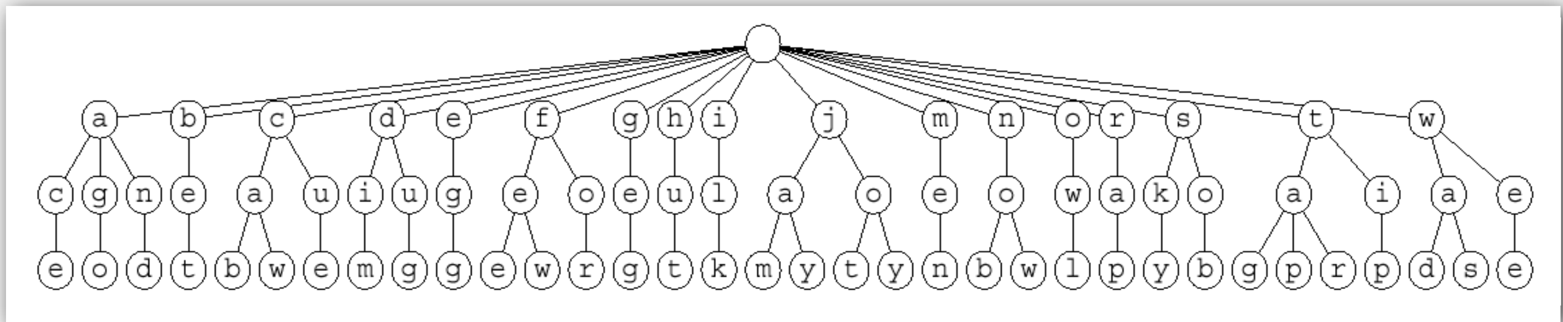
Search hit. Node where search ends has a non-null value.

Search miss. Reach a null link or node where search ends has null value.



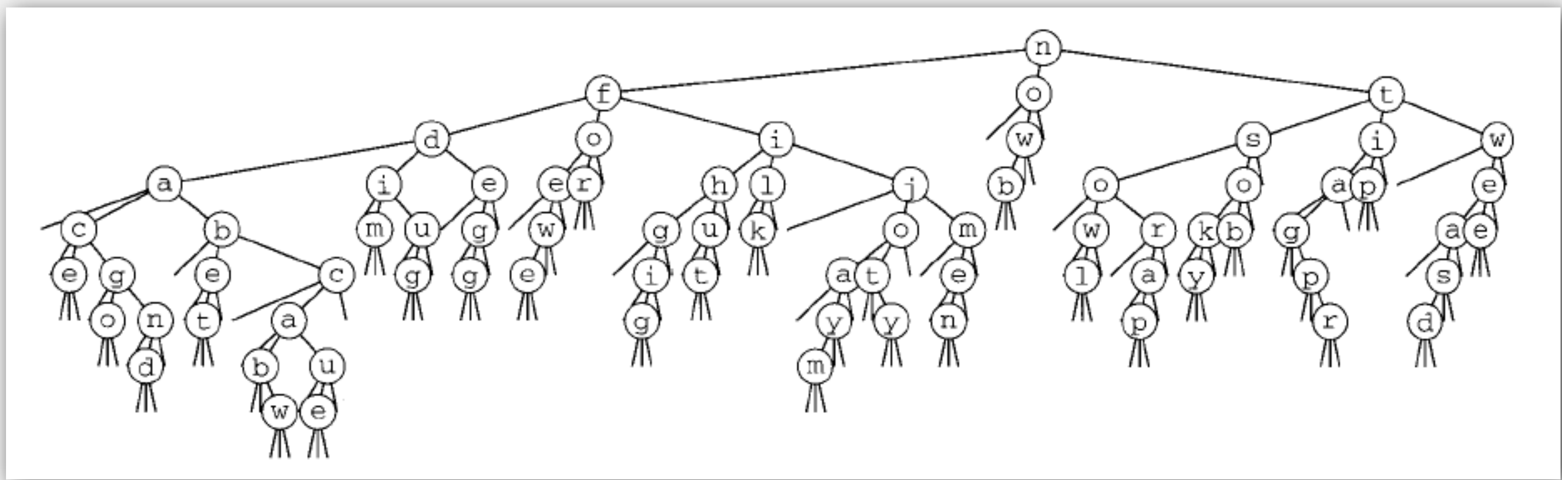
26-way trie vs. TST

26-way trie. 26 null links in each leaf.



26-way trie (1035 null links, not shown)

TST. 3 null links in each leaf.



TST (155 null links)

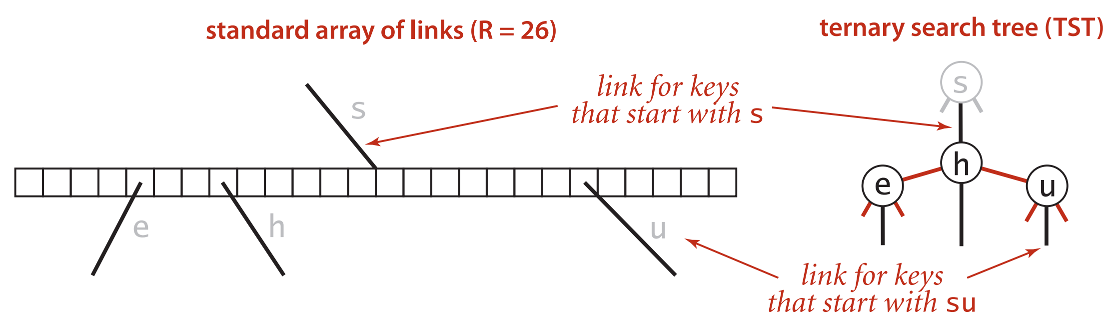
now
for
tip
ilk
dim
tag
jot
sob
nob
sky
hut
ace
bet
men
egg
few
jay
owl
joy
rap
gig
wee
was
cab
wad
caw
cue
fee
tap
ago
tar
jam
dug
and

TST representation in Java

A TST node is five fields:

- A value.
- A character c.
- A reference to a left TST.
- A reference to a middle TST.
- A reference to a right TST.

```
private class Node
{
   private Value val;
   private char c;
   private Node left, mid, right;
}
```



Trie node representations

TST: Java implementation

```
public class TST<Value>
{
   private Node root;

   private class Node
   {  /* see previous slide */  }

   public void put(String key, Value val)
   {  root = put(root, key, val, 0);  }

   private Node put(Node x, String key, Value val, int d)
   {
      char c = s.charAt(d);
      if (x == null) {  x = new Node(); x.c = c;  }
      if      (c < x.c)             x.left  = put(x.left,  key, val, d);
      else if (c > x.c)             x.right = put(x.right, key, val, d);
      else if (d < s.length() - 1) x.mid   = put(x.mid,   key, val, d+1);
      else                          x.val   = val;
      return x;
   }

}
```

TST: Java implementation (continued)

```
public boolean contains(String key)
{  return get(key) != null;  }

public Value get(String key)
{
   Node x = get(root, key, 0);
   if (x == null) return null;
   return x.val;
}

private Node get(Node x, String key, int d)
{
   if (x == null) return null;
   char c = s.charAt(d);
   if      (c < x.c)              return get(x.left,  key, d);
   else if (c > x.c)              return get(x.right, key, d);
   else if (d < key.length() - 1) return get(x.mid,   key, d+1);
   else                           return x;
}
```

String symbol table implementation cost summary

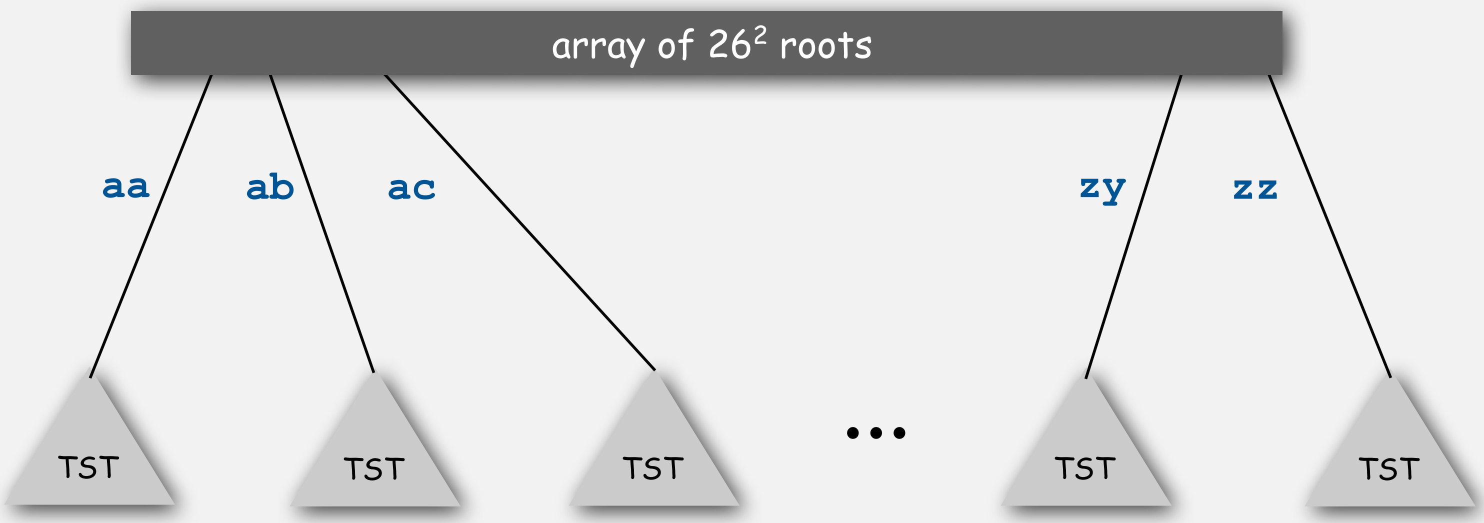
	character accesses (typical case)				dedup	
implementation	search hit	search miss	insert	space (links)	`moby.txt`	`actors.txt`
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4N$	1.40	97.4
hashing	L	L	L	$4N$ to $16N$	0.76	40.6
R-way trie	L	$\log_R N$	L	$(R+1)N$	1.12	out of memory
TST	$L + \ln N$	$\ln N$	$L + \ln N$	$4N$	0.72	38.7

Remark. Can build balanced TSTs via rotations to achieve $L + \log N$ worst-case guarantees.

Bottom line. TST is as fast as hashing (for string keys), space efficient.

Hybrid of R-way trie and TST.

- Do R^2-way branching at root.
- Each of R^2 root nodes points to a TST.



Q. What about one- and two-letter words?

String symbol table implementation cost summary

	character accesses (typical case)				dedup	
implementation	search hit	search miss	insert	space (links)	`moby.txt`	`actors.txt`
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4N$	1.40	97.4
hashing	L	L	L	$4N$ to $16N$	0.76	40.6
R-way trie	L	$\log_R N$	L	$(R+1)N$	1.12	out of memory
TST	$L + \ln N$	$\ln N$	$L + \ln N$	$4N$	0.72	38.7
TST with R^2	$L + \ln N$	$\ln N$	$L + \ln N$	$4N + R^2$	0.51	32.7

TST vs. hashing

Hashing.

- Need to examine entire key.
- Search hits and misses cost about the same.
- Need good hash function for every key type.
- No help for ordered symbol table operations.

TSTs.

- Works only for strings (or digital keys).
- Only examines just enough key characters.
- Search miss may only involve a few characters.
- Can handle ordered symbol table operations (plus others!).

Bottom line. TSTs are:

- Faster than hashing (especially for search misses).
  More flexible than red-black trees (next).

- tries
- TSTs
- **string symbol table API**

String symbol table API

Character-based operations. The string symbol table API supports several useful character-based operations.

```
by sea sells she shells shore the
```

Prefix match. The keys with prefix **"sh"** are **"she"**, **"shells"**, and **"shore"**.

Longest prefix. The key that is the longest prefix of **"shellsort"** is **"shells"**.

Wildcard match. The key that match **".he"** are **"she"** and **"the"**.

String symbol table API

`public class StringST<Value>`

	`StringST()`	*create a symbol table with string keys*
	`StringST(Alphabet alpha)`	*create a symbol table with string keys whose characters are taken from* `alpha`*.*
`void`	`put(String key, Value val)`	*put key-value pair into the symbol table (remove* `key` *from table if value is* `null`*)*
`Value`	`get(String key)`	*value paired with* `key` *(*`null` *if* `key` *is absent)*
`void`	`delete(String key)`	*remove* `key` *(and its value) from table*
`boolean`	`contains(String key)`	*is there a value paired with* `key`*?*
`boolean`	`isEmpty()`	*is the table empty?*
`String`	`longestPrefixOf(String s)`	*return the longest key that is a prefix of* `s`
`Iterable<String>`	`keysWithPrefix(String s)`	*all the keys having* `s` *as a prefix.*
`Iterable<String>`	`keysThatMatch(String s)`	*all the keys that match* `s` *(where* `.` *matches any character).*
`int`	`size()`	*number of key-value pairs in the table*
`Iterable<String>`	`keys()`	*all the keys in the symbol table*

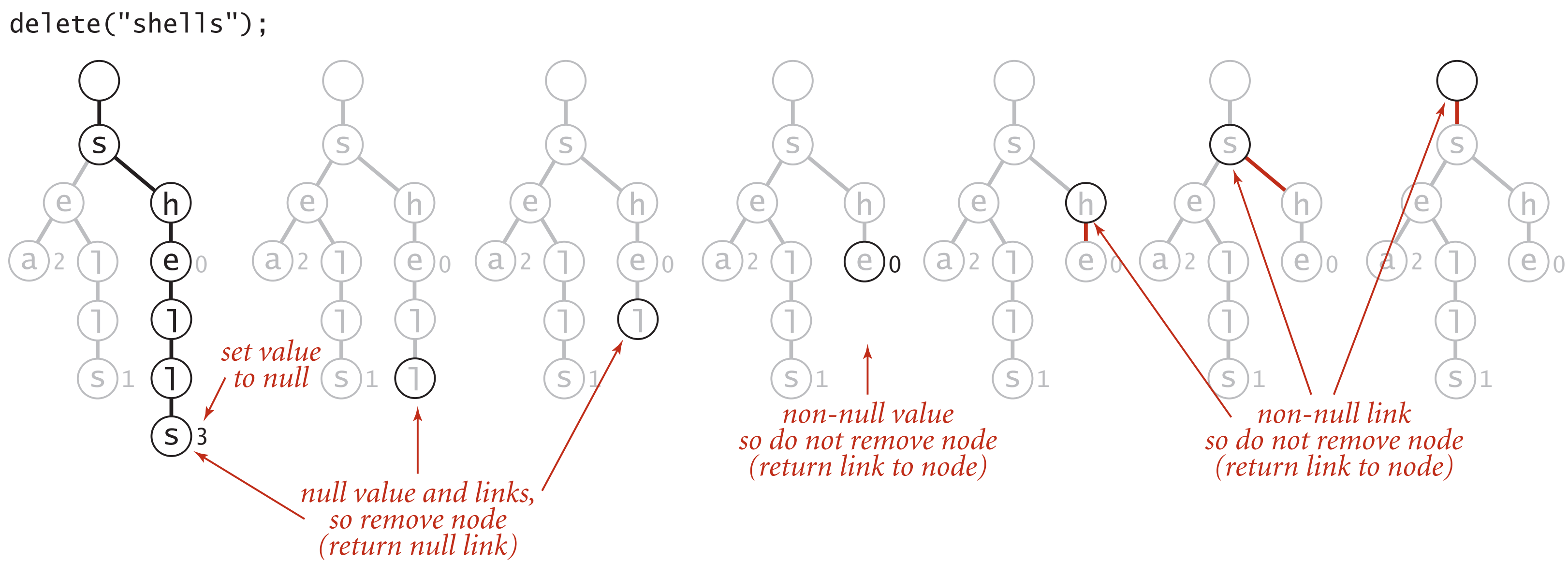
API for a symbol table with string keys

Remark. Can also add other ordered ST methods, e.g., **`floor()`** and **`rank()`**.

Deletion in an R-way trie

To delete a key-value pair:

- Find the node corresponding to key and set value to null.
- If that node has all null links, remove that node (and recur).



Deleting a key (and its associated value) from a trie

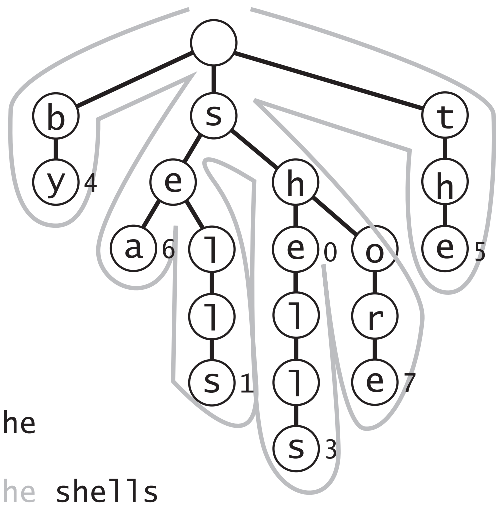
Ordered iteration

To iterate through all keys in sorted order:

- Do inorder traversal of trie; add keys encountered to a queue.
- Maintain sequence of characters on path from root to node.

```
keysWithPrefix("");
```

key	q
b	
by	by
s	
se	
sea	by sea
sel	
sell	
sells	by sea sells
sh	
she	by sea sells she
shell	
shells	by sea sells she shells
sho	
shor	
shore	by sea sells she shells shore
t	
th	
the	by sea sells she shells shore the



Collecting the keys in a trie (trace)

Ordered iteration: Java implementation

To iterate through all keys in sorted order:

- Do inorder traversal of trie; add keys encountered to a queue.
- Maintain sequence of characters on path from root to node.

```
public Iterable<String> keys()
{
   Queue<String> queue = new Queue<String>();
   collect(root, "", queue);
   return queue;
}

private void collect(Node x, String prefix, Queue<String> q)
{
   if (x == null) return;
   if (x.val != null) q.enqueue(prefix);
   for (char c = 0; c < R; c++)
      collect(x.next[c], prefix + c, q);
}
```

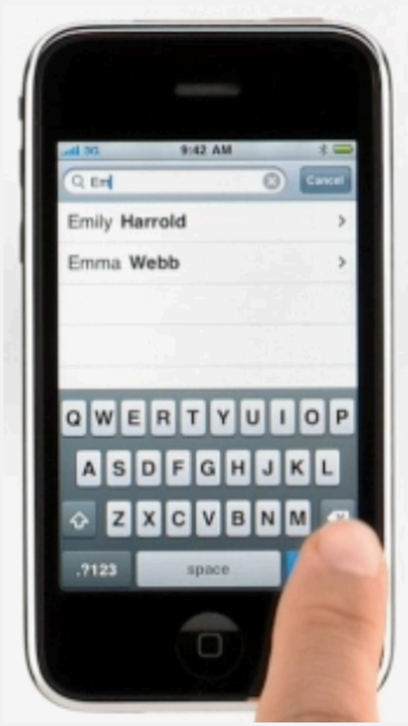
sequence of characters on path from root to x

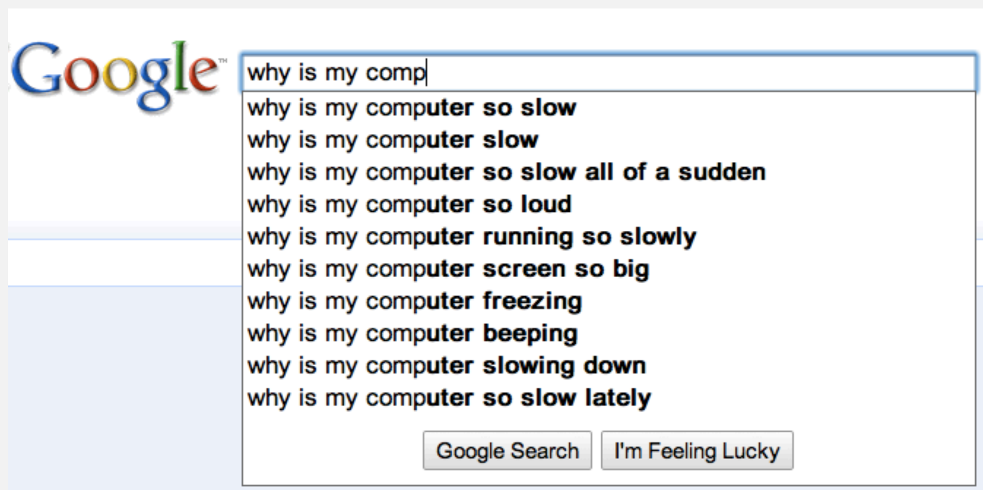
Prefix matches

Find all keys in symbol table starting with a given prefix.

Ex. Autocomplete in a cell phone, search bar, text editor, or shell.

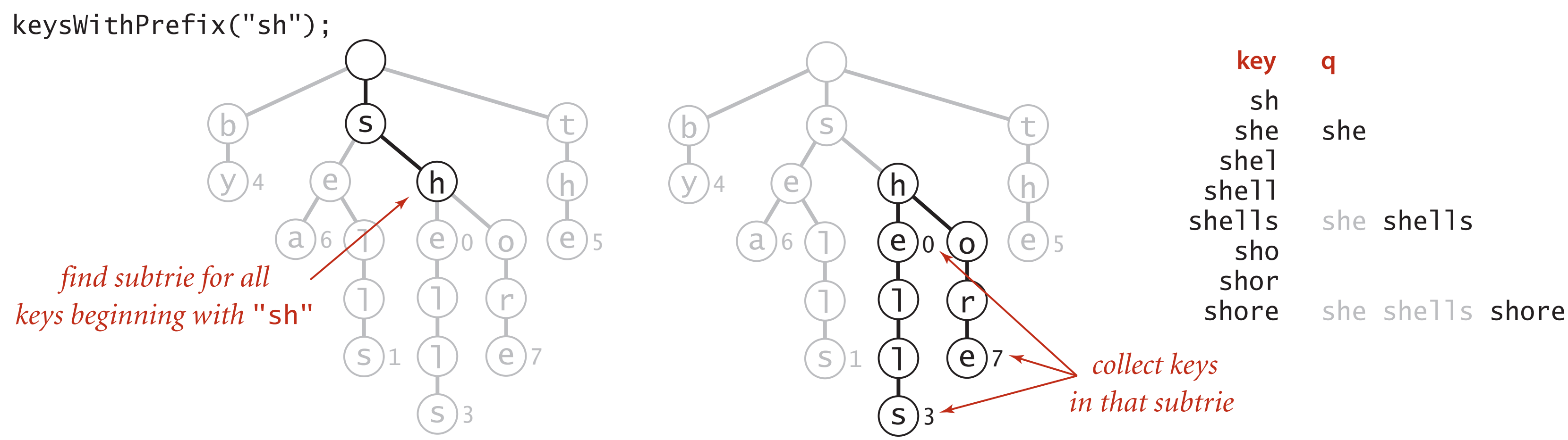
- User types characters one at a time.
- System reports all matching strings.





Prefix matches

Find all keys in symbol table starting with a given prefix.



Prefix match in a trie

```
public Iterable<String> keysWithPrefix(String prefix)
{
   Queue<String> queue = new Queue<String>();
   Node x = get(root, prefix, 0);
   collect(x, prefix, queue);
   return queue;
}
```

root of subtrie for all strings beginning with given prefix

Longest prefix

Find longest key in symbol table that is a prefix of query string.

Ex. Search IP database for longest prefix matching destination IP, and route packets accordingly.

```
"128"
"128.112"
"128.112.055"
"128.112.055.15"
"128.112.136"
"128.112.155.11"
"128.112.155.13"
"128.222"
"128.222.136"

prefix("128.112.136.11") = "128.112.136"
prefix("128.166.123.45") = "128"
```

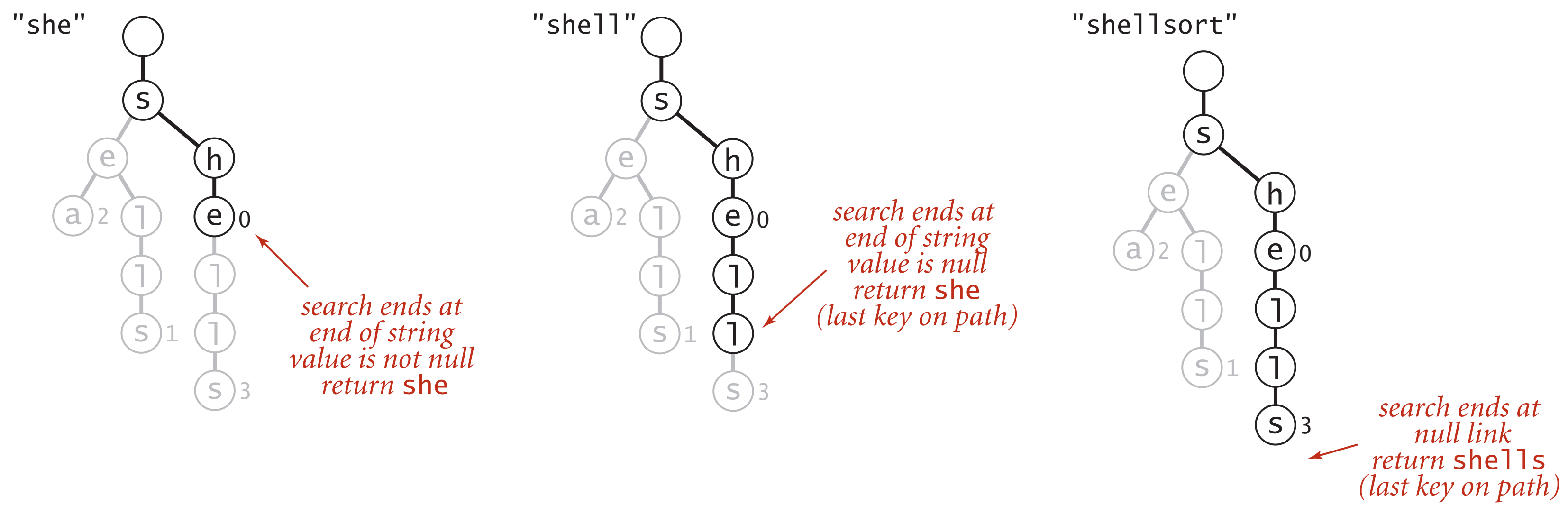
represented as 32-bit binary number for IPv4 (instead of string)

Q. Why isn't longest prefix match the same as floor or ceiling?

Longest prefix

Find longest key in symbol table that is a prefix of query string.

- Search for query string.
- Keep track of longest key encountered.



Possibilities for `longestPrefixOf()`

Longest prefix: Java implementation

Find longest key in symbol table that is a prefix of query string.

- Search for query string.
- Keep track of longest key encountered.

```
public String longestPrefixOf(String query)
{
   int length = search(root, query, 0, 0);
   return query.substring(0, length);
}

private int search(Node x, String query, int d, int length)
{
   if (x == null) return length;
   if (x.val != null) length = d;
   if (d == query.length()) return length;
   char c = query.charAt(d);
   return search(x.next[c], query, d+1, length);
}
```

T9 texting

Goal. Type text messages on a phone keypad.

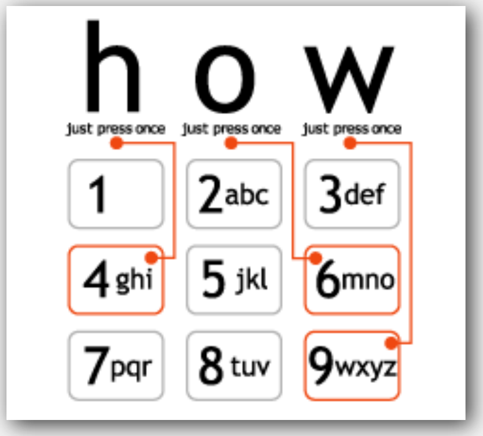
Multi-tap input. Enter a letter by repeatedly pressing a key until the desired letter appears.

T9 text input. ["A much faster and more fun way to enter text."]

- Find all words that correspond to given sequence of numbers.
- Press 0 to see all completion options.

Ex. **hello**

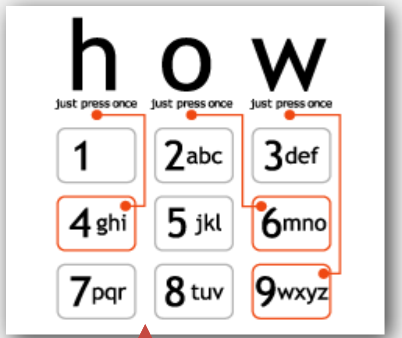
- Multi-tap: `4 4 3 3 5 5 5 5 5 5 6 6 6`
- T9: `4 3 5 5 6`



www.t9.com

A Letter to t9.com

```
To: info@t9support.com
Date: Tue, 25 Oct 2005 14:27:21 -0400 (EDT)

Dear T9 texting folks,

I enjoyed learning about the T9 text system
from your webpage, and used it as an example
in my data structures and algorithms class.
However, one of my students noticed a bug
in your phone keypad

   http://www.t9.com/images/how.gif

Somehow, it is missing the letter s. (!)

Just wanted to bring this information to
your attention and thank you for your website.

Regards,

Kevin
```



where's the "s" ??

A world without "s" ??

```
To: "'Kevin Wayne'" <wayne@CS.Princeton.EDU>
Date: Tue, 25 Oct 2005 12:44:42 -0700

Thank you Kevin.

I am glad that you find T9 o valuable for your
cla.  I had not noticed thi before.  Thank for
writing in and letting u know.

Take care,

Brooke nyder
OEM Dev upport
AOL/Tegic Communication
1000 Dexter Ave N. uite 300
eattle, WA 98109

ALL INFORMATION CONTAINED IN THIS EMAIL IS CONSIDERED
CONFIDENTIAL AND PROPERTY OF AOL/TEGIC COMMUNICATIONS
```

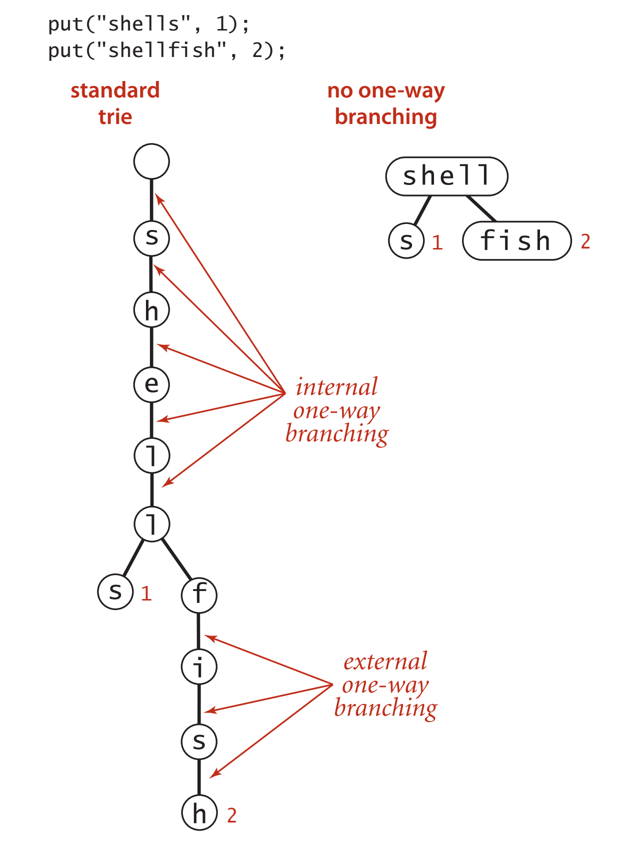
Compressing a trie

Collapsing 1-way branches at bottom.

Internal node stores character; leaf node stores suffix (or full key).

Collapsing interior 1-way branches.

Node stores a sequence of characters.

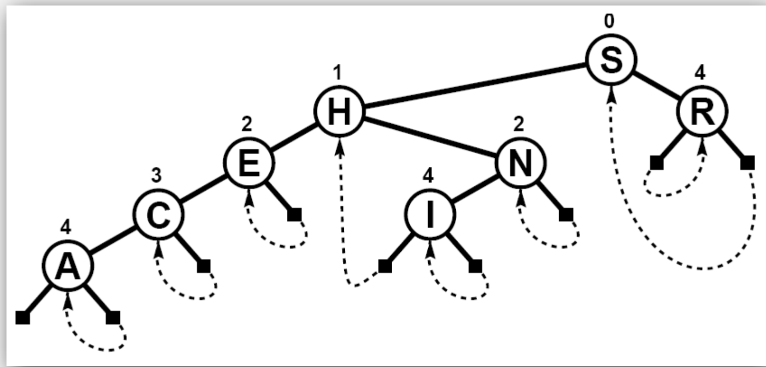


Removing one-way branching in a trie

A classic algorithm

Patricia tries. [Practical Algorithm to Retrieve Information Coded in Alphanumeric]

- Collapse one-way branches in binary trie.
- Thread trie to eliminate multiple node types.

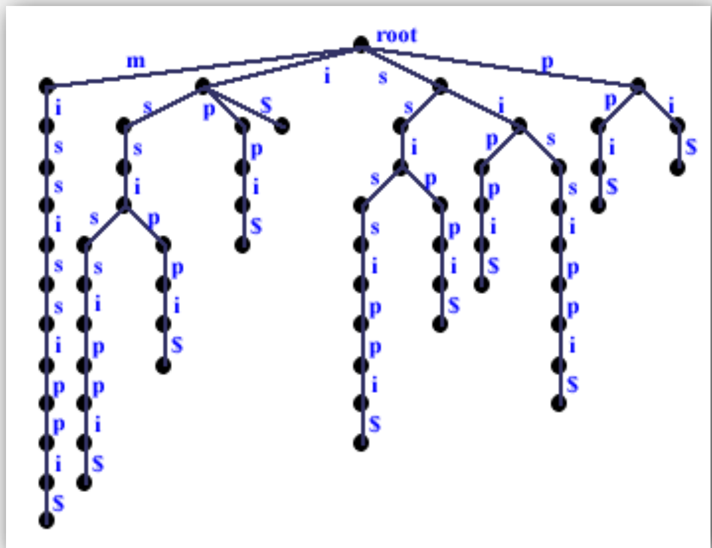


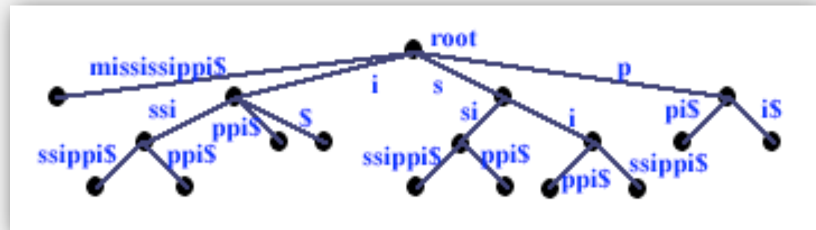
Applications.

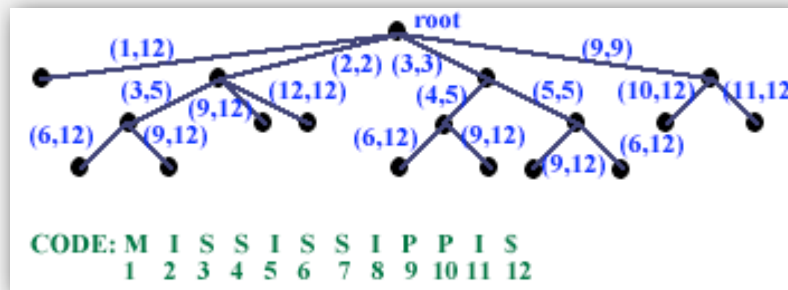
- Database search.
- P2P network search.
- IP routing tables: find longest prefix match.
- Compressed quad-tree for N-body simulation.
- Efficiently storing and querying XML documents.

Implementation. One step beyond this lecture.

Suffix tree. Threaded trie with collapsed 1-way branching for string suffixes.







Applications.

- Linear-time longest repeated substring.
- Computational biology databases (BLAST, FASTA).

Implementation. One step beyond this lecture.

String symbol tables summary

A success story in algorithm design and analysis.

Red-black tree.

- Performance guarantee: log N key compares.
- Supports ordered symbol table API.

Hash tables.

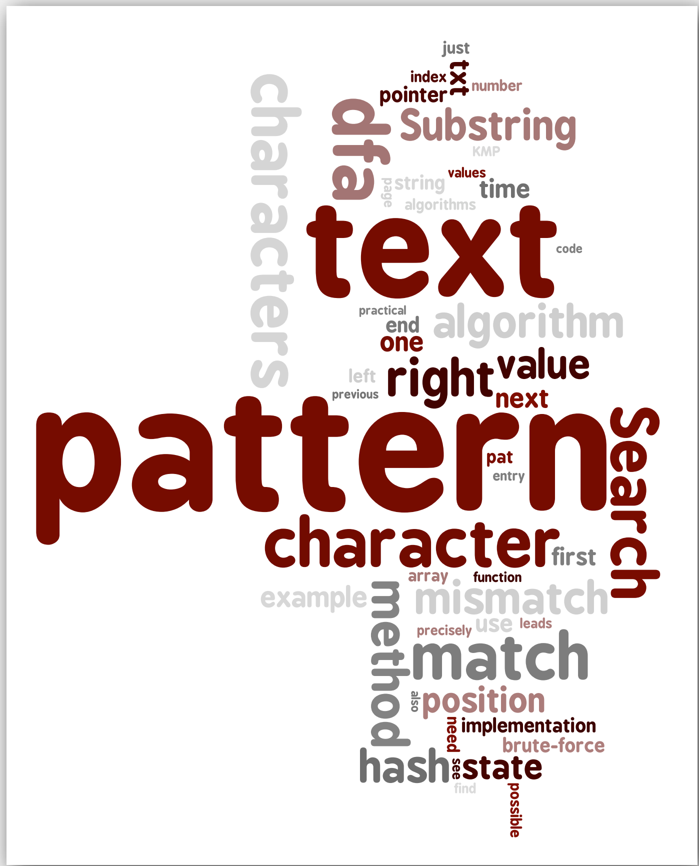
- Performance guarantee: constant number of probes.
- Requires good hash function for key type.

Tries. R-way, TST.

- Performance guarantee: log N characters accessed.
- Supports extensions to API based on partial keys.

Bottom line. You can get at anything by examining 50-100 bits (!!!)

5.3 Substring Search



- brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp

Substring search

Goal. Find pattern of length M in a text of length N.

typically N >> M

pattern → N E E D L E

text → I N A H A Y S T A C K N E E D L E I N A

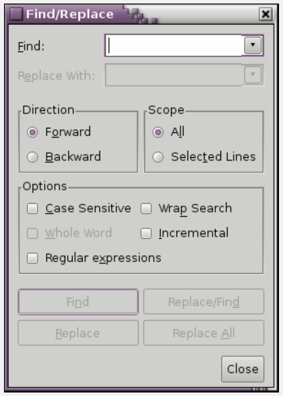
match

Computer forensics. Search memory or disk for signatures, e.g., all URLs or RSA keys that the user has entered.

http://citp.princeton.edu/memory

Applications

- Parsers.
- Spam filters.
- Digital libraries.
- Screen scrapers.
- Word processors.
- Web search engines.
- Electronic surveillance.
- Natural language processing.
- Computational molecular biology.
- FBIs Digital Collection System 3000.
- Feature detection in digitized images.
- ...

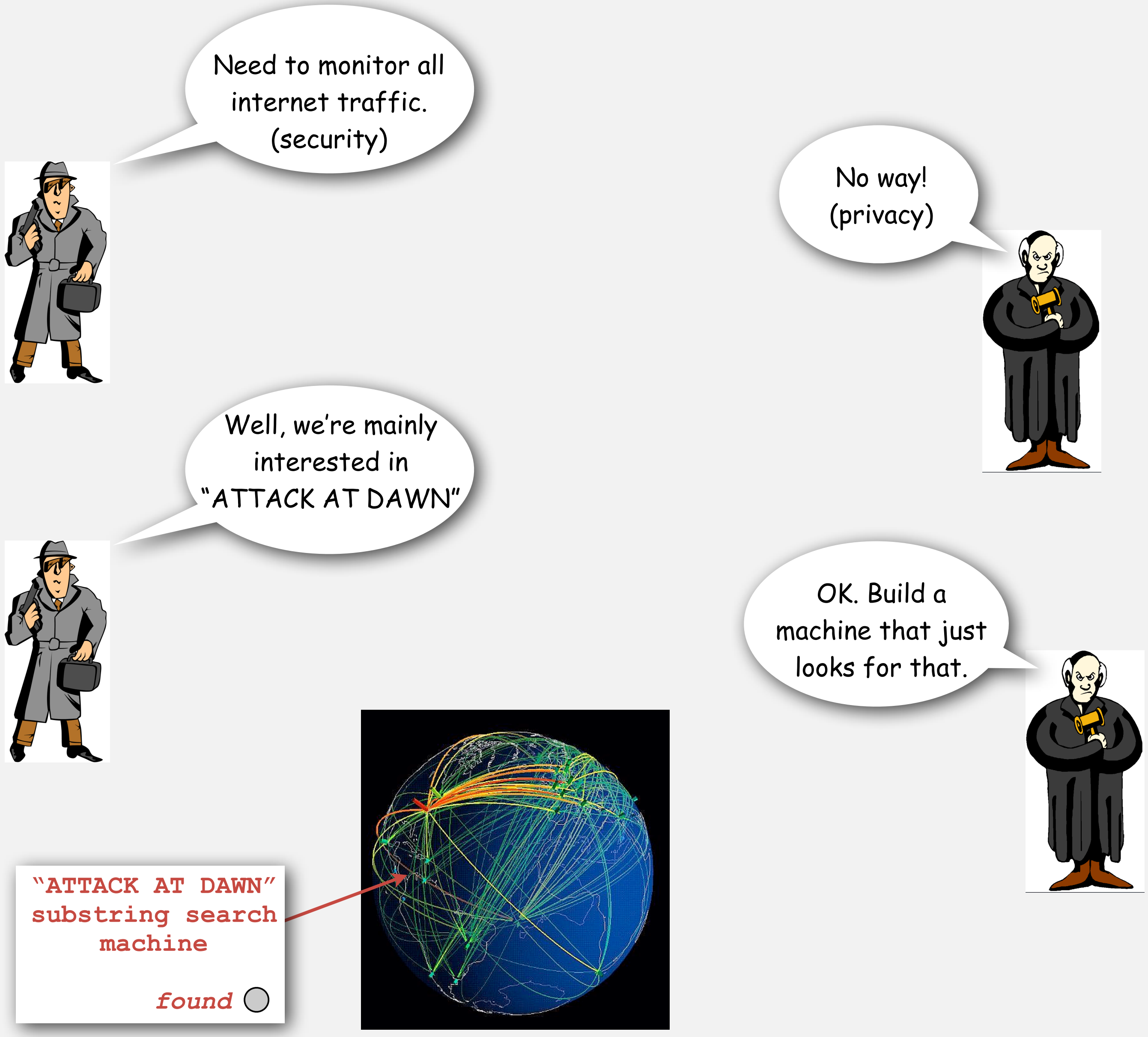


Application: Spam filtering

Identify patterns indicative of spam.

- `PROFITS`
- `L0SE WE1GHT`
- `herbal Viagra`
- `There is no catch.`
- `L0W M0RTGAGE RATES`
- `This is a one-time mailing.`
- `This message is sent in compliance with spam regulations.`
- `You're getting this message because you registered with one of our marketing partners.`

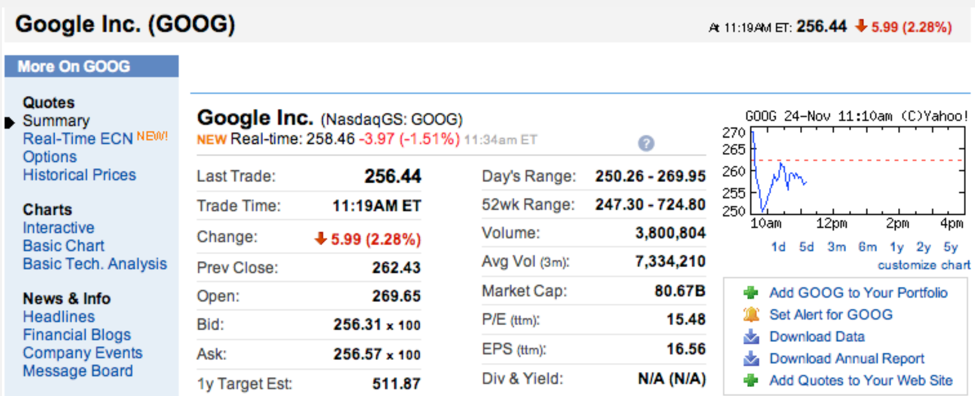
Application: Electronic surveillance



Application: Screen scraping

Goal. Extract relevant data from web page.

Ex. Find string delimited by and after first occurrence of pattern **Last Trade:**.



`http://finance.yahoo.com/q?s=goog`

```
...
<tr>
<td class= "yfnc_tablehead1"
width= "48%">
Last Trade:
</td>
<td class= "yfnc_tabledata1">
<big><b>452.92</b></big>
</td></tr>
<td class= "yfnc_tablehead1"
width= "48%">
Trade Time:
</td>
<td class= "yfnc_tabledata1">
...
```

Screen scraping: Java implementation

Java library. The `indexOf()` method in Java's string library returns the index of the first occurrence of a given string, starting at a given offset.

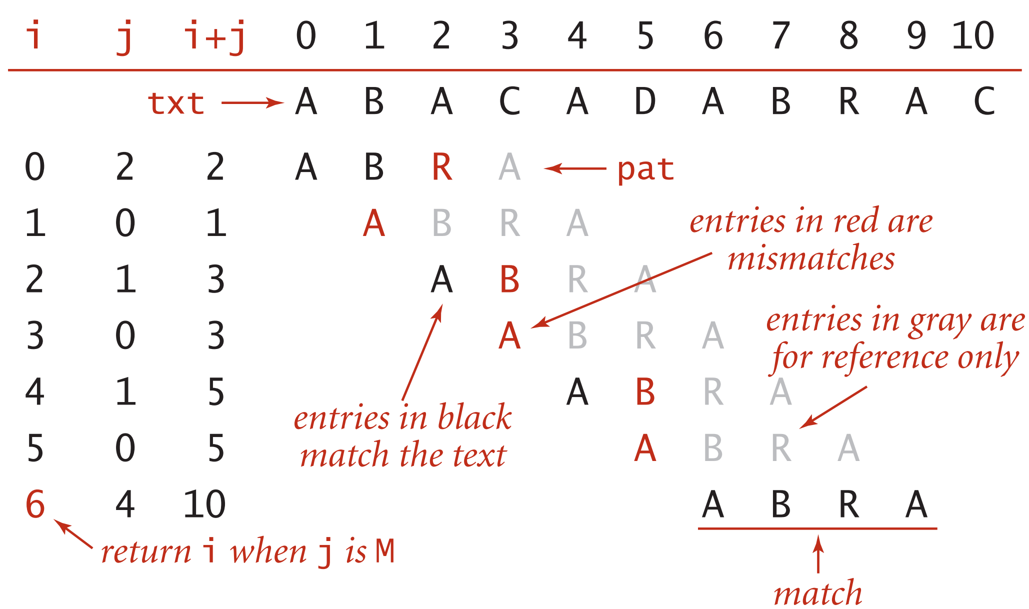
```
public class StockQuote
{
   public static void main(String[] args)
   {
      String name = "http://finance.yahoo.com/q?s=";
      In in = new In(name + args[0]);
      String text = in.readAll();
      int start    = text.indexOf("Last Trade:", 0);
      int from     = text.indexOf("<b>",  start);
      int to       = text.indexOf("</b>", from);
      String price = text.substring(from + 3, to);
      StdOut.println(price);
   }
}
```

```
% java StockQuote goog
256.44

% java StockQuote msft
19.68
```

- brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp

Brute-force substring search

Check for pattern starting at each text position.



Brute-force substring search

Brute-force substring search: Java implementation

Check for pattern starting at each text position.

```
public static int search(String pat, String txt)
{
   int M = pat.length();
   int N = txt.length();
   for (int i = 0; i <= N - M; i++)
   {
      int j;
      for (j = 0; j < M; j++)
         if (txt.charAt(i+j) != pat.charAt(j))
            break;
      if (j == M) return i;   ← index in text where pattern starts
   }
   return N;   ← not found
}
```

Brute-force substring search: worst case

Brute-force algorithm can be slow if text and pattern are repetitive.

i	j	i+j	0	1	2	3	4	5	6	7	8	9
		txt →	A	A	A	A	A	A	A	A	A	B
0	4	4	A	A	A	A	B ← pat					
1	4	5		A	A	A	A	B				
2	4	6			A	A	A	A	B			
3	4	7				A	A	A	A	B		
4	4	8					A	A	A	A	B	
5	4	9						A	A	A	A	B

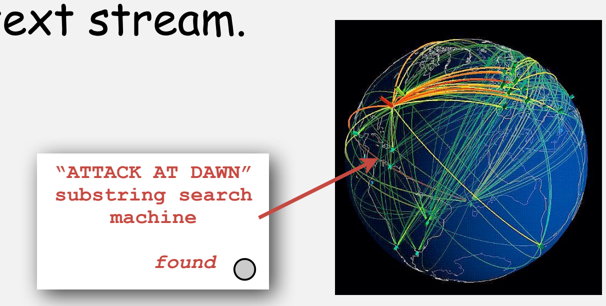
Brute-force substring search (worst case)

Worst case. ~ M N char compares.

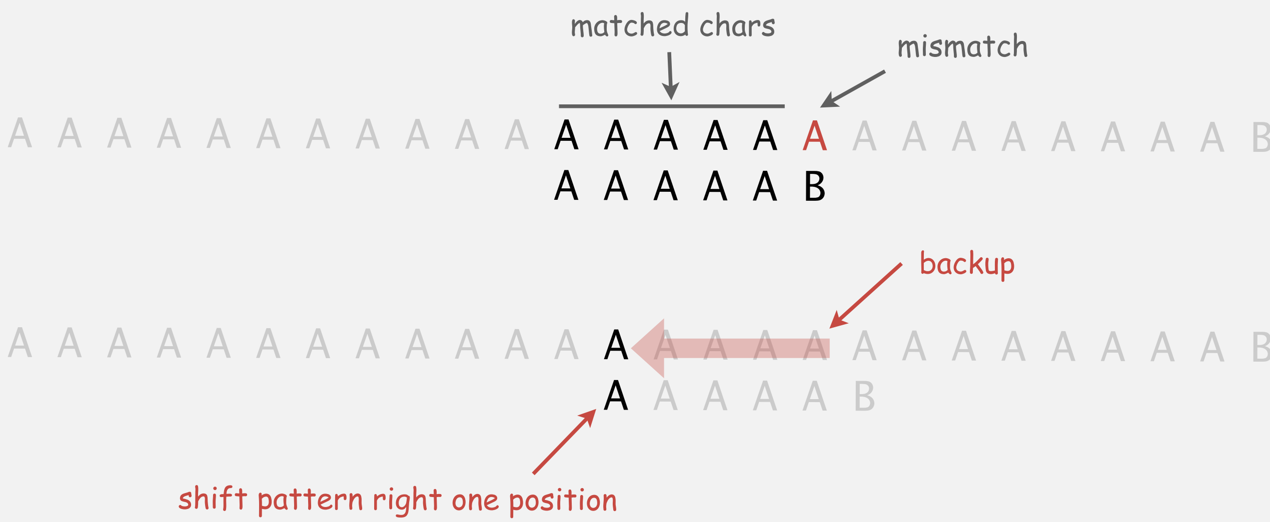
Backup

In typical applications, we want to avoid backup in text stream.

- Treat input as stream of data.
- Abstract model: `StdIn`.



Brute-force algorithm needs backup for every mismatch



Approach 1. Maintain buffer of size `M` (build backup into `StdIn`)

Approach 2. Stay tuned.

Brute-force substring search: alternate implementation

Same sequence of char compares as previous implementation.

- `i` points to end of sequence of already-matched chars in text.
- `j` stores number of already-matched chars (end of sequence in pattern).

```
public static int search(String pat, String txt)
{
   int i, N = txt.length();
   int j, M = pat.length();
   for (i = 0, j = 0; i < N && j < M; i++)
   {
      if (txt.charAt(i) == pat.charAt(j)) j++;
      else { i -= j; j = 0;  }
   }
   if (j == M) return i - M;
   else             return N;
}
```

backup

Algorithmic challenges in substring search

Brute-force is often not good enough.

Theoretical challenge. Linear-time guarantee. ← fundamental algorithmic problem

Practical challenge. Avoid backup in text stream. ← often no room or time to save text

```
Now is the time for all people to come to the aid of their party. Now is the time for all good people to
come to the aid of their party. Now is the time for many good people to come to the aid of their party.
Now is the time for all good people to come to the aid of their party. Now is the time for a lot of good
people to come to the aid of their party. Now is the time for all of the good people to come to the aid of
their party. Now is the time for all good people to come to the aid of their party. Now is the time for
each good person to come to the aid of their party. Now is the time for all good people to come to the aid
of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the
time for all good people to come to the aid of their party. Now is the time for many or all good people to
come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now
is the time for all good Democrats to come to the aid of their party. Now is the time for all people to
come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now
is the time for many good people to come to the aid of their party. Now is the time for all good people to
come to the aid of their party. Now is the time for a lot of good people to come to the aid of their
party. Now is the time for all of the good people to come to the aid of their party. Now is the time for
all good people to come to the aid of their attack at dawn party. Now is the time for each person to come
to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is
the time for all good Republicans to come to the aid of their party. Now is the time for all good people
to come to the aid of their party. Now is the time for many or all good people to come to the aid of their
party. Now is the time for all good people to come to the aid of their party. Now is the time for all good
Democrats to come to the aid of their party.
```

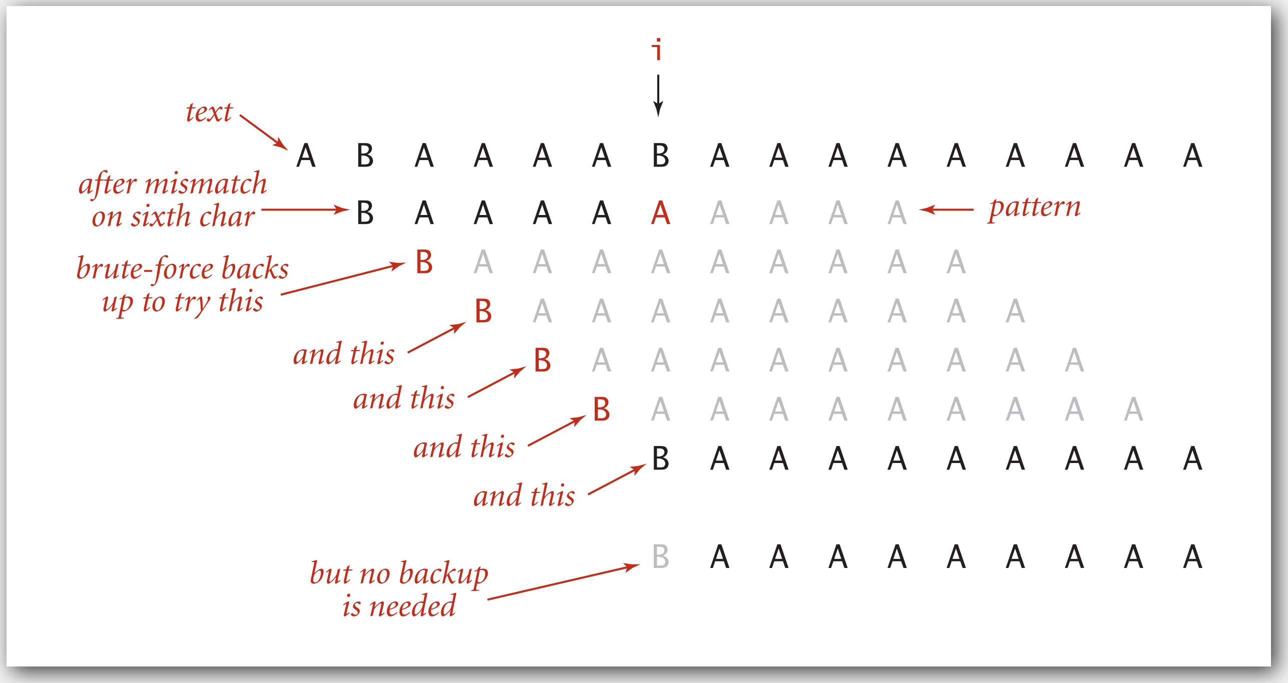
- brute force
- **Knuth-Morris-Pratt**
- Boyer-Moore
- Rabin-Karp

Knuth-Morris-Pratt substring search

Intuition. Suppose we are searching in text for pattern **BAAAAAAAAA**.

- Suppose we match 5 chars in pattern, with mismatch on 6^{th} char.
- We know previous 6 chars in text are **BAAAAB**.
- Don't need to back up text pointer!

assuming {A, B} alphabet

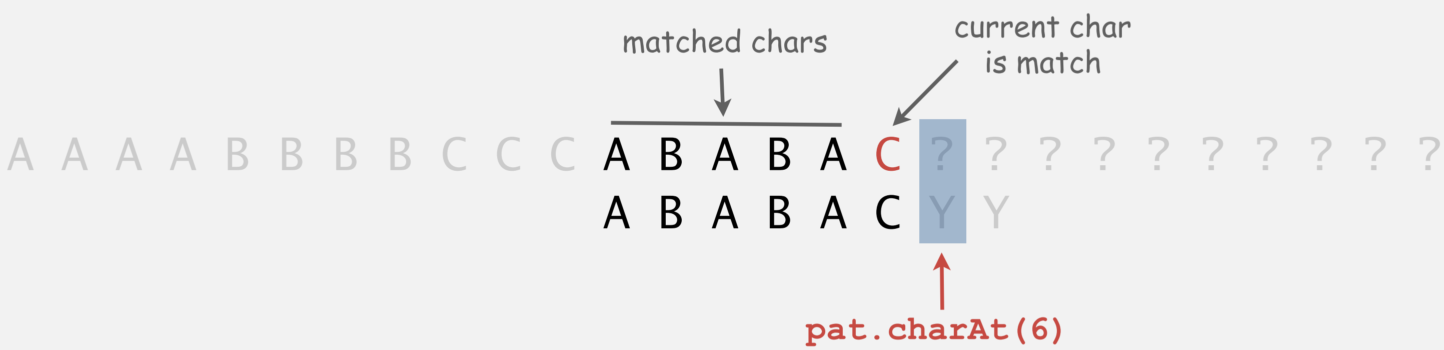


Remark. It is always possible to avoid backup (!)

KMP substring search preprocessing (concept)

Q. What pattern char do we compare to the next text char on match?

A. Easy: compare next pattern char to next text char.



	j	0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

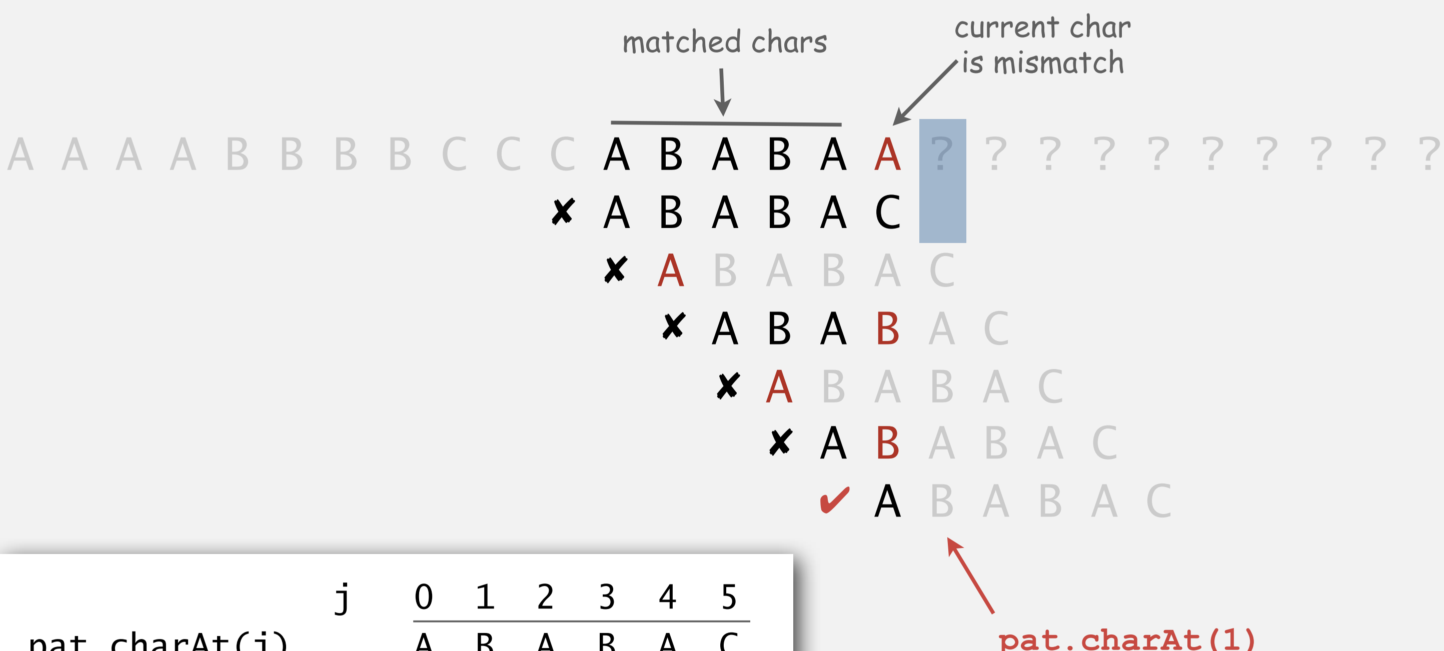
table giving pattern char to compare to the next text char

current text char: `c`
current pattern index: `j`
next pattern index: `dfa[c][j]`

KMP substring search preprocessing (concept)

Q. What pattern char do we compare to the next text char on mismatch?

A. Check each position, working from left to right.



	j	0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

table giving pattern char to compare to the next text char

KMP substring search preprocessing (concept)

Q. What pattern char do we compare to the next text char on mismatch?

A. Check each position, working from left to right.

matched chars

current char is mismatch

A A A A B B B B C C C A B A B A B ? ? ? ? ? ? ? ? ? ?

✘ A B A B A C

✘ A B A B A C

✔ A B A B A C

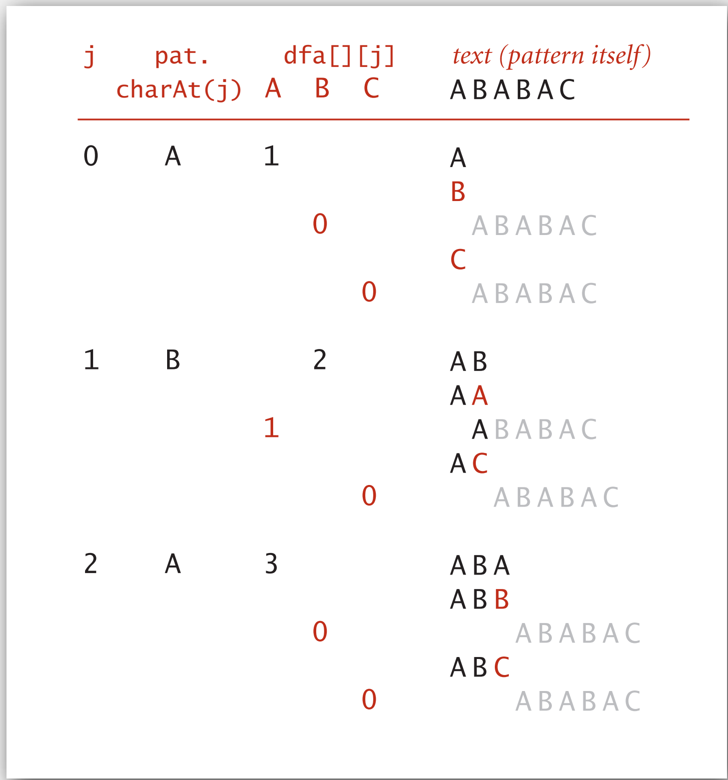
pat.charAt(4)

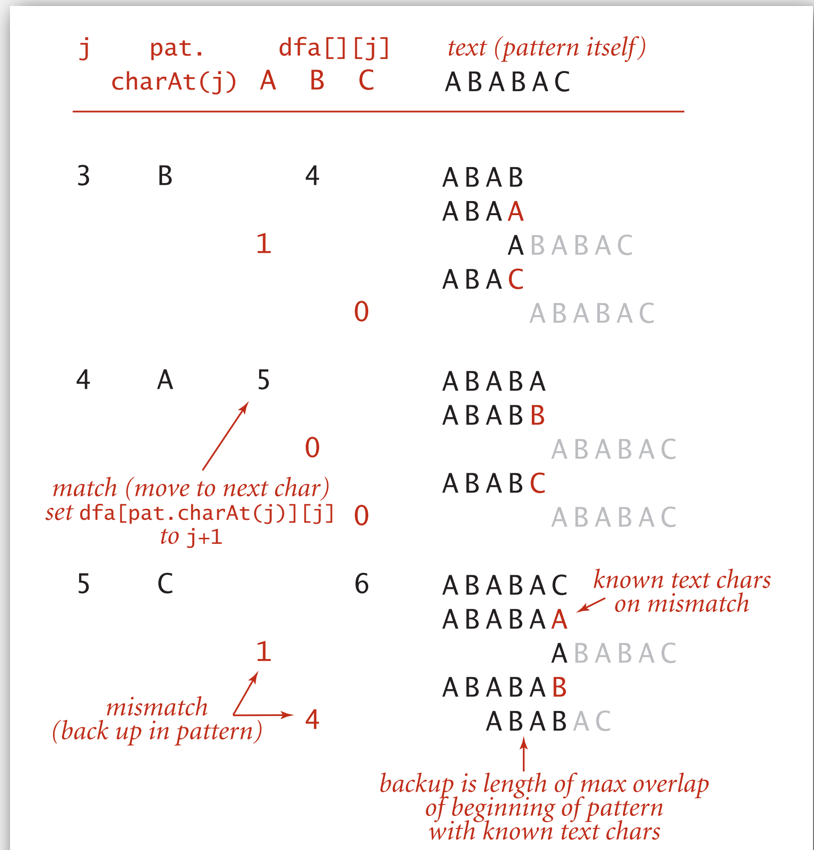
	j	0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

table giving pattern char to compare to the next text char

KMP substring search preprocessing (concept)

Fill in table columns by doing computation for each possible mismatch position.





Pattern backup for A B A B A C in KMP substring search

Deterministic finite state automaton (DFA)

DFA is abstract string-searching machine.

- Finite number of states (including start and halt).
- Exactly one transition for each char in alphabet.
- Accept if sequence of transitions leads to halt state.

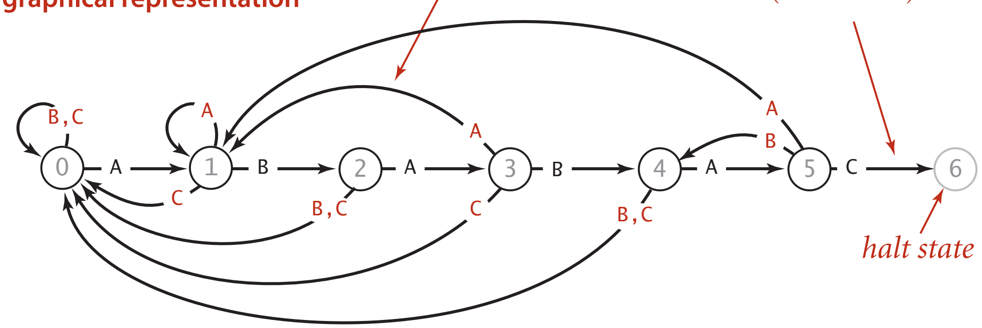
internal representation

		j	0	1	2	3	4	5
pat.charAt(j)			A	B	A	B	A	C
dfa[][j]	A		1	1	3	1	5	1
	B		0	2	0	4	0	4
	C		0	0	0	0	0	6

mismatch transition (back up)

match transition (increment)

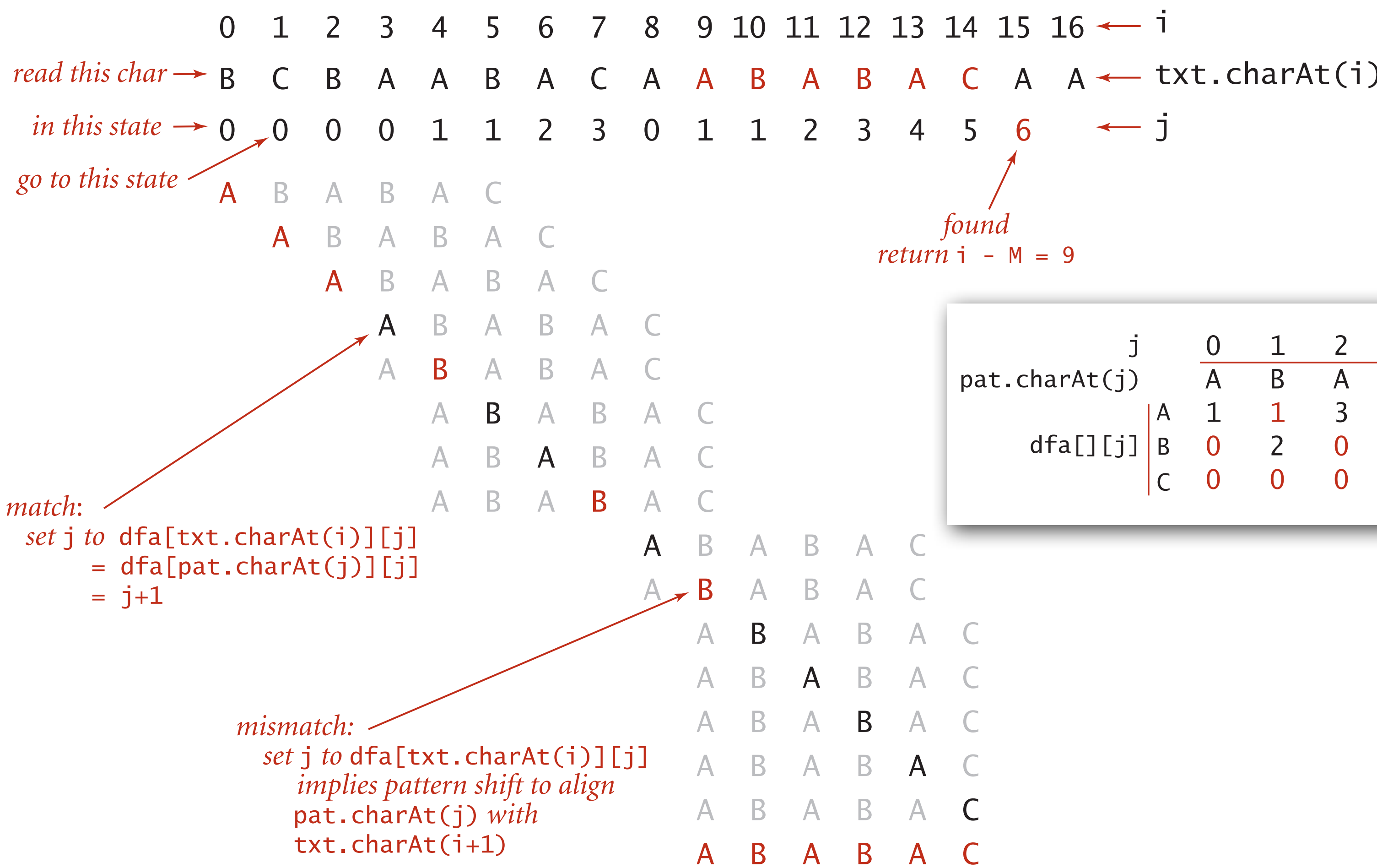
graphical representation



DFA corresponding to the string A B A B A C

If in state **j** reading char **c**:
halt if **j** is **6**
else move to state **dfa[c][j]**

KMP substring search: trace



j		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

Trace of KMP substring search (DFA simulation) for A B A B A C

KMP implementation. Build machine for pattern, simulate it on text.

Key differences from brute-force implementation.

- Text pointer `i` never decrements.
- Need to precompute `dfa[][]` table from pattern.

```
public int search(String txt)
{
   int i, j, N = txt.length();
   for (i = 0, j = 0; i < N && j < M; i++)
      j = dfa[txt.charAt(i)][j];
   if (j == M) return i - M;
   else        return N;
}
```

Running time.

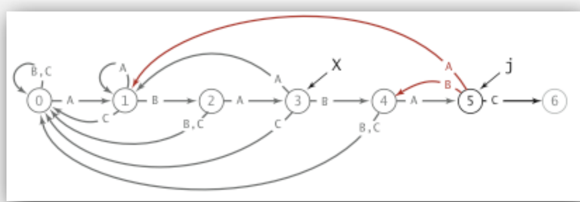
- Simulate DFA: at most N character accesses.
- Build DFA: at most $M^2 R$ character accesses (stay tuned for better method).

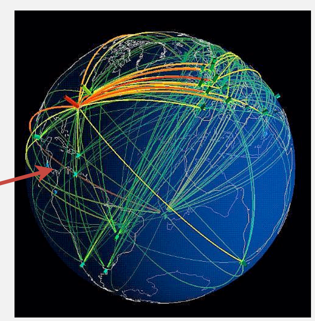
KMP search: Java implementation

Key differences from brute-force implementation.

- Text pointer `i` never decrements.
- Need to precompute `dfa[][]` table from pattern.
- Could use input stream.

```
public int search(In in)
{
   int i, j;
   for (i = 0, j = 0; !in.isEmpty() && j < M; i++)
      j = dfa[in.readChar()][j];
   if (j == M) return i - M;
   else        return i;
}
```





Efficiently constructing the DFA for KMP substring search

Q. What state X would the DFA be in if it were restarted to correspond to shifting the pattern one position to the right?

matched chars

next char

A B B B B C C C A B A B A ? ? ? ? ? ? ?

A B A B A C

A. Use the (partially constructed) DFA to find X!

A B B B B C C C A B A B A ? ? ? ? ? ? ? ?

0 0 1 2 3

X

	j	0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	?
	B	0	2	0	4	0	?
	C	0	0	0	0	0	?

Consequence.

- We want the same transitions as X for the next state on mismatch.
  copy `dfa[][X]` to `dfa[][j]`
- But a different transition (to j+1) on match.
  set `dfa[pat.charAt(j)][j]` to `j+1`

	j	0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

Efficiently constructing the DFA for KMP substring search

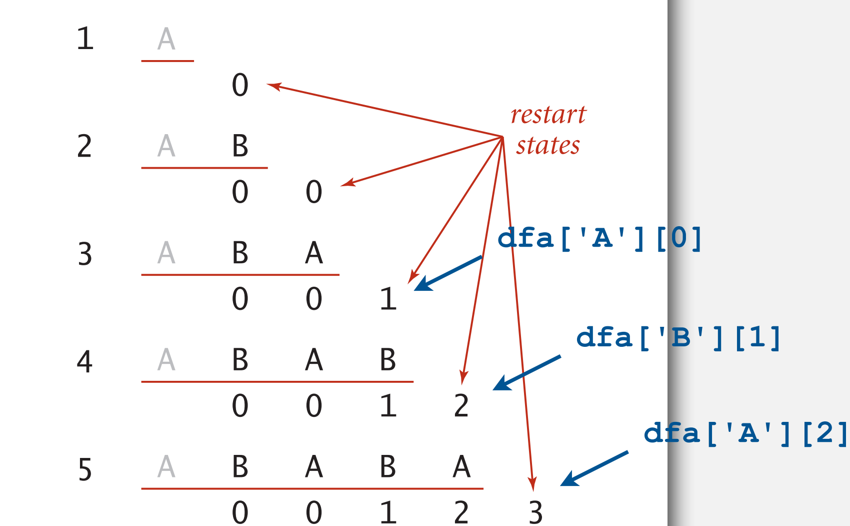
Build table by finding answer to Q for each pattern position.

Q. What state X would the DFA be in if it were restarted to correspond to shifting the pattern one position to the right?

	j	0	1	2	3	4	5
	pat.charAt(j)	A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

Observation. No need to restart DFA.

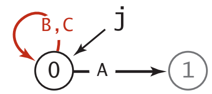
- Remember last restart state in X.
- Use DFA to update X.
- `X = dfa[pat.charAt(j)][X]`



DFA simulations to compute restart states for A B A B A C

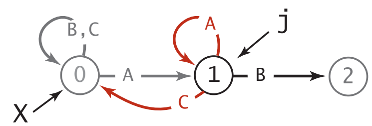
Constructing the DFA for KMP substring search: example

j	0
pat.charAt(j)	A
dfa[][j] A	1
dfa[][j] B	0
dfa[][j] C	0



X

j	0	1
pat.charAt(j)	A	B
dfa[][j] A	1	1
dfa[][j] B	0	2
dfa[][j] C	0	0

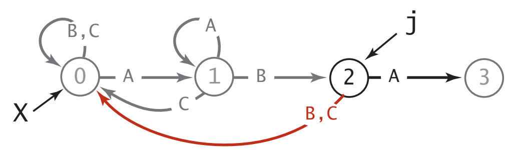


copy `dfa[][X]` *to* `dfa[][j]`

```
dfa[pat.charAt(j)][j] = j+1;
X = dfa[pat.charAt(j)][X]];
```

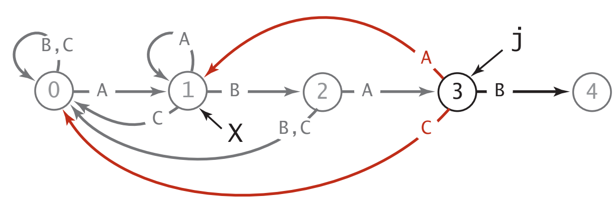
X

j	0	1	2
pat.charAt(j)	A	B	A
dfa[][j] A	1	1	3
dfa[][j] B	0	2	0
dfa[][j] C	0	0	0



X

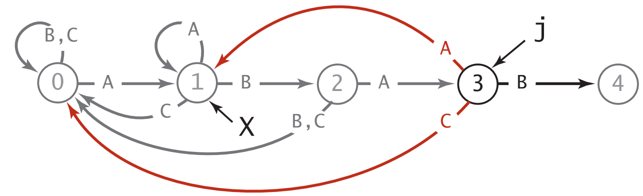
j	0	1	2	3
pat.charAt(j)	A	B	A	B
dfa[][j] A	1	1	3	1
dfa[][j] B	0	2	0	4
dfa[][j] C	0	0	0	0



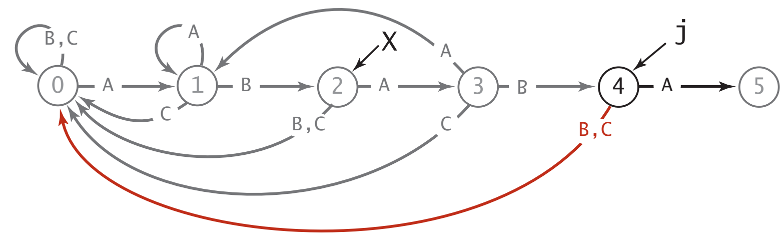
Constructing the DFA for KMP substring search for A B A B A C

Constructing the DFA for KMP substring search: example

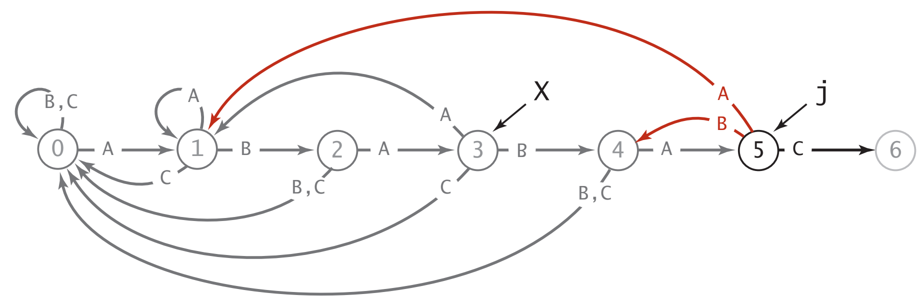
			X		
j		0	1	2	3
pat.charAt(j)		A	B	A	B
dfa[][j]	A	1	1	3	1
	B	0	2	0	4
	C	0	0	0	0



				X		
j		0	1	2	3	4
pat.charAt(j)		A	B	A	B	A
dfa[][j]	A	1	1	3	1	5
	B	0	2	0	4	0
	C	0	0	0	0	0



					X		
j		0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6



Constructing the DFA for KMP substring search for A B A B A C

Constructing the DFA for KMP substring search: Java implementation

For each `j`:

- Copy `dfa[][X]` to `dfa[][j]` for mismatch case.
- Set `dfa[pat.charAt(j)][j]` to `j+1` for match case.
- Update `X`.

```
public KMP(String pat)
{
   this.pat = pat;
   M = pat.length();
   dfa = new int[R][M];
   dfa[pat.charAt(0)][0] = 1;
   for (int X = 0, j = 1; j < M; j++)
   {
      for (int c = 0; c < R; c++)
         dfa[c][j] = dfa[c][X];          // copy mismatch cases
      dfa[pat.charAt(j)][j] = j+1;       // set match case
      X = dfa[pat.charAt(j)][X];         // update restart state
   }
}
```

copy mismatch cases

set match case

update restart state

Running time. M character accesses.

Proposition. KMP substring search accesses no more than M + N chars to search for a pattern of length M in a text of length N.

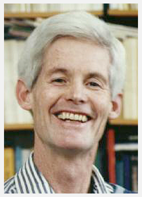
Pf. We access each pattern char once when constructing the DFA, and each text char once (in the worst case) when simulating the DFA.

Remark. Takes time and space proportional to R M to construct `dfa[][]`, but with cleverness, can reduce time and space to M.

Knuth-Morris-Pratt: brief history

Brief history.

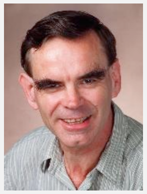
- Inspired by esoteric theorem of Cook.
- Discovered in 1976 independently by two theoreticians and a hacker.
  - Knuth: discovered linear-time algorithm
  - Pratt: made running time independent of alphabet
  - Morris: trying to build a text editor
- Theory meets practice.



Stephen Cook

Don Knuth

Jim Morris



Vaughan Pratt

- brute force
- Knuth-Morris-Pratt
- **Boyer-Moore**
- Rabin-Karp

Robert Boyer

J. Strother Moore

Boyer-Moore: mismatched character heuristic

Intuition.

- Scan characters in pattern from right to left.
- Can skip M text chars when finding one not in the pattern.

i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	text →	H	A	Y	S	T	A	C	K	N	E	E	D	L	E	I	N	A
0	5	N	E	E	D	L	E	← *pattern*										
6	5							N	E	E	D	L	E					
8	5									N	E	E	D	L	E			
8	0																	

return i = 8

Boyer-Moore: mismatched character heuristic

Intuition.

- Scan characters in pattern from right to left.
- Can skip M text chars when finding one not in the pattern.

not in pattern

`txt[]`

i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
		A	A	A	B	B	A	A	B	A	B	A	A	A	B	B	A	G	Y
5	6	S	Y	Z	Y	G	Y												
11	6							S	Y	Z	Y	G	Y						
15	3													S	Y	Z	Y	G	Y

return `i = 18` *(no match)*

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

A. Compute `right[c]` = rightmost occurrence of character `c` in `pat[]`.

```
right = new int[R];
for (int c = 0; c < R; c++)
   right[c] = -1;
for (int j = 0; j < M; j++)
   right[pat.charAt(j)] = j;
```

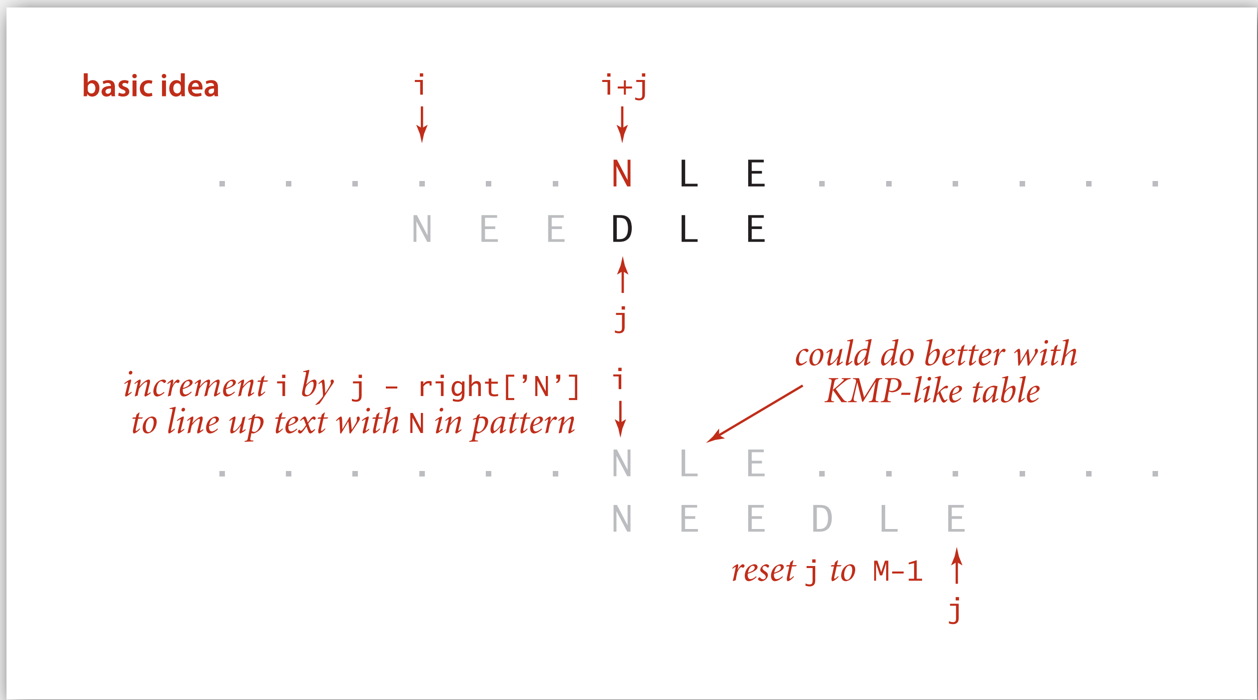
		N	E	E	D	L	E	
c		0	1	2	3	4	5	right[c]
A	-1	-1	-1	-1	-1	-1	-1	-1
B	-1	-1	-1	-1	-1	-1	-1	-1
C	-1	-1	-1	-1	-1	-1	-1	-1
D	-1	-1	-1	-1	3	3	3	3
E	-1	-1	1	2	2	2	5	5
...								-1
L	-1	-1	-1	-1	-1	4	4	4
M	-1	-1	-1	-1	-1	-1	-1	-1
N	-1	0	0	0	0	0	0	0
...								-1

Boyer-Moore skip table computation

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

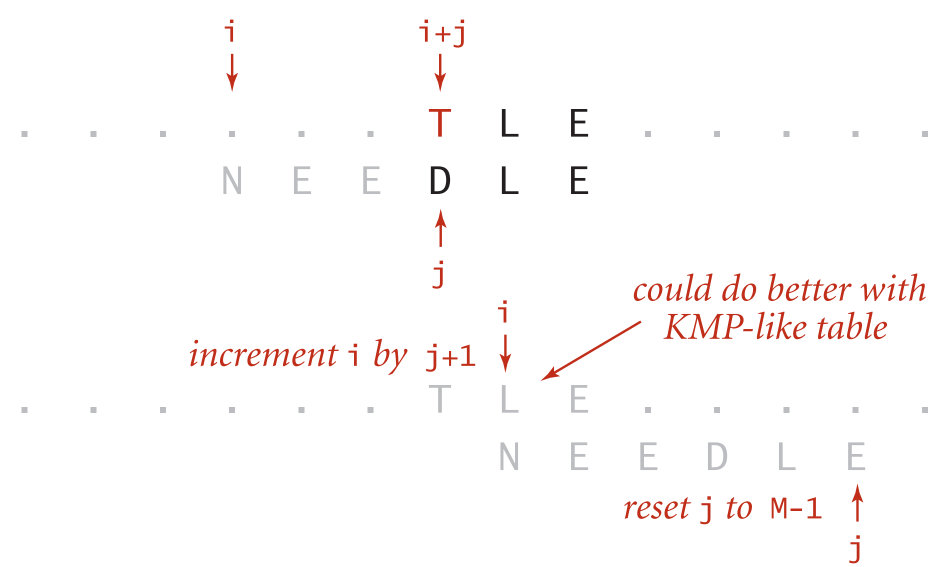
A. Compute `right[c]` = rightmost occurrence of character `c` in `pat[]`.



Boyer-Moore: mismatched character heuristic

Q. How much to skip?

A. Compute `right[c]` = rightmost occurrence of character `c` in `pat[]`.



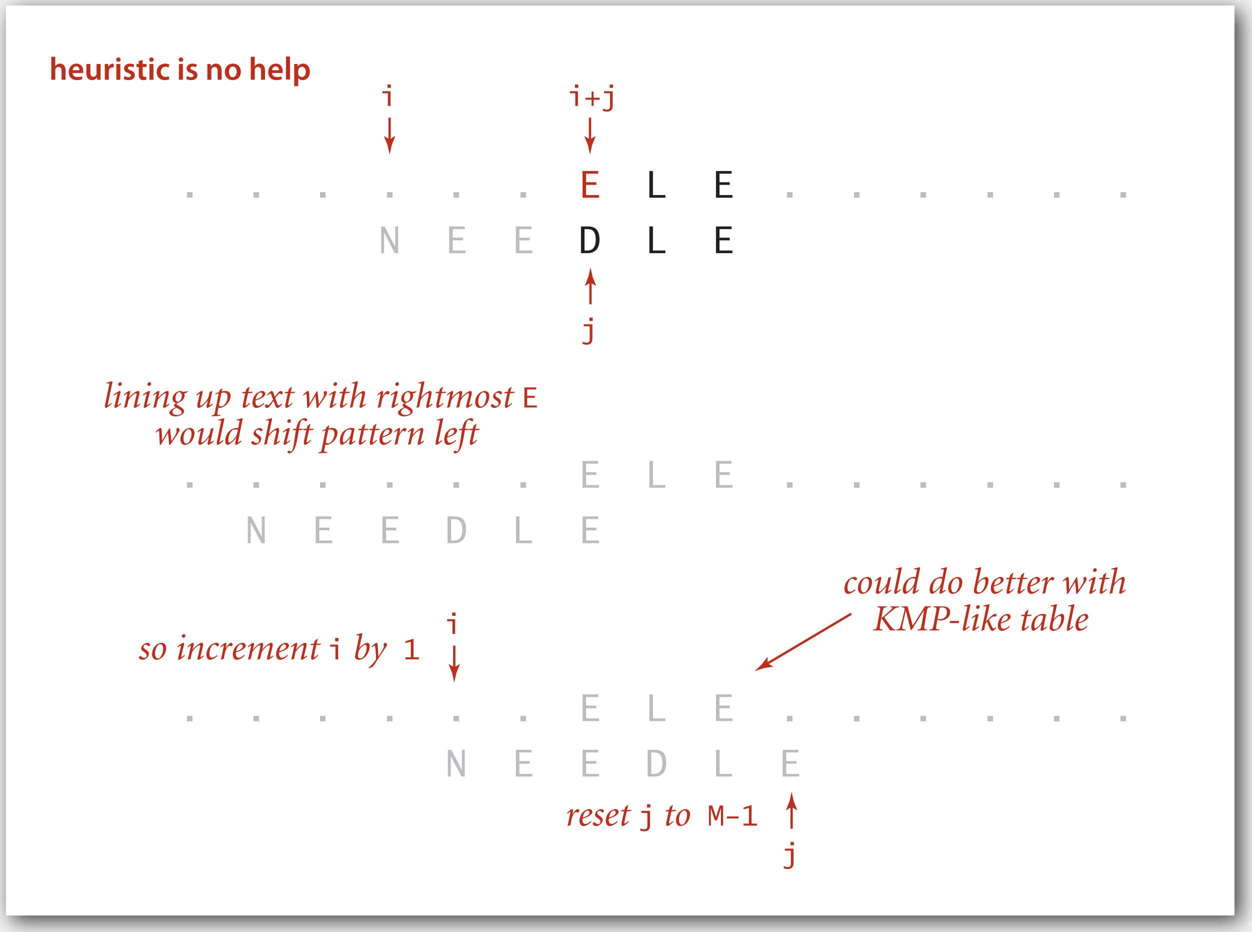
Mismatched character heuristic (mismatch not in pattern)

Easy fix. Set `right[c]` to -1 for characters not in pattern.

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

A. Compute `right[c]` = rightmost occurrence of character `c` in `pat[]`.



Boyer-Moore: Java implementation

```
public int search(String txt)
{
   int N = txt.length();
   int M = pat.length();
   int skip;
   for (int i = 0; i <= N-M; i += skip)
   {
      skip = 0;
      for (int j = M-1; j >= 0; j--)
         if (pat.charAt(j) != txt.charAt(i+j))
         {
            skip = Math.max(1, j - right[txt.charAt(i+j)]);
            break;
         }
      if (skip == 0) return i;
   }
   return N;
}
```

compute skip value

match

Boyer-Moore: analysis

Property. Substring search with the Boyer-Moore mismatched character heuristic takes about ~ N/M character compares to search for a pattern of length M in a text of length N.

sublinear

Worst-case. Can be as bad as ~ $M N$.

i	skip	0	1	2	3	4	5	6	7	8	9	
	txt →	B	B	B	B	B	B	B	B	B	B	
0	0	A	B	B	B	B						← pat
1	1		A	B	B	B	B					
2	1			A	B	B	B	B				
3	1				A	B	B	B	B			
4	1					A	B	B	B	B		
5	1						A	B	B	B	B	

Boyer-Moore variant. Can improve worst case to ~ $3 N$ by adding a KMP-like rule to guard against repetitive patterns.

- brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- **Rabin-Karp**

Michael Rabin, Turing Award '76
and Dick Karp, Turing Award '85

Rabin-Karp fingerprint search

Basic idea.

- Compute a hash of pattern characters 0 to M-1.
- For each i, compute a hash of text characters i to M+i-1.
- If pattern hash = text substring hash, check for a match.

pat.charAt(i)

i	0	1	2	3	4	
	2	6	5	3	5	% 997 = 613

txt.charAt(i)

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3	
0	3	1	4	1	5	% 997 = 508											
1		1	4	1	5	9	% 997 = 201										
2			4	1	5	9	2	% 997 = 715									
3				1	5	9	2	6	% 997 = 971								
4					5	9	2	6	5	% 997 = 442							
5						9	2	6	5	3	% 997 = 929			*match*			
6 ← *return* i = 6							2	6	5	3	5	% 997 = 613					

Basis for Rabin-Karp substring search

Efficiently computing the hash function

Modular hash function. Using the notation t_i for `txt.charAt(i)`, we wish to compute

$$x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \ldots + t_{i+M-1} R^0 \pmod{Q}$$

Intuition. M-digit, base-R integer, modulo Q.

Horner's method. Linear-time method to evaluate degree-M polynomial.

```
        pat.charAt(i)
 i      0  1  2  3  4
        2  6  5  3  5
 0      2  % 997 = 2        R             Q
 1      2  6  % 997 = (2*10 + 6) % 997 = 26
 2      2  6  5  % 997 = (26*10 + 5) % 997 = 265
 3      2  6  5  3  % 997 = (265*10 + 3) % 997 = 659
 4      2  6  5  3  5  % 997 = (651*10 + 5) % 997 = 613
```

Computing the hash value for the pattern with Horner's method

```
// Compute hash for M-digit key
private int hash(String key)
{
   int h = 0;
   for (int i = 0; i < M; i++)
      h = (R * h + key.charAt(j)) % Q;
   return h;
}
```

Efficiently computing the hash function

Challenge. How to efficiently compute x_{i+1} given that we know x_i.

$$x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \ldots + t_{i+M-1} R^0$$

$$x_{i+1} = t_{i+1} R^{M-1} + t_{i+2} R^{M-2} + \ldots + t_{i+M} R^0$$

Key property. Can do it in constant time!

$$x_{i+1} = (x_i - t_i R^{M-1}) R + t_{i+M}$$

i	...	2	3	4	5	6	7	...	
current value	1	4	1	5	9	2	6	5	*text*
new value		4	1	5	9	2	6	5	

	4	1	5	9	2	*current value*
-	4	0	0	0	0	
		1	5	9	2	*subtract leading digit*
			*	1	0	*multiply by radix*
	1	5	9	2	0	
				+	6	*add new trailing digit*
	1	5	9	2	6	*new value*

Rabin-Karp: Java implementation

```
public class RabinKarp {
    private String pat;      // the pattern
    private int patHash;     // pattern hash value
    private int M;           // pattern length
    private int Q = 8355967; // modulus
    private int R;           // radix
    private int RM;          // R^(M-1) % Q

    public RabinKarp(String pat) {
        this.R = 256;
        this.pat = pat;
        this.M = pat.length;

        RM = 1;
        for (int i = 1; i <= M-1; i++)
           RM = (R * RM) % Q;
        patHash = hash(pat);
     }

    private int hash(String key)
    {  /* as before */  }

    public int search(String txt)
    {  /* see next slide */  }
}
```

a large prime, but small enough to avoid 32-bit integer overflow

precompute R^{M-1} (mod Q)

Rabin-Karp: Java implementation (continued)

```
public int search(String txt)
{
   int N = txt.length();
   if (N < M) return N;
   int offset = hashSearch(txt);
   if (offset == N) return N;

   for (int i = 0; i < M; i++)
        if (pat.charAt(i) != txt.charAt(offset + i))
            return N;
    return offset;
}

private int hashSearch(String txt)
{
    int N = txt.length();
    int txtHash = hash(txt);
    if (patHash == txtHash) return 0;
    for (int i = M; i < N; i++)
    {
        txtHash = (txtHash + Q - RM*txt.charAt(i-M) % Q) % Q;
        txtHash = (txtHash*R + txt.charAt(i)) % Q;
        if (patHash == txtHash) return i - M + 1;
    }
    return N;
}
```

check if hash collision corresponds to a match

check for hash collision using rolling hash function

Rabin-Karp substring search example

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3		
0	3																% 997 = 3	Q
1	3	1															% 997 = (3*10 + 1) % 997 = 31	
2	3	1	4														% 997 = (31*10 + 4) % 997 = 314	
3	3	1	4	1													% 997 = (314*10 + 1) % 997 = 150	
4	3	1	4	1	5												% 997 = (150*10 + 5) % 997 = 508	RM R
5		1	4	1	5	9											% 997 = ((508 + 3*(997 - 30))*10 + 9) % 997 = 201	
6			4	1	5	9	2										% 997 = ((201 + 1*(997 - 30))*10 + 2) % 997 = 715	
7				1	5	9	2	6									% 997 = ((715 + 4*(997 - 30))*10 + 6) % 997 = 971	
8					5	9	2	6	5								% 997 = ((971 + 1*(997 - 30))*10 + 5) % 997 = 442	match
9						9	2	6	5	3							% 997 = ((442 + 5*(997 - 30))*10 + 3) % 997 = 929	
10 ← *return* i-M+1 = 6							2	6	5	3	5						% 997 = ((929 + 9*(997 - 30))*10 + 5) % 997 = 613	

Rabin-Karp substring search example

Rabin-Karp analysis

Proposition. Rabin-Karp substring search is extremely likely to be linear-time.

Worst-case. Takes time proportional to MN.

- In worst case, all substrings hash to same value.
- Then, need to check for match at each text position.

Theory. If Q is a sufficiently large random prime (about MN^2), then probability of a false collision is about 1/N $\Rightarrow$ expected running time is linear.

Practice. Choose Q to avoid integer overflow. Under reasonable assumptions, probability of a collision is about 1/Q $\Rightarrow$ linear in practice.

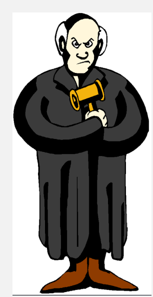
Advantages.

- Extends to 2D patterns.
- Extends to finding multiple patterns.

Disadvantages.

- Arithmetic ops slower than char compares.
- Poor worst-case guarantee.
- Requires backup.

Q. How would you extend Rabin-Karp to efficiently search for any one of P possible patterns in a text of length N?



Substring search cost summary

Cost of searching for an M-character pattern in an N-character text.

algorithm (data structure)	operation count		backup in input?	space grows with
	guarantee	typical		
brute force	MN	$1.1\,N$	*yes*	1
Knuth-Morris-Pratt (full DFA)	$2\,N$	$1.1\,N$	*no*	MR
Knuth-Morris-Pratt (mismatch transitions only)	$3\,N$	$1.1\,N$	*no*	M
Boyer-Moore	$3\,N$	N / M	*yes*	R
Boyer-Moore (mismatched character heuristic only)	MN	N / M	*yes*	R
Rabin-Karp†	$7\,N$ †	$7\,N$	*no*	1

† probabilisitic guarantee, with uniform hash function

Cost summary for substring-search implementations

5.4 Pattern Matching

- regular expressions
- REs and NFAs
- NFA simulation
- NFA construction
- applications

- regular expressions
- NFAs
- NFA simulation
- NFA construction
- applications

Pattern matching

Substring search. Find a single string in text.

Pattern matching. Find one of a specified set of strings in text.

Ex. [genomics]

- Fragile X syndrome is a common cause of mental retardation.
- Human genome contains triplet repeats of `CGG` or `AGG`, bracketed by `GCG` at the beginning and `CTG` at the end.
- Number of repeats is variable, and correlated with syndrome.

pattern `GCG(CGG|AGG)*CTG`

text `GCGGCGTGTGTGCGAGAGAGTGGGTTTAAAGCTGGCGCGGAGGCGGCTGGCGCGGAGGCTG`

Pattern matching: applications

Test if a string matches some pattern.

- Process natural language.
- Scan for virus signatures.
- Access information in digital libraries.
- Filter text (spam, NetNanny, Carnivore, malware).
- Validate data-entry fields (dates, email, URL, credit card).
- Search for markers in human genome using PROSITE patterns.

Parse text files.

- Compile a Java program.
- Crawl and index the Web.
- Read in data stored in ad hoc input file format.
- Automatically create Java documentation from Javadoc comments.

Regular expressions

A regular expression is a notation to specify a (possibly infinite) set of strings.

a "language"

operation	example RE	matches	does not match
concatenation	`AABAAB`	`AABAAB`	every other string
or	`AA | BAAB`	`AA` `BAAB`	every other string
closure	`AB*A`	`AA` `ABBBBBBBBA`	`AB` `ABABA`
parentheses	`A(A|B)AAB`	`AAAAB` `ABAAB`	every other string
parentheses	`(AB)*A`	`A` `ABABABABABA`	`AA` `ABBA`

Regular expression shortcuts

Additional operations are often added for convenience.

Ex. `[A-E]+` is shorthand for `(A|B|C|D|E)(A|B|C|D|E)*`

operation	example RE	matches	does not match
wildcard	`.U.U.U.`	CUMULUS JUGULUM	SUCCUBUS TUMULTUOUS
at least 1	`A(BC)+DE`	ABCDE ABCBCDE	ADE BCDE
character classes	`[A-Za-z][a-z]*`	word Capitalized	camelCase 4illegal
exactly k	`[0-9]{5}-[0-9]{4}`	08540-1321 19072-5541	111111111 166-54-111
complement	`[^AEIOU]{6}`	RHYTHM	DECADE

Regular expression examples

Notation is surprisingly expressive

regular expression	matches	does not match
`.*SPB.*` *(contains the trigraph spb)*	RASPBERRY CRISPBREAD	SUBSPACE SUBSPECIES
`[0-9]{3}-[0-9]{2}-[0-9]{4}` *(Social Security numbers)*	166-11-4433 166-45-1111	11-55555555 8675309
`[a-z]+@([a-z]+\.)+(edu|com)` *(valid email addresses)*	wayne@princeton.edu rs@princeton.edu	spam@nowhere
`[$_A-Za-z][$_A-Za-z0-9]*` *(valid Java identifiers)*	ident3 PatternMatcher	3a ident#3

and plays a well-understood role in the theory of computation.

Regular expressions to the rescue

http://xkcd.com/208/

Can the average web surfer learn to use REs?

Google. Supports * for full word wildcard and | for union.

Can the average programmer learn to use REs?

Perl RE for valid RFC822 email addresses

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:
\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[
 \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\
](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:
(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)
?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[
 \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t]
)*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*
)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)
*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r
\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t
]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](
?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?
:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?
[ \t]))*"(?:(?:\r\n)?[ \t])*)*:(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|
\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"
(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[
\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(
?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([
^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\
]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\
r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]
|\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\
.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?
:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".
\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]
]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)(?:,\s*(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[
\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t
])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|
\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\
]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["
()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>
@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,
;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".
\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\[
"()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])
+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z
|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*))*)?;\s*)
```

http http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html

Regular expression caveat

Writing a RE is like writing a program.

- Need to understand programming model.
- Can be easier to write than read.
- Can be difficult to debug.

> *" Some people, when confronted with a problem, think 'I know I'll use regular expressions.' Now they have two problems. "*
>
> *— Jamie Zawinski (flame war on alt.religion.emacs)*

Bottom line. REs are amazingly powerful and expressive, but using them in applications can be amazingly complex and error-prone.

- regular expressions
- **NFAs**
- NFA simulation
- NFA construction
- applications

Pattern matching implementation: basic plan (first attempt)

Ken Thompson

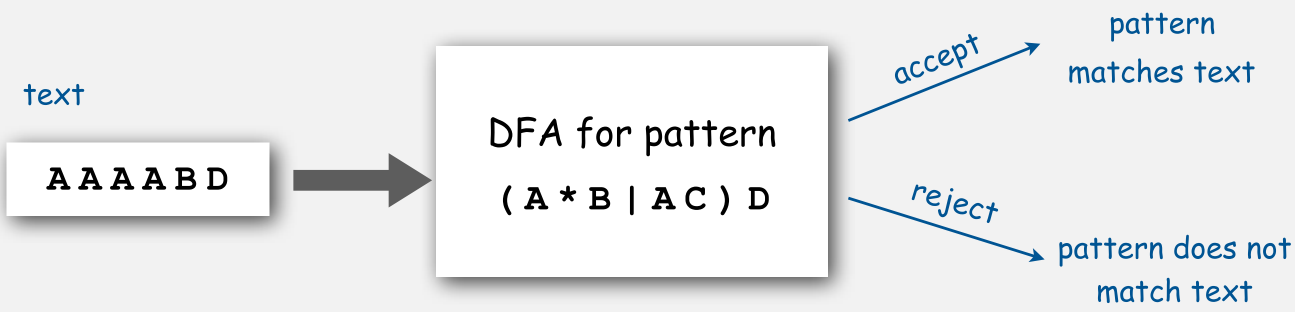
Overview is the same as for KMP!

- No backup in text input stream.
- Linear-time guarantee.

Underlying abstraction. Deterministic finite state automata (DFA).

Basic plan.

- Build DFA from RE.
- Simulate DFA with text as input.



Bad news. Basic plan is infeasible (DFA may have exponential number of states).

Pattern matching implementation: basic plan (revised)

Overview is similar to KMP.

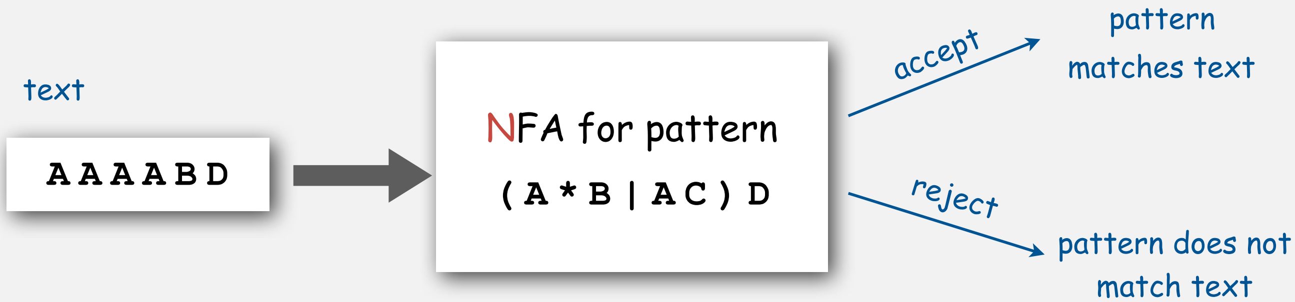
- No backup in text input stream.
- Quadratic-time guarantee (linear-time typical).

Ken Thompson

Underlying abstraction. Nondeterministic finite state automata (NFA).

Basic plan.

- Build NFA from RE.
- Simulate NFA with text as input.



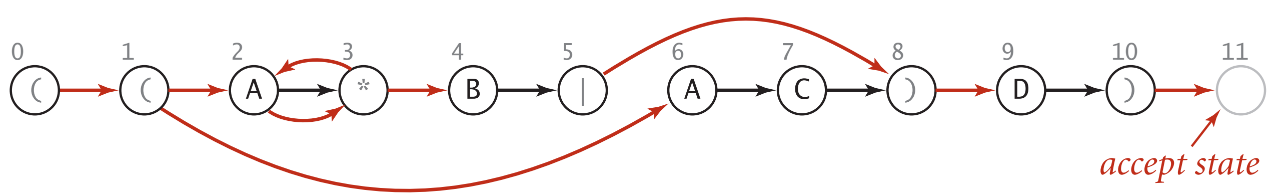
Nondeterministic finite-state automata

Pattern matching NFA.

- Pattern enclosed in parentheses.
- One state per pattern character (start = 0, accept = M).
- Red ε-transition (change state, but don't scan input).
- Black match transition (change state and scan to next char).
- Accept if any sequence of transitions ends in accept state.

Nondeterminism.

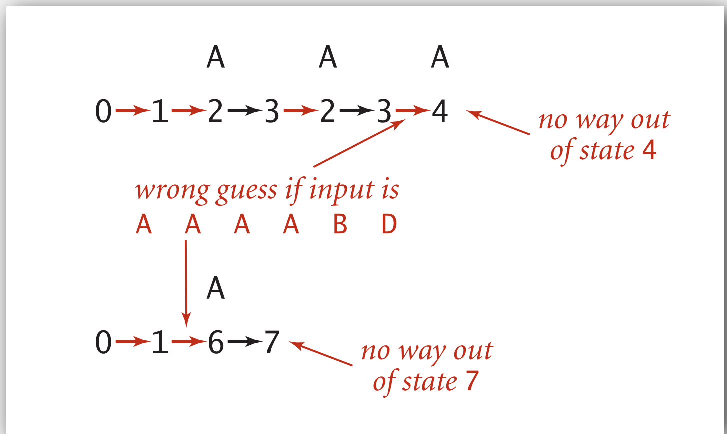
- One view: machine can guess the proper sequence of state transitions.
- Another view: sequence is a proof that the machine accepts the text.

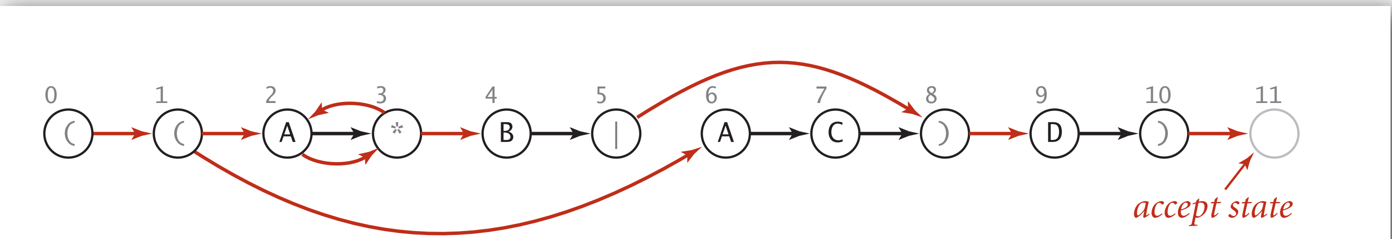


NFA corresponding to the pattern ((A * B | A C) D)

Nondeterministic finite-state automata

Ex. Is **AAAABD** matched by NFA?

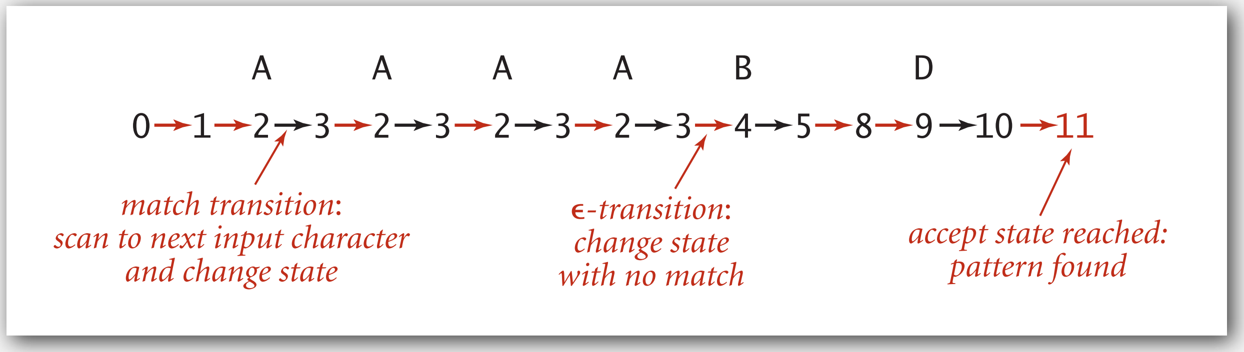




NFA corresponding to the pattern ((A * B | A C) D)

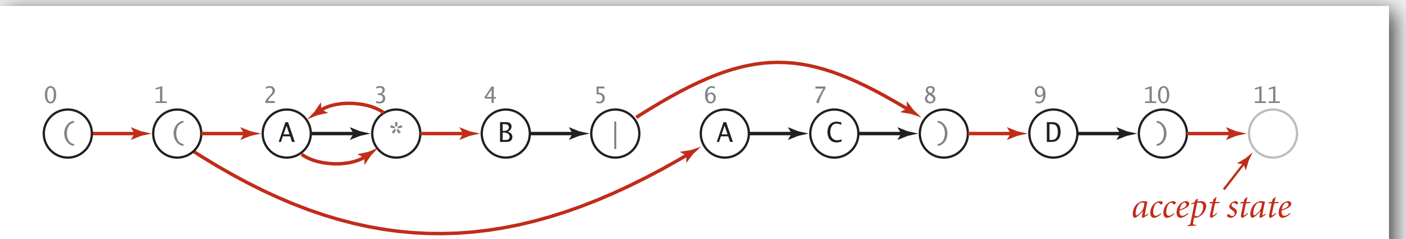
Nondeterministic finite-state automata

Ex. Is **AAAABD** matched by NFA?



yes!

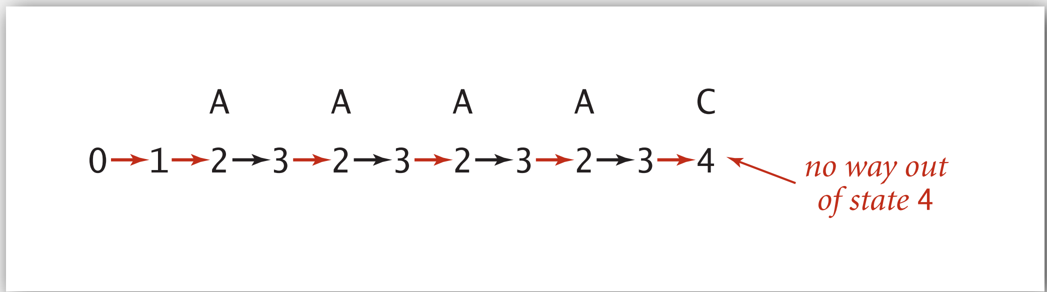
Note: any sequence of legal transitions that ends in state 11 is a proof.



NFA corresponding to the pattern ((A * B | A C) D)

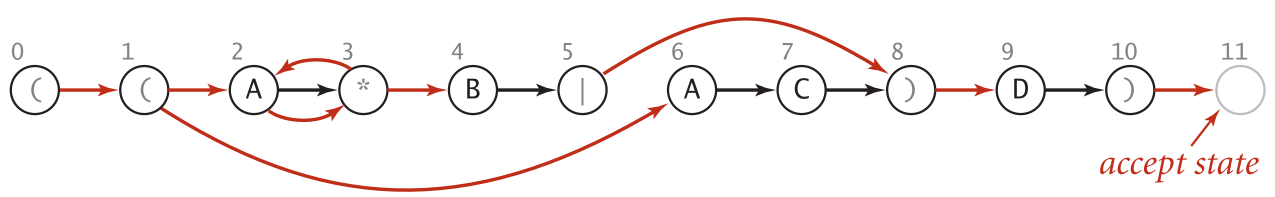
Nondeterministic finite-state automata

Ex. Is `AAAAC` matched by NFA?



no

Note: this is not a complete proof!
(need to mention the infinite number of sequences involving ε-transitions between 2 and 3)



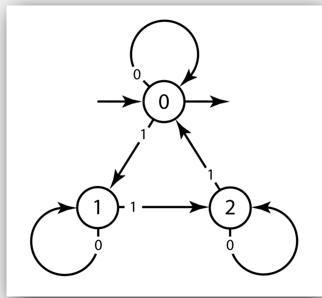
NFA corresponding to the pattern ((A * B | A C) D)

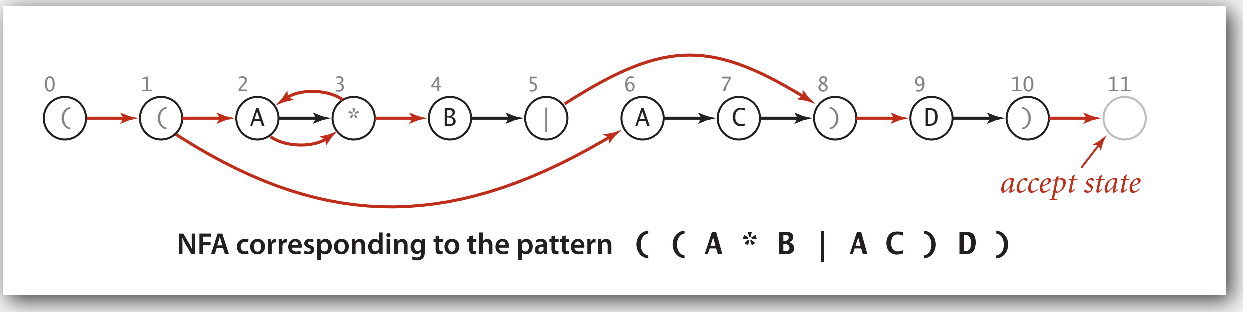
Nondeterminism

Q. How to determine whether a string is recognized by an automaton?

DFA. Deterministic $\Rightarrow$ exactly one applicable transition.

NFA. Nondeterministic $\Rightarrow$ can be several applicable transitions; need to select the right one!





Q. How to simulate NFA?

A. Systematically consider all possible transition sequences.

Pattern matching implementation: basic plan (revised)


Ken Thompson

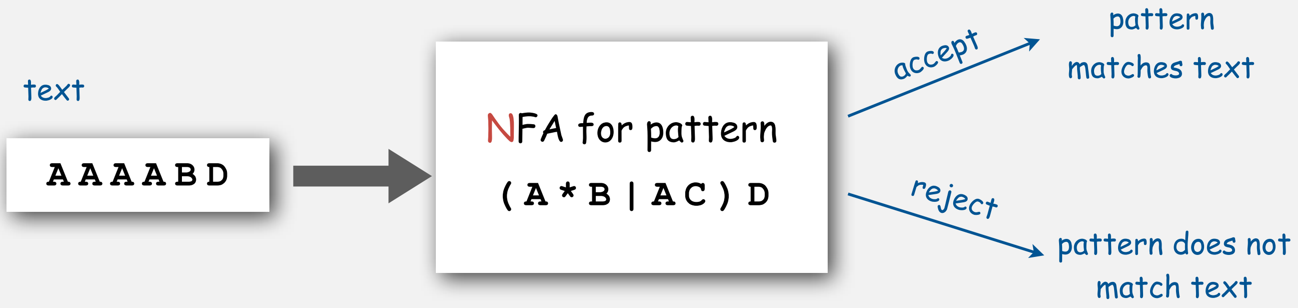
Overview is similar to KMP.

- No backup in text input stream.
- Quadratic-time guarantee (linear-time typical).

Underlying abstraction. Nondeterministic finite state automata (NFA).

Basic plan.

- Build NFA from RE.
- Simulate NFA with text as input.



- regular expressions
- NFAs
- **NFA simulation**
- NFA construction
- applications

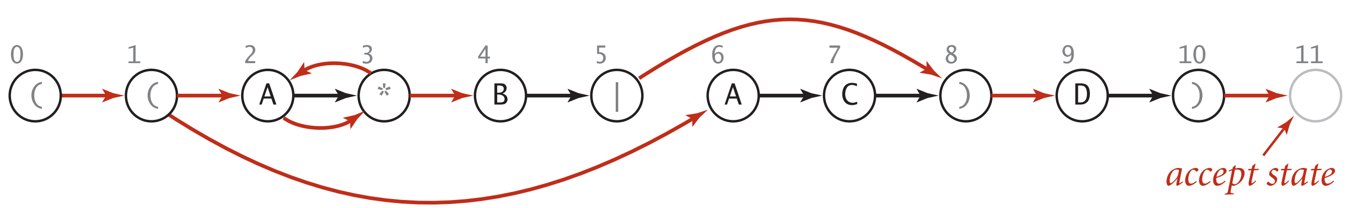
NFA representation

State names. Integers from `0` to `M`.

Match-transitions. Keep regular expression in array `re[]`.

ε-transitions. Store in a digraph `G`.

- `0→1, 1→2, 1→6, 2→3, 3→2, 3→4, 5→8, 8→9, 10→11`

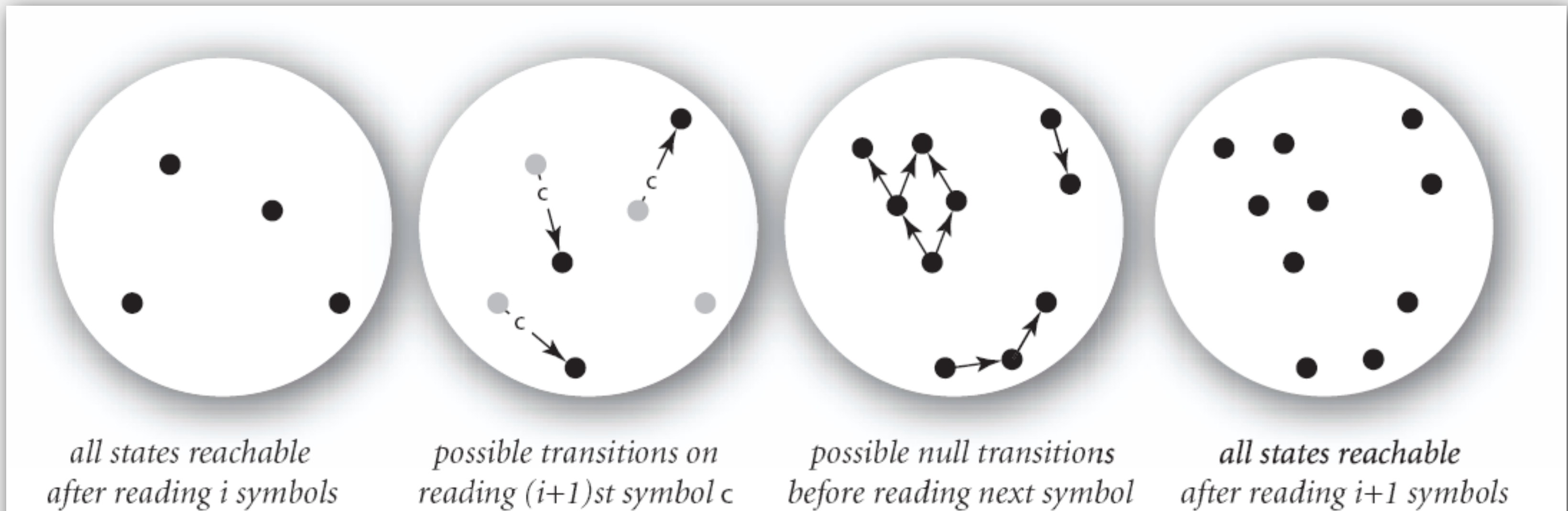


NFA corresponding to the pattern `( ( A * B | A C ) D )`

NFA simulation

Q. How to efficiently simulate an NFA?

A. Maintain set of all possible states that NFA could be in after reading in the first i text characters.



One step in simulating an NFA

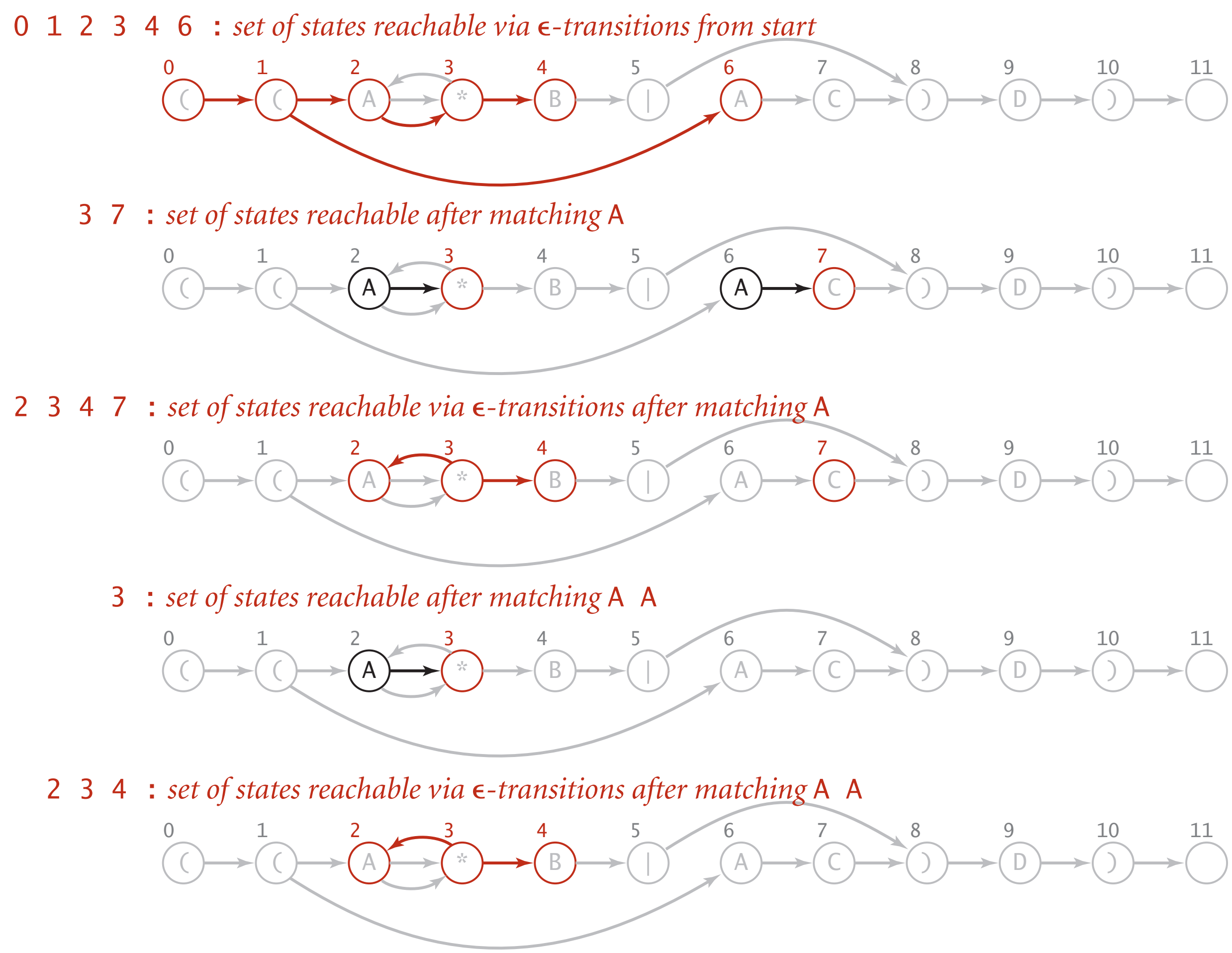
Q. How to perform reachability?

Digraph reachability

Find all vertices reachable from a given set of vertices.

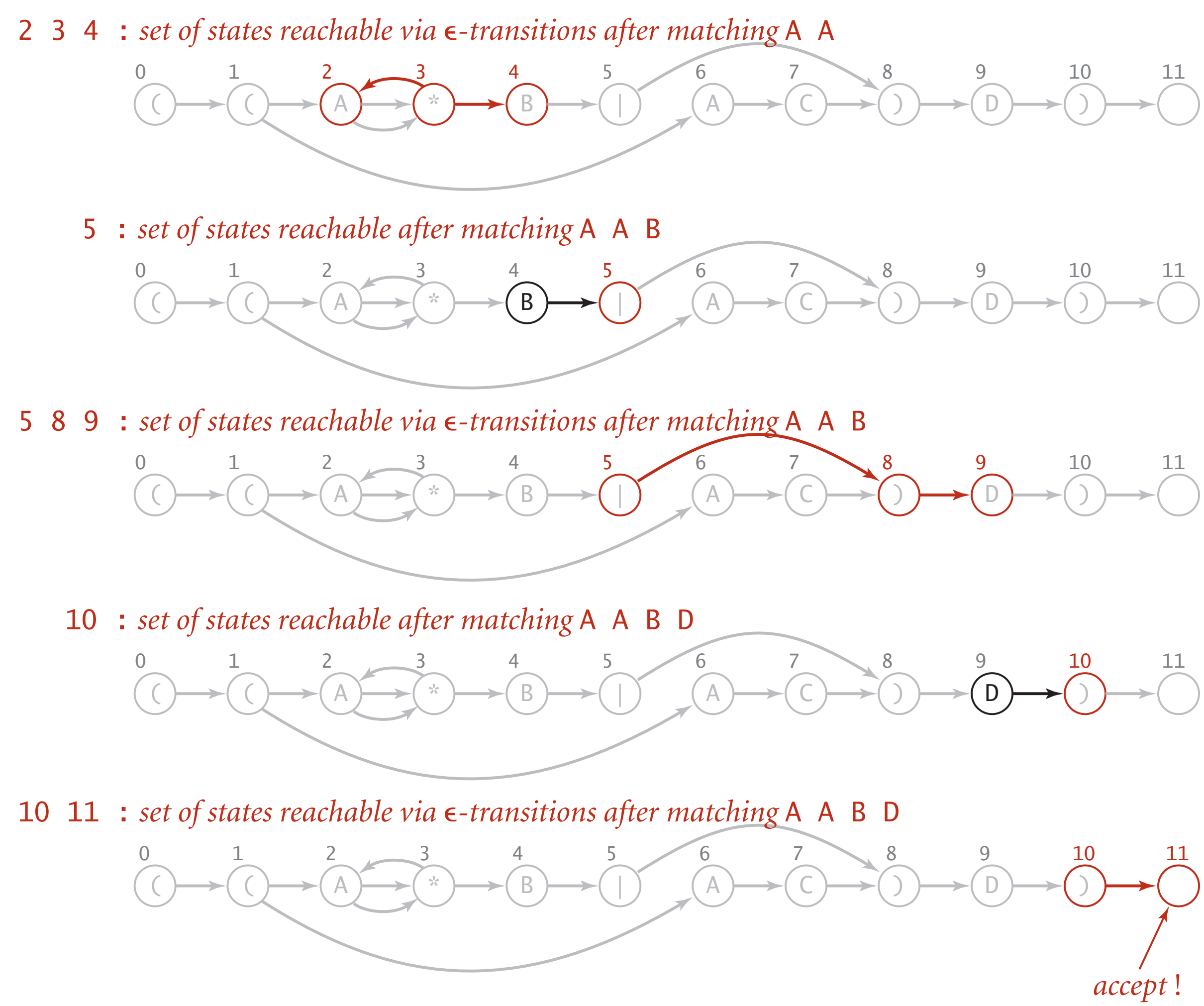
```
public class DFS
{
   private SET<Integer> marked;
   private Digraph G;

   public DFS(Digraph G)
   {  this.G = G; }

   private void search(int v)
   {
      marked.add(v);
      for (int w : G.adj(v))
         if (!marked.contains(w)) search(w);
   }

   public SET<Integer> reachable(SET<Integer> s)
   {
      marked = new SET<Integer>();
      for (int v : s) search(v);
      return marked;
   }
}
```

NFA simulation example



Simulation of ((A * B | A C) D) NFA for input A A B D

NFA simulation example



Simulation of ((A * B | A C) D) NFA for input A A B D

NFA simulation: Java implementation

```
public boolean recognizes(String txt)
{
   DFS dfs = new DFS(G);

   SET<Integer> pc = new dfs.reachable(0);

   for (int i = 0; i < txt.length(); i++)
   {
      SET<Integer> match = new SET<Integer>();
      for (int v : pc) {
         if (v == M) continue;
         if ((re[v] == txt.charAt(i)) || re[v] == '.')
            match.add(v+1);
      }

      pc = dfs.reachable(match);
   }

   return pc.contains(M);
}
```

states reachable from start by ε-transitions

all possible states after scanning past `txt.charAt(i)`

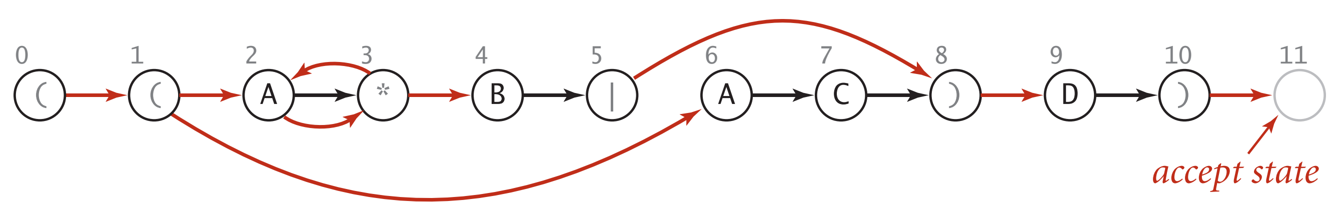
follow ε-transitions

accept if you can end in state M

NFA simulation: analysis

Proposition 1. Determining whether an N-character text string is recognized by the NFA corresponding to an M-character pattern takes time proportional to NM in the worst case.

Pf. For each of the N text characters, we iterate through a set of states of size no more than M and run DFS on the graph of ε-transitions.
(The construction we consider ensures the number of edges is at most M.)

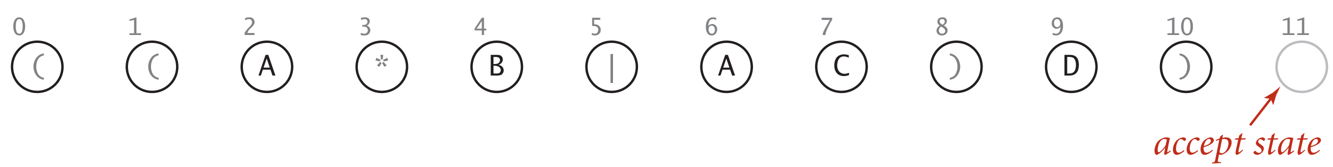


NFA corresponding to the pattern ((A * B | A C) D)

- regular expressions
- NFAs
- NFA simulation
- **NFA construction**
- applications

Building an NFA corresponding to an RE

States. Include a state for each symbol in the RE, plus an accept state.



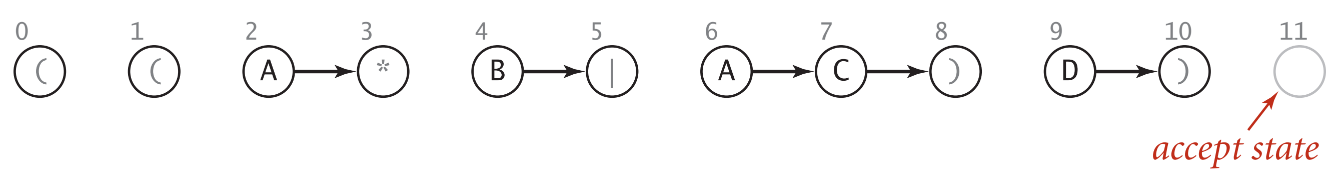
NFA corresponding to the pattern ((A * B | A C) D)

Building an NFA corresponding to an RE

Concatenation. Add match-transition edge from state corresponding to letters in the alphabet to next state.

Alphabet. **A B C D**

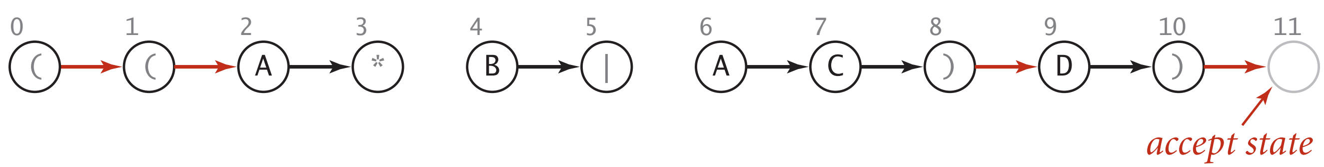
Metacharacters. **( ) . * |**



NFA corresponding to the pattern **((A * B | A C) D)**

Building an NFA corresponding to an RE

Parentheses. Add ε-transition edge from parentheses to next state.

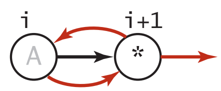


NFA corresponding to the pattern ((A * B | A C) D)

Building an NFA corresponding to an RE

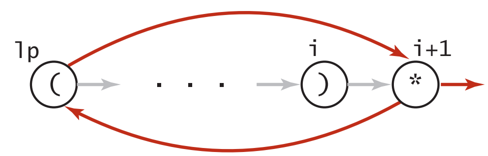
Closure. Add three ε-transition edges for each * operator.

single-character closure

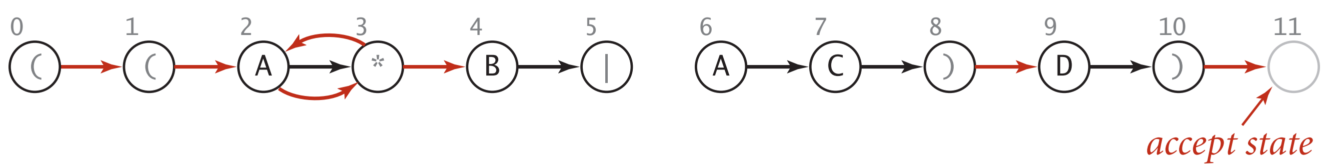



```
G.addEdge(i, i+1);
G.addEdge(i+1, i);
```

closure expression



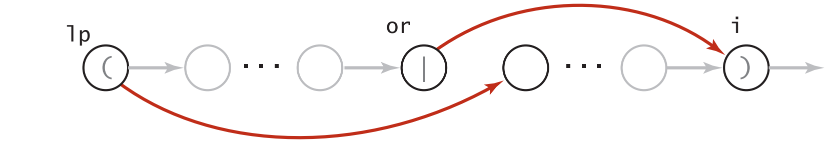

```
G.addEdge(lp, i+1);
G.addEdge(i+1, lp);
```



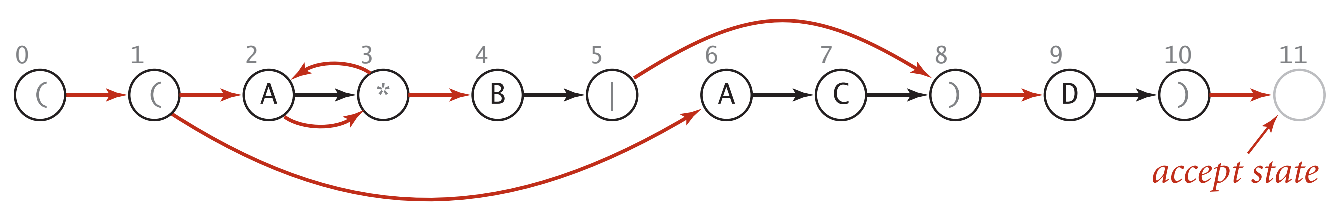
NFA corresponding to the pattern ((A * B | A C) D)

Building an NFA corresponding to an RE

Or. Add two ε-transition edges for each | operator.




```
G.addEdge(lp, or+1);
G.addEdge(or, i);
```



NFA corresponding to the pattern ((A * B | A C) D)

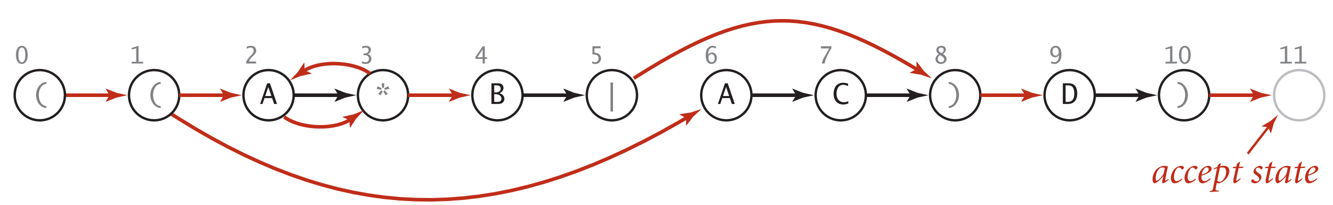
NFA construction: implementation

Goal. Write a program to build the ε-transition digraph.

Challenge. Need to remember left parentheses to implement closure and or; need to remember | to implement or.

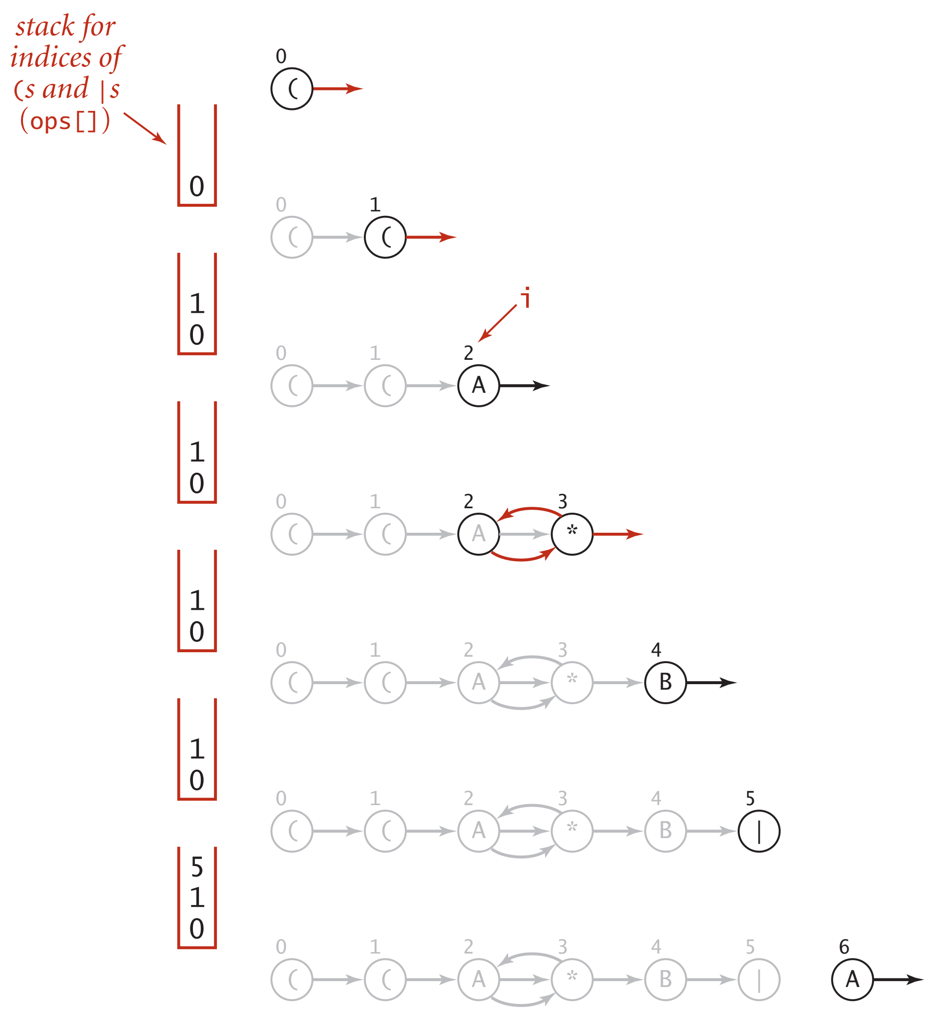
Solution. Maintain a stack.

- Left parenthesis: push onto stack.
- | symbol: push onto stack.
- Right parenthesis: add edges for closure and or.



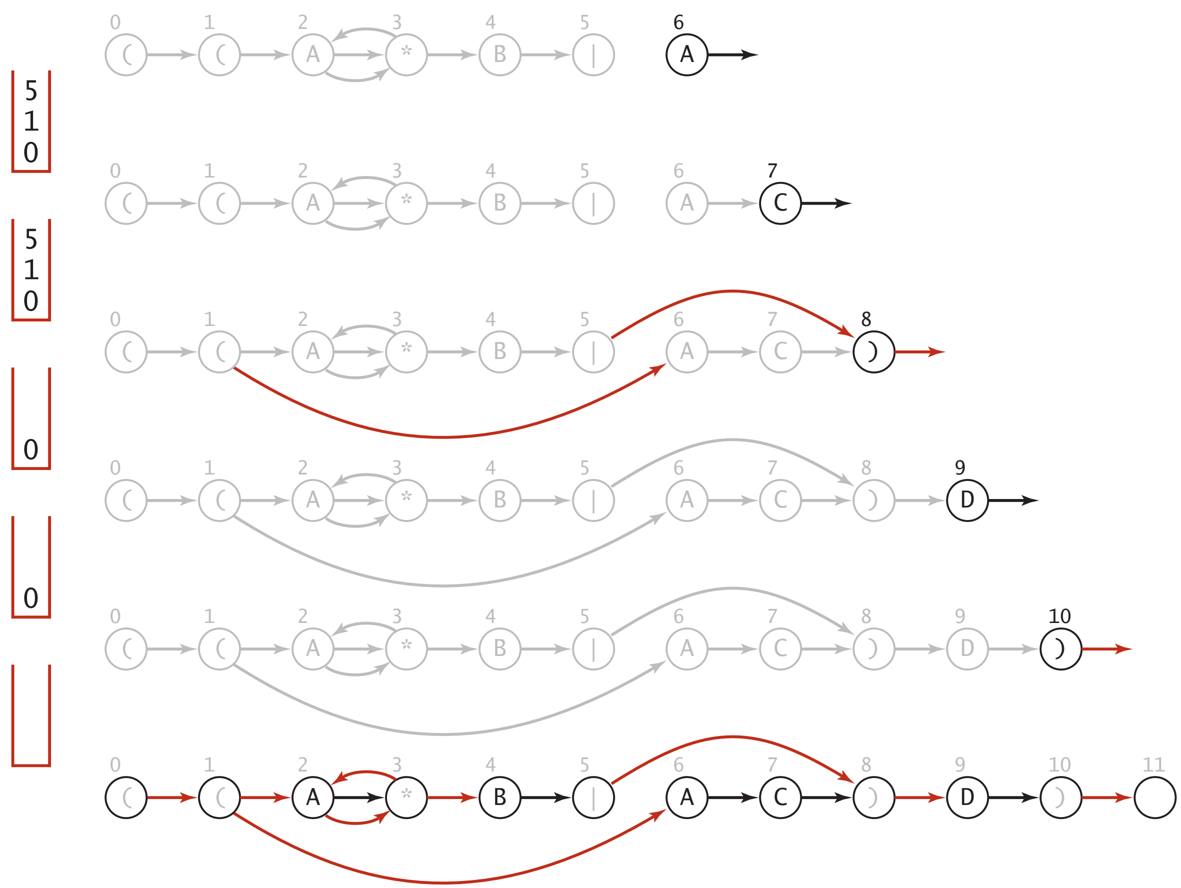
NFA corresponding to the pattern ((A * B | A C) D)

NFA construction: example



Building the NFA corresponding to ((A * B | A C) D)

NFA construction: example



Building the NFA corresponding to ((A * B | A C) D)

NFA construction: Java implementation

```
public NFA(String regexp) {
   Stack<Integer> ops = new Stack<Integer>();
   this.re = re.toCharArray();
   M = re.length;
   G = new Digraph(M+1);
   for (int i = 0; i < M; i++) {
      int lp = i;

      if (re[i] == '(' || re[i] == '|') ops.push(i);

      else if (re[i] == ')') {
         int or = ops.pop();
         if (re[or] == '|') {
            lp = ops.pop();
            G.addEdge(lp, or+1);
            G.addEdge(or, i);
         }
         else lp = or;
      }

      if (i < M-1 && re[i+1] == '*') {
         G.addEdge(lp, i+1);
         G.addEdge(i+1, lp);
      }

      if (re[i] == '(' || re[i] == '*' || re[i] == ')')
         G.addEdge(i, i+1);
   }
}
```

left parentheses and |

or

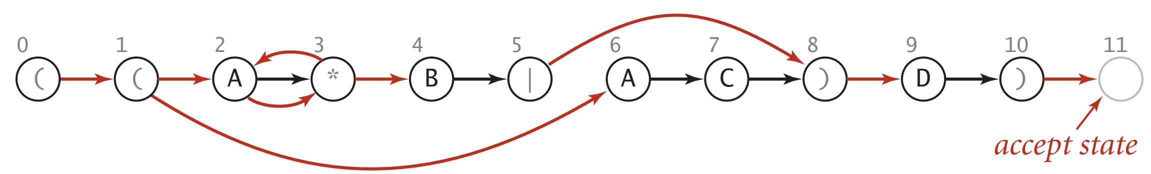
closure
(needs lookahead)

metasymbols

NFA construction: analysis

Proposition 2. Building the NFA corresponding to an M-character pattern takes time and space proportional to M in the worst case.

Pf. For each of the M characters in the pattern, we add one or two ε-transitions and perhaps execute one or two stack operations.



NFA corresponding to the pattern ((A * B | A C) D)

- regular expressions
- NFAs
- NFA simulation
- NFA construction
- **applications**

Generalized regular expression print

Grep. Takes a pattern as a command-line argument and prints the lines from standard input having some substring that is matched by the pattern.

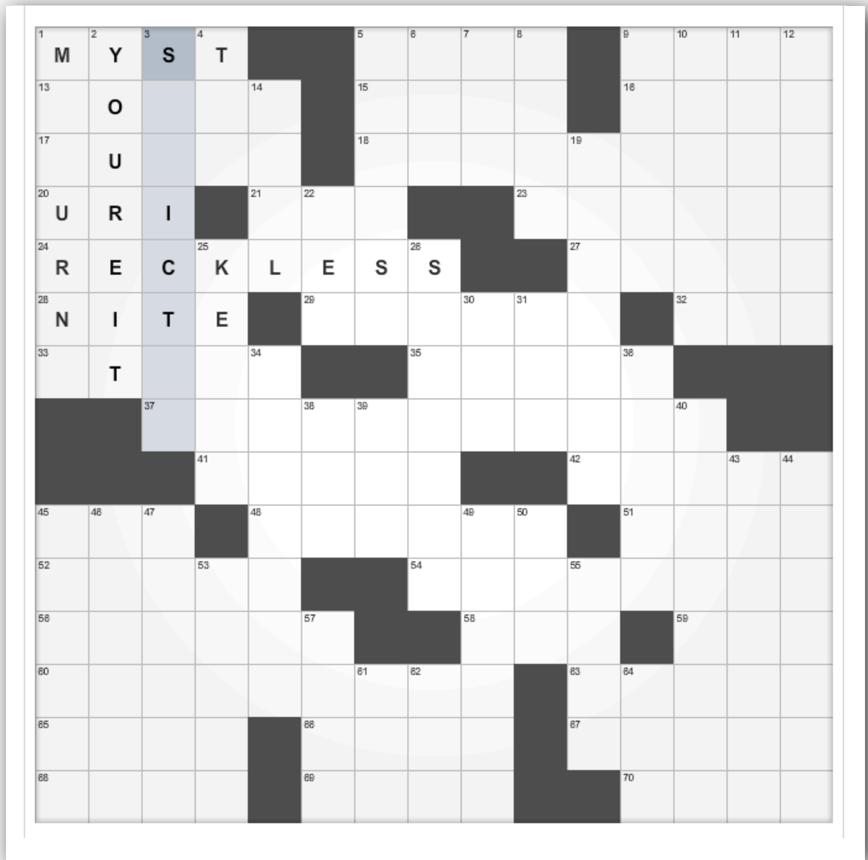
```
public class GREP
{
   public static void main(String[] args)
   {
      String regexp = "(.*" + args[0] + ".*)";
      while (!StdIn.isEmpty())
      {
         String line = StdIn.readLine();
         NFA nfa = new NFA(regexp);
         if (nfa.recognizes(line))
            StdOut.println(line);
      }
   }
}
```

find lines containing RE as a substring

Bottom line. Worst-case for grep (proportional to MN) is the same as for elementary exact substring match.

Typical grep application

Crossword puzzle



dictionary
(standard in UNIX)
also on booksite

```
% more words.txt
a
aback
abacus
abalone
abandon
...

% grep s..ict.. words.txt
constrictor
stricter
stricture
```

Industrial-strength grep implementation

To complete the implementation:

- Add character classes.
- Handling metacharacters.
- Add capturing capabilities.
- Extend the closure operator.
- Error checking and recovery.
- Greedy vs. reluctant matching.

Ex. Which substring(s) should be matched by the RE **<blink>.*</blink>** ?

reluctant reluctant

`<blink>text</blink>some text<blink>more text</blink>`

greedy

Regular expressions in other languages

Broadly applicable programmer's tool.

- Originated in Unix in the 1970s
- Many languages support extended regular expressions.
- Built into grep, awk, emacs, Perl, PHP, Python, JavaScript.

```
% grep NEWLINE */*.java
```

print all lines containing **NEWLINE** which occurs in any file with a **.java** extension

```
% egrep '^[qwertyuiop]*[zxcvbnm]*$' dict.txt | egrep '..........'
```

PERL. Practical Extraction and Report Language.

```
% perl -p -i -e 's|from|to|g' input.txt
```

replace all occurrences of **from** with **to** in the file **input.txt**

```
% perl -n -e 'print if /^[A-Za-z][a-z]*$/' dict.txt
```

print all uppercase words

do for each line

Validity checking. Does the `input` match the `regexp`?

Java string library. Use `input.matches(regexp)` for basic RE matching.

```
public class Validate
{
   public static void main(String[] args)
   {
      String regexp = args[0];
      String input  = args[1];
      StdOut.println(input.matches(regexp));
   }
}
```

```
% java Validate "[$_A-Za-z][$_A-Za-z0-9]*" ident123
true

% java Validate "[a-z]+@([a-z]+\.)+(edu|com)" rs@cs.princeton.edu
true

% java Validate "[0-9]{3}-[0-9]{2}-[0-9]{4}" 166-11-4433
true
```

legal Java identifier

valid email address (simplified)

Social Security number

Harvesting information

Goal. Print all substrings of input that match a RE.

```
% java Harvester "gcg(cgg|agg)*ctg" chromosomeX.txt
gcgcggcggcggcggcggctg
gcgctg
gcgctg
gcgcggcggcggaggcggaggcggctg
```

harvest patterns from DNA

harvest links from website

```
% java Harvester "http://(\\w+\\.)*(\\w+)" http://www.cs.princeton.edu
http://www.princeton.edu
http://www.google.com
http://www.cs.princeton.edu/news
```

Harvesting information

RE pattern matching is implemented in Java's **`Pattern`** and **`Matcher`** classes.

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class Harvester
{
   public static void main(String[] args)
   {
      String regexp   = args[0];
      In in           = new In(args[1]);
      String input    = in.readAll();
      Pattern pattern = Pattern.compile(regexp);
      Matcher matcher = pattern.matcher(input);
      while (matcher.find())
         StdOut.println(matcher.group());
   }
}
```

`compile()` creates a **`Pattern`** (NFA) from RE

`matcher()` creates a **`Matcher`** (NFA simulator) from NFA and text

`find()` looks for the next match

`group()` returns the substring most recently found by **`find()`**

Algorithmic complexity attacks

Warning. Typical implementations do not guarantee performance!

Unix grep, Java, Perl

```
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaac            1.6 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac          3.7 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac        9.7 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac     23.2 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac   62.2 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 161.6 seconds
```

SpamAssassin regular expression.

```
% java RE "[a-z]+@[a-z]+([a-z\.]+\.)+[a-z]+" spammer@x......................
```

- Takes exponential time on pathological email addresses.
- Troublemaker can use such addresses to DOS a mail server.

Not-so-regular expressions

Back-references.

- `\1` notation matches sub-expression that was matched earlier.
- Supported by typical RE implementations.

```
% java Harvester "\b(.+)\1\b"  dictionary.txt
beriberi
couscous
```

word boundary

Some non-regular languages.

- Set of strings of the form ww for some string w: **beriberi**.
- Set of bitstrings with an equal number of 0s and 1s: **01110100**.
- Set of Watson-Crick complemented palindromes: **atttcggaaat**.

Remark. Pattern matching with back-references is intractable.

Context

Abstract machines, languages, and nondeterminism.

- basis of the theory of computation
- intensively studied since the 1930s
- basis of programming languages

Compiler. A program that translates a program to machine code.

- KMP string $\Rightarrow$ DFA.
- `grep` RE $\Rightarrow$ NFA.
- `javac` Java language $\Rightarrow$ Java byte code.

	KMP	grep	Java
pattern	string	RE	program
parser	unnecessary	check if legal	check if legal
compiler output	DFA	NFA	byte code
simulator	DFA simulator	NFA simulator	JVM

Summary of pattern-matching algorithms

Programmer.

- Implement exact pattern matching via DFA simulation.
- Implement RE pattern matching via NFA simulation.

Theoretician.

- RE is a compact description of a set of strings.
- NFA is an abstract machine equivalent in power to RE.
- DFAs and REs have limitations.

You. Practical application of core CS principles.

Example of essential paradigm in computer science.

- Build intermediate abstractions.
- Pick the right ones!
- Solve important practical problems.

5.5 Data Compression

- basics
- run-length encoding
- Huffman compression
- LZW compression

Data compression

Compression reduces the size of a file:

- To save space when storing it.
- To save time when transmitting it.
- Most files have lots of redundancy.

Who needs compression?

- Moore's law: # transistors on a chip doubles every 18-24 months.
- Parkinson's law: data expands to fill space available.
- Text, images, sound, video, …

> "*All of the books in the world contain no more information than is broadcast as video in a single large American city in a single year. Not all bits have equal value.*" — *Carl Sagan*

Basic concepts ancient (1950s), best technology recently developed.

Applications

Generic file compression.

- Files: GZIP, BZIP, BOA.
- Archivers: PKZIP.
- File systems: NTFS.

Multimedia.

- Images: GIF, JPEG.
- Sound: MP3.
- Video: MPEG, DivX™, HDTV.

Communication.

- ITU-T T4 Group 3 Fax.
- V.42bis modem.

Databases. Google.

Lossless compression and expansion

uses fewer bits (you hope)

Message. Binary data B we want to compress.

Compress. Generates a "compressed" representation $C(B)$.

Expand. Reconstructs original bitstream B.

Basic model for data compression

Compression ratio. Bits in $C(B)$ / bits in B.

Ex. 50-75% or better compression ratio for natural language.

Food for thought

Data compression has been omnipresent since antiquity:

- Number systems.
- Natural languages.
- Mathematical notation.

has played a central role in communications technology,

- Braille.
- Morse code.
- Telephone system.

and is part of modern life.

- MP3.
- MPEG.

Q. What role will it play in the future?

- binary I/O
- genomic encoding
- run-length encoding
- Huffman compression
- LZW compression

Reading and writing binary data

Binary standard input and standard output. Libraries to read and write bits from standard input and to standard output.

`public class BinaryStdIn`

`boolean`	`readBoolean()`	*read 1 bit of data and return as a* `boolean` *value*
`char`	`readChar()`	*read 8 bits of data and return as a* `char` *value*
`char`	`readChar(int r)`	*read* `r` *bits of data and return as a* `char` *value*
[similar methods for `byte` *(8 bits);* `short` *(16 bits);* `int` *(32 bits);* `long` *and* `double` *(64 bits)]*		
`boolean`	`isEmpty()`	*is the bitstream empty?*
`void`	`close()`	*close the bitstream*

`public class BinaryStdOut`

`void`	`write(boolean b)`	*write the specified bit*
`void`	`write(char c)`	*write the specified 8-bit* `char`
`void`	`write(char c, int r)`	*write the* `r` *least significant bits of the specified* `char`
[similar methods for `byte` *(8 bits);* `short` *(16 bits);* `int` *(32 bits);* `long` *and* `double` *(64 bits)]*		
`void`	`close()`	*close the bitstream*

Writing binary data

Date representation. Different ways to represent 12/31/1999.

A character stream (StdOut)

```
StdOut.print(month + "/" + day + "/" + year);
```

`00110001 00110010 00101111 00110111 00110001 00101111 00110001 00111001 00111001 00111001`

1 2 / 3 1 / 1 9 9 9 — *80 bits*

Three ints (BinaryStdOut)

```
BinaryStdOut.write(month);
BinaryStdOut.write(day);
BinaryStdOut.write(year);
```

`00000000 00000000 00000000 00001100 00000000 00000000 00000000 00011111 00000000 00000000 00000111 11001111`

12 31 1999 — *96 bits*

Two chars and a short (BinaryStdOut)

```
BinaryStdOut.write((char) month);
BinaryStdOut.write((char) day);
BinaryStdOut.write((short) year);
```

`00001100 00011111 00000111 11001111`

12 31 1999 — *32 bits*

A 4-bit field, a 5-bit field, and a 12-bit field (BinaryStdOut)

```
BinaryStdOut.write(month, 4);
BinaryStdOut.write(day, 5);
BinaryStdOut.write(year, 12);
```

`11001111 10111110 01111000`

12 31 1999 — *21 bits (+ 3 bits for byte alignment at close)*

Four ways to put a date onto standard output

Binary dumps

Q. How to examine the contents of a bitstream?

Standard character stream

```
% more abra.txt
ABRACADABRA!
```

Bitstream represented as 0 and 1 characters

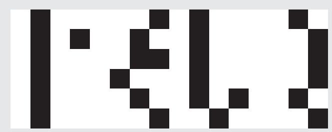
```
% java BinaryDump 16 < abra.txt
0100000101000010
0101001001000001
0100001101000001
0100010001000001
0100001001010010
0100000100100001
96 bits
```

Bitstream represented with hex digits

```
%  java HexDump 4 < abra.txt
41 42 52 41
43 41 44 41
42 52 41 21
96 bits
```

Bitstream represented as pixels in a `Picture`

```
% java PictureDump 16 < abra.txt
```



16-by-6 pixel window, magnified

```
96 bits
```

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	“	#	$	%	&	‘	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Hexadecimal to ASCII conversion table

Universal data compression

US Patent 5,533,051 on "Methods for Data Compression", which is capable of compression all files.

Slashdot reports of the Zero Space Tuner™ and BinaryAccelerator™.

> *" ZeoSync has announced a breakthrough in data compression that allows for 100:1 lossless compression of random data. If this is true, our bandwidth problems just got a lot smaller.… "*

Universal data compression

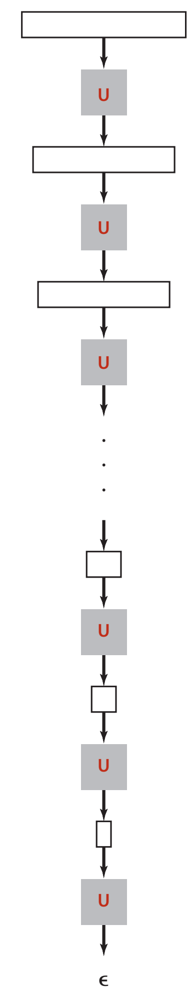
Proposition. No algorithm can compress every bitstring.

Pf 1. [by contradiction]

- Suppose you have a universal data compression algorithm U that can compress every bitstream.
- Given bintstring B_0, compress it to get smaller bitstring B_1.
- Compress B_1 to get a smaller bitstring B_2.
- Continue until reaching bitstring of size 0.
- Implication: all bitstrings can be compressed with 0 bits!

Pf 2. [by counting]

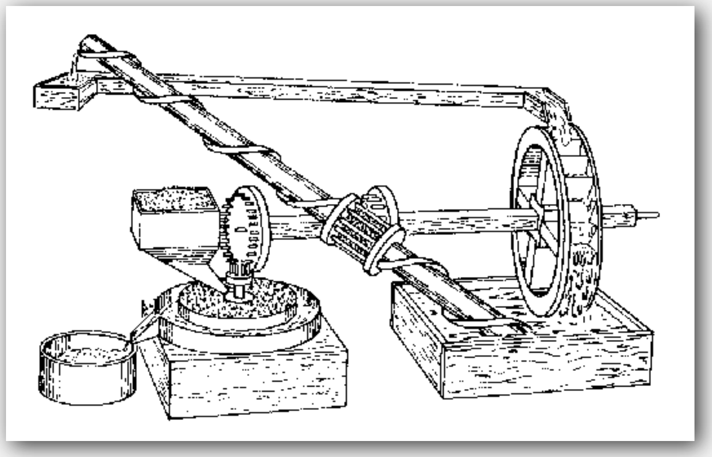
- Suppose your algorithm that can compress all 1,000-bit strings.
- 2^{1000} possible bitstrings with 1000 bits.
- Only $1 + 2 + 4 + \ldots + 2^{998} + 2^{999}$ can be encoded with ≤ 999 bits.
- Similarly, only 1 in 2^{499} bitstrings can be encoded with ≤ 500 bits!



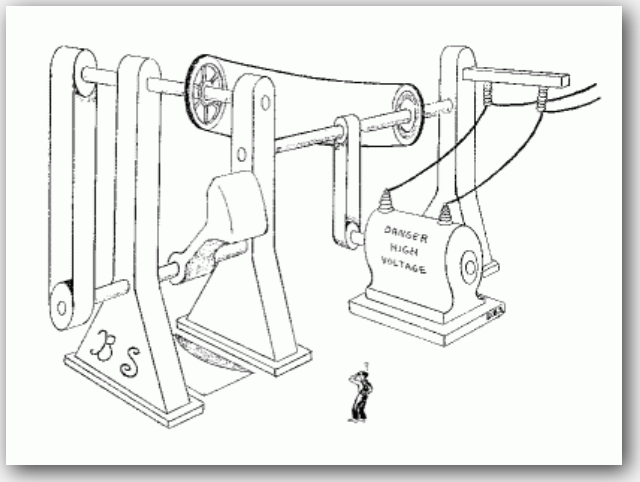
Universal data compression?

Perpetual motion machines

Universal data compression is the analog of perpetual motion.



Closed-cycle mill by Robert Fludd, 1618

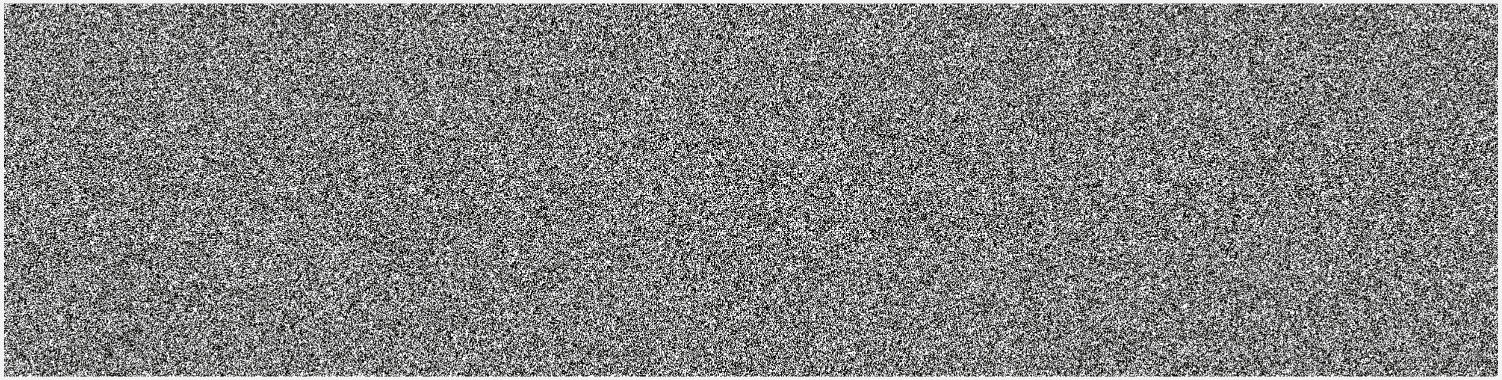


Gravity engine by Bob Schadewald

Reference: Museum of Unworkable Devices by Donald E. Simanek
http://www.lhup.edu/~dsimanek/museum/unwork.htm

Undecidability

```
% java RandomBits | java PictureDump 2000 500
```



```
1000000 bits
```

A difficult file to compress: one million (pseudo-) random bits

```
public class RandomBits
{
   public static void main(String[] args)
   {
      int x = 11111;
      for (int i = 0; i < 1000000; i++)
      {
         x = x * 314159 + 218281;
         BinaryStdOut.write(x > 0);
      }
      BinaryStdOut.close();
   }
}
```

Q. How much redundancy is in the English language?

" ... randomising letters in the middle of words [has] little or no effect on the ability of skilled readers to understand the text. This is easy to denmtrasote. In a pubiltacion of New Scnieitst you could ramdinose all the letetrs, keipeng the first two and last two the same, and reibadailty would hadrly be aftcfeed. My ansaylis did not come to much beucase the thoery at the time was for shape and senqeuce retigcionon. Saberi's work sugsegts we may have some pofrweul palrlael prsooscers at work. The resaon for this is suerly that idnetiyfing coentnt by paarllel prseocsing speeds up regnicoiton. We only need the first and last two letetrs to spot chganes in meniang." — *Graham Rawlinson*

A. Quite a bit.

- genomic encoding
- run-length encoding
- Huffman compression
- LZW compression

Genomic code

Genome. String over the alphabet { A, C, T, G }.

Goal. Encode an N-character genome: **ATAGATGCATAG...**

Standard ASCII encoding.

- 8 bits per char.
- 8N bits.

char	hex	binary
A	41	01000001
C	43	01000011
T	54	01010100
G	47	01000111

Two-bit encoding encoding.

- 2 bits per char.
- 2N bits.

char	binary
A	00
C	01
T	10
G	11

Amazing but true. Initial genomic databases in 1990s did not use such a code!

Fixed-length code. k-bit code supports alphabet of size 2^k.

Genomic code

```
public class Genome {

   public static void compress() {
      Alphabet DNA = new Alphabet("ACTG");
      String s = BinaryStdIn.readString();
      int N = s.length();
      BinaryStdOut.write(N);
      for (int i = 0; i < N; i++) {
         int d = DNA.toIndex(s.charAt(i));
         BinaryStdOut.write(d, 2);
      }
      BinaryStdOut.close();
   }

   public static void expand() {
      Alphabet DNA = new Alphabet("ACTG");
      int N = BinaryStdIn.readInt();
      for (int i = 0; i < N; i++) {
         char c = BinaryStdIn.readChar(2);
         BinaryStdOut.write(DNA.toChar(c));
      }
      BinaryStdOut.close();
   }
}
```

`Alphabet` data type converts between symbols { A, C, T, G } and integers 0—3.

read genomic string from stdin; write to stdout using 2-bit code

read 2-bit code from stdin; write genomic string to stdout

Genomic code: test client and sample execution

```
public static void main(String[] args)
{
    if (args[0].equals("-")) compress();
    if (args[0].equals("+")) expand();
}
```

Tiny test case (264 bits)

```
% more genomeTiny.txt
ATAGATGCATAGCGCATAGCTAGATGTGCTAGC

java BitsDump 64 < genomeTiny.txt
0100000101010100010000010100011101000001010101000100011101000011
0100000101010100010000010100011101000011010001110100001101000001
0101010001000001010001110100001101010100010000010100011101000001
0101010001000111010101000100011101000011010101000100000101000111
01000011
264 bits

% java Genome - < genomeTiny.txt
??
```

cannot see bitstream on standard output

```
% java Genome - < genomeTiny.txt | java BinaryDump 64
0000000000000000000000000010000100100011001011010010001101110100
10001101100011001011101101100011010000000
104 bits

% java Genome - < genomeTiny.txt | java HexDump 8
00 00 00 21 23 2d 23 74
8d 8c bb 63 40
104 bits

% java Genome - < genomeTiny.txt | java Genome +
ATAGATGCATAGCGCATAGCTAGATGTGCTAGC
```

compress-expand cycle produces original input

- genomic encoding
- **run-length encoding**
- Huffman compression
- LZW compression

Run-length encoding

Simple type of redundancy in a bitstream. Long runs of repeated bits.

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1
```

Representation. Use 4-bit counts to represent alternating runs of 0s and 1s: 15 0s, then 7 1s, then 7 0s, then 11 1s.

$$\underbrace{1\,1\,1\,1}_{15}\;\underbrace{0\,1\,1\,1}_{7}\;\underbrace{0\,1\,1\,1}_{7}\;\underbrace{1\,0\,1\,1}_{11} \longleftarrow \text{16 bits (instead of 40)}$$

Q. How many bits to store the counts?

A. We'll use 8.

Q. What to do when run length exceeds max count?

A. If longer than 255, intersperse runs of length 0.

Applications. JPEG, ITU-T T4 Group 3 Fax, ...

Run-length encoding: Java implementation

```
public class RunLength
{
   private final static int R = 256;

   public static void compress()
   {  /* see textbook */   }

   public static void expand()
   {
      boolean b = false;
      while (!BinaryStdIn.isEmpty())
      {
         char run = BinaryStdIn.readChar();
         for (int i = 0; i < run; i++)
            BinaryStdOut.write(b);
         b = !b;
      }
      BinaryStdOut.close();
   }

}
```

read 8-bit count from standard input

write 1 bit to standard output

An application: compress a bitmap

Typical black-and-white-scanned image.

- 300 pixels/inch.
- 8.5-by-11 inches.
- 300 × 8.5 × 300 × 11 = 8.415 million bits.

Observation. Bits are mostly white.

Typical amount of text on a page.

40 lines × 75 chars per line = 3,000 chars.

7 1s

```
% java BinaryDump 32 < q32x48.bin
00000000000000000000000000000000     32
00000000000000000000000000000000     32
00000000000000011111110000000000     15  7 10
00000000000011111111111111100000     12 15  5
00000000001111000011111111100000     10  4  4  9  5
00000000111100000000011111100000      8  4  9  6  5
00000001110000000000001111100000      7  3 12  5  5
00000011110000000000001111100000      6  4 12  5  5
00000111100000000000001111100000      5  4 13  5  5
00001111000000000000001111100000      4  4 14  5  5
00001111000000000000001111100000      4  4 14  5  5
00011110000000000000001111100000      3  4 15  5  5
00011110000000000000001111100000      2  5 15  5  5
00111110000000000000001111100000      2  5 15  5  5
00111110000000000000001111100000      2  5 15  5  5
00111110000000000000001111100000      2  5 15  5  5
00111110000000000000001111100000      2  5 15  5  5
00111110000000000000001111100000      2  5 15  5  5
00111110000000000000001111100000      2  5 15  5  5
00111110000000000000001111100000      2  5 15  5  5
00111110000000000000001111100000      2  5 15  5  5
00111111000000000000001111100000      2  6 14  5  5
00111111000000000000001111100000      2  6 14  5  5
00011111100000000000001111100000      3  6 13  5  5
00011111100000000000001111100000      3  6 13  5  5
00001111110000000000001111100000      4  6 12  5  5
00001111111000000000001111100000      4  7 11  5  5
00000111111100000000001111100000      5  7 10  5  5
00000011111111000000011111100000      6  8  7  6  5
00000001111111111111111111100000      7 20  5
00000000011111111111001111100000      9 11  2  5  5
00000000000011111000001111100000     22  5  5
00000000000000000000001111100000     22  5  5
00000000000000000000001111100000     22  5  5
00000000000000000000001111100000     22  5  5
00000000000000000000001111100000     22  5  5
00000000000000000000001111100000     22  5  5
00000000000000000000001111100000     22  5  5
00000000000000000000001111100000     22  5  5
00000000000000000000001111100000     22  5  5
00000000000000000000001111100000     22  5  5
00000000000000000000001111100000     22  5  5
00000000000000000000001111100000     22  5  5
00000000000000000000011111110000     21  7  4
00000000000000000011111111111100     18 12  2
00000000000000000111111111111110     17 14  1
00000000000000000000000000000000     32
00000000000000000000000000000000     32
1536 bits
```

17 0s

A typical bitmap, with run lengths for each row

- genomic encoding
- run-length encoding
- **Huffman compression**
- LZW compression

Variable-length codes

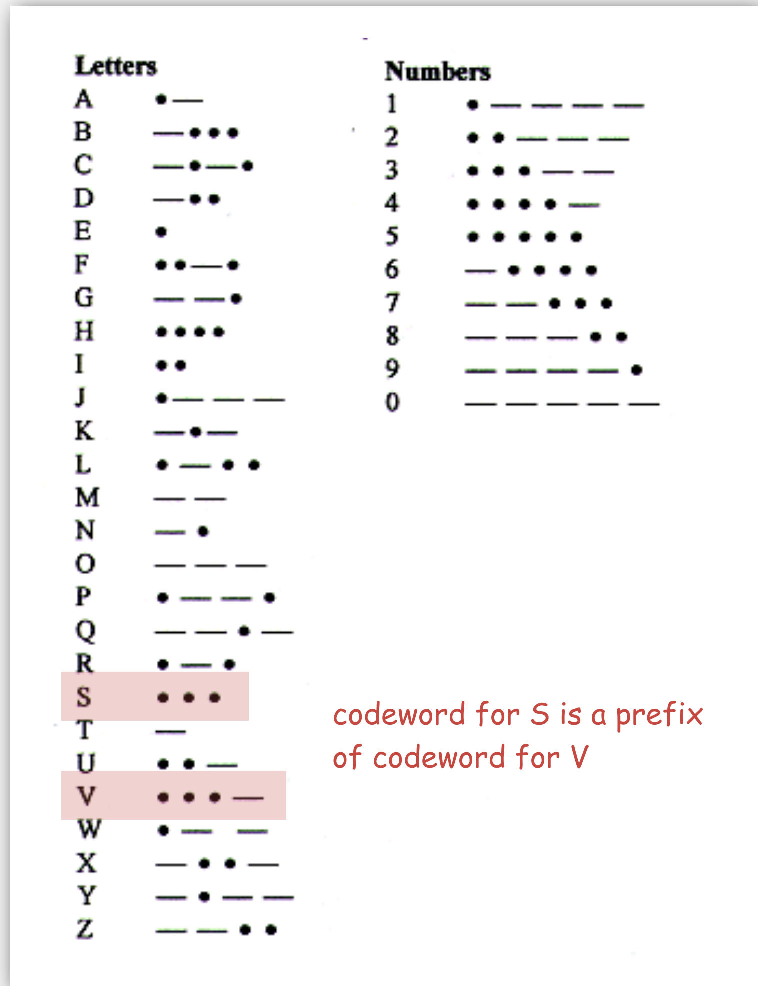
Use different number of bits to encode different chars.

Ex. Morse code: • • • – – – • • •

Issue. Ambiguity.

```
SOS ?
IAMIE ?
EEWNI ?
V7 ?
```

In practice. Use a medium gap to separate codewords.



Letters		Numbers	
A	•—	1	•————
B	—•••	2	••———
C	—•—•	3	•••——
D	—••	4	••••—
E	•	5	•••••
F	••—•	6	—••••
G	——•	7	——•••
H	••••	8	———••
I	••	9	————•
J	•———	0	—————
K	—•—		
L	•—••		
M	——		
N	—•		
O	———		
P	•——•		
Q	——•—		
R	•—•		
S	•••		
T	—		
U	••—		
V	•••—		
W	•——		
X	—••—		
Y	—•——		
Z	——••		

codeword for S is a prefix of codeword for V

Variable-length codes

Q. How do we avoid ambiguity?

A. Ensure that no codeword is a prefix of another.

Ex 1. Fixed-length code.

Ex 2. Append special stop char to each codeword.

Ex 3. General prefix-free code.

Codeword table

key	*value*
!	101
A	0
B	1111
C	110
D	100
R	1110

Compressed bitstring

011111110011001000111111100101 ← *30 bits*

A B RA CA DA B RA !

Codeword table

key	*value*
!	101
A	11
B	00
C	010
D	100
R	011

Compressed bitstring

11000111101011100110001111101 ← *29 bits*

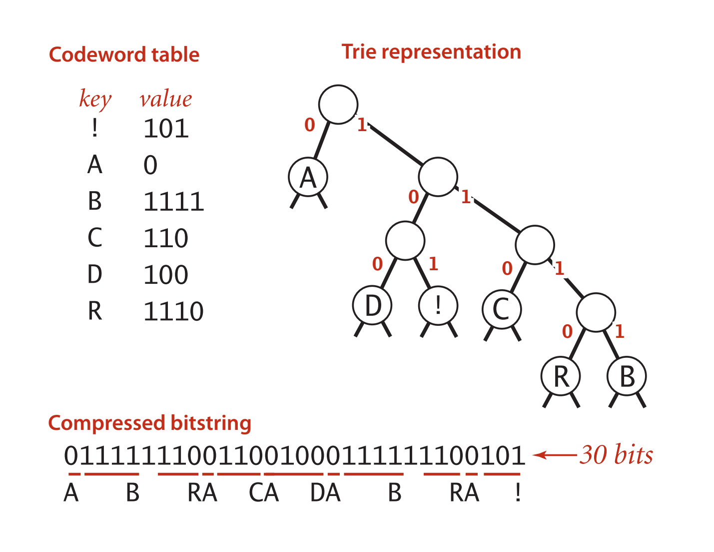
A B R A C A D A B R A !

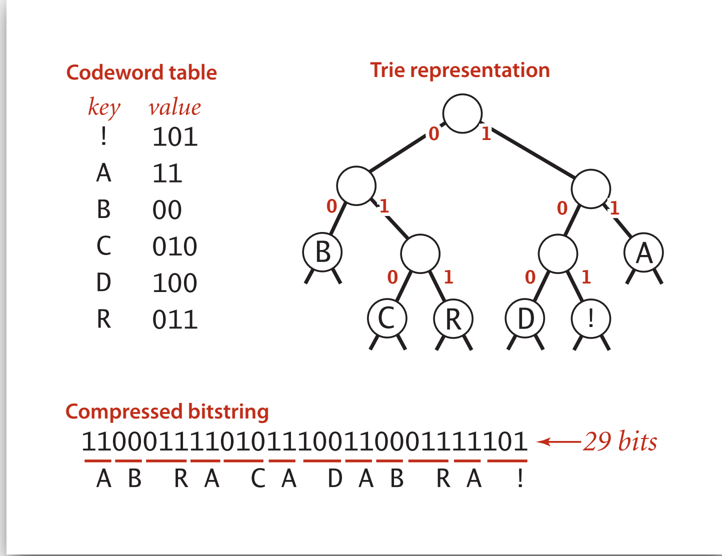
Prefix-free codes: trie representation

Q. How to represent the prefix-free code?

A. A binary trie!

- Chars in leaves.
- Codeword is path from root to leaf.





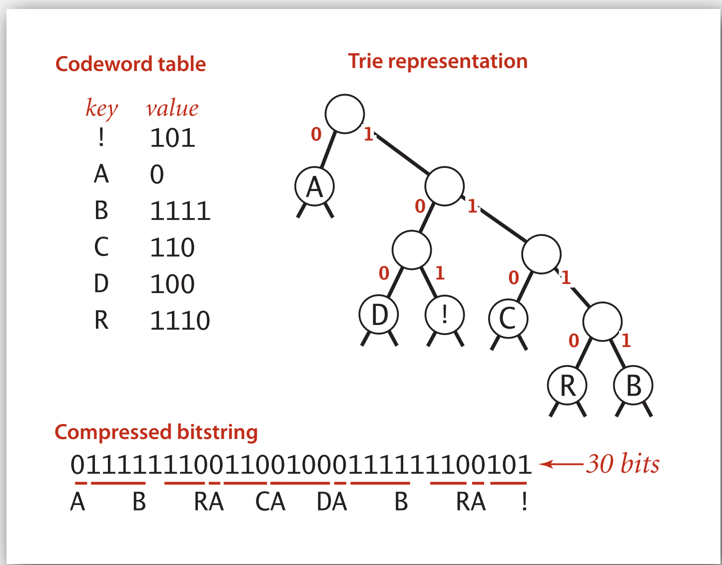
Prefix-free codes: compression and expansion

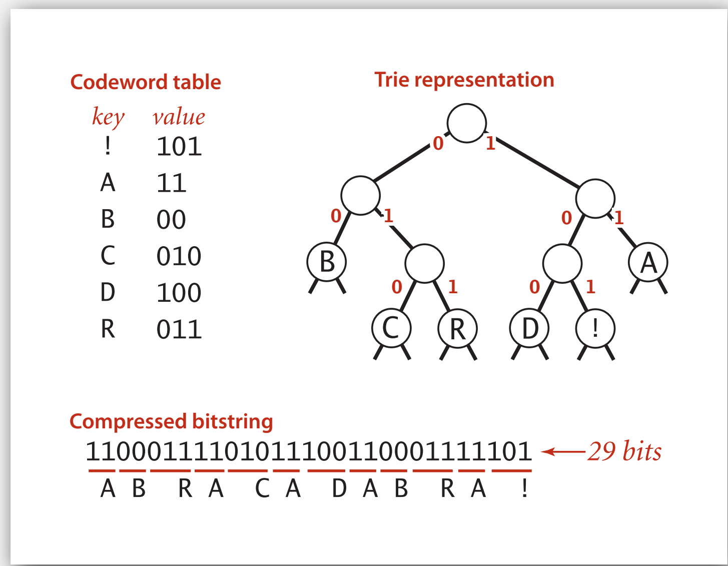
Compression.

- Method 1: start at leaf; follow path up to the root; print bits in reverse.
- Method 2: create ST of key-value pairs.

Expansion.

- Start at root.
- Go left if bit is 0; go right if 1.
- If leaf node, print char and return to root.





Huffman trie node data type

```
private static class Node implements Comparable<Node>
{
   private char ch;   // Unused for internal nodes.
   private int freq;  // Unused for expand.
   private final Node left, right;

   public Node(char ch, int freq, Node left, Node right)
   {
      this.ch    = ch;
      this.freq  = freq;
      this.left  = left;
      this.right = right;
   }

   public boolean isLeaf()
   {  return left == null && right == null; }

   public int compareTo(Node that)
   {  return this.freq - that.freq;  }
}
```

Prefix-free codes: expansion

```
public void expand()
{
   Node root = readTrie();
   int N = BinaryStdIn.readInt();

   for (int i = 0; i < N; i++)
   {
      Node x = root;
      while (!x.isLeaf())
      {
         if (BinaryStdIn.readBoolean())
            x = x.left;
         else
            x = x.right;
      }
      BinaryStdOut.write(x.ch);
   }
   BinaryStdOut.close();
}
```

read in encoding trie

read in number of chars

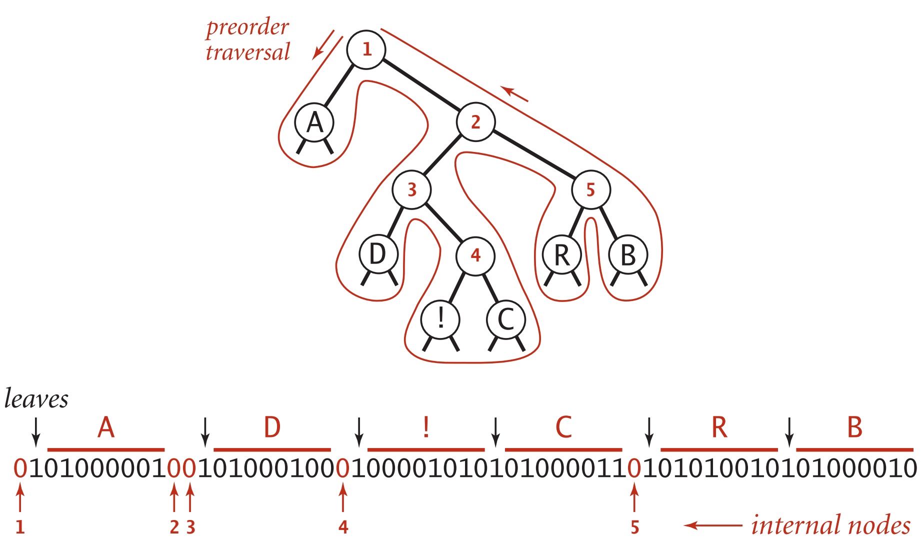
expand codeword for i^{th} char

Running time. Linear in input size (constant amount of work per bit read).

Prefix-free codes: how to transmit

Q. How to write the trie?

A. Write preorder traversal of trie; mark leaf and internal nodes with a bit.



Using preorder traversal to encode a trie as a bitstream

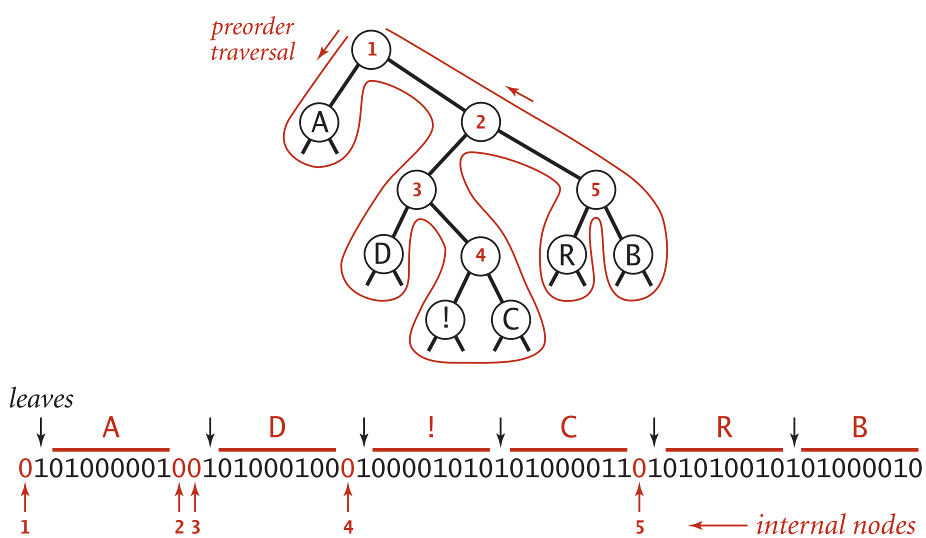
```
private static void writeTrie(Node x)
{
   if (x.isLeaf())
   {
      BinaryStdOut.write(true);
      BinaryStdOut.write(x.ch);
      return;
   }
   BinaryStdOut.write(false);
   writeTrie(x.left);
   writeTrie(x.right);
}
```

Note. If message is long, overhead of transmitting trie is small.

Prefix-free codes: how to transmit

Q. How to read in the trie?

A. Reconstruct from preorder traversal of trie.



Using preorder traversal to encode a trie as a bitstream

```
private static Node readTrie()
{
   if (BinaryStdIn.readBoolean())
   {
      char c = BinaryStdIn.readChar();
      return new Node(c, 0, null, null);
   }
   Node x = readTrie();
   Node y = readTrie();
   return new Node('\0', 0, x, y);
}
```

Q. How to find best prefix-free code?

A. Huffman algorithm.

David Huffman

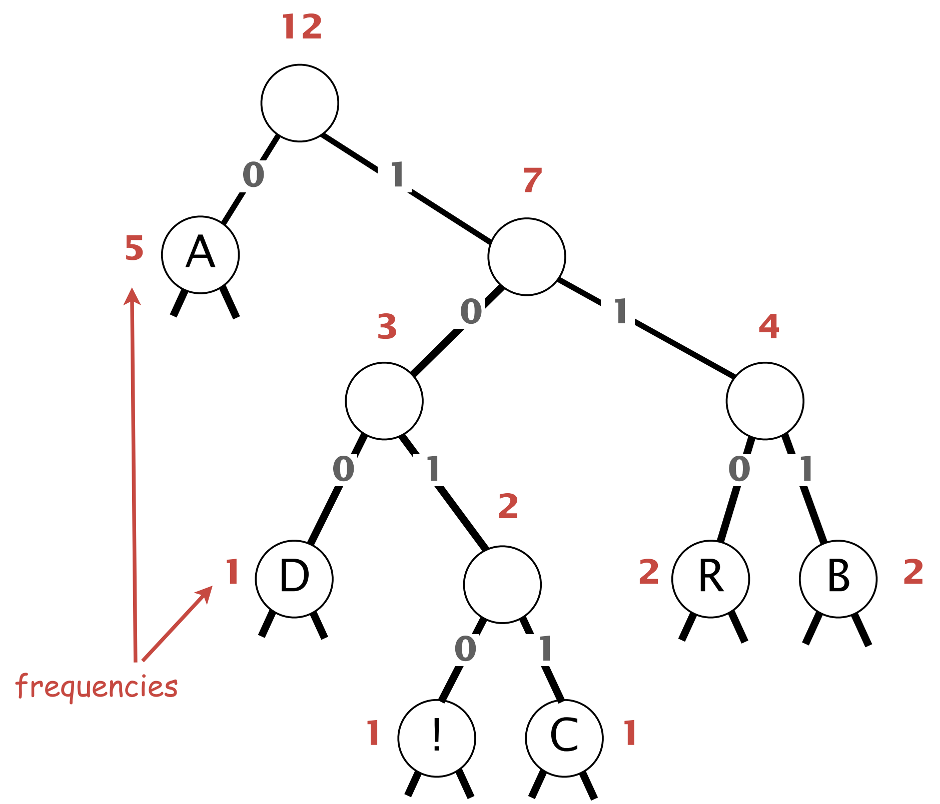
Huffman algorithm (to compute optimal prefix-free code):

- Count frequency `freq[i]` for each char `i` in input.
- Start with one node corresponding to each char `i` (with weight `freq[i]`).
- Repeat until single trie formed:
  - select two tries with min weight `freq[i]` and `freq[j]`
  - merge into single trie with weight `freq[i] + freq[j]`

Applications. JPEG, MP3, MPEG, PKZIP, GZIP, …

Constructing a Huffman encoding trie

char	freq	encoding
A	5	0
B	2	1 1 1
C	1	1 0 1 1
D	1	1 0 0
R	2	1 1 0
!	1	1 0 1 0



Huffman code construction for A B R A C A D A B R A !

Constructing a Huffman encoding trie: Java implementation

```
private static Node buildTrie(int[] freq)
{
    MinPQ<Node> pq = new MinPQ<Node>();
    for (char i = 0; i < R; i++)
       if (freq[i] > 0)
          pq.insert(new Node(i, freq[i], null, null));

    while (pq.size() > 1)
    {
       Node x = pq.delMin();
       Node y = pq.delMin();
       Node parent = new Node('\0', x.freq + y.freq, x, y);
       pq.insert(parent);
    }

    return pq.delMin();
}
```

initialize PQ with singleton tries

merge two smallest tries

not used

total frequency

two subtries

Huffman encoding summary

Proposition. [Huffman 1950s] Huffman algorithm produces an optimal prefix-free code.

no prefix-free code uses fewer bits

Pf. See textbook.

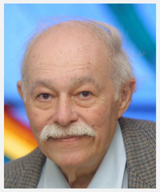
Implementation.

- Pass 1: tabulate char frequencies and build trie.
- Pass 2: encode file by traversing trie or lookup table.

Running time. Using a binary heap $\Rightarrow$ $O(\underset{\text{input size}}{N} + \underset{\text{alphabet size}}{R} \log R)$.

Q. Can we do better? [stay tuned]

- genomic encoding
- run-length encoding
- Huffman compression
- **LZW compression**



Abraham Lempel

Jacob Ziv

Statistical methods

Static model. Same model for all texts.

- Fast.
- Not optimal: different texts have different statistical properties.
- Ex: ASCII, Morse code.

Dynamic model. Generate model based on text.

- Preliminary pass needed to generate model.
- Must transmit the model.
- Ex: Huffman code.

Adaptive model. Progressively learn and update model as you read text.

- More accurate modeling produces better compression.
- Decoding must start from beginning.
- Ex: LZW.

Lempel-Ziv-Welch compression example

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B	R	A	C	A	D	A B		R A		B R		A B R			A
value	41	42	52	41	43	41	44	81		83		82		88			41

LZW compression for ABRACADABRABRABRA

key	value
...	
A	41
B	42
C	43
D	44
...	

key	value
AB	81
BR	82
RA	83
AC	84
CA	85
AD	86

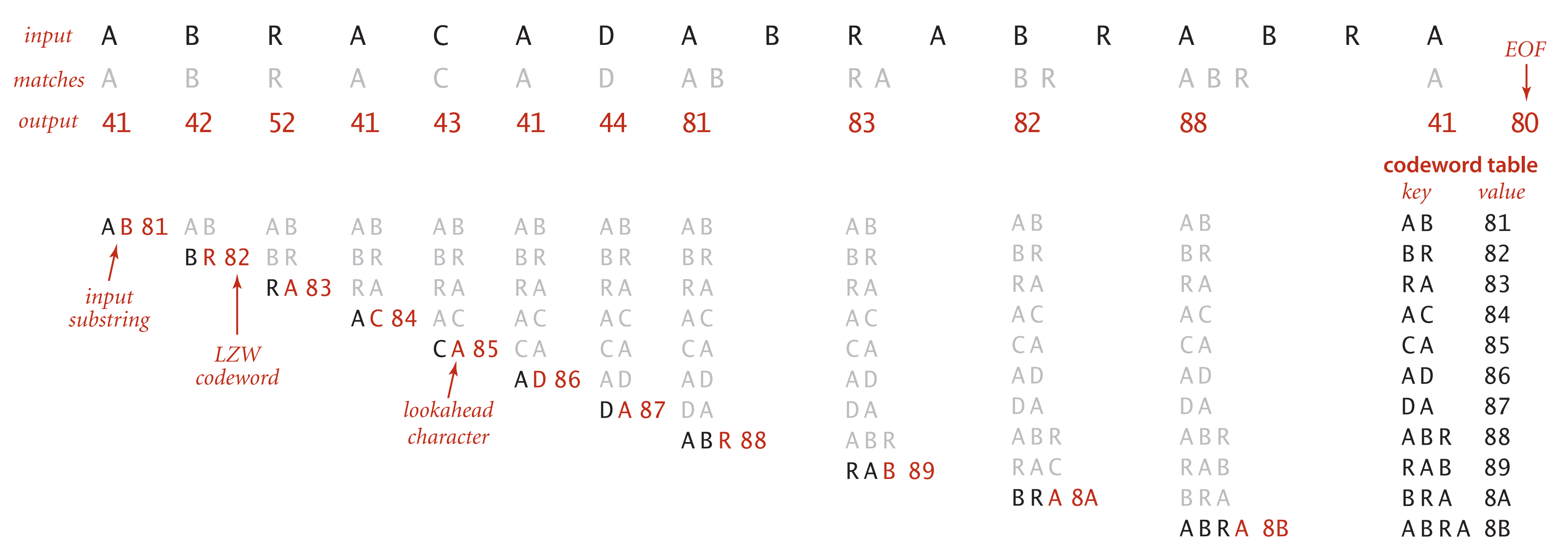
key	value
DA	87
ABR	88
RAB	89
BRA	8A
ABRA	8B

codeword table

Lempel-Ziv-Welch compression

LZW compression.

- Create ST associating W-bit codewords with string keys.
- Initialize ST with codewords for single-char keys.
- Find longest string s in ST that is a prefix of unscanned part of input.
- Write the W-bit codeword associated with s.
- Add s + c to ST, where c is next char in the input.

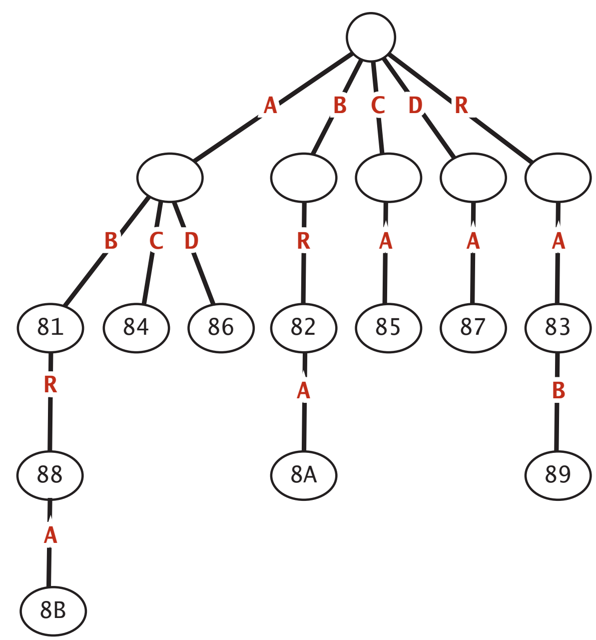


LZW compression for A B R A C A D A B R A B R A B R A

Representation of LZW code table

Q. How to represent LZW code table?

A. A trie: supports efficient longest prefix match.



Trie representation of LZW code table

Remark. Every prefix of a key in encoding table is also in encoding table.

LZW compression: Java implementation

```
public static void compress()
{
  String input = BinaryStdIn.readString();

   TST<Integer> st = new TST<Integer>();
   for (int i = 0; i < R; i++)
       st.put("" + (char) i, i);
   int code = R+1;

   while (input.length() > 0)
   {
      String s = st.longestPrefixOf(input);
      BinaryStdOut.write(st.get(s), W);
      int t = s.length();
      if (t < input.length() && code < L)
          st.put(input.substring(0, t+1), code++);
      input = input.substring(t);
   }

   BinaryStdOut.write(R, W);
   BinaryStdOut.close();
}
```

read in input as a string

codewords for single-char, radix R keys

find longest prefix match s

write W-bit codeword for s

add new codeword

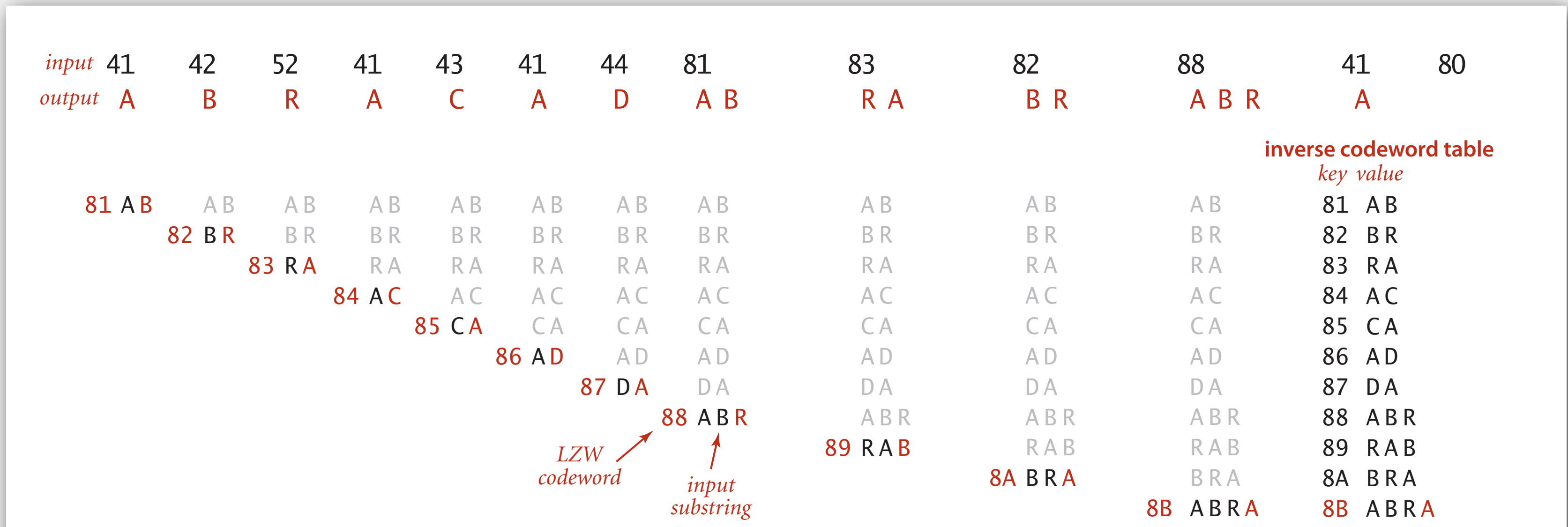
scan past s in input

write last codeword and close input stream

LZW expansion

LZW expansion.

- Create ST associating string values with W-bit keys.
- Initialize ST to contain with single-char values.
- Read a W-bit key.
- Find associated string value in ST and write it out.
- Update ST.



LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

LZW expansion: tricky situation

Q. What to do when next codeword is not yet in ST when needed?

compression

input	A	B	A	B	A	B	A	
matches	A	B	A B		A B A			
output	41	42	81		83			80

A B 81 A B
B A 82 A B
B R
A B A 83

codeword table

key	*value*
A B	81
B R	82
A B A	83

expansion

input	41	42	81	83	80
output	A	B	A B	?	

must be A B A *(see below)*

81 A B A B A B
82 B A B A
83 A B ?

need lookahead character to complete entry

next character in output—the lookahead character!

LZW implementation details

How big to make ST?

- How long is message?
- Whole message similar model?
- [many variations have been developed]

What to do when ST fills up?

- Throw away and start over. [GIF]
- Throw away when not effective. [Unix compress]
- [many other variations]

Why not put longer substrings in ST?

- [many variations have been developed]

LZW in the real world

Lempel-Ziv and friends.

- LZ77.
- LZ78.
- LZW.
- Deflate = LZ77 variant + Huffman.

LZ77 not patented ⇒ widely used in open source

LZW patent #4,558,302 expired in US on June 20, 2003

some versions copyrighted

PNG: LZ77.

Winzip, gzip, jar: deflate.

Unix compress: LZW.

Pkzip: LZW + Shannon-Fano.

GIF, TIFF, V.42bis modem: LZW.

Google: zlib which is based on deflate.

never expands a file

Lossless data compression benchmarks

year	scheme	bits / char
1967	ASCII	7.00
1950	Huffman	4.70
1977	LZ77	3.94
1984	LZMW	3.32
1987	LZH	3.30
1987	move-to-front	3.24
1987	LZB	3.18
1987	gzip	2.71
1988	PPMC	2.48
1994	SAKDC	2.47
1994	PPM	2.34
1995	Burrows-Wheeler	2.29
1997	BOA	1.99
1999	RK	1.89

next programming assignment

data compression using Calgary corpus

Data compression summary

Lossless compression.

- Represent fixed-length symbols with variable-length codes. [Huffman]
- Represent variable-length symbols with fixed-length codes. [LZW]

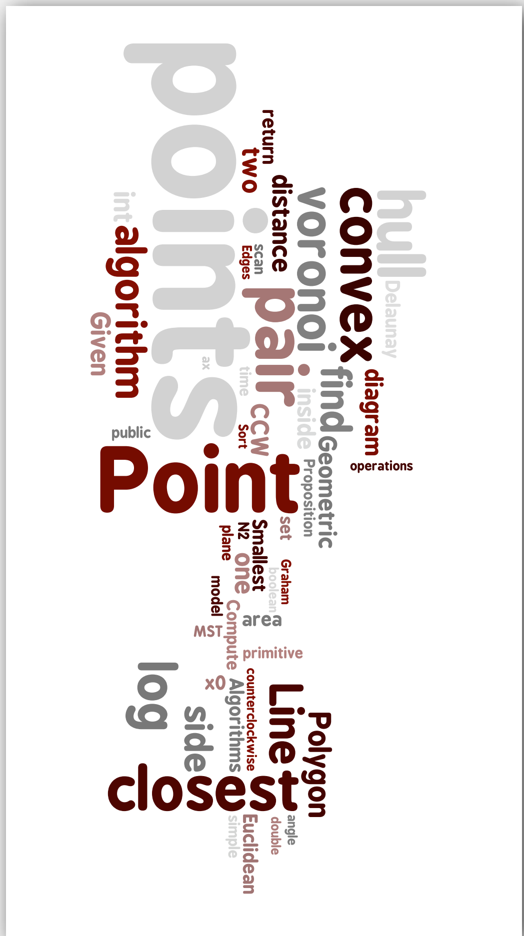
Lossy compression. [not covered in this course]

- JPEG, MPEG, MP3, …
- FFT, wavelets, fractals, …

Theoretical limits on compression. Shannon entropy.

Practical compression. Use extra knowledge whenever possible.

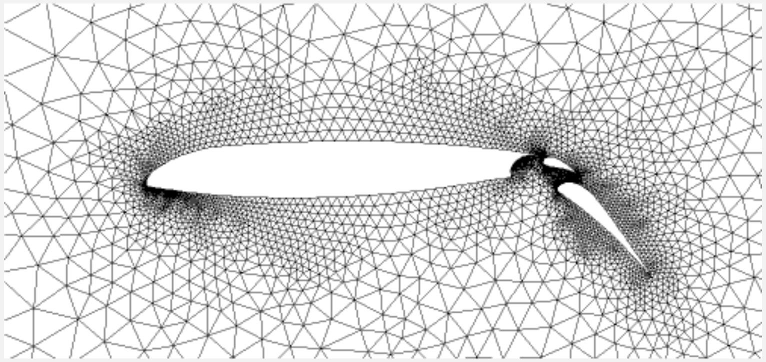
6.1 Geometric Primitives



- primitive operations
- convex hull
- closest pair
- voronoi diagram

Geometric algorithms

Applications.



airflow around an aircraft wing

- Data mining.
- VLSI design.
- Computer vision.
- Mathematical models.
- Astronomical simulation.
- Geographic information systems.
- Computer graphics (movies, games, virtual reality).
- Models of physical world (maps, architecture, medical imaging).

http://www.ics.uci.edu/~eppstein/geom.html

History.

- Ancient mathematical foundations.
- Most geometric algorithms less than 25 years old.

- primitive operations
- convex hull
- closest pair
- voronoi diagram

Geometric primitives

Point: two numbers (x, y).

Line: two numbers a and b. [ax + by = 1] any line not through origin

Line segment: two points.

Polygon: sequence of points.

Primitive operations.

- Is a polygon simple?
- Is a point inside a polygon?
- Do two line segments intersect?
- What is Euclidean distance between two points?
- Given three points p_1, p_2, p_3, is $p_1 \to p_2 \to p_3$ a counterclockwise turn?

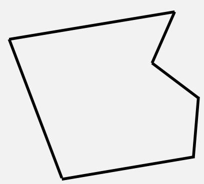
Other geometric shapes.

- Triangle, rectangle, circle, sphere, cone, …
- 3D and higher dimensions sometimes more complicated.

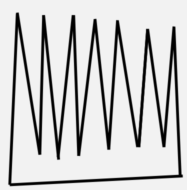
Warning: intuition may be misleading.

- Humans have spatial intuition in 2D and 3D.
- Computers do not.
- Neither has good intuition in higher dimensions!

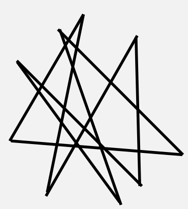
Q. Is a given polygon simple? ← no crossings



x	1	6	5	8	7	2
y	7	8	6	4	2	1



x	1	15	14	13	12	11	10	9	8	7	6	5	4	3	2
y	1	2	18	4	18	4	19	4	19	4	20	3	20	2	20



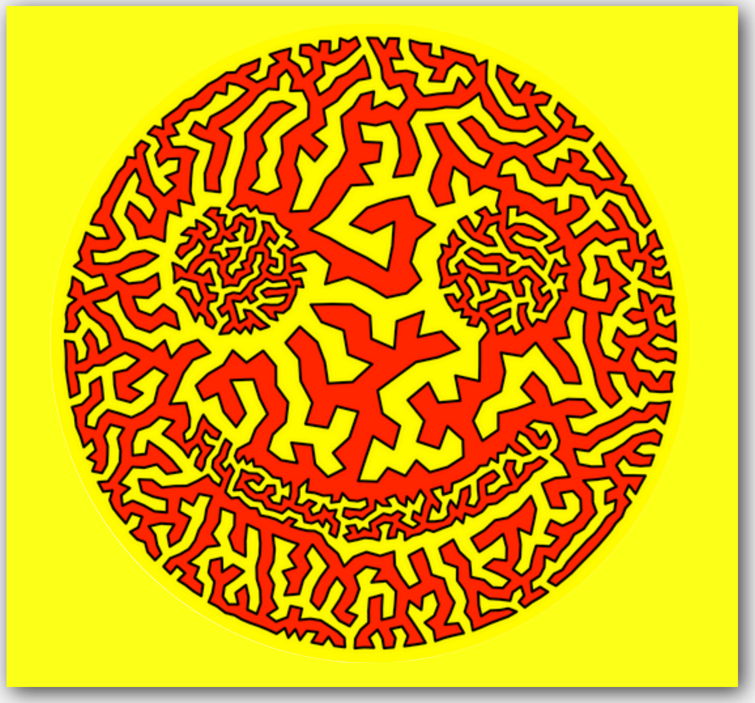
x	1	10	3	7	2	8	8	3	4
y	6	5	15	1	11	3	14	2	16

we think of this

algorithm sees this

Jordan curve theorem. [Jordan 1887, Veblen 1905] Any continuous simple closed curve cuts the plane in exactly two pieces: the inside and the outside.

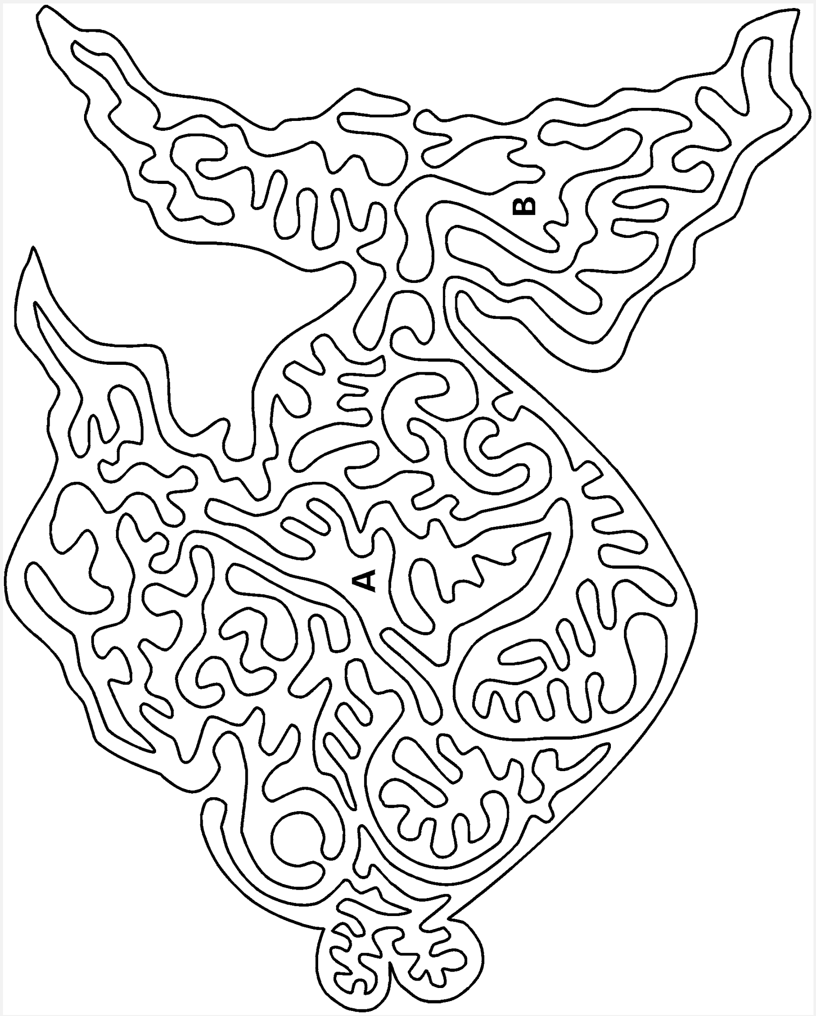
Q. Is a point inside a simple polygon?



Application. Draw a filled polygon on the screen.

Fishy maze

Puzzle. Are A and B inside or outside the maze?

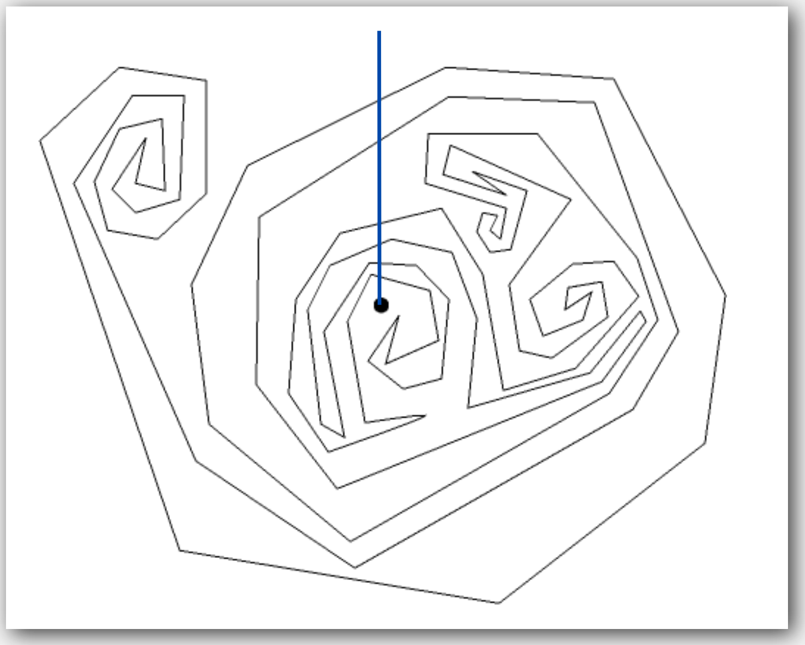


http://britton.disted.camosun.bc.ca/fishmaze.pdf

Polygon inside, outside

Jordan curve theorem. [Jordan 1887, Veblen 1905] Any continuous simple closed curve cuts the plane in exactly two pieces: the inside and the outside.

Q. Is a point inside a simple polygon?



http://www.ics.uci.edu/~eppstein/geom.html

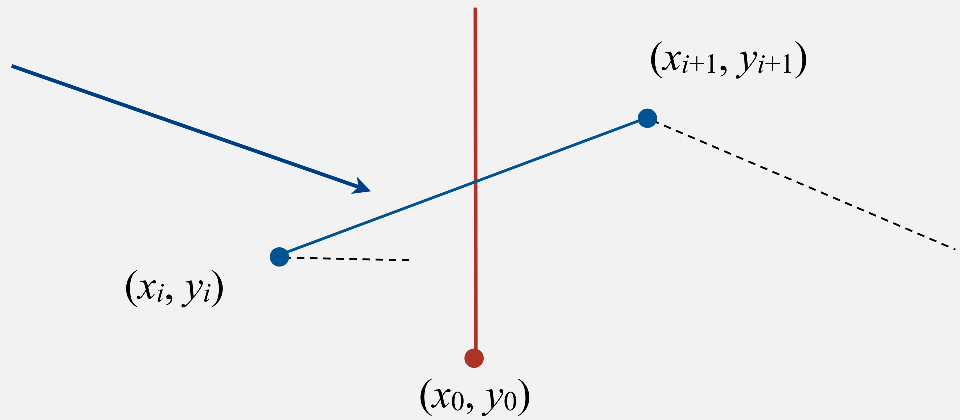
Application. Draw a filled polygon on the screen.

Polygon inside, outside: crossing number

Q. Does line segment intersect ray?

$$y_0 = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x_0 - x_i) + y_i$$

$$x_i \le x_0 \le x_{i+1}$$

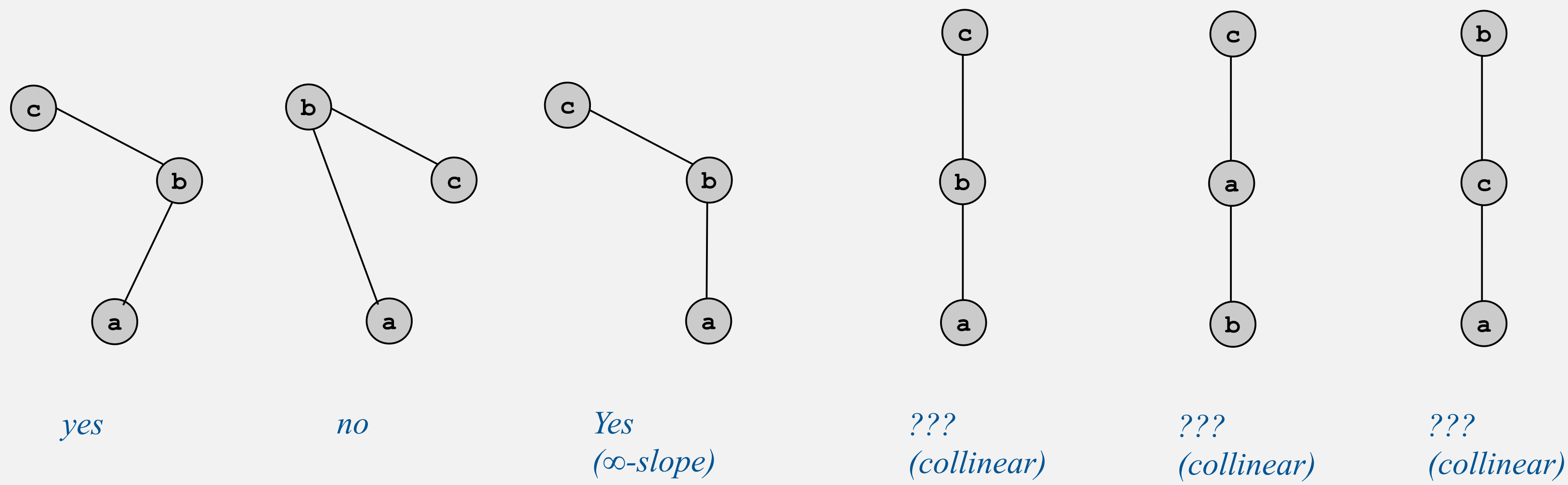



```
public boolean contains(double x0, double y0)
{
    int crossings = 0;
    for (int i = 0; i < N; i++)
    {
       double  slope = (y[i+1] - y[i]) / (x[i+1] - x[i]);
       boolean cond1 = (x[i] <= x0) && (x0 < x[i+1]);
       boolean cond2 = (x[i+1] <= x0) && (x0 < x[i]);
       boolean above = (y0 < slope * (x0 - x[i]) + y[i]);
       if ((cond1 || cond2)  && above) crossings++;
    }
    return crossings % 2 != 0;
 }
```

Implementing ccw

CCW. Given three point a, b, and c, is a-b-c a counterclockwise turn?

- Analog of compares in sorting.
- Idea: compare slopes.



Lesson. Geometric primitives are tricky to implement.

- Dealing with degenerate cases.
- Coping with floating-point precision.

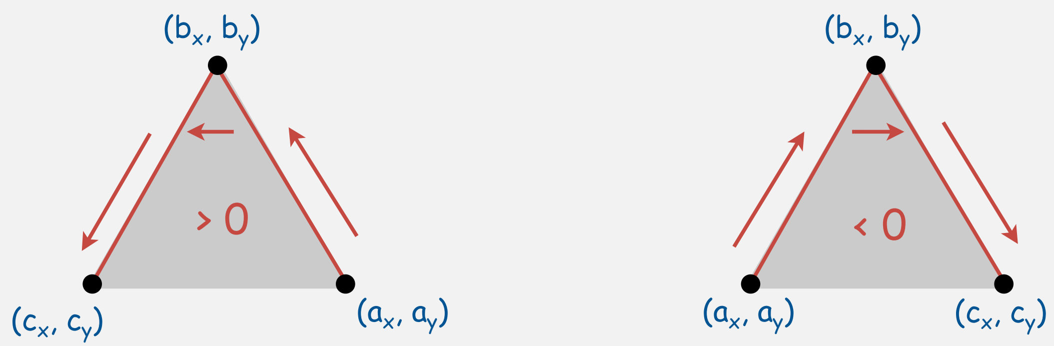
Implementing ccw

CCW. Given three point a, b, and c, is a→b→c a counterclockwise turn?

- Determinant gives twice signed area of triangle.

$$2 \times Area(a,b,c) = \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} = (b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x)$$

- If area > 0 then a→b→c is counterclockwise.
- If area < 0, then a→b→c is clockwise.
- If area = 0, then a→b→c are collinear.



Immutable point data type

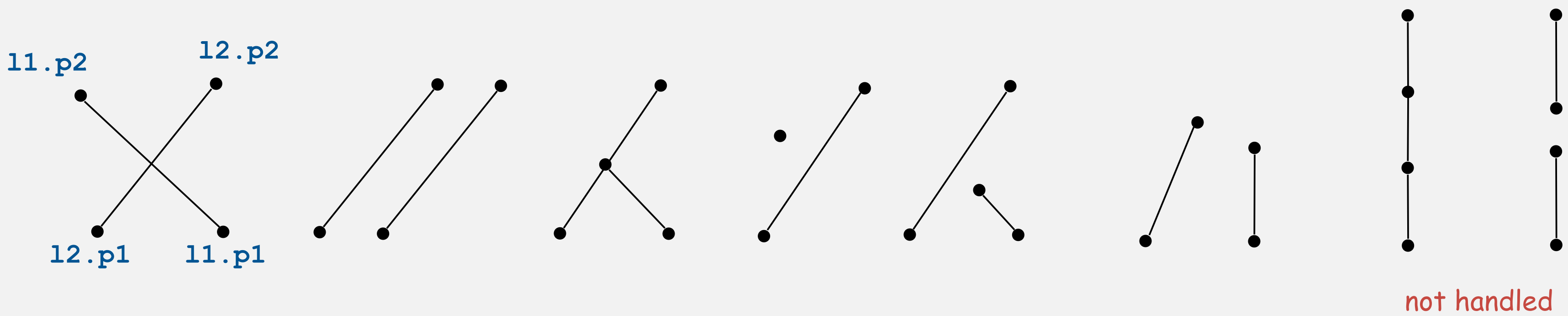
```
public class Point
{
   private final int x;
   private final int y;

   public Point(int x, int y)
   {  this.x = x; this.y = y;  }

   public double distanceTo(Point that)
   {
      double dx = this.x - that.x;
      double dy = this.y - that.y;
      return Math.sqrt(dx*dx + dy*dy);
   }

   public static int ccw(Point a, Point b, Point c)
   {
      int area2 = (b.x-a.x)*(c.y-a.y) - (b.y-a.y)*(c.x-a.x);
      if      (area2 < 0) return -1;
      else if (area2 > 0) return +1;
      else                return  0;
   }

   public static boolean collinear(Point a, Point b, Point c)
   {  return ccw(a, b, c) == 0;  }
}
```

cast to `long` to avoid overflowing an `int`

Sample ccw client: line intersection

Intersect. Given two line segments, do they intersect?

- Idea 1: find intersection point using algebra and check.
- Idea 2: check if the endpoints of one line segment are on different "sides" of the other line segment (4 calls to ccw).

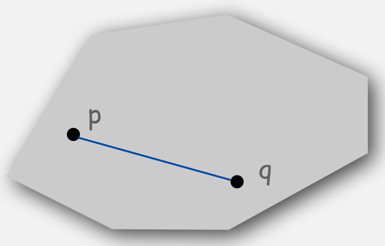



```
public static boolean intersect(LineSegment l1, LineSegment l2)
{
   int test1 = Point.ccw(l1.p1, l1.p2, l2.p1) * Point.ccw(l1.p1, l1.p2, l2.p2);
   int test2 = Point.ccw(l2.p1, l2.p2, l1.p1) * Point.ccw(l2.p1, l2.p2, l1.p2);
   return (test1 <= 0) && (test2 <= 0);
}
```

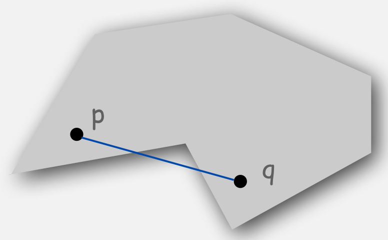
- primitive operations
- **convex hull**
- closest pair
- voronoi diagram

A set of points is convex if for any two points p and q in the set, the line segment $\overline{pq}$ is completely in the set.

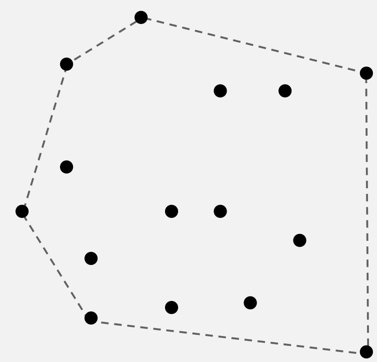
Convex hull. Smallest convex set containing all the points.



convex



not convex



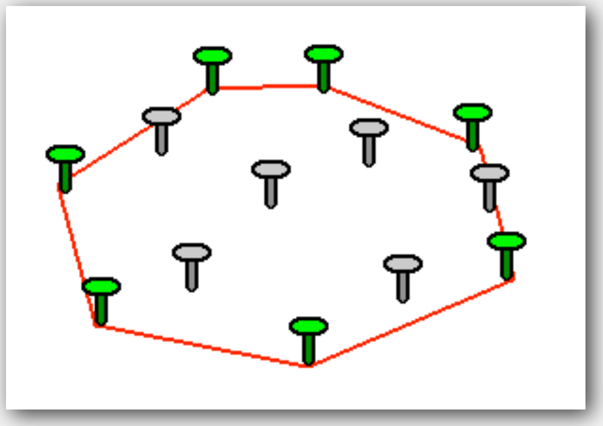
convex hull

Properties.

- "Simplest" shape that approximates set of points.
- Shortest perimeter fence surrounding the points.
- Smallest area convex polygon enclosing the points.

Mechanical solution

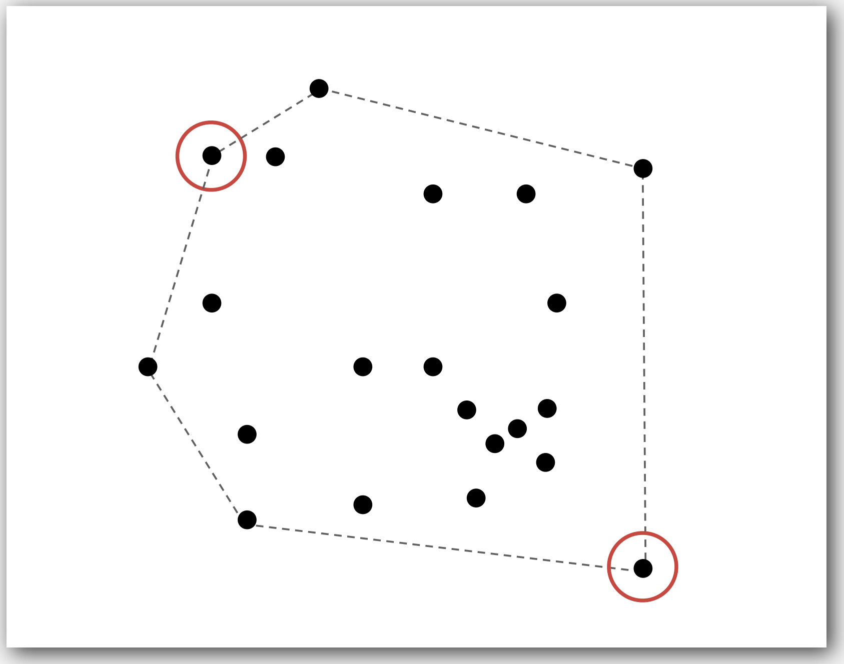
Mechanical convex hull algorithm. Hammer nails perpendicular to plane; stretch elastic rubber band around points.



http://www.dfanning.com/math_tips/convexhull_1.gif

An application: farthest pair

Farthest pair problem. Given N points in the plane, find a pair of points with the largest Euclidean distance between them.



Fact. Farthest pair of points are on convex hull.

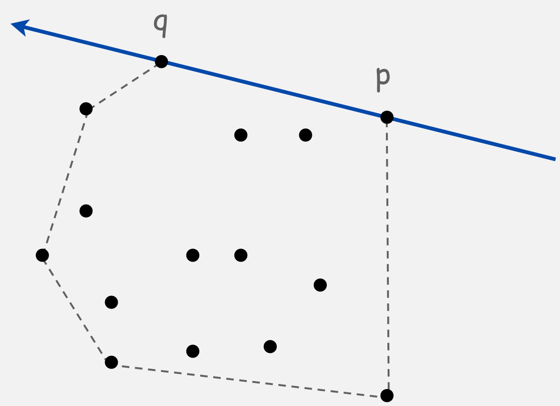
Brute-force algorithm

Observation 1.

Edges of convex hull of P connect pairs of points in P.

Observation 2.

p-q is on convex hull if all other points are counterclockwise of $\overrightarrow{pq}$.



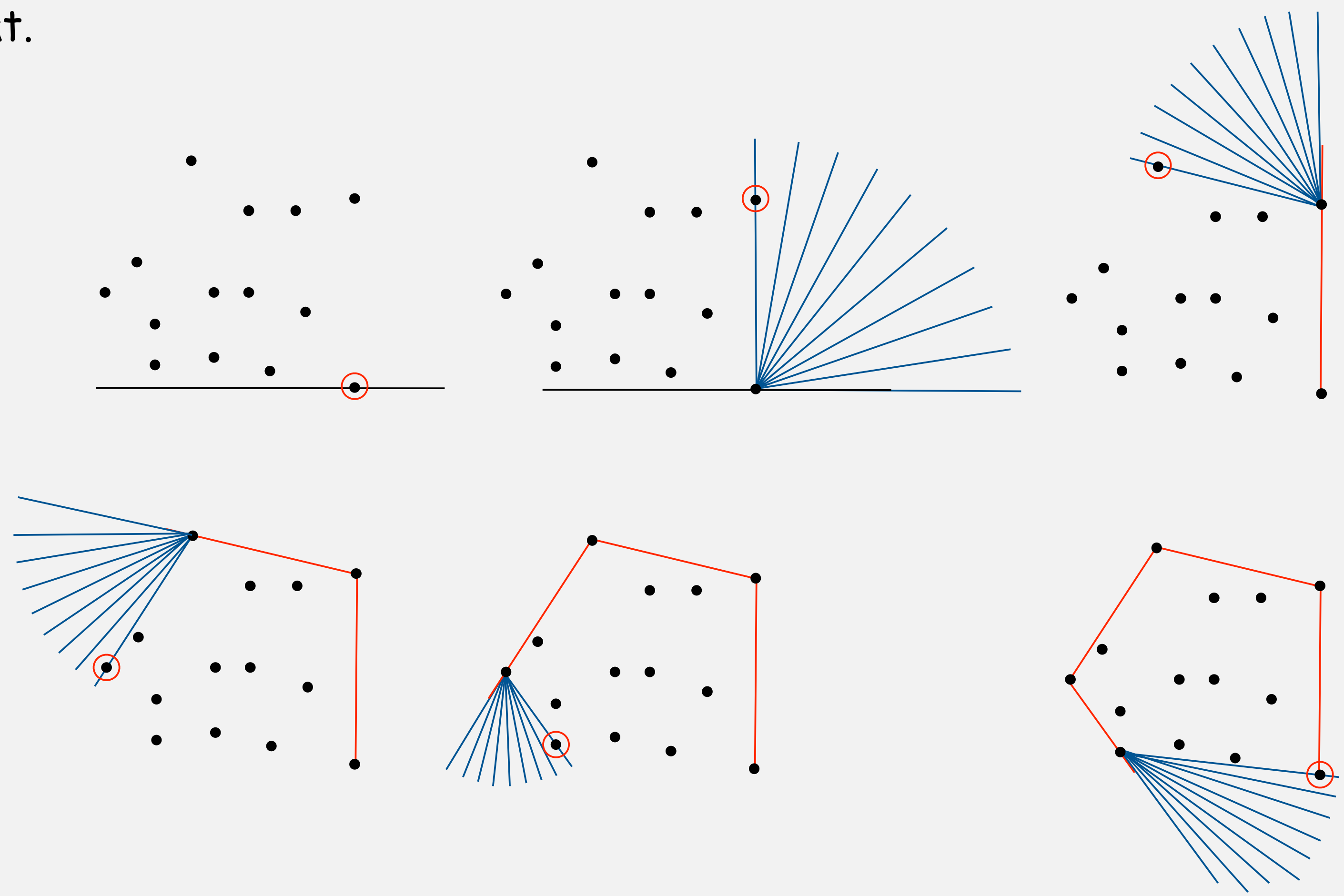
O(N^3) algorithm. For all pairs of points p and q:

- Compute `ccw(p, q, x)` for all other points `x`.
- p-q is on hull if all values are positive.

Package wrap (Jarvis march)

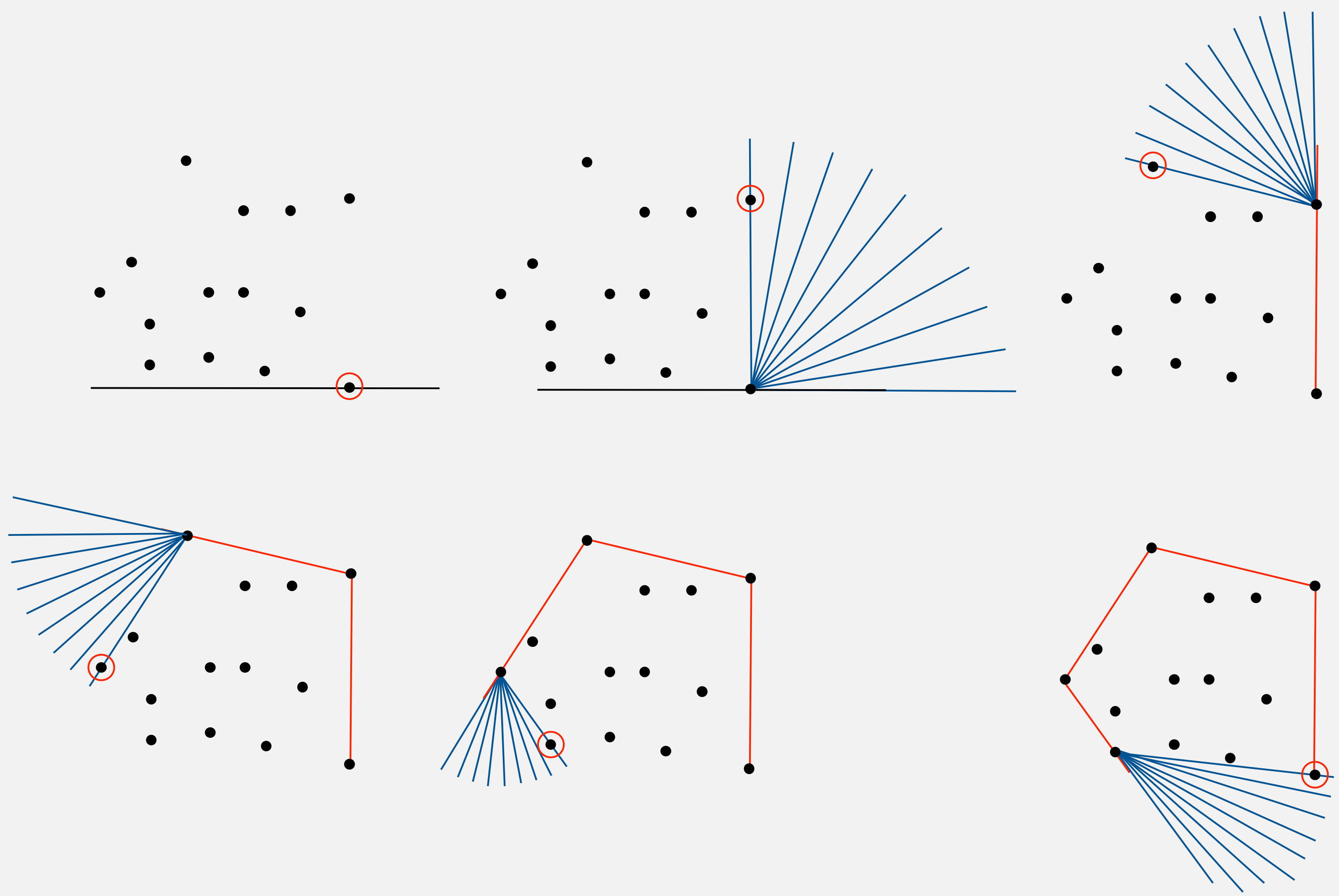
Package wrap.

- Start with point with smallest (or largest) y-coordinate.
- Rotate sweep line around current point in ccw direction.
- First point hit is on the hull.
- Repeat.

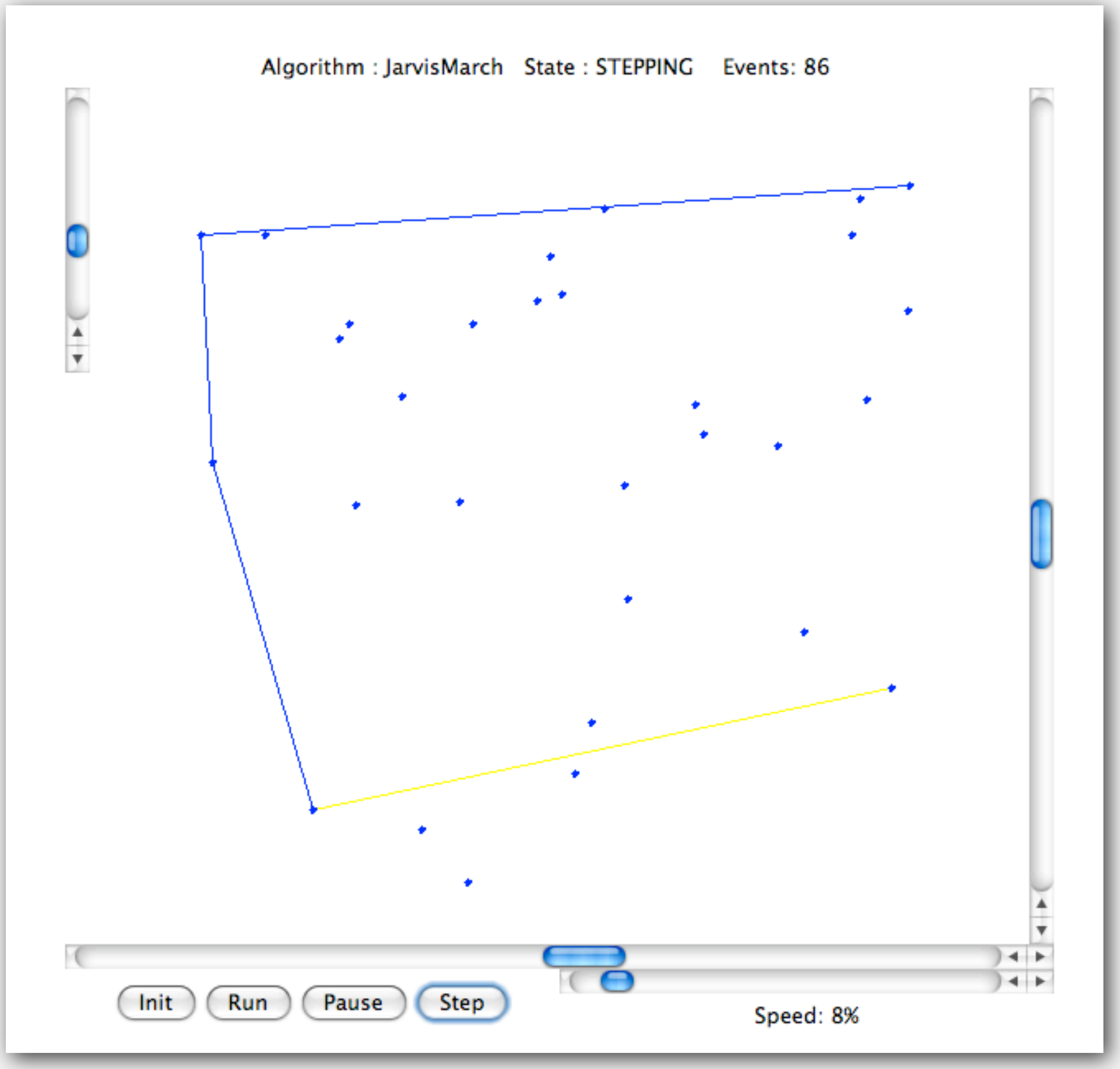


Implementation.

- Compute angle between current point and all remaining points.
- Pick smallest angle larger than current angle.
- $\Theta(N)$ per iteration.

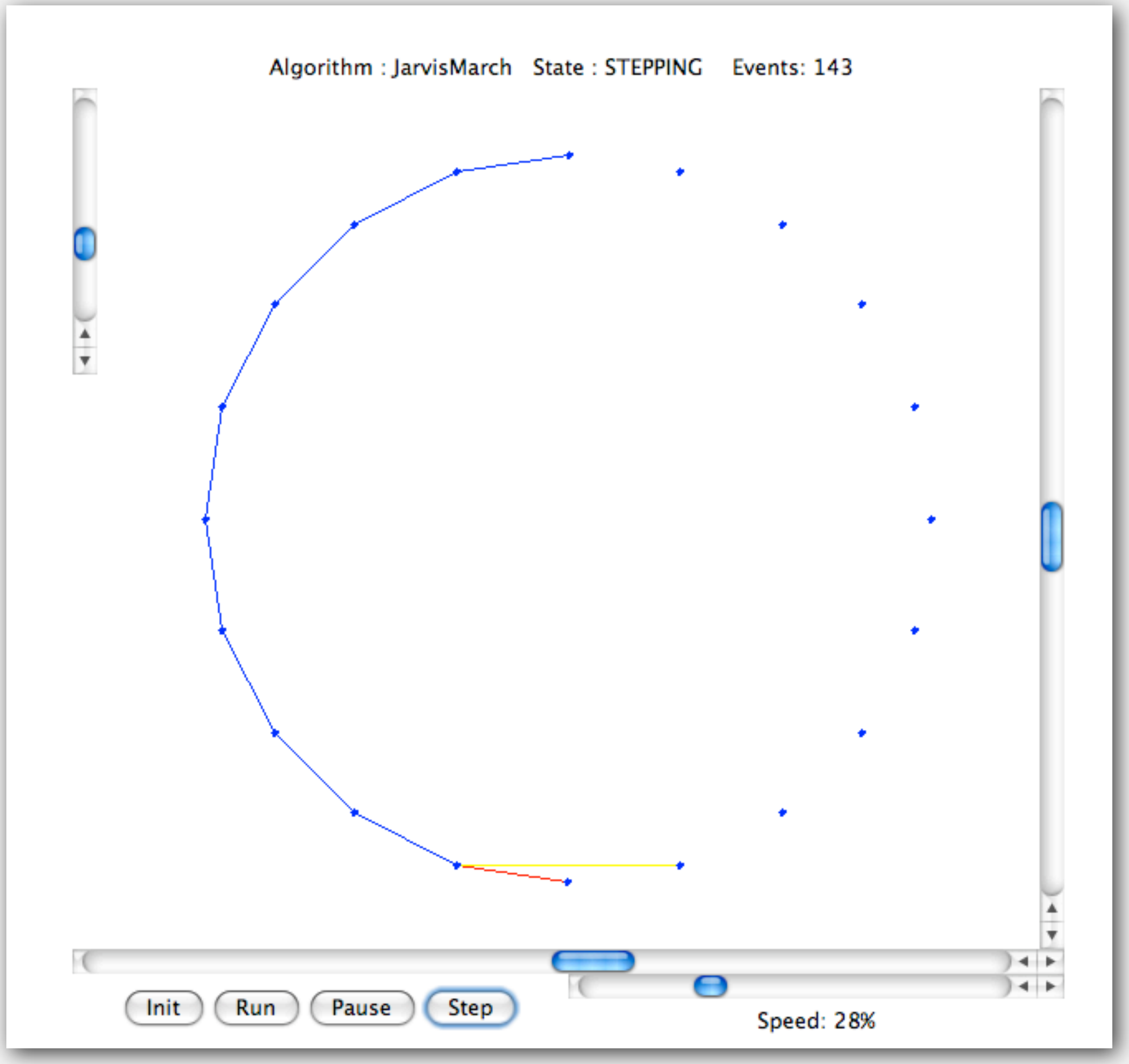


Jarvis march: demo



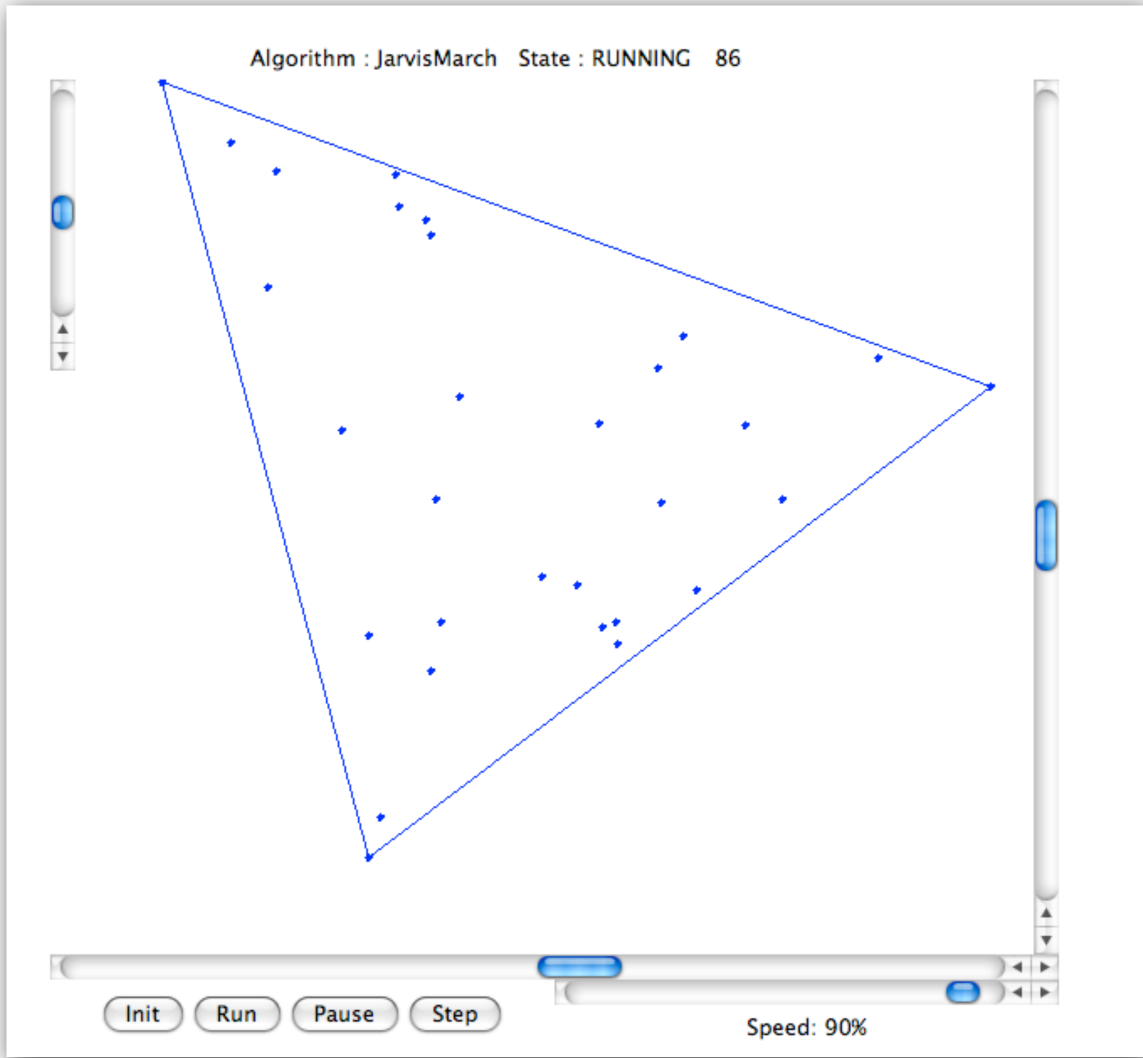
http://www.cs.princeton.edu/courses/archive/fall08/cos226/demo/ah/JarvisMarch.html

Jarvis march: demo



http://www.cs.princeton.edu/courses/archive/fall08/cos226/demo/ah/JarvisMarch.html

Jarvis march: demo



http://www.cs.princeton.edu/courses/archive/fall08/cos226/demo/ah/JarvisMarch.html

How many points on the hull?

Parameters.

- N = number of points.
- h = number of points on the hull.

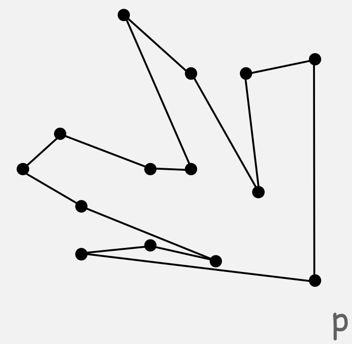
Package wrap running time. $\Theta(N\,h)$.

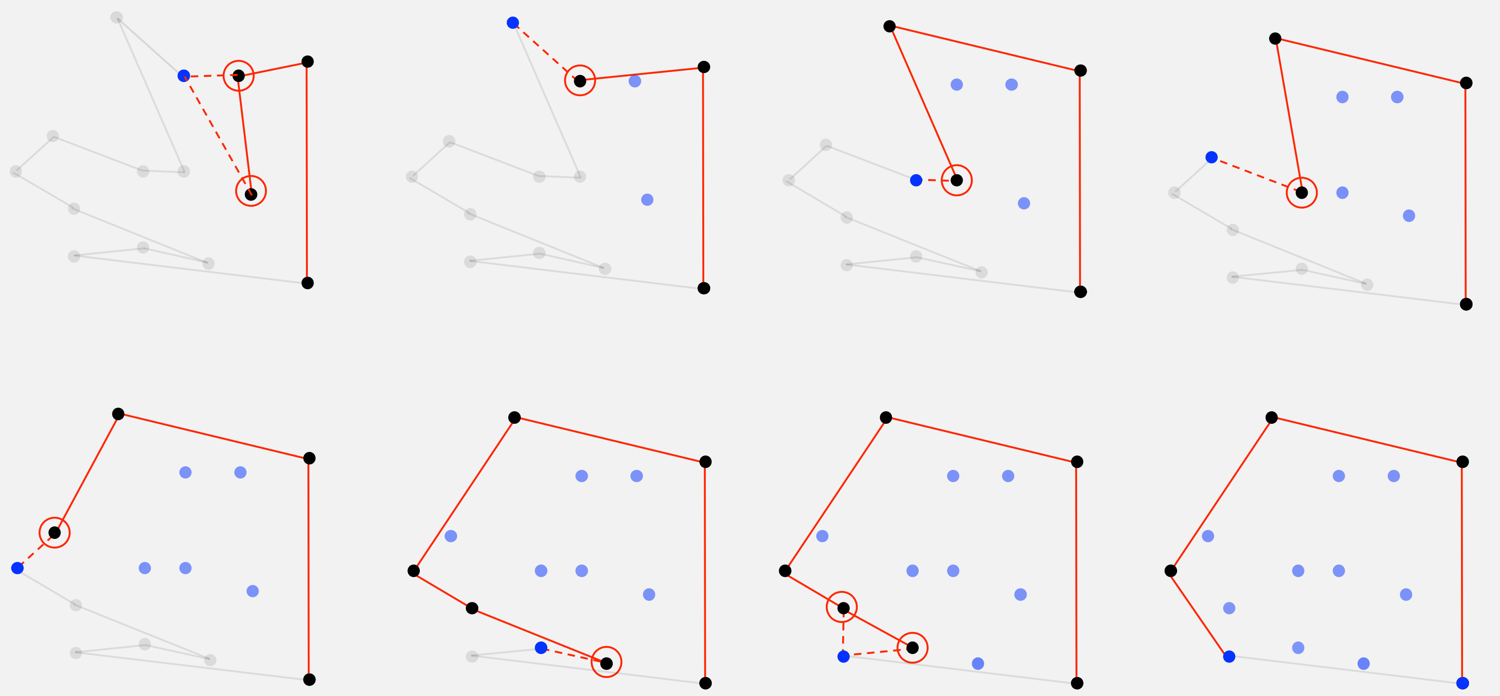
How many points on hull?

- Worst case: $h = N$.
- Average case: difficult problems in stochastic geometry.
  - uniformly at random in a disc: $h = N^{1/3}$
  - uniformly at random in a convex polygon with $O(1)$ edges: $h = \log N$

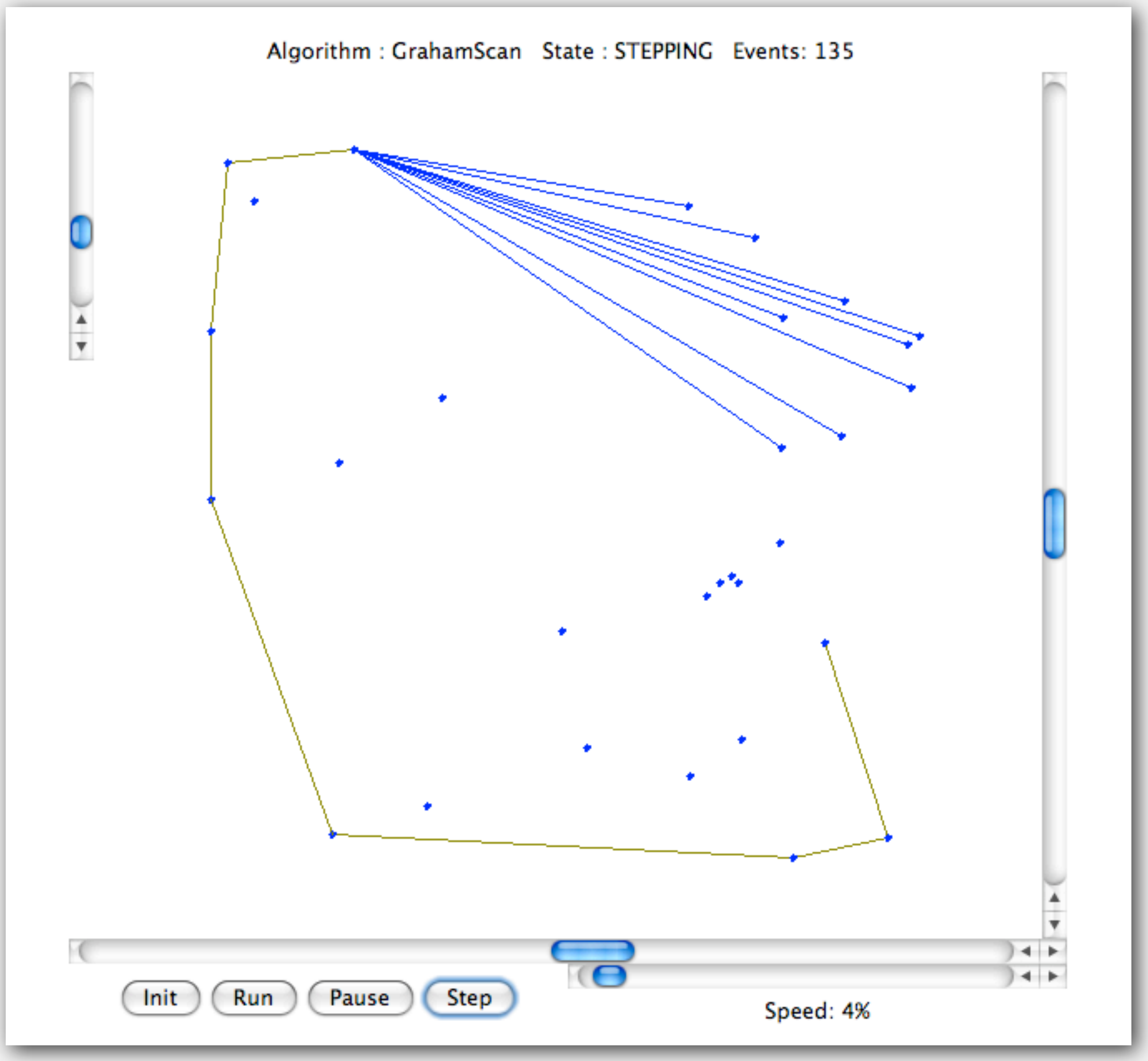
Graham scan.

- Choose point p with smallest (or largest) y-coordinate.
- Sort points by polar angle with p to get simple polygon.
- Consider points in order, and discard those that would create a clockwise turn.



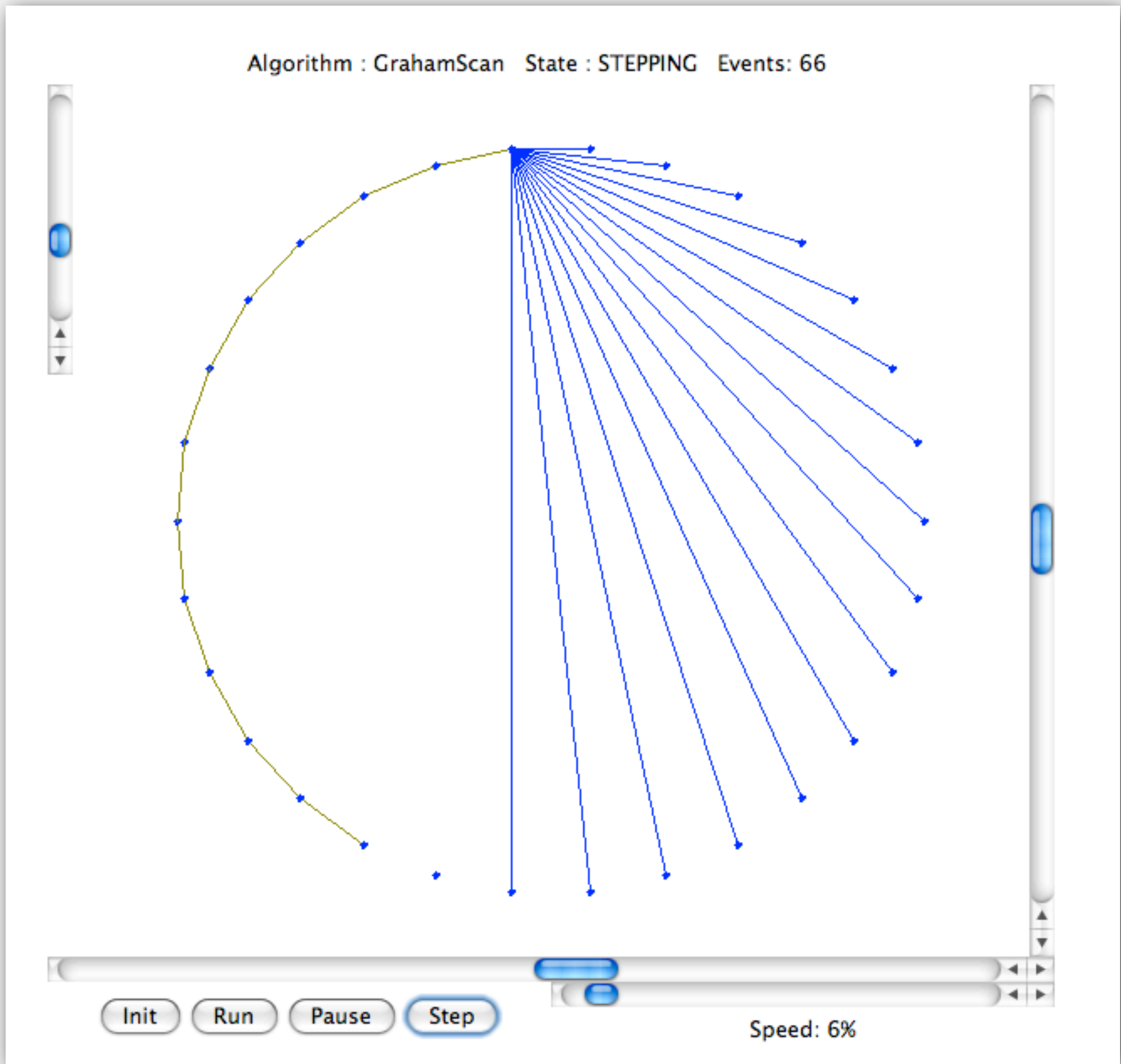


Graham scan: demo



http://www.cs.princeton.edu/courses/archive/fall08/cos226/demo/ah/GrahamScan.html

Graham scan: demo



http://www.cs.princeton.edu/courses/archive/fall08/cos226/demo/ah/GrahamScan.html

Graham scan: implementation

Implementation.

- Input: `p[1], p[2], …, p[N]` are distinct points.
- Output: `M` and rearrangement so that `p[1], p[2], …, p[M]` is convex hull.

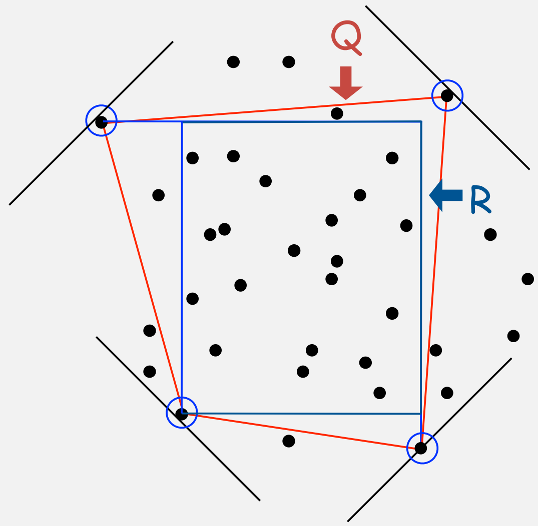
```
// preprocess so that p[1] has smallest y-coordinate
// sort by polar angle with respect to p[1]

p[0] = p[N];   // sentinel
int M = 2;
for (int i = 3; i <= N; i++)
{
   while (Point.ccw(p[M-1], p[M], p[i]) <= 0)
      M--;
   M++;
   swap(p, M, i);   ← add i to putative hull
}
```

discard points that would create clockwise turn

Running time. O(N log N) for sort and $\underbrace{\text{O(N)}}_{}$ for rest. why?

Quick elimination.

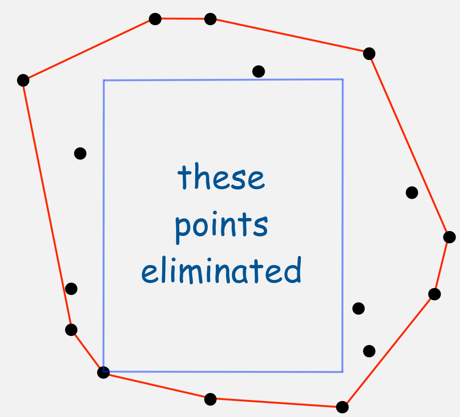
- Choose a quadrilateral Q or rectangle R with 4 points as corners.
- Any point inside cannot be on hull.
  - 4 ccw tests for quadrilateral
  - 4 compares for rectangle



Three-phase algorithm.

- Pass through all points to compute R.
- Eliminate points inside R.
- Find convex hull of remaining points.

In practice. Eliminates almost all points in linear time.



Convex hull algorithms costs summary

Asymptotic cost to find h-point hull in N-point set.

algorithm	running time
package wrap	N h
Graham scan	N log N
quickhull	N log N
mergehull	N log N
sweep line	N log N
quick elimination	N †
marriage-before-conquest	N log h

← output sensitive (package wrap)

← output sensitive (marriage-before-conquest)

† assumes "reasonable" point distribution

Convex hull: lower bound

Models of computation.

- Compare-based: compare coordinates.
  (impossible to compute convex hull in this model of computation)

```
(a.x < b.x) || ((a.x == b.x) && (a.y < b.y)))
```

- Quadratic decision tree model: compute any quadratic function of the coordinates and compare against 0.

```
(a.x*b.y - a.y*b.x + a.y*c.x - a.x*c.y + b.x*c.y - c.x*b.y) < 0
```

higher constant-degree polynomial tests don't help either [Ben-Or, 1983]

Proposition. [Andy Yao, 1981] In quadratic decision tree model, any convex hull algorithm requires $\Omega(N \log N)$ ops.

even if hull points are not required to be output in counterclockwise order

- primitive operations
- convex hull
- **closest pair**
- voronoi diagram

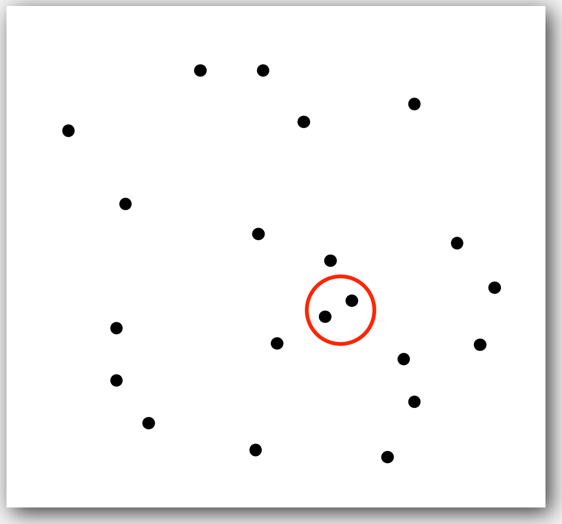
Closest pair

Closest pair problem. Given N points in the plane, find a pair of points with the smallest Euclidean distance between them.

Fundamental geometric primitive.

- Graphics, computer vision, geographic information systems, molecular modeling, air traffic control.
- Special case of nearest neighbor, Euclidean MST, Voronoi.

fast closest pair inspired fast algorithms for these problems



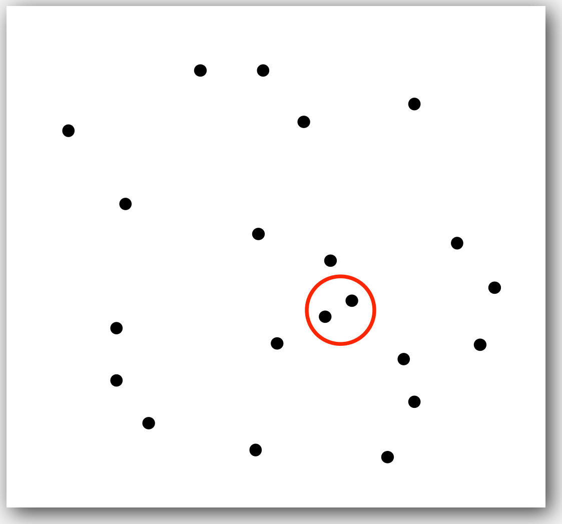
Closest pair

Closest pair problem. Given N points in the plane, find a pair of points with the smallest Euclidean distance between them.

Brute force. Check all pairs with N^2 distance calculations.

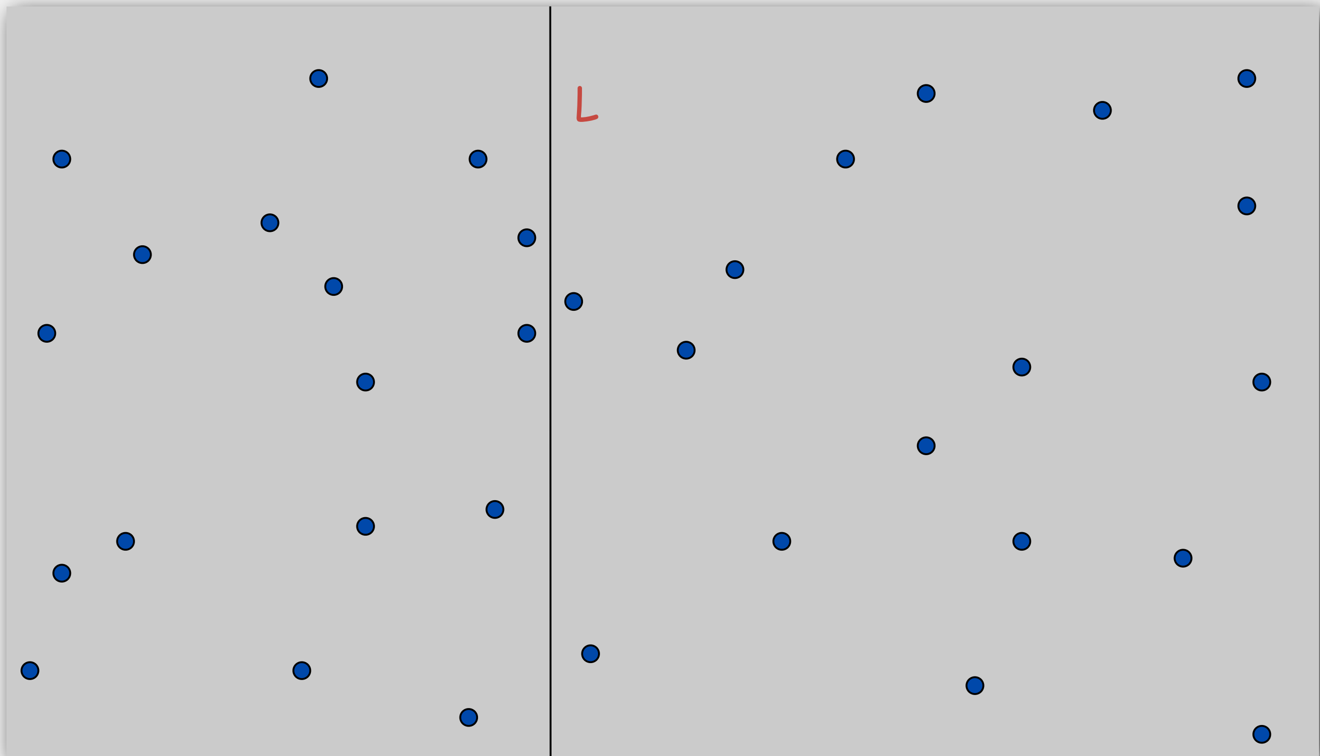
1-D version. Easy N log N algorithm if points are on a line.

Degeneracies complicate solutions.
[assumption for lecture: no two points have same x-coordinate]



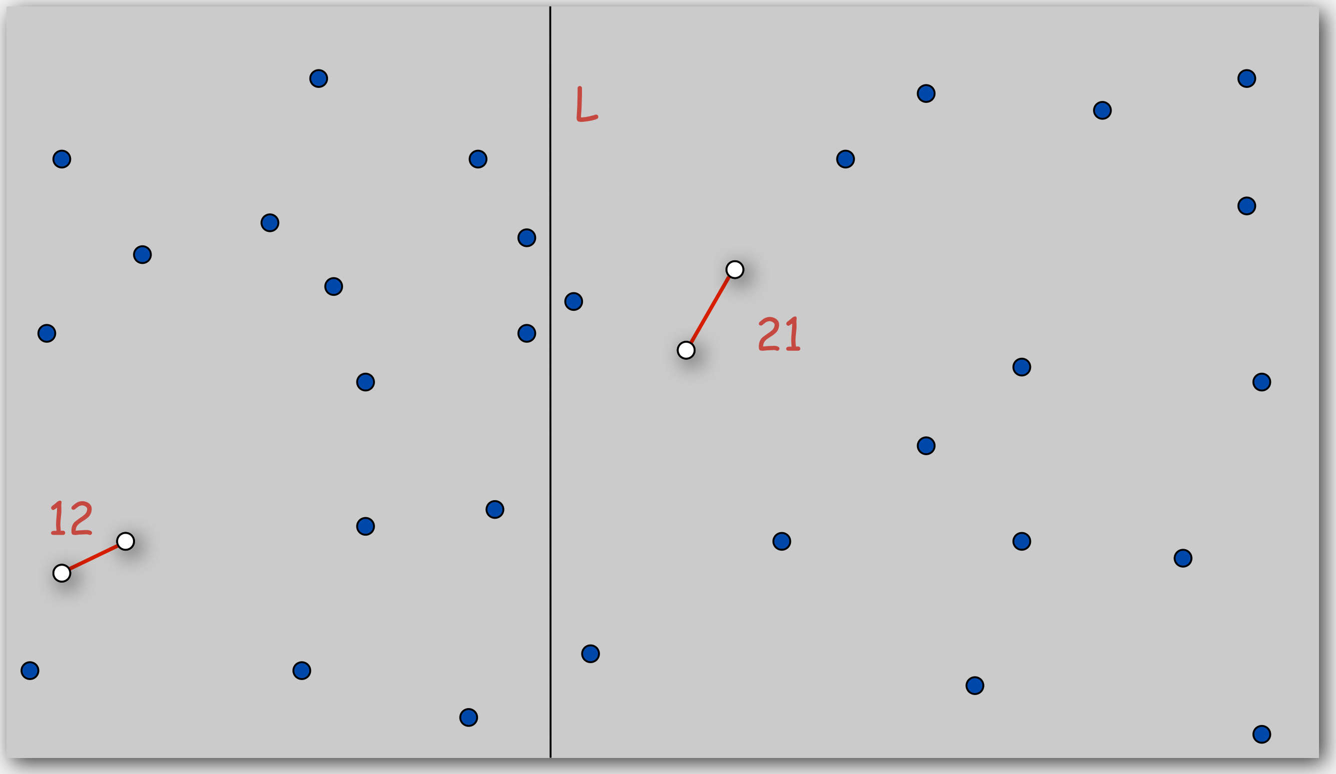
Divide-and-conquer algorithm

- Divide: draw vertical line L so that ~ $\frac{1}{2}$N points on each side.



Divide-and-conquer algorithm

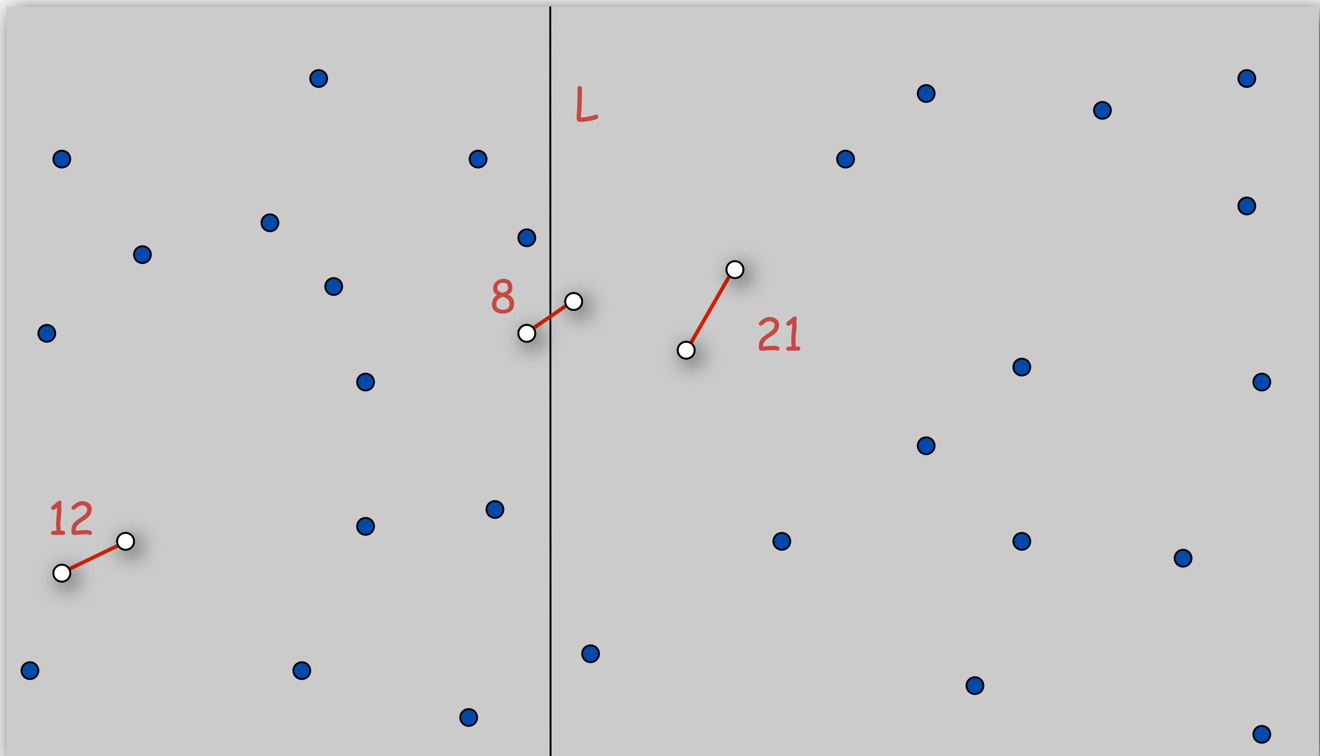
- Divide: draw vertical line L so that ~ $\frac{1}{2}$N points on each side.
- Conquer: find closest pair in each side recursively.



Divide-and-conquer algorithm

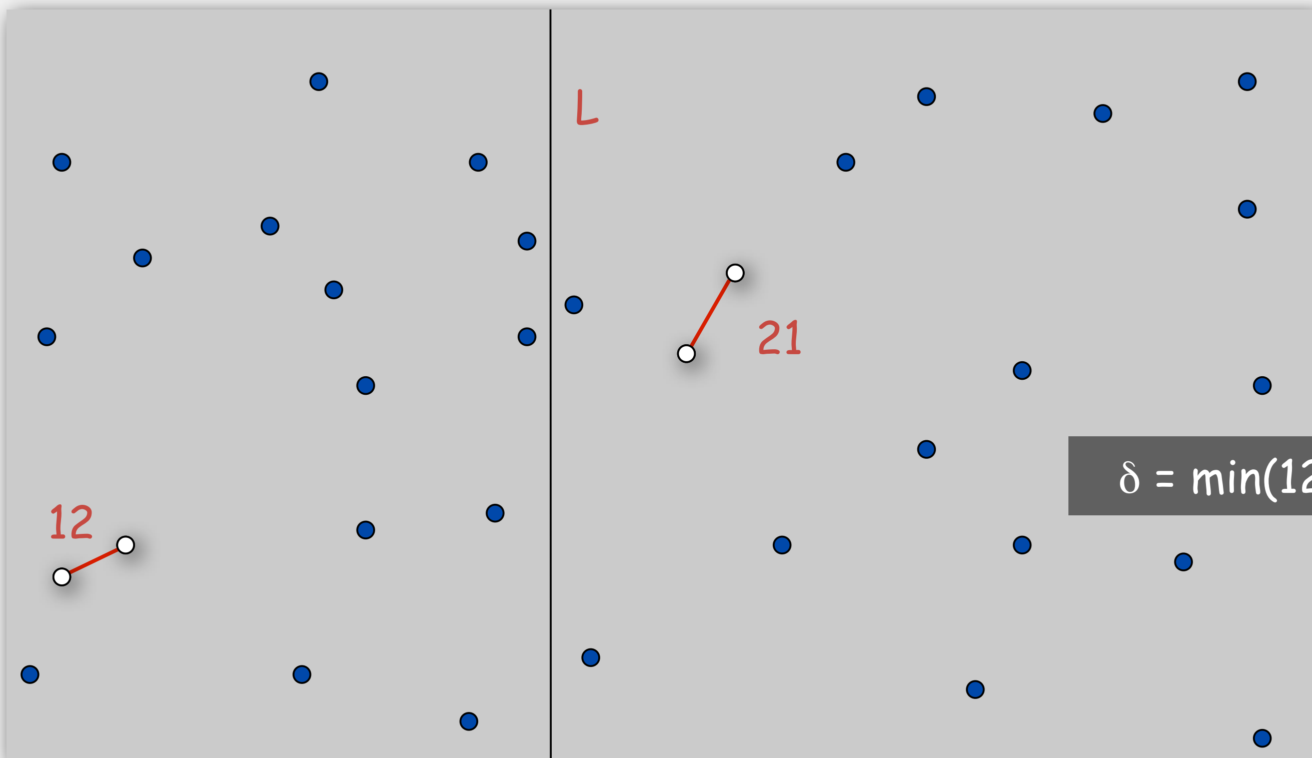
- Divide: draw vertical line L so that ~ $\frac{1}{2}N$ points on each side.
- Conquer: find closest pair in each side recursively.
- Combine: find closest pair with one point in each side.
- Return best of 3 solutions.

seems like $\Theta(N^2)$



How to find closest pair with one point in each side?

Find closest pair with one point in each side, assuming that distance < δ.

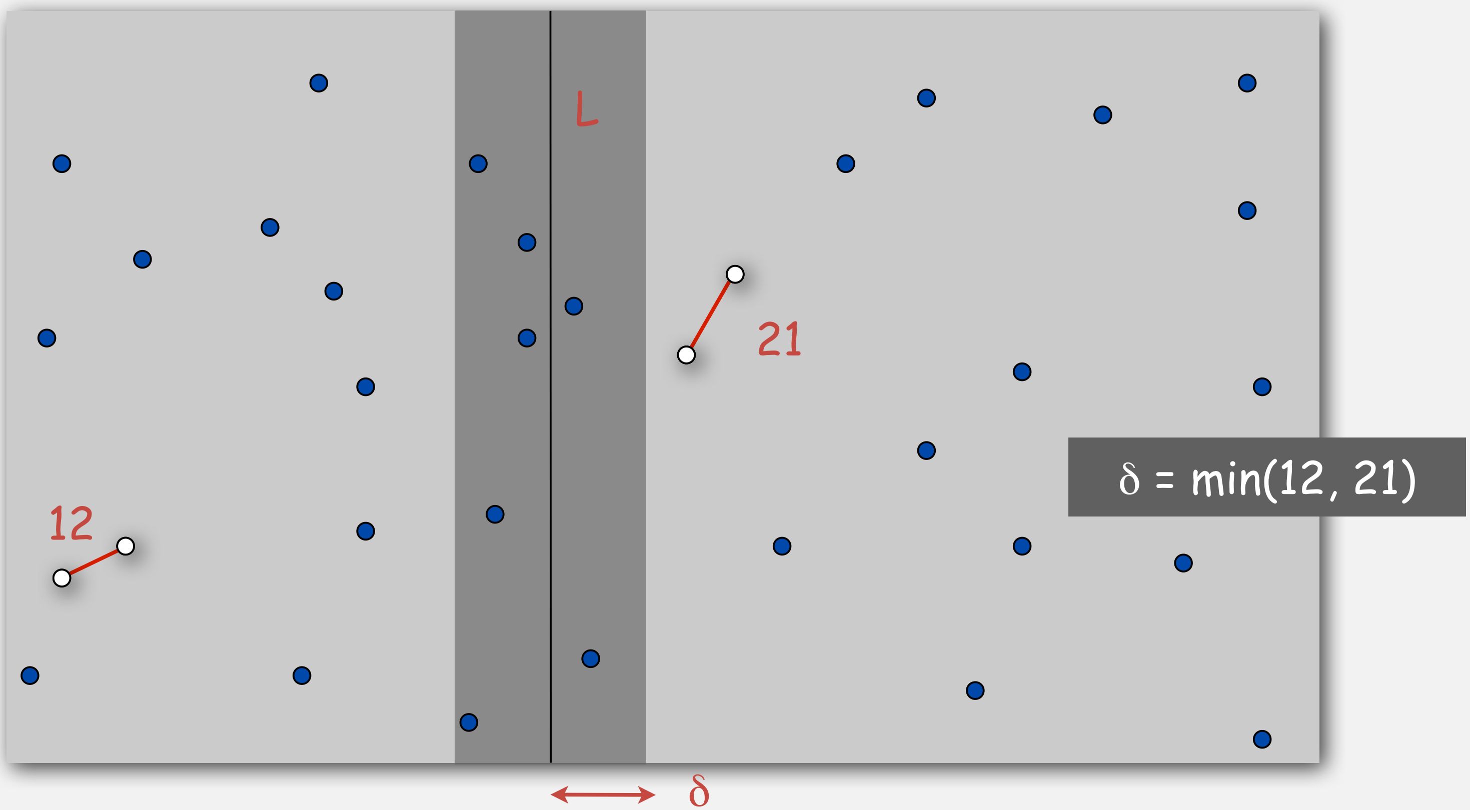


δ = min(12, 21)

How to find closest pair with one point in each side?

Find closest pair with one point in each side, assuming that distance < δ.

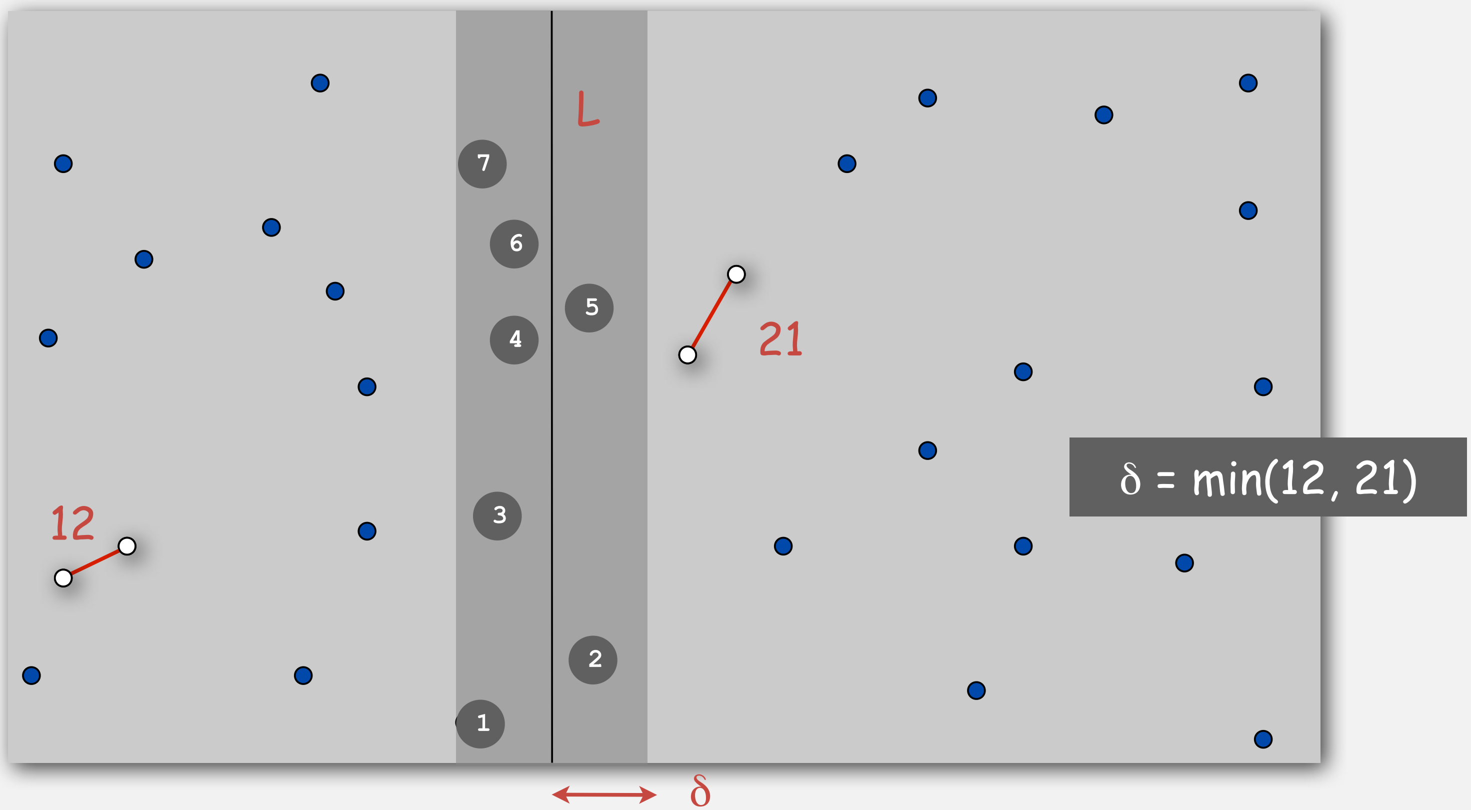
- Observation: only need to consider points within δ of line L.



How to find closest pair with one point in each side?

Find closest pair with one point in each side, assuming that distance < δ.

- Observation: only need to consider points within δ of line L.
- Sort points in 2δ-strip by their y coordinate.

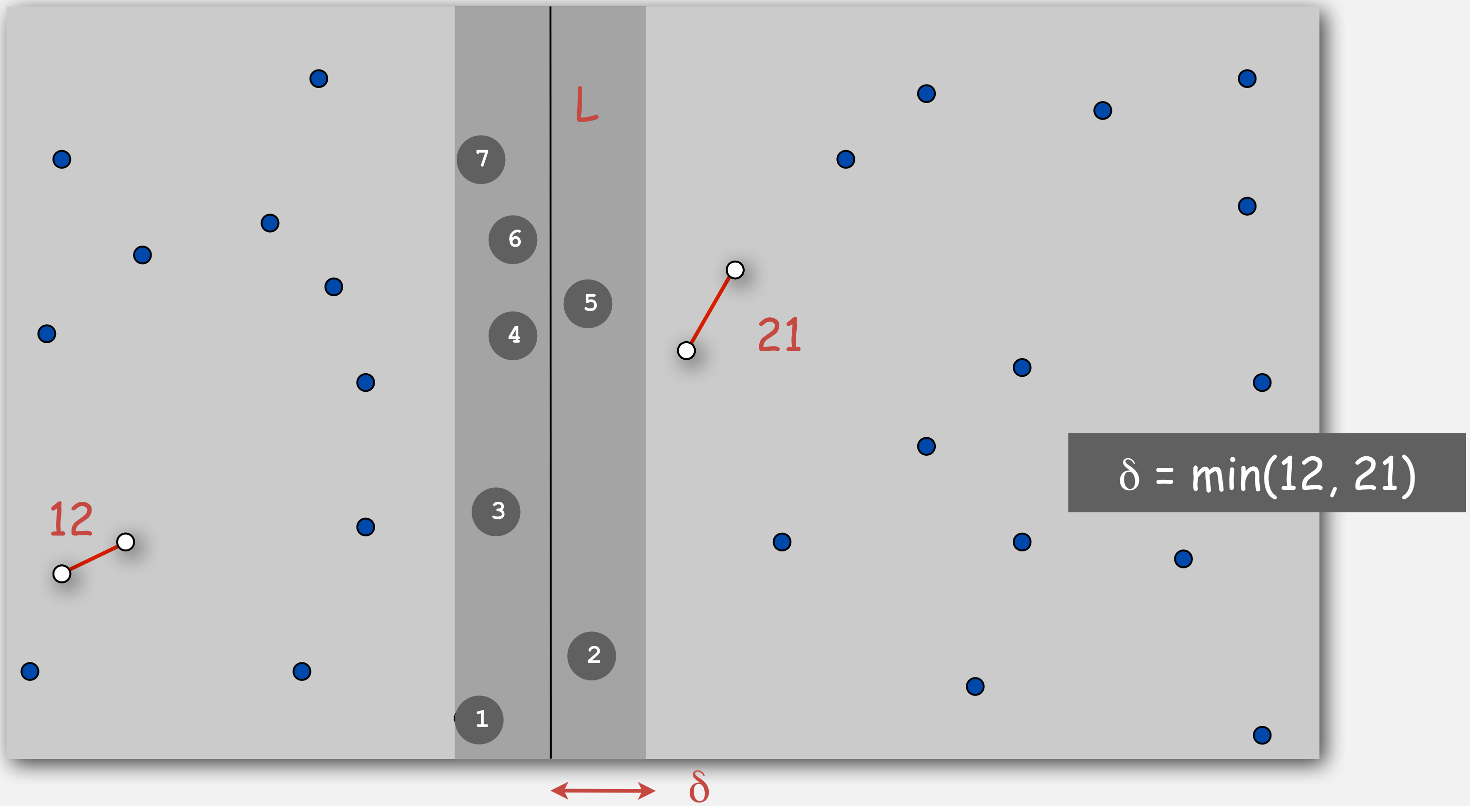


How to find closest pair with one point in each side?

Find closest pair with one point in each side, assuming that distance < δ.

- Observation: only need to consider points within δ of line L.
- Sort points in 2δ-strip by their y coordinate.
- Only check distances of those within 11 positions in sorted list!

why 11?



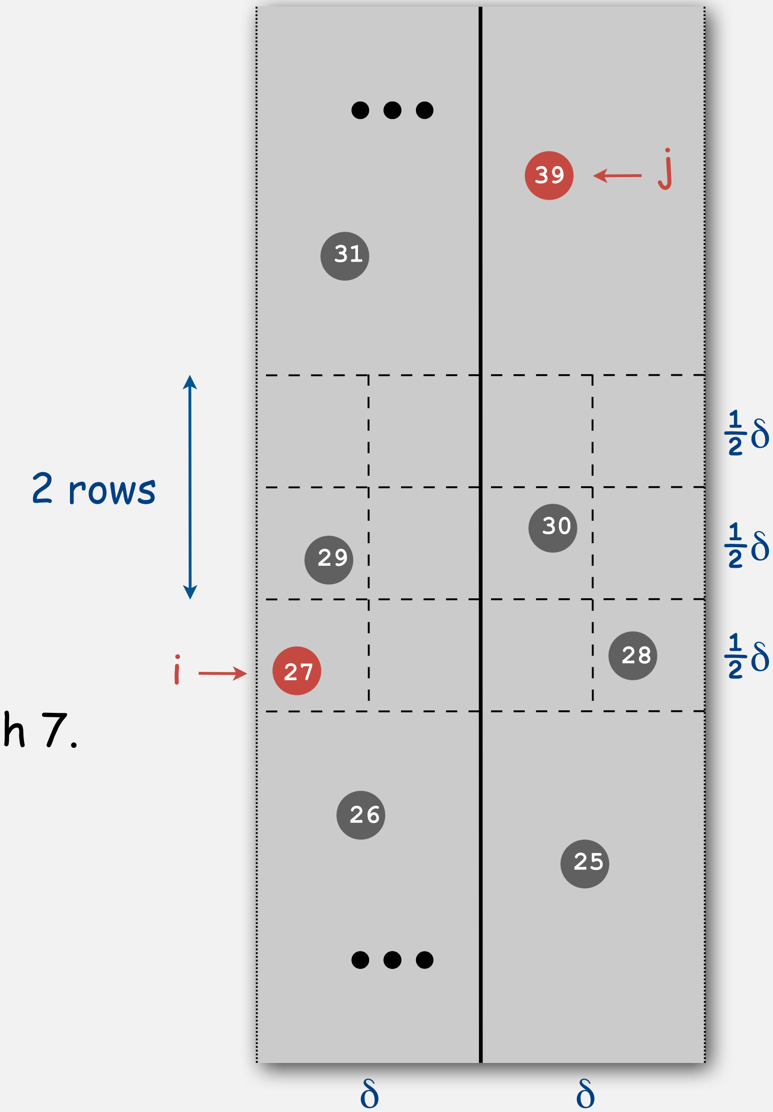
How to find closest pair with one point in each side?

Def. Let s_i be the point in the 2δ-strip, with the i^{th} smallest y-coordinate.

Claim. If $|i - j| \geq 12$, then the distance between s_i and s_j is at least δ.

Pf.

- No two points lie in same $\frac{1}{2}\delta$-by-$\frac{1}{2}\delta$ box.
- Two points at least 2 rows apart have distance $\geq 2(\frac{1}{2}\delta)$. ▪



Fact. Claim remains true if we replace 12 with 7.

Divide-and-conquer algorithm

```
Closest-Pair(p1, …, pn)
{
   Compute separation line L such that half the points
   are on one side and half on the other side.

   δ1 = Closest-Pair(left half)
   δ2 = Closest-Pair(right half)
   δ  = min(δ1, δ2)

   Delete all points further than δ from separation line L

   Sort remaining points by y-coordinate.

   Scan points in y-order and compare distance between
   each point and next 11 neighbors. If any of these
   distances is less than δ, update δ.

   return δ.
}
```

$O(N \log N)$

$2T(N / 2)$

$O(N)$

$O(N \log N)$

$O(N)$

Divide-and-conquer algorithm: analysis

Running time recurrence. $T(N) \le 2T(N/2) + O(N \log N)$.

Solution. $T(N) = O(N (\log N)^2)$.

Remark. Can be improved to $O(N \log N)$.

sort by x- and y-coordinates once
(reuse later to avoid re-sorting)

$(x_1 - x_2)^2 + (y_1 - y_2)^2$

Lower bound. In quadratic decision tree model, any algorithm for closest pair requires $\Omega(N \log N)$ steps.

- primitive operations
- convex hull
- closest pair
- **voronoi diagram**

1854 cholera outbreak, Golden Square, London

Life-or-death question.

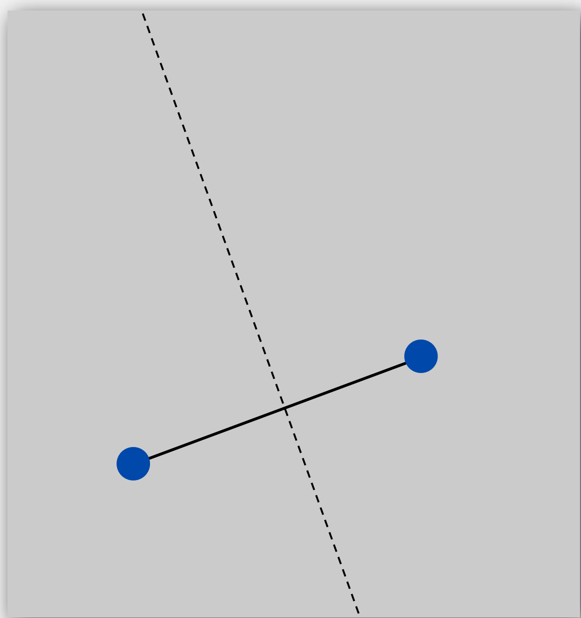
Given a new cholera patient p, which water pump is closest to p's home?

http://content.answers.com/main/content/wp/en/c/c7/Snow-cholera-map.jpg

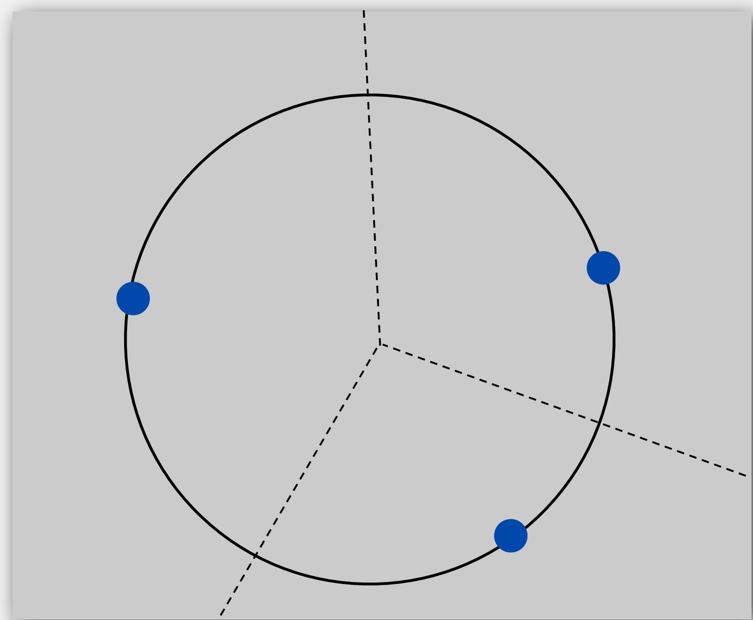
Voronoi region. Set of all points closest to a given point.

Voronoi diagram. Planar subdivision delineating Voronoi regions.

Fact. Voronoi edges are perpendicular bisector segments.



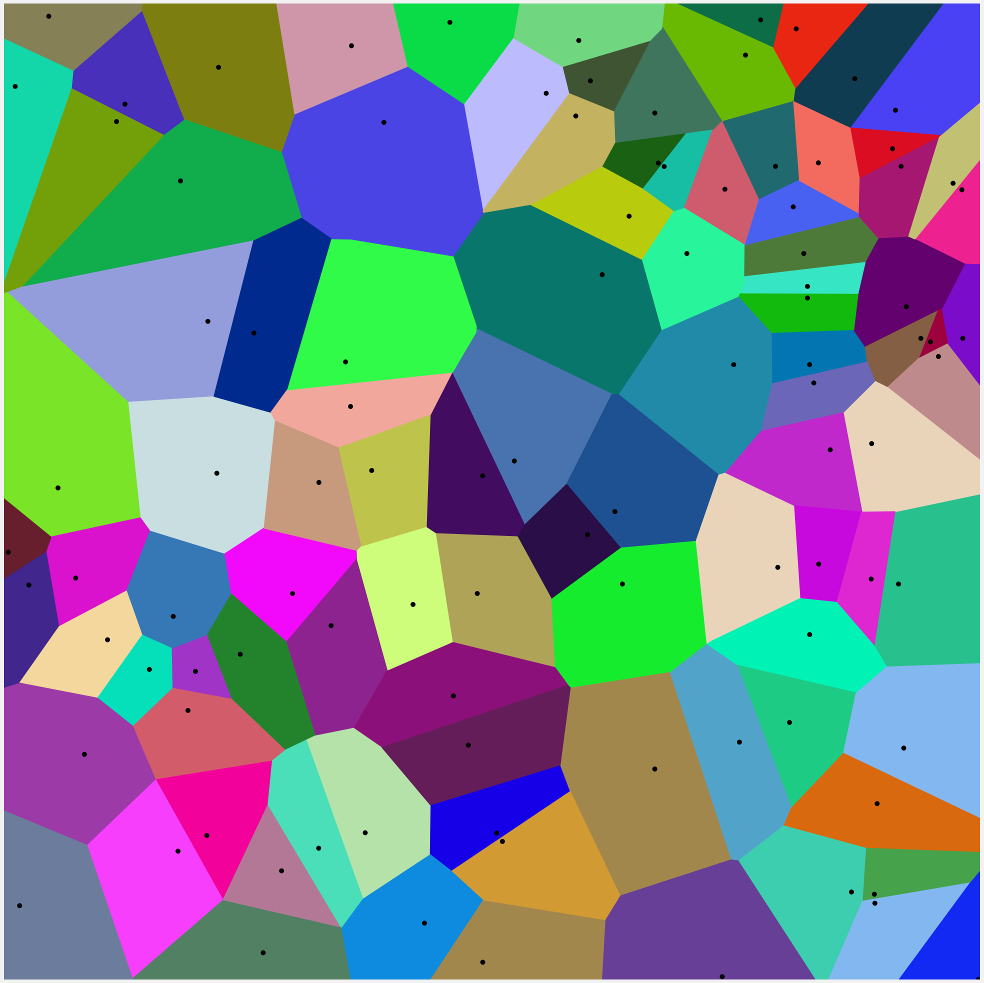
Voronoi of 2 points
(perpendicular bisector)



Voronoi of 3 points
(passes through circumcenter)

Voronoi region. Set of all points closest to a given point.

Voronoi diagram. Planar subdivision delineating Voronoi regions.



Voronoi diagram: more applications

Anthropology. Identify influence of clans and chiefdoms on geographic regions.

Astronomy. Identify clusters of stars and clusters of galaxies.

Biology, Ecology, Forestry. Model and analyze plant competition.

Cartography. Piece together satellite photographs into large "mosaic" maps.

Crystallography. Study Wigner-Setiz regions of metallic sodium.

Data visualization. Nearest neighbor interpolation of 2D data.

Finite elements. Generating finite element meshes which avoid small angles.

Fluid dynamics. Vortex methods for inviscid incompressible 2D fluid flow.

Geology. Estimation of ore reserves in a deposit using info from bore holes.

Geo-scientific modeling. Reconstruct 3D geometric figures from points.

Marketing. Model market of US metro area at individual retail store level.

Metallurgy. Modeling "grain growth" in metal films.

Physiology. Analysis of capillary distribution in cross-sections of muscle tissue.

Robotics. Path planning for robot to minimize risk of collision.

Typography. Character recognition, beveled and carved lettering.

Zoology. Model and analyze the territories of animals.

http://voronoi.com http://www.ics.uci.edu/~eppstein/geom.html

Scientific rediscoveries

year	discoverer	discipline	name
1644	Descartes	astronomy	"Heavens"
1850	Dirichlet	math	Dirichlet tesselation
1908	Voronoi	math	Voronoi diagram
1909	Boldyrev	geology	area of influence polygons
1911	Thiessen	meteorology	Thiessen polygons
1927	Niggli	crystallography	domains of action
1933	Wigner-Seitz	physics	Wigner-Seitz regions
1958	Frank-Casper	physics	atom domains
1965	Brown	ecology	area of potentially available
1966	Mead	ecology	plant polygons
1985	Hoofd et al.	anatomy	capillary domains

Reference: Kenneth E. Hoff III

Fortune's algorithm

Industrial-strength Voronoi implementation.

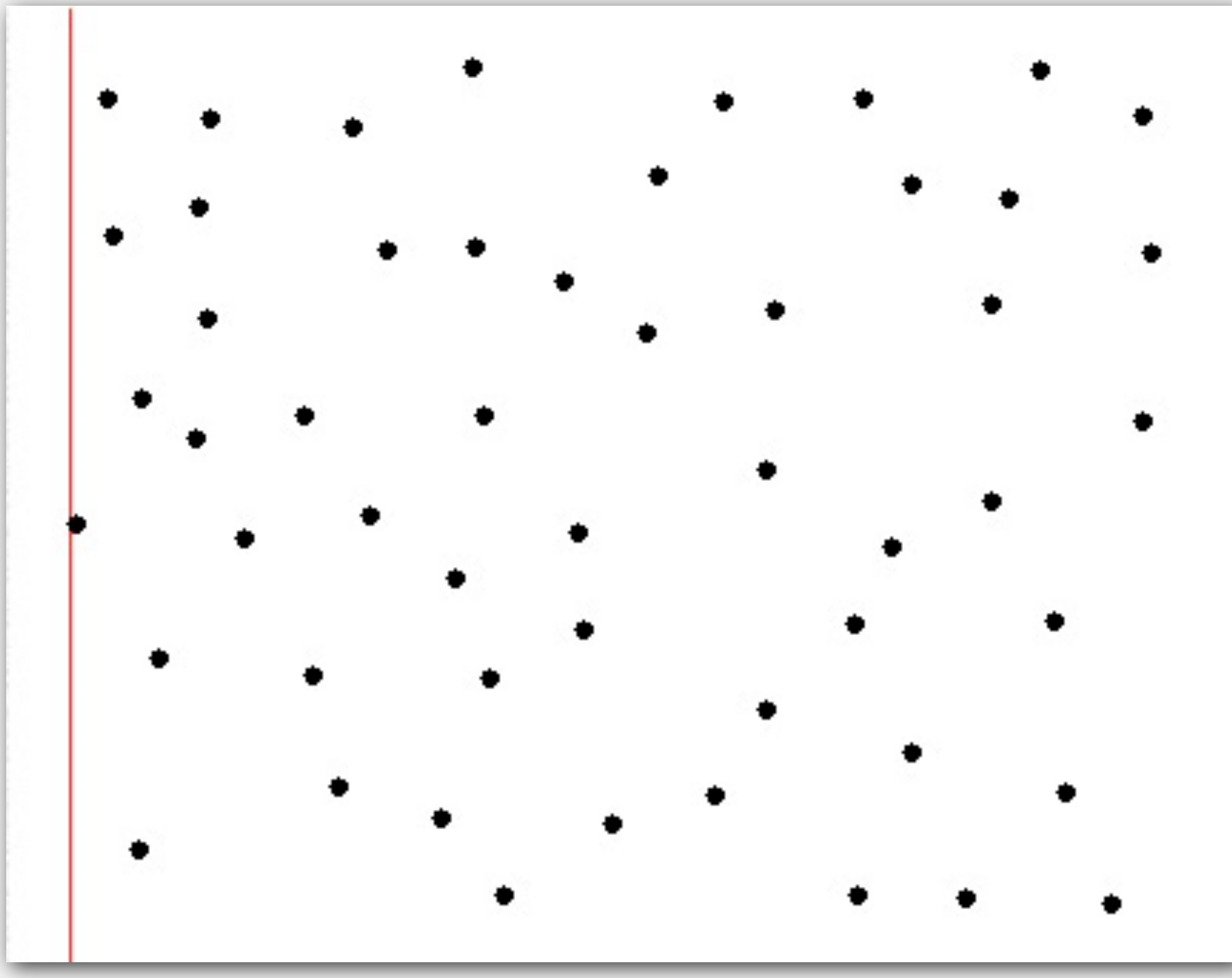
- Sweep-line algorithm.
- O(N log N) time.
- Properly handles degeneracies.
- Properly handles floating-point computations.

algorithm	preprocess	query
brute	1	N
Fortune	N log N	log N

Try it yourself! `http://www.diku.dk/hjemmesider/studerende/duff/Fortune/`

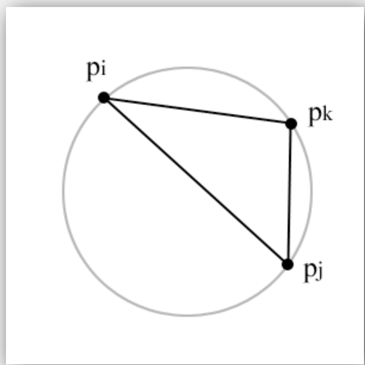
Remark. Beyond scope of this course.

Fortune's algorithm in practice

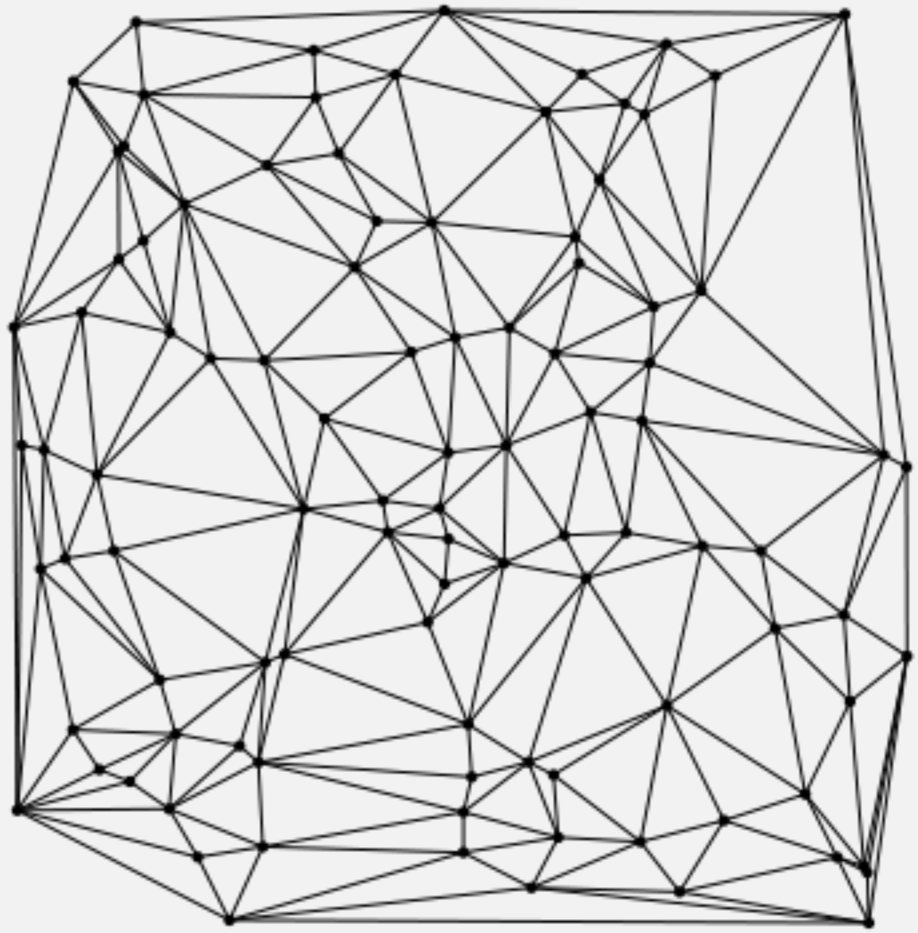


Delaunay triangulation

Def. Triangulation of N points such that no point is inside circumcircle of any other triangle.



circumcircle of 3 points



Delaunay triangulation properties

Proposition 1. It exists and is unique (assuming no degeneracy).

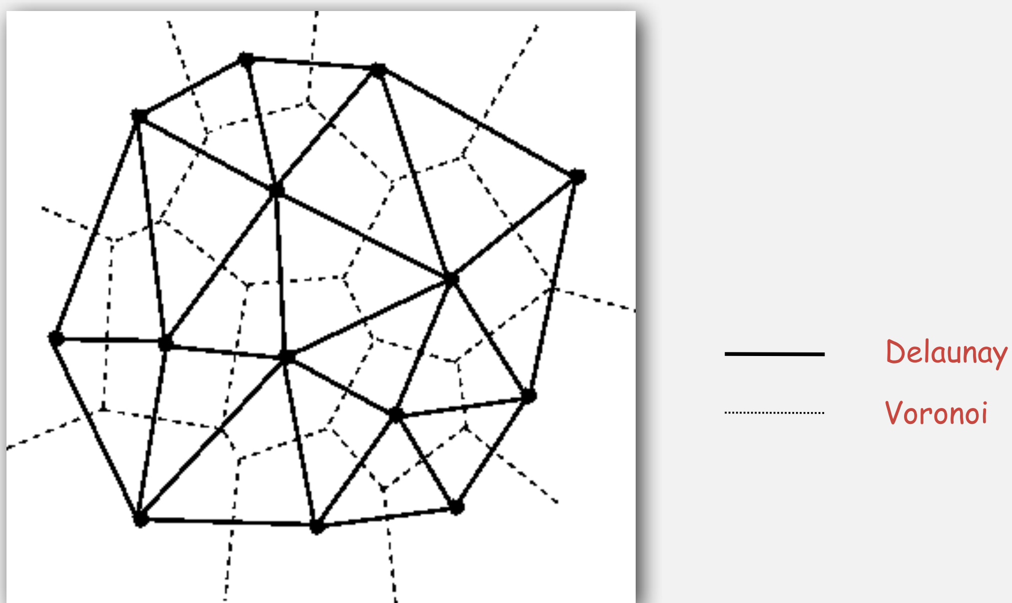
Proposition 2. Dual of Voronoi (connect adjacent points in Voronoi diagram).

Proposition 3. No edges cross $\Rightarrow$ O(N) edges.

Proposition 4. Maximizes the minimum angle for all triangular elements.

Proposition 5. Boundary of Delaunay triangulation is convex hull.

Proposition 6. Shortest Delaunay edge connects closest pair of points.



Delaunay triangulation application: Euclidean MST

Euclidean MST. Given N points in the plane, find MST connecting them.

[distances between point pairs are Euclidean distances]

Brute force. Compute $N^2 / 2$ distances and run Prim's algorithm.

Ingenuity.

- MST is subgraph of Delaunay triangulation.
- Delaunay has O(N) edges.
- Compute Delaunay, then use Prim (or Kruskal) to get MST in O(N log N) !

Geometric algorithms summary

Ingenious algorithms enable solution of large instances for numerous fundamental geometric problems.

problem	brute	clever
convex hull	N^2	N log N
farthest pair	N^2	N log N
closest pair	N^2	N log N
Delaunay/Voronoi	N^4	N log N
Euclidean MST	N^2	N log N

asymptotic time to solve a 2D problem with N points

Note. 3D and higher dimensions test limits of our ingenuity.

6.3 Geometric Search

- range search
- space partitioning trees
- intersection search

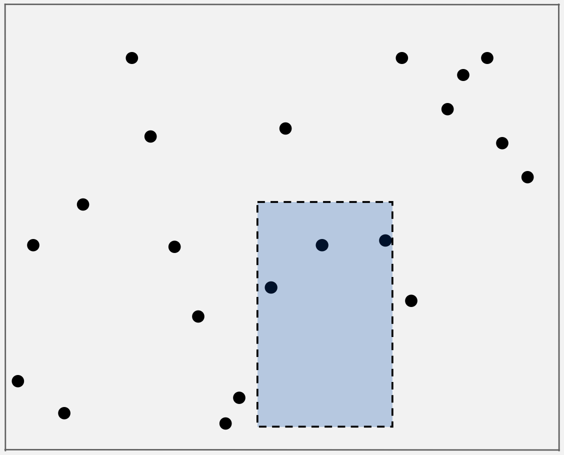
Overview

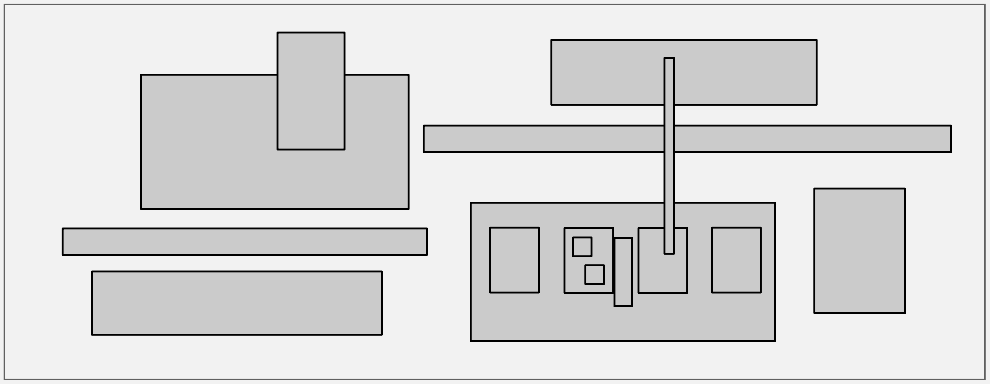
Geometric objects. Points, lines, intervals, circles, rectangles, polygons, ...

This lecture. Intersection among N objects.

Example problems.

- 1D range search.
- 2D range search.
- Find all intersections among h-v line segments.
- Find all intersections among h-v rectangles.





- range search
- space partitioning trees
- intersection search

1d range search

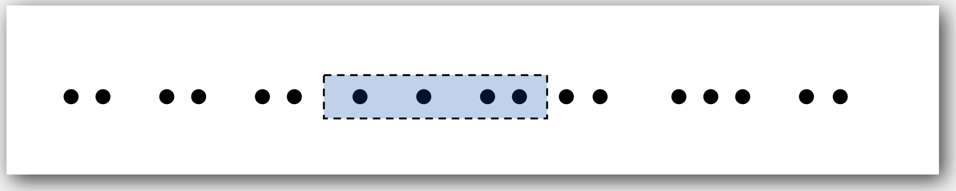
Extension of ordered symbol table.

- Insert key-value pair.
- Search for key k.
- Rank: how many keys less than k?
- Range search: find all keys between k_1 and k_2.

Application. Database queries.

Geometric interpretation.

- Keys are point on a line.
- How many points in a given interval?



```
insert B          B
insert D          B D
insert A          A B D
insert I          A B D I
insert H          A B D H I
insert F          A B D F H I
insert P          A B D F H I P
count G to K      2
search G to K     H I
```

1d range search: implementations

Ordered array. Slow insert, binary search for `lo` and `hi` to find range.

Hash table. No reasonable algorithm (key order lost in hash).

data structure	insert	rank	range count	range search
ordered array	N	log N	log N	R + log N
hash table	1	N	N	N
BST	log N	log N	log N	R + log N

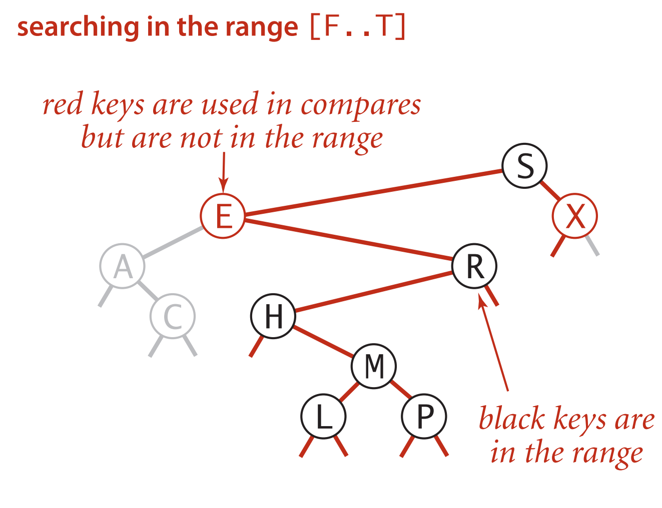
N = # keys
R = # keys that match

BST. All operations fast.

1d range search: BST implementation

Range search. Find all keys between `lo` and `hi`?

- Recursively find all keys in left subtree (if any could fall in range).
- Check key in current node.
- Recursively find all keys in right subtree (if any could fall in range).



Range search in a BST

Worst-case running time. R + log N (assuming BST is balanced).

2d orthogonal range search

Extension of ordered symbol-table to 2d keys.

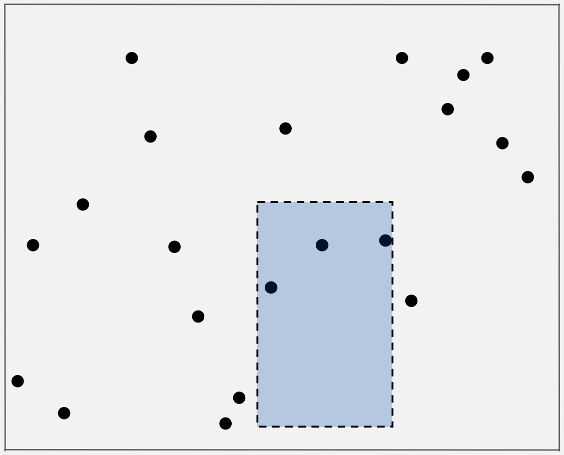
- Insert a 2d key.
- Search for a 2d key.
- Range search: find all keys that lie in a 2d range?

Applications. Networking, circuit design, databases.

Geometric interpretation.

- Keys are point in the plane.
- How many points in a given h-v rectangle.

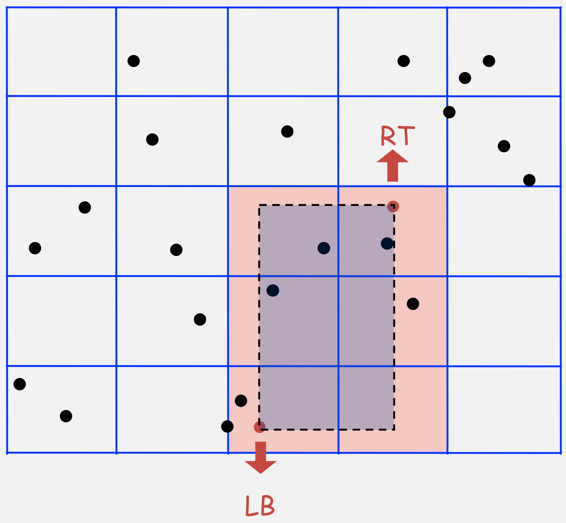
rectangle is axis-aligned



2d orthogonal range search: grid implementation

Grid implementation.

- Divide space into M-by-M grid of squares.
- Create list of points contained in each square.
- Use 2d array to directly index relevant square.
- Insert: add (x, y) to list for corresponding square.
- Range search: examine only those squares that intersect 2d range query.



2d orthogonal range search: grid implementation costs

Space-time tradeoff.

- Space: $M^2 + N$.
- Time: $1 + N / M^2$ per square examined, on average.

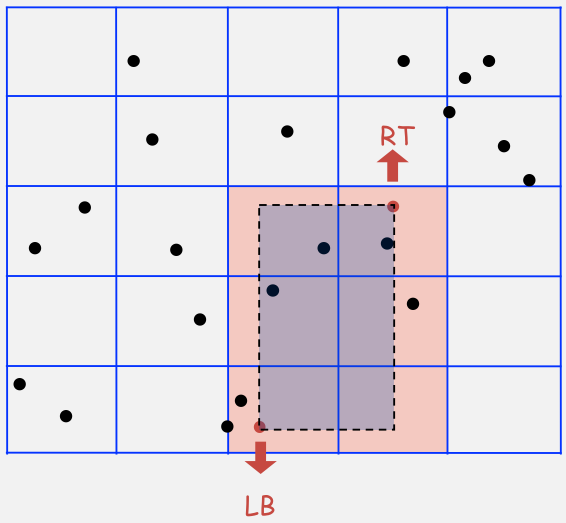
Choose grid square size to tune performance.

- Too small: wastes space.
- Too large: too many points per square.
- Rule of thumb: $\sqrt{N}$-by-$\sqrt{N}$ grid.

Running time. [if points are evenly distributed]

- Initialize: $O(N)$.
- Insert: $O(1)$.
- Range: $O(1)$ per point in range.

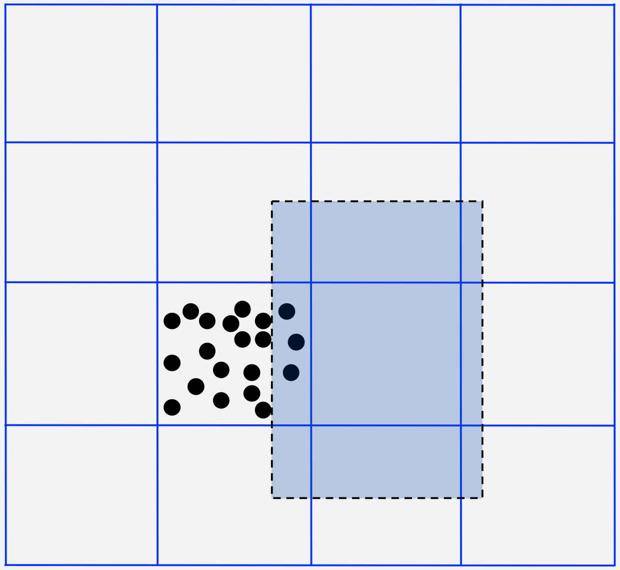
$M \sim \sqrt{N}$



Clustering

Grid implementation. Fast, simple solution for well-distributed points.

Problem. Clustering a well-known phenomenon in geometric data.



Lists are too long, even though average length is short.

Need data structure that gracefully adapts to data.

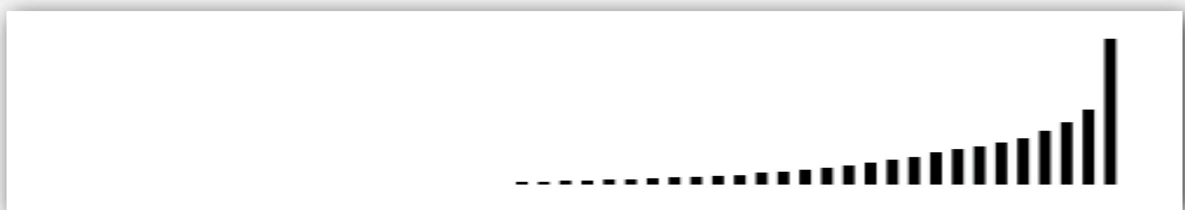
Clustering

Grid implementation. Fast, simple solution for well-distributed points.

Problem. Clustering a well-known phenomenon in geometric data.

Ex. USA map data.

13,000 points, 1000 grid squares



half the squares are empty

half the points are in 10% of the squares

- range search
- **space partitioning trees**
- intersection search

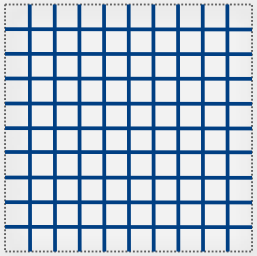
Space-partitioning trees

Use a tree to represent a recursive subdivision of 2D space.

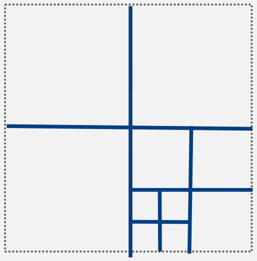
Quadtree. Recursively divide space into four quadrants.

2d tree. Recursively divide space into two halfplanes.

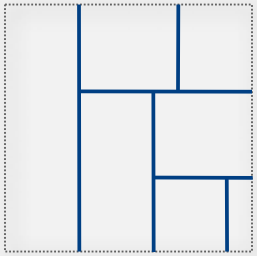
BSP tree. Recursively divide space into two regions.



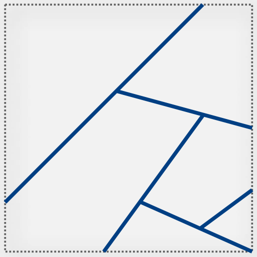
Grid



Quadtree



2D tree

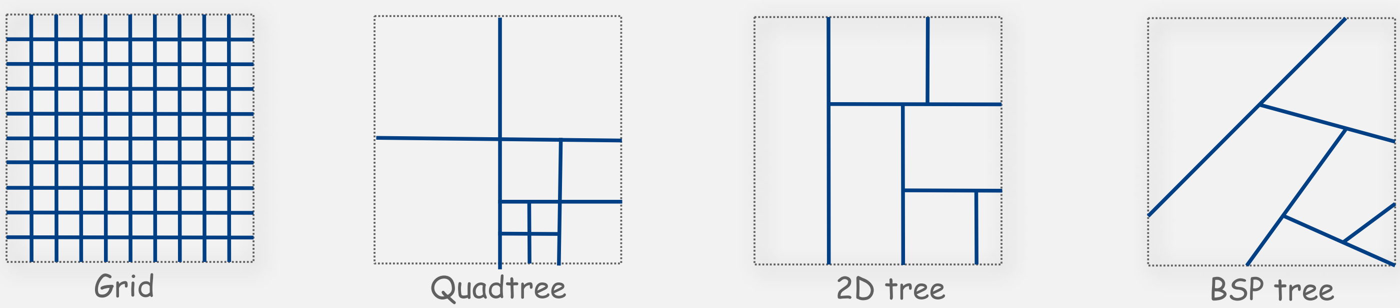


BSP tree

Space-partitioning trees: applications

Applications.

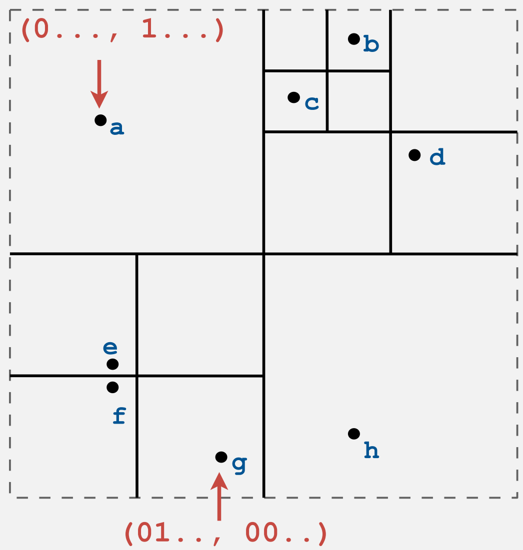
- Ray tracing.
- 2d range search.
- Flight simulators.
- N-body simulation.
- Collision detection.
- Astronomical databases.
- Nearest neighbor search.
- Adaptive mesh generation.
- Accelerate rendering in Doom.
- Hidden surface removal and shadow casting.

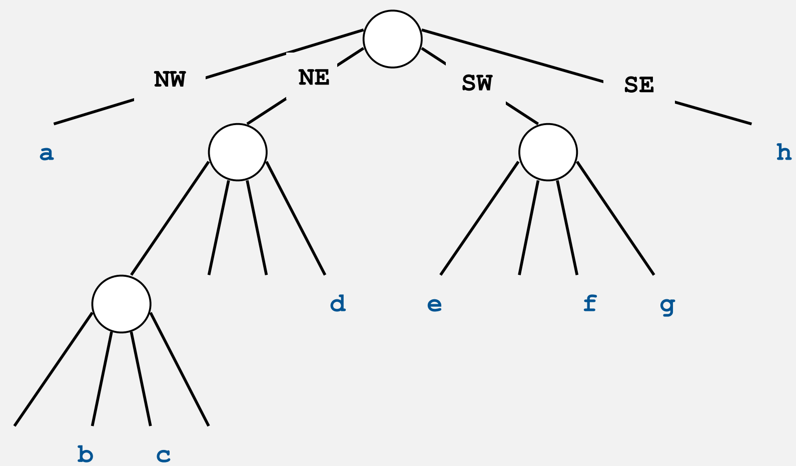


Quadtree

Idea. Recursively divide space into 4 quadrants.

Implementation. 4-way tree (actually a trie).



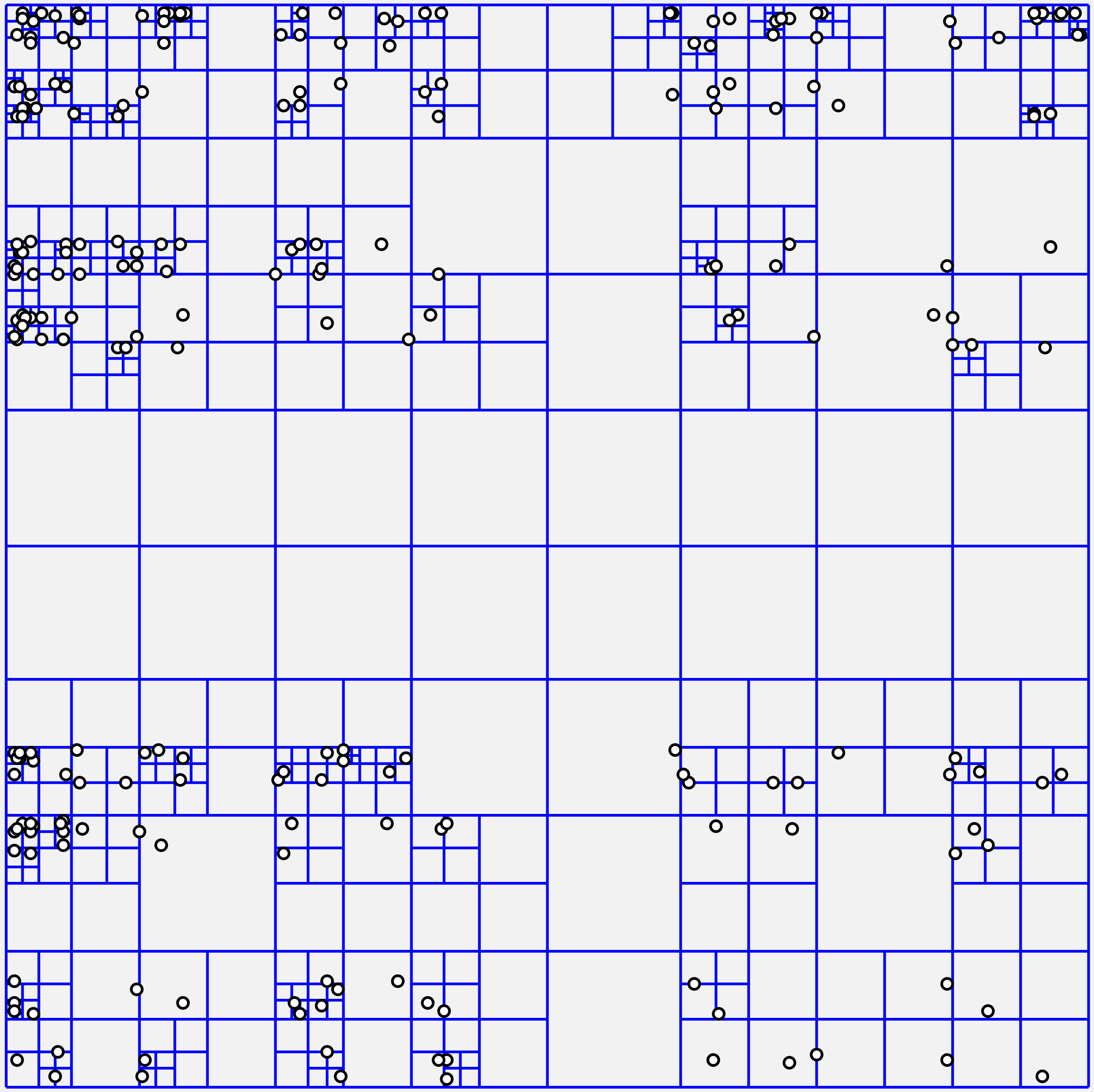



```
public class QuadTree
{
   private Quad quad;
   private Value val;
   private QuadTree NW, NE, SW, SE;
}
```

Benefit. Good performance in the presence of clustering.

Drawback. Arbitrary depth!

Quadtree: larger example

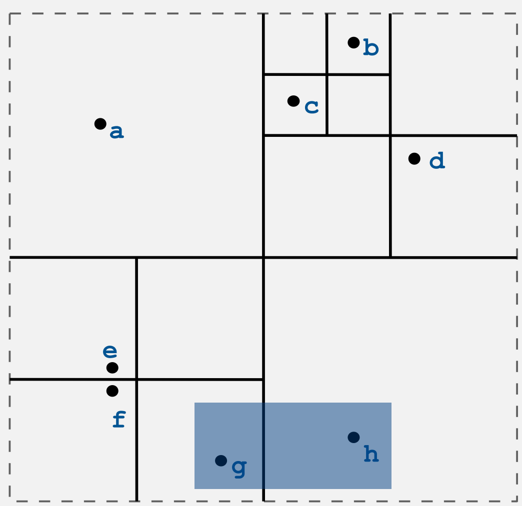


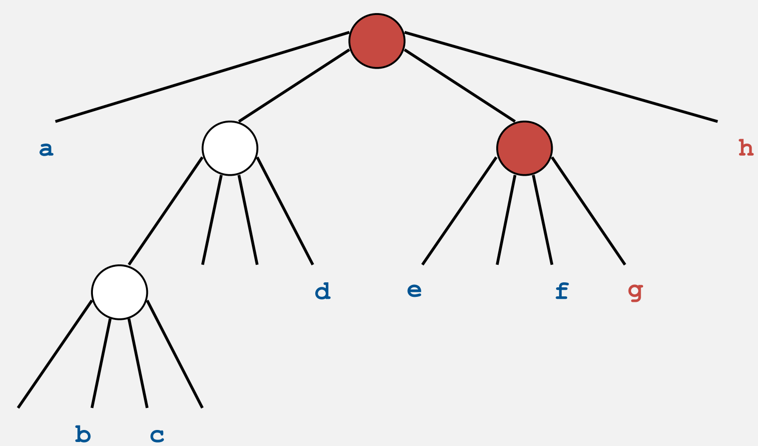
http://en.wikipedia.org/wiki/Image:Point_quadtree.svg

Quadtree: 2d range search

Range search. Find all keys in a given 2D range.

- Recursively find all keys in NE quad (if any could fall in range).
- Recursively find all keys in NW quad (if any could fall in range).
- Recursively find all keys in SE quad (if any could fall in range).
- Recursively find all keys in SW quad (if any could fall in range).

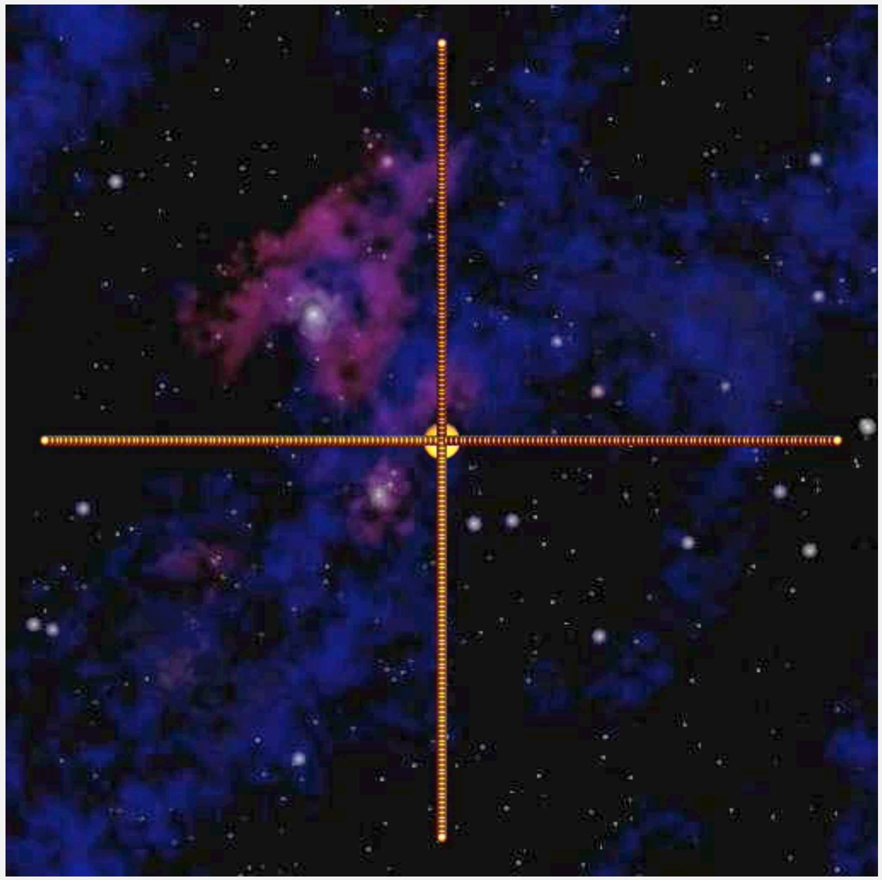




Typical running time. R + log N.

N-body simulation

Goal. Simulate the motion of N particles, mutually affected by gravity.



Brute force. For each pair of particles, compute force. $$F = \frac{G\, m_1\, m_2}{r^2}$$

Subquadratic N-body simulation

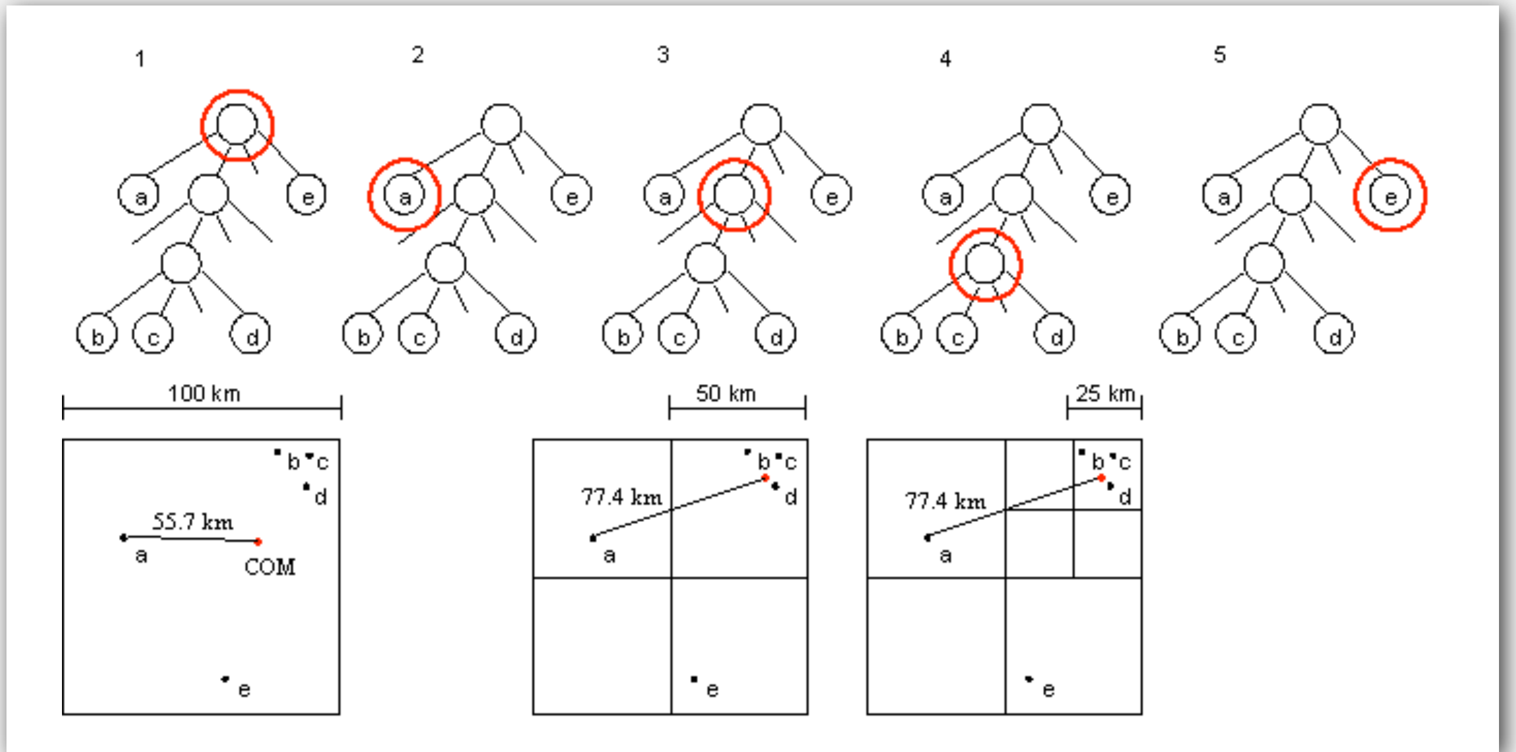
Key idea. Suppose particle is far, far away from cluster of particles.

- Treat cluster of particles as a single aggregate particle.
- Compute force between particle and center of mass of aggregate particle.

Barnes-Hut algorithm for N-body simulation.

Barnes-Hut.

- Build quadtree with N particles as external nodes.
- Store center-of-mass of subtree in each internal node.
- To compute total force acting on a particle, traverse tree, but stop as soon as distance from particle to quad is sufficiently large.



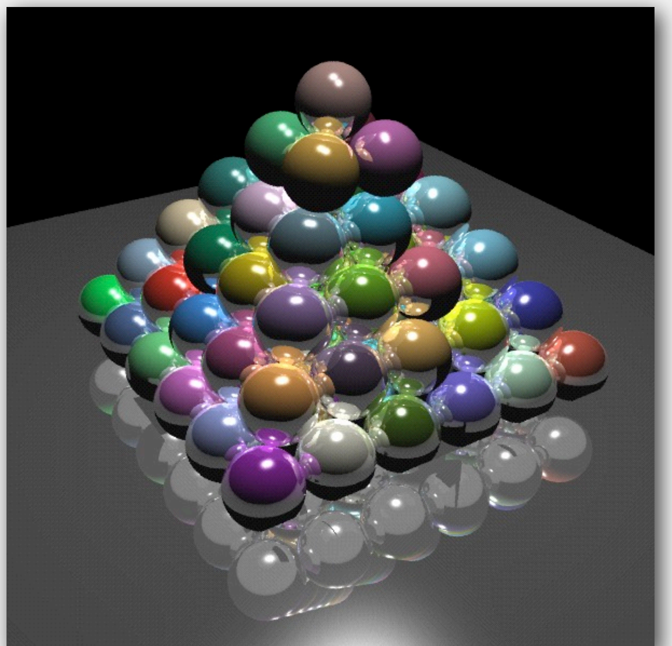
Curse of dimensionality

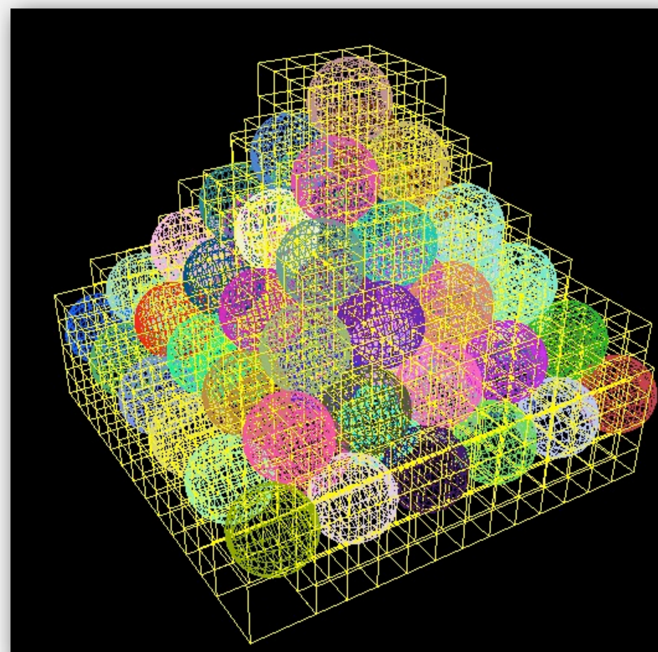
Range search / nearest neighbor in k dimensions?

Main application. Multi-dimensional databases.

3d space. Octrees: recursively divide 3d space into 8 octants.

100d space. Centrees: recursively divide 100d space into 2^{100} centrants???

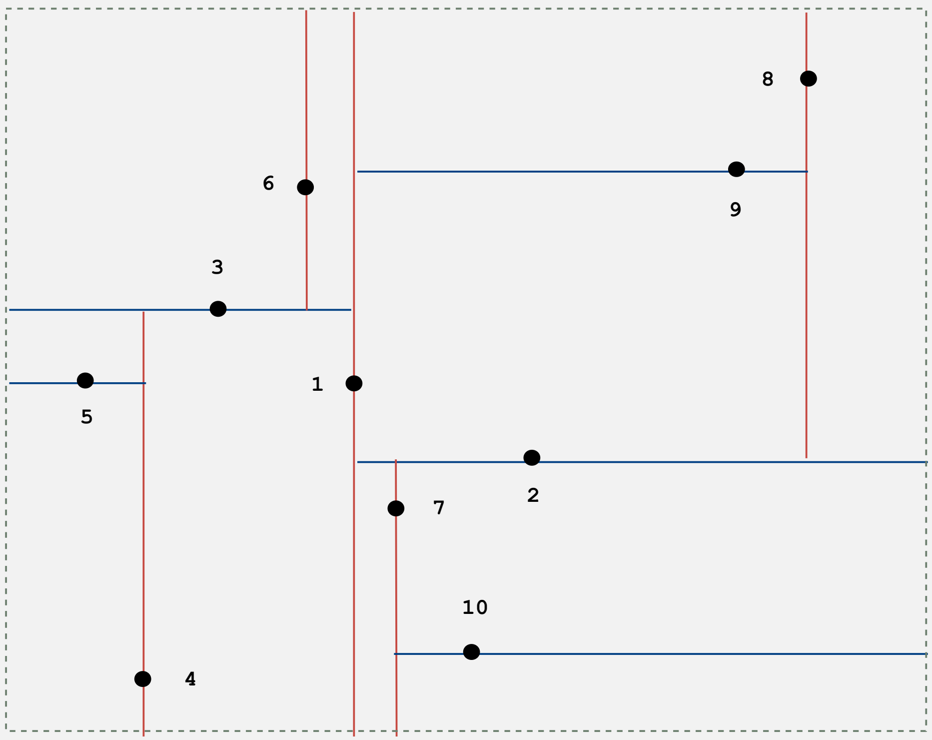


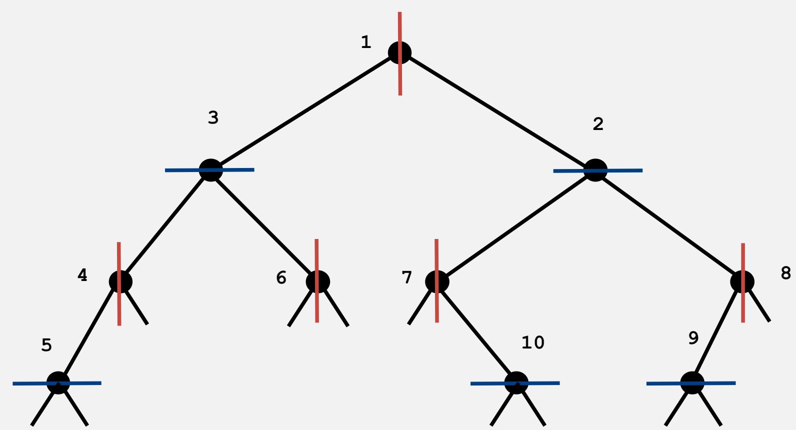


Raytracing with octrees
http://graphics.cs.ucdavis.edu/~gregorsk/graphics/275.html

2d tree

Recursively partition plane into two halfplanes.

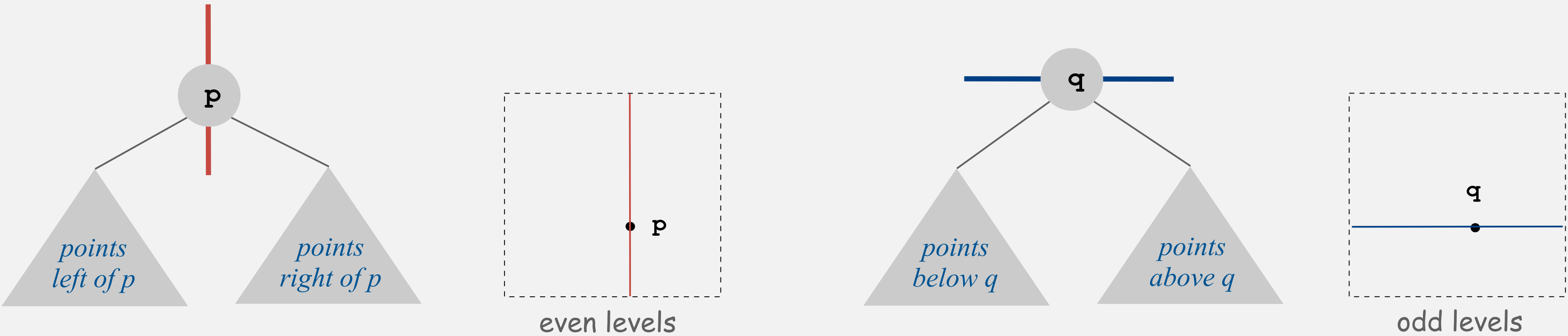


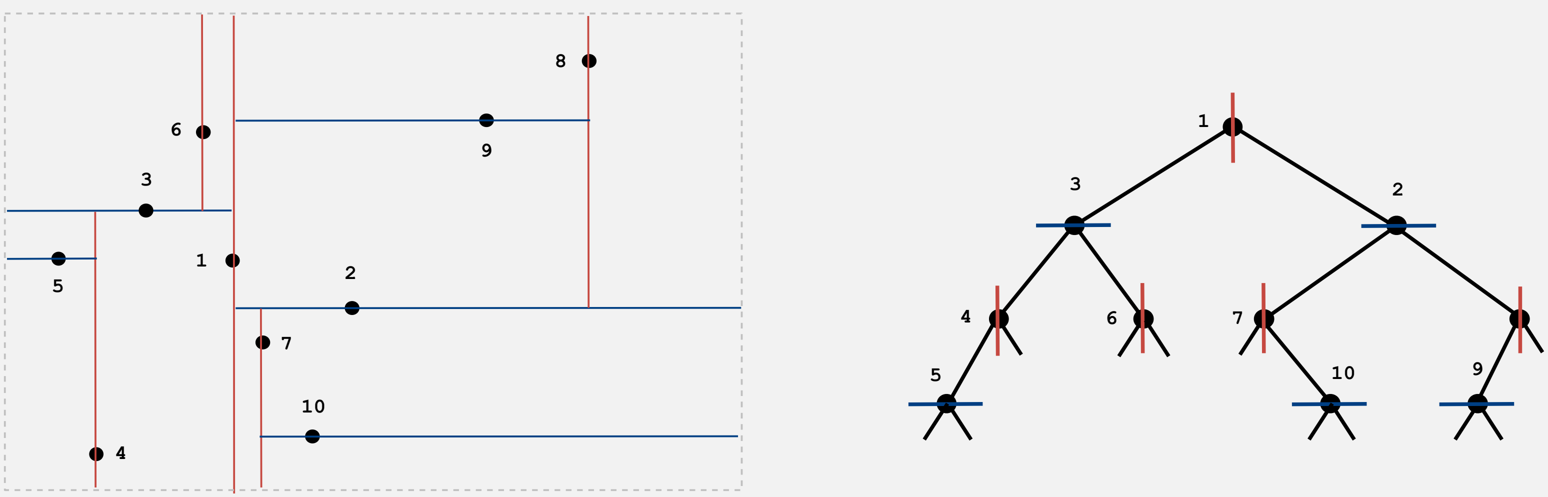


2d tree

Implementation. BST, but alternate using x- and y-coordinates as key.

- Search gives rectangle containing point.
- Insert further subdivides the plane.





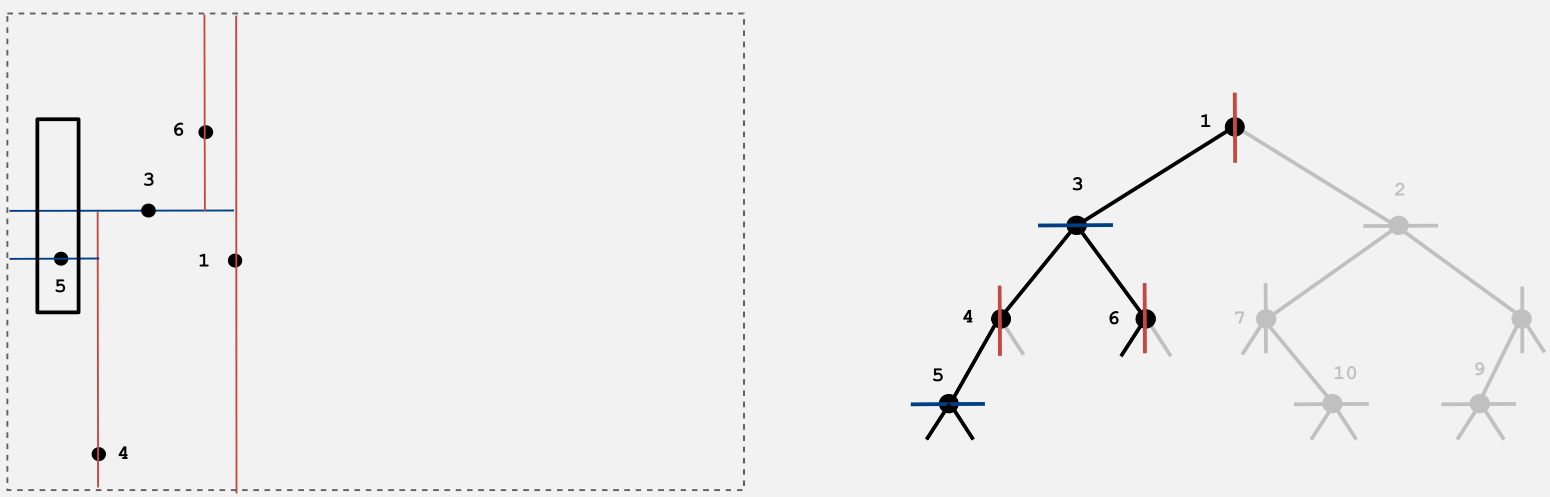
2d tree: 2d range search

Range search. Find all points in a query axis-aligned rectangle.

- Check if point in node lies in given rectangle.
- Recursively search left/top subdivision (if any could fall in rectangle).
- Recursively search right/bottom subdivision (if any could fall in rectangle).

Typical case. R + log N

Worst case (assuming tree is balanced). R + √N.



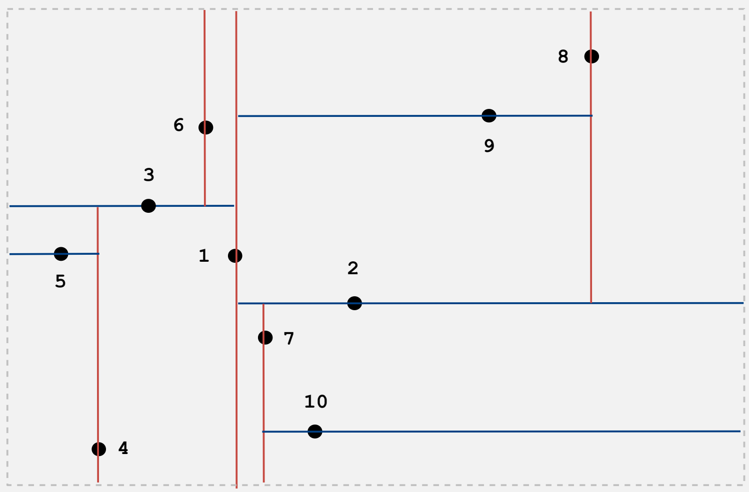
2d tree: nearest neighbor search

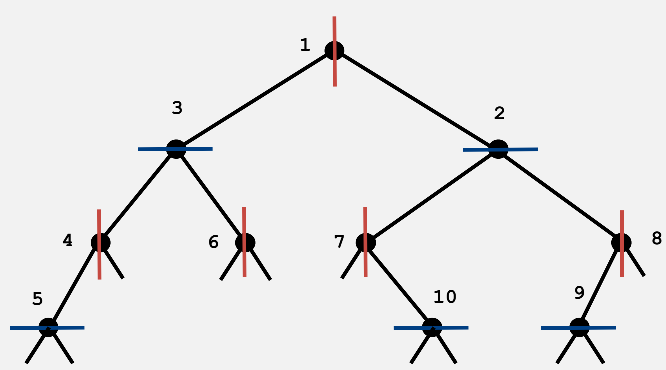
Nearest neighbor search. Given a query point, find the closest point.

- Check distance from point in node to query point.
- Recursively search left/top subdivision (if it could contain a closer point).
- Recursively search right/bottom subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

Typical case. log N

Worst case (even if tree is balanced). N

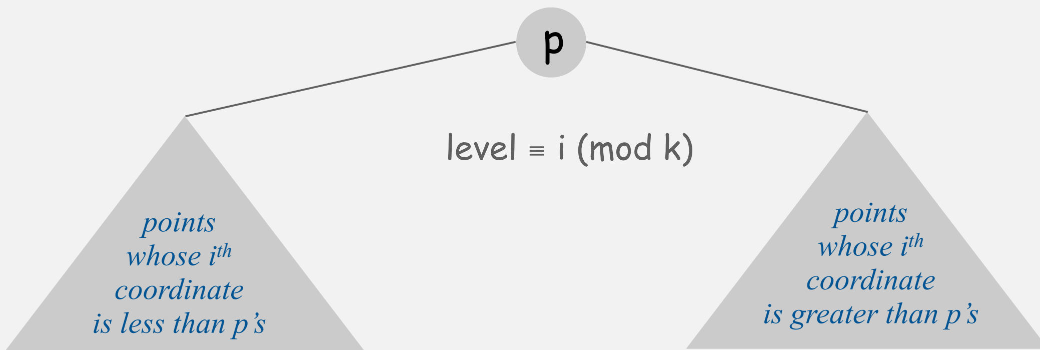


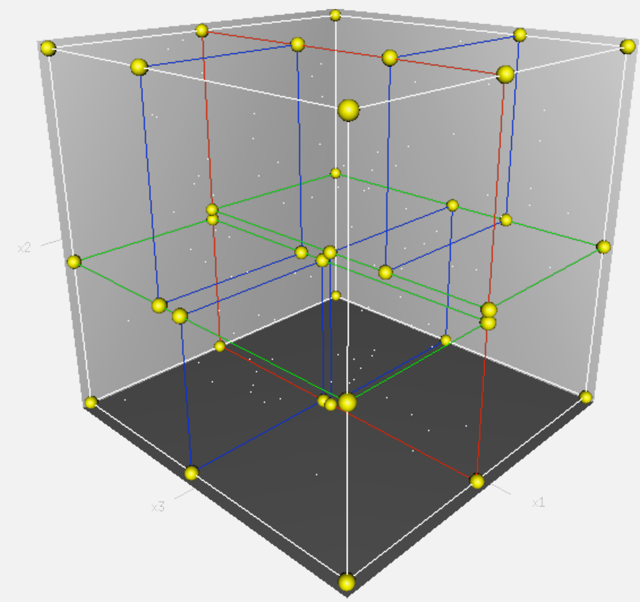


Kd tree

Kd tree. Recursively partition k-dimensional space into 2 halfspaces.

Implementation. BST, but cycle through dimensions ala 2d trees.





Efficient, simple data structure for processing k-dimensional data.

- Widely used.
- Discovered by an undergrad in an algorithms class!
- Adapts well to high-dimensional and clustered data.

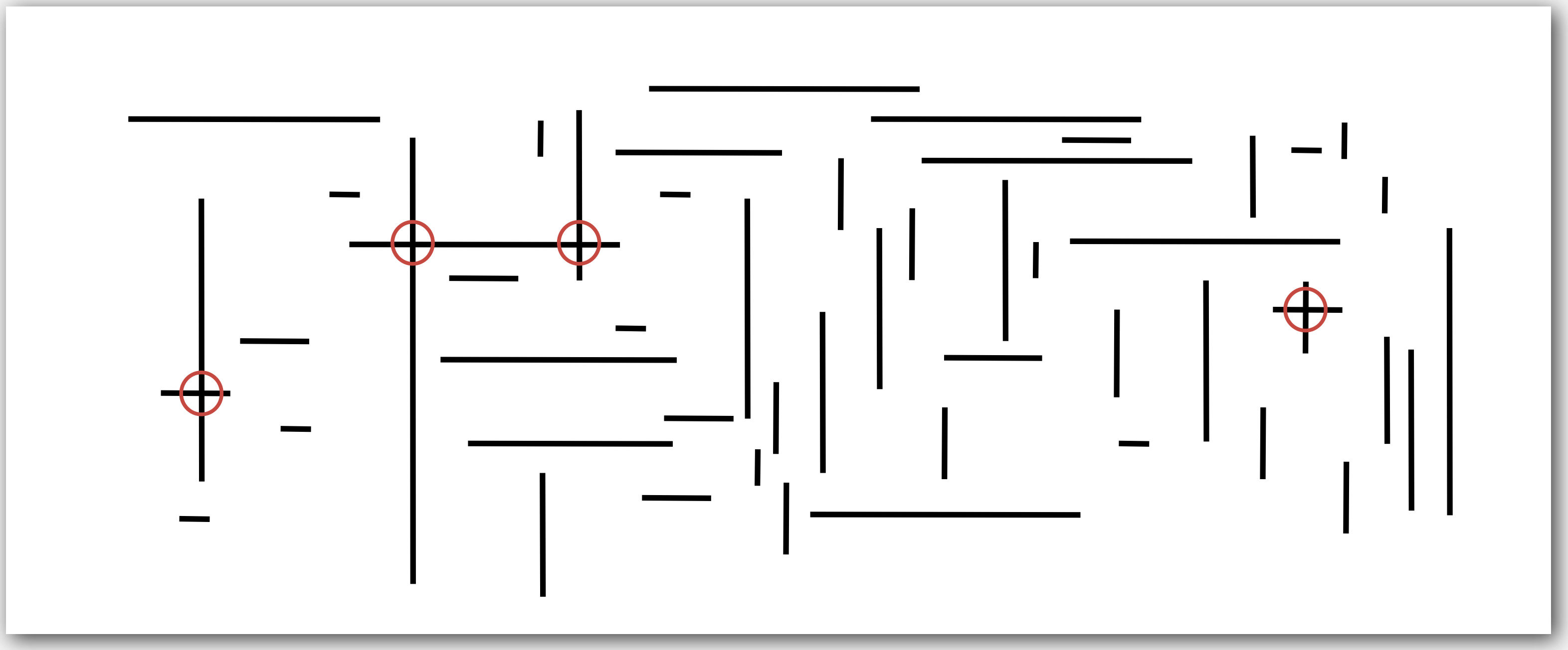
- range search
- space partitioning trees
- **intersection search**

Search for intersections

Problem. Find all intersecting pairs among N geometric objects.

Applications. CAD, games, movies, virtual reality.

Simple version. 2D, all objects are horizontal or vertical line segments.

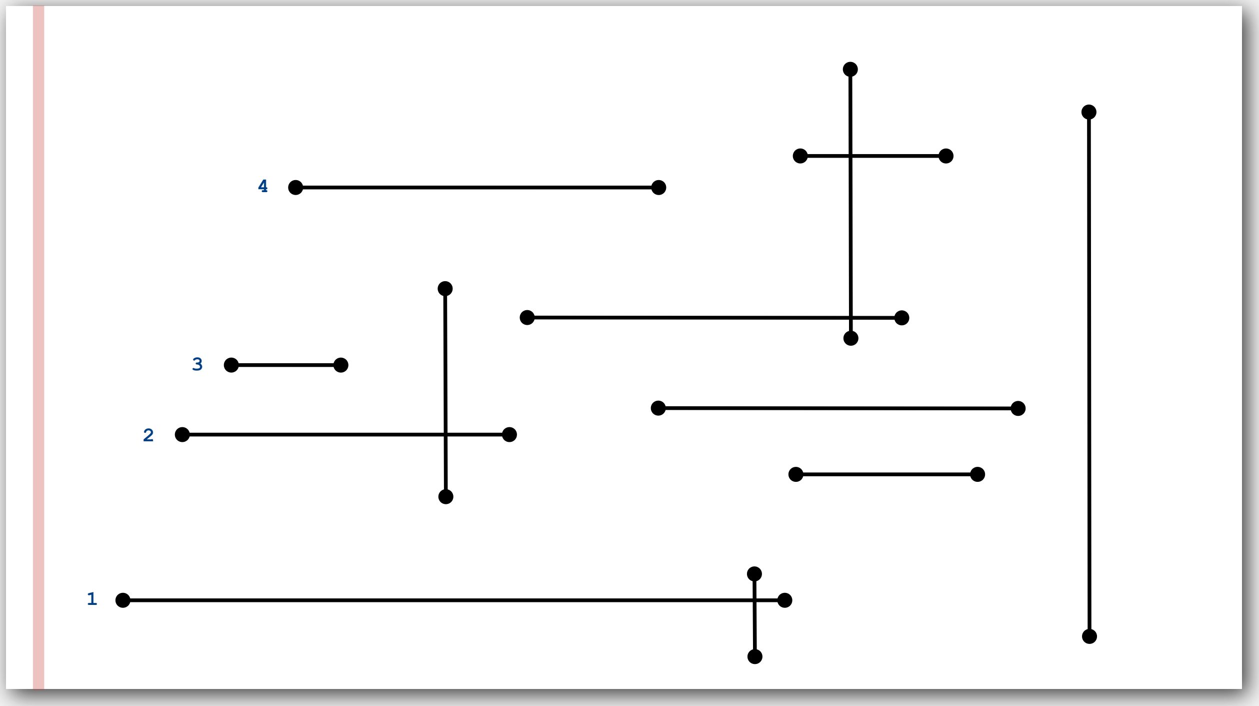


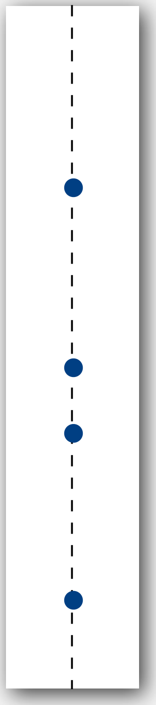
Brute force. Test all $\Theta(N^2)$ pairs of line segments for intersection.

Orthogonal segment intersection search: sweep-line algorithm

Sweep vertical line from left to right.

- x-coordinates define events.
- Left endpoint of h-segment: insert y-coordinate into ST.



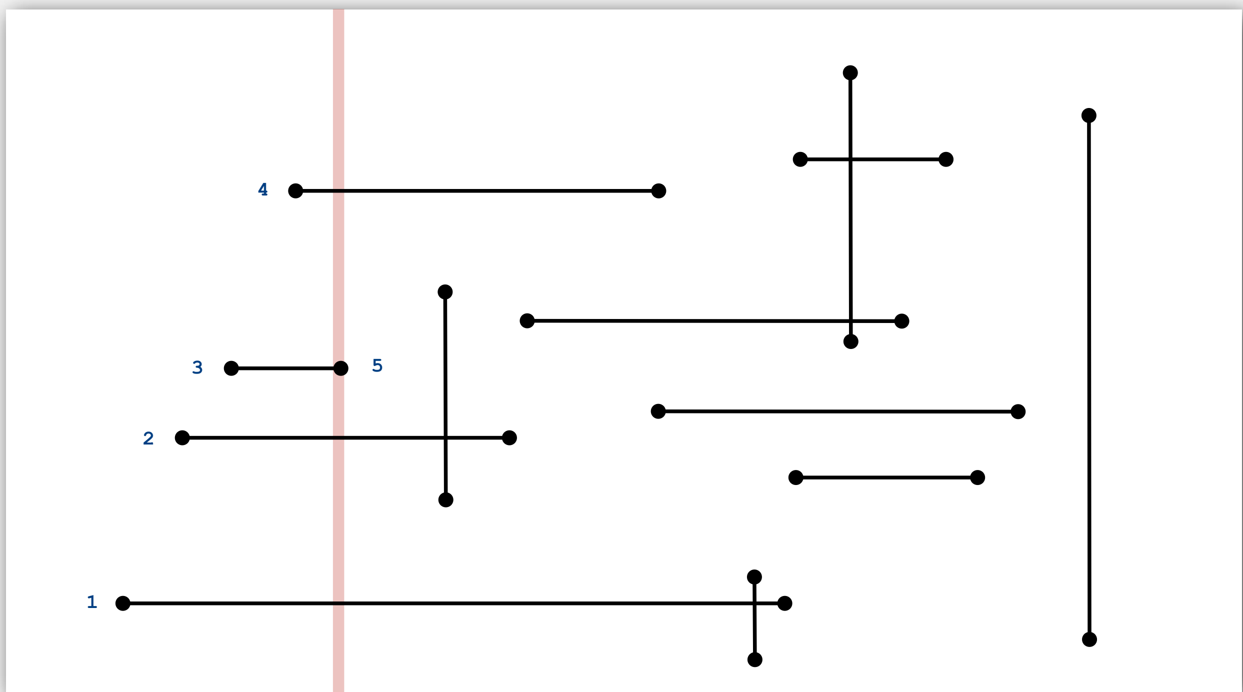


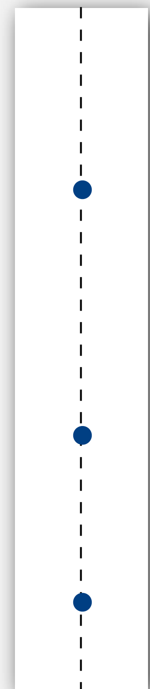
y-coordinates

Orthogonal segment intersection search: sweep-line algorithm

Sweep vertical line from left to right.

- x-coordinates define events.
- Left endpoint of h-segment: insert y-coordinate into ST.
- Right endpoint of h-segment: remove y-coordinate from ST.



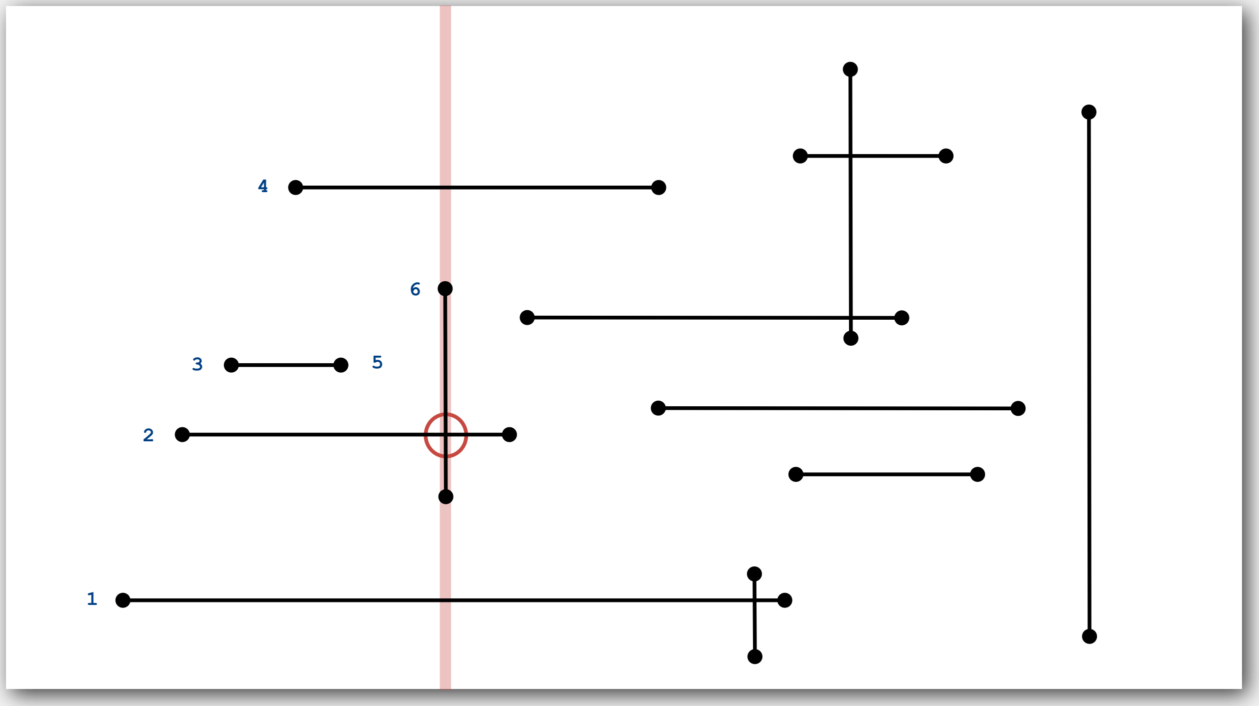


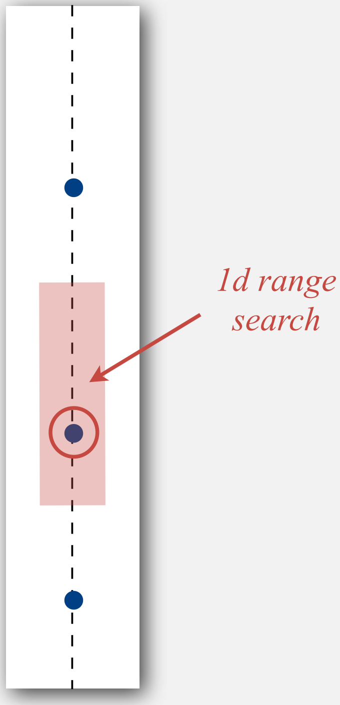
y-coordinates

Orthogonal segment intersection search: sweep-line algorithm

Sweep vertical line from left to right.

- x-coordinates define events.
- Left endpoint of h-segment: insert y-coordinate into ST.
- Right endpoint of h-segment: remove y-coordinate from ST.
- v-segment: range search for interval of y endpoints.





y-coordinates

Orthogonal segment intersection search: sweep-line algorithm

Reduces 2D orthogonal segment intersection search to 1D range search!

Running time of sweep line algorithm.

- Put x-coordinates on a PQ (or sort). O(N log N)
- Insert y-coordinate into ST. O(N log N)
- Delete y-coordinate from ST. O(N log N)
- Range search. O(R + N log N)

N = # line segments
R = # intersections

Efficiency relies on judicious use of data structures.

Remark. Sweep-line solution extends to 3D and more general shapes.

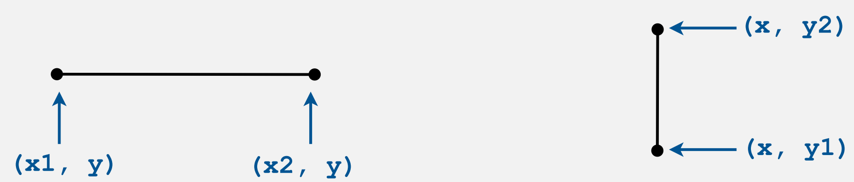
Immutable h-v segment data type

```
public final class SegmentHV implements Comparable<SegmentHV>
{
   public final int x1, y1;
   public final int x2, y2;

   public SegmentHV(int x1, int y1, int x2, int y2)
   {  ...  }

   public boolean isHorizontal()
   {  ...  }
   public boolean isVertical()
   {  ...  }

   public int compareTo(SegmentHV that)
   {  ...  }
}
```

constructor

is segment horizontal?
is segment vertical?

compare by x-coordinate;
break ties by y-coordinate



horizontal segment

vertical segment

Sweep-line event subclass

```
private class Event implements Comparable<Event>
{
   private int time;
   private SegmentHV segment;

   public Event(int time, SegmentHV segment)
   {
      this.time    = time;
      this.segment = segment;
   }

   public int compareTo(Event that)
   {  return this.time - that.time;   }
}
```

Sweep-line algorithm: initialize events

```
MinPQ<Event> pq = new MinPQ<Event>();

for (int i = 0; i < N; i++)
{
   if (segments[i].isVertical())
   {
      Event e = new Event(segments[i].x1, segments[i]);
      pq.insert(e);
   }

   else if (segments[i].isHorizontal())
   {
      Event e1 = new Event(segments[i].x1, segments[i]);
      Event e2 = new Event(segments[i].x2, segments[i]);
      pq.insert(e1);
      pq.insert(e2);
   }
}
```

initialize PQ

vertical segment

horizontal segment

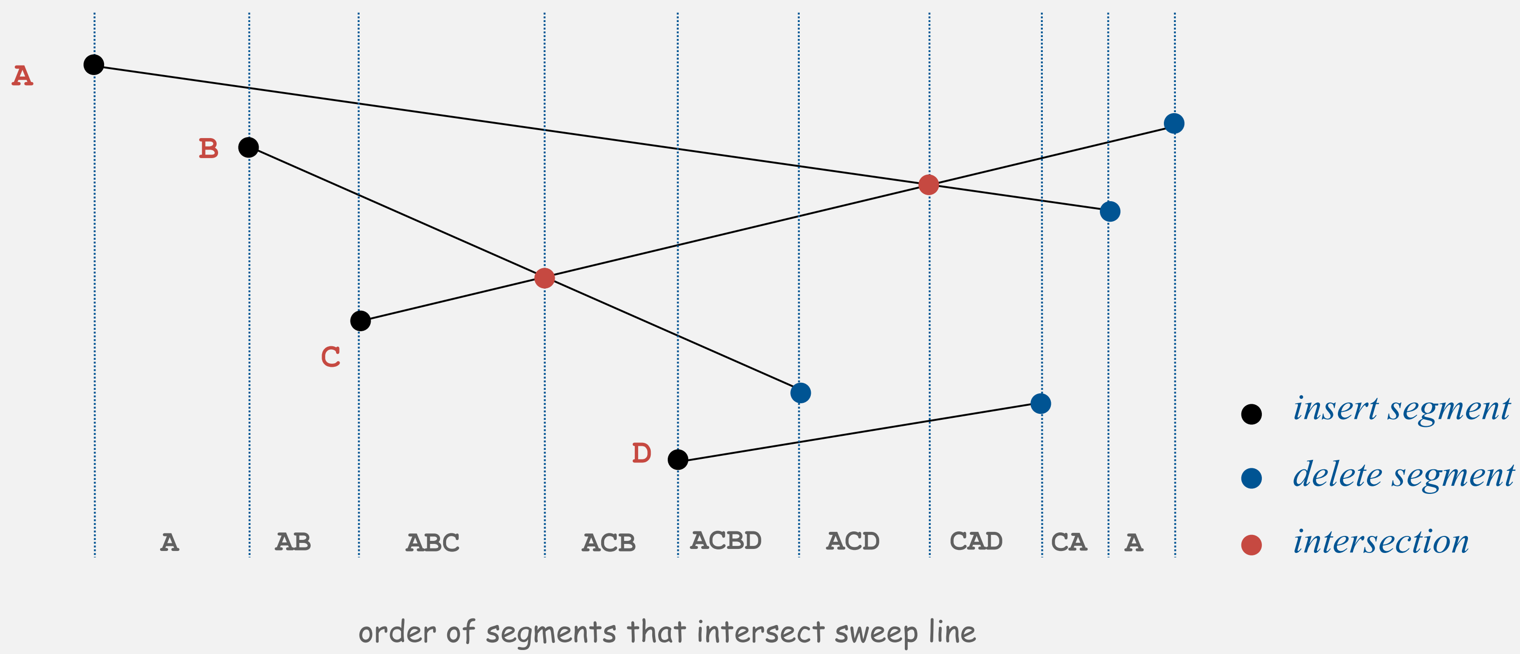
Sweep-line algorithm: simulate the sweep line

```
int INF = Integer.MAX_VALUE;

SET<SegmentHV> set = new SET<SegmentHV>();

while (!pq.isEmpty())
{
   Event event = pq.delMin();
   int sweep = event.time;
   SegmentHV segment = event.segment;

   if (segment.isVertical())
   {
      SegmentHV seg1, seg2;
      seg1 = new SegmentHV(-INF, segment.y1, -INF, segment.y1);
      seg2 = new SegmentHV(+INF, segment.y2, +INF, segment.y2);
      for (SegmentHV seg : set.range(seg1, seg2))
          StdOut.println(segment + " intersects " + seg);
   }

   else if (sweep == segment.x1) set.add(segment);
   else if (sweep == segment.x2) set.remove(segment);
}
```

General line segment intersection search

Extend sweep-line algorithm

- Maintain segments that intersect sweep line ordered by y-coordinate.
- Intersections can only occur between adjacent segments.
- Add/delete line segment $\Rightarrow$ one new pair of adjacent segments.
- Intersection $\Rightarrow$ swap adjacent segments.



Line segment intersection: implementation

Efficient implementation of sweep line algorithm.

- Maintain PQ of important x-coordinates: endpoints and intersections.
- Maintain set of segments intersecting sweep line, sorted by y.
- O(R log N + N log N).

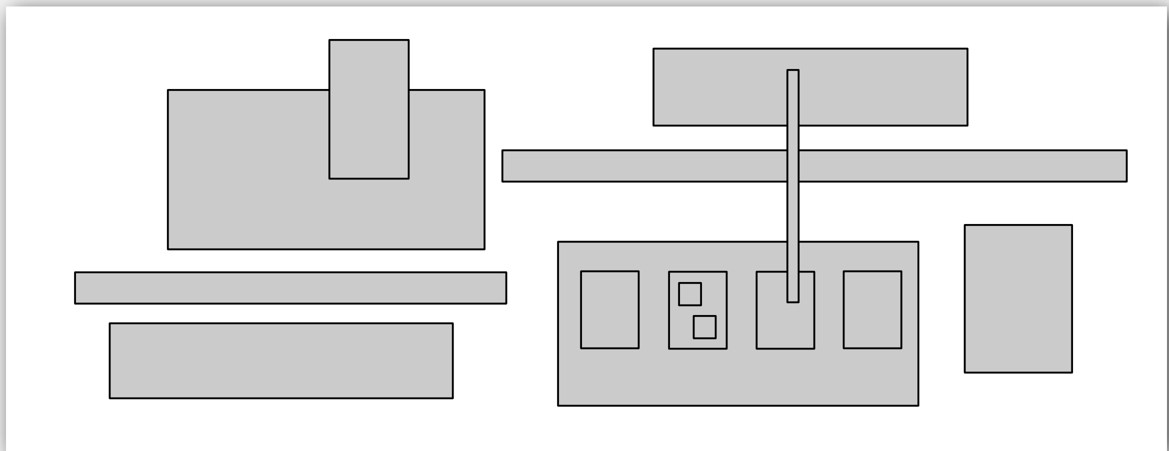
to support "next largest" and "next smallest" queries

Implementation issues.

- Degeneracy.
- Floating point precision.
- Use PQ, not presort (intersection events are unknown ahead of time).

Rectangle intersection search

Goal. Find all intersections among h-v rectangles.



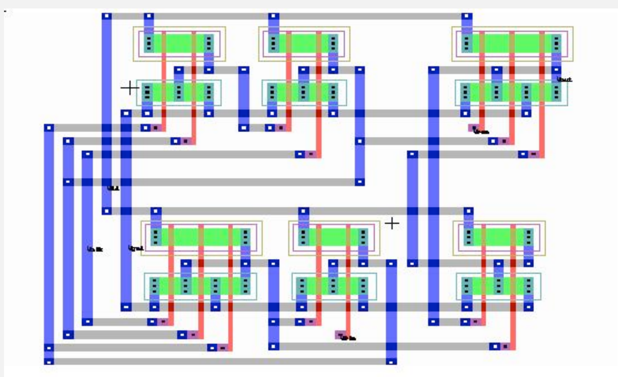
Application. Design-rule checking in VLSI circuits.

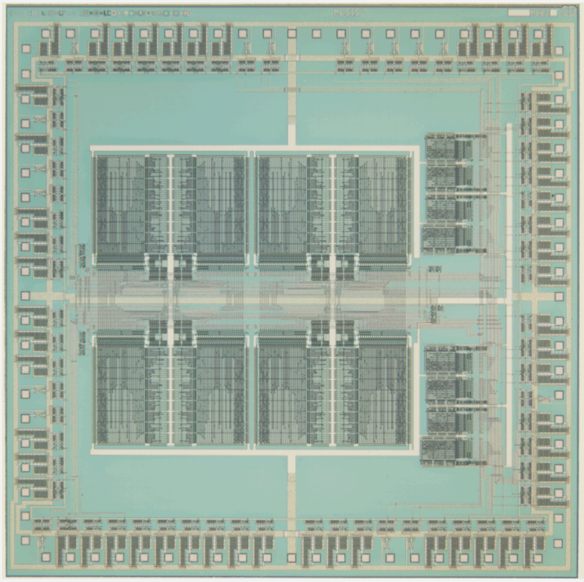
Early 1970s. microprocessor design became a geometric problem.

- Very Large Scale Integration (VLSI).
- Computer-Aided Design (CAD).

Design-rule checking.

- Certain wires cannot intersect.
- Certain spacing needed between different types of wires.
- Debugging = rectangle intersection search.





Algorithms and Moore's law

"Moore's law." Processing power doubles every 18 months.

- 197x: need to check N rectangles.
- 197(x+1.5): need to check 2N rectangles on a 2x-faster computer.

Bootstrapping. We get to use the faster computer for bigger circuits.

But bootstrapping is not enough if using a quadratic algorithm:

- 197x: takes M days.
- 197(x+1.5): takes (4M)/2 = 2M days. (!)

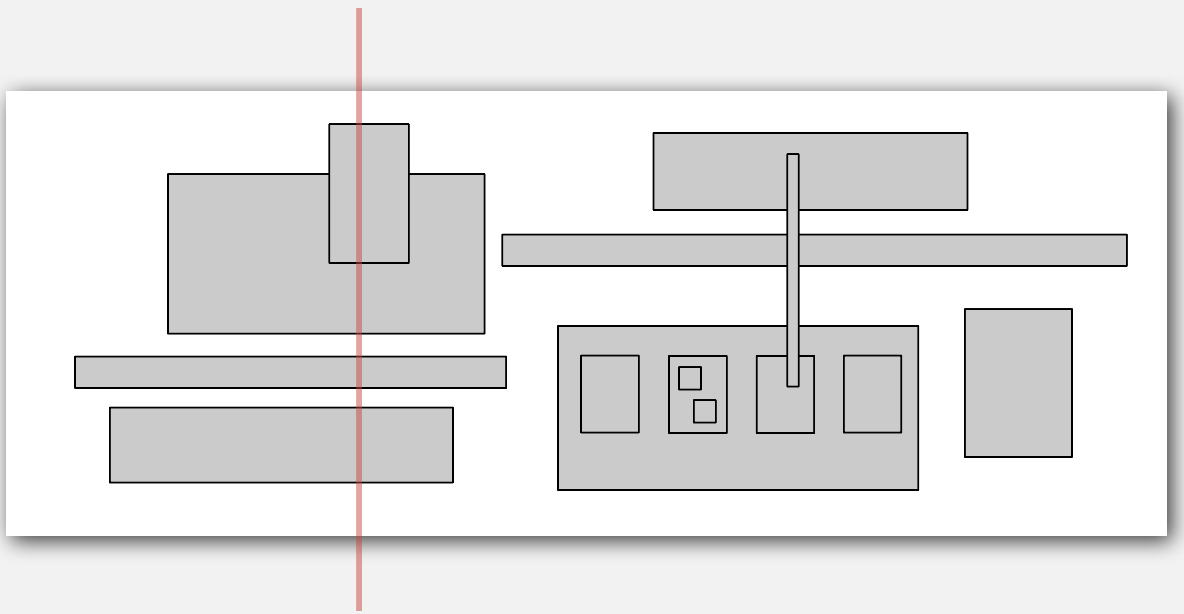
quadratic algorithm

2x-faster computer

Bottom line. Linearithmic CAD algorithm is necessary to sustain Moore's Law.

Sweep vertical line from left to right.

- x-coordinates of rectangles define events.
- Maintain set of y-intervals intersecting sweep line.
- Left endpoint: search set for y-interval; insert y-interval.
- Right endpoint: delete y-interval.

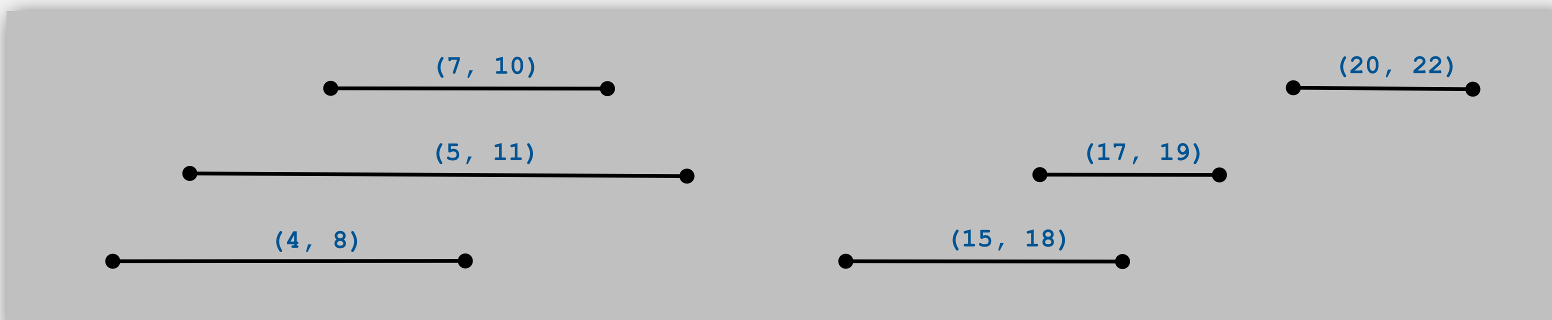


Interval search trees

operation	brute	interval search tree	best in theory
insert interval	1	log N	log N
delete interval	N	log N	log N
find an interval that intersects (lo, hi)	N	log N	log N
find all intervals that intersects (lo, hi)	N	R log N	R + log N

augmented red-black tree

N = # intervals
R = # intersections



Rectangle intersection search: costs summary

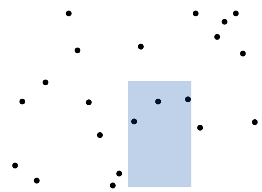
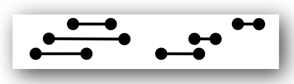
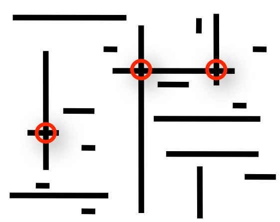
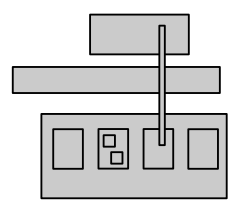
Reduces 2D orthogonal rectangle intersection search to 1D interval search!

Running time of sweep line algorithm.

• Put x-coordinates on a PQ (or sort).	O(N log N)	N = # rectangles
• Insert y-interval into ST.	O(N log N)	R = # intersections
• Delete y-interval from ST.	O(N log N)	
• Interval search.	O(R + N log N)	

Efficiency relies on judicious use of data structures.

Geometric search summary: algorithms of the day

1D range search		BST
kD range search		kD tree
1D interval intersection search		interval search tree
2D orthogonal line intersection search		sweep line reduces to 1D range search
2D orthogonal rectangle intersection search		sweep line reduces to 1D interval intersection search

7.5 Reductions

- designing algorithms
- establishing lower bounds
- intractability

Bird's-eye view

Desiderata. Classify problems according to computational requirements.

complexity	order of growth	examples
linear	N	min, max, median, Burrows-Wheeler transform, ...
linearithmic	$N \log N$	sorting, convex hull, closest pair, farthest pair, ...
quadratic	N^2	???
	...	
exponential	c^N	???

Frustrating news. Huge number of problems have defied classification.

Bird's-eye view

Desiderata. Classify problems according to computational requirements.

Desiderata'.

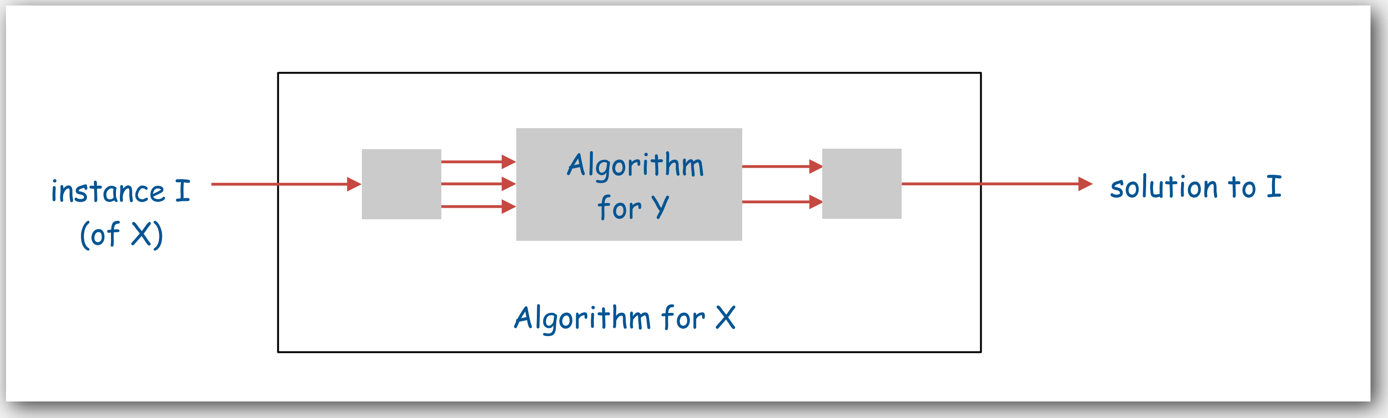
Suppose we could (couldn't) solve problem X efficiently.

What else could (couldn't) we solve efficiently?

" Give me a lever long enough and a fulcrum on which to place it, and I shall move the world. " — *Archimedes*

Reduction

Def. Problem X reduces to problem Y if you can use an algorithm that solves Y to help solve X.



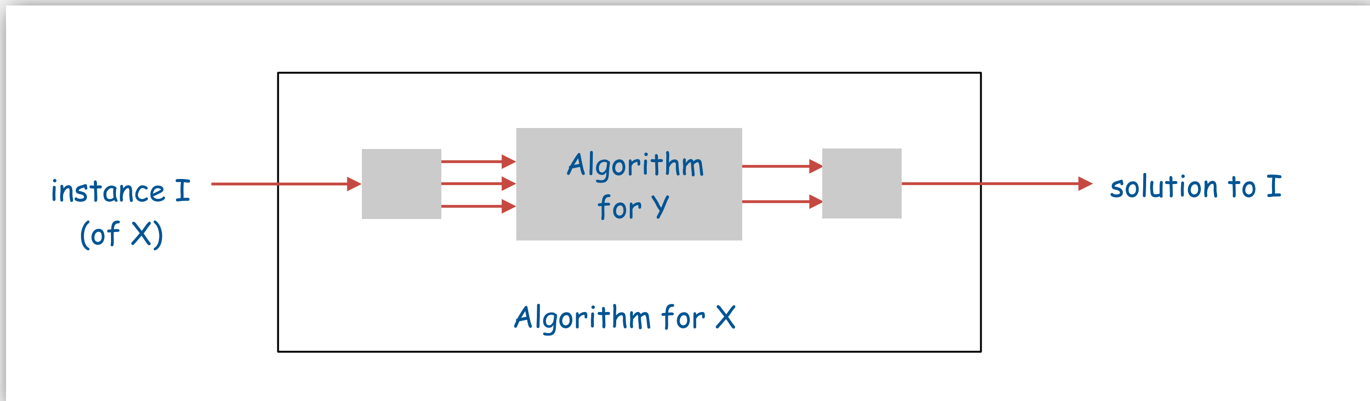
Cost of solving X = total cost of solving Y + cost of reduction.

perhaps many calls to Y on problems of different sizes

preprocessing and postprocessing

Reduction

Def. Problem X reduces to problem Y if you can use an algorithm that solves Y to help solve X.



Ex 1. [element distinctness reduces to sorting]

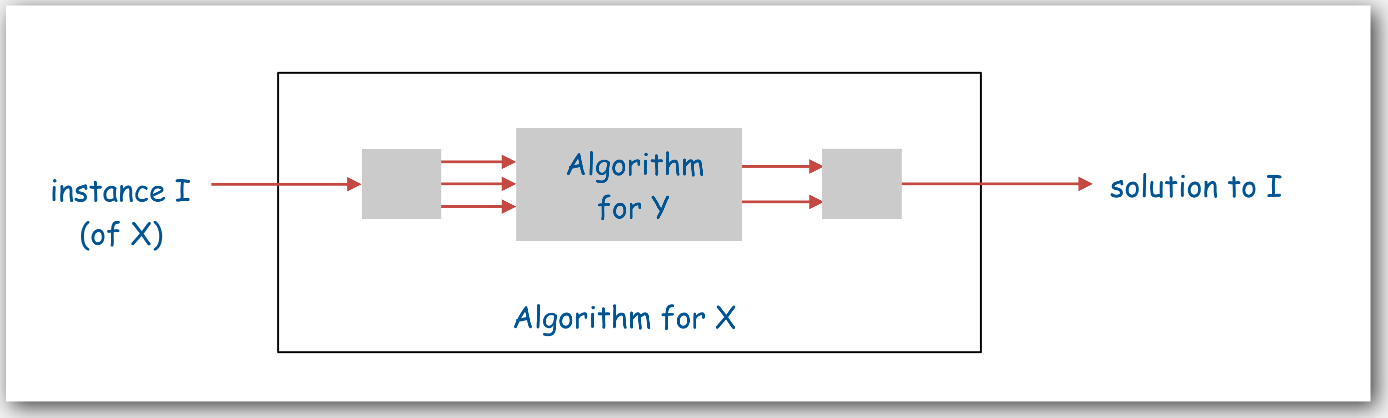
To solve element distinctness on N integers:

- Sort N integers.
- Check adjacent pairs for equality.

Cost of solving element distinctness. $\underbrace{N \log N}_{\text{cost of sorting}} + \underbrace{N}_{\text{cost of reduction}}$

Reduction

Def. Problem X reduces to problem Y if you can use an algorithm that solves Y to help solve X.



Ex 2. [3-collinear reduces to sorting]

To solve 3-collinear instance on N points in the plane:

- For each point, sort other points by polar angle.
  - check adjacent triples for collinearity

cost of sorting

cost of reduction

Cost of solving 3-collinear. $N^2 \log N + N^2$.

- designing algorithms
- establishing lower bounds
- intractability

Reduction: design algorithms

Def. Problem X reduces to problem Y if you can use an algorithm that solves Y to help solve X.

Design algorithm. Given algorithm for Y, can also solve X.

Ex.

- Element distinctness reduces to sorting.
- 3-collinear reduces to sorting.
- PERT reduces to topological sort. [see digraph lecture]
- h-v line intersection reduces to 1D range searching. [see geometry lecture]
- Burrows-Wheeler transform reduces to suffix sort. [see assignment 8]

Mentality. Since I know how to solve Y, can I use that algorithm to solve X?

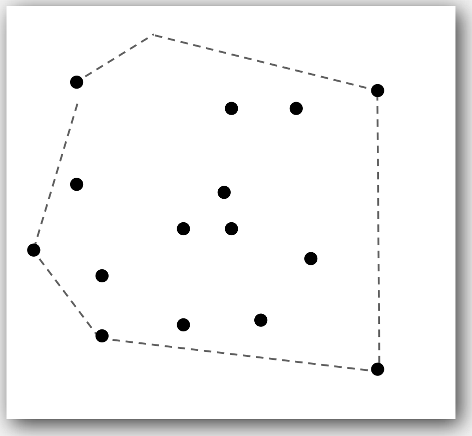
↑

programmer's version: I have code for Y. Can I use it for X?

Convex hull reduces to sorting

Sorting. Given N distinct integers, rearrange them in ascending order.

Convex hull. Given N points in the plane, identify the extreme points of the convex hull (in counter-clockwise order).



convex hull

1251432
2861534
3988818
4190745
13546464
89885444
43434213
34435312

sorting

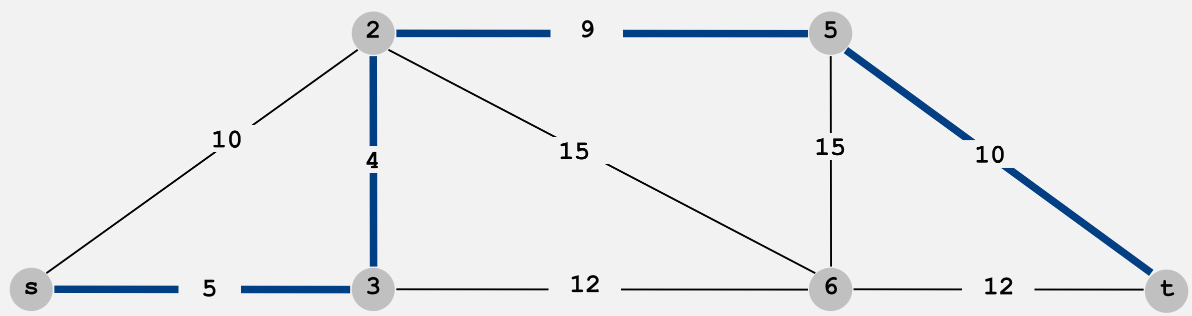
Proposition. Convex hull reduces to sorting.

Pf. Graham scan algorithm.

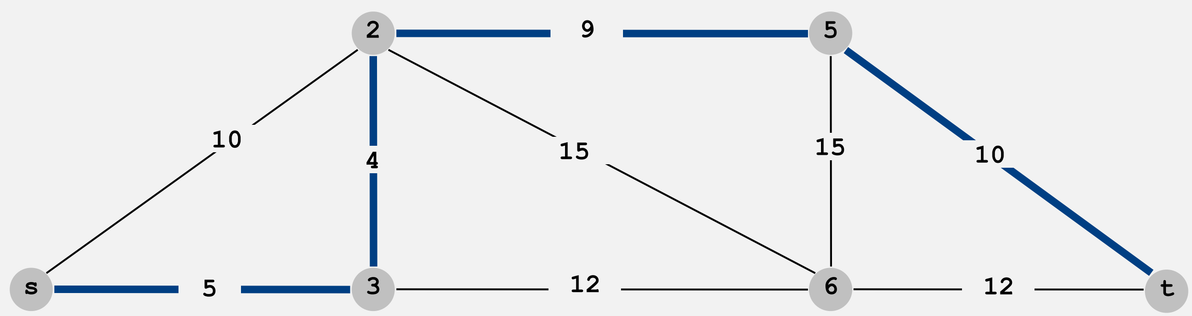
Cost of convex hull. $\underbrace{N \log N}_{\text{cost of sorting}} + \underbrace{N}_{\text{cost of reduction}}$.

Shortest path on graphs and digraphs

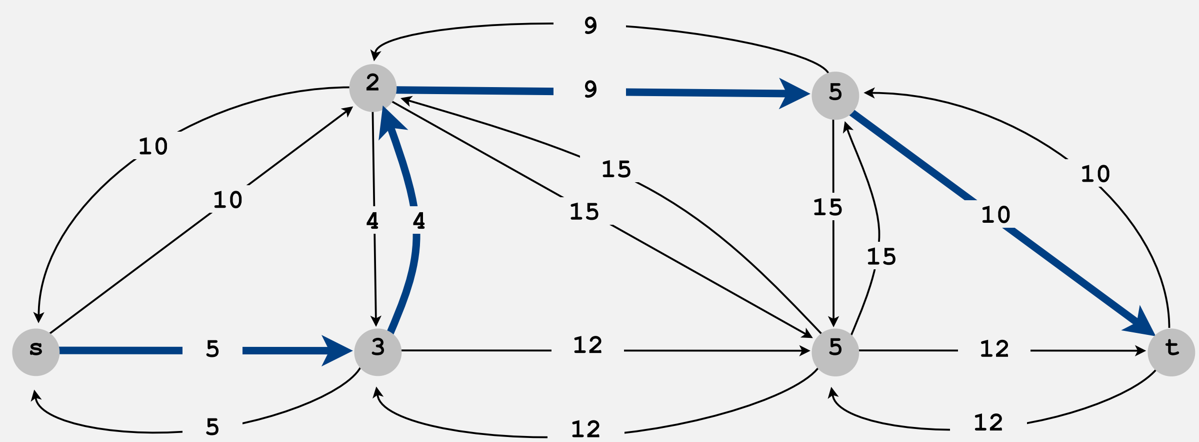
Proposition. Undirected shortest path (with nonnegative weights) reduces to directed shortest path.



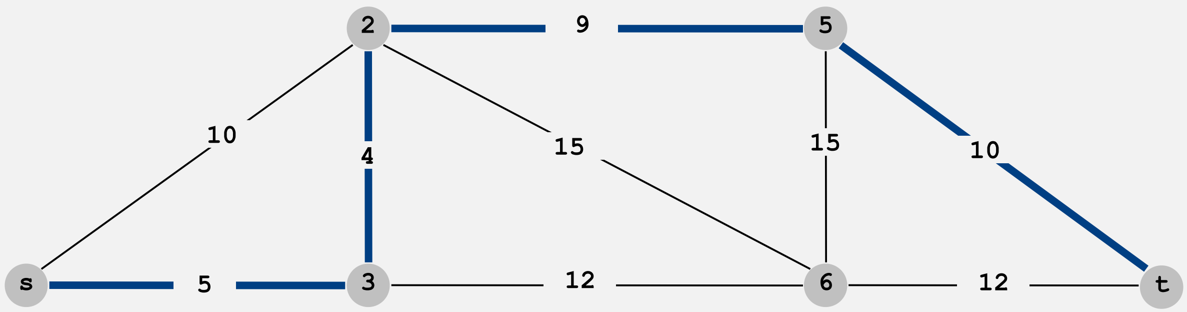
Proposition. Undirected shortest path (with nonnegative weights) reduces to directed shortest path.



Pf. Replace each undirected edge by two directed edges.



Proposition. Undirected shortest path (with nonnegative weights) reduces to directed shortest path.



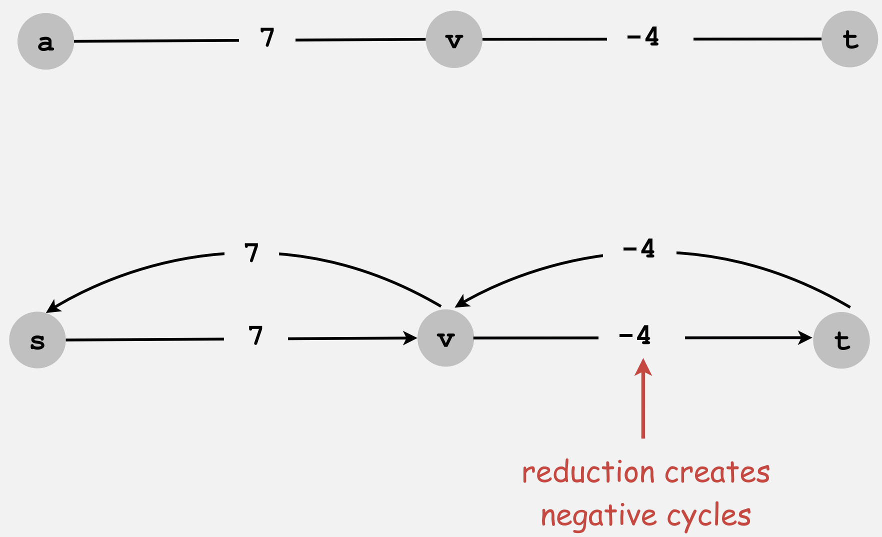
Cost of undirected shortest path. $E \log E + E$.

cost of shortest path in digraph

cost of reduction

Shortest path with negative weights

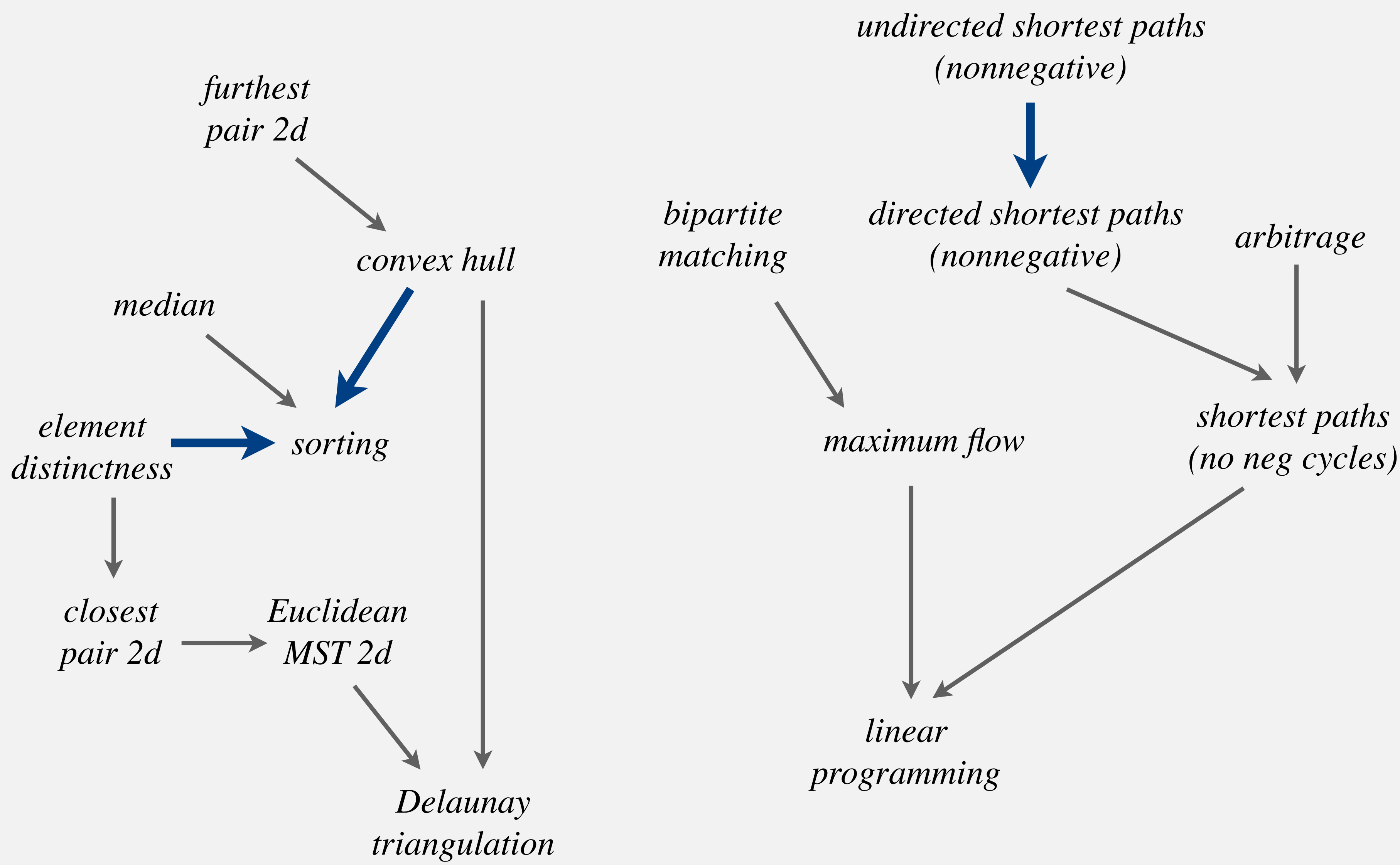
Caveat. Reduction is invalid in networks with negative weights (even if no negative cycles).



Remark. Can still solve shortest path problem in undirected graphs (if no negative cycles), but need more sophisticated techniques.

reduces to weighted
non-bipartite matching (!)

Some reductions involving familiar problems



- designing algorithms
- **linear programming**
- establishing lower bounds
- establishing intractability
- classifying problems

What is it? [see ORF 307]

- Quintessential tool for optimal allocation of scarce resources
- Powerful and general problem-solving method

Why is it significant?

- Widely applicable.
- Dominates world of industry.
- Fast commercial solvers available: CPLEX, OSL.
- Powerful modeling languages available: AMPL, GAMS.
- Ranked among most important scientific advances of 20th century.

Ex: Delta claims that LP saves $100 million per year.

Present context. Many important problems reduce to LP.

Applications

Agriculture. Diet problem.

Computer science. Compiler register allocation, data mining.

Electrical engineering. VLSI design, optimal clocking.

Energy. Blending petroleum products.

Economics. Equilibrium theory, two-person zero-sum games.

Environment. Water quality management.

Finance. Portfolio optimization.

Logistics. Supply-chain management.

Management. Hotel yield management.

Marketing. Direct mail advertising.

Manufacturing. Production line balancing, cutting stock.

Medicine. Radioactive seed placement in cancer treatment.

Operations research. Airline crew assignment, vehicle routing.

Physics. Ground states of 3-D Ising spin glasses.

Plasma physics. Optimal stellarator design.

Telecommunication. Network design, Internet routing.

Sports. Scheduling ACC basketball, handicapping horse races.

Model problem as maximizing an objective function subject to constraints.

Input: real numbers a_{ij}, c_j, and b_i.

Output: real numbers x_j.

n variables

maximize $c_1 x_1 + c_2 x_2 + \ldots + c_n x_n$

subject to the constraints (m equations)

$$
\begin{aligned}
a_{11} x_1 + a_{12} x_2 + \ldots + a_{1n} x_n &\le b_1 \\
a_{21} x_1 + a_{22} x_2 + \ldots + a_{2n} x_n &\le b_2 \\
&\ldots \\
a_{m1} x_1 + a_{m2} x_2 + \ldots + a_{mn} x_n &\le b_m \\
x_1, x_2, \ldots, x_n &\ge 0
\end{aligned}
$$

matrix version

maximize $c^T x$

subject to the constraints

$$
\begin{aligned}
A x &\le b \\
x &\ge 0
\end{aligned}
$$

Solutions. [see ORF 307]

- Simplex algorithm has been used for decades to solve practical LP instances.
- Newer algorithms guarantee fast solution.

Linear programming

"Linear programming"

- Process of formulating an LP model for a problem.
- Solution to LP for a specific problem gives solution to the problem.
- Equivalent to "reducing the problem to LP."

1. Identify variables.
2. Define constraints (inequalities and equations).
3. Define objective function.

Examples:

- Shortest paths
- Maximum flow.
- Bipartite matching.

  . . .

- [a very long list]

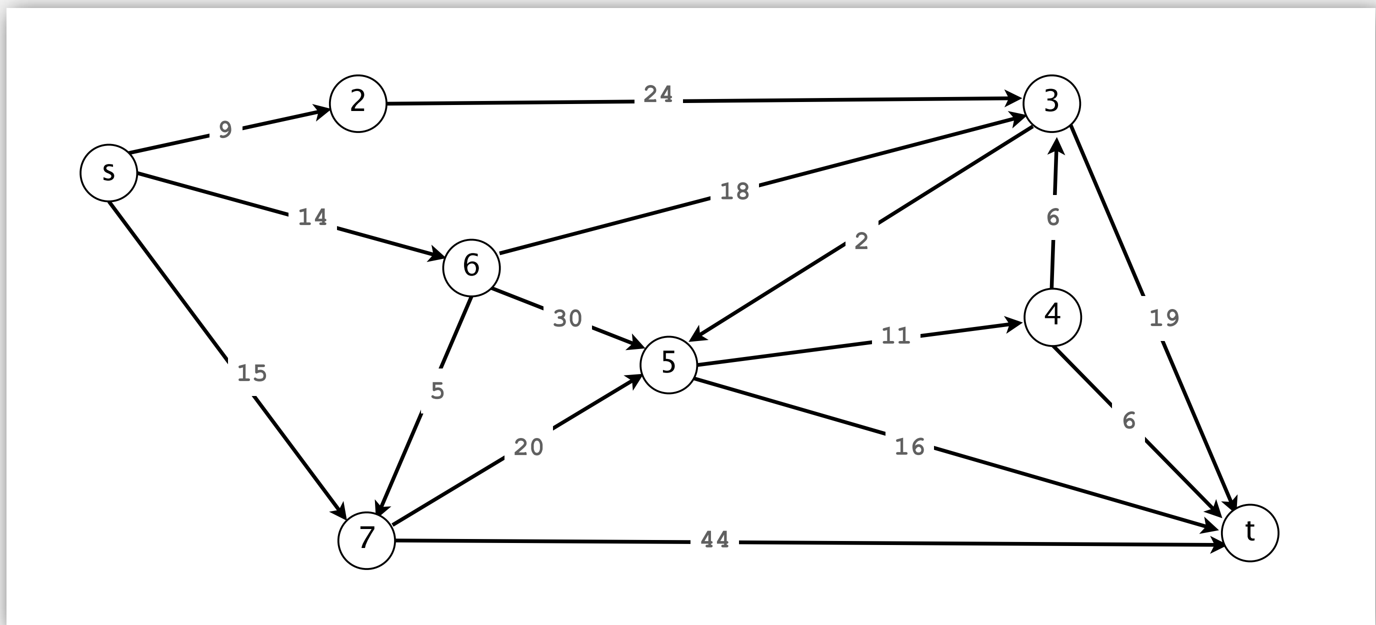
stay tuned (next)

Single-source shortest-paths problem (revisited)

Given. Weighted digraph, single source s.

Distance from s to v. Length of the shortest path from s to v .

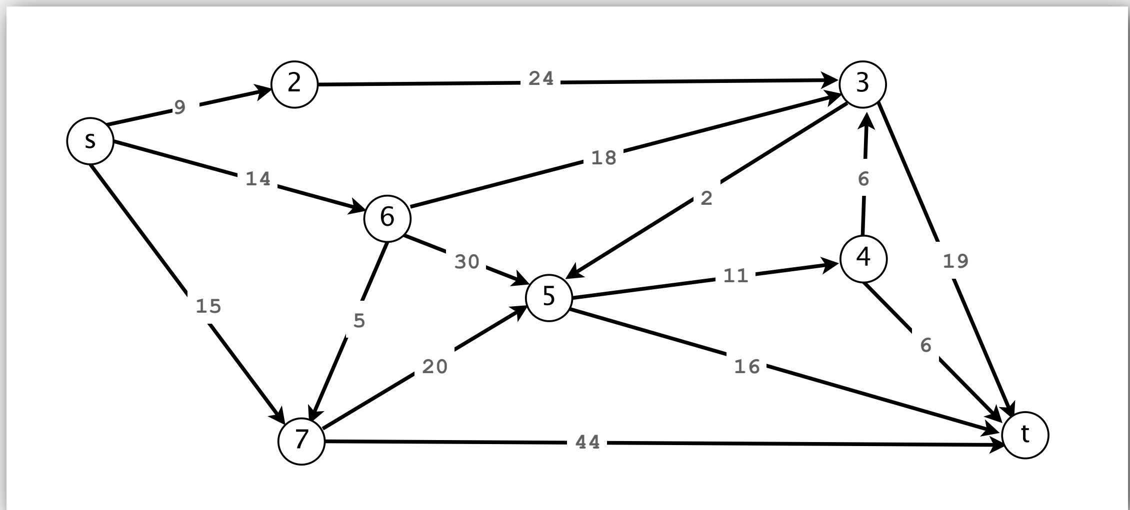
Goal. Find distance (and shortest path) from s to every other vertex.



Single-source shortest-paths problem reduces to LP

LP formulation.

- One variable per vertex, one inequality per edge.
- Interpretation: x_i = length of shortest path from s to i.



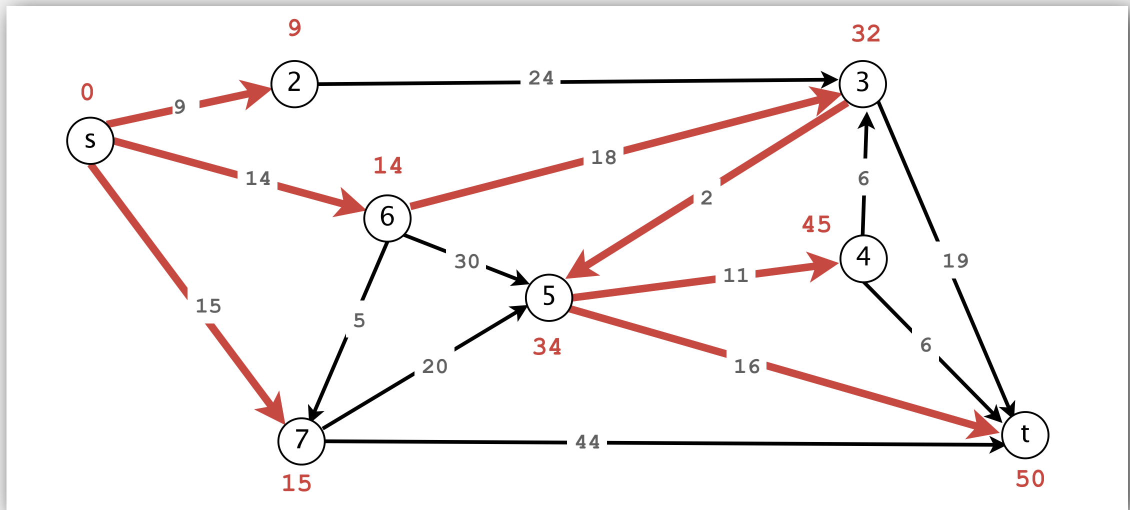
maximize	x_t
subject to the constraints	$x_s + 9 \geq x_2$
	$x_s + 14 \geq x_6$
	$x_s + 15 \geq x_7$
	$x_2 + 24 \geq x_3$
	$x_3 + 2 \geq x_5$
	$x_3 + 19 \geq x_t$
	$x_4 + 6 \geq x_3$
	$x_4 + 6 \geq x_t$
	$x_5 + 11 \geq x_4$
	$x_5 + 16 \geq x_t$
	$x_6 + 18 \geq x_3$
	$x_6 + 30 \geq x_5$
	$x_6 + 5 \geq x_7$
	$x_7 + 20 \geq x_5$
	$x_7 + 44 \geq x_t$
	$x_s = 0$

Single-source shortest-paths problem reduces to LP

LP formulation.

- One variable per vertex, one inequality per edge.
- Interpretation: x_i = length of shortest path from s to i.

maximize	x_t
subject to the constraints	$x_s + 9 \geq x_2$
	$x_s + 14 \geq x_6$
	$x_s + 15 \geq x_7$
	$x_2 + 24 \geq x_3$
	$x_3 + 2 \geq x_5$
	$x_3 + 19 \geq x_t$
	$x_4 + 6 \geq x_3$
	$x_4 + 6 \geq x_t$
	$x_5 + 11 \geq x_4$
	$x_5 + 16 \geq x_t$
	$x_6 + 18 \geq x_3$
	$x_6 + 30 \geq x_5$
	$x_6 + 5 \geq x_7$
	$x_7 + 20 \geq x_5$
	$x_7 + 44 \geq x_t$
	$x_s = 0$

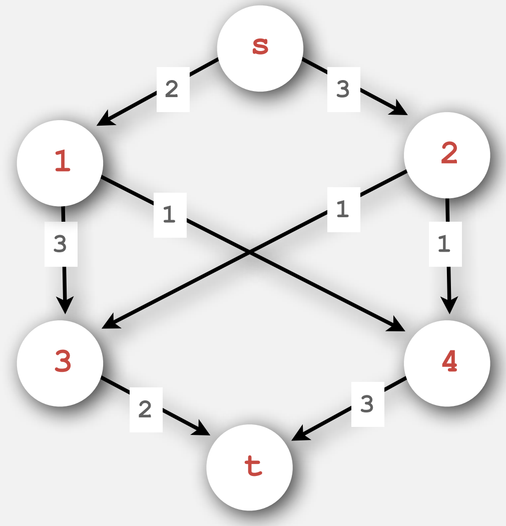


$x_s = 0$	$x_5 = 34$
$x_2 = 9$	$x_6 = 14$
$x_3 = 32$	$x_7 = 15$
$x_4 = 45$	$x_t = 50$

solution

Maxflow problem

Given: Weighted digraph, source `s`, destination `t`.

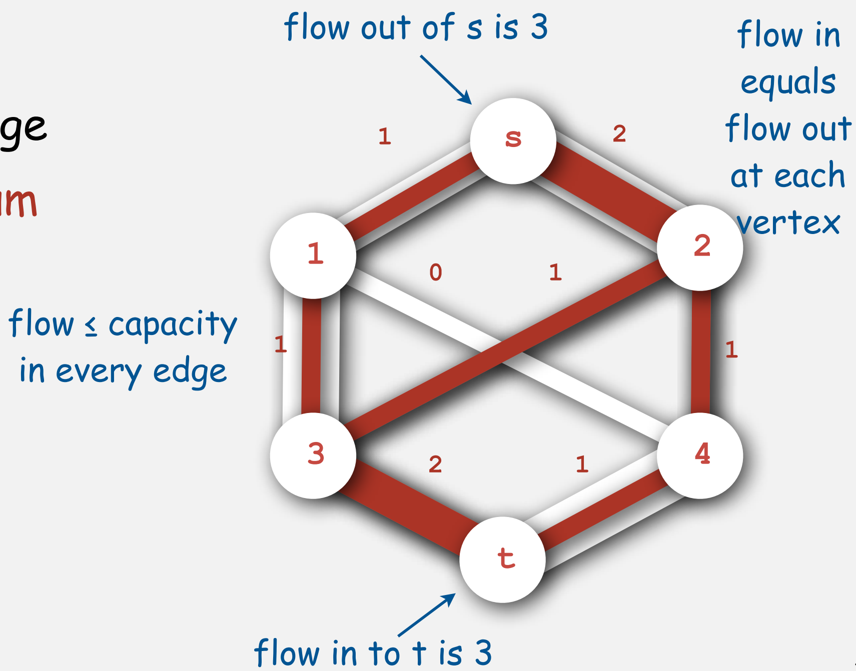


Interpret edge weights as capacities

- Models material flowing through network
- Ex: oil flowing through pipes
- Ex: goods in trucks on roads
- [many other examples]

Flow: A different set of edge weights

- flow does not exceed capacity in any edge
- flow at every vertex satisfies equilibrium
  [flow in equals flow out]



Goal: Find maximum flow from s to t.

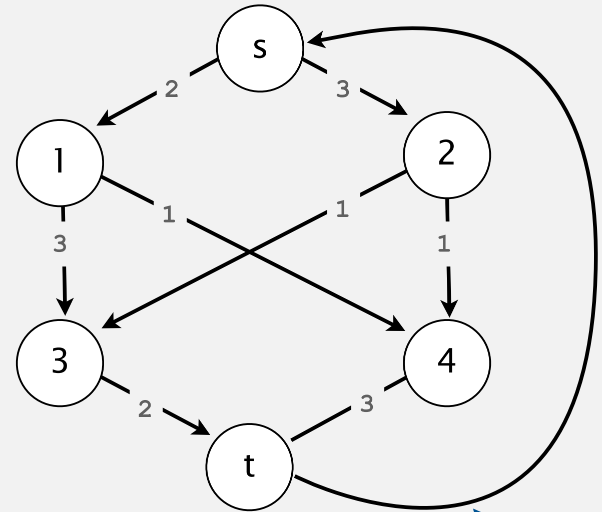
Maximum flow reduces to LP

One variable per edge.
One inequality per edge, one equality per vertex.

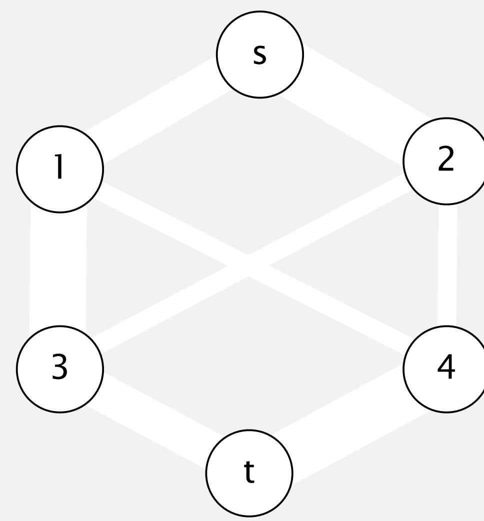
maximize	$x_{3t} + x_{4t}$
subject to the constraints	$x_{s1} \le 2$
	$x_{s2} \le 3$
	$x_{13} \le 3$
	$x_{14} \le 1$
	$x_{23} \le 1$
	$x_{24} \le 1$
	$x_{3t} \le 2$
	$x_{4t} \le 3$
	$x_{s1} = x_{13} + x_{14}$
equilibrium constraints	$x_{s2} = x_{23} + x_{24}$
	$x_{13} + x_{23} = x_{3t}$
	$x_{14} + x_{24} = x_{4t}$
	all $x_{ij} \ge 0$

capacity constraints

interpretation:
x_{ij} = flow in edge i-j



add dummy edge from t to s



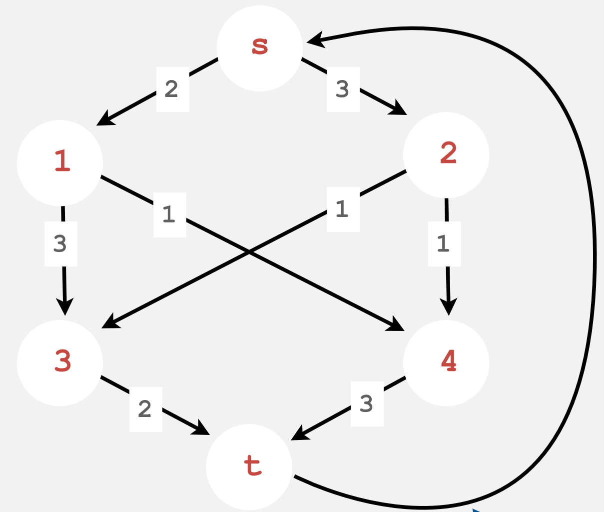
Maxflow problem reduces to LP

One variable per edge.
One inequality per edge, one equality per vertex.

maximize	$x_{3t} + x_{4t}$
subject to the constraints	$x_{s1} \le 2$
	$x_{s2} \le 3$
	$x_{13} \le 3$
	$x_{14} \le 1$
	$x_{23} \le 1$
	$x_{24} \le 1$
	$x_{3t} \le 2$
	$x_{4t} \le 3$
equilibrium constraints	$x_{s1} = x_{13} + x_{14}$
	$x_{s2} = x_{23} + x_{24}$
	$x_{13} + x_{23} = x_{3t}$
	$x_{14} + x_{24} = x_{4t}$
	all $x_{ij} \ge 0$

capacity constraints

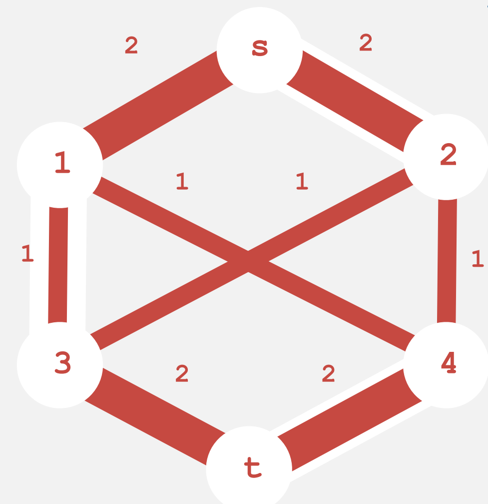
interpretation:
x_{ij} = flow in edge i-j



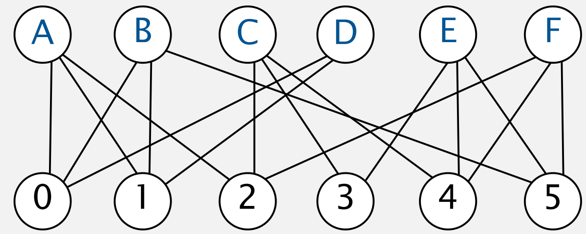
add dummy edge from t to s

solution

$x_{s1} = 2$
$x_{s2} = 2$
$x_{13} = 1$
$x_{14} = 1$
$x_{23} = 1$
$x_{24} = 1$
$x_{3t} = 2$
$x_{4t} = 2$



Maximum cardinality bipartite matching problem



Bipartite graph. Two sets of vertices; edges connect vertices in one set to the other.

Matching. Set of edges with no vertex appearing twice.

Goal. Find a maximum cardinality matching.

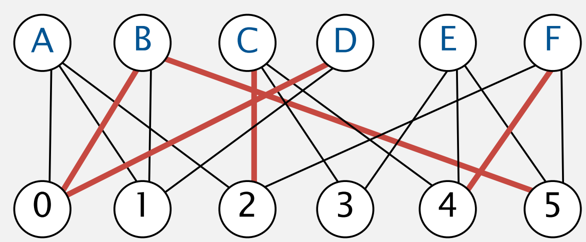
Alice
  Adobe, Apple, Google
Bob
  Adobe, Apple, Yahoo
Carol
  Google, IBM, Sun
Dave
  Adobe, Apple
Eliza
  IBM, Sun, Yahoo
Frank
  Google, Sun, Yahoo

Adobe
  Alice, Bob, Dave
Apple
  Alice, Bob, Dave
Google
  Alice, Carol, Frank
IBM
  Carol, Eliza
Sun
  Carol, Eliza, Frank
Yahoo
  Bob, Eliza, Frank

job offers

Interpretation. Mutual preference constraints.

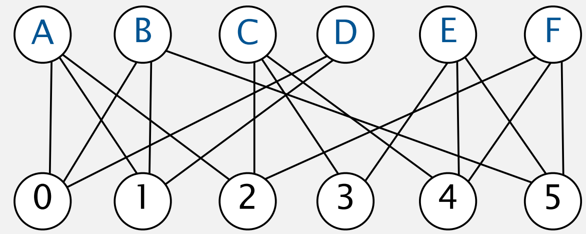
- Ex: people to jobs.
- Ex: Medical students to residence positions.
- Ex: students to writing seminars.
- [many other examples]



Maximum cardinality bipartite matching reduces to LP

LP formulation.

- One variable per edge, one equality per vertex.
- Interpretation: an edge is in matching iff $x_i = 1$.



maximize	$x_{A0} + x_{A1} + x_{A2} + x_{B0} + x_{B1} + x_{B5} + x_{C2} + x_{C3} + x_{C4} + x_{D0} + x_{D1} + x_{E3} + x_{E4} + x_{E5} + x_{F2} + x_{F4} + x_{F5}$	
subject to the constraints	$x_{A0} + x_{A1} + x_{A2} = 1$	$x_{A0} + x_{B0} + x_{D0} = 1$
	$x_{B0} + x_{B1} + x_{B5} = 1$	$x_{A1} + x_{B1} + x_{D1} = 1$
	$x_{C2} + x_{C3} + x_{C4} = 1$	$x_{A2} + x_{C2} + x_{F2} = 1$
	$x_{D0} + x_{D1} = 1$	$x_{C3} + x_{E3} = 1$
	$x_{E3} + x_{E4} + x_{E5} = 1$	$x_{C4} + x_{E4} + x_{F4} = 1$
	$x_{F2} + x_{F4} + x_{F5} = 1$	$x_{B5} + x_{E5} + x_{F5} = 1$
	all $x_{ij} \geq 0$	

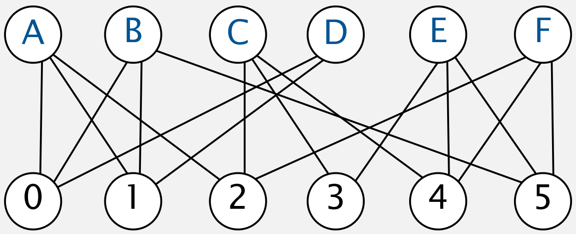
constraints on top vertices (left) and bottom vertices (right)

crucial point: not always so lucky!

Theorem. [Birkhoff 1946, von Neumann 1953]
All extreme points of the above polyhedron have integer (0 or 1) coordinates.

Corollary. Can solve bipartite matching problem by solving LP.

Maximum cardinality bipartite matching reduces to LP



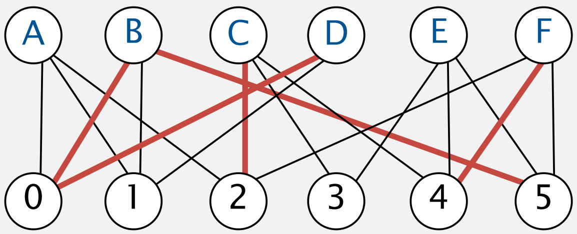
LP formulation.

- One variable per edge, one equality per vertex.
- Interpretation: an edge is in matching iff $x_i = 1$.

maximize	$x_{A0} + x_{A1} + x_{A2} + x_{B0} + x_{B1} + x_{B5} + x_{C2} + x_{C3} + x_{C4} + x_{D0} + x_{D1} + x_{E3} + x_{E4} + x_{E5} + x_{F2} + x_{F4} + x_{F5}$	
subject to the constraints	$x_{A0} + x_{A1} + x_{A2} = 1$	$x_{A0} + x_{B0} + x_{D0} = 1$
	$x_{B0} + x_{B1} + x_{B5} = 1$	$x_{A1} + x_{B1} + x_{D1} = 1$
	$x_{C2} + x_{C3} + x_{C4} = 1$	$x_{A2} + x_{C2} + x_{F2} = 1$
	$x_{D0} + x_{D1} = 1$	$x_{C3} + x_{E3} = 1$
	$x_{E3} + x_{E4} + x_{E5} = 1$	$x_{C4} + x_{E4} + x_{F4} = 1$
	$x_{F2} + x_{F4} + x_{F5} = 1$	$x_{B5} + x_{E5} + x_{F5} = 1$
	all $x_{ij} \geq 0$	

solution

$x_{A1} = 1$
$x_{B5} = 1$
$x_{C2} = 1$
$x_{D0} = 1$
$x_{E3} = 1$
$x_{F4} = 1$
all other $x_{ij} = 0$



Linear programming perspective

Got an optimization problem?

Ex. Shortest paths, maximum flow, matching, ….

Approach 1. Use a specialized algorithm to solve it.

- Algorithms in Java.
- Vast literature on complexity.
- Performance on real problems not always well-understood.

Approach 2. Reduce to a LP model; use a commercial solver.

- A direct mathematical representation of the problem often works.
- Immediate solution to the problem at hand is often available.
- Might miss faster specialized solution, but might not care.

Got an LP solver? Learn to use it!

```
% ampl
AMPL Version 20010215 (SunOS 5.7)
ampl: model maxflow.mod;
ampl: data maxflow.dat;
ampl: solve;
CPLEX 7.1.0: optimal solution;
objective 4;
```

- designing algorithms
- establishing lower bounds
- intractability

Bird's-eye view

Goal. Prove that a problem requires a certain number of steps.

Ex. $\Omega(N \log N)$ lower bound for sorting.

```
1251432
2861534
3988818
4190745
13546464
89885444
43434213
```

argument must apply to all conceivable algorithms

Bad news. Very difficult to establish lower bounds from scratch.

Good news. Can spread $\Omega(N \log N)$ lower bound to Y by reducing sorting to Y.

assuming cost of reduction is not too high

Linear-time reductions

Def. Problem X **linear-time reduces** to problem Y if X can be solved with:

- Linear number of standard computational steps.
- Constant number of calls to Y.

Ex. Almost all of the reductions we've seen so far. [Which one wasn't?]

Establish lower bound:

- If X takes $\Omega(N \log N)$ steps, then so does Y.
- If X takes $\Omega(N^2)$ steps, then so does Y.

Mentality.

- If I could easily solve Y, then I could easily solve X.
- I can't easily solve X.
- Therefore, I can't easily solve Y.

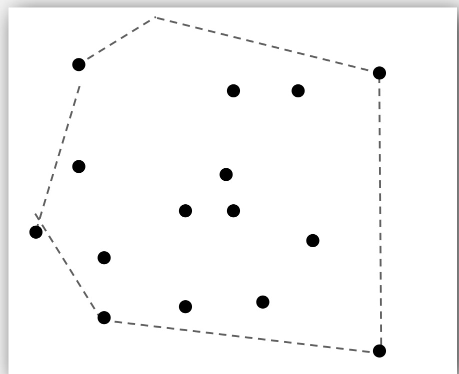
Proposition. In quadratic decision tree model, any algorithm for sorting N integers requires $\Omega(N \log N)$ steps.

allows quadratic tests of the form:
$x_i < x_j$ or $(x_j - x_i)(x_k - x_i) - (x_j)(x_j - x_i) < 0$

Proposition. Sorting linear-time reduces to convex hull.
Pf. [see next slide]

```
1251432
2861534
3988818
4190745
13546464
89885444
43434213
```

sorting



convex hull

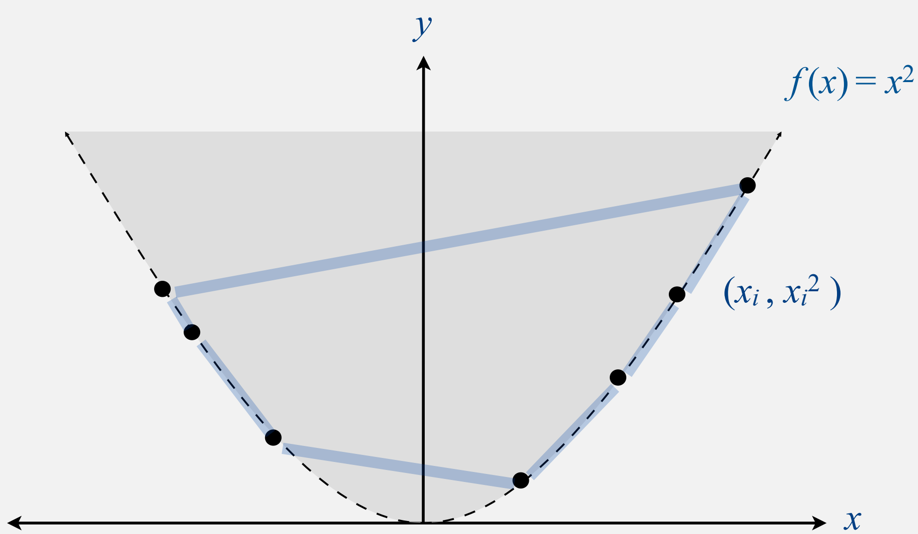
a quadratic test

Implication. Any ccw-based convex hull algorithm requires $\Omega(N \log N)$ ccw's.

Sorting linear-time reduces to convex hull

Proposition. Sorting linear-time reduces to convex hull.

- Sorting instance: $x_1, x_2, \ldots, x_N$.
- Convex hull instance: $(x_1, x_1^2), (x_2, x_2^2), \ldots, (x_N, x_N^2)$.



Pf.

- Region $\{x : x^2 \geq x\}$ is convex $\Rightarrow$ all points are on hull.
- Starting at point with most negative x, counter-clockwise order of hull points yields integers in ascending order.

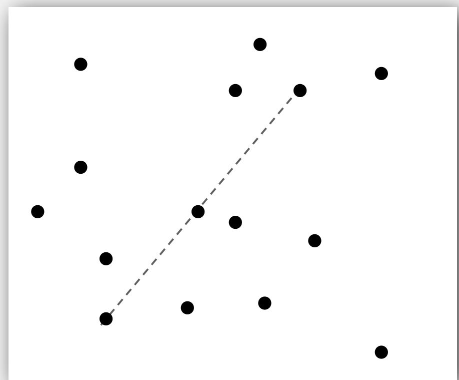
Lower bound for 3-COLLINEAR

3-SUM. Given N distinct integers, are there three that sum to 0?

3-COLLINEAR. Given N distinct points in the plane, are there 3 that all lie on the same line? ← recall Assignment 3

```
1251432
-2861534
3988818
-4190745
13546464
89885444
-43434213
```

3-sum



3-collinear

Lower bound for 3-COLLINEAR

3-SUM. Given N distinct integers, are there three that sum to 0?

3-COLLINEAR. Given N distinct points in the plane, are there 3 that all lie on the same line?

Proposition. 3-SUM linear-time reduces to 3-COLLINEAR.

Pf. [see next 2 slide]

Conjecture. Any algorithm for 3-SUM requires $\Omega(N^2)$ steps.

Implication. No sub-quadratic algorithm for 3-COLLINEAR likely.

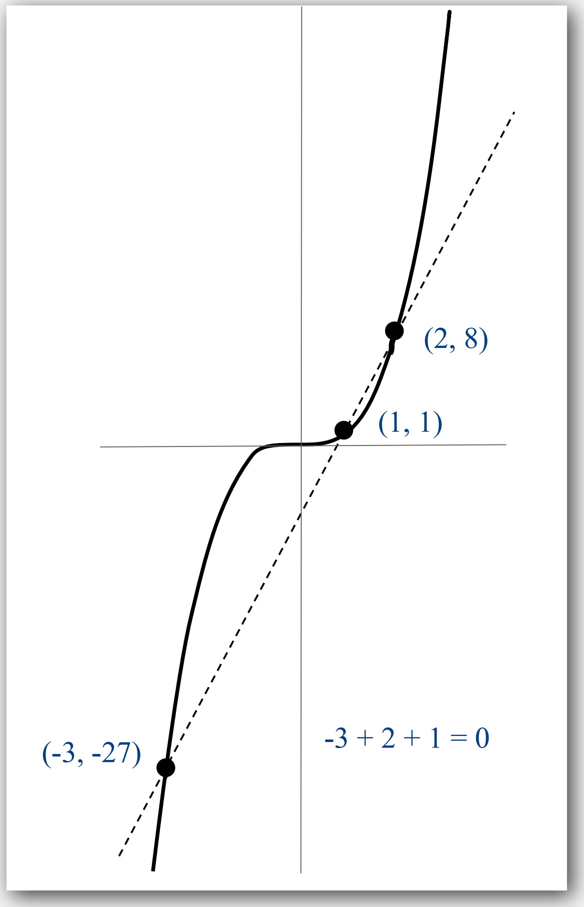
your $N^2 \log N$ algorithm was pretty good

Proposition. 3-SUM linear-time reduces to 3-COLLINEAR.

- 3-SUM instance: $x_1, x_2, \ldots, x_N$.
- 3-COLLINEAR instance: $(x_1, x_1^3), (x_2, x_2^3), \ldots, (x_N, x_N^3)$.

Lemma. If $a, b,$ and c are distinct, then $a + b + c = 0$ if and only if $(a, a^3), (b, b^3)$, and (c, c^3) are collinear.

$f(x) = x^3$



3-SUM linear-time reduces to 3-COLLINEAR

Proposition. 3-SUM linear-time reduces to 3-COLLINEAR.

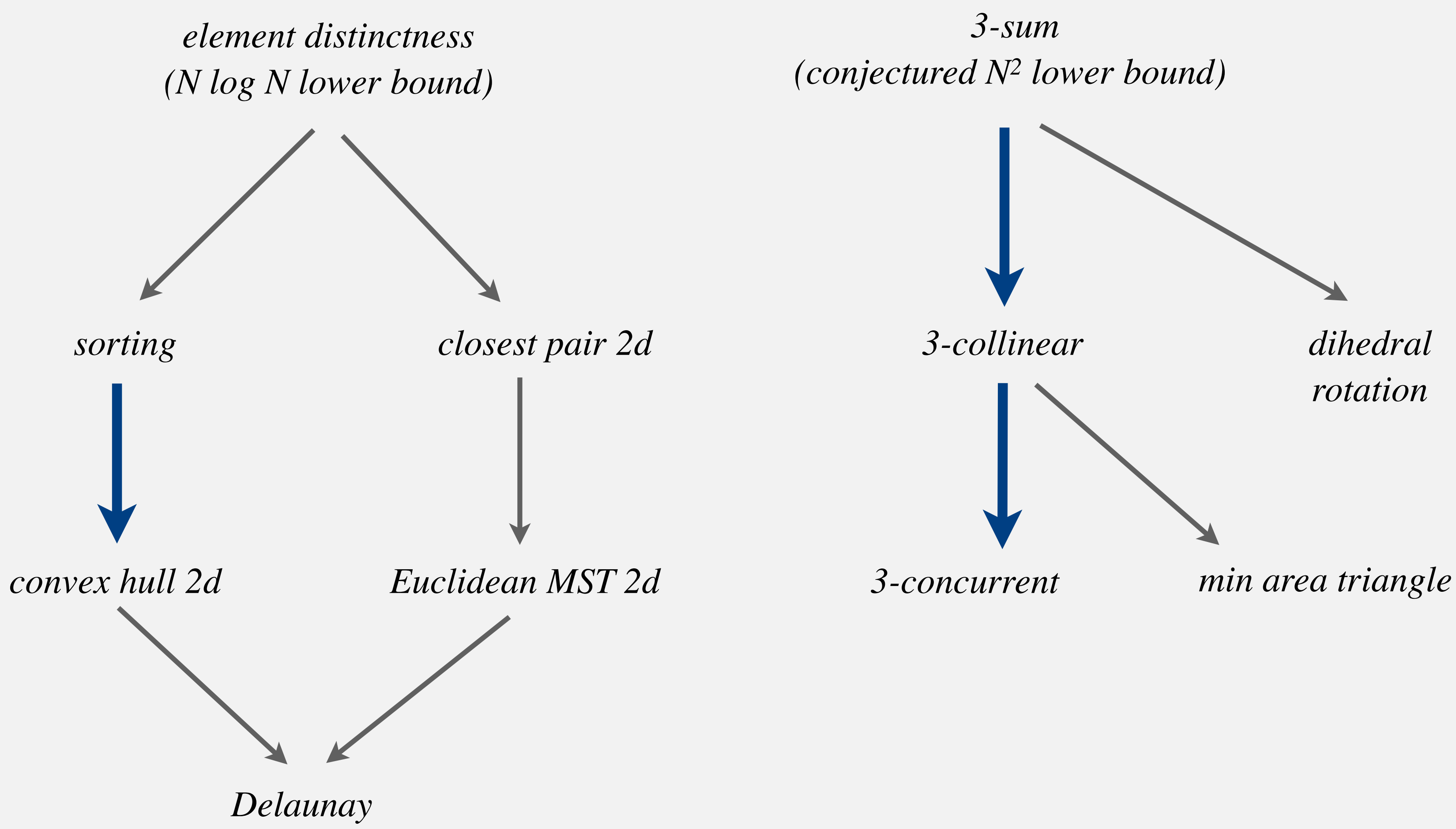
- 3-SUM instance: $x_1, x_2, \ldots, x_N$.
- 3-COLLINEAR instance: $(x_1, x_1^3), (x_2, x_2^3), \ldots, (x_N, x_N^3)$.

Lemma. If $a, b,$ and c are distinct, then $a + b + c = 0$ if and only if $(a, a^3), (b, b^3),$ and (c, c^3) are collinear.

Pf. Three distinct points $(a, a^3), (b, b^3),$ and (c, c^3) are collinear iff:

$$\begin{aligned} 0 &= \begin{vmatrix} a & a^3 & 1 \\ b & b^3 & 1 \\ c & c^3 & 1 \end{vmatrix} \\ &= a(b^3 - c^3) - b(a^3 - c^3) + c(a^3 - b^3) \\ &= (a-b)(b-c)(c-a)(a+b+c) \end{aligned}$$

More linear-time reductions and lower bounds



Establishing lower bounds: summary

Establishing lower bounds through reduction is an important tool in guiding algorithm design efforts.

Q. How to convince yourself no linear-time convex hull algorithm exists?

A1. [hard way] Long futile search for a linear-time algorithm.

A2. [easy way] Linear-time reduction from sorting.

Q. How to convince yourself no sub-quadratic 3-COLLINEAR algorithm exists.

A1. [hard way] Long futile search for a sub-quadratic algorithm.

A2. [easy way] Linear-time reduction from 3-SUM.

- designing algorithms
- establishing lower bounds
- intractability

Bird's-eye view

Def. A problem is intractable if it can't be solved in polynomial time.

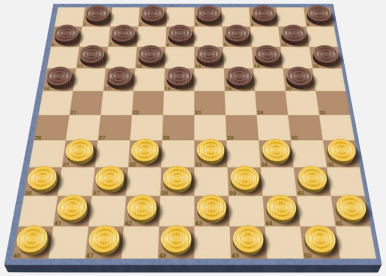
Desiderata. Prove that a problem is intractable.

Two problems that require exponential time.

input size = c + lg K

- Given a constant-size program, does it halt in at most K steps?
- Given N-by-N checkers board position, can the first player force a win?

using forced capture rule



Frustrating news. Few successes.

3-satisfiability

Literal. A boolean variable or its negation. x_i or $\neg x_i$

Clause. An *or* of 3 distinct literals. $C_1 = (\neg x_1 \lor x_2 \lor x_3)$

Conjunctive normal form. An *and* of clauses. $\Phi = (C_1 \land C_2 \land C_3 \land C_4 \land C_5)$

3-SAT. Given a CNF formula Φ consisting of k clauses over n literals, does it have a satisfying truth assignment?

$$\Phi = (\neg x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_2 \lor x_4) \land (\neg x_2 \lor x_3 \lor x_4)$$

yes instance

x_1	x_2	x_3	x_4
T	T	F	T

$$(\neg T \lor T \lor F) \land (T \lor \neg T \lor F) \land (\neg T \lor \neg T \lor \neg F) \land (\neg T \lor \neg T \lor T) \land (\neg T \lor F \lor T)$$

Applications. Circuit design, program correctness, ...

3-satisfiability is believed intractable

Q. How to solve an instance of 3-SAT with n variables?

A. Exhaustive search: try all 2^n truth assignments.

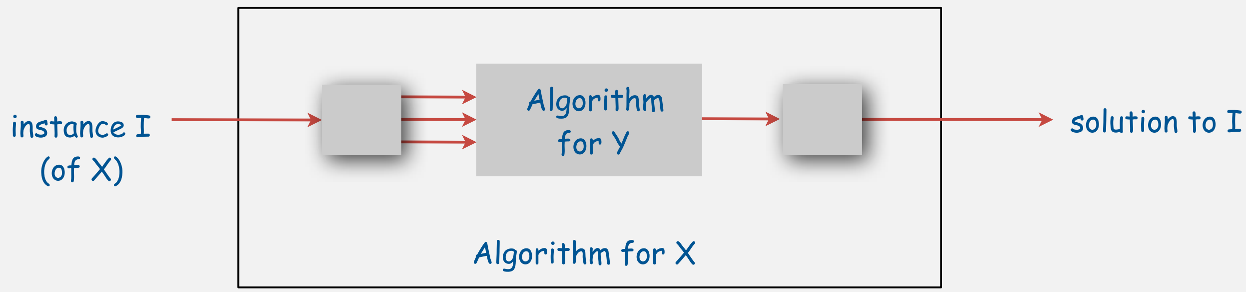
Q. Can we do anything substantially more clever?

Conjecture (P ≠ NP). 3-SAT is intractable (no poly-time algorithm).

Polynomial-time reductions

Def. Problem X poly-time (Cook) reduces to problem Y if X can be solved with:

- Polynomial number of standard computational steps.
- Polynomial number of calls to Y.



Establish intractability. If 3-SAT poly-time reduces to Y, then Y is intractable. (assuming 3-SAT is intractable)

Mentality.

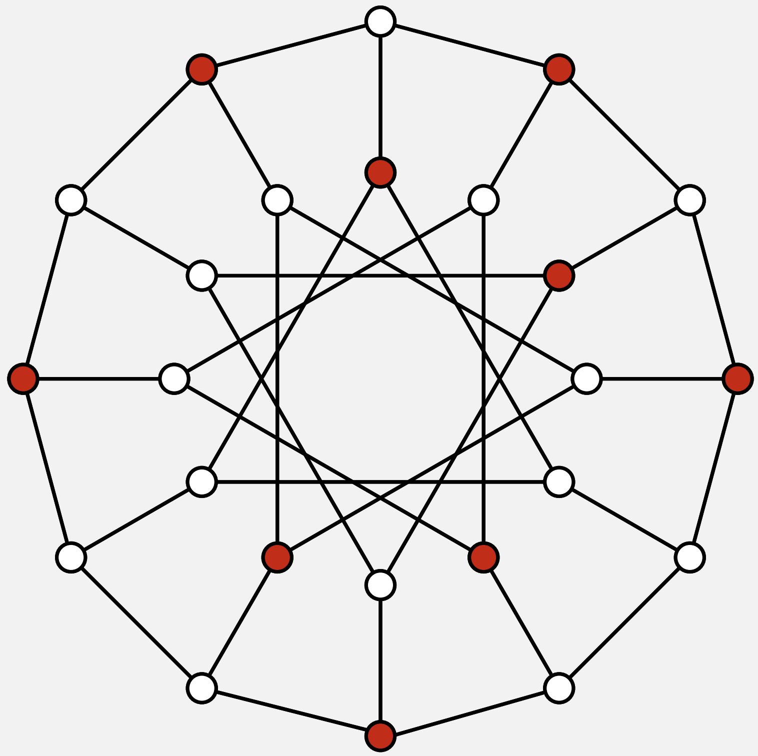
- If I could solve Y in poly-time, then I could also solve 3-SAT in poly-time.
- 3-SAT is believed to be intractable.
- Therefore, so is Y.

Independent set

Def. An independent set is a set of vertices, no two of which are adjacent.

IND-SET. Given a graph G and an integer k, find an independent set of size k.

k = 9

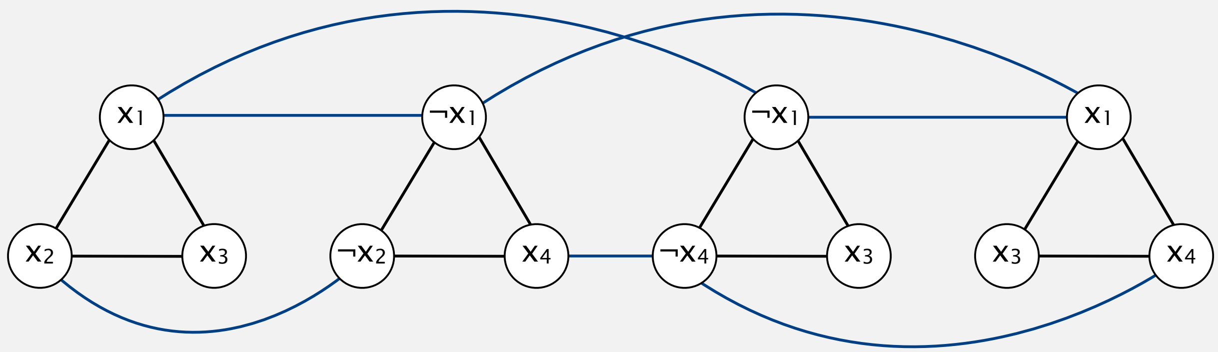


Applications. Scheduling, computer vision, clustering, ...

Proposition. 3-SAT poly-time reduces to IND-SET.

Pf. Given an instance Φ of 3-SAT, create an instance G of IND-SET:
- For each clause in Φ, create 3 vertices in a triangle.
- Add an edge between each literal and its negation.

k = 4



$$\Phi = (x_1 \lor x_2 \lor x_3) \land (\neg x_1 \lor \neg x_2 \lor x_4) \land (\neg x_1 \lor x_3 \lor \neg x_4) \land (x_1 \lor x_3 \lor x_4)$$

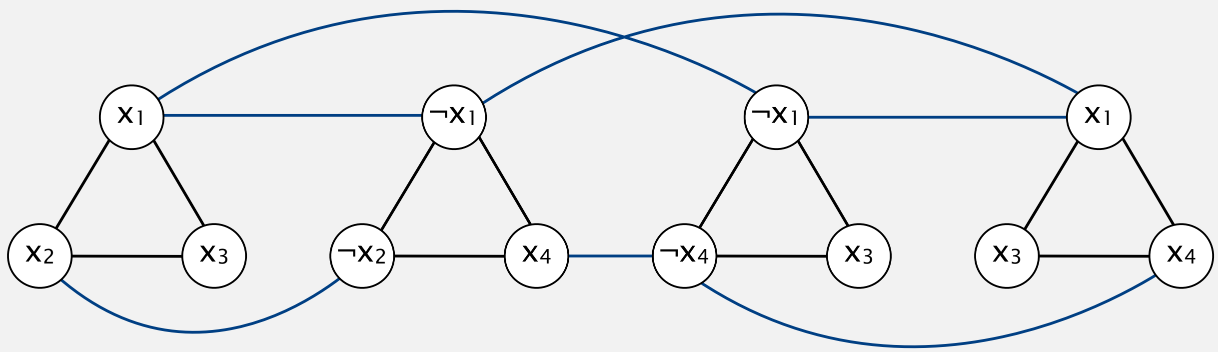
3-satisfiability reduces to independent set

Proposition. 3-SAT poly-time reduces to IND-SET.

Pf. Given an instance Φ of 3-SAT, create an instance G of IND-SET:

- For each clause in Φ, create 3 vertices in a triangle.
- Add an edge between each literal and its negation.

k = 4



$$\Phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (x_1 \vee x_3 \vee x_4)$$

- G has independent set of size $k \Rightarrow \Phi$ satisfiable.

set literals corresponding to vertices in independent to true;
set remaining literals in consistent manner

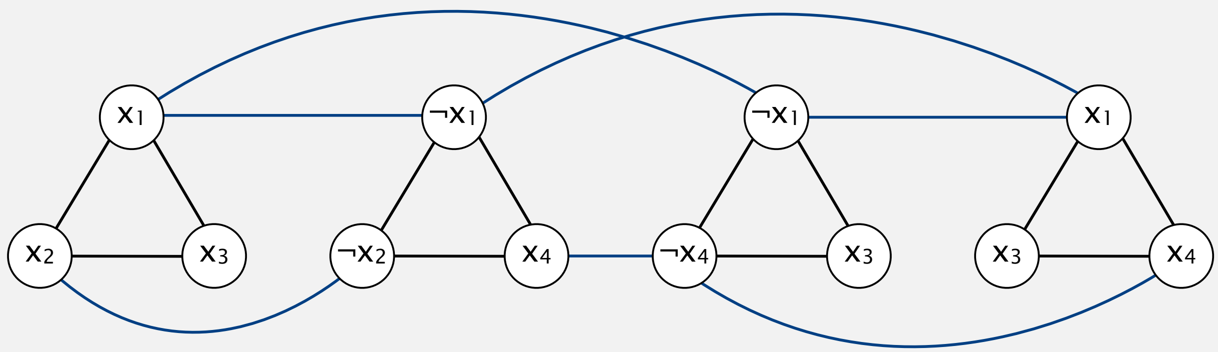
3-satisfiability reduces to independent set

Proposition. 3-SAT poly-time reduces to IND-SET.

Pf. Given an instance Φ of 3-SAT, create an instance G of IND-SET:

- For each clause in Φ, create 3 vertices in a triangle.
- Add an edge between each literal and its negation.

k = 4



$$\Phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (x_1 \vee x_3 \vee x_4)$$

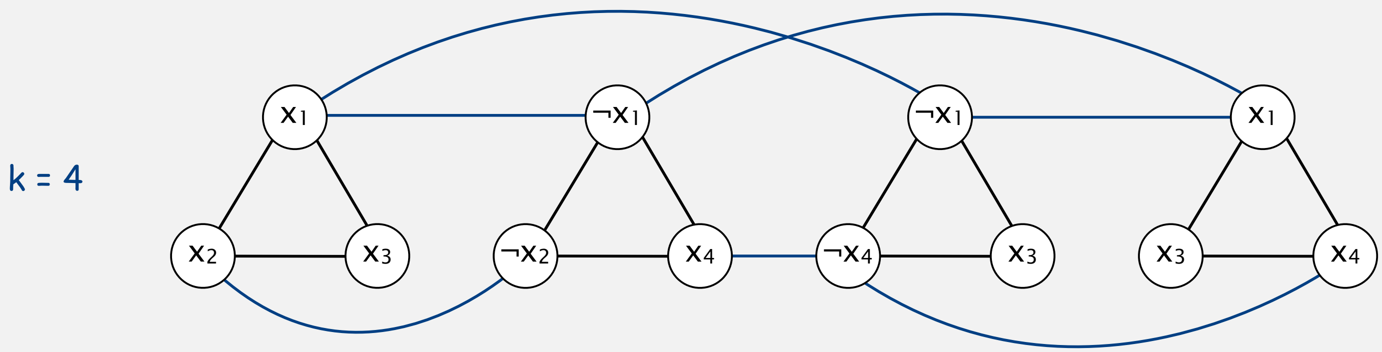
- G has independent set of size $k \Rightarrow \Phi$ satisfiable.
- Φ satisfiable $\Rightarrow G$ has independent set of size k.

for each clause, take vertex corresponding to one true literal

3-satisfiability reduces to independent set

Proposition. 3-SAT poly-time reduces to IND-SET.

Implication. Assuming 3-SAT is intractable, so is IND-SET.



$$\Phi = (x_1 \lor x_2 \lor x_3) \land (\neg x_1 \lor \neg x_2 \lor x_4) \land (\neg x_1 \lor x_3 \lor \neg x_4) \land (x_1 \lor x_3 \lor x_4)$$

Integer linear programming

ILP. Given a system of linear inequalities, find an integral solution.

$$3x_1 + 5x_2 + 2x_3 + x_4 + 4x_5 \ \geq \ 10$$

$$5x_1 + 2x_2 + 4x_4 + 1x_5 \ \leq \ 7$$

$$x_1 + x_3 + 2x_4 \ \leq \ 2$$

$$3x_1 + 4x_3 + 7x_4 \ \leq \ 7$$

$$x_1 + x_4 \ \leq \ 1$$

$$x_1 + x_3 + x_5 \ \leq \ 1$$

$$\text{all } x_i \ = \ \{\, 0, 1 \,\}$$

linear inequalities

integer variables

Context. Cornerstone problem in operations research.

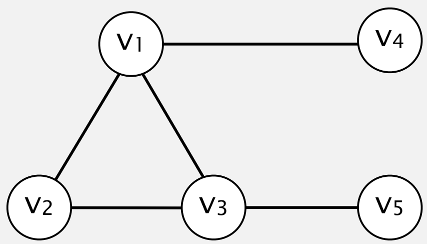
Remark. Finding a real-valued solution is tractable (linear programming).

Independent set reduces to integer linear programming

Proposition. IND-SET poly-time reduces to ILP.

Pf. Given an instance G, k of IND-SET, create an instance of ILP as follows:

Intuition. $x_i = 1$ if and only if vertex v_i is in independent set.



is there an independent set of size 3 ?

$$x_1 + x_2 + x_3 + x_4 + x_5 = 3$$

number of vertices selected

$$x_1 + x_2 \leq 1$$
$$x_2 + x_3 \leq 1$$
$$x_1 + x_3 \leq 1$$
$$x_1 + x_4 \leq 1$$
$$x_3 + x_5 \leq 1$$

at most one vertex selected from each edge

$$\text{all } x_i = \{ 0, 1 \}$$

binary variables

is there a feasible solution?

3-satisfiability reduces to integer linear programming

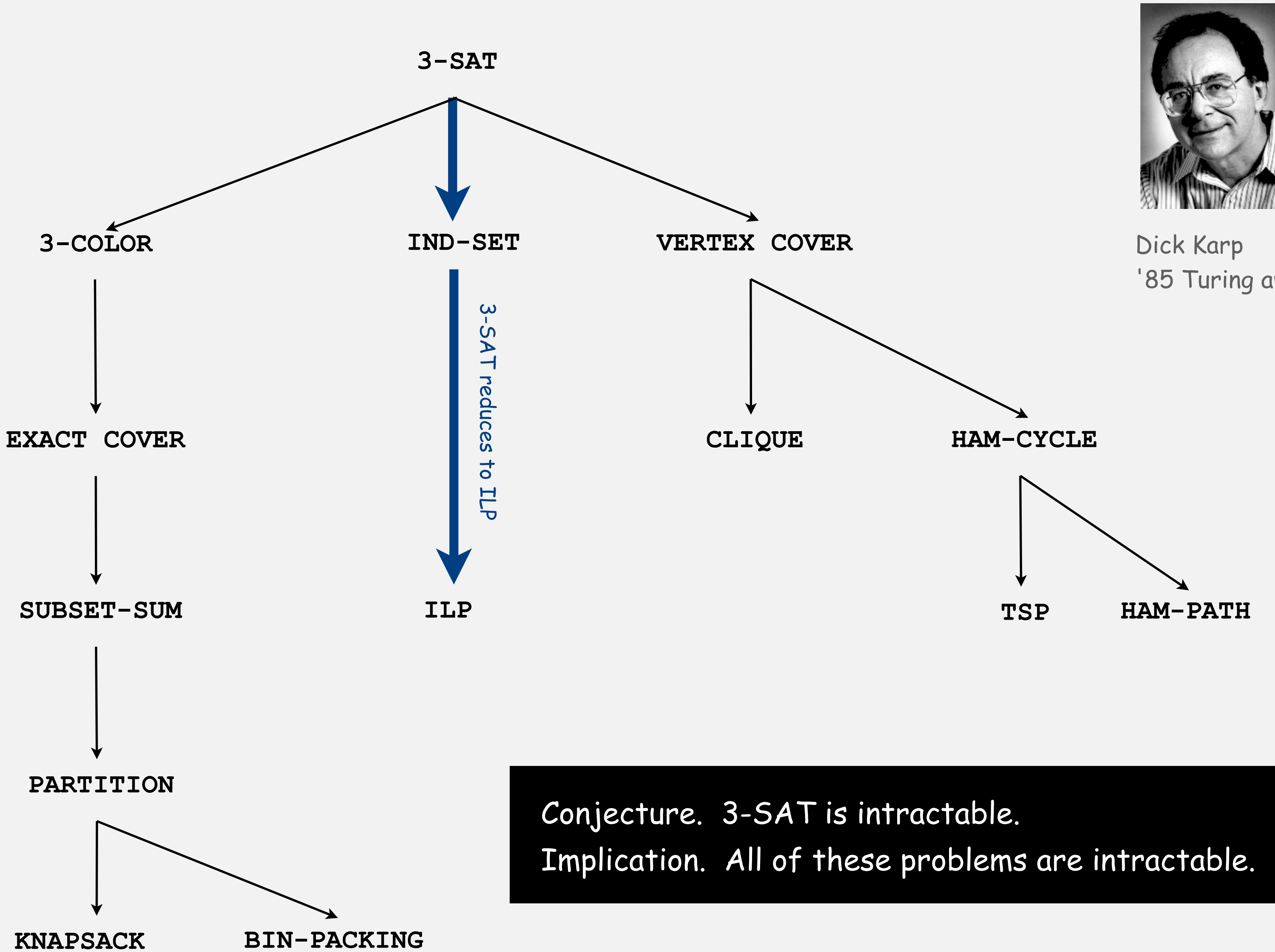
Proposition. 3-SAT poly-time reduces to IND-SET.

Proposition. IND-SET poly-time reduces to ILP.

Transitivity. If X poly-time reduces to Y and Y poly-time reduces to Z, then X-poly-time reduces to Z.

Implication. Assuming 3-SAT is intractable, so is ILP.

More poly-time reductions from 3-satisfiability



Dick Karp
'85 Turing award

Conjecture. 3-SAT is intractable.

Implication. All of these problems are intractable.

Implications of poly-time reductions from 3-satisfiability

Establishing intractability through poly-time reduction is an important tool in guiding algorithm design efforts.

Q. How to convince yourself that a new problem is (probably) intractable?
A1. [hard way] Long futile search for an efficient algorithm (as for 3-SAT).
A2. [easy way] Reduction from 3-SAT.

Caveat. Intricate reductions are common.

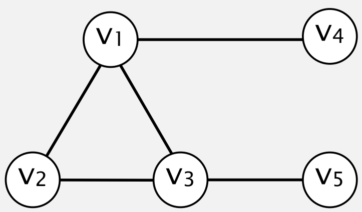
Search problem. Problem where you can check a solution in poly-time.

Ex 1. 3-SAT.

$$\Phi = (x_1 \lor x_2 \lor x_3) \land (\neg x_1 \lor \neg x_2 \lor x_4) \land (\neg x_1 \lor x_3 \lor \neg x_4) \land (x_1 \lor x_3 \lor x_4)$$

$x_1 = true, \; x_2 = true, \; x_3 = true, \; x_4 = true$

Ex 2. IND-SET.



$k = 3$

$\{ v_2 , \; x_4, \; v_5 \}$

P vs. NP

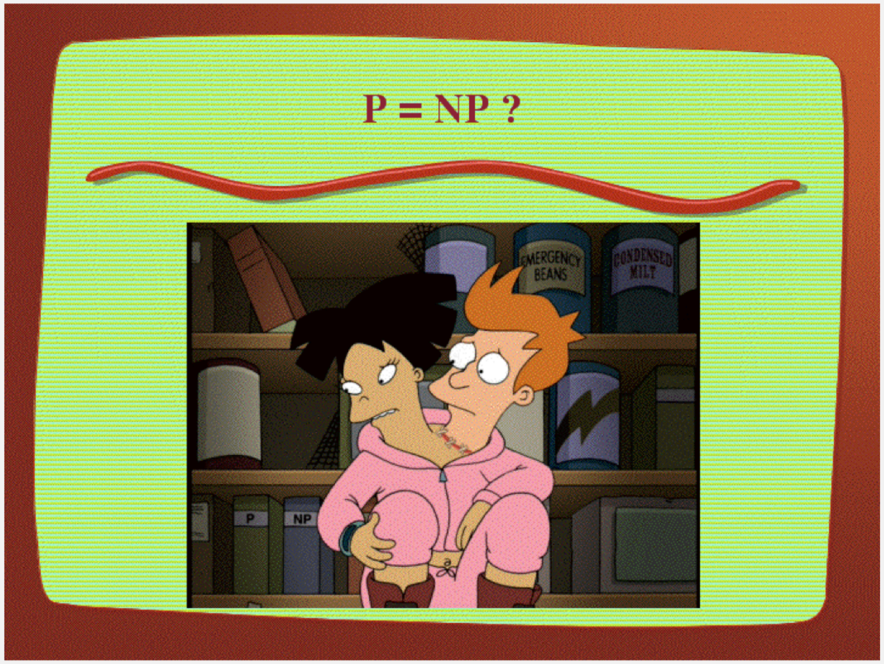
P. Set of search problems solvable in poly-time.

Importance. What scientists and engineers can compute feasibly.

NP. Set of search problems.

Importance. What scientists and engineers aspire to compute feasibly.

Fundamental question.



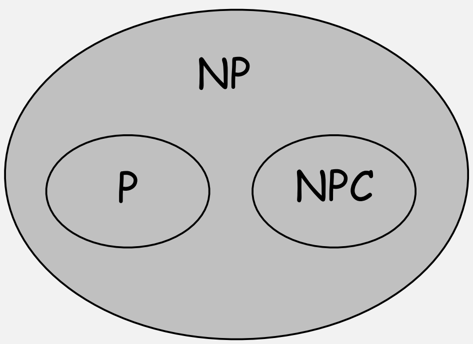
Consensus opinion. No.

Def. An NP is NP-complete if all problems in NP poly-time to reduce to it.

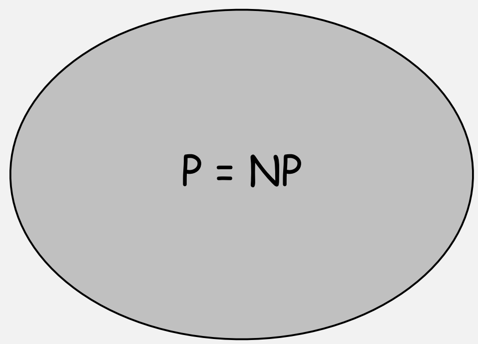
Cook's theorem. 3-SAT is NP-complete.

Corollary. 3-SAT is tractable if and only if P = NP.

Two worlds.

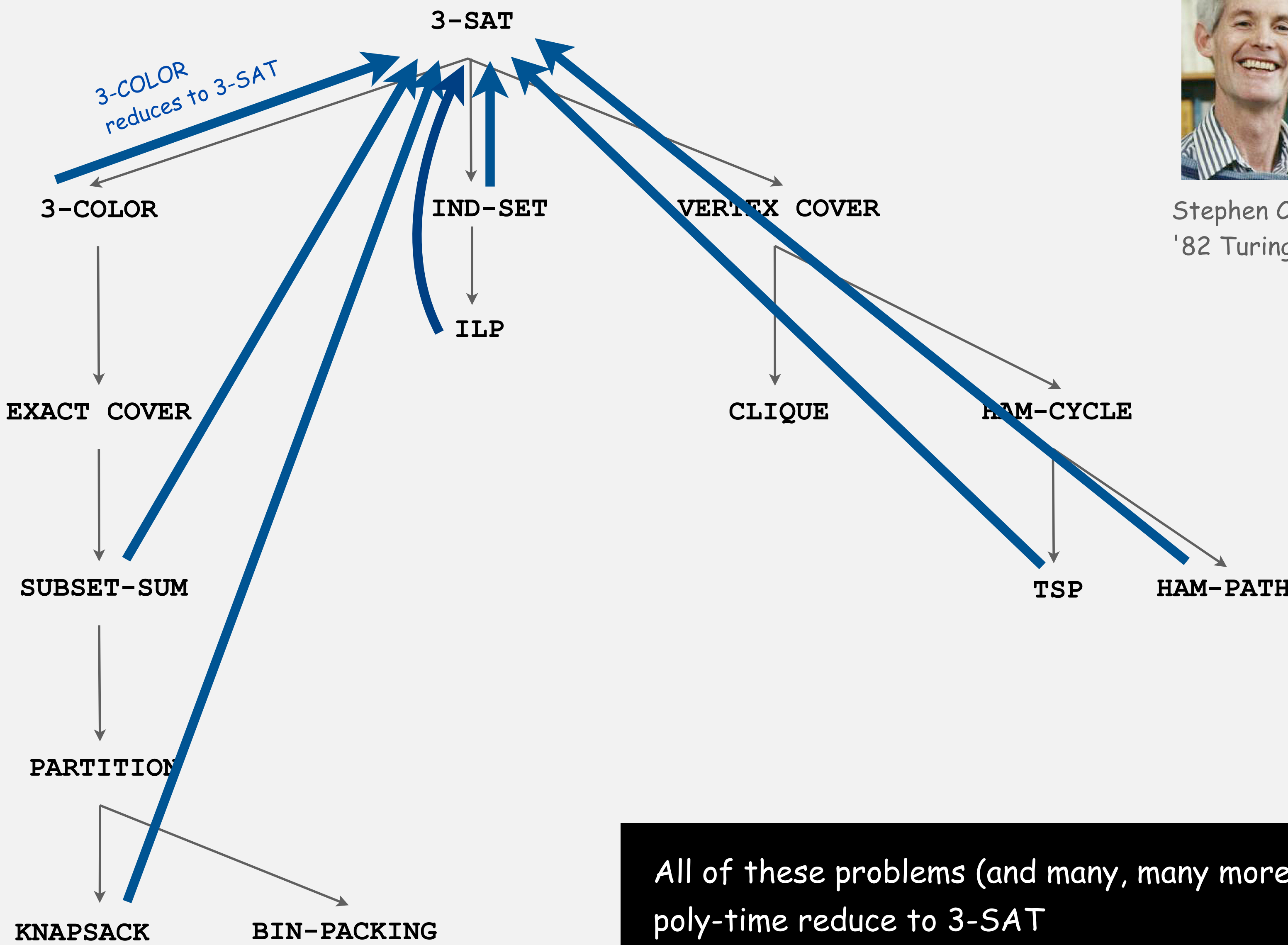


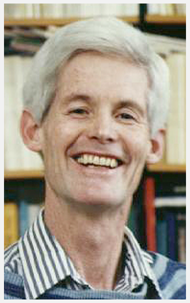
P ≠ NP



P = NP

Implications of Cook's theorem

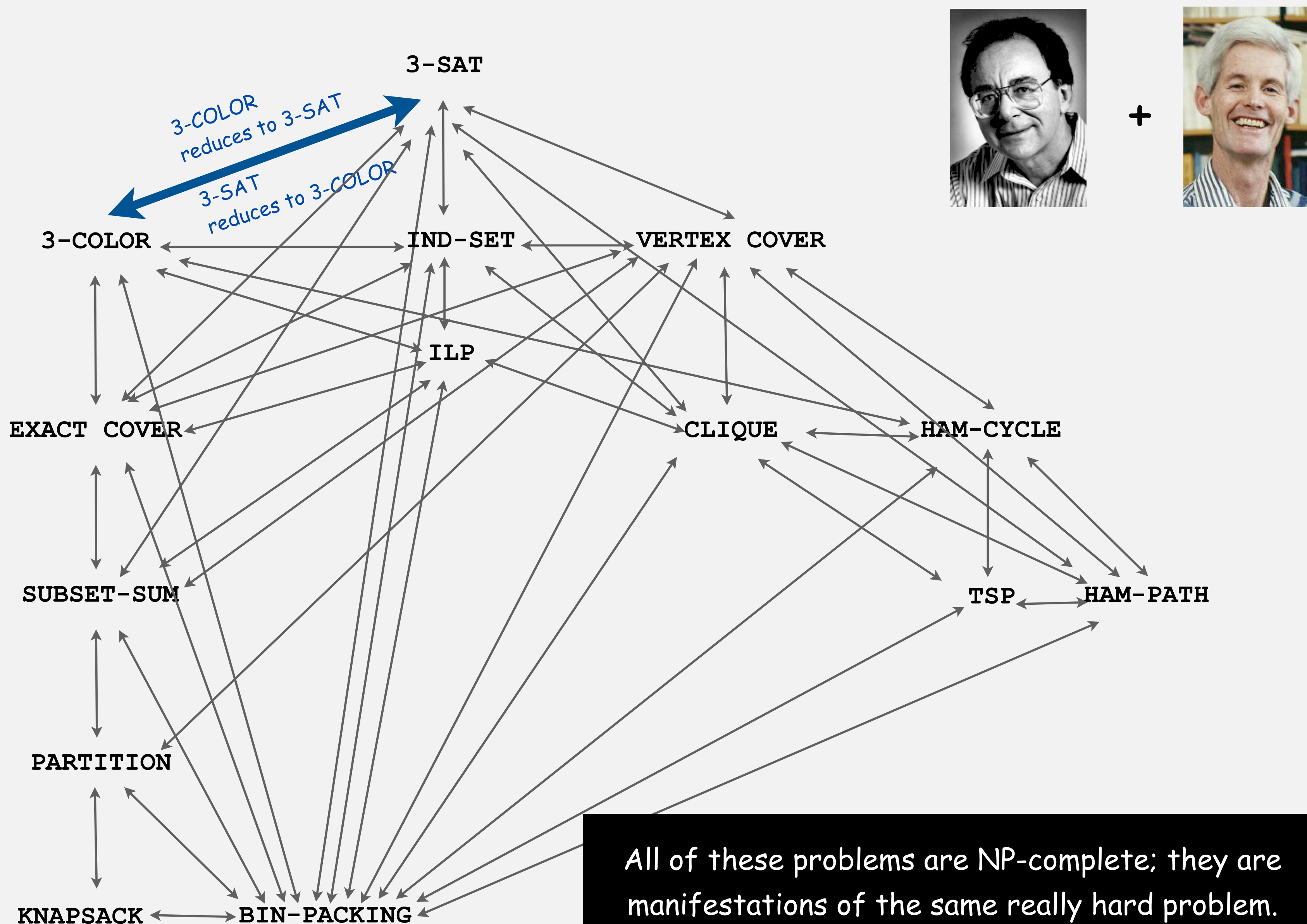




Stephen Cook
'82 Turing award

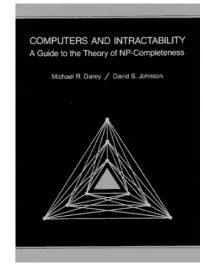
All of these problems (and many, many more) poly-time reduce to 3-SAT

Implications of Karp + Cook



All of these problems are NP-complete; they are manifestations of the same really hard problem.

Implications of NP-completeness



"I can't find an efficient algorithm, but neither can all these famous people."

Birds-eye view: review

Desiderata. Classify problems according to computational requirements.

complexity	order of growth	examples
linear	N	min, max, median, Burrows-Wheeler transform, …
linearithmic	$N \log N$	sorting, convex hull. closest pair, farthest pair, …
quadratic	N^2	???
	…	
exponential	c^N	???

Frustrating news. Huge number of problems have defied classification.

Birds-eye view: revised

Desiderata. Classify problems according to computational requirements.

complexity	order of growth	examples
linear	N	min, max, median, Burrows-Wheeler transform, …
linearithmic	$N \log N$	sorting, convex hull. closest pair, farthest pair, …
3-SUM complete	probably N^2	3-SUM, 3-COLLINEAR, 3-CONCURRENT, …
	…	
NP-complete	probably c^N	3-SAT, IND-SET, ILP, …

Good news. Can put problems in equivalence classes.

Summary

Reductions are important in theory to:

- Establish tractability.
- Establish intractability.
- Classify problems according to their computational requirements.

Reductions are important in practice to:

- Design algorithms.
- Design reusable software modules.
  - stack, queue, priority queue, symbol table, set, graph
  - sorting, regular expression, Delaunay triangulation
  - minimum spanning tree, shortest path, maximum flow, linear programming
- Determine difficulty of your problem and choose the right tool.
  - use exact algorithm for tractable problems
  - use heuristics for intractable problems

Combinatorial Search

- permutations
- backtracking
- counting
- subsets
- paths in a graph

Overview

Exhaustive search. Iterate through all elements of a search space.

Applicability. Huge range of problems (include intractable ones).

Caveat. Search space is typically exponential in size $\Rightarrow$ effectiveness may be limited to relatively small instances.

Backtracking. Systematic method for examining feasible solutions to a problem, by systematically pruning infeasible solutions.

Warmup: enumerate N-bit strings

Goal. Process all 2^N bit strings of length N.

- Maintain `a[i]` where `a[i]` represents bit `i`.
- Simple recursive method does the job.

```
// enumerate bits in a[k] to a[N-1]
private void enumerate(int k)
{
   if (k == N)
   {  process(); return;   }
   enumerate(k+1);
   a[k] = 1;
   enumerate(k+1);
   a[k] = 0;
}
```

clean up

`N = 3`

0	0	0
0	0	1
0	0	0
0	1	0
0	1	1
0	1	0
0	0	0
1	0	0
1	0	1
1	0	0
1	1	0
1	1	1
1	1	0
1	0	0
0	0	0

`N = 4`

0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

`a[0]` `a[N-1]`

Remark. Equivalent to counting in binary from 0 to $2^N - 1$.

Warmup: enumerate N-bit strings

```
public class BinaryCounter
{
   private int N;    // number of bits
   private int[] a;  // a[i] = ith bit

   public BinaryCounter(int N)
   {
      this.N = N;
      this.a = new int[N];
      enumerate(0);
   }

   private void process()
   {
      for (int i = 0; i < N; i++)
         StdOut.print(a[i]) + " ";
      StdOut.println();
   }

   private void enumerate(int k)
   {
     if (k == N)
     {  process(); return;  }
     enumerate(k+1);
     a[k] = 1;
     enumerate(k+1);
     a[k] = 0;
   }
}
```

```
public static void main(String[] args)
{
   int N = Integer.parseInt(args[0]);
   new BinaryCounter(N);
}
```

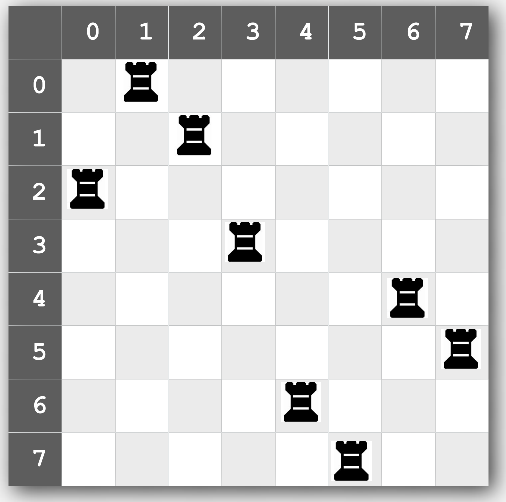
```
% java BinaryCounter 4
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1
```

all programs in this lecture are variations on this theme

- permutations
- backtracking
- counting
- subsets
- paths in a graph

N-rooks problem

Q. How many ways are there to place N rooks on an N-by-N board so that no rook can attack any other?




```
int[] a = { 2, 0, 1, 3, 6, 7, 4, 5 };
```

Representation. No two rooks in the same row or column $\Rightarrow$ permutation.

Challenge. Enumerate all N! permutations of 0 to N-1.

Enumerating permutations

Recursive algorithm to enumerate all N! permutations of size N.

- Start with permutation `a[0]` to `a[N-1]`.
- For each value of `i`:
  - swap `a[i]` into position `0`
  - enumerate all (N-1)! permutations of `a[1]` to `a[N-1]`
  - clean up (swap `a[i]` back to original position)

`N = 2`

```
0 1
1 0
0 1
```

`N = 3`

```
0 1 2
0 2 1
0 1 2
1 0 2
1 2 0
1 0 2
0 1 2
2 1 0
2 0 1
2 1 0
0 1 2
```

0 followed by perms of 1 2 3	1 followed by perms of 0 2 3	2 followed by perms of 1 0 3	3 followed by perms of 1 2 0
0 1 2 3	1 0 2 3	2 1 0 3	3 1 2 0
0 1 3 2	1 0 3 2	2 1 3 0	3 1 0 2
0 2 1 3	1 2 0 3	2 0 1 3	3 2 1 0
0 2 3 1	1 2 3 0	2 0 3 1	3 2 0 1
0 3 2 1	1 3 2 0	2 3 0 1	3 0 2 1
0 3 1 2	1 3 0 2	2 3 1 0	3 0 1 2

`a[0]` `a[N-1]`

cleanup swaps that bring perm back to original

Enumerating permutations

Recursive algorithm to enumerate all N! permutations of size N.

- Start with permutation `a[0]` to `a[N-1]`.
- For each value of `i`:
  - swap `a[i]` into position `0`
  - enumerate all (N-1)! permutations of `a[1]` to `a[N-1]`
  - clean up (swap `a[i]` back to original position)

```
// place N-k rooks in a[k] to a[N-1]
private void enumerate(int k)
{
   if (k == N)
   {  process(); return;  }

   for (int i = k; i < N; i++)
   {
      exch(k, i);
      enumerate(k+1);
      exch(i, k);
   }
}
```

clean up

```
% java Rooks 4
0 1 2 3
0 1 3 2
0 2 1 3
0 2 3 1
0 3 2 1
0 3 1 2
1 0 2 3
1 0 3 2
1 2 0 3
1 2 3 0
1 3 2 0
1 3 0 2
2 1 0 3
2 1 3 0
2 0 1 3
2 0 3 1
2 3 0 1
2 3 1 0
3 1 2 0
3 1 0 2
3 2 1 0
3 2 0 1
3 0 2 1
3 0 1 2
```

0 followed by perms of 1 2 3

1 followed by perms of 0 2 3

2 followed by perms of 1 0 3

3 followed by perms of 1 2 0

`a[0]` `a[N-1]`

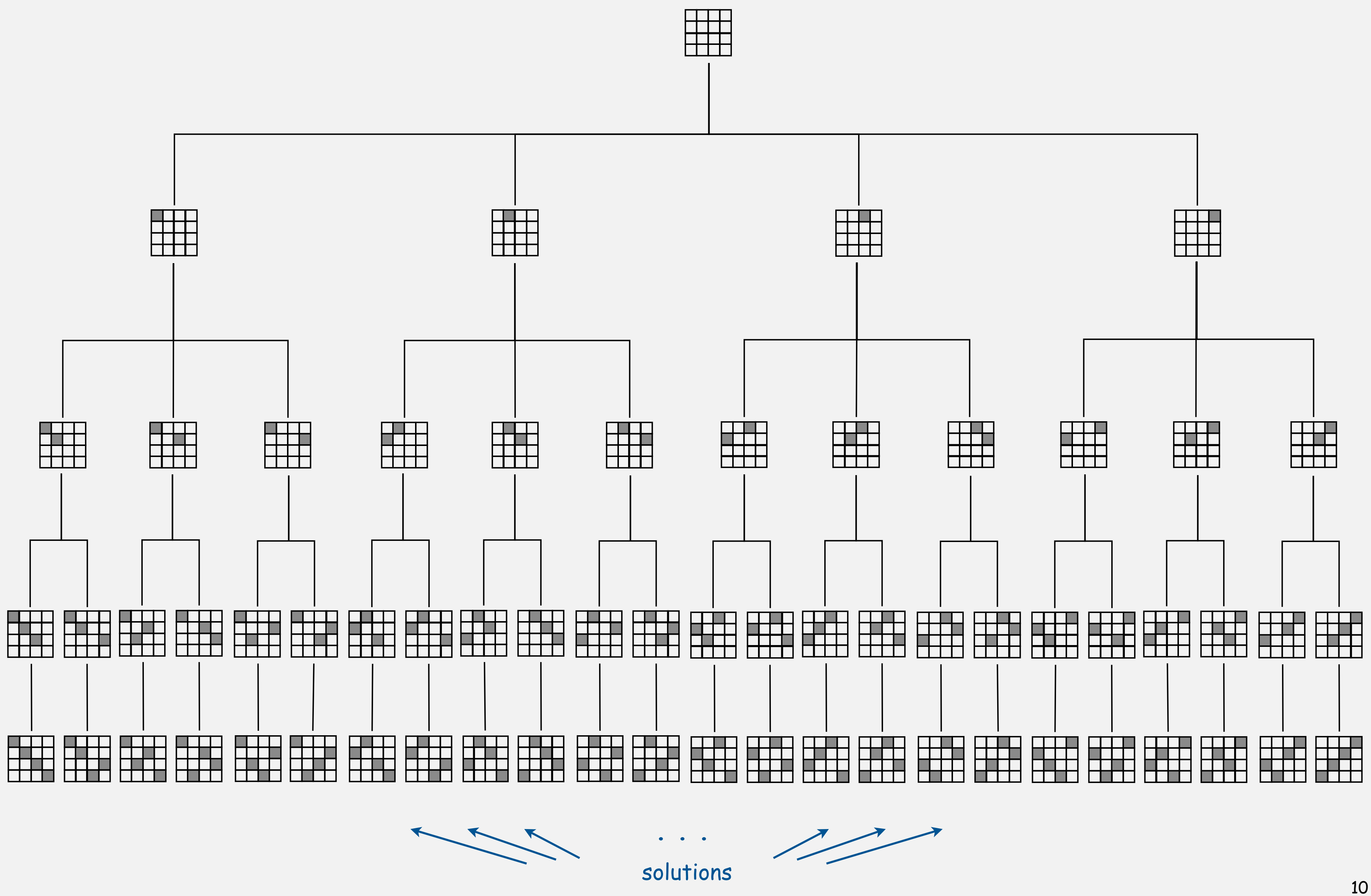
Enumerating permutations

```
public class Rooks
{
   private int N;
   private int[] a; // bits (0 or 1)

   public Rooks(int N)
   {
      this.N = N;
      a = new int[N];
      for (int i = 0; i < N; i++)
         a[i] = i;
      enumerate(0);
   }

   private void enumerate(int k)
   { /* see previous slide */  }

   private void exch(int i, int j)
   {  int t = a[i]; a[i] = a[j]; a[j] = t;   }

   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      new Rooks(N);
   }
}
```

initial permutation

```
% java Rooks 2
0 1
1 0

% java Rooks 3
0 1 2
0 2 1
1 0 2
1 2 0
2 1 0
2 0 1
```

4-rooks search tree



N-rooks problem: back-of-envelope running time estimate

Slow way to compute N!.

```
% java Rooks 7 | wc -l
5040

% java Rooks 8 | wc -l
40320

% java Rooks 9 | wc -l
362880

% java Rooks 10 | wc -l
3628800

% java Rooks 25 | wc -l
...
```

instant

1.6 seconds

15 seconds

170 seconds

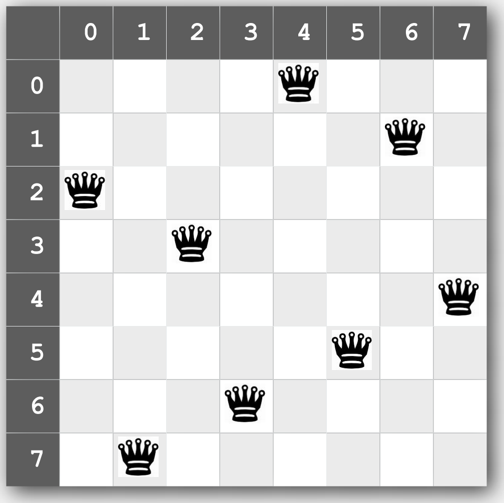
forever

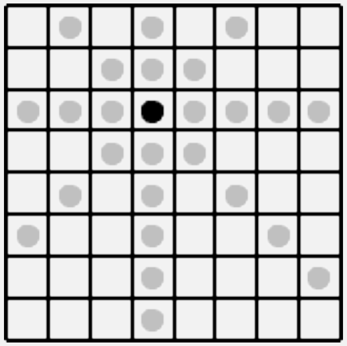
Hypothesis. Running time is about 2(N! / 8!) seconds.

- permutations
- **backtracking**
- counting
- subsets
- paths in a graph

N-queens problem

Q. How many ways are there to place N queens on an N-by-N board so that no queen can attack any other?





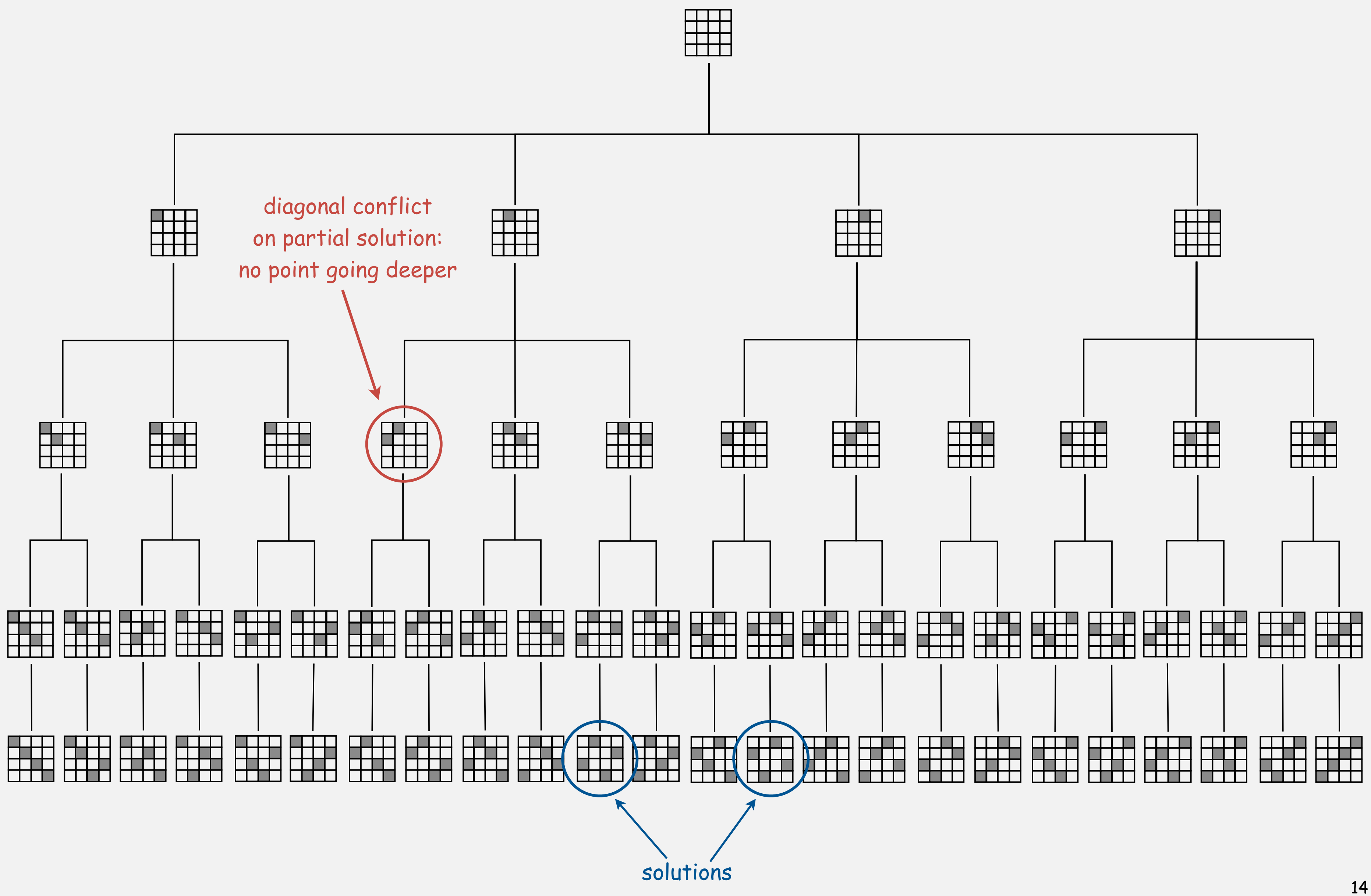
```
int[] a = { 2, 7, 3, 6, 0, 5, 1, 4 };
```

Representation. No two queens in the same row or column $\Rightarrow$ permutation.

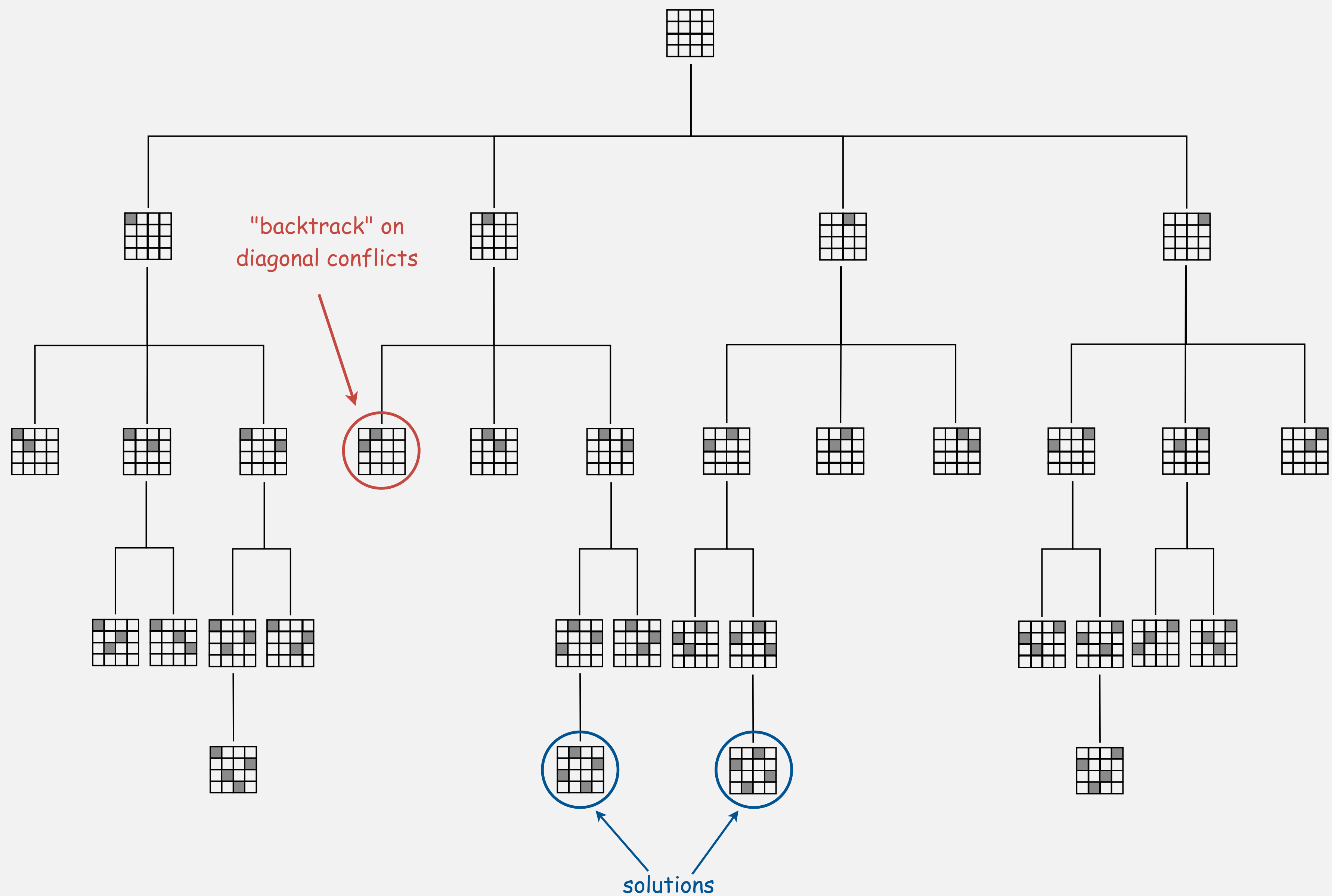
Additional constraint. No diagonal attack is possible.

Challenge. Enumerate (or even count) the solutions. ←— unlike N-rooks problem, nobody knows answer for N > 30

4-queens search tree



4-queens search tree (pruned)



N-queens problem: backtracking solution

Backtracking paradigm. Iterate through elements of search space.

- When there are several possible choices, make one choice and recur.
- If the choice is a dead end, backtrack to previous choice, and make next available choice.

Benefit. Identifying dead ends allows us to prune the search tree.

Ex. [backtracking for N-queens problem]

- Dead end: a diagonal conflict.
- Pruning: backtrack and try next column when diagonal conflict found.

N-queens problem: backtracking solution

```
private boolean backtrack(int k)
{
   for (int i = 0; i < k; i++)
   {
      if ((a[i] - a[k]) == (k - i)) return true;
      if ((a[k] - a[i]) == (k - i)) return true;
   }
   return false;
}

// place N-k queens in a[k] to a[N-1]
private void enumerate(int k)
{
   if (k == N)
   {  process(); return;  }

   for (int i = k; i < N; i++)
   {
      exch(k, i);
      if (!backtrack(k)) enumerate(k+1);
      exch(i, k);
   }
}
```

stop enumerating if adding queen k leads to a diagonal violation

```
% java Queens 4
1 3 0 2
2 0 3 1

% java Queens 5
0 2 4 1 3
0 3 1 4 2
1 3 0 2 4
1 4 2 0 3
2 0 3 1 4
2 4 1 3 0
3 1 4 2 0
3 0 2 4 1
4 1 3 0 2
4 2 0 3 1

% java Queens 6
1 3 5 0 2 4
2 5 1 4 0 3
3 0 4 1 5 2
4 2 0 5 3 1
```

a[0]   a[N-1]

N-queens problem: effectiveness of backtracking

Pruning the search tree leads to enormous time savings.

N	$Q(N)$	$N!$
2	0	2
3	0	6
4	2	24
5	10	120
6	4	720
7	40	5,040
8	92	40,320
9	352	362,880
10	724	3,628,800
11	2,680	39,916,800
12	14,200	479,001,600
13	73,712	6,227,020,800
14	365,596	87,178,291,200

N-queens problem: How many solutions?

```
% java Queens 13 | wc -l
73712

% java Queens 14 | wc -l
365596

% java Queens 15 | wc -l
2279184

% java Queens 16 | wc -l
14772512

% java Queens 17 | wc -l
...
```

← 1.1 seconds

← 5.4 seconds

← 29 seconds

← 210 seconds

← 1352 seconds

Hypothesis. Running time is about $(N! / 2.5^N) / 43{,}000$ seconds.

Conjecture. $Q(N)$ is $\sim N! / c^N$, where c is about 2.54.

Counting: Java implementation

Goal. Enumerate all N-digit base-R numbers.

Solution. Generalize binary counter in lecture warmup.

```
// enumerate base-R numbers in a[k] to a[N-1]
private static void enumerate(int k)
{
   if (k == N)
   {  process(); return;  }

   for (int r = 0; r < R; r++)
   {
      a[k] = r;
      enumerate(k+1);
   }
   a[k] = 0;
}
```

cleanup not needed; why?

```
% java Counter 2 4
0 0
0 1
0 2
0 3
1 0
1 1
1 2
1 3
2 0
2 1
2 2
2 3
3 0
3 1
3 2
3 3

% java Counter 3 2
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
```

a[0] a[N-1]

Counting application: Sudoku

Goal. Fill 9-by-9 grid so that every row, column, and box contains each of the digits 1 through 9.

7		8				3		
			2		1			
5								
	4						2	6
3				8				
			1				9	
	9		6					4
				7		5		

Remark. Natural generalization is NP-complete.

Counting application: Sudoku

Goal. Fill 9-by-9 grid so that every row, column, and box contains each of the digits 1 through 9.

7	2	8	9	4	6	3	1	5
9	3	4	2	5	1	6	7	8
5	1	6	7	3	8	2	4	9
1	4	7	5	9	3	8	2	6
3	6	9	4	8	2	1	5	7
8	5	2	1	6	7	4	9	3
2	9	3	6	1	5	7	8	4
4	8	1	3	7	9	5	6	2
6	7	5	8	2	4	9	3	1

Solution. Enumerate all 81-digit base-9 numbers (with backtracking).

`a[]`

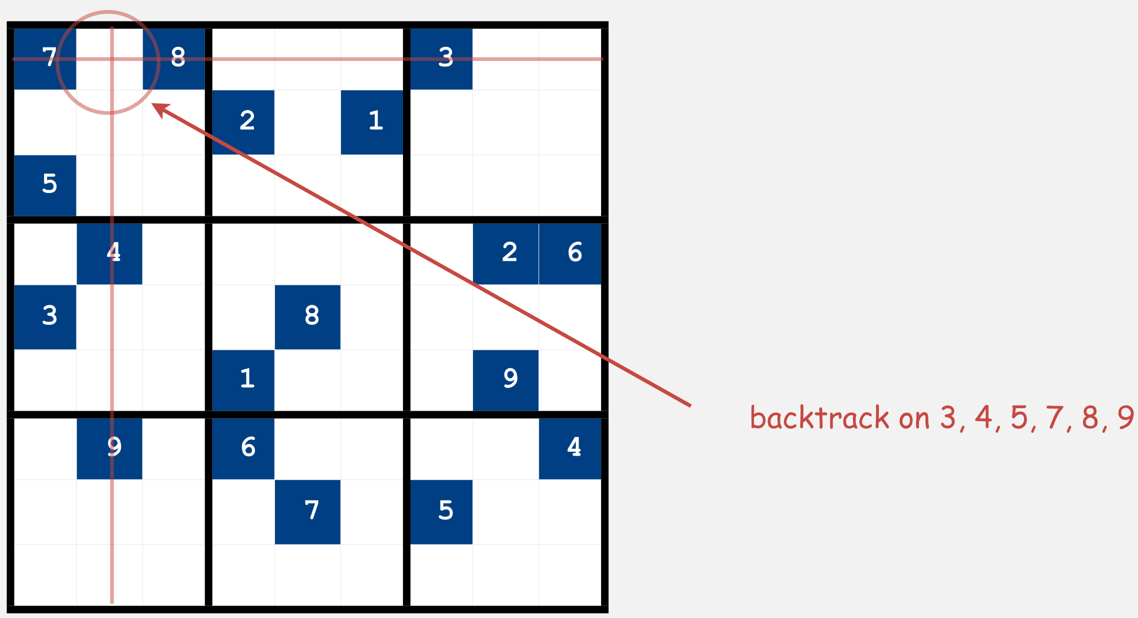
using digits 1 to 9 →

7		8					3		...	
0	1	2	3	4	5	6	7	8		80

Sudoku: backtracking solution

Iterate through elements of search space.

- For each empty cell, there are 9 possible choices.
- Make one choice and recur.
- If you find a conflict in row, column, or box, then backtrack.



Sudoku: Java implementation

```
private void enumerate(int k)
{
   if (k == 81)
   {  process(); return;  }

   if (a[k] != 0)
   {  enumerate(k+1); return;  }

   for (int r = 1; r <= 9; r++)
   {
         a[k] = r;
         if (!backtrack(k))
            enumerate(k+1);
   }

   a[k] = 0;
}
```

found a solution

cell k initially filled in; recur on next cell

try 9 possible digits for cell k

unless it violates a Sudoku constraint (see booksite for code)

clean up

```
% more board.txt
7 0 8 0 0 0 3 0 0
0 0 0 2 0 1 0 0 0
5 0 0 0 0 0 0 0 0
0 4 0 0 0 0 0 2 6
3 0 0 0 8 0 0 0 0
0 0 0 1 0 0 0 9 0
0 9 0 6 0 0 0 0 4
0 0 0 0 7 0 5 0 0
0 0 0 0 0 0 0 0 0

% java Sudoku < board.txt
7 2 8 9 4 6 3 1 5
9 3 4 2 5 1 6 7 8
5 1 6 7 3 8 2 4 9
1 4 7 5 9 3 8 2 6
3 6 9 4 8 2 1 5 7
8 5 2 1 6 7 4 9 3
2 9 3 6 1 5 7 8 4
4 8 1 3 7 9 5 6 2
6 7 5 8 2 4 9 3 1
```

Enumerating subsets: natural binary encoding

Given N items, enumerate all 2^N subsets.

- Count in binary from 0 to $2^N - 1$.
- Bit i represents item i.
- If 0, in subset; if 1, not in subset.

i	*binary*	*subset*	*complement*
0	0 0 0 0	empty	4 3 2 1
1	0 0 0 1	1	4 3 2
2	0 0 1 0	2	4 3 1
3	0 0 1 1	2 1	4 3
4	0 1 0 0	3	4 2 1
5	0 1 0 1	3 1	4 2
6	0 1 1 0	3 2	4 1
7	0 1 1 1	3 2 1	4
8	1 0 0 0	4	3 2 1
9	1 0 0 1	4 1	3 2
10	1 0 1 0	4 2	3 1
11	1 0 1 1	4 2 1	3
12	1 1 0 0	4 3	2 1
13	1 1 0 1	4 3 1	2
14	1 1 1 0	4 3 2	1
15	1 1 1 1	4 3 2 1	empty

Enumerating subsets: natural binary encoding

Given N items, enumerate all 2^N subsets.

- Count in binary from 0 to 2^N - 1.
- Maintain `a[i]` where `a[i]` represents item `i`.
- If `0`, `a[i]` in subset; if `1`, `a[i]` not in subset.

Binary counter from warmup does the job.

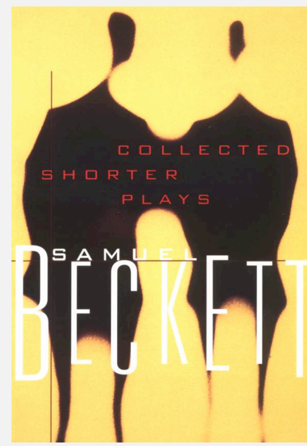
```
private void enumerate(int k)
{
  if (k == N)
  {  process(); return;   }
  enumerate(k+1);
  a[k] = 1;
  enumerate(k+1);
  a[n] = 0;
}
```

Digression: Samuel Beckett play

Quad. Starting with empty stage, 4 characters enter and exit one at a time, such that each subset of actors appears exactly once.

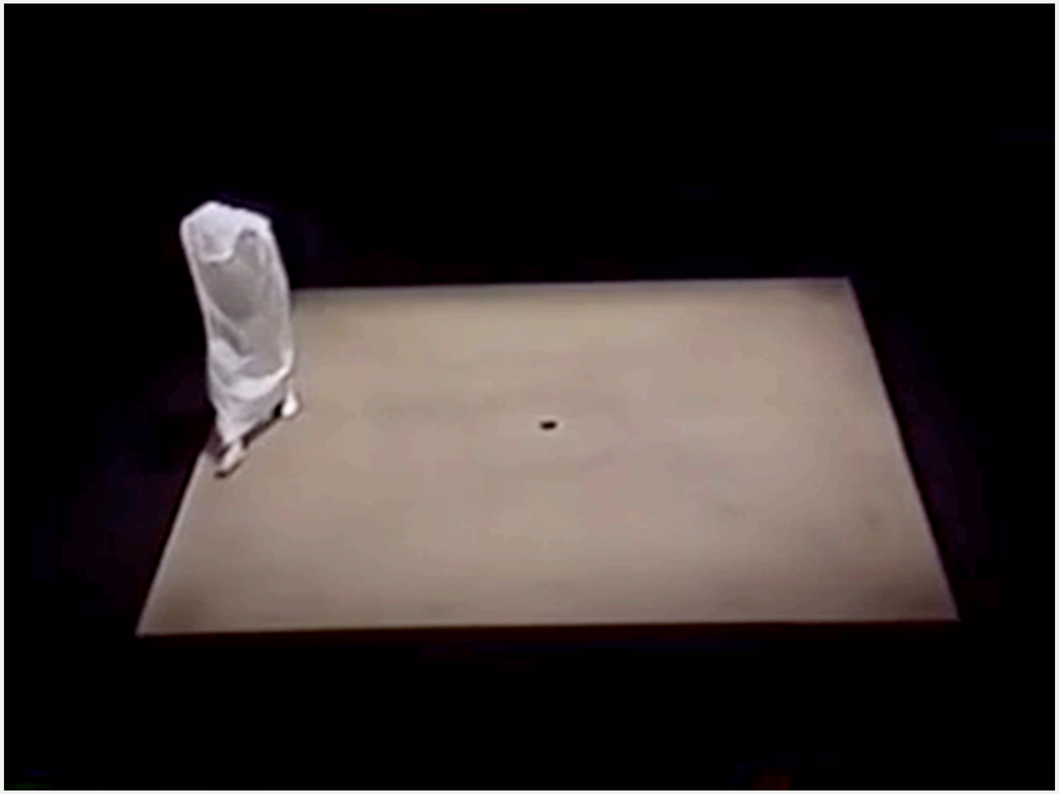
code	subset	move
0 0 0 0	empty	
0 0 0 1	1	enter 1
0 0 1 1	2 1	enter 2
0 0 1 0	2	exit 1
0 1 1 0	3 2	enter 3
0 1 1 1	3 2 1	enter 1
0 1 0 1	3 1	exit 2
0 1 0 0	3	exit 1
1 1 0 0	4 3	enter 4
1 1 0 1	4 3 1	enter 1
1 1 1 1	4 3 2 1	enter 2
1 1 1 0	4 3 2	exit 1
1 0 1 0	4 2	exit 3
1 0 1 1	4 2 1	enter 1
1 0 0 1	4 1	exit 2
1 0 0 0	4	exit 1

ruler function



Digression: Samuel Beckett play

Quad. Starting with empty stage, 4 characters enter and exit one at a time, such that each subset of actors appears exactly once.

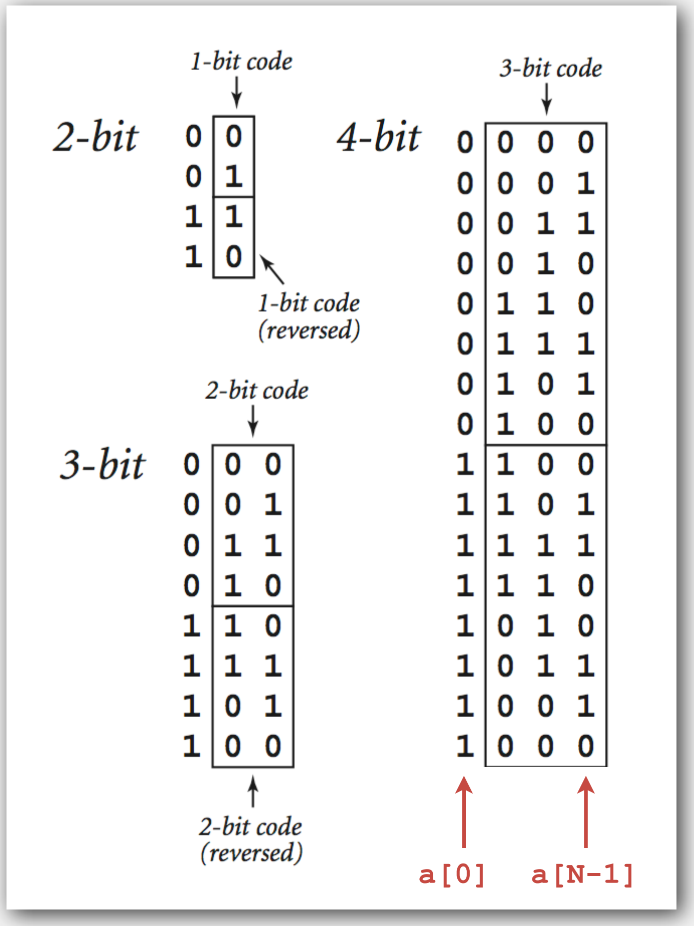


"faceless, emotionless one of the far future, a world where people are born, go through prescribed movements, fear non-being even though their lives are meaningless, and then they disappear or die." — Sidney Homan

Binary reflected gray code

Def. The k-bit **binary reflected Gray code** is:

- the (k-1) bit code with a 0 prepended to each word, followed by
- the (k-1) bit code in reverse order, with a 1 prepended to each word.



Enumerating subsets using Gray code

Two simple changes to binary counter from warmup:

- Flip `a[k]` instead of setting it to 1.
- Eliminate cleanup.

Gray code binary counter

```
// all bit strings in a[k] to a[N-1]
private void enumerate(int k)
{
  if (k == N)
  { process(); return; }
  enumerate(k+1);
  a[k] = 1 - a[k];
  enumerate(k+1);
}
```

0	0	0
0	0	1
0	1	1
0	1	0
1	1	0
1	1	1
1	0	1
1	0	0

same values since no cleanup

standard binary counter (from warmup)

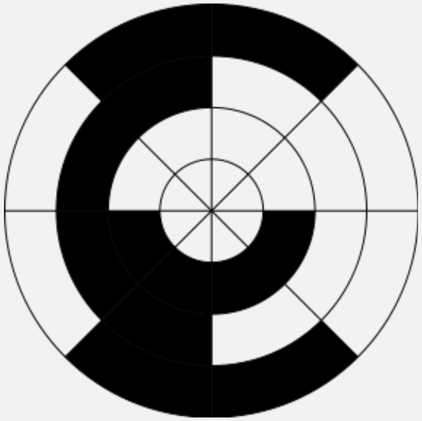
```
// all bit strings in a[k] to a[N-1]
private void enumerate(int k)
{
  if (k == N)
  { process(); return; }
  enumerate(k+1);
  a[k] = 1;
  enumerate(k+1);
  a[k] = 0;
}
```

0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

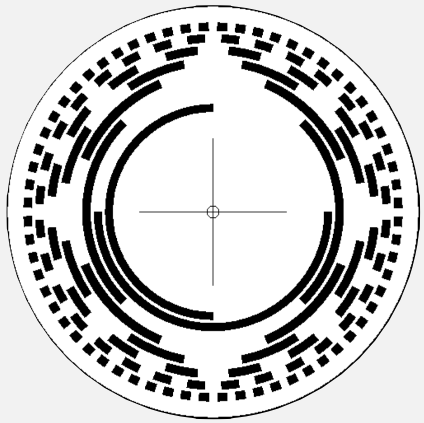
`a[0]` `a[N-1]`

Advantage. Only one item in subset changes at a time.

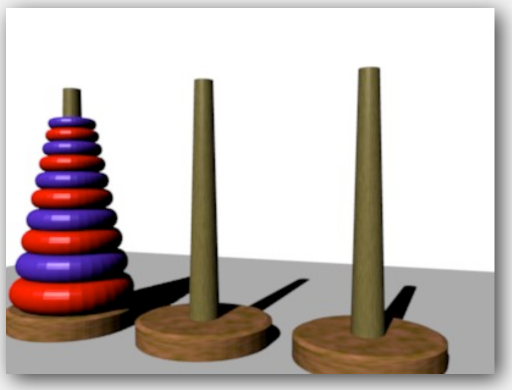
More applications of Gray codes



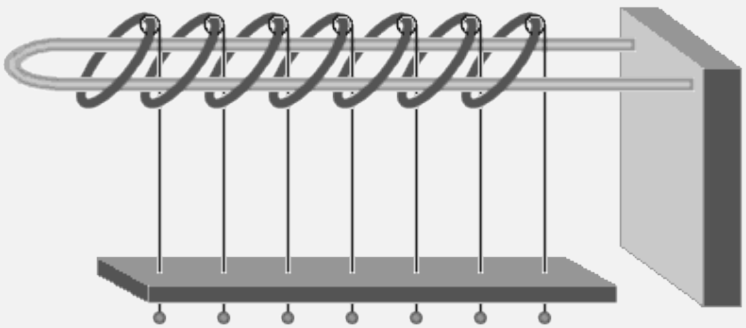
3-bit rotary encoder



8-bit rotary encoder



Towers of Hanoi



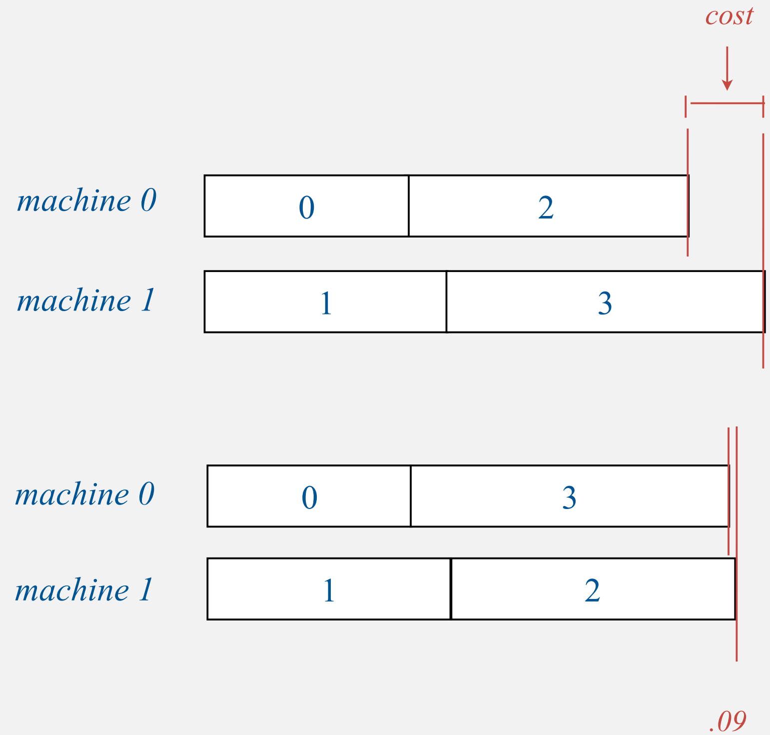
Chinese ring puzzle

Scheduling

Scheduling (set partitioning). Given n jobs of varying length, divide among two machines to minimize the makespan (time the last job finishes).

or, equivalently, difference between finish times

job	length
0	1.41
1	1.73
2	2.00
3	2.23



Remark. This scheduling problem is NP-complete.

Scheduling (full implementation)

```
public class Scheduler
{
   private int N;            // Number of jobs.
   private int[] a;          // Subset assignments.
   private int[] b;          // Best assignment.
   private double[] jobs;    // Job lengths.

   public Scheduler(double[] jobs)
   {
      this.N = jobs.length;
      this.jobs = jobs;
      a = new int[N];
      b = new int[N];
      enumerate(N);
   }

   public int[] best()
   {  return b;  }

   private void enumerate(int k)
   {  /* Gray code enumeration. */   }

   private void process()
   {
     if (cost(a) < cost(b))
       for (int i = 0; i < N; i++)
         b[i] = a[i];
   }

   public static void main(String[] args)
   {  /* create Scheduler, print results */  }
}
```

trace of

```
% java Scheduler 4 < jobs.txt
```

a[]				finish times		cost
0	0	0	0	7.38	0.00	7.38
0	0	0	1	5.15	2.24	2.91
0	0	1	1	3.15	4.24	1.09
0	0	1	0	5.38	2.00	
0	1	1	0	3.65	3.73	0.08
0	1	1	1	1.41	5.97	
0	1	0	1	3.41	3.97	
0	1	0	0	5.65	1.73	
1	1	0	0	4.24	3.15	
1	1	0	1	2.00	5.38	
1	1	1	1	0.00	7.38	
1	1	1	0	2.24	5.15	
1	0	1	0	3.97	3.41	
1	0	1	1	1.73	5.65	
1	0	0	1	3.73	3.65	
1	0	0	0	5.97	1.41	

```
  MACHINE 0     MACHINE 1
 1.4142135624
              1.7320508076
              2.0000000000
 2.2360679775
------------------------------
 3.6502815399 3.7320508076
```

Scheduling (larger example)

Observation. Large number of subsets leads to remarkably low cost.

```
% java Scheduler < jobs.txt
  MACHINE 0       MACHINE 1
 1.4142135624
 1.7320508076
                2.0000000000
 2.2360679775
 2.4494897428
                2.6457513111
                2.8284271247
                3.0000000000
 3.1622776602
                3.3166247904
                3.4641016151
                3.6055512755
                3.7416573868
 3.8729833462
                4.0000000000
 4.1231056256
                4.2426406871
 4.3588989435
                4.4721359550
 4.5825756950
 4.6904157598
 4.7958315233
 4.8989794856
                5.0000000000
-------------------------------
42.3168901295 42.3168901457
```

cost < 10^{-8} →

Scheduling: improvements

Many opportunities (details omitted).

- Fix last job to be on machine 0 (quick factor-of-two improvement).
- Maintain difference in finish times (instead of recomputing from scratch).
- Backtrack when partial schedule cannot beat best known.
  (check total against goal: half of total job times)

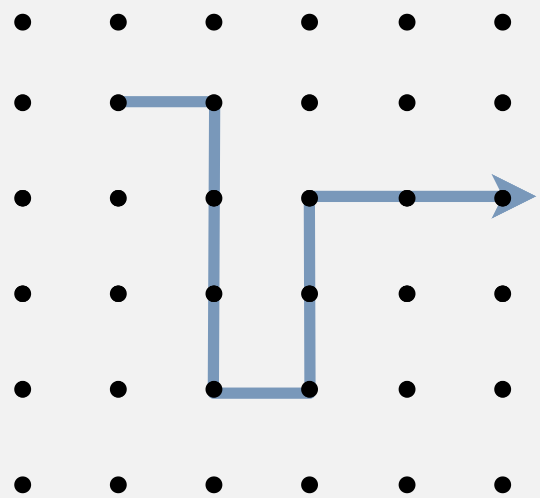
```
private void enumerate(int k)
{
   if (k == N-1)
   {  process(); return;  }
   if (backtrack(k)) return;
   enumerate(k+1);
   a[k] = 1 - a[k];
   enumerate(k+1);
}
```

- Process all 2^k subsets of last k jobs, keep results in memory,
  (reduces time to 2^{N-k} when 2^k memory available).

- permutations
- backtracking
- counting
- subsets
- **paths in a graph**

Enumerating all paths on a grid

Goal. Enumerate all simple paths on a grid of adjacent sites.

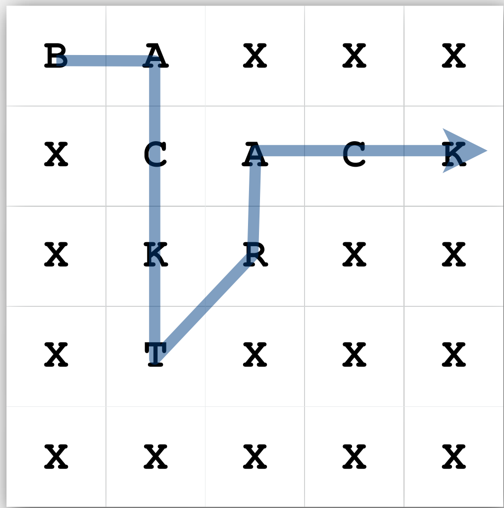


no two atoms can occupy same position at same time

Application. Self-avoiding lattice walk to model polymer chains.

Enumerating all paths on a grid: Boggle

Boggle. Find all words that can be formed by tracing a simple path of adjacent cubes (left, right, up, down, diagonal).



Pruning. Stop as soon as no word in dictionary contains string of letters on current path as a prefix ⇒ use a trie.

B
BA
BAX

Boggle: Java implementation

string of letters on current path to (i, j)

```
private void dfs(String prefix, int i, int j)
{
   if ((i < 0 || i >= N) ||
       (j < 0 || j >= N) ||
       (visited[i][j])    ||
       !dictionary.containsAsPrefix(prefix))
      return;

   visited[i][j] = true;
   prefix = prefix + board[i][j];

   if (dictionary.contains(prefix))
      found.add(prefix);

   for (int ii = -1; ii <= 1; ii++)
      for (int jj = -1; jj <= 1; jj++)
         dfs(prefix, i + ii, j + jj);

   visited[i][j] = false;
}
```

backtrack

add current character

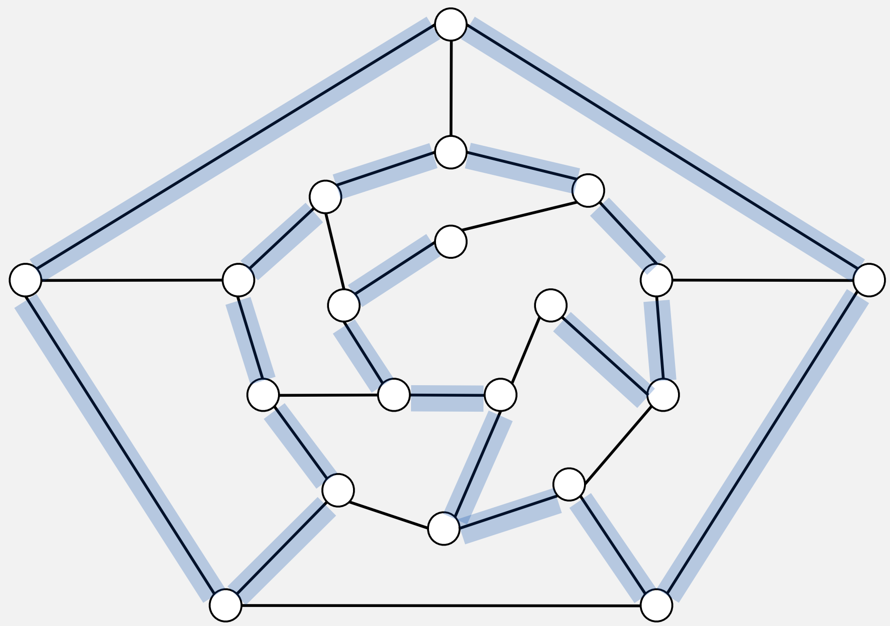
add to set of found words

try all possibilities

clean up

Hamilton path

Goal. Find a simple path that visits every vertex exactly once.

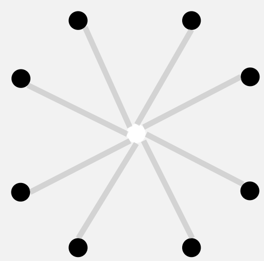


visit every edge exactly once

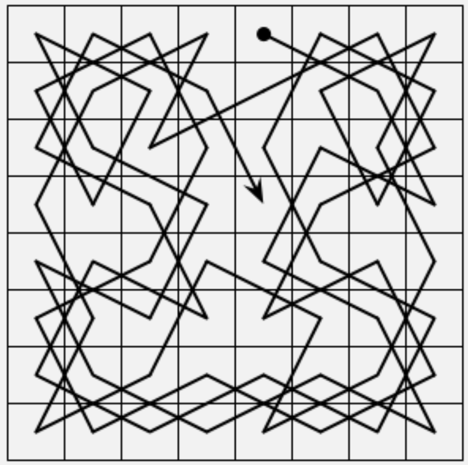
Remark. Euler path easy, but Hamilton path is NP-complete.

Knight's tour

Goal. Find a sequence of moves for a knight so that (starting from any desired square) it visits every square on a chessboard exactly once.



legal knight moves



a knight's tour

Solution. Find a Hamilton path in knight's graph.

Hamilton path: backtracking solution

Backtracking solution. To find Hamilton path starting at `v`:

- Add `v` to current path.
- For each vertex `w` adjacent to `v`
  - find a simple path starting at `w` using all remaining vertices
- Clean up: remove `v` from current path.

Q. How to implement?

A. Add cleanup to DFS (!!)

Hamilton path: Java implementation

```
public class HamiltonPath
{
   private boolean[] marked;    // vertices on current path
   private int count = 0;       // number of Hamiltonian paths

   public HamiltonPath(Graph G)
   {
      marked = new boolean[G.V()];
      for (int v = 0; v < G.V(); v++)
         dfs(G, v, 1);
   }

   private void dfs(Graph G, int v, int depth)
   {
      marked[v] = true;
      if (depth == G.V()) count++;

      for (int w : G.adj(v))
         if (!marked[w]) dfs(G, w, depth+1);

      marked[v] = false;
   }
}
```

found one

length of current path (depth of recursion)

backtrack if w is already part of path

clean up

Exhaustive search: summary

problem	enumeration	backtracking
N-rooks	permutations	no
N-queens	permutations	yes
Sudoku	base-9 numbers	yes
scheduling	subsets	yes
Boggle	paths in a grid	yes
Hamilton path	paths in a graph	yes

The longest path

Woh-oh-oh-oh, find the longest path!
Woh-oh-oh-oh, find the longest path!

If you said P is NP tonight,
There would still be papers left to write,
I have a weakness,
I'm addicted to completeness,
And I keep searching for the longest path.

The algorithm I would like to see
Is of polynomial degree,
But it's elusive:
Nobody has found conclusive
Evidence that we can find a longest path.

I have been hard working for so long.
I swear it's right, and he marks it wrong.
Some how I'll feel sorry when it's done: GPA 2.1
Is more than I hope for.

Garey, Johnson, Karp and other men (and women)
Tried to make it order N log N.
Am I a mad fool
If I spend my life in grad school,
Forever following the longest path?

Woh-oh-oh-oh, find the longest path!
Woh-oh-oh-oh, find the longest path!
Woh-oh-oh-oh, find the longest path.

Recorded by Dan Barrett in 1988
while a student at Johns Hopkins
during a difficult algorithms final

That's all, folks: Keep searching!

The world's longest path (Chile): 8500 km