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

<table>
<thead>
<tr>
<th>graph</th>
<th>vertex</th>
<th>edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>communication</td>
<td>telephone, computer</td>
<td>fiber optic cable</td>
</tr>
<tr>
<td>circuit</td>
<td>gate, register, processor</td>
<td>wire</td>
</tr>
<tr>
<td>mechanical</td>
<td>joint</td>
<td>rod, beam, spring</td>
</tr>
<tr>
<td>financial</td>
<td>stock, currency</td>
<td>transactions</td>
</tr>
<tr>
<td>transportation</td>
<td>street intersection, airport</td>
<td>highway, airway route</td>
</tr>
<tr>
<td>internet</td>
<td>class C network</td>
<td>connection</td>
</tr>
<tr>
<td>game</td>
<td>board position</td>
<td>legal move</td>
</tr>
<tr>
<td>social relationship</td>
<td>person, actor</td>
<td>friendship, movie cast</td>
</tr>
<tr>
<td>neural network</td>
<td>neuron</td>
<td>synapse</td>
</tr>
<tr>
<td>protein network</td>
<td>protein</td>
<td>protein-protein interaction</td>
</tr>
<tr>
<td>chemical compound</td>
<td>molecule</td>
<td>bond</td>
</tr>
</tbody>
</table>
Graph terminology

- vertex
- cycle
- edge
- clique
- spanning tree
- path
- tree
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

```java
public class Graph

    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);
for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
      /* process edge v-w */
```

% more tiny.txt
7
0 1
0 2
0 5
0 6
3 4
3 5
4 6
Maintain a list of the edges (linked list or array).

Set of edges representation
Adjacency-matrix representation

Maintain a two-dimensional V-by-V boolean array; for each edge v-w in graph: \( \text{adj}[v][w] = \text{adj}[w][v] = \text{true}. \)
Adjacency-matrix representation: Java implementation

```java
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-list representation

Maintain vertex-indexed array of lists (implementation omitted).

two entries for each edge
**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

```java
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];
    }
}
```
**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.”

<table>
<thead>
<tr>
<th>representation</th>
<th>space</th>
<th>insert edge</th>
<th>edge between v and w?</th>
<th>iterate over edges incident to v?</th>
</tr>
</thead>
<tbody>
<tr>
<td>list of edges</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>adjacency matrix</td>
<td>$V^2$</td>
<td>1</td>
<td>1</td>
<td>V</td>
</tr>
<tr>
<td>adjacency list</td>
<td>$E + V$</td>
<td>degree(v)</td>
<td>degree(v)</td>
<td>degree(v)</td>
</tr>
<tr>
<td>adjacency set</td>
<td>$E + V$</td>
<td>log (degree(v))</td>
<td>log (degree(v))</td>
<td>degree(v)</td>
</tr>
</tbody>
</table>
› 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

Instructions:
1) "place walls" as you wish
2) "find path"
3) "clear maze" or "clear path" and repeat
NOTE: instead of making a maze you can use a "premade maze"

- start tile
- finish tile
- wall tile
- path tile
- blocked tile
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.

```java
// print all vertices connected to s
In in = new In(args[0]);
Graph G = new Graph(in);
int s = 0;
DFSSearcher dfs = new DFSSearcher(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., `DFSSearcher`.
- Query the graph-processing routine for information.
Depth-first search (connectivity)

```java
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`
Flood fill

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.
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
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 $t$?

<table>
<thead>
<tr>
<th>method</th>
<th>preprocessing time</th>
<th>query time</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>union-find</td>
<td>$V + E \log^* V$</td>
<td>$\log^* V \uparrow$</td>
<td>$V$</td>
</tr>
<tr>
<td>DFS</td>
<td>$E + V$</td>
<td>$1$</td>
<td>$E + V$</td>
</tr>
</tbody>
</table>

$\uparrow$ 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 \( \text{pred}[v] \) (parent-link representation).

**Retrace path.** To find path between \( s \) and \( v \), follow \( \text{pred}[] \) 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)

```java
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 \( s \).
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 */  }
}
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.
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 $s = \text{Kevin Bacon}$.
› 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*

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
</tr>
</tbody>
</table>

**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.

---

<table>
<thead>
<tr>
<th>preprocess time</th>
<th>query time</th>
<th>extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E + V$</td>
<td>1</td>
<td>$V$</td>
</tr>
</tbody>
</table>
Depth-first search for connected components

```java
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**: `cc` array for storing component labels.
- **Constant-time connectivity query**: `connected(int v, int w)` function for checking connectivity in constant time.
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++;
       }
}

private void dfs(Graph G, int v)
{
    cc[v] = components;
    for (int w : G.adj(v))
       if (cc[w] == UNMARKED) dfs(G, w);
}
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.

**Input.** Scanned image.

**Output.** Number of red and blue states.

assuming contiguous states
**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.
**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.

**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.

**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.
Bridges of Königsberg

The Seven Bridges of Königsberg. [Leonhard Euler 1736]

“The ... 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.
Graph-processing challenge 6

Problem. Lay out a graph in the plane without crossing edges?

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
Directed graphs

**Digraph.** Set of vertices connected pairwise by oriented edges.
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.
Vertex = synset.
Edge = hypernym relationship.

WordNet graph
# Digraph applications

<table>
<thead>
<tr>
<th>graph</th>
<th>vertex</th>
<th>edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>transportation</td>
<td>street intersection</td>
<td>one-way street</td>
</tr>
<tr>
<td>web</td>
<td>web page</td>
<td>hyperlink</td>
</tr>
<tr>
<td>food web</td>
<td>species</td>
<td>predator-prey relationship</td>
</tr>
<tr>
<td>WordNet</td>
<td>synset</td>
<td>hypernym</td>
</tr>
<tr>
<td>scheduling</td>
<td>task</td>
<td>precedence constraint</td>
</tr>
<tr>
<td>financial</td>
<td>stock, currency</td>
<td>transaction</td>
</tr>
<tr>
<td>cell phone</td>
<td>person</td>
<td>placed call</td>
</tr>
<tr>
<td>infectious disease</td>
<td>person</td>
<td>infection</td>
</tr>
<tr>
<td>game</td>
<td>board position</td>
<td>legal move</td>
</tr>
<tr>
<td>citation</td>
<td>journal article</td>
<td>citation</td>
</tr>
<tr>
<td>object graph</td>
<td>object</td>
<td>pointer</td>
</tr>
<tr>
<td>inheritance hierarchy</td>
<td>class</td>
<td>inherits from</td>
</tr>
<tr>
<td>control flow</td>
<td>code block</td>
<td>jump</td>
</tr>
</tbody>
</table>
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**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Digraph(int V)</code></td>
<td>create an empty digraph with ( V ) vertices</td>
</tr>
<tr>
<td><code>Digraph(In in)</code></td>
<td>create a digraph from input stream</td>
</tr>
<tr>
<td><code>void addEdge(int v, int w)</code></td>
<td>add an edge from ( v ) to ( w )</td>
</tr>
<tr>
<td><code>Iterable&lt;Integer&gt; adj(int v)</code></td>
<td>return an iterator over the neighbors of ( v )</td>
</tr>
<tr>
<td><code>int V()</code></td>
<td>return number of vertices</td>
</tr>
</tbody>
</table>

```java
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 \( v \rightarrow w \) */
```
Set of edges representation

Store a list of the edges (linked list or array).
Adjacency-matrix representation

Maintain a two-dimensional $V$-by-$V$ boolean array; for each edge $v \rightarrow w$ in the digraph: $\text{adj}[v][w] = \text{true}$. 

```
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
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  1  0  0  0  0  0  0  0  0  0  0
5  0  0  0  1  0  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  1  0
12 0  0  0  0  0  0  0  0  0  0  0  0  0
```
Adjacency-list representation

Maintain vertex-indexed array of lists.

same as undirected graph, but one entry for each edge
Adjacency-set representation

Maintain vertex-indexed array of sets.

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.

```java
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];
    }
}
```
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.

<table>
<thead>
<tr>
<th>representation</th>
<th>space</th>
<th>insert edge from v to w</th>
<th>edge from v to w?</th>
<th>iterate over edges leaving v?</th>
</tr>
</thead>
<tbody>
<tr>
<td>list of edges</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>adjacency matrix</td>
<td>V^2</td>
<td>1</td>
<td>1</td>
<td>V</td>
</tr>
<tr>
<td>adjacency list</td>
<td>E + V</td>
<td>outdegree(v)</td>
<td>outdegree(v)</td>
<td>outdegree(v)</td>
</tr>
<tr>
<td>adjacency set</td>
<td>E + V</td>
<td>log (outdegree(v))</td>
<td>log (outdegree(v))</td>
<td>outdegree(v)</td>
</tr>
</tbody>
</table>

huge number of vertices, small average vertex degree
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).

```java
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.

**Property.** Visits vertices in increasing distance from s.

---

**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.

---
Digraph BFS application: web crawler

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";
qu.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.
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.

\(\checkmark\) can’t do better than \( V^2 \)
(reduction from boolean matrix multiplication)
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 $G$. 

**Transitive closure**

- **digraph $G$**
  
- **transitive closure $TC(G)$**

- **Table:**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- **Graph:**

  - **digraph $G$ is usually sparse**
  - **$TC(G)$ is usually dense**
Digraph-processing challenge 1 (revised)

**Problem.** Is there a **directed** path from $v$ to $w$?

**Goals.** $\sim V^2$ 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.
Digraph-processing challenge 1 (revised again)

Problem. Is there a directed path from $v$ to $w$?

Goals. $\sim V E$ preprocessing time, $\sim V^2$ 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.
Transitive closure: Java implementation

Use an array of `DFSearcher` objects, one for each row of transitive closure.

```java
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
**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 $v \rightarrow w$ if task $v$ must precede task $w$.

0. read programming assignment
1. download files
2. write code
3. attend precept
...
12. sleep
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

\[
\begin{align*}
0 &\rightarrow 1 \\
0 &\rightarrow 6 \\
0 &\rightarrow 2 \\
0 &\rightarrow 5 \\
2 &\rightarrow 3 \\
4 &\rightarrow 9 \\
6 &\rightarrow 4 \\
6 &\rightarrow 9 \\
7 &\rightarrow 6 \\
8 &\rightarrow 7 \\
9 &\rightarrow 10 \\
9 &\rightarrow 11 \\
9 &\rightarrow 12 \\
11 &\rightarrow 12
\end{align*}
\]
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;  }
    }
}
Topological sort in a DAG: trace

*Visit means call tsort() and leave means return from tsort.*

<table>
<thead>
<tr>
<th>marked[]</th>
<th>sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>visit 0:</td>
<td>1 0 0 0 0 0 0 -</td>
</tr>
<tr>
<td>visit 1:</td>
<td>1 1 0 0 0 0 0 -</td>
</tr>
<tr>
<td>visit 4:</td>
<td>1 1 0 0 1 0 0 -</td>
</tr>
<tr>
<td>leave 4:</td>
<td>1 1 0 0 1 0 0 4</td>
</tr>
<tr>
<td>leave 1:</td>
<td>1 1 0 0 1 0 0 4 1</td>
</tr>
<tr>
<td>visit 2:</td>
<td>1 1 1 0 1 0 0 4 1</td>
</tr>
<tr>
<td>leave 2:</td>
<td>1 1 1 0 1 0 0 4 1 2</td>
</tr>
<tr>
<td>visit 5:</td>
<td>1 1 1 0 1 1 0 4 1 2</td>
</tr>
<tr>
<td>check 2:</td>
<td>1 1 1 0 1 1 0 4 1 2</td>
</tr>
<tr>
<td>leave 5:</td>
<td>1 1 1 0 1 1 0 4 1 2 5</td>
</tr>
<tr>
<td>leave 0:</td>
<td>1 1 1 0 1 1 0 4 1 2 5 0</td>
</tr>
<tr>
<td>check 1:</td>
<td>1 1 1 0 1 1 0 4 1 2 5 0</td>
</tr>
<tr>
<td>check 2:</td>
<td>1 1 1 0 1 1 0 4 1 2 5 0</td>
</tr>
<tr>
<td>visit 3:</td>
<td>1 1 1 1 1 1 0 4 1 2 5 0</td>
</tr>
<tr>
<td>check 2:</td>
<td>1 1 1 1 1 1 0 4 1 2 5 0</td>
</tr>
<tr>
<td>check 4:</td>
<td>1 1 1 1 1 1 0 4 1 2 5 0</td>
</tr>
<tr>
<td>check 5:</td>
<td>1 1 1 1 1 1 0 4 1 2 5 0</td>
</tr>
<tr>
<td>visit 6:</td>
<td>1 1 1 1 1 1 1 4 1 2 5 0 6</td>
</tr>
<tr>
<td>leave 6:</td>
<td>1 1 1 1 1 1 1 4 1 2 5 0 6</td>
</tr>
<tr>
<td>leave 3:</td>
<td>1 1 1 1 1 1 1 4 1 2 5 0 6 3</td>
</tr>
<tr>
<td>check 4:</td>
<td>1 1 1 1 1 1 0 4 1 2 5 0 6 3</td>
</tr>
<tr>
<td>check 5:</td>
<td>1 1 1 1 1 1 0 4 1 2 5 0 6 3</td>
</tr>
<tr>
<td>check 6:</td>
<td>1 1 1 1 1 1 0 4 1 2 5 0 6 3</td>
</tr>
</tbody>
</table>
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 $v \rightarrow w$. When $\text{tsort}(G, v)$ is called,
  - Case 1: $\text{tsort}(G, w)$ has already been called and returned. Thus, $w$ will appear after $v$ in topological order.
  - Case 2: $\text{tsort}(G, w)$ has not yet been called, so it will get called directly or indirectly by $\text{tsort}(G, v)$ and it will finish before $\text{tsort}(G, v)$. Thus, $w$ will appear after $v$ in topological order.
  - Case 3: $\text{tsort}(G, w)$ 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 $v \rightarrow w$ yields a directed cycle, contradicting DAG.
Digraph-processing challenge 2b

**Problem.** Given a digraph, is there a directed cycle?

**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.

---

run DFS-based topological sort algorithm; if it yields a topological sort, no directed cycle (can modify code to find cycle)

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.

```java
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.

```plaintext
% 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 $\text{time}[v]$ 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.

Ex.

<table>
<thead>
<tr>
<th>index</th>
<th>task</th>
<th>time</th>
<th>prereqs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>begin</td>
<td>0</td>
<td>–</td>
</tr>
<tr>
<td>B</td>
<td>framing</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>roofing</td>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>D</td>
<td>siding</td>
<td>6</td>
<td>B</td>
</tr>
<tr>
<td>E</td>
<td>windows</td>
<td>5</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td>plumbing</td>
<td>3</td>
<td>D</td>
</tr>
<tr>
<td>G</td>
<td>electricity</td>
<td>4</td>
<td>C, E</td>
</tr>
<tr>
<td>H</td>
<td>paint</td>
<td>6</td>
<td>C, E</td>
</tr>
<tr>
<td>I</td>
<td>finish</td>
<td>0</td>
<td>F, H</td>
</tr>
</tbody>
</table>
Program Evaluation and Review Technique / Critical Path Method

**PERT/CPM algorithm.**
- Compute topological order of vertices.
- Initialize $\text{fin}[v] = \text{time}[v]$ for all vertices $v$.
- Consider vertices $v$ in topologically sorted order.
  - for each edge $v \rightarrow w$, set $\text{fin}[w] = \max(\text{fin}[w], \text{fin}[v] + \text{time}[w])$
PERT/CPM algorithm.

- Compute topological order of vertices.
- Initialize $\text{fin}[v] = \text{time}[v]$ for all vertices $v$.
- Consider vertices $v$ in topologically sorted order.
  - for each edge $v\rightarrow w$, set $\text{fin}[w] = \max(\text{fin}[w], \text{fin}[v] + \text{time}[w])$
Program Evaluation and Review Technique / Critical Path Method

**PERT/CPM algorithm.**

- Compute topological order of vertices.
- Initialize $\text{fin}[v] = \text{time}[v]$ for all vertices $v$.
- Consider vertices $v$ in topologically sorted order.
  - for each edge $v \rightarrow w$, set $\text{fin}[w] = \max(\text{fin}[w], \text{fin}[v] + \text{time}[w])$
Program Evaluation and Review Technique / Critical Path Method

**PERT/CPM algorithm.**

- Compute topological order of vertices.
- Initialize $\text{fin}[v] = \text{time}[v]$ for all vertices $v$.
- Consider vertices $v$ in topologically sorted order.
  - for each edge $v \rightarrow w$, set $\text{fin}[w] = \max(\text{fin}[w], \text{fin}[v] + \text{time}[w])$
Program Evaluation and Review Technique / Critical Path Method

**PERT/CPM algorithm.**

- **Compute topological order of vertices.**
- **Initialize** \( \text{fin}[v] = \text{time}[v] \) for all vertices \( v \).
- **Consider vertices** \( v \) in topologically sorted order.
  - for each edge \( v \rightarrow w \), set \( \text{fin}[w] = \max(\text{fin}[w], \text{fin}[v] + \text{time}[w]) \)
Program Evaluation and Review Technique / Critical Path Method

PERT/CPM algorithm.

- Compute topological order of vertices.
- Initialize $\text{fin}[v] = \text{time}[v]$ for all vertices $v$.
- Consider vertices $v$ in topologically sorted order.
  - for each edge $v \rightarrow w$, set $\text{fin}[w] = \max(\text{fin}[w], \text{fin}[v] + \text{time}[w])$
PERT/CPM algorithm.

- Compute topological order of vertices.
- Initialize $\text{fin}[v] = \text{time}[v]$ for all vertices $v$.
- Consider vertices $v$ in topologically sorted order.
  
  - for each edge $v \rightarrow w$, set $\text{fin}[w] = \max(\text{fin}[w], \text{fin}[v] + \text{time}[w])$
Program Evaluation and Review Technique / Critical Path Method

PERT/CPM algorithm.
• Compute topological order of vertices.
• Initialize $\text{fin}[v] = \text{time}[v]$ for all vertices $v$.
• Consider vertices $v$ in topologically sorted order.
  - for each edge $v \to w$, set $\text{fin}[w] = \max(\text{fin}[w], \text{fin}[v] + \text{time}[w])$
PERT/CPM algorithm.

- Compute topological order of vertices.
- Initialize $\text{fin}[v] = \text{time}[v]$ for all vertices $v$.
- Consider vertices $v$ in topologically sorted order.
  - for each edge $v \rightarrow w$, set $\text{fin}[w] = \max(\text{fin}[w], \text{fin}[v] + \text{time}[w])$
Program Evaluation and Review Technique / Critical Path Method

**PERT/CPM algorithm.**

- Compute topological order of vertices.
- Initialize $\text{fin}[v] = \text{time}[v]$ for all vertices $v$.
- Consider vertices $v$ in topologically sorted order.
  - for each edge $v \rightarrow w$, set $\text{fin}[w] = \max(\text{fin}[w], \text{fin}[v] + \text{time}[w])$
Program Evaluation and Review Technique / Critical Path Method

PERT/CPM algorithm.

- Compute topological order of vertices.
- Initialize $\text{fin}[v] = \text{time}[v]$ for all vertices $v$.
- Consider vertices $v$ in topologically sorted order.
  - for each edge $v \rightarrow w$, set $\text{fin}[w] = \max(\text{fin}[w], \text{fin}[v] + \text{time}[w])$
Critical path. Longest path from source to sink.

To compute:
- Remember vertex that set value (parent-link).
- Work backwards from sink.

<table>
<thead>
<tr>
<th>index</th>
<th>time</th>
<th>prereqs</th>
<th>finish</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>B</td>
<td>6</td>
</tr>
<tr>
<td>D</td>
<td>6</td>
<td>B</td>
<td>10</td>
</tr>
<tr>
<td>E</td>
<td>5</td>
<td>D</td>
<td>15</td>
</tr>
<tr>
<td>F</td>
<td>3</td>
<td>D</td>
<td>13</td>
</tr>
<tr>
<td>G</td>
<td>4</td>
<td>C, E</td>
<td>19</td>
</tr>
<tr>
<td>H</td>
<td>6</td>
<td>C, E</td>
<td>25</td>
</tr>
<tr>
<td>I</td>
<td>0</td>
<td>F, H</td>
<td>25</td>
</tr>
</tbody>
</table>
PERT/CPM: Java implementation

double[] fin = new double[G.V()];
for (int v = 0; v < G.V(); v++)
    fin[v] = time[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]);

G = DAG of precedence constraints

fin[v] = finishing time of task v

apply updates to vertices in topological order
› digraph API
› digraph search
› transitive closure
› topological sort
› strong components
Strongly connected components

**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.
Problem. Are $v$ and $w$ strongly connected?

Goal. 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 (or a COS 423 student).
- Intractable.
- No one knows.
- Impossible.

Implementation: use DFS twice to find strong components (see textbook)

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.

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.
Kosaraju's algorithm

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.

**Proposition.** Trees in second DFS are strong components. (!)

**Pf.** [see COS 423]
## Digraph-processing summary: algorithms of the day

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
<th>Method(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-source reachability</td>
<td>-</td>
<td>DFS</td>
</tr>
<tr>
<td>Transitive closure</td>
<td>-</td>
<td>DFS (from each vertex)</td>
</tr>
<tr>
<td>Topological sort (DAG)</td>
<td>-</td>
<td>DFS</td>
</tr>
<tr>
<td>Strong components</td>
<td>-</td>
<td>Kosaraju DFS (twice)</td>
</tr>
</tbody>
</table>
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.

[Diagram of 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.
**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.
**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.

**Brute force.** Try all spanning trees.

spanning tree $T$: cost = $50 = 4 + 6 + 8 + 5 + 11 + 9 + 7$
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).

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 $T$.

Proposition. Both greedy algorithms compute MST.

“Greed is good. Greed is right. Greed works. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit.” — Gordon Gecko
weighted graph API
- cycles and cuts
- Kruskal’s algorithm
- Prim’s algorithm
- advanced topics
Edge API

Edge abstraction needed for weighted edges.

```java
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
```

![Graph representation](image-url)
**Weighted graph API**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>WeightedGraph(int V)</code></td>
<td>create an empty graph with V vertices</td>
</tr>
<tr>
<td><code>WeightedGraph(In in)</code></td>
<td>create a graph from input stream</td>
</tr>
<tr>
<td><code>void addEdge(Edge e)</code></td>
<td>add edge e</td>
</tr>
<tr>
<td><code>void removeEdge(Edge e)</code></td>
<td>delete edge e</td>
</tr>
<tr>
<td><code>Iterable&lt;Edge&gt; adj(int v)</code></td>
<td>return an iterator over edges incident to v</td>
</tr>
<tr>
<td><code>int V()</code></td>
<td>return number of vertices</td>
</tr>
</tbody>
</table>

**Conventions.**

- Allow self-loops.
- Allow parallel edges (provided they have different weights).
Weighted graph API

```java
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
}
```

```java
for (int v = 0; v < G.V(); v++)
{
    for (Edge e : G.adj(v))
    {
        int w = e.other(v);
        // process edge v-w
    }
}
```
Weighted graph: adjacency-set implementation

```java
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**
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.
}

Weighted edge: Java implementation
constructor
either endpoint
other endpoint
weight of edge
Weighted edge: Java implementation (cont)

```java
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;
    }
}

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;
}
```

- Order edges by weight (for sorting in Kruskal).
- Lexicographic order, breaking ties by weight (for use in a symbol table).
• weighted graph API
• cycles and cuts
• Kruskal’s algorithm
• Prim’s algorithm
• advanced topics
Cycle and cut properties

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 $e$. 

![Diagram](image-url)

- $f$ 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 $w_e < w_f$, $\text{weight}(T) < \text{weight}(T^*)$.
- 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 \( T^* \). □
- weighted graph API
- cycles and cuts
- Kruskal’s algorithm
- Prim’s algorithm
- advanced topics
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.
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 $C$.
- Edge $e$ is not in the MST (cycle property).
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 $S$.
- Edge $e$ is in the MST (cut property).

\[ \text{why min weight?} \]
Problem. Check if adding an edge \( v-w \) to \( T \) creates a cycle.

How difficult?
- \( O(E + V) \) time.
- \( O(V) \) time.
- \( O(\log V) \) time.
- \( O(\log^* V) \) time.
- Constant time.

run DFS from \( v \), check if \( w \) is reachable (\( T \) has at most \( V-1 \) edges)

use the union-find data structure!
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

```java
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 the 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.**

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
<th>time per op</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
<td>1</td>
<td>$E \log E$</td>
</tr>
<tr>
<td>union</td>
<td>$V$</td>
<td>$\log^* V$</td>
</tr>
<tr>
<td>find</td>
<td>$E$</td>
<td>$\log^* V$</td>
</tr>
</tbody>
</table>

† 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: $\log^* V \leq 5$ in this universe
Kruskal’s algorithm example

25%

75%

50%

100%
‣ weighted graph API
‣ cycles and cuts
‣ Kruskal’s algorithm
‣ **Prim’s algorithm**
‣ advanced topics
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.
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). □
Problem. Find min weight edge with exactly one endpoint in S.

How difficult?
- $O(E)$ time.  
- $O(V)$ time.
- $O(\log E)$ time.
- $O(\log^* E)$ time.
- Constant time.

Prim implementation challenge

try all edges
use a priority queue!
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)

black = PQ edge with exactly one endpoint in S, sorted by weight
gray = PQ edge with both endpoints in S (obsolete)
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.
}
Lazy implementation of Prim's algorithm

```java
private void scan(WeightedGraph G, int v) {
    scanned[v] = true;
    for (Edge e : G.adj(v))
        if (!scanned[e.other(v)])
            pq.insert(e);
}

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);
    }
}
```

- For each edge v-w, add to PQ if w not already in S.
- 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.
Proposition. Prim's algorithm computes MST in \(O(E \log E)\) time.

**Pf.**

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
<th>time per op</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete min</td>
<td>(E)</td>
<td>(E \log E)</td>
</tr>
<tr>
<td>insert</td>
<td>(E)</td>
<td>(E \log E)</td>
</tr>
</tbody>
</table>

**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.

```java
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;
}
```

**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**

<table>
<thead>
<tr>
<th>year</th>
<th>worst case</th>
<th>discovered by</th>
</tr>
</thead>
<tbody>
<tr>
<td>1975</td>
<td>$E \log \log V$</td>
<td>Yao</td>
</tr>
<tr>
<td>1976</td>
<td>$E \log \log V$</td>
<td>Cheriton-Tarjan</td>
</tr>
<tr>
<td>1984</td>
<td>$E \log^* V, E + V \log V$</td>
<td>Fredman-Tarjan</td>
</tr>
<tr>
<td>1986</td>
<td>$E \log (\log^* V)$</td>
<td>Gabow-Galil-Spencer-Tarjan</td>
</tr>
<tr>
<td>1997</td>
<td>$E \alpha(V) \log \alpha(V)$</td>
<td>Chazelle</td>
</tr>
<tr>
<td>2000</td>
<td>$E \alpha(V)$</td>
<td>Chazelle</td>
</tr>
<tr>
<td>2002</td>
<td>optimal</td>
<td>Pettie-Ramachandran</td>
</tr>
<tr>
<td>20xx</td>
<td>$E$</td>
<td>???</td>
</tr>
</tbody>
</table>

**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 $\sim N^2/2$ distances and run Prim's algorithm.

**Ingenuity.** Exploit geometry and do it in $\sim c N \log N$. 
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.

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. 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.** 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
4.4 Shortest Paths

- Dijkstra's algorithm
- implementation
- negative weights

References:  Algorithms in Java, 3rd edition, Chapter 21
Shortest paths in a weighted digraph

Given a weighted digraph $G$, find the shortest directed path from $s$ to $t$. 

shortest path: $s \rightarrow 6 \rightarrow 3 \rightarrow 5 \rightarrow t$

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


Ford (1956). RAND, economics of transportation.


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.

› Dijkstra's algorithm
› implementation
› negative weights
“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.”
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.
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 $s$, $\text{dist}[s]$ 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 $\text{dist}[v] + e \cdot \text{weight}()$.
  - set $\text{dist}[w] = \text{dist}[v] + e \cdot \text{weight}()$ and $\text{pred}[w] = e$
  - add $w$ to $S$
Dijkstra's algorithm example

- **Edge with v in S and w not in S**
- **Edge in shortest path tree**

0→1 (0.41)
0→5 (0.29)
1→2 (0.51)
1→4 (0.32)
2→3 (0.50)
3→0 (0.45)
3→5 (0.38)
4→2 (0.32)
4→3 (0.36)
5→1 (0.29)
5→4 (0.21)

- 0→5 (0.29)
- 0→1 (0.41)
- 5→4 (0.50 = 0.29 + 0.21)
- 1→2 (0.92 = 0.41 + 0.51)

- 4→2 (0.82 = 0.50 + 0.32)
- 4→3 (0.86 = 0.50 + 0.36)
- 1→2 (0.92)
Dijkstra's algorithm: correctness proof

**Invariant.** For $v$ in $S$, $\text{dist}[v]$ 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 $s \xrightarrow{} w$ path through $v$.
- Consider any other $s \xrightarrow{} w$ 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, $\text{dist}[w]$ is the length of the shortest path from $s$ to $w$. 

![Diagram](image-url)
Remark. Dijkstra examines vertices in increasing distance from source.
Dijkstra's algorithm
implementation
negative weights
# Weighted directed graph API

**public class DirectedEdge** implements Comparable<DirectedEdge>  

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DirectedEdge(int v, int w, double weight)</td>
<td>create a weighted edge v→w</td>
</tr>
<tr>
<td>int from()</td>
<td>vertex v</td>
</tr>
<tr>
<td>int to()</td>
<td>vertex w</td>
</tr>
<tr>
<td>double weight()</td>
<td>the weight</td>
</tr>
</tbody>
</table>

**public class WeightedDigraph**  

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WeightedDigraph(int V)</td>
<td>create an empty digraph with V vertices</td>
</tr>
<tr>
<td>WeightedDigraph(In in)</td>
<td>create a digraph from input stream</td>
</tr>
<tr>
<td>void addEdge(DirectedEdge e)</td>
<td>add a weighted edge from v to w</td>
</tr>
<tr>
<td>Iterable&lt;DirectedEdge&gt; adj(int v)</td>
<td>return an iterator over edges leaving v</td>
</tr>
<tr>
<td>int V()</td>
<td>return number of vertices</td>
</tr>
</tbody>
</table>
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;  }
}
Weighted directed edge: implementation in Java

```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;
Dijkstra 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 $\text{dist}[v] + e\cdot\text{weight}()$. 

How difficult?
- Intractable.
- $O(E)$ time.
- $O(V)$ time.
- $O(\log E)$ time.
- $O(\log^* E)$ time.
- Constant time.
Lazy Dijkstra’s algorithm example

```
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)
1→4 (.73)
4→2 (.82 = .50 + .32)
4→3 (.86 = .50 + .36)
1→2 (.92)
4→3 (.86)
1→2 (.92)
2→3 (1.32 = .82 + .50)
2→3 (1.32)
```

priority queue
Lazy implementation of Dijkstra's algorithm

```java
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
\[ \text{dist}[v] + \text{e.weight()} \]
Lazy implementation of Dijkstra's algorithm

```java
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);
    }
}

private void scan(WeightedDigraph G, int v) {
    scanned[v] = true;
    for (DirectedEdge e : G.adj(v))
        if (!scanned[e.to()]) pq.insert(e);
}
```

- both endpoints in S
- found shortest path to w
- add all edges v→w to pq, provided w not already in S
Proposition. Dijkstra's algorithm computes shortest paths in $O(E \log E)$ time.

Pf.

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
<th>time per op</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete min</td>
<td>$E$</td>
<td>$\log E$</td>
</tr>
<tr>
<td>insert</td>
<td>$E$</td>
<td>$\log E$</td>
</tr>
</tbody>
</table>

Improvements.

- Eagerly eliminate obsolete edges from PQ.
- Maintain on PQ at most one edge incident to each vertex $v$ not in $T$ ⇒ at most $V$ edges on PQ.
- Use fancier priority queue: best in theory yields $O(E + V \log V)$. 

Dijkstra's algorithm running time
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
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.
- 1 oz. gold ⇒ 455.2 Francs ⇒ 304.39 Euros ⇒ $327.28.

<table>
<thead>
<tr>
<th>currency</th>
<th>£</th>
<th>Euro</th>
<th>¥</th>
<th>Franc</th>
<th>$</th>
<th>Gold</th>
</tr>
</thead>
<tbody>
<tr>
<td>UK pound</td>
<td>1.0000</td>
<td>0.6853</td>
<td>0.005290</td>
<td>0.4569</td>
<td>0.6368</td>
<td>208.100</td>
</tr>
<tr>
<td>Euro</td>
<td>1.45999</td>
<td>1.0000</td>
<td>0.007721</td>
<td>0.6677</td>
<td>0.9303</td>
<td>304.028</td>
</tr>
<tr>
<td>Japanese Yen</td>
<td>189.50</td>
<td>129.520</td>
<td>1.0000</td>
<td>85.4694</td>
<td>120.400</td>
<td>39346.7</td>
</tr>
<tr>
<td>Swiss Franc</td>
<td>2.1904</td>
<td>1.4978</td>
<td>0.01574</td>
<td>1.0000</td>
<td>1.3941</td>
<td>455.200</td>
</tr>
<tr>
<td>US dollar</td>
<td>1.5714</td>
<td>1.0752</td>
<td>0.008309</td>
<td>0.7182</td>
<td>1.0000</td>
<td>327.250</td>
</tr>
<tr>
<td>Gold (oz.)</td>
<td>0.004816</td>
<td>0.003295</td>
<td>0.0000255</td>
<td>0.002201</td>
<td>0.003065</td>
<td>1.0000</td>
</tr>
</tbody>
</table>
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 \( v \rightarrow w \) 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 $w$.

- $\text{dist}[v]$ is length of some path from $s$ to $v$.
- $\text{dist}[w]$ is length of some path from $s$ to $w$.
- If $v \rightarrow w$ gives a shorter path to $w$ through $v$, update $\text{dist}[w]$ and $\text{pred}[w]$.

```java
int v = e.from(), w = e.to();
if (dist[w] > dist[v] + e.weight())
{
    dist[w] = dist[v] + e.weight();
    pred[w] = e;
}
```
A simple solution that works!

- Initialize $\text{dist}[v] = \infty$, $\text{dist}[s] = 0$.
- Repeat $v$ times: relax each edge $e$.

```java
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;
            }
        }
```
Dynamic programming algorithm trace

relaxed edges that update \( \text{dist}[v] \)

relaxed edges that update \( \text{dist}[] \)

can stop early since no entries in \( \text{dist}[] \) updated
Dynamic programming algorithm: analysis

Running time. Proportional to $E V$.

Invariant. At end of phase $i$, $\text{dist}[v] \leq$ length of any path from $s$ to $v$ using at most $i$ edges.

Proposition. If there are no negative cycles, upon termination $\text{dist}[v]$ is the length of the shortest path from $s$ to $v$. and pred[] gives the shortest paths.
**Bellman-Ford-Moore algorithm**

*Observation.* If $\text{dist}[v]$ 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.

\[
\uparrow \quad \text{be careful to keep at most one copy of each vertex on queue}
\]

**Running time.**
- Proportional to $EV$ in worst case.
- Much faster than that in practice.
### Remark 1.
Negative weights makes the problem harder.

### Remark 2.
Negative cycles makes the problem intractable.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>worst case</th>
<th>typical case</th>
</tr>
</thead>
<tbody>
<tr>
<td>nonnegative costs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dijkstra (binary heap)</td>
<td>$E \log E$</td>
<td>$E$</td>
</tr>
<tr>
<td>dynamic programming</td>
<td>$E V$</td>
<td>$E V$</td>
</tr>
<tr>
<td>Bellman-Ford</td>
<td>$E V$</td>
<td>$E$</td>
</tr>
<tr>
<td>no negative cycles</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

![Diagram](image)

**Proposition.** If any vertex $v$ is updated in phase $v$, there exists a negative cycle, and we can trace back $\text{pred}[v]$ to find it.
**Goal.** Identify a negative cycle (reachable from any vertex).

**Solution.** Initialize Bellman-Ford by setting $\text{dist}[v] = 0$ 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

- 5.1 Sorting Strings
- 5.2 String Symbol Tables
- 5.3 Substring Search
- 5.4 Pattern Matching
- 5.5 Data Compression
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.

**Java char data type.** A 16-bit unsigned integer.

- Supports original 16-bit Unicode.
- Awkwardly supports 21-bit Unicode 3.0.

### Hexadecimal to ASCII conversion table

<table>
<thead>
<tr>
<th>Hex</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
</tr>
<tr>
<td>1</td>
<td>SOH</td>
</tr>
<tr>
<td>2</td>
<td>STX</td>
</tr>
<tr>
<td>3</td>
<td>ETX</td>
</tr>
<tr>
<td>4</td>
<td>EOT</td>
</tr>
<tr>
<td>5</td>
<td>ENQ</td>
</tr>
<tr>
<td>6</td>
<td>ACK</td>
</tr>
<tr>
<td>7</td>
<td>BEL</td>
</tr>
<tr>
<td>8</td>
<td>BS</td>
</tr>
<tr>
<td>9</td>
<td>HT</td>
</tr>
<tr>
<td>A</td>
<td>LF</td>
</tr>
<tr>
<td>B</td>
<td>VT</td>
</tr>
<tr>
<td>C</td>
<td>FF</td>
</tr>
<tr>
<td>D</td>
<td>CR</td>
</tr>
<tr>
<td>E</td>
<td>SO</td>
</tr>
<tr>
<td>F</td>
<td>SI</td>
</tr>
<tr>
<td>10</td>
<td>DLE</td>
</tr>
<tr>
<td>11</td>
<td>DC1</td>
</tr>
<tr>
<td>12</td>
<td>DC2</td>
</tr>
<tr>
<td>13</td>
<td>DC3</td>
</tr>
<tr>
<td>14</td>
<td>DC4</td>
</tr>
<tr>
<td>15</td>
<td>NA</td>
</tr>
<tr>
<td>16</td>
<td>SYN</td>
</tr>
<tr>
<td>17</td>
<td>ETB</td>
</tr>
<tr>
<td>18</td>
<td>CAN</td>
</tr>
<tr>
<td>19</td>
<td>EM</td>
</tr>
<tr>
<td>20</td>
<td>SUB</td>
</tr>
<tr>
<td>21</td>
<td>ESC</td>
</tr>
<tr>
<td>22</td>
<td>FS</td>
</tr>
<tr>
<td>23</td>
<td>GS</td>
</tr>
<tr>
<td>24</td>
<td>RS</td>
</tr>
<tr>
<td>25</td>
<td>US</td>
</tr>
<tr>
<td>26</td>
<td>SP</td>
</tr>
<tr>
<td>27</td>
<td>!</td>
</tr>
<tr>
<td>28</td>
<td>#</td>
</tr>
<tr>
<td>29</td>
<td>$</td>
</tr>
<tr>
<td>30</td>
<td>%</td>
</tr>
<tr>
<td>31</td>
<td>&amp;</td>
</tr>
<tr>
<td>32</td>
<td>'</td>
</tr>
<tr>
<td>33</td>
<td>(</td>
</tr>
<tr>
<td>34</td>
<td>)</td>
</tr>
<tr>
<td>35</td>
<td>*</td>
</tr>
<tr>
<td>36</td>
<td>+</td>
</tr>
<tr>
<td>37</td>
<td>,</td>
</tr>
<tr>
<td>38</td>
<td>-</td>
</tr>
<tr>
<td>39</td>
<td>.</td>
</tr>
<tr>
<td>40</td>
<td>/</td>
</tr>
<tr>
<td>41</td>
<td>0</td>
</tr>
<tr>
<td>42</td>
<td>1</td>
</tr>
<tr>
<td>43</td>
<td>2</td>
</tr>
<tr>
<td>44</td>
<td>3</td>
</tr>
<tr>
<td>45</td>
<td>4</td>
</tr>
<tr>
<td>46</td>
<td>5</td>
</tr>
<tr>
<td>47</td>
<td>6</td>
</tr>
<tr>
<td>48</td>
<td>7</td>
</tr>
<tr>
<td>49</td>
<td>8</td>
</tr>
<tr>
<td>50</td>
<td>9</td>
</tr>
<tr>
<td>51</td>
<td>:</td>
</tr>
<tr>
<td>52</td>
<td>;</td>
</tr>
<tr>
<td>53</td>
<td>&lt;</td>
</tr>
<tr>
<td>54</td>
<td>=</td>
</tr>
<tr>
<td>55</td>
<td>&gt;</td>
</tr>
<tr>
<td>56</td>
<td>?</td>
</tr>
<tr>
<td>57</td>
<td>@</td>
</tr>
<tr>
<td>58</td>
<td>A</td>
</tr>
<tr>
<td>59</td>
<td>B</td>
</tr>
<tr>
<td>60</td>
<td>C</td>
</tr>
<tr>
<td>61</td>
<td>D</td>
</tr>
<tr>
<td>62</td>
<td>E</td>
</tr>
<tr>
<td>63</td>
<td>F</td>
</tr>
<tr>
<td>64</td>
<td>G</td>
</tr>
<tr>
<td>65</td>
<td>H</td>
</tr>
<tr>
<td>66</td>
<td>I</td>
</tr>
<tr>
<td>67</td>
<td>J</td>
</tr>
<tr>
<td>68</td>
<td>K</td>
</tr>
<tr>
<td>69</td>
<td>L</td>
</tr>
<tr>
<td>70</td>
<td>M</td>
</tr>
<tr>
<td>71</td>
<td>N</td>
</tr>
<tr>
<td>72</td>
<td>O</td>
</tr>
<tr>
<td>73</td>
<td>P</td>
</tr>
<tr>
<td>74</td>
<td>Q</td>
</tr>
<tr>
<td>75</td>
<td>R</td>
</tr>
<tr>
<td>76</td>
<td>S</td>
</tr>
<tr>
<td>77</td>
<td>T</td>
</tr>
<tr>
<td>78</td>
<td>U</td>
</tr>
<tr>
<td>79</td>
<td>V</td>
</tr>
<tr>
<td>80</td>
<td>W</td>
</tr>
<tr>
<td>81</td>
<td>X</td>
</tr>
<tr>
<td>82</td>
<td>Y</td>
</tr>
<tr>
<td>83</td>
<td>Z</td>
</tr>
<tr>
<td>84</td>
<td>[</td>
</tr>
<tr>
<td>85</td>
<td>\</td>
</tr>
<tr>
<td>86</td>
<td>]</td>
</tr>
<tr>
<td>87</td>
<td>^</td>
</tr>
<tr>
<td>88</td>
<td>_</td>
</tr>
<tr>
<td>89</td>
<td>`</td>
</tr>
<tr>
<td>90</td>
<td>a</td>
</tr>
<tr>
<td>91</td>
<td>b</td>
</tr>
<tr>
<td>92</td>
<td>c</td>
</tr>
<tr>
<td>93</td>
<td>d</td>
</tr>
<tr>
<td>94</td>
<td>e</td>
</tr>
<tr>
<td>95</td>
<td>f</td>
</tr>
<tr>
<td>96</td>
<td>g</td>
</tr>
<tr>
<td>97</td>
<td>h</td>
</tr>
<tr>
<td>98</td>
<td>i</td>
</tr>
<tr>
<td>99</td>
<td>j</td>
</tr>
<tr>
<td>100</td>
<td>k</td>
</tr>
<tr>
<td>101</td>
<td>l</td>
</tr>
<tr>
<td>102</td>
<td>m</td>
</tr>
<tr>
<td>103</td>
<td>n</td>
</tr>
<tr>
<td>104</td>
<td>o</td>
</tr>
<tr>
<td>105</td>
<td>p</td>
</tr>
<tr>
<td>106</td>
<td>q</td>
</tr>
<tr>
<td>107</td>
<td>r</td>
</tr>
<tr>
<td>108</td>
<td>s</td>
</tr>
<tr>
<td>109</td>
<td>t</td>
</tr>
<tr>
<td>110</td>
<td>u</td>
</tr>
<tr>
<td>111</td>
<td>v</td>
</tr>
<tr>
<td>112</td>
<td>w</td>
</tr>
<tr>
<td>113</td>
<td>x</td>
</tr>
<tr>
<td>114</td>
<td>y</td>
</tr>
<tr>
<td>115</td>
<td>z</td>
</tr>
<tr>
<td>116</td>
<td>{</td>
</tr>
<tr>
<td>117</td>
<td></td>
</tr>
<tr>
<td>118</td>
<td>}</td>
</tr>
<tr>
<td>119</td>
<td>~</td>
</tr>
<tr>
<td>120</td>
<td>DEL</td>
</tr>
</tbody>
</table>
The String data type

**Character extraction.** Get the i\textsuperscript{th} character.

**Substring extraction.** Get a contiguous sequence of characters from a string.

**String concatenation.** Append one character to end of another string.

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.

```java
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
}
```
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

<table>
<thead>
<tr>
<th>operation</th>
<th>guarantee</th>
<th>extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td>charAt()</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>substring()</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>concat()</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
String VS. StringBuilder

**String.** [immutable] Constant substring, linear concatenation.

**StringBuilder.** [mutable] Linear substring, constant (amortized) append.

**Ex.** Reverse a String.

```java
public static String reverse(String s)
{
    String rev = "";
    for (int i = s.length() - 1; i >= 0; i--)
        rev += s.charAt(i);
    return rev;
}
```

```java
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();
}
```

quadratic time

linear time
String challenge: array of suffixes

**Challenge.** How to efficiently form array of suffixes?

<table>
<thead>
<tr>
<th>input string</th>
</tr>
</thead>
<tbody>
<tr>
<td>a a c a a g t t t a c a a g c</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>suffixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 a a c a a g t t t a c a a g c</td>
</tr>
<tr>
<td>1 a c a a g t t t a c a a g c</td>
</tr>
<tr>
<td>2 c a a g t t t a c a a g c</td>
</tr>
<tr>
<td>3 a a g t t t a c a a g c</td>
</tr>
<tr>
<td>4 a g t t t a c a a g c</td>
</tr>
<tr>
<td>5 g t t t a c a a g c</td>
</tr>
<tr>
<td>6 t t t a c a a g c</td>
</tr>
<tr>
<td>7 t t a c a a g c</td>
</tr>
<tr>
<td>8 t a c a a g c</td>
</tr>
<tr>
<td>9 a c a a g c</td>
</tr>
<tr>
<td>10 c a a g c</td>
</tr>
<tr>
<td>11 a a g c</td>
</tr>
<tr>
<td>12 a g c</td>
</tr>
<tr>
<td>13 g c</td>
</tr>
<tr>
<td>14 c</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>name</th>
<th>R()</th>
<th>IgR()</th>
<th>characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY</td>
<td>2</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>OCTAL</td>
<td>8</td>
<td>3</td>
<td>01234567</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>10</td>
<td>4</td>
<td>0123456789</td>
</tr>
<tr>
<td>HEXADECIMAL</td>
<td>16</td>
<td>4</td>
<td>0123456789ABCDEF</td>
</tr>
<tr>
<td>DNA</td>
<td>4</td>
<td>2</td>
<td>ACTG</td>
</tr>
<tr>
<td>LOWERCASE</td>
<td>26</td>
<td>5</td>
<td>abcdefghijklmnopqrstuvwxyz</td>
</tr>
<tr>
<td>UPPERCASE</td>
<td>26</td>
<td>5</td>
<td>ABCDEFGHIJKLMNOPQRSTUVWXYZ</td>
</tr>
<tr>
<td>PROTEIN</td>
<td>20</td>
<td>5</td>
<td>ACDEFGHIJKLMNOPQRSTUVWXYZ</td>
</tr>
<tr>
<td>BASE64</td>
<td>64</td>
<td>6</td>
<td>ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/</td>
</tr>
<tr>
<td>ASCII</td>
<td>128</td>
<td>7</td>
<td>ASCII characters</td>
</tr>
<tr>
<td>EXTENDED_ASCII</td>
<td>256</td>
<td>8</td>
<td>extended ASCII characters</td>
</tr>
<tr>
<td>UNICODE16</td>
<td>65536</td>
<td>16</td>
<td>Unicode characters</td>
</tr>
</tbody>
</table>

*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.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>guarantee</th>
<th>random</th>
<th>extra space</th>
<th>stable?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion sort</td>
<td>$N^2/2$</td>
<td>$N^2/4$</td>
<td>no</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>mergesort</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>$N$</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>quicksort</td>
<td>$1.39 \times N \log N$ *</td>
<td>$1.39 \times N \log N$</td>
<td>$c \log N$</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>heapsort</td>
<td>$2 \times N \log N$</td>
<td>$2 \times N \log N$</td>
<td>no</td>
<td>no</td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

* probabilistic

Lower bound. $\sim N \log N$ 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.

<table>
<thead>
<tr>
<th>name</th>
<th>section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anderson</td>
<td>2</td>
</tr>
<tr>
<td>Brown</td>
<td>3</td>
</tr>
<tr>
<td>Davis</td>
<td>3</td>
</tr>
<tr>
<td>Garcia</td>
<td>4</td>
</tr>
<tr>
<td>Harris</td>
<td>1</td>
</tr>
<tr>
<td>Jackson</td>
<td>3</td>
</tr>
<tr>
<td>Johnson</td>
<td>4</td>
</tr>
<tr>
<td>Jones</td>
<td>3</td>
</tr>
<tr>
<td>Martin</td>
<td>1</td>
</tr>
<tr>
<td>Martinez</td>
<td>2</td>
</tr>
<tr>
<td>Miller</td>
<td>2</td>
</tr>
<tr>
<td>Moore</td>
<td>1</td>
</tr>
<tr>
<td>Robinson</td>
<td>2</td>
</tr>
<tr>
<td>Smith</td>
<td>4</td>
</tr>
<tr>
<td>Taylor</td>
<td>3</td>
</tr>
<tr>
<td>Thomas</td>
<td>4</td>
</tr>
<tr>
<td>Thompson</td>
<td>4</td>
</tr>
<tr>
<td>White</td>
<td>2</td>
</tr>
<tr>
<td>Williams</td>
<td>3</td>
</tr>
<tr>
<td>Wilson</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Harris</td>
<td>1</td>
</tr>
<tr>
<td>Martin</td>
<td>1</td>
</tr>
<tr>
<td>Moore</td>
<td>1</td>
</tr>
<tr>
<td>Anderson</td>
<td>2</td>
</tr>
<tr>
<td>Martinez</td>
<td>2</td>
</tr>
<tr>
<td>Miller</td>
<td>2</td>
</tr>
<tr>
<td>Jones</td>
<td>3</td>
</tr>
<tr>
<td>Robinson</td>
<td>2</td>
</tr>
<tr>
<td>Taylor</td>
<td>3</td>
</tr>
<tr>
<td>Williams</td>
<td>3</td>
</tr>
<tr>
<td>Wilson</td>
<td>4</td>
</tr>
</tbody>
</table>

Typical candidate for key-indexed counting

Input: names, section

Sorted result: names, section (by section)

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.
- 
- 
- 

```java
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];
```
**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.

```java
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];
```

<table>
<thead>
<tr>
<th>i</th>
<th>a[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>f</td>
</tr>
<tr>
<td>4</td>
<td>f</td>
</tr>
<tr>
<td>5</td>
<td>b</td>
</tr>
<tr>
<td>6</td>
<td>d</td>
</tr>
<tr>
<td>7</td>
<td>b</td>
</tr>
<tr>
<td>8</td>
<td>f</td>
</tr>
<tr>
<td>9</td>
<td>b</td>
</tr>
<tr>
<td>10</td>
<td>e</td>
</tr>
<tr>
<td>11</td>
<td>a</td>
</tr>
</tbody>
</table>

**r count[r]**

| a | 0 |
| b | 2 |
| c | 5 |
| d | 6 |
| e | 8 |
| f | 9 |
|   | 12 |

Key-indexed counting

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.

```java
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];
```
**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.

```java
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];
```
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];
```
**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.

```java
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];
```
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.

```java
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];
```
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.

```java
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];
```

<table>
<thead>
<tr>
<th>i</th>
<th>a[i]</th>
<th>i</th>
<th>aux[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>f</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>f</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>b</td>
<td>5</td>
<td>c</td>
</tr>
<tr>
<td>6</td>
<td>d</td>
<td>6</td>
<td>d</td>
</tr>
<tr>
<td>7</td>
<td>b</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>f</td>
<td>8</td>
<td>f</td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td>9</td>
<td>f</td>
</tr>
<tr>
<td>10</td>
<td>e</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>a</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>
**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.

```java
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];
```
**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.

```java
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];
```

---

**Key-indexed counting**

<table>
<thead>
<tr>
<th>( i )</th>
<th>( a[i] )</th>
<th>( i )</th>
<th>( aux[i] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>f</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>f</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>b</td>
<td>5</td>
<td>c</td>
</tr>
<tr>
<td>6</td>
<td>d</td>
<td>6</td>
<td>d</td>
</tr>
<tr>
<td>7</td>
<td>b</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>f</td>
<td>8</td>
<td>f</td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td>9</td>
<td>f</td>
</tr>
<tr>
<td>10</td>
<td>e</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>a</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>
**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];
```
**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.

```java
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];
```
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.

```java
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];
```
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.

```java
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];
```
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.

```java
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];
```
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.

```java
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];
```

<table>
<thead>
<tr>
<th>i</th>
<th>( a[i] )</th>
<th>i</th>
<th>( aux[i] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>f</td>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>4</td>
<td>f</td>
<td>4</td>
<td>b</td>
</tr>
<tr>
<td>5</td>
<td>b</td>
<td>5</td>
<td>c</td>
</tr>
<tr>
<td>6</td>
<td>d</td>
<td>6</td>
<td>d</td>
</tr>
<tr>
<td>7</td>
<td>b</td>
<td>7</td>
<td>d</td>
</tr>
<tr>
<td>8</td>
<td>f</td>
<td>8</td>
<td>e</td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td>9</td>
<td>f</td>
</tr>
<tr>
<td>10</td>
<td>e</td>
<td>10</td>
<td>f</td>
</tr>
<tr>
<td>11</td>
<td>a</td>
<td>11</td>
<td>f</td>
</tr>
</tbody>
</table>
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.

```java
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];
```
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).

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>d</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>d</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>f</td>
<td>a</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>f</td>
<td>e</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>b</td>
<td>a</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>d</td>
<td>a</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>b</td>
<td>e</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>f</td>
<td>e</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td>e</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>e</td>
<td>b</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>a</td>
<td>c</td>
<td>e</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>d</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>f</td>
<td>a</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>a</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td>a</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>e</td>
<td>b</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td>c</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>a</td>
<td>d</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>f</td>
<td>e</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td>e</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>f</td>
<td>e</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>b</td>
<td>e</td>
<td>e</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>a</th>
<th>c</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>d</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>a</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>e</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>e</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>d</td>
<td>a</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>e</td>
<td>b</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>f</td>
<td>a</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>f</td>
<td>e</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>f</td>
<td>e</td>
<td>e</td>
<td></td>
</tr>
</tbody>
</table>

Sort must be stable (arrows do not cross)
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.
LSD string sort: Java implementation

```java
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

<table>
<thead>
<tr>
<th>Input</th>
<th>d = 6</th>
<th>d = 5</th>
<th>d = 4</th>
<th>d = 3</th>
<th>d = 2</th>
<th>d = 1</th>
<th>d = 0</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>4PGC938</td>
<td>2IYE230</td>
<td>3CI0720</td>
<td>2IYE230</td>
<td>2RLA629</td>
<td>1ICK750</td>
<td>3ATW723</td>
<td>1ICK750</td>
<td>1ICK750</td>
</tr>
<tr>
<td>2IYE230</td>
<td>3CI0720</td>
<td>3CI0720</td>
<td>4JZY524</td>
<td>2RLA629</td>
<td>1ICK750</td>
<td>3CI0720</td>
<td>1ICK750</td>
<td>1ICK750</td>
</tr>
<tr>
<td>3CI0720</td>
<td>1ICK750</td>
<td>3ATW723</td>
<td>2RLA629</td>
<td>4PGC938</td>
<td>4PGC938</td>
<td>3CI0720</td>
<td>1OHV845</td>
<td>1OHV845</td>
</tr>
<tr>
<td>1ICK750</td>
<td>1ICK750</td>
<td>4JZY524</td>
<td>2RLA629</td>
<td>2IYE230</td>
<td>1OHV845</td>
<td>1ICK750</td>
<td>1OHV845</td>
<td>1OHV845</td>
</tr>
<tr>
<td>1OHV845</td>
<td>3CI0720</td>
<td>2RLA629</td>
<td>3CI0720</td>
<td>1ICK750</td>
<td>1OHV845</td>
<td>1ICK750</td>
<td>1OHV845</td>
<td>1OHV845</td>
</tr>
<tr>
<td>4JZY524</td>
<td>3ATW723</td>
<td>2RLA629</td>
<td>3CI0720</td>
<td>1ICK750</td>
<td>1OHV845</td>
<td>2IYE230</td>
<td>2IYE230</td>
<td>2IYE230</td>
</tr>
<tr>
<td>1ICK750</td>
<td>4JZY524</td>
<td>2IYE230</td>
<td>3ATW723</td>
<td>3CI0720</td>
<td>3CI0720</td>
<td>4JZY524</td>
<td>2RLA629</td>
<td>2RLA629</td>
</tr>
<tr>
<td>3CI0720</td>
<td>1OHV845</td>
<td>4PGC938</td>
<td>1ICK750</td>
<td>3CI0720</td>
<td>3CI0720</td>
<td>1OHV845</td>
<td>2RLA629</td>
<td>2RLA629</td>
</tr>
<tr>
<td>10HV845</td>
<td>1OHV845</td>
<td>1OHV845</td>
<td>1ICK750</td>
<td>1OHV845</td>
<td>2RLA629</td>
<td>1OHV845</td>
<td>3ATW723</td>
<td>3ATW723</td>
</tr>
<tr>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>2RLA629</td>
<td>1OHV845</td>
<td>3CI0720</td>
<td>3CI0720</td>
<td>3CI0720</td>
</tr>
<tr>
<td>2RLA629</td>
<td>4PGC938</td>
<td>1OHV845</td>
<td>10HV845</td>
<td>10HV845</td>
<td>3ATW723</td>
<td>4PGC938</td>
<td>3CI0720</td>
<td>3CI0720</td>
</tr>
<tr>
<td>2RLA629</td>
<td>2RLA629</td>
<td>1ICK750</td>
<td>1OHV845</td>
<td>3ATW723</td>
<td>2IYE230</td>
<td>2RLA629</td>
<td>4JZY524</td>
<td>4JZY524</td>
</tr>
<tr>
<td>3ATW723</td>
<td>2RLA629</td>
<td>1ICK750</td>
<td>4PGC938</td>
<td>4JZY524</td>
<td>4JZY524</td>
<td>2RLA629</td>
<td>4PGC938</td>
<td>4PGC938</td>
</tr>
</tbody>
</table>

To sort an array `a[]` of strings that each have exactly `W` characters, we do `W` key-indexed counting sorts: one for each character position, proceeding from right to left.
## Summary of the performance of sorting algorithms

### Frequency of operations.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Guarantee</th>
<th>Random</th>
<th>Extra Space</th>
<th>Stable?</th>
<th>Operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion sort</td>
<td>( N^2 /2 )</td>
<td>( N^2 /4 )</td>
<td>1</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>mergesort</td>
<td>( N \lg N )</td>
<td>( N \lg N )</td>
<td>( N )</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>quicksort</td>
<td>1.39 ( N \lg N ) *</td>
<td>1.39 ( N \lg N )</td>
<td>( c \lg N )</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>heapsort</td>
<td>( 2 N \lg N )</td>
<td>( 2 N \lg N )</td>
<td>1</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>LSD †</td>
<td>( 2 W N )</td>
<td>( 2 W N )</td>
<td>( N + R )</td>
<td>yes</td>
<td>charAt()</td>
</tr>
</tbody>
</table>

* 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.
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)

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

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).

<table>
<thead>
<tr>
<th>0</th>
<th>d</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>f</td>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>f</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>5</td>
<td>b</td>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td>6</td>
<td>d</td>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td>7</td>
<td>b</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>8</td>
<td>f</td>
<td>e</td>
<td>d</td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td>e</td>
<td>d</td>
</tr>
<tr>
<td>10</td>
<td>e</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>11</td>
<td>a</td>
<td>c</td>
<td>e</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>a</th>
<th>d</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>c</td>
<td>e</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>e</td>
<td>d</td>
</tr>
<tr>
<td>5</td>
<td>c</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>6</td>
<td>d</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>7</td>
<td>d</td>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td>8</td>
<td>e</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>9</td>
<td>f</td>
<td>e</td>
<td>d</td>
</tr>
<tr>
<td>10</td>
<td>f</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>11</td>
<td>f</td>
<td>e</td>
<td>d</td>
</tr>
</tbody>
</table>

**count[]**

```
0 1 2 3 4 5 6 7 8 9 10 11
a 0 2 2 0 0 1 0 0 0 0 0 0
b 1 1 1 2 2 2 2 2 2 2 2 2
c 5 5 5 5 5 5 5 5 5 5 5 5
d 6 6 6 6 6 6 6 6 6 6 6 6
e 8 8 8 8 8 8 8 8 8 8 8 8
f 9 9 9 9 9 9 9 9 9 9 9 9
```

Sort these independently (recursive)
MSD string sort: top level trace

use key-indexed counting on first character

count frequencies
transform counts to indices

distribute and copy back

indices at completion of distribute phase

recursively sort subarrays

start of subarray

1 + end of subarray
**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).

```
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 ⇒ no extra work needed.
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);
}
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.

```java
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!

<table>
<thead>
<tr>
<th>Random (sublinear)</th>
<th>Non-random with duplicates (nearly linear)</th>
<th>Worst case (linear)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1EI0402</td>
<td>are</td>
<td>1DNB377</td>
</tr>
<tr>
<td>1HYL490</td>
<td>by</td>
<td>1DNB377</td>
</tr>
<tr>
<td>1ROZ572</td>
<td>sea</td>
<td>1DNB377</td>
</tr>
<tr>
<td>2HXE734</td>
<td>seashells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>2IYE230</td>
<td>seashells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>2XOR846</td>
<td>sells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3CDB573</td>
<td>sells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3CVP720</td>
<td>she</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3IGJ319</td>
<td>she</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3KNA382</td>
<td>shells</td>
<td>1DNB377</td>
</tr>
<tr>
<td>3TAV879</td>
<td>shore</td>
<td>1DNB377</td>
</tr>
<tr>
<td>4CQP781</td>
<td>surely</td>
<td>1DNB377</td>
</tr>
<tr>
<td>4QGI284</td>
<td>the</td>
<td>1DNB377</td>
</tr>
<tr>
<td>4YHV229</td>
<td>the</td>
<td>1DNB377</td>
</tr>
</tbody>
</table>

Characters examined by MSD string sort
### Summary of the performance of sorting algorithms

#### Frequency of operations.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Guarantee</th>
<th>Random</th>
<th>Extra Space</th>
<th>Stable?</th>
<th>Operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion sort</td>
<td>(N^2/2)</td>
<td>(N^2/4)</td>
<td>1</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>Merge sort</td>
<td>(N \lg N)</td>
<td>(N \lg N)</td>
<td>(N)</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>Quick sort</td>
<td>(1.39 , N \lg N) *</td>
<td>(1.39 , N \lg N)</td>
<td>c (\lg N)</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>Heap sort</td>
<td>(2 , N \lg N)</td>
<td>(2 , N \lg N)</td>
<td>1</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>LSD †</td>
<td>(2 , N , W)</td>
<td>(2 , N , W)</td>
<td>(N + R)</td>
<td>yes</td>
<td>charAt()</td>
</tr>
<tr>
<td>MSD ‡</td>
<td>(2 , N , W)</td>
<td>(N , \log_R , N)</td>
<td>(N + D , R)</td>
<td>yes</td>
<td>charAt()</td>
</tr>
</tbody>
</table>

* 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\textsuperscript{th} character.
- Cheaper than R-way partitioning of MSD string sort.
- Need not examine again characters equal to the partitioning char.

```plaintext
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
```

**Diagram:**
- Use first character value to partition into "less", "equal", and "greater" subarrays.
- Recursive calls for 3-way string quicksort (no cutoff for small subarrays).
- Trace of recursive calls for 3-way string quicksort (no cut-off for small subarrays).
- Recursively sort subarrays, excluding first character for "equal" subarray.

- Partitioning element
- Gray bars represent empty subarrays
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

```java
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 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 $2N \ln N$ 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.

**Bottom line.** 3-way string quicksort is the method of choice for sorting strings.
### Summary of the performance of sorting algorithms

#### Frequency of operations.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>guarantee</th>
<th>random</th>
<th>extra space</th>
<th>stable?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion sort</td>
<td>N&lt;sup&gt;2&lt;/sup&gt;/2</td>
<td>N&lt;sup&gt;2&lt;/sup&gt;/4</td>
<td>1</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>mergesort</td>
<td>N lg N</td>
<td>N lg N</td>
<td>N</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>quicksort</td>
<td>1.39 N lg N</td>
<td>1.39 N lg N</td>
<td>c lg N</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>heapsort</td>
<td>2 N lg N</td>
<td>2 N lg N</td>
<td>1</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>LSD      †</td>
<td>2 N W</td>
<td>2 N W</td>
<td>N + R</td>
<td>yes</td>
<td>charAt()</td>
</tr>
<tr>
<td>MSD      †</td>
<td>2 N W</td>
<td>N log&lt;sub&gt;R&lt;/sub&gt;N</td>
<td>N + D R</td>
<td>yes</td>
<td>charAt()</td>
</tr>
<tr>
<td>3-way string quicksort</td>
<td>1.39 W N lg N</td>
<td>1.39 N lg N</td>
<td>log N + W</td>
<td>no</td>
<td>charAt()</td>
</tr>
</tbody>
</table>

* probabilistic
† fixed-length W keys
‡ average-length W keys
key-indexed counting
LSD string sort
MSD string sort
3-way radix quicksort
suffix arrays
LCP. Given two strings, find the longest substring that is a prefix of both.

```
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.

Applications. Bioinformatics, cryptanalysis, data compression, ...
Longest repeated substring: a musical application


Mary Had a Little Lamb

Bach's Goldberg Variations
Longest repeated substring

**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.

```
| a | a | c | a | a | g | t | t | t | a | c | a | a | g | c |
```

**Analysis.** Running time $\leq M N^2$, where $M$ is length of longest match.
Longest repeated substring: a sorting solution

input string

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
</tr>
</tbody>
</table>

form suffixes

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
</tr>
<tr>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
</tr>
<tr>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>g</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>g</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

sort suffixes to bring repeated substrings together

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>g</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

compute longest prefix between adjacent suffixes

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>g</td>
<td>c</td>
</tr>
</tbody>
</table>

65
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;
}

% java LRS < mobydict.txt
,- Such a funny, sporty, gamy, jesty, joky, hoky-poky lad, is the Ocean, oh! Th
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

<table>
<thead>
<tr>
<th>input file</th>
<th>characters</th>
<th>brute</th>
<th>suffix sort</th>
<th>length of LRS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRS.java</td>
<td>2,162</td>
<td>0.6 sec</td>
<td>0.14 sec</td>
<td>73</td>
</tr>
<tr>
<td>amendments.txt</td>
<td>18,369</td>
<td>37 sec</td>
<td>0.25 sec</td>
<td>216</td>
</tr>
<tr>
<td>aesop.txt</td>
<td>191,945</td>
<td>1.2 hours</td>
<td>1.0 sec</td>
<td>58</td>
</tr>
<tr>
<td>mobydict.txt</td>
<td>1.2 million</td>
<td>43 hours†</td>
<td>7.6 sec</td>
<td>79</td>
</tr>
<tr>
<td>chromosome11.txt</td>
<td>7.1 million</td>
<td>2 months†</td>
<td>61 sec</td>
<td>12,567</td>
</tr>
<tr>
<td>pi.txt</td>
<td>10 million</td>
<td>4 months†</td>
<td>84 sec</td>
<td>14</td>
</tr>
</tbody>
</table>

† estimated
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 abedefgh2.txt
abcdefgh
abcdefghabcdefgh
bcdefgh
bcdefghabcdefgh
cdefgh
cdefghabcdefgh
defgh
defghabcdefgh
defgh
defghabcdefgh
defgh
defghabcdefgh
defgh
defghabcdefgh
defgh
defghabcdefgh
fn

<table>
<thead>
<tr>
<th>algorithm</th>
<th>mobydict.txt</th>
<th>aesopaesop.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>brute-force</td>
<td>36,000 †</td>
<td>4000 †</td>
</tr>
<tr>
<td>quicksort</td>
<td>9.5</td>
<td>167</td>
</tr>
<tr>
<td>LSD</td>
<td>not fixed length</td>
<td>not fixed length</td>
</tr>
<tr>
<td>MSD</td>
<td>395</td>
<td>out of memory</td>
</tr>
<tr>
<td>MSD with cutoff</td>
<td>6.8</td>
<td>162</td>
</tr>
<tr>
<td>3-way string quicksort</td>
<td>2.8</td>
<td>400</td>
</tr>
</tbody>
</table>

† 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.
- Linear.
- Nobody knows.

✓ Linearithmic. $\rightarrow$ Manber’s algorithm
✓ Linear. $\rightarrow$ suffix trees (see COS 423)
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 $\log N$ 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 0 |
| 1 | a b a a a a b c b a b a a a a 0 |
| 2 | b a a a a a b c b a b 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 0 |
| 5 | a a b c b a b a a a a 0 |
| 6 | a b c b a b a a a a 0 |
| 7 | b c b a b a a a a 0 |
| 8 | c b a b a a a a 0 |
| 9 | b a b a a a a 0 |
| 10 | a b 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 |
| 16 | a 0 |
| 15 | a a 0 |
| 14 | a a a 0 |
| 13 | a a a a 0 |
| 12 | a a a a 0 |
| 11 | a b a a a a 0 |
| 10 | a b a a a a 0 |
| 9 | b a b a a a a 0 |
| 8 | b a b a a a a 0 |
| 7 | b c b a b a a a a 0 |
| 6 | a b c b a b a a a a 0 |
| 5 | a a b c b a b a a a a 0 |
| 4 | a a a b c b a b a a a 0 |
| 3 | a a a a b c b a b a a a a 0 |
| 2 | b c b a b a a a a 0 |
| 1 | a a a a a b c b a b a a a a 0 |

**Sorted**

- The suffixes are sorted based on their first character.
- The key-indexed counting sort is applied.
- The sorted list is displayed in the right-hand column.

---

(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 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)

| 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 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 b c b a b a a a a a 0 |
| 10 | a b a a a a b c b a b a a a a a 0 |
| 11 | a 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 |

*sorted*
## Linearithmic suffix sort example: phase 2

<table>
<thead>
<tr>
<th>original suffixes</th>
<th>index sort (first four characters)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>b a b a a a a b c b a b a a a a a 0</td>
</tr>
<tr>
<td>1</td>
<td>a b a a a a b c b a b a a a a a 0</td>
</tr>
<tr>
<td>2</td>
<td>b a a a a b c b a b a a a a a 0</td>
</tr>
<tr>
<td>3</td>
<td>a a a a b c b a b a a a a a a 0</td>
</tr>
<tr>
<td>4</td>
<td>a a a b c b a b a a a a a 0</td>
</tr>
<tr>
<td>5</td>
<td>a a b c b a b a a a a a 0</td>
</tr>
<tr>
<td>6</td>
<td>a b c b a b a a a a a 0</td>
</tr>
<tr>
<td>7</td>
<td>b c b a b a a a a a 0</td>
</tr>
<tr>
<td>8</td>
<td>c b a b a a a a a 0</td>
</tr>
<tr>
<td>9</td>
<td>b a b a a a a a 0</td>
</tr>
<tr>
<td>10</td>
<td>a b a a a a a 0</td>
</tr>
<tr>
<td>11</td>
<td>b a a a a a 0</td>
</tr>
<tr>
<td>12</td>
<td>a a a a a 0</td>
</tr>
<tr>
<td>13</td>
<td>a a a a 0</td>
</tr>
<tr>
<td>14</td>
<td>a a a 0</td>
</tr>
<tr>
<td>15</td>
<td>a a 0</td>
</tr>
<tr>
<td>16</td>
<td>a 0</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
</tr>
</tbody>
</table>

sorted
### Linearithmic suffix sort example: phase 3

<table>
<thead>
<tr>
<th>original suffixes</th>
<th>index sort (first eight characters)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: b a b a a a b c b a b a a a a a 0</td>
<td>17: 0</td>
</tr>
<tr>
<td>1: a b a a a a b c b a b a a a a a 0</td>
<td>16: a 0</td>
</tr>
<tr>
<td>2: b a a a a b c b a b a a a a a 0</td>
<td>15: a a 0</td>
</tr>
<tr>
<td>3: a a a a a b c b a b a a a a a a 0</td>
<td>14: a a a 0</td>
</tr>
<tr>
<td>4: a a b c b a b a a a a a 0</td>
<td>13: a a a a a 0</td>
</tr>
<tr>
<td>5: a a b c b a b a a a a a 0</td>
<td>12: a a a a a a 0</td>
</tr>
<tr>
<td>6: a b c b a b a a a a a 0</td>
<td>11: a a a a b c b a b a a a a a 0</td>
</tr>
<tr>
<td>7: b c b a b a a a a a 0</td>
<td>10: a a a a b c b a b a a a a 0</td>
</tr>
<tr>
<td>8: c b a b a a a a a 0</td>
<td>9: a b a a a a a 0</td>
</tr>
<tr>
<td>9: b a b a a a a 0</td>
<td>8: a b a a a a a 0</td>
</tr>
<tr>
<td>10: a b a a a a a 0</td>
<td>7: b a a a a b c b a b a a a a 0</td>
</tr>
<tr>
<td>11: b a a a a a 0</td>
<td>6: a b c b a b a a a a 0</td>
</tr>
<tr>
<td>12: a a a a a 0</td>
<td>5: b a a a a a 0</td>
</tr>
<tr>
<td>13: a a a a 0</td>
<td>4: b a a a a a b c b a b a a a a a 0</td>
</tr>
<tr>
<td>14: a a a 0</td>
<td>3: b a a a a a 0</td>
</tr>
<tr>
<td>15: a a 0</td>
<td>2: b a a a a a b c b a b a a a a 0 a 0</td>
</tr>
<tr>
<td>16: a 0</td>
<td>1: b a a a a a b c b a b a a a a a 0</td>
</tr>
<tr>
<td>17: 0</td>
<td>0: b a b a a a b c b a b a a a a a 0</td>
</tr>
</tbody>
</table>

sorted

FINISHED! (no equal keys)
Achieve constant-time string compare by indexing into inverse

<table>
<thead>
<tr>
<th>original suffixes</th>
<th>index sort (first four characters)</th>
<th>inverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 b a b a a a a b c b a b a a a a a a 0</td>
<td>17 0</td>
<td>0 14</td>
</tr>
<tr>
<td>1 a b a a a a b c b a b a a a a a a 0</td>
<td>16 a 0</td>
<td>1 9</td>
</tr>
<tr>
<td>2 b a a a a b c b a b a a a a a a 0</td>
<td>15 a a 0</td>
<td>2 12</td>
</tr>
<tr>
<td>3 a a a a b c b a b a a a a a a 0</td>
<td>14 a a a 0</td>
<td>3 4</td>
</tr>
<tr>
<td>4 a a b c b a b a a a a a a 0</td>
<td>13 a a a a a a a a 0</td>
<td>4 7</td>
</tr>
<tr>
<td>5 a b c b a b a a a a a 0</td>
<td>12 a a a a a</td>
<td>5 8</td>
</tr>
<tr>
<td>6 a b c b a b a a a a a 0</td>
<td>11 b a a a a a a a a a 0</td>
<td>6 11</td>
</tr>
<tr>
<td>7 b c b a b a a a a a 0</td>
<td>10 b a a a a a a a a a 0</td>
<td>7 16</td>
</tr>
<tr>
<td>8 c b a b a a a a a 0</td>
<td>9 b a a a a a a a a a 0</td>
<td>8 17</td>
</tr>
<tr>
<td>9 b a b a a a a a 0</td>
<td>8 b a a a a a a a a a 0</td>
<td>9 15</td>
</tr>
<tr>
<td>10 a b a a a a a 0</td>
<td>7 b a a a a a a a a a 0</td>
<td>10 10</td>
</tr>
<tr>
<td>11 b a a a a a 0</td>
<td>6 a b c b a b a a a a a 0</td>
<td>11 13</td>
</tr>
<tr>
<td>12 a a a a a 0</td>
<td>5 a b a a a a a a a a a 0</td>
<td>12 5</td>
</tr>
<tr>
<td>13 a a a 0</td>
<td>4 b a a a a a a a a a 0</td>
<td>13 6</td>
</tr>
<tr>
<td>14 a a a 0</td>
<td>3 b a b a a a a a b c b a b a a a a a 0</td>
<td>14 3</td>
</tr>
<tr>
<td>15 a a 0</td>
<td>2 b a b a a a a a 0</td>
<td>15 2</td>
</tr>
<tr>
<td>16 a 0</td>
<td>1 b c b a b a a a a a 0</td>
<td>16 1</td>
</tr>
<tr>
<td>17 0</td>
<td>0 c b a b a a a a a 0</td>
<td>17 0</td>
</tr>
</tbody>
</table>

so suffixes₈[9] ≤ suffixes₈[0]
## Suffix sort: experimental results

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>mobyduck.txt</th>
<th>aesopaeasop.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>brute-force</td>
<td>36.000 †</td>
<td>4000 †</td>
</tr>
<tr>
<td>quicksort</td>
<td>9.5</td>
<td>167</td>
</tr>
<tr>
<td>LSD</td>
<td>not fixed length</td>
<td>not fixed length</td>
</tr>
<tr>
<td>MSD</td>
<td>395</td>
<td>out of memory</td>
</tr>
<tr>
<td>MSD with cutoff</td>
<td>6.8</td>
<td>162</td>
</tr>
<tr>
<td>3-way string quicksort</td>
<td>2.8</td>
<td>400</td>
</tr>
<tr>
<td>Manber MSD</td>
<td>17</td>
<td>8.5</td>
</tr>
</tbody>
</table>

† 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.

<table>
<thead>
<tr>
<th>implementation</th>
<th>search</th>
<th>insert</th>
<th>delete</th>
<th>ordered operations</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>red-black BST</td>
<td>1.00 lg N</td>
<td>1.00 lg N</td>
<td>1.00 lg N</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>hashing</td>
<td>1 †</td>
<td>1 †</td>
<td>1 †</td>
<td>no</td>
<td>equals() hashcode()</td>
</tr>
</tbody>
</table>

† 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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class StringST&lt;Value&gt;</td>
<td>string symbol table type</td>
</tr>
<tr>
<td>StringST()</td>
<td>create an empty symbol table</td>
</tr>
<tr>
<td>void put(String key, Value val)</td>
<td>put key-value pair into the symbol table</td>
</tr>
<tr>
<td>Value get(String key)</td>
<td>return value paired with given key</td>
</tr>
<tr>
<td>boolean contains(String key)</td>
<td>is there a value paired with the given key?</td>
</tr>
</tbody>
</table>

Goal. As fast as hashing, more flexible than binary search trees.
## String symbol table implementations cost summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>character accesses (typical case)</th>
<th>dedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search hit</td>
<td>space (links)</td>
</tr>
<tr>
<td>red-black BST</td>
<td>L + c ( \lg^2 N )</td>
<td>4 N</td>
</tr>
<tr>
<td>hashing</td>
<td>L</td>
<td>4 N to 16 N</td>
</tr>
<tr>
<td></td>
<td>search miss</td>
<td>c ( \lg^2 N )</td>
</tr>
<tr>
<td></td>
<td>insert</td>
<td>c ( \lg^2 N )</td>
</tr>
</tbody>
</table>

### Parameters
- \( N \) = number of strings
- \( L \) = length of string
- \( R \) = radix

### Challenge
Efficient performance for string keys.

<table>
<thead>
<tr>
<th>file</th>
<th>size</th>
<th>words</th>
<th>distinct</th>
</tr>
</thead>
<tbody>
<tr>
<td>moby.txt</td>
<td>1.2 MB</td>
<td>210 K</td>
<td>32 K</td>
</tr>
<tr>
<td>actors.txt</td>
<td>82 MB</td>
<td>11.4 M</td>
<td>900 K</td>
</tr>
</tbody>
</table>
tries

TSTs

string symbol table API
**Tries**

**Tries.** [from retrieval, 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
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.

```
get("shells")
```

```
get("she")
```

- **return the value in the node corresponding to the last key character (3)**
- **search may terminate at an internal node**
- **return the value in the node corresponding to the last key character (0)**
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.

---

**Trie search miss outcomes**

- **get("shell")**
  - No link for the `o`, so return `null`.
- **get("shore")**
  - No value in the node corresponding to the last key character, so return `null`. 
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

key   value
she   0
      root

value is in node corresponding to last character

sells 1

one node for each key character

sea   2

key is sequence of characters from root to value

shells 3

nodes corresponding to characters at the end of the key do not exist, so create them and set the value of the last one

by    4

node corresponding to the last key character exists, so reset its value

shore 7

key   value
sea   6

node corresponding to the last key character exists, so reset its value
Trie representation: Java implementation

**Node.** A value, plus references to R nodes.

```java
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.

Characters are implicitly defined by link index.

Each node has an array of links and a value.
Trie representation: Java implementation

**Node.** A value, plus references to \( R \) nodes.

```java
private static class Node {
    private Object value;
    private Node[] next = new Node[R];
}
```

Trie representation (\( R = 26 \))

- Each node has an array of links and a value.
- Characters are implicitly defined by link index.
- Use `Object` instead of `Value` since no generic array creation in Java.
- Each node has an array of links and a value.
public class TrieST<Value> {
    private static final int R = 256;
    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;
    }
}
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

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Character accesses (typical case)</th>
<th>Dedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search hit</td>
<td>search miss</td>
</tr>
<tr>
<td>red-black BST</td>
<td>L + c (lg^2 N)</td>
<td>c (lg^2 N)</td>
</tr>
<tr>
<td>hashing</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>R-way trie</td>
<td>L</td>
<td>(log_R N)</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>machine</th>
<th>year</th>
<th>address bits</th>
<th>addressable memory</th>
<th>typical actual memory</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDP-8</td>
<td>1960s</td>
<td>12</td>
<td>6 KB</td>
<td>6 KB</td>
<td>$16K</td>
</tr>
<tr>
<td>PDP-10</td>
<td>1970s</td>
<td>18</td>
<td>256 KB</td>
<td>256 KB</td>
<td>$1M</td>
</tr>
<tr>
<td>IBM S/360</td>
<td>1970s</td>
<td>24</td>
<td>4 MB</td>
<td>512 KB</td>
<td>$1M</td>
</tr>
<tr>
<td>VAX</td>
<td>1980s</td>
<td>32</td>
<td>4 GB</td>
<td>1 MB</td>
<td>$1M</td>
</tr>
<tr>
<td>Pentium</td>
<td>1990s</td>
<td>32</td>
<td>4 GB</td>
<td>1 GB</td>
<td>$1K</td>
</tr>
<tr>
<td>Xeon</td>
<td>2000s</td>
<td>64</td>
<td>enough</td>
<td>4 GB</td>
<td>$100</td>
</tr>
<tr>
<td>??</td>
<td>future</td>
<td>128+</td>
<td>enough</td>
<td>enough</td>
<td>$1</td>
</tr>
</tbody>
</table>

“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 \( \approx 2^{59} \) 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.

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.
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).
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).
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.

![TST search diagram](image)
26-way trie vs. TST

26-way trie. 26 null links in each leaf.

TST. 3 null links in each leaf.
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.

Trie node representations

standard array of links (R = 26)

ternary search tree (TST)

link for keys that start with s
link for keys that start with su

TST representation in Java

private class Node
{
    private Value val;
    private char c;
    private Node left, mid, right;
}
TST: Java implementation

```java
public class TST<
{
    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;
    }
}
```
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

<table>
<thead>
<tr>
<th>implementation</th>
<th>character accesses (typical case)</th>
<th>dedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search hit</td>
<td>search miss</td>
</tr>
<tr>
<td>red-black BST</td>
<td>$L + c \log^2 N$</td>
<td>$c \log^2 N$</td>
</tr>
<tr>
<td>hashing</td>
<td>$L$</td>
<td>$L$</td>
</tr>
<tr>
<td>R-way trie</td>
<td>$L$</td>
<td>$\log_R N$</td>
</tr>
<tr>
<td>TST</td>
<td>$L + \ln N$</td>
<td>$\ln N$</td>
</tr>
</tbody>
</table>

**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.
TST with $R^2$ branching at root

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

<table>
<thead>
<tr>
<th>implementation</th>
<th>search hit</th>
<th>search miss</th>
<th>insert</th>
<th>space (links)</th>
<th>moby.txt</th>
<th>actors.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>red-black BST</td>
<td>$L + c \lg^2 N$</td>
<td>$c \lg^2 N$</td>
<td>$c \lg^2 N$</td>
<td>$4N$</td>
<td>1.40</td>
<td>97.4</td>
</tr>
<tr>
<td>hashing</td>
<td>$L$</td>
<td>$L$</td>
<td>$L$</td>
<td>$4N$ to $16N$</td>
<td>0.76</td>
<td>40.6</td>
</tr>
<tr>
<td>R-way trie</td>
<td>$L$</td>
<td>$\log_R N$</td>
<td>$L$</td>
<td>$(R + 1)N$</td>
<td>1.12</td>
<td>out of memory</td>
</tr>
<tr>
<td>TST</td>
<td>$L + \ln N$</td>
<td>$\ln N$</td>
<td>$L + \ln N$</td>
<td>$4N$</td>
<td>0.72</td>
<td>38.7</td>
</tr>
<tr>
<td>TST with $R^2$</td>
<td>$L + \ln N$</td>
<td>$\ln N$</td>
<td>$L + \ln N$</td>
<td>$4N + R^2$</td>
<td>0.51</td>
<td>32.7</td>
</tr>
</tbody>
</table>
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
Character-based operations. The string symbol table API supports several useful character-based operations.

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class StringST&lt;Value&gt;</td>
<td></td>
</tr>
<tr>
<td>StringST()</td>
<td>create a symbol table with string keys</td>
</tr>
<tr>
<td>StringST(Alphabet alpha)</td>
<td>create a symbol table with string keys whose characters are taken from alpha</td>
</tr>
<tr>
<td>void put(String key, Value val)</td>
<td>put key-value pair into the symbol table</td>
</tr>
<tr>
<td></td>
<td>(remove key from table if value is null)</td>
</tr>
<tr>
<td>Value get(String key)</td>
<td>value paired with key</td>
</tr>
<tr>
<td></td>
<td>(null if key is absent)</td>
</tr>
<tr>
<td>void delete(String key)</td>
<td>remove key (and its value) from table</td>
</tr>
<tr>
<td>boolean contains(String key)</td>
<td>is there a value paired with key?</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>is the table empty?</td>
</tr>
<tr>
<td>String longestPrefixOf(String s)</td>
<td>return the longest key that is a prefix of s</td>
</tr>
<tr>
<td>Iterable&lt;String&gt; keysWithPrefix(String s)</td>
<td>all the keys having s as a prefix.</td>
</tr>
<tr>
<td>Iterable&lt;String&gt; keysThatMatch(String s)</td>
<td>all the keys that match s (where . matches any character)</td>
</tr>
<tr>
<td>int size()</td>
<td>number of key-value pairs in the table</td>
</tr>
<tr>
<td>Iterable&lt;String&gt; keys()</td>
<td>all the keys in the symbol table</td>
</tr>
</tbody>
</table>

**Remark.** Can also add other ordered ST methods, e.g., `floor()` and `rank()`. 

This API differs from the general-purpose symbol-table API introduced in Chapter 4 in just the following aspects:

- We replace the generic type `Key` with the concrete type `String`.
- We add a constructor that allows clients to specify the alphabet.
- We add three new methods, `longestPrefixOf()`, `keysWithPrefix()`, and `keysThatMatch()`.

We retain the basic conventions of our symbol-table implementations in Chapter 4 (no duplicate or null keys and no null values). To focus on the main ideas, we concentrate on `put()` and `get()`, assume (as in Chapter 4) default implementations of `contains()` and `isEmpty()` and leave implementations of `size()` and `delete()` for exercises.

Since strings are `Comparable`, extending the API to also include the ordered operations defined in Chapter 4 is also possible (and worthwhile); we leave those implementations (which are generally straightforward) to exercises and booksite code.
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).

delete("shells");

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("");
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.

```java
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);
}
```
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.
Find all keys in symbol table starting with a given prefix.

```
public Iterable<String> keysWithPrefix(String prefix) {
    Queue<String> queue = new Queue<String>();
    Node x = get(root, prefix, 0);
    collect(x, prefix, queue);
    return queue;
}
```
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"

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.

```java
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 6 6 6
- **T9:** 4 3 5 5 6
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
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.
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

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 $N \gg M$

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.
- FBI's Digital Collection System 3000.
- Feature detection in digitized images.
- ...
Application: Spam filtering

Identify patterns indicative of spam.

- PROFITS
- LOSE WEIGHT
- herbal Viagra
- There is no catch.
- LOW MORTGAGE 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

Need to monitor all internet traffic. (security)

Well, we're mainly interested in "ATTACK AT DAWN"

No way! (privacy)

OK. Build a machine that just looks for that.

"ATTACK AT DAWN" substring search machine

found
**Application: Screen scraping**

**Goal.** Extract relevant data from web page.

**Ex.** Find string delimited by `<b>` and `</b>` 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">
<b>452.92</b>
</td>
</tr>
<tr>
<td class="yfnc_tablehead1" width="48%">
Trade Time:
</td>
<td class="yfnc_tabledata1">
...</td>
</tr>
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.

```java
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.

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>i+j</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

txt → A B A C A D A B R A C

entries in red are mismatches
entries in gray are for reference only
entries in black match the text

return i when j is M

match
Brute-force substring search: Java implementation

Check for pattern starting at each text position.

```java
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;
    }
    return N;
}
```

index in text where pattern starts

not found
Brute-force substring search: worst case

Brute-force algorithm can be slow if text and pattern are repetitive.

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>i+j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>txt</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>4</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td></td>
<td>pat</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>7</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>8</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>9</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Brute-force substring search (worst case)

Worst case. \( \sim MN \) char compares.
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).

```java
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;
}
```
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 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 many good Republicans 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 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 many good Republicans 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 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 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 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 `BAAAAAAA`.  
- Suppose we match 5 chars in pattern, with mismatch on 6\(^{th}\) char.  
- We know previous 6 chars in text are `BAAAB`.  
- Don't need to back up text pointer!

**Remark.** It is **always possible** to avoid backup (!)

---

### Diagram

- Text pointer after mismatch on sixth char  
- Brute-force backs up to try this and this and this and this but no backup is needed

### Pattern

```
A B A A A A B A A A A A A A A A A A A A A A
B A A A A A A A A A A A A A A A A A A A A A
B A A A A A A A A A A A A A A A A A A A A A
B A A A A A A A A A A A A A A A A A A A A A
B A A A A A A A A A A A A A A A A A A A A A
B A A A A A A A A A A A A A A A A A A A A A
```

---

### Table

<table>
<thead>
<tr>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

---

### Characters

- A
- B
Q. What pattern char do we compare to the next text char on match?
A. Easy: compare next pattern char to 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.

<table>
<thead>
<tr>
<th>Current char</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>matched chars</td>
<td>×</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>current char is mismatch</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>pat.charAt(j)</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>dfa[][]</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>pat.charAt(1)</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>1</td>
</tr>
</tbody>
</table>

table giving pattern char to compare to the next text char
Q. What pattern char do we compare to the next text char on mismatch?
A. Check each position, working from left to right.
**KMP substring search preprocessing (concept)**

Fill in table columns by doing computation for each possible mismatch position.

<table>
<thead>
<tr>
<th>j</th>
<th>pat. charAt(j)</th>
<th>dfa[j][j]</th>
<th>text (pattern itself)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td></td>
<td>ABABAC</td>
</tr>
<tr>
<td>0</td>
<td>A</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ABABAC</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ABABAC</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ABABAC</td>
</tr>
</tbody>
</table>

Pattern backup for ABABAC in KMP substring search

1. **match (move to next char)**
   - Set dfa[pat.charAt(j)][j] to j+1

2. **known text chars on mismatch**
   - ABAB
   - ABABAC

3. **mismatch (back up in pattern)**
   - ABABAC
   - ABABAC
   - ABABAC
   - ABABAC

4. **backup is length of max overlap of beginning of pattern with known text chars**
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.

<table>
<thead>
<tr>
<th>pat.charAt(j)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

If in state j reading char c:
- halt if j is 6
- else move to state $\text{dfa}[c][j]$

DFA corresponding to the string A B A B A C
KMP substring search: trace

Trace of KMP substring search (DFA simulation) for A B A B A C
KMP search: Java implementation

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.

```java
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 $M^2R$ 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 $\text{dfa}[][]$ table from pattern.
• Could use input stream.

```java
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;
}
```
Q. What state $X$ would the DFA be in if it were restarted to correspond to shifting the pattern one position to the right?

A. Use the (partially constructed) DFA to find $X$!

**Consequence.**

- We want the **same** transitions as $X$ for the next state on mismatch.

  \[
  \text{copy } \text{dfa}[][X] \text{ to } \text{dfa}[][j]
  \]

- But a different transition (to $j+1$) on match.

  \[
  \text{Set } \text{dfa}[	ext{pat.charAt(j)}][j] \text{ to } j+1
  \]
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?

Observation. No need to restart DFA.
- Remember last restart state in X.
- Use DFA to update X.
- \( X = \text{dfa}[\text{pat.charAt(j)}][X] \)

DFA simulations to compute restart states for A B A B A C
Constructing the DFA for KMP substring search: example

```
pat.charAt(j) | 0 1 2 3
--------------|--------------
A             | B A A B
X
pat.charAt(j) | 0 1 2 3
--------------|--------------
A             | B A A B
X
pat.charAt(j) | 0 1 2 3
--------------|--------------
A             | B A A B
X
```

```
copy dfa[X] to dfa[j]
dfa[pat.charAt(j)][j] = j+1;
X = dfa[pat.charAt(j)][X];
```
Constructing the DFA for KMP substring search: example

pat.charAt(j)  A  B  A  B
| A  1  1  3  1
| B  0  2  0  4
| C  0  0  0  0

dfa[][][j]

pat.charAt(j)  A  B  A  B  A
| A  1  1  3  1  5
| B  0  2  0  4  0
| C  0  0  0  0  0

Converting 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 \( \text{dfa}[][X] \) to \( \text{dfa}[][j] \) for mismatch case.
- Set \( \text{dfa}[	ext{pat.charAt}(j)][j] \) to \( j+1 \) for match case.
- Update \( X \).

```java
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];
        dfa[pat.charAt(j)][j] = j+1;
        X = dfa[pat.charAt(j)][X];
    }
}
```

Running time. \( M \) character accesses.
KMP substring search analysis

**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 $RM$ to construct $\text{dfa}[][]$, 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.
brute force
Knuth-Morris-Pratt
Boyer-Moore
Rabin-Karp

Robert Boyer
J. Strother Moore
**Intuition.**

- Scan characters in pattern from right to left.
- Can skip $M$ text chars when finding one not in the pattern.

<p>| | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>j</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>text</td>
<td>H A Y S T A C K N E E D L E I N A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>N E E D L E</td>
<td>→</td>
<td>pattern</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>N E E D L E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>N E E D L E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`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.

```
return i = 18 (no match)
```
**Boyer-Moore: mismatched character heuristic**

**Q.** How much to skip?

**A.** Compute $\text{right}[c] = \text{rightmost occurrence of character } c \text{ in } \text{pat}[].$

```java
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;
```

<table>
<thead>
<tr>
<th>c</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>right[c]</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>B</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>C</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>D</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>M</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>N</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-1</td>
</tr>
</tbody>
</table>

Boyer-Moore skip table computation
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 \( \text{right}[c] = \text{rightmost occurrence of character } c \text{ in } \text{pat}[] \).

\[
\begin{array}{cccccc}
\text{i} & \downarrow & \text{i+j} & \downarrow & \ldots & \ldots & \text{T L E} & \ldots & \ldots \\
\ldots & \ldots & \text{N E E D L E} & \text{j} & \downarrow & \text{i+j+1} & \downarrow & \text{KMP-like table} \\
\ldots & \ldots & \text{T L E} & \ldots & \ldots \\
\ldots & \ldots & \text{N E E D L E} & \text{j} & \downarrow & \text{M-1} & \uparrow & \text{j} \\
\end{array}
\]

Mismatched character heuristic (mismatch not in pattern)

Easy fix. Set \( \text{right}[c] \) to -1 for characters not in pattern.
Q. How much to skip?
A. Compute $right[c] = \text{rightmost occurrence of character } c \text{ in } \text{pat}[]$.

The heuristic is no help

```
  i       i+j
     ↓      ↓
  . . . . . E L E . . . . . .
N E E D L E
     ↑
  j
```

Lining up text with rightmost E

would shift pattern left

```
  . . . . . E L E . . . . . .
N E E D L E
     ↑
  j

so increment i by 1

```

```
  . . . . . E L E . . . . . .
N E E D L E
     ↑
  j
reset j to M-1
```

could do better with KMP-like table
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;
}
Boyer-Moore: analysis

**Property.** Substring search with the Boyer-Moore mismatched character heuristic takes about $\sim \frac{N}{M}$ character compares to search for a pattern of length $M$ in a text of length $N$.

**Worst-case.** Can be as bad as $\sim MN$.

<table>
<thead>
<tr>
<th>$i$</th>
<th>skip</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>txt</td>
<td></td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td></td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td></td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Boyer-Moore variant.** Can improve worst case to $\sim 3N$ 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.

```
<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>
```

```
pat.charAt(i)
```

```
<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>% 997 = 613</td>
</tr>
</tbody>
</table>
```

```
txt.charAt(i)
```

```
<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>7</td>
<td>9</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>% 997 = 508</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>% 997 = 201</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>% 997 = 715</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>% 997 = 971</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>% 997 = 442</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>% 997 = 929</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>return i = 6</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>% 997 = 613</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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.

// 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(i)) % Q;
   return h;
}

Computing the hash value for the pattern with Horner’s method
Challenge. How to efficiently compute $x_{i+1}$ given that we know $x_i$.

\[ x_i = t_i \cdot R^{M-1} + t_{i+1} \cdot R^{M-2} + \ldots + t_{i+M-1} \cdot R^0 \]

\[ x_{i+1} = t_{i+1} \cdot R^{M-1} + t_{i+2} \cdot R^{M-2} + \ldots + t_{i+M} \cdot R^0 \]

Key property. Can do it in constant time!

\[ x_{i+1} = (x_i - t_i \cdot R^{M-1}) \cdot R + t_{i+M} \]
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 */
    }
}
Rabin-Karp: Java implementation (continued)

```java
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;
}
```
Rabin-Karp substring search example

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>7</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>% 997 = 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>% 997 = (3*10 + 1) % 997 = 31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>% 997 = (31*10 + 4) % 997 = 314</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>% 997 = (314*10 + 1) % 997 = 150</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>% 997 = (150*10 + 5) % 997 = 508</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>% 997 = ((508 + 3*(997 - 30))*10 + 9) % 997 = 201</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>% 997 = ((201 + 1*(997 - 30))*10 + 2) % 997 = 715</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>% 997 = ((715 + 4*(997 - 30))*10 + 6) % 997 = 971</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>% 997 = ((971 + 1*(997 - 30))*10 + 5) % 997 = 442</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>% 997 = ((442 + 5*(997 - 30))*10 + 3) % 997 = 929</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>return i-M+1 = 6</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>% 997 = ((929 + 9*(997 - 30))*10 + 5) % 997 = 613</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

return i-M+1 = 6

match

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.
Rabin-Karp fingerprint search

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.

<table>
<thead>
<tr>
<th>Algorithm (data structure)</th>
<th>Operation count</th>
<th>Backup in input?</th>
<th>Space grows with</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>guarantee</td>
<td>typical</td>
<td></td>
</tr>
<tr>
<td>brute force</td>
<td>$MN$</td>
<td>1.1 $N$</td>
<td>yes</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt (full DFA)</td>
<td>$2N$</td>
<td>1.1 $N$</td>
<td>no</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt (mismatch transitions only)</td>
<td>$3N$</td>
<td>1.1 $N$</td>
<td>no</td>
</tr>
<tr>
<td>Boyer-Moore</td>
<td>$3N$</td>
<td>$N/M$</td>
<td>yes</td>
</tr>
<tr>
<td>Boyer-Moore (mismatched character heuristic only)</td>
<td>$MN$</td>
<td>$N/M$</td>
<td>yes</td>
</tr>
<tr>
<td>Rabin-Karp†</td>
<td>$7N$†</td>
<td>$7N$</td>
<td>no</td>
</tr>
</tbody>
</table>

*† probabilistic 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 \texttt{CGG} or \texttt{AGG}, bracketed by \texttt{GCG} at the beginning and \texttt{CTG} at the end.
- Number of repeats is variable, and correlated with syndrome.

\begin{tabular}{ll}
\texttt{pattern} & GCG (CGG | AGG) *CTG \\
\texttt{text} & GCGGCGTGTGTCGGAGAGGTTGTTTTAAAGCTG GCGGAGGCGGGCTG GCGGAGGGCTG \\
\end{tabular}
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.

<table>
<thead>
<tr>
<th>operation</th>
<th>example RE</th>
<th>matches</th>
<th>does not match</th>
</tr>
</thead>
<tbody>
<tr>
<td>concatenation</td>
<td>AABAAB</td>
<td>AABAAB</td>
<td>every other string</td>
</tr>
<tr>
<td>or</td>
<td>AA</td>
<td>BAAB</td>
<td>AA</td>
</tr>
<tr>
<td>closure</td>
<td>AB*A</td>
<td>AA</td>
<td>AB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ABBBBBBBBA</td>
<td>ABABA</td>
</tr>
<tr>
<td>parentheses</td>
<td>A (A</td>
<td>B) AAB</td>
<td>AAAAB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ABAAB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(AB) *A</td>
<td>A</td>
<td>AA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ABABABABABABA</td>
<td>ABBA</td>
</tr>
</tbody>
</table>
Regular expression shortcuts

Additional operations are often added for convenience.

**Ex.** \([A-E]+\) is shorthand for \((A|B|C|D|E)(A|B|C|D|E)^*\)

<table>
<thead>
<tr>
<th>operation</th>
<th>example RE</th>
<th>matches</th>
<th>does not match</th>
</tr>
</thead>
<tbody>
<tr>
<td>wildcard</td>
<td>.U.U.U.</td>
<td>CUMULUS</td>
<td>SUCCUBUS TUMULTUOUS</td>
</tr>
<tr>
<td>at least 1</td>
<td>A(BC)+DE</td>
<td>ABCDE</td>
<td>ADE BCDE</td>
</tr>
<tr>
<td>character classes</td>
<td>[A-Za-z][a-z]*</td>
<td>word Capitalized</td>
<td>camelCase 4illegal</td>
</tr>
<tr>
<td>exactly k</td>
<td>[0-9]{5}-[0-9]{4}</td>
<td>08540-1321 19072-5541</td>
<td>111111111 166-54-111</td>
</tr>
<tr>
<td>complement</td>
<td>[^AEIOU]{6}</td>
<td>RHYTHM</td>
<td>DECADE</td>
</tr>
</tbody>
</table>
Regular expression examples

Notation is surprisingly expressive

<table>
<thead>
<tr>
<th>regular expression</th>
<th>matches</th>
<th>does not match</th>
</tr>
</thead>
<tbody>
<tr>
<td>.<em>SPB.</em></td>
<td>RASPBERRY CRISPBREAD</td>
<td>SUBSPACE SUBSPECIES</td>
</tr>
<tr>
<td>(contains the trigraph spb)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0-9]{3}-[0-9]{2}-[0-9]{4}</td>
<td>166-11-4433 166-45-1111</td>
<td>11-55555555 8675309</td>
</tr>
<tr>
<td>(Social Security numbers)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[a-z]+@([a-z]+.)+(edu</td>
<td>com)</td>
<td><a href="mailto:wayne@princeton.edu">wayne@princeton.edu</a></td>
</tr>
<tr>
<td>(valid email addresses)</td>
<td><a href="mailto:rs@princeton.edu">rs@princeton.edu</a></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3a ident#3</td>
<td></td>
</tr>
<tr>
<td>[$ _A-Za-z][A-Za-z0-9]*</td>
<td>ident3 PatternMatcher</td>
<td></td>
</tr>
<tr>
<td>(valid Java identifiers)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
 Perl RE for valid RFC822 email addresses

Can the average programmer learn to use REs?
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)

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).

Underlying abstraction. Nondeterministic finite state automata (NFA).

Basic plan.
• Build NFA from RE.
• Simulate NFA with text as input.

Ken Thompson

NFA for pattern
(A*B|AC)D

accept
pattern matches text

reject
pattern does not match text

AAAABD
Nondeterministic finite-state automata

Pattern matching NFA.
• Pattern enclosed in parentheses.
• One state per pattern character (start = 0, accept = \( M \)).
• Red \( \varepsilon \)-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 \(( ( A * B \mid A C ) D )\)
Nondeterministic finite-state automata

**Ex.** Is `aaaabd` matched by NFA?

The NFA corresponding to the pattern `( ( A * B | A C ) D )`

- **Correct Path:**
  - Input: `AAAABD`
  - States: 0 → 1 → 2 → 3 → 2 → 3 → 4
  - Accept state 11

- **Wrong Guesses:**
  - Input: `AAAABD`
  - States: 0 → 1 → 6 → 7
  - No way out of state 4
  - No way out of state 7

- **No Way Out:**
  - State 4
  - State 7

- **Stalling Sequences:**
  - No way out of state 4
  - No way out of state 7
Nondeterministic finite-state automata

Ex. Is $\text{aaaabd}$ matched by NFA?

Note: any sequence of legal transitions that ends in state 11 is a proof.
Nondeterministic finite-state automata

Ex. Is **aaaac** matched by NFA?

---

Note: this is not a complete proof! (need to mention the infinite number of sequences involving $\epsilon$-transitions between 2 and 3)
**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.

NFA corresponding to the pattern `( ( A * B | A C ) D )`

![Nondeterministic Automaton Diagram](image)
Pattern matching implementation: basic plan (revised)

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.

![Diagram of text matching pattern](image)
- 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.

Q. How to perform reachability?
Digraph reachability

Find all vertices reachable from a given set of vertices.

```java
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 \(( (A \ast B \mid A C) D)\) NFA for input \(A A B D\)
NFA simulation example

Simulation of \(( ( A \ast B \mid A C ) D )\) NFA for input \(A A B D\)

2 3 4: set of states reachable via \(\epsilon\)-transitions after matching \(A A\)

5: set of states reachable after matching \(A A B\)

5 8 9: set of states reachable via \(\epsilon\)-transitions after matching \(A A B\)

10: set of states reachable after matching \(A A B D\)

10 11: set of states reachable via \(\epsilon\)-transitions after matching \(A A B D\)

accept!
public boolean recognizes(String txt) {
    DFS dfs = new DFS(G);

    SET<Integer> pc = new dfsreachable(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 = dfsreachable(match);
    }
    return pc.contains(M);
}

NFA simulation: Java implementation

states reachable from start by $\varepsilon$-transitions

all possible states after scanning past `txt.charAt(i)`

follow $\varepsilon$-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 $\varepsilon$-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.** ( ) . * |
Building an NFA corresponding to an RE

Parentheses. Add $\varepsilon$-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 $\varepsilon$-transition edges for each $*$ operator.

![Closure Diagram]

- For single-character closure:
  - Add edge $\text{G.addEdge}(i, i+1);$ and $\text{G.addEdge}(i+1, i);$ for each $i$ and $i+1.$

- For closure expression:
  - Add edges $\text{G.addEdge}(lp, i+1);$ and $\text{G.addEdge}(i+1, lp);$ for each $i$ and $i+1.$

![NFA Diagram]

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.**

NFA corresponding to the pattern \(( ( A \ast B \mid A C ) D )\)
NFA construction: implementation

**Goal.** Write a program to build the $\varepsilon$-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 $( ( A \ast B \mid A \ C ) D )$
NFA construction: example

Building the NFA corresponding to \( \left( \left( A \ast B \mid A \ C \right) D \right) \)
NFA construction: example

Building the NFA corresponding to \(( ( A \ast B \mid A \ C ) D )\)
NFA construction: Java implementation

```java
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);
    }
}
```
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 $\varepsilon$-transitions and perhaps execute one or two stack operations.
» 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.

```java
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);
        }
    }
}
```

**Bottom line.** Worst-case for grep (proportional to MN) is the same as for elementary exact substring match.
Typical grep application

Crossword puzzle

% more words.txt
a
aback
abacus
abalone
abandon
...

% grep s..ict.. words.txt
constrictor
stricter
stricture

dictionary
(standard in UNIX)
also on booksite
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>`?

```text
<blink>text</blink>some text<blink>more text</blink>
```

reluctant  reluctant  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.

```java
class Validate {
    public static void main(String[] args) {
        String regexp = args[0];
        String input = args[1];
        StdOut.println(input.matches(regexp));
    }
}
```

```bash
% 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
```
Harvesting information

**Goal.** Print all substrings of input that match a RE.

% java Harvester "gcg(cgg|agg)*ctg" chromosomeX.txt

gcgccggcggcggcggcggctg
gcgctg
gcgctg
gcgccggcggcggcggaggcggaggcggctg

% java Harvester "http://(\w+\.)*(\w+)" http://www.cs.princeton.edu

http://www.princeton.edu
http://www.google.com
http://www.cs.princeton.edu/news

harvest patterns from DNA
harvest links from website
Harvesting information

RE pattern matching is implemented in Java’s `Pattern` and `Matcher` classes.

```java
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!

SpamAssassin regular expression.

- 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.

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.
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 ⇒ DFA.
- grep RE ⇒ NFA.
- javac Java language ⇒ Java byte code.

<table>
<thead>
<tr>
<th>pattern</th>
<th>KMP</th>
<th>grep</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>parser</td>
<td>string</td>
<td>RE</td>
<td>program</td>
</tr>
<tr>
<td>compiler output</td>
<td>DFA</td>
<td>NFA</td>
<td>byte code</td>
</tr>
<tr>
<td>simulator</td>
<td>DFA simulator</td>
<td>NFA simulator</td>
<td>JVM</td>
</tr>
</tbody>
</table>
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

**Message.** Binary data $B$ we want to compress.

**Compress.** Generates a "compressed" representation $C(B)$.

**Expand.** Reconstructs original bitstream $B$.

**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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean readBoolean()</td>
<td>read 1 bit of data and return as a boolean value</td>
</tr>
<tr>
<td>char readChar()</td>
<td>read 8 bits of data and return as a char value</td>
</tr>
<tr>
<td>char readChar(int r)</td>
<td>read r bits of data and return as a char value</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>is the bitstream empty?</td>
</tr>
<tr>
<td>void close()</td>
<td>close the bitstream</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void write(boolean b)</td>
<td>write the specified bit</td>
</tr>
<tr>
<td>void write(char c)</td>
<td>write the specified 8-bit char</td>
</tr>
<tr>
<td>void write(char c, int r)</td>
<td>write the r least significant bits of the specified char</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>is the bitstream empty?</td>
</tr>
<tr>
<td>void close()</td>
<td>close the bitstream</td>
</tr>
</tbody>
</table>
Writing binary data

Date representation. Different ways to represent 12/31/1999.

A character stream (StdOut)

```java
StdOut.print(month + "/" + day + "/" + year);
```

```
0011000100110010001011110011011110011000110101111001110001001110010011100100111001
```

80 bits

Three ints (BinaryStdOut)

```java
BinaryStdOut.write(month);
BinaryStdOut.write(day);
BinaryStdOut.write(year);
```

```
0000000000000000000000000000001100000000000000000000000000001111100000000000000000000011111001111
```

96 bits

Two chars and a short (BinaryStdOut)

```java
BinaryStdOut.write((char) month);
BinaryStdOut.write((char) day);
BinaryStdOut.write((short) year);
```

```
0000110000111111000000111101001111
```

32 bits

A 4-bit field, a 5-bit field, and a 12-bit field (BinaryStdOut)

```java
BinaryStdOut.write(month, 4);
BinaryStdOut.write(day, 5);
BinaryStdOut.write(year, 12);
```

```
110011111011111001111000
```

21 bits (+ 3 bits for byte alignment at close)

Four ways to put a date onto standard output
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
0100001001010010
0100000100100001
```

**Bitstream represented with hex digits**

```
% java HexDump 4 < abra.txt
41 42 52 41
43 41 44 41
42 52 41 21
```

**Bitstream represented as pixels in a Picture**

```
% java PictureDump 16 < abra.txt
```

---

Hexadecimal to ASCII conversion table

<table>
<thead>
<tr>
<th>Decimal</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
</tr>
<tr>
<td>1</td>
<td>SOH</td>
</tr>
<tr>
<td>2</td>
<td>STX</td>
</tr>
<tr>
<td>3</td>
<td>ETX</td>
</tr>
<tr>
<td>4</td>
<td>EOT</td>
</tr>
<tr>
<td>5</td>
<td>ENQ</td>
</tr>
<tr>
<td>6</td>
<td>ACK</td>
</tr>
<tr>
<td>7</td>
<td>BEL</td>
</tr>
<tr>
<td>8</td>
<td>BS</td>
</tr>
<tr>
<td>9</td>
<td>HT</td>
</tr>
<tr>
<td>10</td>
<td>LF</td>
</tr>
<tr>
<td>11</td>
<td>VT</td>
</tr>
<tr>
<td>12</td>
<td>FF</td>
</tr>
<tr>
<td>13</td>
<td>CR</td>
</tr>
<tr>
<td>14</td>
<td>SO</td>
</tr>
<tr>
<td>15</td>
<td>SI</td>
</tr>
<tr>
<td>16</td>
<td>DLE</td>
</tr>
<tr>
<td>17</td>
<td>DC1</td>
</tr>
<tr>
<td>18</td>
<td>DC2</td>
</tr>
<tr>
<td>19</td>
<td>DC3</td>
</tr>
<tr>
<td>20</td>
<td>DC4</td>
</tr>
<tr>
<td>21</td>
<td>NAK</td>
</tr>
<tr>
<td>22</td>
<td>SYN</td>
</tr>
<tr>
<td>23</td>
<td>ETB</td>
</tr>
<tr>
<td>24</td>
<td>CAN</td>
</tr>
<tr>
<td>25</td>
<td>EM</td>
</tr>
<tr>
<td>26</td>
<td>SUB</td>
</tr>
<tr>
<td>27</td>
<td>ESC</td>
</tr>
<tr>
<td>28</td>
<td>FS</td>
</tr>
<tr>
<td>29</td>
<td>GS</td>
</tr>
<tr>
<td>30</td>
<td>RS</td>
</tr>
<tr>
<td>31</td>
<td>US</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>33</td>
<td>!</td>
</tr>
<tr>
<td>34</td>
<td>#</td>
</tr>
<tr>
<td>35</td>
<td>$</td>
</tr>
<tr>
<td>36</td>
<td>%</td>
</tr>
<tr>
<td>37</td>
<td>&amp;</td>
</tr>
<tr>
<td>38</td>
<td>'</td>
</tr>
<tr>
<td>39</td>
<td>(</td>
</tr>
<tr>
<td>40</td>
<td>)</td>
</tr>
<tr>
<td>41</td>
<td>*</td>
</tr>
<tr>
<td>42</td>
<td>+</td>
</tr>
<tr>
<td>43</td>
<td>,</td>
</tr>
<tr>
<td>44</td>
<td>-</td>
</tr>
<tr>
<td>45</td>
<td>.</td>
</tr>
<tr>
<td>46</td>
<td>/</td>
</tr>
<tr>
<td>47</td>
<td>0</td>
</tr>
<tr>
<td>48</td>
<td>1</td>
</tr>
<tr>
<td>49</td>
<td>2</td>
</tr>
<tr>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>51</td>
<td>4</td>
</tr>
<tr>
<td>52</td>
<td>5</td>
</tr>
<tr>
<td>53</td>
<td>6</td>
</tr>
<tr>
<td>54</td>
<td>7</td>
</tr>
<tr>
<td>55</td>
<td>8</td>
</tr>
<tr>
<td>56</td>
<td>9</td>
</tr>
<tr>
<td>57</td>
<td>:</td>
</tr>
<tr>
<td>58</td>
<td>;</td>
</tr>
<tr>
<td>59</td>
<td>&lt;</td>
</tr>
<tr>
<td>60</td>
<td>=</td>
</tr>
<tr>
<td>61</td>
<td>&gt;</td>
</tr>
<tr>
<td>62</td>
<td>?</td>
</tr>
<tr>
<td>63</td>
<td>`</td>
</tr>
<tr>
<td>64</td>
<td>a</td>
</tr>
<tr>
<td>65</td>
<td>b</td>
</tr>
<tr>
<td>66</td>
<td>c</td>
</tr>
<tr>
<td>67</td>
<td>d</td>
</tr>
<tr>
<td>68</td>
<td>e</td>
</tr>
<tr>
<td>69</td>
<td>f</td>
</tr>
<tr>
<td>70</td>
<td>g</td>
</tr>
<tr>
<td>71</td>
<td>h</td>
</tr>
<tr>
<td>72</td>
<td>i</td>
</tr>
<tr>
<td>73</td>
<td>j</td>
</tr>
<tr>
<td>74</td>
<td>k</td>
</tr>
<tr>
<td>75</td>
<td>l</td>
</tr>
<tr>
<td>76</td>
<td>m</td>
</tr>
<tr>
<td>77</td>
<td>n</td>
</tr>
<tr>
<td>78</td>
<td>o</td>
</tr>
<tr>
<td>79</td>
<td>p</td>
</tr>
<tr>
<td>80</td>
<td>q</td>
</tr>
<tr>
<td>81</td>
<td>r</td>
</tr>
<tr>
<td>82</td>
<td>s</td>
</tr>
<tr>
<td>83</td>
<td>t</td>
</tr>
<tr>
<td>84</td>
<td>u</td>
</tr>
<tr>
<td>85</td>
<td>v</td>
</tr>
<tr>
<td>86</td>
<td>w</td>
</tr>
<tr>
<td>87</td>
<td>x</td>
</tr>
<tr>
<td>88</td>
<td>y</td>
</tr>
<tr>
<td>89</td>
<td>z</td>
</tr>
<tr>
<td>90</td>
<td>{</td>
</tr>
<tr>
<td>91</td>
<td></td>
</tr>
<tr>
<td>92</td>
<td>}</td>
</tr>
<tr>
<td>93</td>
<td>~</td>
</tr>
<tr>
<td>94</td>
<td>DEL</td>
</tr>
</tbody>
</table>
- binary I/O
- limitations
- genomic encoding
- run-length encoding
- Huffman compression
- LZW compression
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 a bitstring $B_0$, compress it to get a smaller bitstring $B_1$.
• Compress $B_1$ to get a smaller bitstring $B_2$.
• Continue until reaching a 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 $\leq 999$ bits.
• Similarly, only 1 in $2^{499}$ bitstrings can be encoded with $\leq 500$ bits!
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

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 demonstrate. In a publication of New Scientist you could randomise all the letters, keeping the first two and last two the same, and rebadalilty would hardly be aftcfeeed. My ansaylis did not come to much because 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.

<table>
<thead>
<tr>
<th>char</th>
<th>hex</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>41</td>
<td>01000001</td>
</tr>
<tr>
<td>C</td>
<td>43</td>
<td>01000011</td>
</tr>
<tr>
<td>T</td>
<td>54</td>
<td>01010100</td>
</tr>
<tr>
<td>G</td>
<td>47</td>
<td>01000111</td>
</tr>
</tbody>
</table>

**Two-bit encoding encoding.**
- 2 bits per char.
- 2N bits.

<table>
<thead>
<tr>
<th>char</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>00</td>
</tr>
<tr>
<td>C</td>
<td>01</td>
</tr>
<tr>
<td>T</td>
<td>10</td>
</tr>
<tr>
<td>G</td>
<td>11</td>
</tr>
</tbody>
</table>

**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

```java
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
010000010101010001000010101010001010100010001101000001
010000001001000001001000110100001110100001101000001
01010001010000010111011101010001100010000000010001
010100100001000011101101000001101000010001000011
010010000111010000000110101001000100010101000010001
01000011
264 bits

% java Genome - < genomeTiny.txt
??
```
cannot see bitstream on standard output
```

```
% java Genome - < genomeTiny.txt | java BinaryDump 64
00000000000000000000000100000010010010100101001000100010000000
100001000001010101000101010000100010000100010000110010001110001
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 1 1 1 1 1 1 1 0 0 0 0 0 0 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.

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

```java
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.
• \(300 \times 8.5 \times 300 \times 11 = 8.415\) million bits.

Observation. Bits are mostly white.

Typical amount of text on a page.
40 lines \(\times 75\) chars per line = 3,000 chars.
- 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.
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.

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>101</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1111</td>
</tr>
<tr>
<td>C</td>
<td>110</td>
</tr>
<tr>
<td>D</td>
<td>100</td>
</tr>
<tr>
<td>R</td>
<td>1110</td>
</tr>
</tbody>
</table>

```
A   B   R A  C A  D A   B   R A  !
01111111110011001000111111100101  30 bits
```

```
key  value
!  101
A  11
B  00
C  010
D  100
R  011

Compressed bitstring
11000111101011100110011100111101  29 bits
```

```
A   B   R A  C A  D A   B   R A  !
```

```
key  value
!  101
A  11
B  00
C  010
D  100
R  011

Compressed bitstring
11000111101011100110011100111101  29 bits
```
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.

Trie representation
Codeword table
key  value
A  0
B  1111
C  110
D  100
R  1110

Compressed bitstring
011111111001100011111111100101
A  B  RA  CA  DA  B  RA  !

30 bits

Trie representation
Codeword table
key  value
A  101
B  11
C  00
D  010
R  011

Compressed bitstring
1100011110101110110011000111111101
A  B  R  A  C  A  D  A  B  R  A  !

29 bits
Huffman trie node data type

```java
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();
}

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.

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.
Huffman codes

Q. How to find best prefix-free code?
A. Huffman algorithm.

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

<table>
<thead>
<tr>
<th>char</th>
<th>freq</th>
<th>encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>111</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1011</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>R</td>
<td>2</td>
<td>110</td>
</tr>
<tr>
<td>!</td>
<td>1</td>
<td>1010</td>
</tr>
</tbody>
</table>

Huffman code construction for A B R A C A D A B R A !
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();
}
Huffman encoding summary

**Proposition.** [Huffman 1950s] Huffman algorithm produces an optimal prefix-free code.

*Proof.* 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 $\implies O(N + R \log R)$.

**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

<table>
<thead>
<tr>
<th>input</th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>A</th>
<th>C</th>
<th>A</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>matches</td>
<td>A</td>
<td>B</td>
<td>R</td>
<td>A</td>
<td>C</td>
<td>A</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>R</td>
<td>A</td>
<td>B</td>
<td>R</td>
<td>A</td>
<td>B</td>
<td>R</td>
<td>A</td>
</tr>
<tr>
<td>value</td>
<td>41</td>
<td>42</td>
<td>52</td>
<td>41</td>
<td>43</td>
<td>41</td>
<td>44</td>
<td>81</td>
<td>83</td>
<td>82</td>
<td>88</td>
<td>41</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LZW compression for ABRACADABRABRABRA

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>A</td>
<td>41</td>
</tr>
<tr>
<td>B</td>
<td>42</td>
</tr>
<tr>
<td>C</td>
<td>43</td>
</tr>
<tr>
<td>D</td>
<td>44</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>81</td>
</tr>
<tr>
<td>BR</td>
<td>82</td>
</tr>
<tr>
<td>RA</td>
<td>83</td>
</tr>
<tr>
<td>AC</td>
<td>84</td>
</tr>
<tr>
<td>CA</td>
<td>85</td>
</tr>
<tr>
<td>AD</td>
<td>86</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DA</td>
<td>87</td>
</tr>
<tr>
<td>ABR</td>
<td>88</td>
</tr>
<tr>
<td>RAB</td>
<td>89</td>
</tr>
<tr>
<td>BRA</td>
<td>8A</td>
</tr>
<tr>
<td>ABRA</td>
<td>8B</td>
</tr>
</tbody>
</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
Q. How to represent LZW code table?
A. A trie: supports efficient longest prefix match.

Remark. Every prefix of a key in encoding table is also in encoding table.
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();
}
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.
Q. What to do when next codeword is not yet in ST when needed?

**LZW expansion: tricky situation**

<table>
<thead>
<tr>
<th>compression</th>
<th>input</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>matches</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>output</td>
<td>41</td>
<td>42</td>
<td>81</td>
<td></td>
<td>83</td>
<td></td>
<td>80</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>expansion</th>
<th>input</th>
<th>41</th>
<th>42</th>
<th>81</th>
<th></th>
<th>83</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**codeword table**

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>81</td>
</tr>
<tr>
<td>B</td>
<td>82</td>
</tr>
<tr>
<td>A B</td>
<td>83</td>
</tr>
</tbody>
</table>

**need lookahead character to complete entry**

Next character in output—the lookahead character!

Must be A B A (see below)
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.
  LZ77 not patented ⇒ widely used in open source
  LZ77 not patented ⇒ widely used in open source

• LZ78.

• LZW.
  LZW patent #4,558,302 expired in US on June 20, 2003
  some versions copyrighted

• Deflate = LZ77 variant + Huffman.

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

<table>
<thead>
<tr>
<th>Year</th>
<th>Scheme</th>
<th>Bits / Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>1967</td>
<td>ASCII</td>
<td>7.00</td>
</tr>
<tr>
<td>1950</td>
<td>Huffman</td>
<td>4.70</td>
</tr>
<tr>
<td>1977</td>
<td>LZ77</td>
<td>3.94</td>
</tr>
<tr>
<td>1984</td>
<td>LZMW</td>
<td>3.32</td>
</tr>
<tr>
<td>1987</td>
<td>LZH</td>
<td>3.30</td>
</tr>
<tr>
<td>1987</td>
<td>move-to-front</td>
<td>3.24</td>
</tr>
<tr>
<td>1987</td>
<td>LZB</td>
<td>3.18</td>
</tr>
<tr>
<td>1987</td>
<td>gzip</td>
<td>2.71</td>
</tr>
<tr>
<td>1988</td>
<td>PPMC</td>
<td>2.48</td>
</tr>
<tr>
<td>1994</td>
<td>SAKDC</td>
<td>2.47</td>
</tr>
<tr>
<td>1994</td>
<td>PPM</td>
<td>2.34</td>
</tr>
<tr>
<td>1995</td>
<td>Burrows-Wheeler</td>
<td>2.29</td>
</tr>
<tr>
<td>1997</td>
<td>BOA</td>
<td>1.99</td>
</tr>
<tr>
<td>1999</td>
<td>RK</td>
<td>1.89</td>
</tr>
</tbody>
</table>

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.
• 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 \(b\). \([ax + by = 1]\)
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 \(p_1 \rightarrow p_2 \rightarrow p_3\) a counterclockwise turn?

Other geometric shapes.
• Triangle, rectangle, circle, sphere, cone, ...
• 3D and higher dimensions sometimes more complicated.
Geometric intuition

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?

we think of this

algorithm sees this

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>6</th>
<th>5</th>
<th>8</th>
<th>7</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

|   | 1 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 |
|---|---|-----|-----|-----|-----|-----|-----|---|---|---|---|---|---|---|---|---|
| x |   | 1   | 18  | 4   | 19  | 4   | 19  | 4 | 20 | 3 | 20 | 2 | 20 |
| y | 1  | 2   | 18  | 4   | 18  | 4   | 19  | 4 | 19 | 4 | 20 | 3 | 20 | 2 | 20 |

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>10</th>
<th>3</th>
<th>7</th>
<th>2</th>
<th>8</th>
<th>8</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td>6</td>
<td>5</td>
<td>15</td>
<td>1</td>
<td>11</td>
<td>3</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>y</td>
<td>6</td>
<td>5</td>
<td>15</td>
<td>1</td>
<td>11</td>
<td>3</td>
<td>14</td>
<td>2</td>
<td>16</td>
</tr>
</tbody>
</table>
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?

**Application.** Draw a filled polygon on the screen.

http://www.ics.uci.edu/~eppstein/geom.html
Q. Does line segment intersect ray?

\[ y_0 = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} (x_0 - x_i) + y_i \]

\[ X_i \leq x_0 \leq X_{i+1} \]

```
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 points a, b, and c, is a-b-c a counterclockwise turn?

- Analog of compares in sorting.
- Idea: compare slopes.

![Diagram of ccw](image)

### Lesson

**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 \( \vec{a} \rightarrow \vec{b} \rightarrow \vec{c} \) a counterclockwise turn?

- Determinant gives twice signed area of triangle.

\[
2 \times \text{Area}(a, b, c) = \begin{vmatrix}
a_x & a_y & 1 \\
b_x & b_y & 1 \\
c_x & c_y & 1
\end{vmatrix} = (b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x)
\]

- If area \( > 0 \) then \( \vec{a} \rightarrow \vec{b} \rightarrow \vec{c} \) is counterclockwise.
- If area \( < 0 \), then \( \vec{a} \rightarrow \vec{b} \rightarrow \vec{c} \) is clockwise.
- If area \( = 0 \), then \( \vec{a} \rightarrow \vec{b} \rightarrow \vec{c} \) are collinear.
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;
    }
}
**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).

```java
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
Convex hull

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.

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.

[Image: 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 $pq$.

**$O(N^3)$ 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.
Package wrap (Jarvis march)

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. \( \Theta(Nh) \).

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:** $p[1], p[2], \ldots, p[N]$ are distinct points.
- **Output:** $M$ and rearrangement so that $p[1], p[2], \ldots, p[M]$ is convex hull.

```c
// 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
}
```

**Running time.** $O(N \log N)$ for sort and $O(N)$ for rest.
Quick elimination

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.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>running time</th>
<th>output sensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>package wrap</td>
<td>$N \cdot h$</td>
<td></td>
</tr>
<tr>
<td>Graham scan</td>
<td>$N \log N$</td>
<td></td>
</tr>
<tr>
<td>quickhull</td>
<td>$N \log N$</td>
<td></td>
</tr>
<tr>
<td>mergehull</td>
<td>$N \log N$</td>
<td></td>
</tr>
<tr>
<td>sweep line</td>
<td>$N \log N$</td>
<td></td>
</tr>
<tr>
<td>quick elimination</td>
<td>$N$</td>
<td></td>
</tr>
<tr>
<td>marriage-before-conquest</td>
<td>$N \log h$</td>
<td></td>
</tr>
</tbody>
</table>

$^\dagger$ 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)

  \[(a.x < b.x) \text{ || } ((a.x == b.x) \&\& (a.y < b.y))\]

• Quadratic decision tree model: compute any quadratic function of the coordinates and compare against 0.

  \[(a.x*b.y - a.y*b.x + a.y*c.x - a.x*c.y + b.x*c.y - c.x*b.y) < 0\]

Proposition. [Andy Yao, 1981] In quadratic decision tree model, any convex hull algorithm requires \(\Omega(N \log N)\) ops.

higher constant-degree polynomial tests don’t help either [Ben-Or, 1983]

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 \( \sim \frac{1}{2}N \) points on each side.
Divide-and-conquer algorithm

- **Divide**: draw vertical line $L$ so that $\sim \frac{1}{2}N$ 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.

\( \Theta(N^2) \)
How to find closest pair with one point in each side?

Find closest pair with one point in each side, assuming that distance $< \delta$.

$\delta = \min(12, 21)$
How to find closest pair with one point in each side?

Find closest pair with one point in each side, assuming that distance < $\delta$.
• Observation: only need to consider points within $\delta$ of line L.

$L$

$\delta = \min(12, 21)$
How to find closest pair with one point in each side?

Find closest pair with one point in each side, assuming that distance \( < \delta \).
- Observation: only need to consider points within \( \delta \) of line \( L \).
- Sort points in \( 2\delta \)-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!

\[ \delta = \min(12, 21) \]
How to find closest pair with one point in each side?

**Def.** Let $s_i$ be the point in the $2\delta$-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 $\delta$.

**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.
}
Divide-and-conquer algorithm: analysis

Running time recurrence. $T(N) \leq 2T(N/2) + O(N \log N)$.

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 diagram

**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 diagram

**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.

## Scientific rediscoveries

<table>
<thead>
<tr>
<th>year</th>
<th>discoverer</th>
<th>discipline</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1644</td>
<td>Descartes</td>
<td>astronomy</td>
<td>&quot;Heavens&quot;</td>
</tr>
<tr>
<td>1850</td>
<td>Dirichlet</td>
<td>math</td>
<td>Dirichlet tesselation</td>
</tr>
<tr>
<td>1908</td>
<td>Voronoi</td>
<td>math</td>
<td>Voronoi diagram</td>
</tr>
<tr>
<td>1909</td>
<td>Boldyrev</td>
<td>geology</td>
<td>area of influence polygons</td>
</tr>
<tr>
<td>1911</td>
<td>Thiessen</td>
<td>meteorology</td>
<td>Thiessen polygons</td>
</tr>
<tr>
<td>1927</td>
<td>Niggli</td>
<td>crystallography</td>
<td>domains of action</td>
</tr>
<tr>
<td>1933</td>
<td>Wigner-Seitz</td>
<td>physics</td>
<td>Wigner-Seitz regions</td>
</tr>
<tr>
<td>1958</td>
<td>Frank-Casper</td>
<td>physics</td>
<td>atom domains</td>
</tr>
<tr>
<td>1965</td>
<td>Brown</td>
<td>ecology</td>
<td>area of potentially available</td>
</tr>
<tr>
<td>1966</td>
<td>Mead</td>
<td>ecology</td>
<td>plant polygons</td>
</tr>
<tr>
<td>1985</td>
<td>Hoofd et al.</td>
<td>anatomy</td>
<td>capillary domains</td>
</tr>
</tbody>
</table>

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.

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.
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 $O(N \log N)$!
Ingenious algorithms enable solution of large instances for numerous fundamental geometric problems.

<table>
<thead>
<tr>
<th>problem</th>
<th>brute</th>
<th>clever</th>
</tr>
</thead>
<tbody>
<tr>
<td>convex hull</td>
<td>$N^2$</td>
<td>$N \log N$</td>
</tr>
<tr>
<td>farthest pair</td>
<td>$N^2$</td>
<td>$N \log N$</td>
</tr>
<tr>
<td>closest pair</td>
<td>$N^2$</td>
<td>$N \log N$</td>
</tr>
<tr>
<td>Delaunay/Voronoi</td>
<td>$N^4$</td>
<td>$N \log N$</td>
</tr>
<tr>
<td>Euclidean MST</td>
<td>$N^2$</td>
<td>$N \log N$</td>
</tr>
</tbody>
</table>

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).

<table>
<thead>
<tr>
<th>data structure</th>
<th>insert</th>
<th>rank</th>
<th>range count</th>
<th>range search</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordered array</td>
<td>$N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$R + \log N$</td>
</tr>
<tr>
<td>hash table</td>
<td>1</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>BST</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$R + \log N$</td>
</tr>
</tbody>
</table>

$N = \#$ 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).

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 horizontal-vertical 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.
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.

**Quadtrees.** Recursively divide space into four quadrants.

**2d tree.** Recursively divide space into two halfplanes.

**BSP tree.** Recursively divide space into two regions.
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.

Grid
Quadtree
2D tree
BSP tree
Quadtree

**Idea.** Recursively divide space into 4 quadrants.

**Implementation.** 4-way tree (actually a trie).

**Benefit.** Good performance in the presence of clustering.

**Drawback.** Arbitrary depth!

```java
public class QuadTree {
    private Quad quad;
    private Value val;
    private QuadTree NW, NE, SW, SE;
}
```
Quadtree: larger example

**Quadtrees:** 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.** $R + \log N$. 

![Diagram of a quadtree](image)
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
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). \( R + \sqrt{N} \).
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.
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.
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.
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. $O(R + N \log N)$

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

```java
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)
    {  ...  }
}
```

compare by x-coordinate;
break ties by y-coordinate

horizontal segment
vertical segment
Sweep-line event subclass

```java
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;  }
}
```
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);
  }
}
Sweep-line algorithm: simulate the sweep line

```java
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 ⇒ one new pair of adjacent segments.
- Intersection ⇒ 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.
- $O(R \log N + N \log N)$.

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.
Microprocessors and geometry

**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.**
"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. (!)

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

<table>
<thead>
<tr>
<th>operation</th>
<th>brute</th>
<th>interval search tree</th>
<th>best in theory</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert interval</td>
<td>1</td>
<td>log N</td>
<td>log N</td>
</tr>
<tr>
<td>delete interval</td>
<td>N</td>
<td>log N</td>
<td>log N</td>
</tr>
<tr>
<td>find an interval that intersects (lo, hi)</td>
<td>N</td>
<td>log N</td>
<td>log N</td>
</tr>
<tr>
<td>find all intervals that intersects (lo, hi)</td>
<td>N</td>
<td>R log N</td>
<td>R + log N</td>
</tr>
</tbody>
</table>

N = # intervals
R = # intersections

![Augmented red-black tree](image)
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). \( O(N \log N) \)
- Insert y-interval into ST. \( O(N \log N) \)
- 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

<table>
<thead>
<tr>
<th>1D range search</th>
<th>... ... ... ...</th>
<th>BST</th>
</tr>
</thead>
<tbody>
<tr>
<td>kD range search</td>
<td><img src="image" alt="kd tree" /></td>
<td>kD tree</td>
</tr>
<tr>
<td>1D interval intersection search</td>
<td><img src="image" alt="interval search tree" /></td>
<td>interval search tree</td>
</tr>
<tr>
<td>2D orthogonal line intersection search</td>
<td><img src="image" alt="sweep line reduces to 1D range search" /></td>
<td>sweep line reduces to 1D range search</td>
</tr>
<tr>
<td>2D orthogonal rectangle intersection search</td>
<td><img src="image" alt="sweep line reduces to 1D interval intersection search" /></td>
<td>sweep line reduces to 1D interval intersection search</td>
</tr>
</tbody>
</table>
7.5 Reductions

- designing algorithms
- establishing lower bounds
- intractability
Desiderata. Classify problems according to computational requirements.

<table>
<thead>
<tr>
<th>complexity</th>
<th>order of growth</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear</td>
<td>N</td>
<td>min, max, median, Burrows-Wheeler transform, ...</td>
</tr>
<tr>
<td>linearithmic</td>
<td>N log N</td>
<td>sorting, convex hull, closest pair, farthest pair, ...</td>
</tr>
<tr>
<td>quadratic</td>
<td>N²</td>
<td>???</td>
</tr>
<tr>
<td>exponential</td>
<td>cN</td>
<td>???</td>
</tr>
</tbody>
</table>
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.

\[
\text{Cost of solving } X = \text{total cost of solving } Y + \text{cost of reduction.}
\]

- 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.* \( N \log N + N \)
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 solving 3-collinear. $N^2 \log N + N^2$.  

Cost of sorting $N^2 \log N$.  

Cost of reduction $N^2$.  

‣ 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).

**Proposition.** Convex hull reduces to sorting.

**Pf.** Graham scan algorithm.

**Cost of convex hull.** \( N \log N + N. \)
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

- Furthest pair 2d
- Convex hull
- Median
- Element distinctness
- Closest pair 2d
- Euclidean MST 2d
- Delaunay triangulation
- Bipartite matching
- Directed shortest paths (nonnegative)
- Arbitrage
- Shortest paths (no neg cycles)
- Maximum flow
- Linear programming
• designing algorithms
• linear programming
• establishing lower bounds
• establishing intractability
• classifying problems
Linear Programming

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.

Present context. Many important problems reduce to LP.

Ex: Delta claims that LP saves $100 million per year.
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.


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.
Linear programming

Model problem as maximizing an objective function subject to constraints.

Input: real numbers $a_{ij}$, $c_j$, and $b_i$.

Output: real numbers $x_j$.

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 ]
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: \( x_i = \text{length of shortest path from } s \text{ to } i \).

\[
\begin{align*}
\text{maximize} & \quad x_t \\
\text{subject to} \quad x_s + 9 & \geq x_2 \\
\quad x_s + 14 & \geq x_6 \\
\quad x_s + 15 & \geq x_7 \\
\quad x_2 + 24 & \geq x_3 \\
\quad x_3 + 2 & \geq x_5 \\
\quad x_3 + 19 & \geq x_t \\
\quad x_4 + 6 & \geq x_3 \\
\quad x_4 + 6 & \geq x_t \\
\quad x_5 + 11 & \geq x_4 \\
\quad x_5 + 16 & \geq x_t \\
\quad x_6 + 18 & \geq x_3 \\
\quad x_6 + 30 & \geq x_5 \\
\quad x_6 + 5 & \geq x_7 \\
\quad x_7 + 20 & \geq x_5 \\
\quad x_7 + 44 & \geq x_t \\
\quad x_s &= 0
\end{align*}
\]
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 \( i \).

\[
\begin{align*}
\text{maximize} & \quad x_t \\
\text{subject to the constraints} & \\
& x_s + 9 \geq x_2 \\
& x_s + 14 \geq x_6 \\
& x_s + 15 \geq x_7 \\
& x_2 + 24 \geq x_3 \\
& x_3 + 2 \geq x_5 \\
& x_3 + 19 \geq x_t \\
& x_4 + 6 \geq x_3 \\
& x_4 + 6 \geq x_t \\
& x_5 + 11 \geq x_4 \\
& x_5 + 16 \geq x_t \\
& x_6 + 18 \geq x_3 \\
& x_6 + 30 \geq x_5 \\
& x_6 + 5 \geq x_7 \\
& x_7 + 20 \geq x_5 \\
& x_7 + 44 \geq x_t \\
& x_s = 0
\end{align*}
\]

Solution:
- \( x_s = 0 \)
- \( x_2 = 9 \)
- \( x_3 = 32 \)
- \( x_4 = 45 \)
- \( x_5 = 34 \)
- \( x_6 = 14 \)
- \( x_7 = 15 \)
- \( x_t = 50 \)
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.

<table>
<thead>
<tr>
<th>maximize</th>
<th>$x_{3t} + x_{4t}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>subject</td>
<td></td>
</tr>
<tr>
<td>to the constraints</td>
<td></td>
</tr>
<tr>
<td>$x_{s1}$ ≤ 2</td>
<td></td>
</tr>
<tr>
<td>$x_{s2}$ ≤ 3</td>
<td></td>
</tr>
<tr>
<td>$x_{13}$ ≤ 3</td>
<td></td>
</tr>
<tr>
<td>$x_{14}$ ≤ 1</td>
<td></td>
</tr>
<tr>
<td>$x_{23}$ ≤ 1</td>
<td></td>
</tr>
<tr>
<td>$x_{24}$ ≤ 1</td>
<td></td>
</tr>
<tr>
<td>$x_{3t}$ ≤ 2</td>
<td></td>
</tr>
<tr>
<td>$x_{4t}$ ≤ 3</td>
<td></td>
</tr>
</tbody>
</table>

**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 $x_{ij} ≥ 0$**

**interpretation:**
$x_{ij} = \text{flow in edge } i-j$

**Diagram:**
- Add dummy edge from $t$ to $s$.
- Equilibrium constraints.
Maxflow problem reduces to LP

One variable per edge.
One inequality per edge, one equality per vertex.

<table>
<thead>
<tr>
<th>maximize</th>
<th>$x_{3t} + x_{4t}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>subject to the</td>
<td></td>
</tr>
<tr>
<td>constraints</td>
<td>$x_{s1} \leq 2$</td>
</tr>
<tr>
<td></td>
<td>$x_{s2} \leq 3$</td>
</tr>
<tr>
<td></td>
<td>$x_{13} \leq 3$</td>
</tr>
<tr>
<td></td>
<td>$x_{14} \leq 1$</td>
</tr>
<tr>
<td></td>
<td>$x_{23} \leq 1$</td>
</tr>
<tr>
<td></td>
<td>$x_{24} \leq 1$</td>
</tr>
<tr>
<td></td>
<td>$x_{3t} \leq 2$</td>
</tr>
<tr>
<td></td>
<td>$x_{4t} \leq 3$</td>
</tr>
</tbody>
</table>

interpretation:
$x_{ij} = \text{flow in edge } i-j$

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 $x_{ij} \geq 0$

capacity constraints

solution

$x_{s1} = 2$
$x_{s2} = 2$
$x_{13} = 1$
$x_{14} = 1$
$x_{23} = 1$
$x_{24} = 1$
$x_{3t} = 2$
$x_{4t} = 2$

add dummy edge from $t$ to $s$
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.

**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 $x_i = 1$.

<table>
<thead>
<tr>
<th>maximize</th>
<th>$x_{A0} + x_{A1} + x_{A2} + x_{B0} + x_{B1} + x_{B5} + x_{C2} + x_{C3} + x_{C4}$ $+ x_{D0} + x_{D1} + x_{E3} + x_{E4} + x_{E5} + x_{F2} + x_{F4} + x_{F5}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>subject to the constraints</td>
<td>$x_{A0} + x_{A1} + x_{A2} = 1$ $x_{A0} + x_{B0} + x_{D0} = 1$ $x_{A1} + x_{B1} + x_{D1} = 1$ $x_{A2} + x_{C2} + x_{F2} = 1$ $x_{C3} + x_{E3} = 1$ $x_{C4} + x_{E4} + x_{F4} = 1$ $x_{B5} + x_{E5} + x_{F5} = 1$ $x_{E3} + x_{E4} + x_{E5} = 1$ $x_{F2} + x_{F4} + x_{F5} = 1$</td>
</tr>
<tr>
<td>all $x_{ij} \geq 0$</td>
<td>constraints on top vertices (left) and bottom vertices (right)</td>
</tr>
</tbody>
</table>

**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 $x_i = 1$.

\[
\text{maximize} \quad x_{A0} + x_{A1} + x_{A2} + x_{B0} + x_{B1} + x_{B5} + x_{C2} + x_{C3} + x_{C4} + x_{D0} + x_{D1} + x_{E3} + x_{E4} + x_{E5} + x_{F2} + x_{F4} + x_{F5}
\]

subject to the constraints

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_{A0} + x_{A1} + x_{A2} = 1$</td>
<td>$x_{A0} + x_{B0} + x_{D0} = 1$</td>
</tr>
<tr>
<td>$x_{B0} + x_{B1} + x_{B5} = 1$</td>
<td>$x_{A1} + x_{B1} + x_{D1} = 1$</td>
</tr>
<tr>
<td>$x_{C2} + x_{C3} + x_{C4} = 1$</td>
<td>$x_{A2} + x_{C2} + x_{F2} = 1$</td>
</tr>
<tr>
<td>$x_{D0} + x_{D1} = 1$</td>
<td>$x_{C3} + x_{E3} = 1$</td>
</tr>
<tr>
<td>$x_{E3} + x_{E4} + x_{E5} = 1$</td>
<td>$x_{C4} + x_{E4} + x_{F4} = 1$</td>
</tr>
<tr>
<td>$x_{F2} + x_{F4} + x_{F5} = 1$</td>
<td>$x_{B5} + x_{E5} + x_{F5} = 1$</td>
</tr>
</tbody>
</table>

all $x_{ij} \geq 0$

**Solution**

- $x_{A1} = 1$
- $x_{B5} = 1$
- $x_{C2} = 1$
- $x_{D0} = 1$
- $x_{E3} = 1$
- $x_{E4} = 1$
- 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
**Goal.** Prove that a problem requires a certain number of steps.

**Ex.** $\Omega(N \log N)$ lower bound for sorting.

**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

argument must apply to all conceivable algorithms
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 $Y$. 
Lower bound for convex hull

**Proposition.** In quadratic decision tree model, any algorithm for sorting \( N \) integers requires \( \Omega(N \log N) \) steps.

allows quadratic tests of the form:

\[ x_i < x_j \text{ or } (x_j - x_i)(x_k - x_i) - (x_j)(x_j - x_i) < 0 \]

**Proposition.** Sorting linear-time reduces to convex hull.

**Pf.** [see next slide]

**Implication.** Any ccw-based convex hull algorithm requires \( \Omega(N \log N) \) ccw's.
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?

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
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.
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:

\[
0 = \begin{vmatrix} a & a^3 & 1 \\ b & b^3 & 1 \\ c & c^3 & 1 \end{vmatrix}
= a(b^3 - c^3) - b(a^3 - c^3) + c(a^3 - b^3)
= (a - b)(b - c)(c - a)(a + b + c)
\]
More linear-time reductions and lower bounds

- **element distinctness**  
  \((N \log N\) lower bound)

- 3-sum  
  (conjectured \(N^2\) lower bound)

- **sorting**
- **closest pair 2d**
- **convex hull 2d**
- **Euclidean MST 2d**

- **3-collinear**
- **3-concurrent**
- **dihedral rotation**
- **min area triangle**

- **Delaunay**
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?
A2. [easy way] Linear-time reduction from sorting.

Q. How to convince yourself no sub-quadratic 3-COLLINEAR algorithm exists.
A2. [easy way] Linear-time reduction from 3-SUM.
designing algorithms
establishing lower bounds
intractability
**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.**
- *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?

**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 \( \Phi \) 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

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>( x_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

\[
(\neg T \lor T \lor F) \land (T \lor \neg T \lor F) \land (\neg T \lor \neg T \lor \neg F) \land (\neg T \lor \neg T \lor T) \land (\neg T \lor F \lor T)
\]

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 ($P \neq NP$). 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 $k$.

**Applications.** Scheduling, computer vision, clustering, ...
Proposition. 3-SAT poly-time reduces to IND-SET.

Pf. Given an instance $\Phi$ of 3-SAT, create an instance $G$ of IND-SET:
- For each clause in $\Phi$, create 3 vertices in a triangle.
- Add an edge between each literal and its negation.

$\Phi = (x_1 \lor x_2 \lor x_3) \land (\neg x_1 \lor \neg x_2 \lor x_4) \land (\neg x_1 \lor x_3 \lor \neg x_4) \land (x_1 \lor x_3 \lor x_4)$
Proposition. 3-SAT poly-time reduces to IND-SET.

Pf. Given an instance $\Phi$ of 3-SAT, create an instance $G$ of IND-SET:
- For each clause in $\Phi$, create 3 vertices in a triangle.
- Add an edge between each literal and its negation.

$\Phi = (x_1 \lor x_2 \lor x_3) \land (\neg x_1 \lor \neg x_2 \lor x_4) \land (\neg x_1 \lor x_3 \lor \neg x_4) \land (x_1 \lor x_3 \lor x_4)$

- $G$ has independent set of size $k \Rightarrow \Phi$ satisfiable.
3-satisfiability reduces to independent set

Proposition. 3-SAT poly-time reduces to IND-SET.

Pf. Given an instance $\Phi$ of 3-SAT, create an instance $G$ of IND-SET:

- For each clause in $\Phi$, create 3 vertices in a triangle.
- Add an edge between each literal and its negation.

$$k = 4$$

$$\Phi = (x_1 \lor x_2 \lor x_3) \land (\neg x_1 \lor \neg x_2 \lor x_4) \land (\neg x_1 \lor x_3 \lor \neg x_4) \land (x_1 \lor x_3 \lor x_4)$$

- $G$ has independent set of size $k \Rightarrow \Phi$ satisfiable.
- $\Phi$ 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.

\[ \Phi = (x_1 \vee x_2 \vee x_3) \land (\neg x_1 \vee \neg x_2 \vee x_4) \land (\neg x_1 \vee x_3 \vee \neg x_4) \land (x_1 \vee x_3 \vee x_4) \]
**Integer linear programming**

**ILP.** Given a system of linear inequalities, find an integral solution.

\[
\begin{align*}
3x_1 + 5x_2 + 2x_3 + x_4 + 4x_5 & \geq 10 \\
5x_1 + 2x_2 + 4x_4 + 1x_5 & \leq 7 \\
x_1 + x_3 + 2x_4 & \leq 2 \\
3x_1 + 4x_3 + 7x_4 & \leq 7 \\
x_1 + x_4 & \leq 1 \\
x_1 + x_3 + x_5 & \leq 1 \\
\text{all } x_i & = \{ 0, 1 \}
\end{align*}
\]

**Context.** Cornerstone problem in operations research.

**Remark.** Finding a real-valued solution is tractable (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.

\[
\begin{align*}
x_1 + x_2 + x_3 + x_4 + x_5 &= 3 \\
x_1 + x_2 &\leq 1 \\
x_2 + x_3 &\leq 1 \\
x_1 + x_3 &\leq 1 \\
x_1 + x_4 &\leq 1 \\
x_3 + x_5 &\leq 1 \\
\text{all } x_i &= \{0, 1\}
\end{align*}
\]
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

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 problems

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.

\[ \{ v_2, \ x_4, \ v_5 \} \]

\[ k = 3 \]
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.
Cook's theorem

**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.
Implications of Cook’s theorem

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.

<table>
<thead>
<tr>
<th>complexity</th>
<th>order of growth</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear</td>
<td>$N$</td>
<td>min, max, median, Burrows-Wheeler transform, ...</td>
</tr>
<tr>
<td>linearithmic</td>
<td>$N \log N$</td>
<td>sorting, convex hull. closest pair, farthest pair, ...</td>
</tr>
<tr>
<td>quadratic</td>
<td>$N^2$</td>
<td>???</td>
</tr>
<tr>
<td>exponential</td>
<td>$c^N$</td>
<td>???</td>
</tr>
</tbody>
</table>

**Frustrating news.** Huge number of problems have defied classification.
**Birds-eye view: revised**

**Desiderata.** Classify problems according to computational requirements.

<table>
<thead>
<tr>
<th>complexity</th>
<th>order of growth</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear</td>
<td>$N$</td>
<td>min, max, median, Burrows-Wheeler transform, ...</td>
</tr>
<tr>
<td>linearithmic</td>
<td>$N \log N$</td>
<td>sorting, convex hull, closest pair, farthest pair, ...</td>
</tr>
<tr>
<td>3-SUM complete</td>
<td>probably $N^2$</td>
<td>3-SUM, 3-COLLINEAR, 3-CONCURRENT, ...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>NP-complete</td>
<td>probably $c^N$</td>
<td>3-SAT, IND-SET, ILP, ...</td>
</tr>
</tbody>
</table>

**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 ⇒ effectiveness may be limited to relatively small instances.

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.

```java
private void enumerate(int k) {
  if (k == N) {
    process(); return;
  }
  enumerate(k+1);
  a[k] = 1;
  enumerate(k+1);
  a[k] = 0;
}
```

**Remark.** Equivalent to counting in binary from 0 to $2^N - 1$. 

N = 3

```
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
```

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
```
Warmup: enumerate N-bit strings

```java
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;
    }
}

class Main {
    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
```
• 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 ⇒ 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)

<table>
<thead>
<tr>
<th>N = 2</th>
<th>N = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>0 1 2</td>
</tr>
<tr>
<td>1 0</td>
<td>0 1 2</td>
</tr>
<tr>
<td>0 1</td>
<td>0 1 2</td>
</tr>
<tr>
<td>1 0 2</td>
<td>0 2 1</td>
</tr>
<tr>
<td>1 0 2</td>
<td>0 2 1</td>
</tr>
<tr>
<td>0 1 2</td>
<td>0 1 2</td>
</tr>
<tr>
<td>2 1 0</td>
<td>2 1 0</td>
</tr>
<tr>
<td>2 1 0</td>
<td>2 1 0</td>
</tr>
<tr>
<td>2 0 1</td>
<td>2 0 1</td>
</tr>
<tr>
<td>2 0 1</td>
<td>2 0 1</td>
</tr>
<tr>
<td>0 1 2</td>
<td>0 1 2</td>
</tr>
<tr>
<td>0 1 2</td>
<td>0 1 2</td>
</tr>
</tbody>
</table>

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

cleanup swaps that bring perm back to original

\[ a[0] \quad a[N-1] \]
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)

```java
// 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);
    }
}
```

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

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);
    }
}
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
... 

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 ⇒ 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

diagonal conflict on partial solution: no point going deeper

solutions
4-queens search tree (pruned)

"backtrack" on diagonal conflicts

solutions
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

```java
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);
    }
}
```

```text
% 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
```
Pruning the search tree leads to enormous time savings.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$Q(N)$</th>
<th>$N!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>120</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>720</td>
</tr>
<tr>
<td>7</td>
<td>40</td>
<td>5,040</td>
</tr>
<tr>
<td>8</td>
<td>92</td>
<td>40,320</td>
</tr>
<tr>
<td>9</td>
<td>352</td>
<td>362,880</td>
</tr>
<tr>
<td>10</td>
<td>724</td>
<td>3,628,800</td>
</tr>
<tr>
<td>11</td>
<td>2,680</td>
<td>39,916,800</td>
</tr>
<tr>
<td>12</td>
<td>14,200</td>
<td>479,001,600</td>
</tr>
<tr>
<td>13</td>
<td>73,712</td>
<td>6,227,020,800</td>
</tr>
<tr>
<td>14</td>
<td>365,596</td>
<td>87,178,291,200</td>
</tr>
</tbody>
</table>

$N$-queens problem: effectiveness of backtracking
N-queens problem: How many solutions?

Hypothesis. Running time is about \(\frac{N!}{2.5^N}\) / 43,000 seconds.

Conjecture. \(Q(N) \approx N! / c^N\), where \(c\) is about 2.54.
 › permutations
 › backtracking
 › counting
 › subsets
 › paths in a graph
Counting: Java implementation

**Goal.** Enumerate all N-digit base-R numbers.

**Solution.** Generalize binary counter in lecture warmup.

```java
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?
}
```
Goal. Fill 9-by-9 grid so that every row, column, and box contains each of the digits 1 through 9.

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.

![Sudoku Grid]

**Solution.** Enumerate all 81-digit base-9 numbers (with backtracking).
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.

backtrack on 3, 4, 5, 7, 8, 9
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;
}
• permutations
• backtracking
• counting
• subsets
• paths in a graph
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.

<table>
<thead>
<tr>
<th>$i$</th>
<th>binary</th>
<th>subset</th>
<th>complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>empty</td>
<td>11111</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
<td>11110</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
<td>11101</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>21</td>
<td>11011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>3</td>
<td>10111</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>31</td>
<td>10101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>32</td>
<td>10011</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>321</td>
<td>10001</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>4</td>
<td>01111</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>41</td>
<td>01101</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>42</td>
<td>01011</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>421</td>
<td>01001</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>43</td>
<td>01000</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>431</td>
<td>01010</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>432</td>
<td>01100</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>4321</td>
<td>empty</td>
</tr>
</tbody>
</table>
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.

```java
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.

<table>
<thead>
<tr>
<th>code</th>
<th>subset</th>
<th>move</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>empty</td>
<td></td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>1</td>
<td>enter 1</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>2 1</td>
<td>enter 2</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>2</td>
<td>exit 1</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>3 2</td>
<td>enter 3</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>3 2 1</td>
<td>enter 1</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>3 1</td>
<td>exit 2</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>3</td>
<td>exit 1</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>4 3</td>
<td>enter 4</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>4 3 1</td>
<td>enter 1</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>4 3 2 1</td>
<td>enter 2</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>4 3 2</td>
<td>exit 1</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>4 2</td>
<td>exit 3</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>4 2 1</td>
<td>enter 1</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>4 1</td>
<td>exit 2</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>4</td>
<td>exit 1</td>
</tr>
</tbody>
</table>
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
**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**

```java
private void enumerate(int k)
{
    if (k == N)
    {  process(); return;  }
    enumerate(k+1);
    a[k] = 1 - a[k];
    enumerate(k+1);
}
```

**standard binary counter (from warmup)**

```java
private void enumerate(int k)
{
    if (k == N)
    {  process(); return;  }
    enumerate(k+1);
    a[k] = 1;
    enumerate(k+1);
    a[k] = 0;
}
```

---

**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 (set partitioning). Given \( n \) jobs of varying length, divide among two machines to minimize the makespan (time the last job finishes).

### Table

<table>
<thead>
<tr>
<th>job</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.41</td>
</tr>
<tr>
<td>1</td>
<td>1.73</td>
</tr>
<tr>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td>3</td>
<td>2.23</td>
</tr>
</tbody>
</table>

### Diagram

- Machine 0: Jobs 0 and 2
- Machine 1: Jobs 1 and 3

Cost: 0.09

**Remark.** This scheduling problem is NP-complete.
Scheduling (full implementation)

```java
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
```
Observation. Large number of subsets leads to remarkably low cost.
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)

```java
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.

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.
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;
}
**Hamilton path**

**Goal.** Find a simple path that visits every vertex 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.

**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 (!!)
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;
    }
}
## Exhaustive search: summary

<table>
<thead>
<tr>
<th>problem</th>
<th>enumeration</th>
<th>backtracking</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-rooks</td>
<td>permutations</td>
<td>no</td>
</tr>
<tr>
<td>N-queens</td>
<td>permutations</td>
<td>yes</td>
</tr>
<tr>
<td>Sudoku</td>
<td>base-9 numbers</td>
<td>yes</td>
</tr>
<tr>
<td>scheduling</td>
<td>subsets</td>
<td>yes</td>
</tr>
<tr>
<td>Boggle</td>
<td>paths in a grid</td>
<td>yes</td>
</tr>
<tr>
<td>Hamilton path</td>
<td>paths in a graph</td>
<td>yes</td>
</tr>
</tbody>
</table>
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