Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

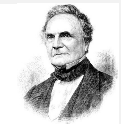
1.4 Analysis of Algorithms

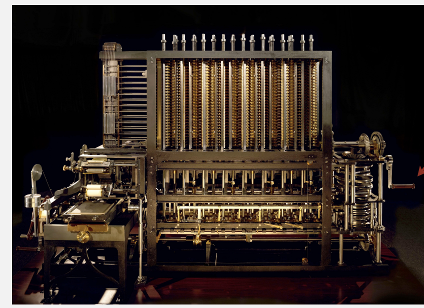
- *introduction*
- *observations*
- *mathematical models*
- *order-of-growth classifications*
- *memory*

Last updated on 2/3/16 9:16 AM

Running time

" As soon as an Analytical Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise—By what course of calculation can these results be arrived at by the machine in the shortest time? " — *Charles Babbage (1864)*





how many times do you have to turn the crank?

Running time

Concerns about running time preceded actual working computers by almost a century!

One of the early achievements of computer science

The ability to estimate and bound the running time of a piece of code
as a function of the size of the input
without seeing the actual input data
and only minimal knowledge of the system it will run on

i>clicker

Required device for lecture.

- Use default frequency AA.
- Available at Labyrinth Books ($25).
- You must register your i▸clicker in Blackboard.
  (sorry, insufficient WiFi in this room to support i▸clicker GO)

Is this just a plot to get you to buy more devices?
Let's find out!

Which model of i▸clicker are you using?

A. i▸clicker.

B. i▸clicker+.

C. i▸clicker 2.

D. All of the above

E. I am a conscientious objector

F. I feel constrained by the limited choices in this poll

http://algs4.cs.princeton.edu

1.4 Analysis of Algorithms

Reasons to analyze algorithms

Predict performance.

Compare algorithms.

Provide guarantees.

Understand theoretical basis.

this course (COS 226)

theory of algorithms (COS 423)

Primary practical reason: avoid performance bugs.

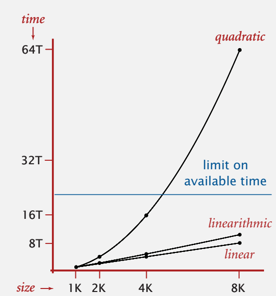
client gets poor performance because programmer did not understand performance characteristics

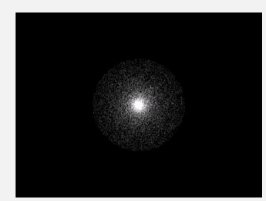
Efficient algorithms enable new discoveries

n-body simulation.

- Simulate gravitational interactions among n bodies.
- Applications: cosmology, fluid dynamics, semiconductors, ...
- Brute force: n^2 steps.
- Barnes-Hut algorithm: $n \log n$ steps, enables new research.

Andrew Appel
PU '81



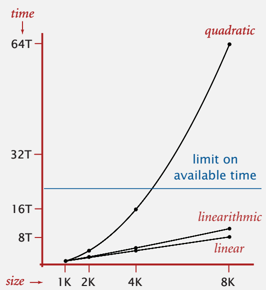


Efficient algorithms enable new products

Discrete Fourier transform.

- Express signal as weighted sum of sines and cosines.
- Applications: DVD, JPEG, MRI, astrophysics, ...
- Brute force: n^2 steps.
- FFT algorithm: $n \log n$ steps, enables new technology.


James Cooley
John Tukey

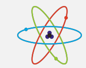


Scientific method applied to the analysis of algorithms

A framework for predicting performance and comparing algorithms.

Scientific method.

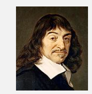
- Observe some feature of the natural world.
- Hypothesize a model that is consistent with the observations.
- Predict events using the hypothesis.
- Verify the predictions by making further observations.
- Validate by repeating until the hypothesis and observations agree.

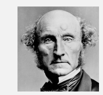


Principles.

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


Francis Bacon


René Descartes


John Stuart Mills

Feature of the natural world. Computer itself.

http://algs4.cs.princeton.edu

1.4 Analysis of Algorithms

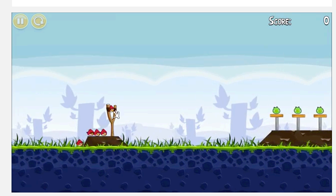
- introduction
- **observations**
- mathematical models
- order-of-growth classifications
- memory

Example: 3-Sum

3-Sum. Given n distinct integers, how many triples sum to exactly zero?

```
Input:
30 -40 -20 -10 40 0 10 5
Output:
4
```

	a[i]	a[j]	a[k]	sum
1	30	-40	10	0
2	30	-20	-10	0
3	-40	40	0	0
4	-10	0	10	0



Context. Deeply related to problems in computational geometry.

3-SUM: brute-force algorithm

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

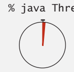
← check each triple

Ignore integer overflow in computing a[i] + a[j] + a[k]

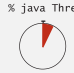
Measuring the running time

Q. How to time a program?
A. Manual.

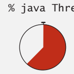
```
% java ThreeSum 1Kints.txt
```




```
70
% java ThreeSum 2Kints.txt
```



```
528
% java ThreeSum 4Kints.txt
```



```
4039
```


Measuring the running time

Q. How to time a program?
A. Automatic.

public class Stopwatch (part of stdlib.jar)		
	Stopwatch()	*create a new stopwatch*
double	elapsedTime()	*time since creation (in seconds)*

```
public static void main(String[] args)
{
   In in = new In(args[0]);
   int[] a = in.readAllInts();
   Stopwatch stopwatch = new Stopwatch();
   StdOut.println(ThreeSum.count(a));
   double time = stopwatch.elapsedTime();
   StdOut.println("elapsed time = " + time);
}
```


Empirical analysis

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

```
%
```

Empirical analysis

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

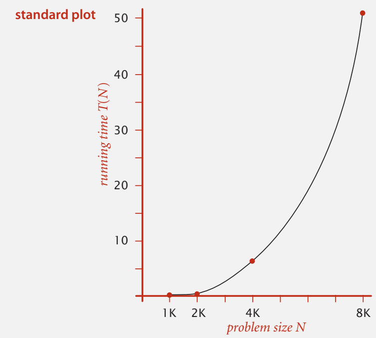
n	time (seconds) †
250	0
500	0
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1
16,000	?

† on a 2.8GHz Intel PU-226 with 64GB DDR E3 memory and 32MB L3 cache; running Oracle Java 1.7.0_45-b18 on Springdale Linux v. 6.5

(not consistent with prev. slide)

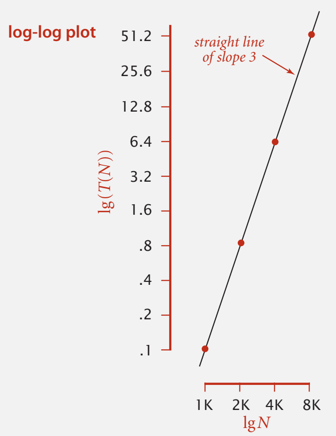
Data analysis

Standard plot. Plot running time $T(n)$ vs. input size n.



Data analysis

Log-log plot. Plot running time $T(n)$ vs. input size n using log-log scale.



$T(n) = a\, n^{b}$

$\lg(T(n)) = b \lg n + lg(a)$

Slope = b

y-intercept = $lg(a)$

Hypothesis. The running time is $\sim 1.006 \times 10^{-10} \times n^{2.999}$ seconds.

lg = base 2 logarithm

Prediction and validation

Hypothesis. The running time is about $1.006 \times 10^{-10} \times n^{2.999}$ seconds.

"order of growth" of running time is about n^3 [stay tuned]

Predictions.

- 51.0 seconds for $n = 8{,}000$.
- 408.1 seconds for $n = 16{,}000$.

Observations.

n	time (seconds) †
8,000	51.1
8,000	51
8,000	51.1
16,000	410.8

validates hypothesis!

Doubling hypothesis

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

Run program, doubling the size of the input.

n	time (seconds) †	ratio	lg ratio
250	0		–
500	0	4.8	2.3
1,000	0.1	6.9	2.8
2,000	0.8	7.7	2.9
4,000	6.4	8	3
8,000	51.1	8	3

$$\frac{T(N)}{T(N/2)} = \frac{aN^b}{a(N/2)^b} = 2^b$$

lg (6.4 / 0.8) = 3.0

seems to converge to a constant b ≈ 3

Hypothesis. Running time is about $a\, n^b$ with b = lg ratio.
Caveat. Cannot identify logarithmic factors with doubling hypothesis.

Doubling hypothesis

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

Q. How to estimate a (assuming we know b) ?
A. Run the program (for a sufficient large value of n) and solve for a.

n	time (seconds) †
8,000	51.1
8,000	51
8,000	51.1

$51.1 = a \times 8000^3$

$\Rightarrow a = 0.998 \times 10^{-10}$

Hypothesis. Running time is about $0.998 \times 10^{-10} \times n^3$ seconds.

almost identical hypothesis
to one obtained via log-log plot

Analysis of algorithms quiz 1

Estimate the running time to solve a problem of size n = 96,000.

A. *39 seconds.*

B. *52 seconds.*

C. *117 seconds.*

D. *350 seconds.*

E. *I don't know.*

n	time (seconds) †
1000	0.02
2000	0.05
4,000	0.2
8,000	0.81
16,000	3.25
32,000	13

Two surprises

Approximate running time is a simple mathematical expression
- Generally holds true even for much more complex programs!

Running time on different systems differs only by a constant factor!
- Running time on system 1: $a_1 n^b$
- Running time on system 2: $a_2 n^b$

Experimental algorithmics

System independent effects.

- Algorithm.
- (Rarely) Input data.

determines exponent b in power law $a\,n^b$

System dependent effects.

- Hardware: CPU, memory, cache, …
- Software: compiler, interpreter, garbage collector, …
- System: operating system, network, other apps, …
- Input data

determines constant a in power law $a\,n^b$

Theorist vs. pragmatist view of algorithmic efficiency

$a\,n^b$ ← property of algorithm

↑ system-dependent

Theorist: Worrying about constant factors is tedious and crass!
The asymptotic efficiency of an algorithm is a mathematical fact.
I study properties of the universe. The computer is irrelevant!

Novice: My program ran in 3 seconds on my laptop when I fed it data.
That's pretty good, right?

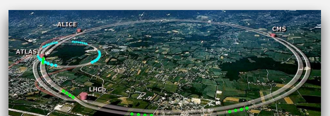
Pragmatist: I will use math. model to compute *b*, then verify empirically.
I will use a combination of math and observation to estimate *a*.

Bad news. Sometimes difficult to get precise measurements.
Good news. Much easier and cheaper than other sciences.

An aside

Algorithmic experiments are virtually free by comparison with other sciences.

Chemistry
(1 experiment)



Physics
(1 experiment)

Computer Science
(1 million experiments)

Bottom line. No excuse for not running experiments to understand costs.

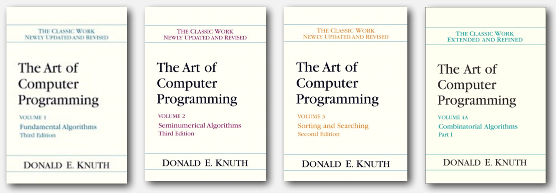
http://algs4.cs.princeton.edu

1.4 Analysis of Algorithms

Mathematical models for running time

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

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



Donald Knuth
1974 Turing Award

In principle, accurate mathematical models are available.

Cost of basic operations

Challenge. How to estimate constants.

operation	example	nanoseconds †
integer add	`a + b`	2.1
integer multiply	`a * b`	2.4
integer divide	`a / b`	5.4
floating-point add	`a + b`	4.6
floating-point multiply	`a * b`	4.2
floating-point divide	`a / b`	13.5
sine	`Math.sin(theta)`	91.3
arctangent	`Math.atan2(y, x)`	129
...	...	...

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

Frequency of basic operations: Example: 1-SUM

Q. How many instructions as a function of input size n ?

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

n array accesses

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

Simplification 1: cost model

Cost model. Use some basic operation as a proxy for running time.

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

Heuristic: pick an operation that's both frequent and costly

Assumption:

Array access dominates running time

This is a hypothesis that can be tested

Memory access is a good candidate: communicating outside CPU is often costly

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

cost model = array accesses

(we assume compiler/JVM do not optimize any array accesses away!)

Simplification 2: tilde notation

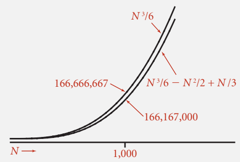
- Estimate running time (or memory) as a function of input size n.
- Ignore lower order terms.
  - when n is large, terms are negligible
  - when n is small, we don't care

Ex 1. $\frac{1}{6} n^3 + 20 n + 16 \sim \frac{1}{6} n^3$

Ex 2. $\frac{1}{6} n^3 + 100 n^{4/3} + 56 \sim \frac{1}{6} n^3$

Ex 3. $\frac{1}{6} n^3 - \frac{1}{2} n^2 + \frac{1}{3} n \sim \frac{1}{6} n^3$

discard lower-order terms
(e.g., n = 1000: 166.67 million vs. 166.17 million)



Leading-term approximation

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

Example: 2-SUM

Q. Approximately how many array accesses as a function of input size n?

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

"inner loop"

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

A. $\sim n^2$ array accesses.

Bottom line. Use cost model and tilde notation to simplify counts.

Example: 3-SUM

Q. Approximately how many array accesses as a function of input size n?

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

"inner loop"

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

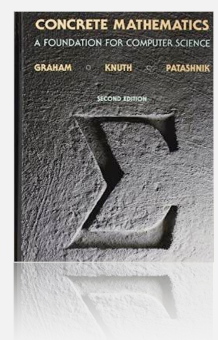
A. $\sim \frac{1}{2} n^3$ array accesses.

Bottom line. Use cost model and tilde notation to simplify counts.

Estimating a discrete sum

Q. How to estimate a discrete sum?

A1. Take a discrete mathematics course (COS 340).



Estimating a discrete sum

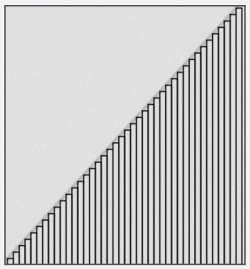
Q. How to estimate a discrete sum?

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

Ex. $1 + 2 + \ldots + n$.

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

N



N

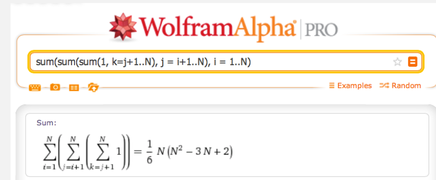
Visual proof:

Area occupied by the sum
$\approx$
Half the area of the square

Estimating a discrete sum

Q. How to estimate a discrete sum?

A3. Use Maple or Wolfram Alpha.



wolframalpha.com

```
[wayne:nobel.princeton.edu] > maple15
    |\^/|     Maple 15 (X86 64 LINUX)
._|\|   |/|_. Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2011
 \  MAPLE  /  All rights reserved. Maple is a trademark of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> factor(sum(sum(sum(1, k=j+1..n), j = i+1..n), i = 1..n));

                              n (n - 1) (n - 2)
                              -----------------
                                      6
```


Analysis of algorithms quiz 2

How many array accesses does the following code fragment make as a function of n?

```
int count = 0;
for (int i = 0; i < n; i++)
   for (int j = i+1; j < n; j++)
      for (int k = 1; k < n; k = k*2)
         if (a[i] + a[j] >= a[k])
            count++;
```

A. $\sim n^2 \lg n$

B. $\sim 3/2 \ n^2 \lg n$

C. $\sim 1/2 \ n^3$

D. $\sim 3/2 \ n^3$

E. *I don't know.*

$k = 1, 2, 4, \ldots$ — $\lg n$ times

http://algs4.cs.princeton.edu

1.4 Analysis of Algorithms

- introduction
- observations
- mathematical models
- *order-of-growth classifications*
- memory

Common order-of-growth classifications

Definition. If $f(n) \sim c\, g(n)$ for some constant $c > 0$, then the order of growth of $f(n)$ is $g(n)$.

- Ignores leading coefficient.
- Ignores lower-order terms.

Ex. The order of growth of the running time of this code is n^3.

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

Typical usage. Mathematical analysis of running times.

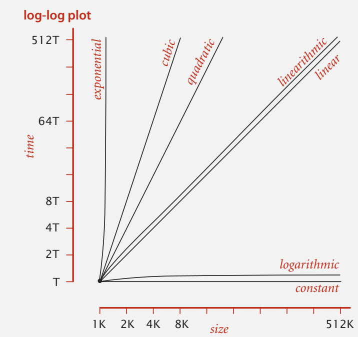
where leading coefficient depends on machine, compiler, JVM, ...

Common order-of-growth classifications

Good news. The set of functions

$1,\ \log n,\ n,\ n \log n,\ n^2,\ n^3$, and 2^n

suffices to describe the order of growth of most common algorithms.



Typical orders of growth

Common order-of-growth classifications

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

Binary search

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

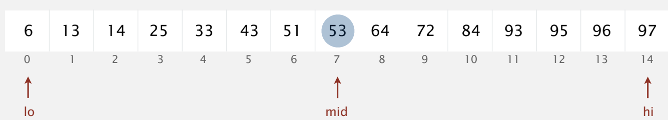
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

successful search for 33



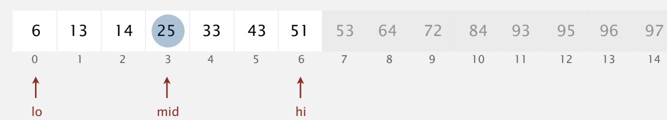
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

successful search for 33



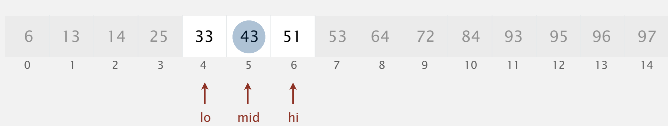
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

successful search for 33



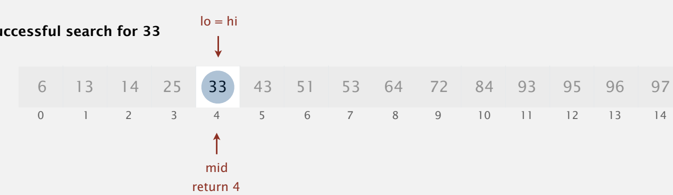
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

successful search for 33



Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

unsuccessful search for 34

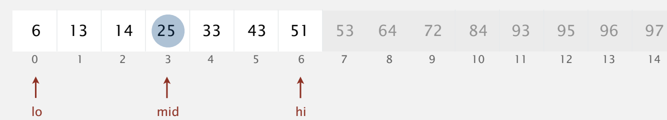
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

unsuccessful search for 34



Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

unsuccessful search for 34

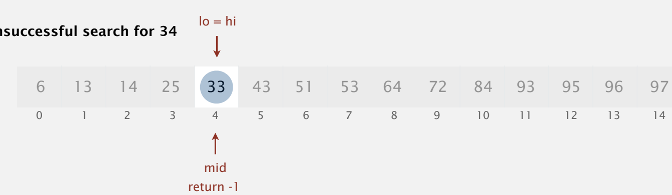
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

unsuccessful search for 34



Binary search: Java implementation

Invariant. If key appears in array a[], then a[lo] ≤ key ≤ a[hi].

Cost model. key comparisons. [Why?]

```
public static int binarySearch(int[] a, int key)
{
   int lo = 0, hi = a.length - 1;
   while (lo <= hi)
   {
      int mid = lo + (hi - lo) / 2;
      if      (key < a[mid]) hi = mid - 1;
      else if (key > a[mid]) lo = mid + 1;
      else return mid;
   }
   return -1;
}
```

why not mid = (lo + hi) / 2 ?

one "3-way compare"

Binary search: mathematical analysis

Proposition. Binary search uses at most $1 + \lg n$ key compares to search in a sorted array of size n.

Def. $T(n)$ = # key compares to binary search a sorted subarray of size ≤ n.

Binary search recurrence. $T(n) \le T(n/2) + 1$ for $n > 1$, with $T(1) = 1$.

left or right half (floored division)

possible to implement with one 2-way compare (instead of 3-way)

Pf sketch. [assume n is a power of 2]

$T(n)$	≤	$T(n/2) + 1$	[given]
	≤	$T(n/4) + 1 + 1$	[apply recurrence to first term]
	≤	$T(n/8) + 1 + 1 + 1$	[apply recurrence to first term]
	⋮		
	≤	$T(n/n) + 1 + 1 + \ldots + 1$	[stop applying, $T(1) = 1$]
	=	$1 + \lg n$	

($\lg n$ ones)

The 3-sum problem: an $n^2 \log n$ algorithm

Algorithm.
- Step 1: Sort the n (distinct) numbers.
- Step 2: For each pair of numbers a[i] and a[j], binary search for -(a[i] + a[j]).

Analysis. Order of growth is $n^2 \log n$.
- Step 1: n^2 with insertion sort (or $n \log n$ with mergesort).
- Step 2: $n^2 \log n$ with binary search.

input

30 -40 -20 -10 40 0 10 5

sort

-40 -20 -10 0 5 10 30 40

binary search

```
(-40, -20)   60
(-40, -10)   50
(-40,   0)   40
(-40,   5)   35
(-40,  10)   30
    ⋮         ⋮
(-20, -10)   30
    ⋮         ⋮
(-10,   0)   10
    ⋮         ⋮
( 10,  30)  -40
( 10,  40)  -50
( 30,  40)  -70
```

only count if a[i] < a[j] < a[k] to avoid double counting

Comparing programs

Hypothesis. The sorting-based $n^2 \log n$ algorithm for 3-SUM is significantly faster in practice than the brute-force n^3 algorithm.

n	time (seconds)
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1

ThreeSum.java

n	time (seconds)
1,000	0.14
2,000	0.18
4,000	0.34
8,000	0.96
16,000	3.67
32,000	14.88
64,000	59.16

ThreeSumDeluxe.java

Guiding principle. Typically, better order of growth ⇒ faster in practice.

Theorist vs. pragmatist view of algorithmic efficiency

$a\, n^{b}$ ← property of algorithm

↑ system-dependent

Theorist: Worrying about constant factors is tedious and crass!
The asymptotic efficiency of an algorithm is a mathematical fact.
I study properties of the universe. The computer is irrelevant!

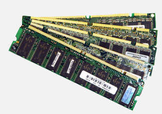
Pragmatist: I will use mathematical model to compute b, then verify empirically.
I will use a combination of math and observation to estimate a.

When I need to pick between algorithms,
models provide a strong clue to practical performance.

1.4 Analysis of Algorithms

Basics

Bit. 0 or 1.
Byte. 8 bits.
Megabyte (MB). 2^{20} bytes (about 1 million).
Gigabyte (GB). 2^{30} bytes (about 1 billion).



64-bit machine. We assume a 64-bit machine with 8-byte pointers.

↑ some JVMs "compress" ordinary object pointers to 4 bytes to avoid this cost

Typical memory usage for primitive types and arrays

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

primitive types

type	bytes
char[]	$2n + 24$
int[]	$4n + 24$
double[]	$8n + 24$

one-dimensional arrays

type	bytes
char[][]	$\sim 2\, m\, n$
int[][]	$\sim 4\, m\, n$
double[][]	$\sim 8\, m\, n$

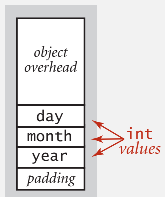
two-dimensional arrays

Typical memory usage for objects in Java

Object overhead. 16 bytes.
Reference. 8 bytes.
Padding. Each object uses a multiple of 8 bytes.

Ex 1. A `Date` object uses 32 bytes of memory.

```
public class Date
{
   private int day;
   private int month;
   private int year;
...
}
```



16 bytes (object overhead)
4 bytes (int)
4 bytes (int)
4 bytes (int)
4 bytes (padding)
32 bytes

Typical memory usage summary

Total memory usage for a data type value:

- Primitive type: 4 bytes for `int`, 8 bytes for `double`, ...
- Object reference: 8 bytes.
- Array: 24 bytes + memory for each array entry.
- Object: 16 bytes + memory for each instance variable.
- Padding: round up to multiple of 8 bytes.

+ 8 extra bytes per inner class object (for reference to enclosing class)

Note. Depending on application, we may want to count memory for any referenced objects (recursively).

Analysis of algorithms quiz 3

How much memory does a `WeightedQuickUnionUF` use as a function of n ?

A. $\sim 4\,n$ *bytes*
B. $\sim 8\,n$ *bytes*
C. $\sim 4\,n^2$ *bytes*
D. $\sim 8\,n^2$ *bytes*
E. *I don't know.*

```
public class WeightedQuickUnionUF
{
   private int[] parent;
   private int[] size;
   private int count;

   public WeightedQuickUnionUF(int n)
   {
      parent = new int[n];
      size = new int[n];
      count = 0;
      for (int i = 0; i < n; i++)
          parent[i] = i;
      for (int i = 0; i < n; i++)
          size[i] = 1;
   }
   ...
}
```


Analysis of algorithms quiz 3

How much memory does a `WeightedQuickUnionUF` use as a function of n ?

16 bytes (object overhead)
8 + (4n + 24) bytes each (reference + `int[]` array)
4 bytes (`int`)
4 bytes (padding)
8n + 88 ~ 8n bytes

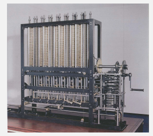

```
public class WeightedQuickUnionUF
{
   private int[] parent;
   private int[] size;
   private int count;

   public WeightedQuickUnionUF(int n)
   {
      parent = new int[n];
      size = new int[n];
      count = 0;
      for (int i = 0; i < n; i++)
          parent[i] = i;
      for (int i = 0; i < n; i++)
          size[i] = 1;
   }
   ...
}
```

Turning the crank: summary

Empirical analysis.

- Execute program to perform experiments.
- Assume power law.
- Formulate a hypothesis for running time.
- Model enables us to make predictions.



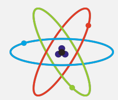
Mathematical analysis.

- Analyze algorithm to count frequency of operations.
- Use tilde notation to simplify analysis.
- Model enables us to explain behavior.

$$\sum_{h=0}^{\lfloor \lg N \rfloor} \lceil N/2^{h+1} \rceil \, h \sim N$$

Scientific method.

- Mathematical model is independent of a particular system; applies to machines not yet built.
- Empirical analysis is necessary to validate mathematical models and to make predictions.



Announcement

Unlike COS 126, you get only 10 checks in Dropbox per assignment

More announcements (re. exercises, etc.): see Piazza