BBM 202 - ALGORITHMS

HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

ANALYSIS OF ALGORITHMS

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick
and K. Wayne of Princeton University.

TODAY

- **Analysis of Algorithms**
- Observations
- Mathematical models
- Order-of-growth classifications
- Dependencies on inputs
- Memory

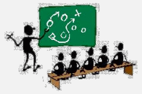
Cast of characters

Programmer needs to develop a working solution.

Student might play any or all of these roles someday.

Client wants to solve problem efficiently.

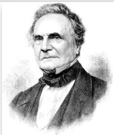
Theoretician wants to understand.

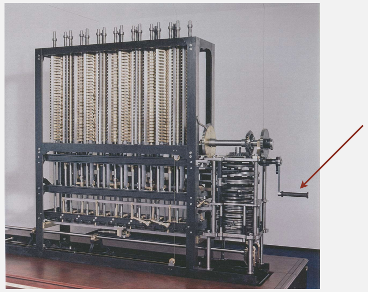


Basic blocking and tackling is sometimes necessary. [this lecture]

Running time

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

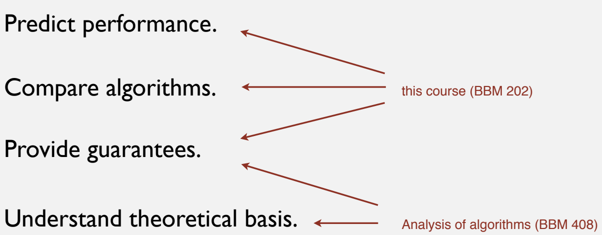




how many times do you have to turn the crank?

Analytic Engine

Reasons to analyse algorithms



Predict performance.

Compare algorithms.

Provide guarantees.

Understand theoretical basis.

this course (BBM 202)

Analysis of algorithms (BBM 408)

Primary practical reason: avoid performance bugs.

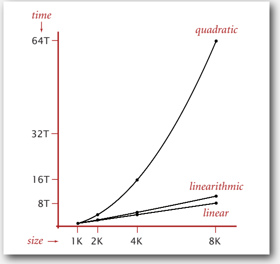
client gets poor performance because programmer did not understand performance characteristics

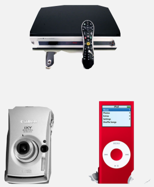
Some algorithmic successes

Discrete Fourier transform.

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

Friedrich Gauss
1805





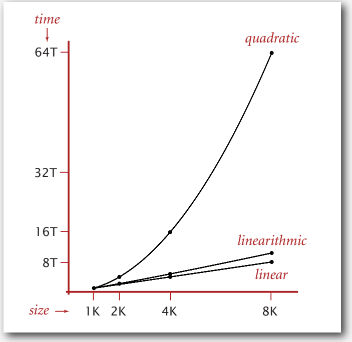
- sFFT: Sparse Fast Fourier Transform algorithm (Hassanieh et al., 2012)
  - A faster Fourier Transform: $k \log N$ steps (with k sparse coefficients)

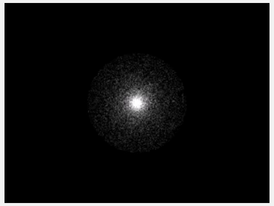
Some algorithmic successes

N-body simulation.

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

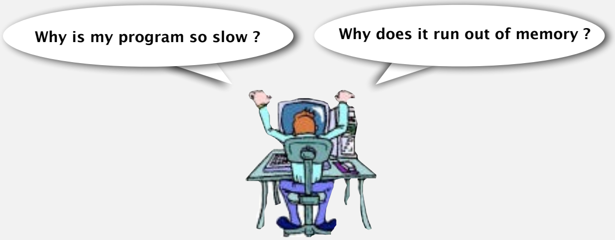
Andrew Appel
PU '81





The challenge

Q. Will my program be able to solve a large practical input?



Key insight. [Knuth 1970s] Use scientific method to understand performance.

Scientific method applied to analysis of algorithms

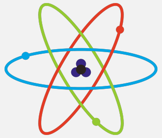
A framework for predicting performance and comparing algorithms.

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.



Feature of the natural world = computer itself.

ANALYSIS OF ALGORITHMS

- **Observations**
- Mathematical models
- Order-of-growth classifications
- Dependencies on inputs
- Memory

Example: 3-sum

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

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

% java ThreeSum 8ints.txt
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 class ThreeSum
{
   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;
   }

   public static void main(String[] args)
   {
      int[] a = In.readInts(args[0]);
      StdOut.println(count(a));
   }
}
```

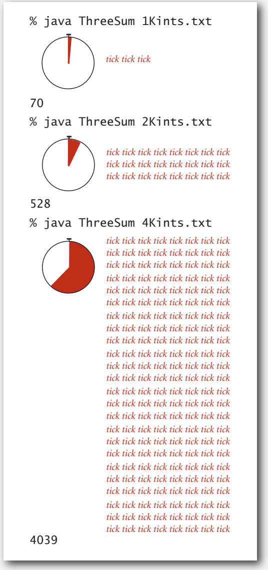
check each triple

for simplicity, ignore integer overflow

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)
{
   int[] a = In.readInts(args[0]);
   Stopwatch stopwatch = new Stopwatch();
   StdOut.println(ThreeSum.count(a));
   double time = stopwatch.elapsedTime();
}
```

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

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

implementation (part of `stdlib.jar`)

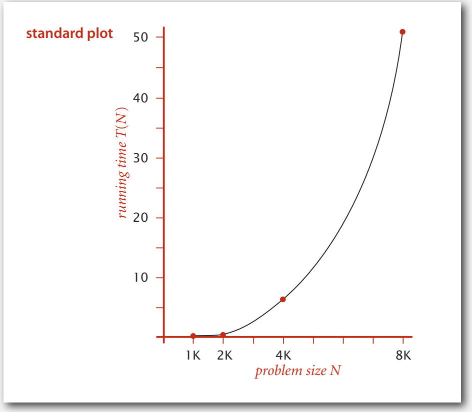
Empirical analysis

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	?

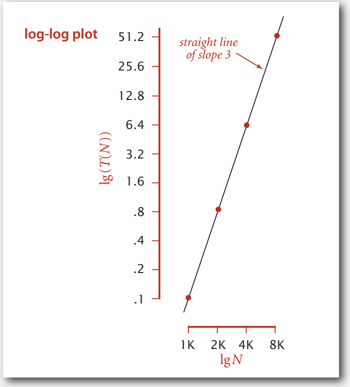
Data analysis

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.



$$\lg(T(N)) = b \lg N + c$$
$$b = 2.999$$
$$c = -33.2103$$

$$T(N) = a N^b, \text{ where } a = 2^c$$

power law

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

slope

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

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

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 hypothesis.

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 linear regression

Experimental algorithmics

System independent effects.

- Algorithm.
- Input data.

determines exponent b
in power law

System dependent effects.

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

determines constant a
in power law

Bad news. Difficult to get precise measurements.

Good news. Much easier and cheaper than other sciences.

e.g., can run huge number of experiments

In practice, constant factors matter too!

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

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

N	time
1.000	0,11
2.000	0,35
4.000	1,6
8.000	6,5

Jenny ~ $c_1 N^2$ seconds

N	time
250	0,5
500	1,1
1.000	1,9
2.000	3,9

Kenny ~ $c_2 N$ seconds

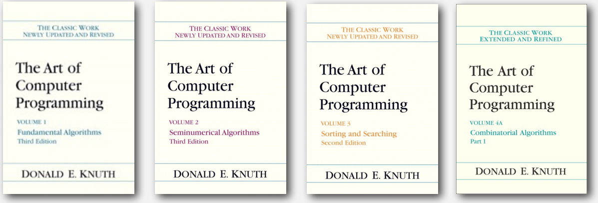
ANALYSIS OF ALGORITHMS

- Observations
- **Mathematical models**
- Order-of-growth classifications
- Dependencies on inputs
- Memory

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

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

Cost of basic operations

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

Novice mistake. Abusive string concatenation.

Example: 1-sum

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

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

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

Example: 2-sum

Q. How many instructions 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++;
```

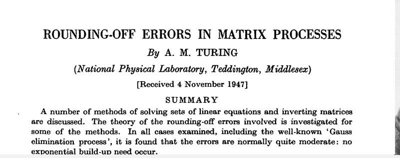
$$0 + 1 + 2 + \ldots + (N-1) = \frac{1}{2} N (N-1) = \binom{N}{2}$$

operation	frequency
variable declaration	N + 2
assignment statement	N + 2
less than compare	½ (N + 1) (N + 2)
equal to compare	½ N (N – 1)
array access	N (N – 1)
increment	½ N (N – 1) to N (N – 1)

tedious to count exactly

Simplifying the calculations

> *" It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process and then given them various weights. We might, for instance, count the number of additions, subtractions, multiplications, divisions, recording of numbers, and extractions of figures from tables. In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and we shall therefore only attempt to count the number of multiplications and recordings. "* — *Alan Turing*



ROUNDING-OFF ERRORS IN MATRIX PROCESSES

By A. M. TURING

(*National Physical Laboratory, Teddington, Middlesex*)

[Received 4 November 1947]

SUMMARY

A number of methods of solving sets of linear equations and inverting matrices are discussed. The theory of the rounding-off errors involved is investigated for some of the methods. In all cases examined, including the well-known 'Gauss elimination process', it is found that the errors are normally quite moderate: no exponential build-up need occur.

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

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

operation	frequency
variable declaration	N + 2
assignment statement	N + 2
less than compare	½ (N + 1) (N + 2)
equal to compare	½ N (N – 1)
array access	N (N – 1)
increment	½ N (N – 1) to N (N – 1)

cost model = array accesses

Simplification 2: tilde notation

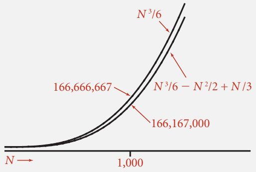
- Estimate running time (or memory) as a function of input size N.
- Ignore lower order terms.
  - when N is large, terms are negligible
  - when N is small, we don't care

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

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

Ex 3. $\frac{1}{6} N^3 \underbrace{- \frac{1}{2} N^2 + \frac{1}{3} N}_{\text{discard lower-order terms}} \sim \frac{1}{6} N^3$

(e.g., N = 1000: 500 thousand vs. 166 million)



Leading-term approximation

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

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

operation	frequency	tilde notation
variable declaration	N + 2	~ N
assignment statement	N + 2	~ N
less than compare	½ (N + 1) (N + 2)	~ ½ N²
equal to compare	½ N (N – 1)	~ ½ N²
array access	N (N – 1)	~ N²
increment	½ N (N – 1) to N (N – 1)	~ ½ N² to ~ N²

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"

A. ~ N^2 array accesses.

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

Bottom line. Use cost model and tilde notation to simplify frequency 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"

A. ~ ½ N^3 array accesses.

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

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

Estimating a discrete sum

Q. How to estimate a discrete sum?

A1. Take discrete mathematics course.

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

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

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

Ex 2. $1 + 1/2 + 1/3 + \ldots + 1/N.$

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

Ex 3. 3-sum triple loop.

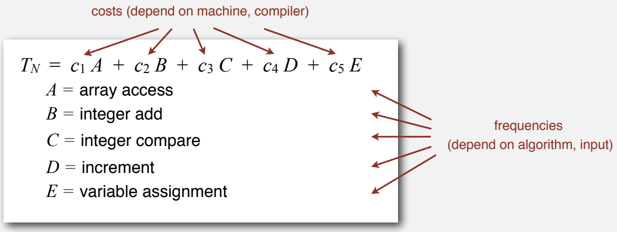
$$\sum_{i=1}^{N} \sum_{j=i}^{N} \sum_{k=j}^{N} 1 \sim \int_{x=1}^{N} \int_{y=x}^{N} \int_{z=y}^{N} dz\, dy\, dx \sim \frac{1}{6} N^3$$

Mathematical models for running time

In principle, accurate mathematical models are available.

In practice,

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



costs (depend on machine, compiler)

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

A = array access
B = integer add
C = integer compare
D = increment
E = variable assignment

frequencies (depend on algorithm, input)

Bottom line. We use approximate models in this course: $T(N) \sim c\,N^3$.

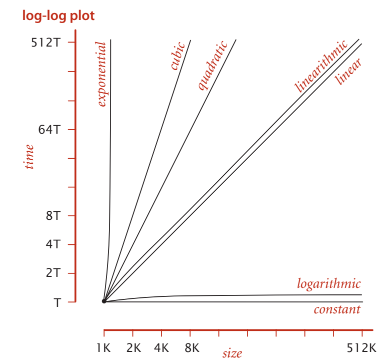
ANALYSIS OF ALGORITHMS

- Observations
- Mathematical models
- Order-of-growth classifications
- Dependencies on inputs
- Memory

Common order-of-growth classifications

Good news. the small set of functions
$1,\ \log N,\ N,\ N \log N,\ N^2,\ N^3$, and 2^N
suffices to describe order-of-growth of typical algorithms.

order of growth discards leading coefficient



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++)` `{   ...         }`	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	T(N)

Practical implications of order-of-growth

growth rate	problem size solvable in minutes			
	1970s	1980s	1990s	2000s
1	any	any	any	any
log N	any	any	any	any
N	millions	tens of millions	hundreds of millions	billions
N log N	hundreds of thousands	millions	millions	hundreds of millions
N^2	hundreds	thousand	thousands	tens of thousands
N^3	hundred	hundreds	thousand	thousands
2^N	20	20s	20s	30

Practical implications of order-of-growth

growth rate	problem size solvable in minutes				time to process millions of inputs			
	1970s	1980s	1990s	2000s	1970s	1980s	1990s	2000s
1	any	any	any	any	instant	instant	instant	instant
log N	any	any	any	any	instant	instant	instant	instant
N	millions	tens of millions	hundreds of millions	billions	minutes	seconds	second	instant
N log N	hundreds of thousands	millions	millions	hundreds of millions	hour	minutes	tens of seconds	seconds
N^2	hundreds	thousand	thousands	tens of thousands	decades	years	months	weeks
N^3	hundred	hundreds	thousand	thousands	never	never	never	millennia

---

Practical implications of order-of-growth

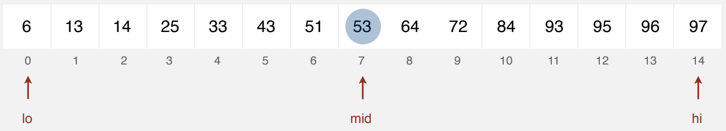
growth rate	name	description	effect on a program that runs for a few seconds	
			time for 100x more data	size for 100x faster computer
1	constant	independent of input size	–	–
log N	logarithmic	nearly independent of input size	–	–
N	linear	optimal for N inputs	a few minutes	100x
N log N	linearithmic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

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.

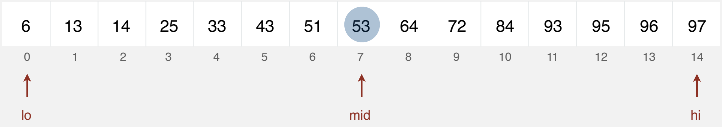


6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
lo							mid							hi

Binary search demo

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

Successful search. Binary search for 33.

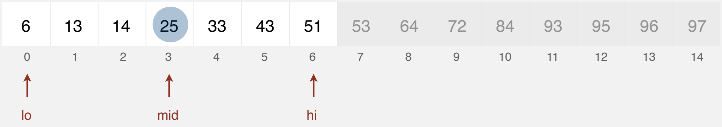


6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
lo							mid							hi

Binary search demo

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

Successful search. Binary search for 33.

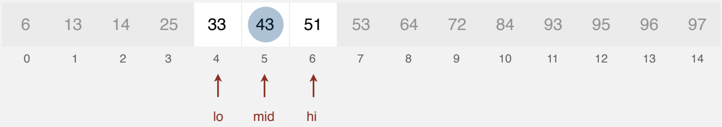


6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
lo			mid			hi								

Binary search demo

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

Successful search. Binary search for 33.

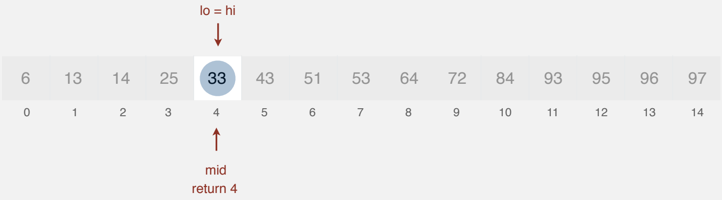


6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				lo	mid	hi								

Binary search demo

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

Successful search. Binary search for 33.

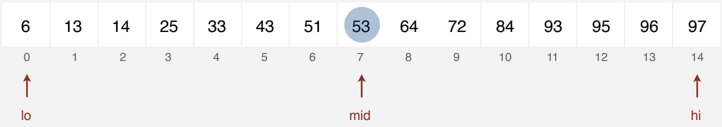


				lo = hi										
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				mid return 4										

Binary search demo

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

Unsuccessful search. Binary search for 34.

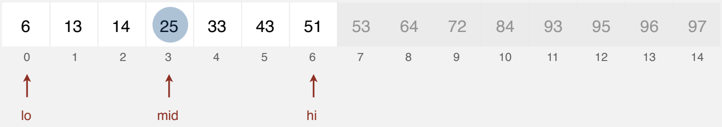


6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Binary search demo

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

Unsuccessful search. Binary search for 34.

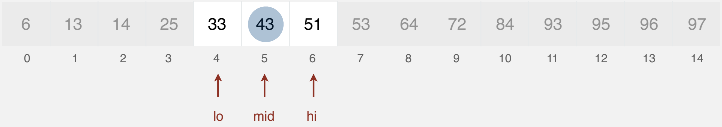


6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Binary search demo

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

Unsuccessful search. Binary search for 34.

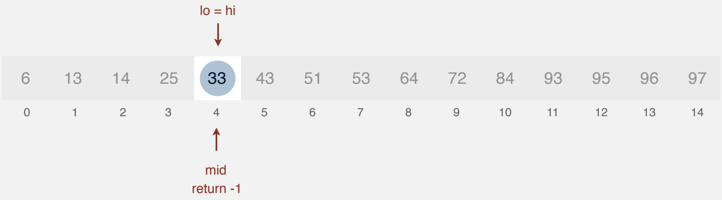


6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Binary search demo

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

Unsuccessful search. Binary search for 34.



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: Java implementation

Trivial to implement?

- First binary search published in 1946; first bug-free one published in 1962.
- Bug in Java's `Arrays.binarySearch()` discovered in 2006.

```
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;
}
```

one "3-way compare"

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

Binary search: mathematical analysis

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

Def. $T(N) \equiv$ # compares to binary search in a sorted subarray of size at most N.

Binary search recurrence. $T(N) \le \underset{\uparrow}{T(N/2)} + \underset{\uparrow}{1}$ for $N > 1$, with $T(1) = 1$.

left or right half

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

Pf sketch.

$$\begin{aligned} T(N) &\le T(N/2) + 1 && \text{given} \\ &\le T(N/4) + 1 + 1 && \text{apply recurrence to first term} \\ &\le T(N/8) + 1 + 1 + 1 && \text{apply recurrence to first term} \\ &\;\;\cdots \\ &\le T(N/N) + 1 + 1 + \ldots + 1 \\ &= 1 + \lg N && \text{stop applying, T(1) = 1} \end{aligned}$$

Binary search: mathematical analysis

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

Def. $T(N) \equiv$ # compares to binary search in a sorted subarray of size at most N.

Binary search recurrence. $T(N) \le T(\lfloor N/2 \rfloor) + 1$ for $N > 1$, with $T(0) = 0$.

For simplicity, we prove when $N = 2^n - 1$ for some n, so $\lfloor N/2 \rfloor = 2^{n-1} - 1$.

$$\begin{aligned} T(2^n - 1) &\le T(2^{n-1} - 1) + 1 && \text{given} \\ &\le T(2^{n-2} - 1) + 1 + 1 && \text{apply recurrence to first term} \\ &\le T(2^{n-3} - 1) + 1 + 1 + 1 && \text{apply recurrence to first term} \\ &\;\;\cdots \\ &\le T(2^0 - 1) + 1 + 1 + \ldots + 1 \\ &= n && \text{stop applying, T(0) = 1} \end{aligned}$$

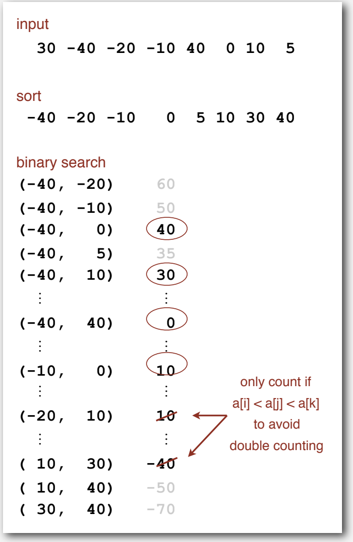
An N² log N algorithm for 3-sum

Algorithm.

- Sort the N (distinct) numbers.
- 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.
- 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
⋮ ⋮
(-40, 40) 0
⋮ ⋮
(-10, 0) 10
⋮ ⋮
(-20, 10) ~~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 $N^2 \log N$ three-sum algorithm 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.

ANALYSIS OF ALGORITHMS

- Observations
- Mathematical models
- Order-of-growth classifications
- **Dependencies on inputs**
- Memory

Types of analyses

Best case. Lower bound on cost.

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

Worst case. Upper bound on cost.

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

Average case. Expected cost for random input.

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

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

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

Ex 2. Compares for binary search.

Best:	~ 1
Average:	$\sim \lg N$
Worst:	$\sim \lg N$

Types of analyses

Best case. Lower bound on cost.

Worst case. Upper bound on cost.

Average case. "Expected" cost.

Actual data might not match input model?

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

Theory of Algorithms

Goals.

- Establish "difficulty" of a problem.
- Develop "optimal" algorithms.

Approach.

- Suppress details in analysis: analyze "to within a constant factor".
- Eliminate variability in input model by focusing on the worst case.

Optimal algorithm.

- Performance guarantee (to within a constant factor) for any input.
- No algorithm can provide a better performance guarantee.

Commonly-used notations

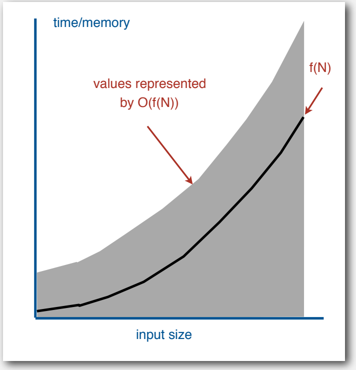
notation	provides	example	shorthand for	used to
Tilde	leading term	$\sim 10\ N^2$	$10\ N^2$ $10\ N^2 + 22\ N \log N$ $10\ N^2 + 2\ N + 37$	provide approximate model
Big Theta	asymptotic growth rate	$\Theta(N^2)$	$\frac{1}{2}\ N^2$ $10\ N^2$ $5\ N^2 + 22\ N \log N + 3N$	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10\ N^2$ $100\ N$ $22\ N \log N + 3\ N$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2}\ N^2$ N^5 $N^3 + 22\ N \log N + 3\ N$	develop lower bounds

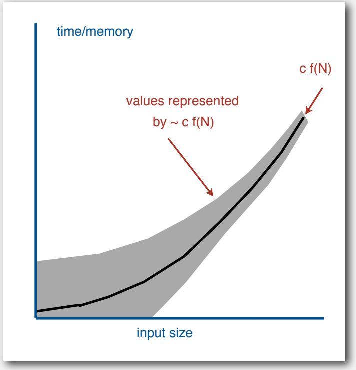
Common mistake. Interpreting big-Oh as an approximate model.

Tilde notation vs. big-Oh notation

We use tilde notation whenever possible.

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





Theory of algorithms: example 1

Goals.

- Establish "difficulty" of a problem and develop "optimal" algorithms.
- Ex. 1-SUM = "Is there a 0 in the array? "

Upper bound. A specific algorithm.

- Ex. Brute-force algorithm for 1-SUM: Look at every array entry.
- Running time of the optimal algorithm for 1-SUM is O(N).

Lower bound. Proof that no algorithm can do better.

- Ex. Have to examine all N entries (any unexamined one might be 0).
- Running time of the optimal algorithm for 1-SUM is Ω(N).

Optimal algorithm.

- Lower bound equals upper bound (to within a constant factor).
- Ex. Brute-force algorithm for 1-SUM is optimal: its running time is Θ(N).

Theory of algorithms: example 2

Goals.

- Establish "difficulty" of a problem and develop "optimal" algorithms.
- Ex. 3-SUM

Upper bound. A specific algorithm.

- Ex. Brute-force algorithm for 3-SUM
- Running time of the optimal algorithm for 3-SUM is $O(N^3)$.

Theory of algorithms: example 2

Goals.

- Establish "difficulty" of a problem and develop "optimal" algorithms.
- Ex. 3-SUM

Upper bound. A specific algorithm.

- Ex. Improved algorithm for 3-SUM
- Running time of the optimal algorithm for 3-SUM is $O(N^2 \log N)$.

Lower bound. Proof that no algorithm can do better.

- Ex. Have to examine all N entries to solve 3-SUM.
- Running time of the optimal algorithm for solving 3-SUM is $\Omega(N)$.

Open problems.

- Optimal algorithm for 3-SUM?
- Subquadratic algorithm for 3-SUM?
- Quadratic lower bound for 3-SUM?

Algorithm design approach

Start.

- Develop an algorithm.
- Prove a lower bound.

Gap?

- Lower the upper bound (discover a new algorithm).
- Raise the lower bound (more difficult).

Golden Age of Algorithm Design.

- 1970s-.
- Steadily decreasing upper bounds for many important problems.
- Many known optimal algorithms.

Caveats.

- Overly pessimistic to focus on worst case?
- Need better than "to within a constant factor" to predict performance.

ANALYSIS OF ALGORITHMS

- Observations
- Mathematical models
- Order-of-growth classifications
- Dependencies on inputs
- **Memory**

Basics

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

NIST ↓ most computer scientists ↓

Old machine. We used to assume a 32-bit machine with 4 byte pointers.

Modern machine. We now assume a 64-bit machine with 8 byte pointers.
- Can address more memory.
- Pointers use more space.

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

Typical memory usage for primitive types and arrays

Primitive types.

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

for primitive types

Array overhead. 24 bytes.

type	bytes
`char[]`	2N + 24
`int[]`	4N + 24
`double[]`	8N + 24

for one-dimensional arrays

type	bytes
`char[][]`	~ 2 M N
`int[][]`	~ 4 M N
`double[][]`	~ 8 M N

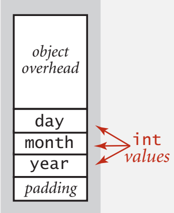
for two-dimensional arrays

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)

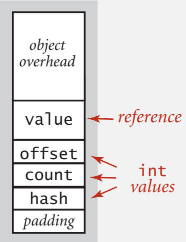
32 bytes

Typical memory usage for objects in Java

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

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

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



16 bytes (object overhead)

8 bytes (reference to array)
2N + 24 bytes (char[] array)
4 bytes (int)
4 bytes (int)
4 bytes (int)
4 bytes (padding)

2N + 64 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 + 8 if inner class.

padding: round up to multiple of 8

extra pointer to enclosing class

Shallow memory usage: Don't count referenced objects.

Deep memory usage: If array entry or instance variable is a reference, add memory (recursively) for referenced object.

Memory profiler

Classmexer library. Measure memory usage of a Java object by querying JVM.

http://www.javamex.com/classmexer

```
import com.javamex.classmexer.MemoryUtil;

public class Memory {
   public static void main(String[] args) {
      Date date = new Date(12, 31, 1999);
      StdOut.println(MemoryUtil.memoryUsageOf(date));
      String s = "Hello, World";
      StdOut.println(MemoryUtil.memoryUsageOf(s));
      StdOut.println(MemoryUtil.deepMemoryUsageOf(s));
   }
}
```

shallow

deep

```
% javac -cp .:classmexer.jar Memory.java
% java  -cp .:classmexer.jar -javaagent:classmexer.jar Memory
32
40
88
```

don't count char[]

2N + 64

use `-XX:-UseCompressedOops` on OS X to match our model

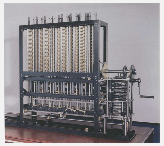
Turning the crank: summary

Empirical analysis.

- Execute program to perform experiments.
- Assume power law and 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 behaviour.



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.