Introduction to Algorithms

Lecture 2: Algorithm Analysis

Yao-Chung Fan
yfan@nchu.edu.tw

Example Problem: 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

3-SUM: brute-force algorithm

Brute-force algorithm. Check each triple.

```
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)
  {
    In in = new In(args[0]);
    int[] a = in.readAllInts();
    StdOut.println(count(a));
```

check each triple

for simplicity, ignore integer overflow

Measuring the running time

Q. How to time a program?

A. Automatic.

public class Stopwatch		
	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);
}
```

Measuring the running time

Q. How to time a program?

A. Automatic.

public class Stopwatch		
	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;
  }
}
```

Empirical analysis

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

```
%
```

The challenge: Understand the Performance of Your Algorithm

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

Why is my program so slow ?

What happen when the input is scale to 100x?

Experimental algorithmics

System independent effects.

- Algorithm.
- Input data.

System dependent effects.

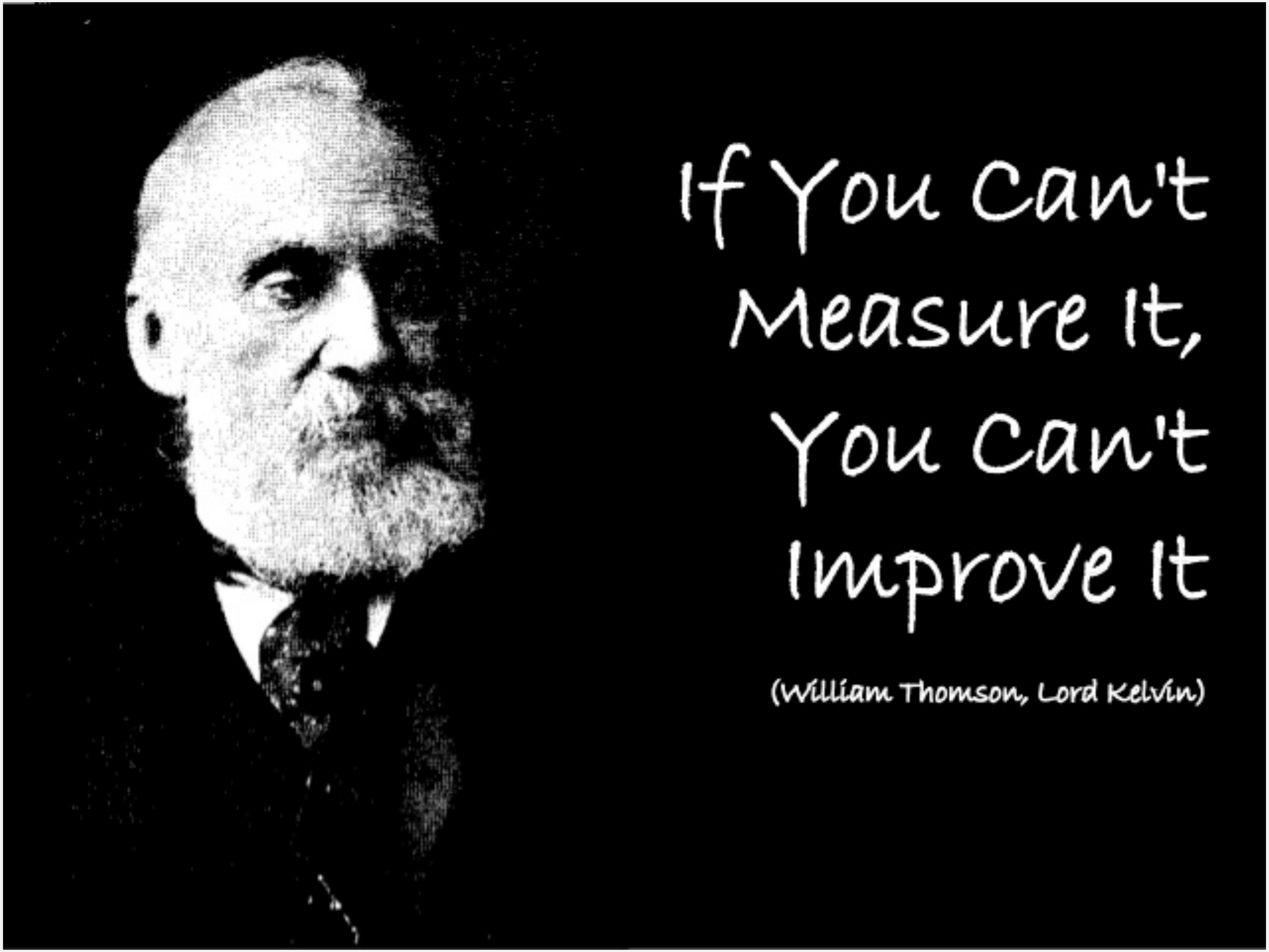
- Hardware: CPU, memory, cache, …
- Software: compiler, interpreter, garbage collector, …
- System: operating system, network, other apps, …

Bad news. Difficult to get precise measurements.

Good news. Much easier and cheaper than other sciences.

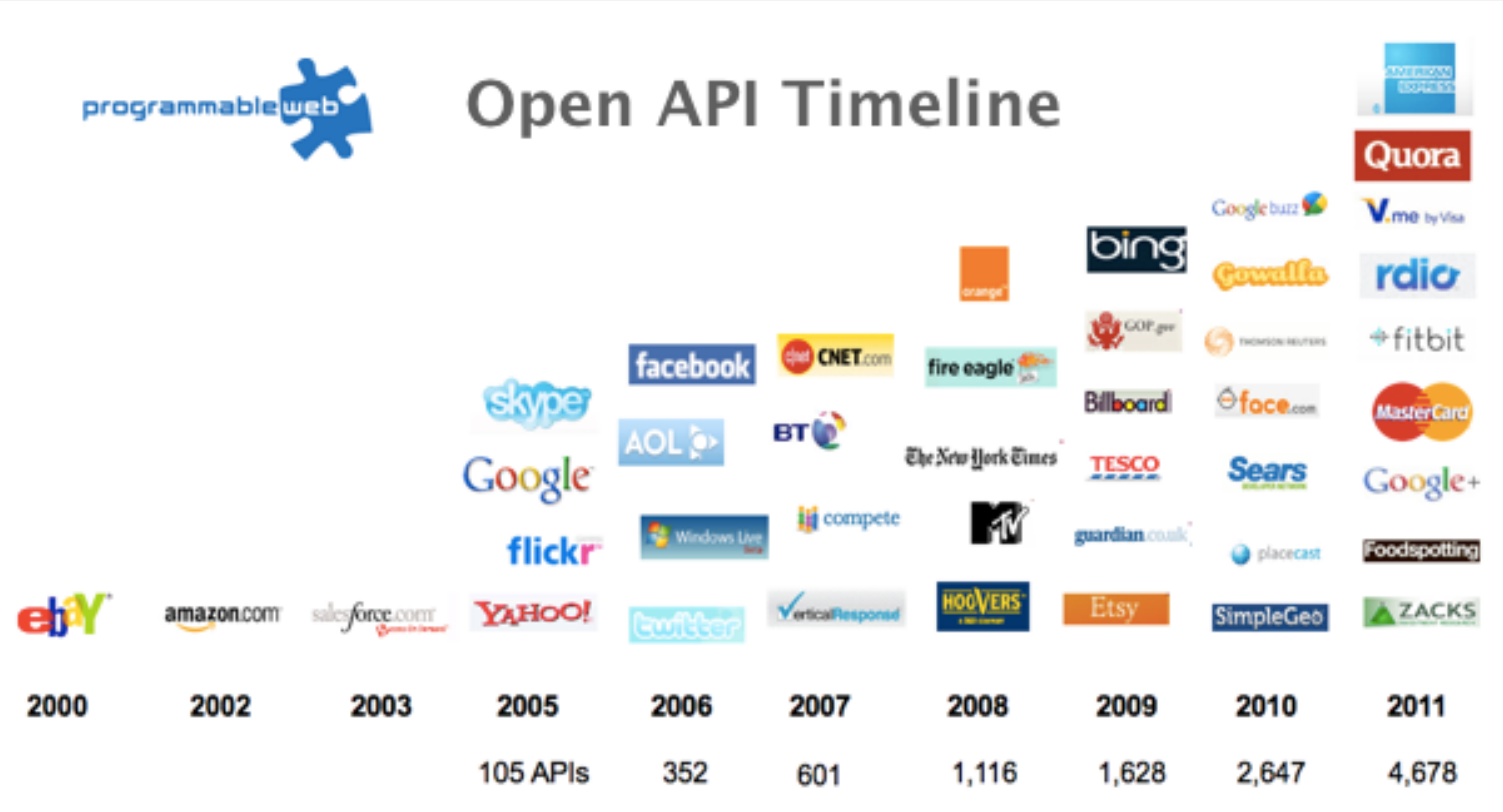
e.g., can run huge number of experiments

About Algorithm Analysis




Note to self: Always measure twice.
GOD DAMN IT.
© channelATE.com

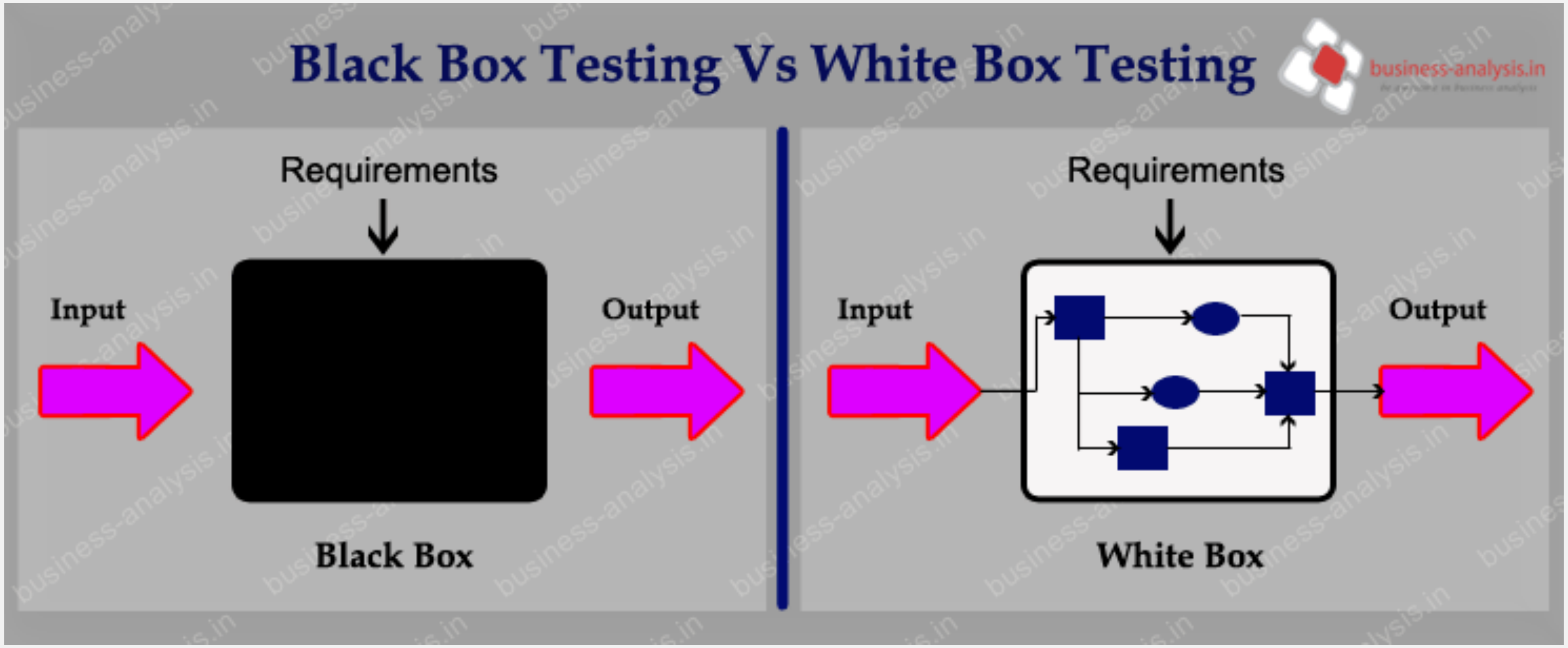
It's API world now



About Software Performance Understanding

黑箱測試 (Black Box Testing)

白箱測試 (White Box Testing)



Analysis of Algorithms

- Introduction
- doubling hypothesis
- mathematical models
- order-of-growth classifications
- theory of algorithms
- memory

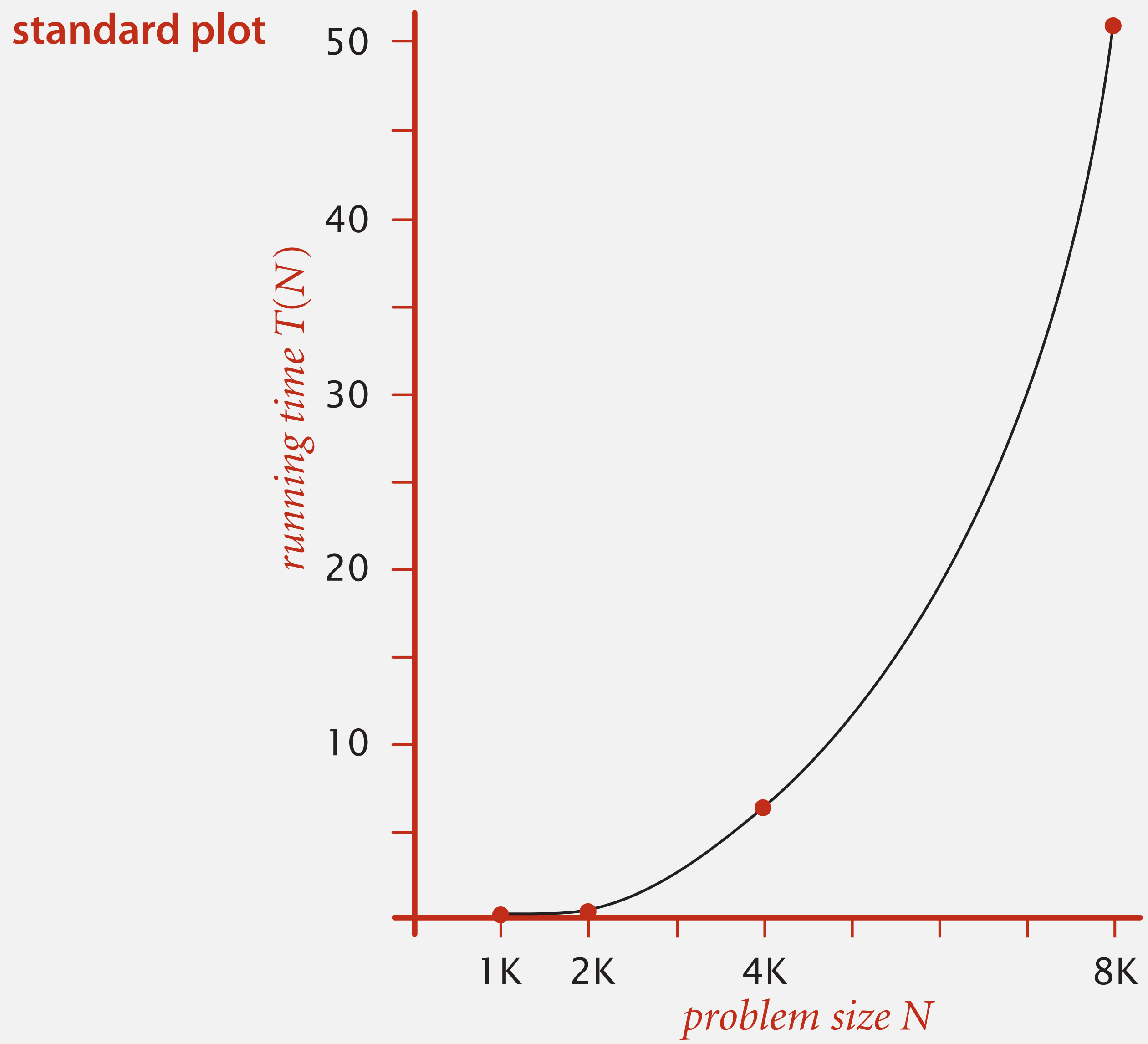
Empirical analysis

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.



$$a N^{b}$$

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

lg (6.4 / 0.8) = 3.0

seems to converge to a constant b ≈ 3

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

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.

Prediction and validation

Hypothesis. The running time is about $0.998 \times 10^{-10} \times N^3$ 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!

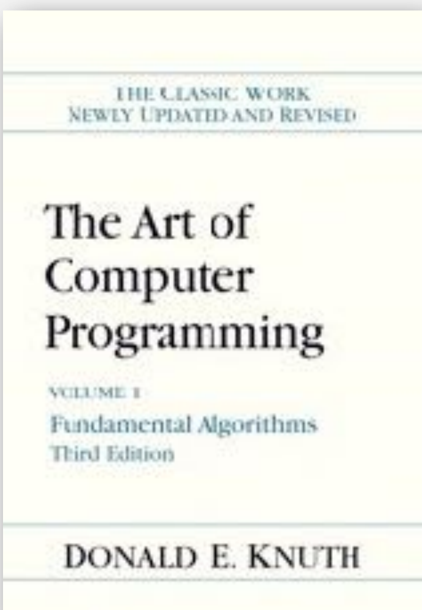
Analysis of Algorithms

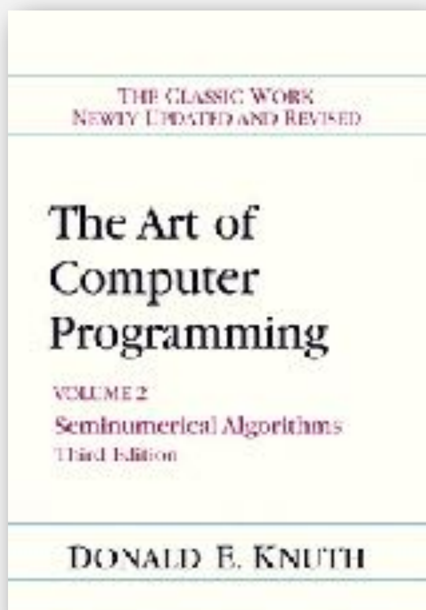
- introduction
- doubling hypothesis
- **mathematical models**
- order-of-growth classifications
- theory of algorithms
- memory

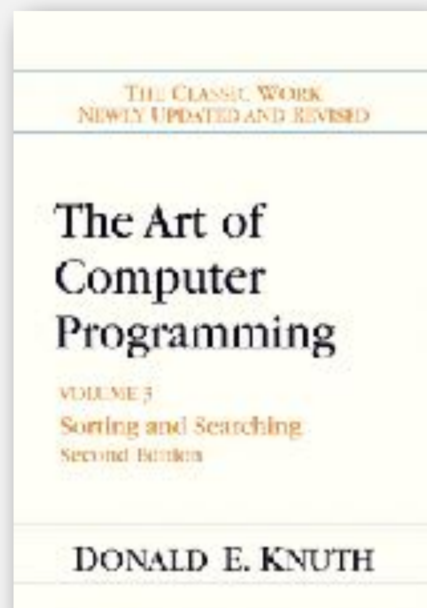
Mathematical models for running time

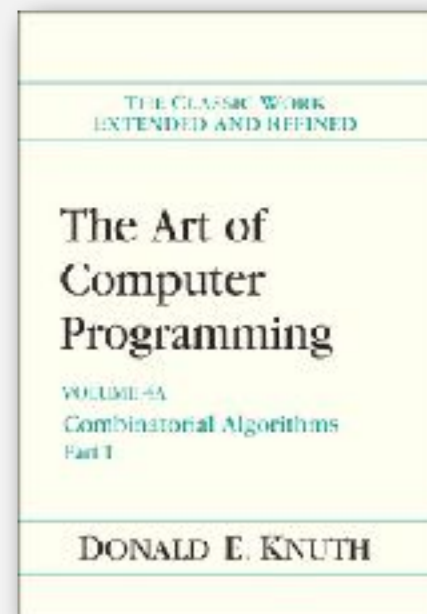
Total running time: sum of cost × frequency for all operations.

- 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

Observation. Most primitive operations take constant time.

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]	$c_7 N^2$

Caveat. Non-primitive operations often take more than constant time.

Mathematical models for running time

In principle, accurate mathematical models are available.

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**.

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)

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 $2N$

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	
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

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++;
```

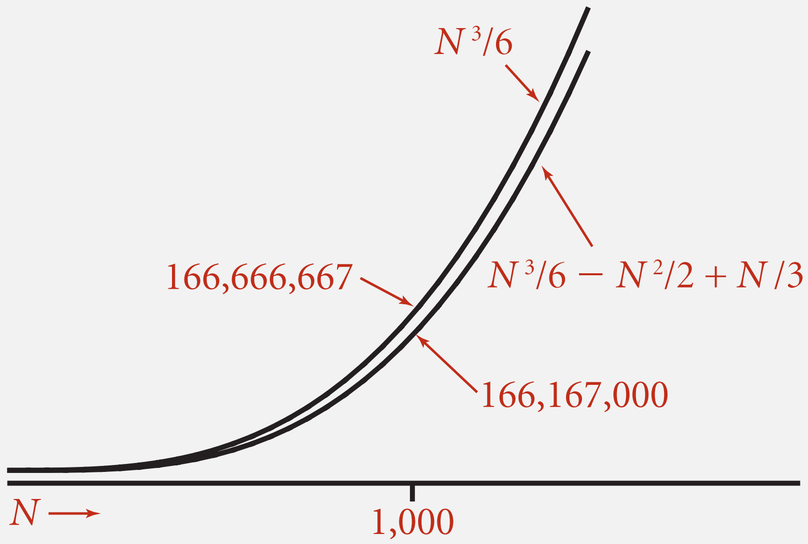
$$0 + 1 + 2 + \ldots + (N-1) = \frac{1}{2} N (N-1) = \binom{N}{2}$$

operation	frequency
variable declaration	$N + 2$
assignment statement	$N + 2$
less than compare	$\frac{1}{2} (N+1)(N+2)$
equal to compare	$\frac{1}{2} N (N-1)$
array access	$N (N-1)$
increment	$\frac{1}{2} N (N-1)$ 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



Leading-term approximation

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

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

Ex 3. $\frac{1}{6} N^3 \underbrace{- \frac{1}{2} N^2 + \frac{1}{3} N}_{} \quad \sim \frac{1}{6} N^3$

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

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	$N + 2$	$\sim N$
assignment statement	$N + 2$	$\sim N$
less than compare	$\frac{1}{2}\,(N + 1)\,(N + 2)$	$\sim \frac{1}{2}\,N^2$
equal to compare	$\frac{1}{2}\,N\,(N - 1)$	$\sim \frac{1}{2}\,N^2$
array access	$N\,(N - 1)$	$\sim N^2$
increment	$\frac{1}{2}\,N\,(N - 1)$ to $N\,(N - 1)$	$\sim \frac{1}{2}\,N^2$ to $\sim N^2$

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. $\sim 1/6\, N^3$ array accesses.

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

Diversion: estimating a discrete sum

Q. How to estimate a discrete sum?

A1. Take a 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. $1^k + 2^k + \ldots + N^k$.

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

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

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

Ex 4. 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

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

Common order-of-growth classifications

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

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

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

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

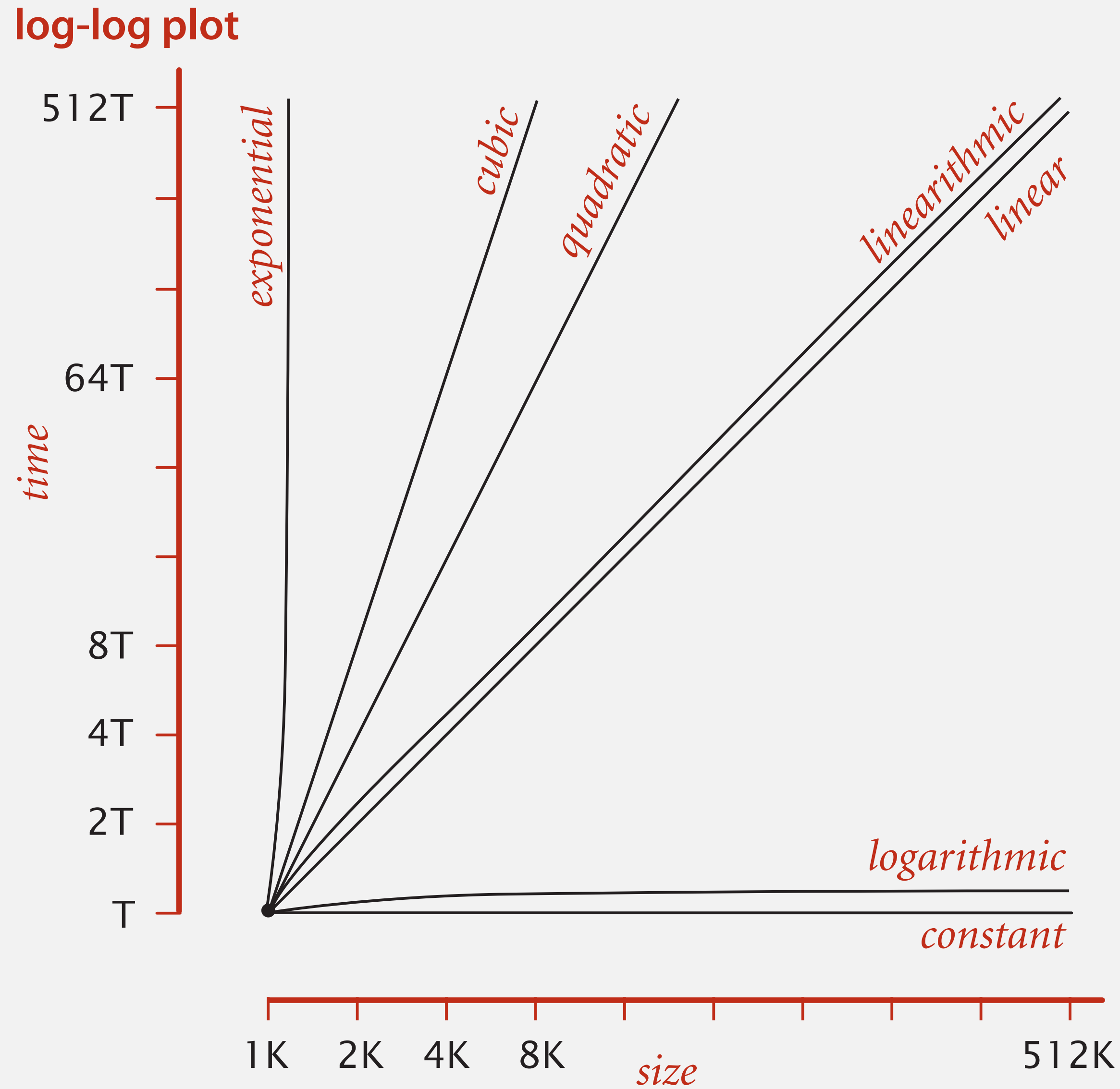
```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      for (int k = j+1; k < N; k++)
         if (a[i] + a[j] + a[k] == 0)
            count++;
```

Common order-of-growth classifications

Good news. The set of functions

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

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++) { ... }`	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			
log N	any			
N	millions			
N log N	hundreds of thousands			
N^2	hundreds			
N^3	hundred			
2^N	20			

N size scale？N的規模。

A 5MB hard drive being shipped by IBM, 1956

Practical implications of order-of-growth

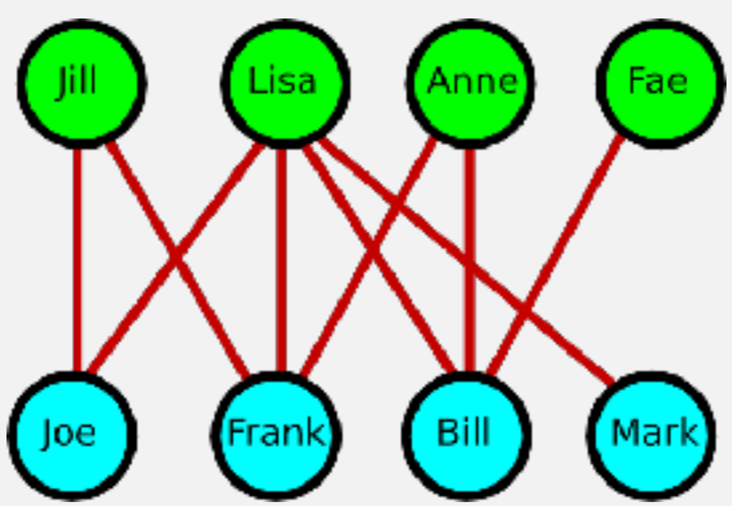
growth rate	problem size solvable in minutes				time to process millions of inputs			
	1970s	1980s	1990s	2000s	1970s	1980s	1990s	2000s
1	any	any	any	any	instant	instant	instant	instant
log N	any	any	any	any	instant	instant	instant	instant
N	millions	tens of millions	hundreds of millions	billions	minutes	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

$N = 1{,}000{,}000$

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

Consider the following scenario in your future

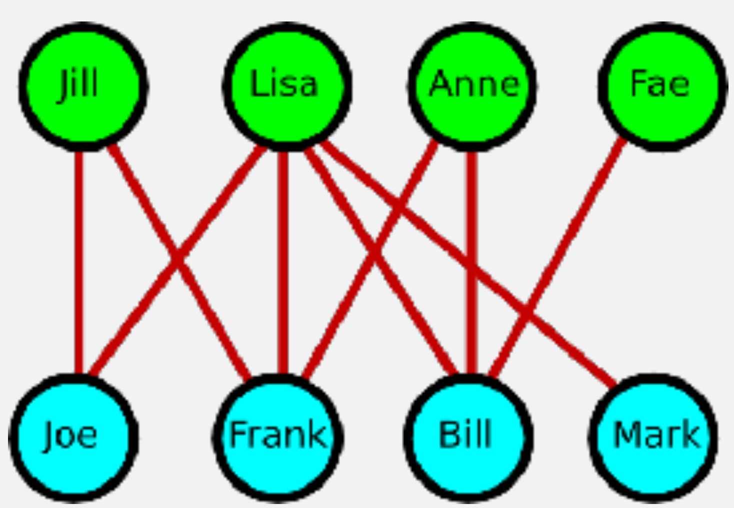


how to write a problem to assist couple matching?

Consider the following scenario in your future

Here Comes a Challenge

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



how to address the challenge?

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

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

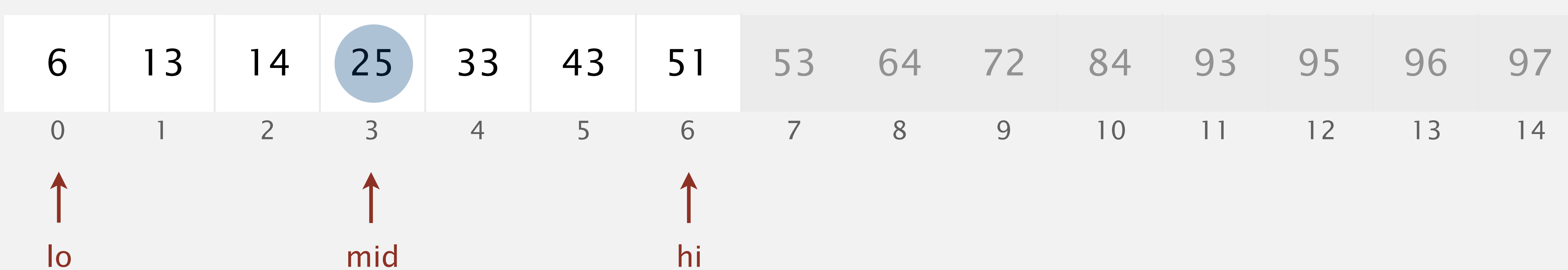
Binary search demo

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

Binary search. Compare key against middle entry.

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

successful search for 33



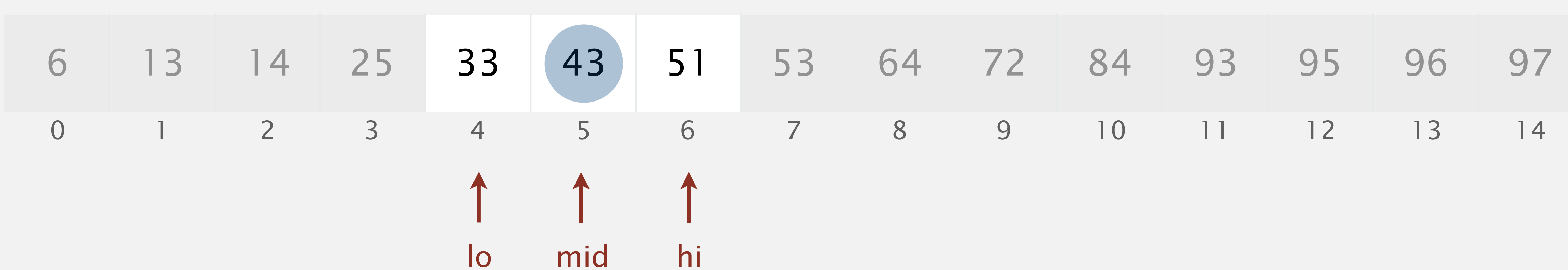
Binary search demo

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



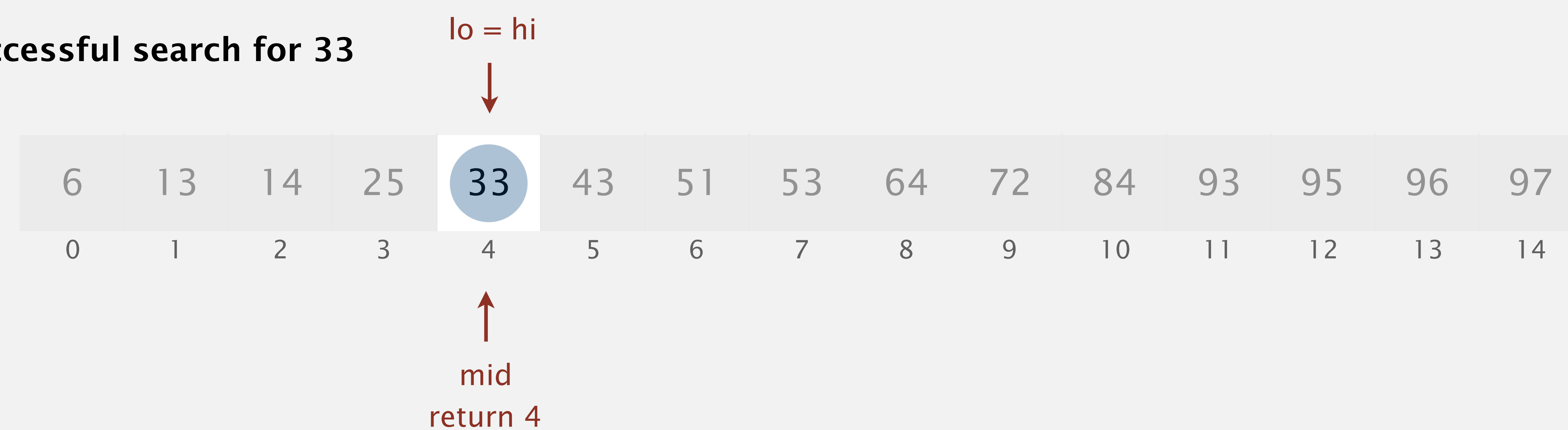
Binary search demo

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

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?

Binary search. Compare key against middle entry.

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

unsuccessful 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
↑ lo			↑ mid			↑ hi								

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

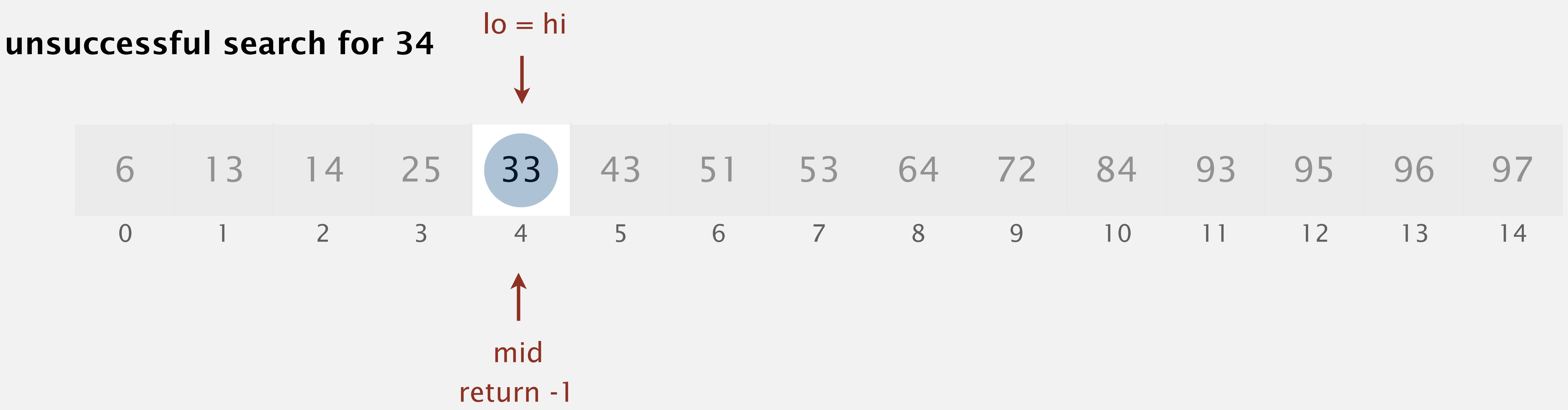
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				↑ lo	↑ mid	↑ hi								

Binary search demo

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

Binary search. Compare key against middle entry.

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



Binary search: Java implementation

Trivial to implement

```
public static int rank(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"

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)

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

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

TwoSumFast

```
import java.util.Arrays;

public class TwoSumFast
{
   public static int count(int[] a)
   {  // Count pairs that sum to 0.
      Arrays.sort(a);
      int N = a.length;
      int cnt = 0;
      for (int i = 0; i < N; i++)
         if (BinarySearch.rank(-a[i], a) > i)
            cnt++;
      return cnt;
   }

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

ThreeSumFast ???

An $N^2 \log N$ algorithm for 3-SUM

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 a sort.
- Step 2: $N^2 \log N$ with binary search.

Remark. Can achieve N^2 by modifying binary search step.

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

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

ThreeSumFast.java

Guiding principle. Typically, better order of growth $\Rightarrow$ faster in practice.

Homework Assignment #2

We need more speed for threesum problem

public class Algorithm3SumFastest		
int	count(int a[])	*return the number of triples whose sum equals to zero*

No delay is allowed. Submit your Java code and class file to E-campus system

Deadline is 3/27 Monday pm23:59

Analysis of Algorithms

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

Types of analyses

Best case. Lower bound on cost.

- 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.

this course

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: $\sim \lg N$

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: focus on the worst case.

Upper bound. Performance guarantee of algorithm for any input.

Lower bound. Proof that no algorithm can do better.

Optimal algorithm. Lower bound = upper bound (to within a constant factor).

Commonly-used notations in the theory of algorithms

notation	provides	example	shorthand for	used to
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$ $\vdots$	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$ $\vdots$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2} N^2$ N^5 $N^3 + 22 N \log N + 3 N$ $\vdots$	develop lower bounds

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 $\Omega(N)$.

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 $\Theta(N)$.

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

Commonly-used notations in the theory of algorithms

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 order of growth	$\Theta(N^2)$	$\frac{1}{2}\,N^2$ $10\,N^2$ $5\,N^2 + 22\,N \log N + 3N$	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.

This course. Focus on approximate models: use Tilde-notation

ANALYSIS OF ALGORITHMS

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

Basics

Bit. 0 or 1.

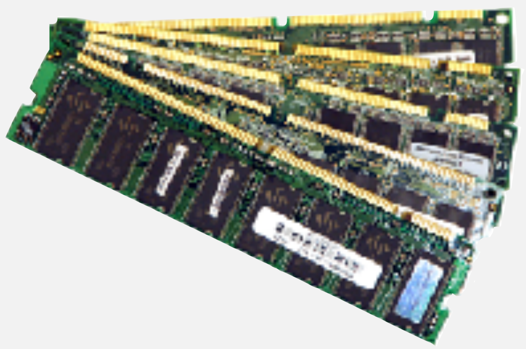
NIST

most computer scientists

Byte. 8 bits.

Megabyte (MB). 1 million or 2^{20} bytes.

Gigabyte (GB). 1 billion or 2^{30} bytes.



64-bit machine. We 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

primitive types

Array overhead. 24 bytes.

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

one-dimensional arrays

type	bytes
char[][]	$\sim 2MN$
int[][]	$\sim 4MN$
double[][]	$\sim 8MN$

two-dimensional arrays

Typical memory usage for objects in Java

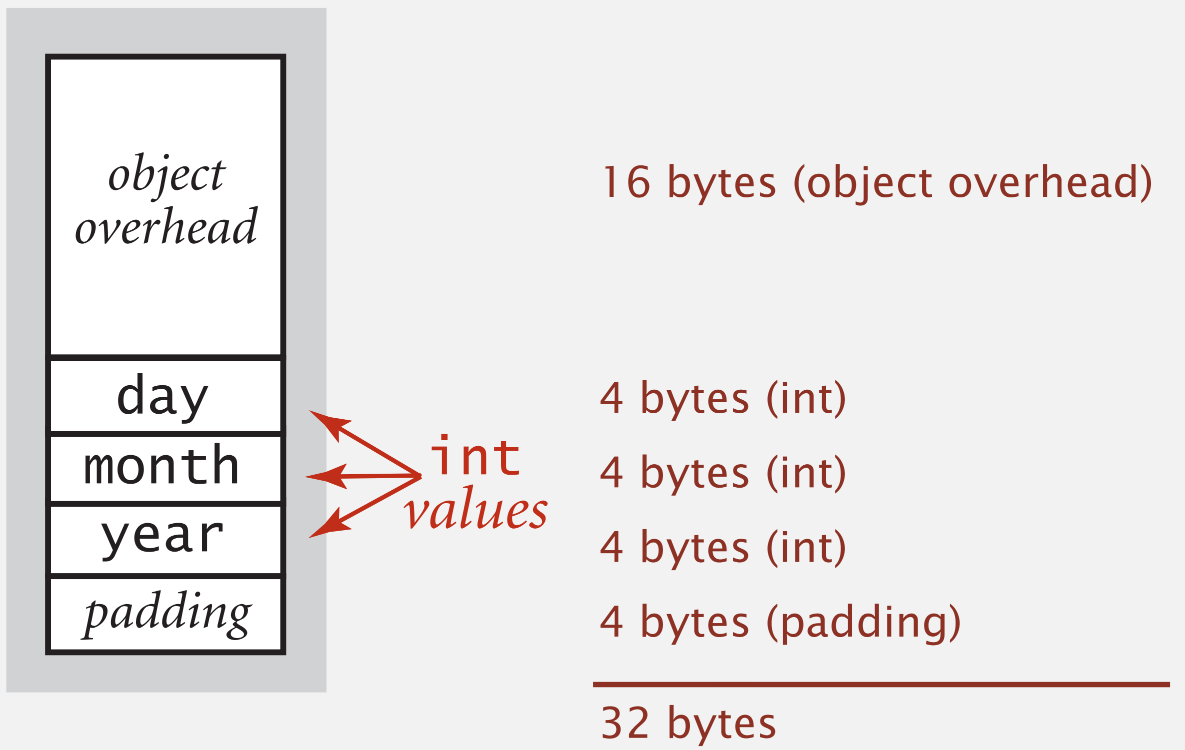
Object overhead. 16 bytes.

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;
...
}
```



Typical memory usage for objects in Java

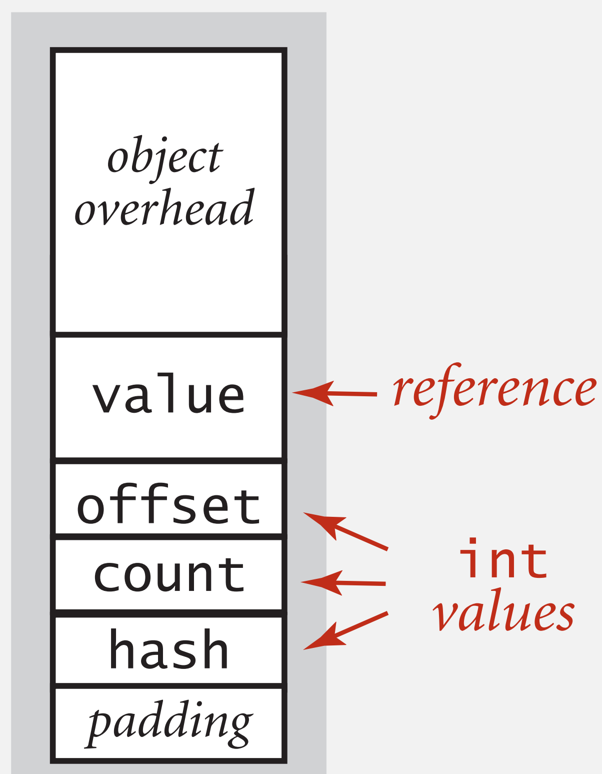
Object overhead. 16 bytes.

Reference. 8 bytes.

Padding. Each object uses a multiple of 8 bytes.

Ex 2. A virgin String of length N uses $\sim 2N$ 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.
- Padding: round up to multiple of 8 bytes.

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

Shallow memory usage: Don't count referenced objects.

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

Memory profiler

Classmexer library. Measure memory usage 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

Example

Q. How much memory does WeightedQuickUnionUF use as a function of N ?
Use tilde notation to simplify your answer.

```
public class WeightedQuickUnionUF
{
   private int[] id;
   private int[] sz;
   private int count;

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

16 bytes (object overhead)

8 + (4N + 24) bytes each (reference + int[] array)

4 bytes (int)

4 bytes (padding)

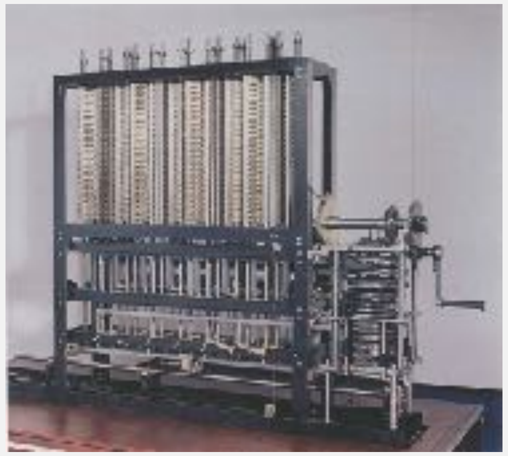
8N + 88 bytes

A.

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 behavior.

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.