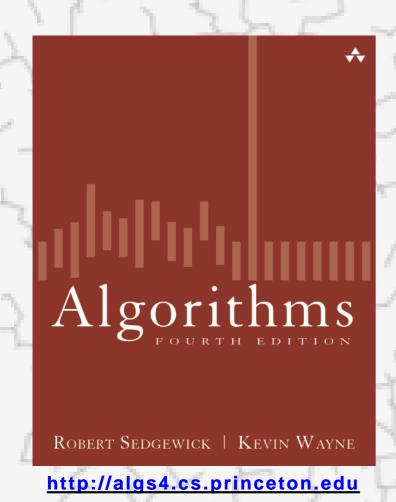
## Algorithms



## 1.4 ANALYSIS OF ALGORITHMS

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

## 1.4 ANALYSIS OF ALGORITHMS

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

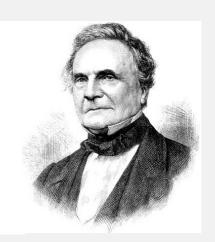
# Algorithms

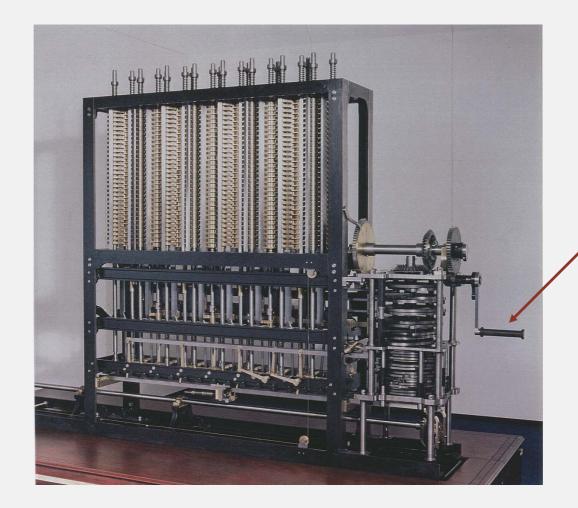
ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

## 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** 

#### Cast of characters



Programmer needs to develop a working solution.





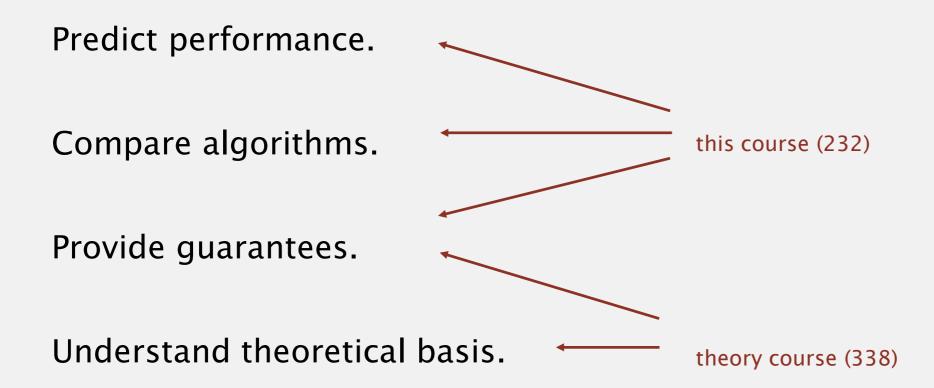
Client wants to solve problem efficiently.

Student might play any or all of these roles someday.



Theoretician wants to understand.

### Reasons to analyze algorithms



Primary practical reason: avoid performance bugs.



client gets poor performance because programmer did not understand performance characteristics



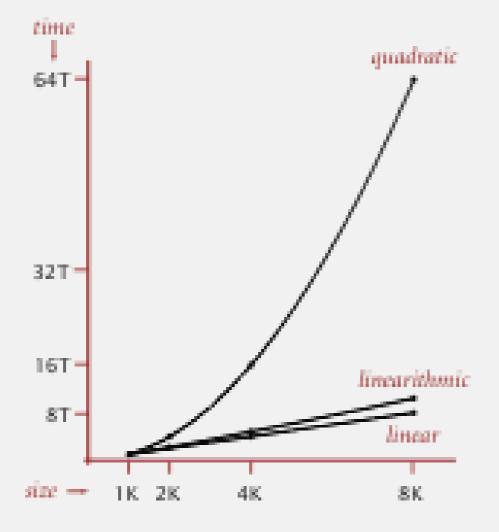
### Some algorithmic successes

#### Discrete Fourier transform.

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



Friedrich Gauss 1805









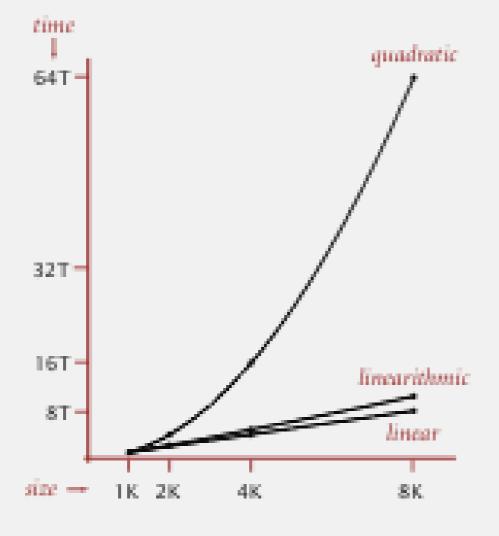
## Some algorithmic successes

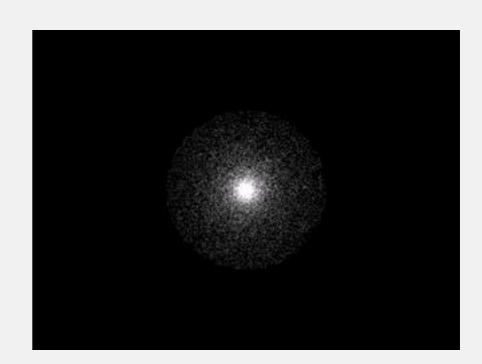
#### N-body simulation.

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



**Andrew Appel** 





## The challenge

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

Why is my program so slow?

Why does it run out of memory?



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



## Sometimes We Can't Afford To Do It

- We look at how expensive something is before we buy it
- Same with computers
  - Even if our problem is decidable, the algorithm may take too many resources to decide it
  - How long will it take to compute something
  - Some take so many resources that they become the same as undecidable problems



## What Resources Are We Talking About?

- Running time
- Memory space
- Tied to the input
  - Size/amount of the input



## Complexity

- Focusing on running time
  - Called time complexity
- Size of the input (not the input itself)
  - Let's call it N to be generic
- If we say an algorithm runs in time 5N
  - We mean that it never performs more than 5 x N (5N) basic actions with N input



## Time Complexity

- Big O Notation
- https://www.youtube.com/watch?v=v4cd1O4zkGw





```
#include <stdio.h>
// Implements the selection sort
// algorithm
void SelectionSort(int arr[], int size) {
  int i, j;
  for (i = 0; i < size; i++) {
     for (j = i ; j < size; j++) {
       if (arr[i] > arr[j]) {
          Swap(&arr[i], &arr[i]);
```

What's the complexity?

For every trip through the outer loop, we will go through the inner for loop N times

Our time complexity is then

Try it for an array of 4 items

The performance of this sort is The O means tends toward a limit



## Measuring Complexity

- Let's think about searching a long list of names for a particular name
  - Inefficient algorithm
    - Starting at the beginning of the list
    - Search each list item to see if it matches the given name
    - For each list item, determine if we are at the end of the list
  - This has 2 actions for every name
  - The time complexity is 2N where N stands for the length of the list
  - How could we improve this?



## Measuring Complexity

- We could add the given name to the end of the list
- This means we will always find the name
- When we find the name, we check (only once) if we are at the end of the list
- Our new time complexity is now N
  - 50% improvement



## Searching a List

- Even though we cut our search time complexity from 2N to N, both are proportional to N
- The time grows linearly with N
  - Runs in linear time
- We want to find order of magnitude improvements not just linear improvements
  - We want to improve the performance greater as N increases
- How can we improve our list search by an order of magnitude?



## Searching a List

- By using a sorted list
- binary search
  - We can start in the middle and search either the first half
  - We continue to cut down by halves until we get to the string we are looking for
- O(log<sub>2</sub>N)
- logarithmic time much better than linear time



## Sorting Algorithms

- Search the list for the smallest item
  - Repeat for next smallest item
  - O(N<sup>2</sup>) upper bound
  - Known as quadratic time
- heap sort/merge sort
  - O(Nlog<sub>2</sub>N) lower bound

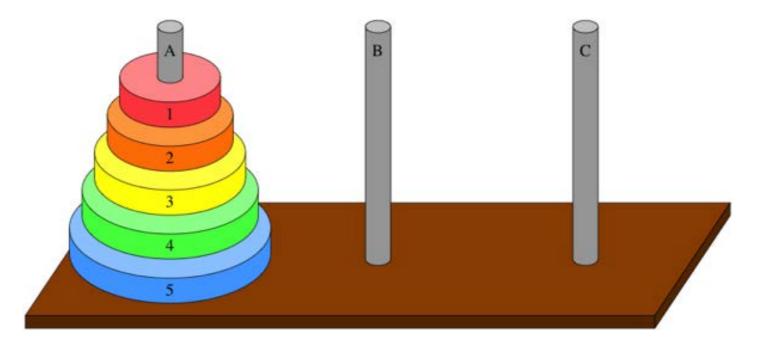


## Towers of Hanoi

- 2<sup>N</sup>-1 moves
- Exponential time
- With 64 rings
- 2<sup>64</sup>-1 moves

• If they could move a ring every 5 seconds, would take 3

trillion years to finish



# Fastest to Slowest Time Complexities



- Constant
  - Linear
- Quadratic
- Polynomial (N<sup>K</sup>)
- •Exponential (2<sup>N</sup>, N!, N<sup>N</sup>)

Gets very bad as N gets large

# Running Time of 4 Algorithms Computer capable of a 1M instructions per sec



#### Input Size

	10	20	50	100	200
$N^2$	1/10000 sec	1/2500 sec	1/400 sec	1/100 sec	1/25 sec
$N^5$	1/10 sec	3.2 secs	5.2 mins	2.8 hrs	3.7 days
2 <sup>N</sup>	1/1000 sec	1 sec	35 years	400 tril cen	45 dig cen
N <sup>N</sup>	2.8 hrs	3.3 tril yrs	70 dig cen	185 dig cen	445 dig cen

Taken from H. R. Lewis and C. H. Papdimitrious (1978), The efficiency of algorithms in Computers LTD, what they really can't do by David Harel

# Fastest to Slowest Time Complexities



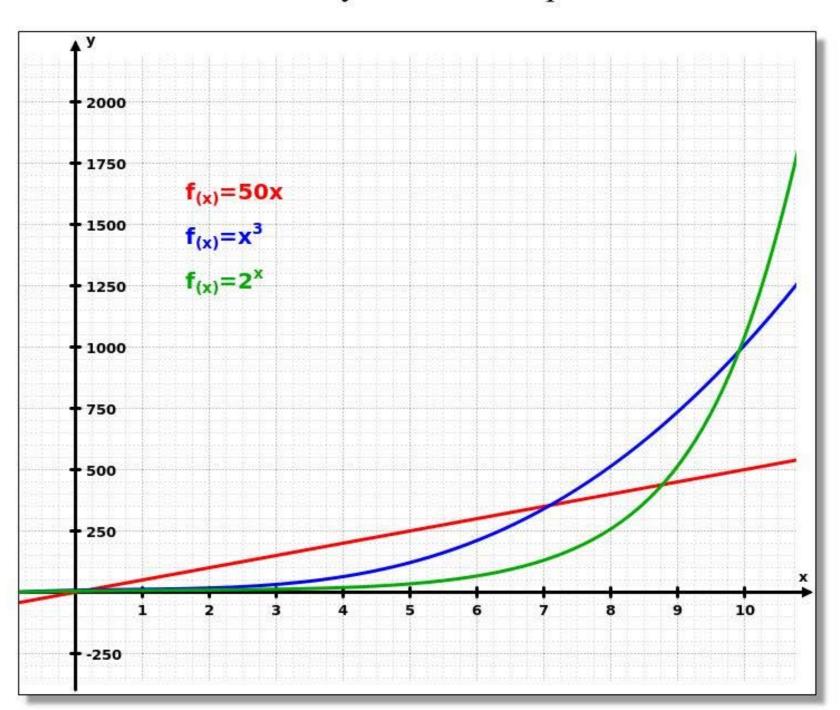
Polynomial (N<sup>K</sup>)
 Exponential

•N<sup>100</sup> is slower than 2<sup>N</sup> until N reaches 996

## Fastest to Slowest Time Complexities

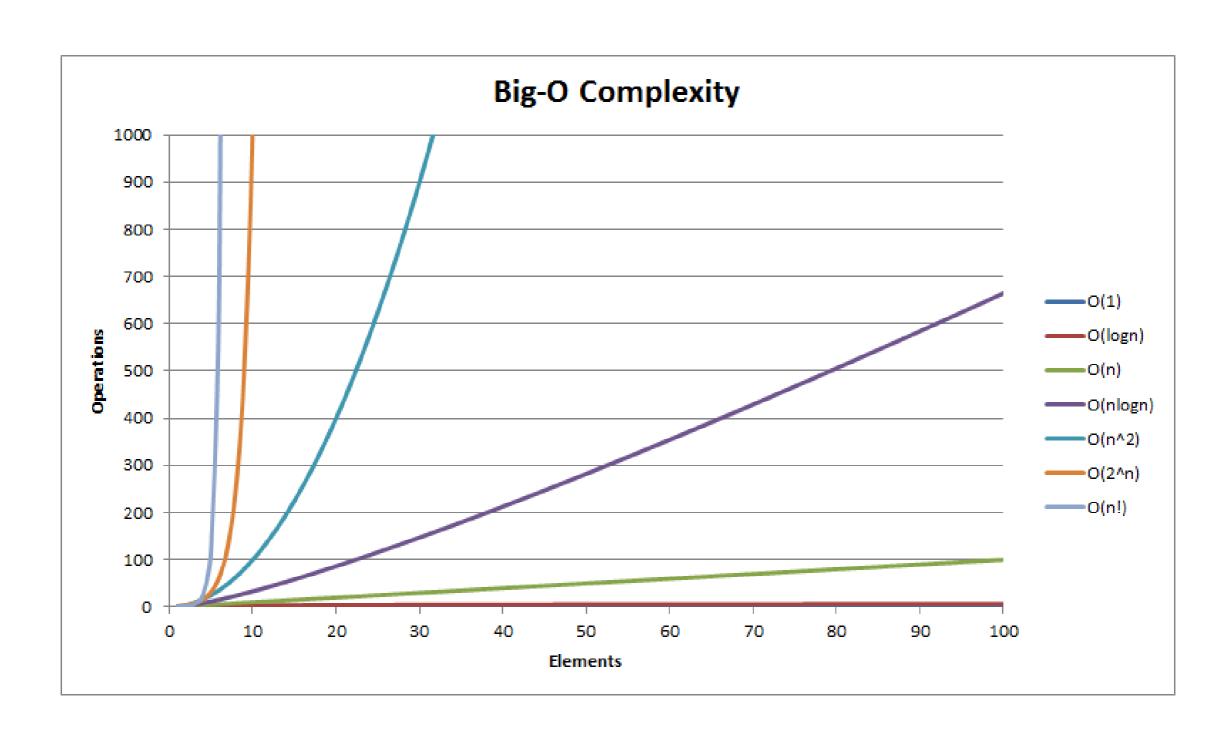


## Linear vs Polynomial vs Exponential



# Fastest to Slowest Time Complexities

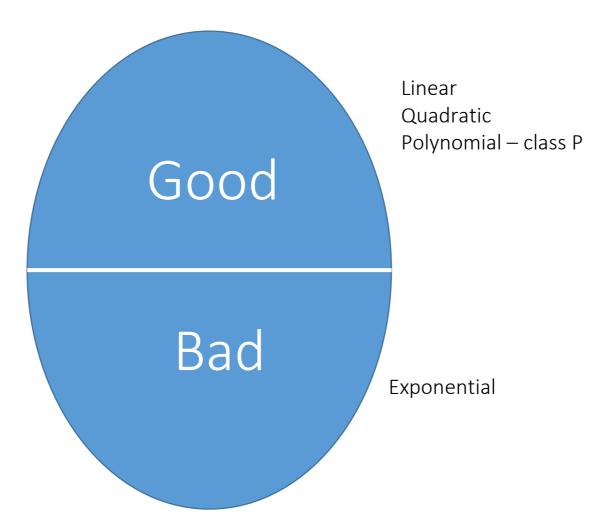






## Classification

Time Complexity



# 1.4 ANALYSIS OF ALGORITHMS

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

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

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

ThreeSum.java \*ints.txt

What is the time complexity of this algorithm?

Context. Deeply related to problems in computational geometry.

## Measuring the running time

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



#### % java ThreeSum 1Kints.txt



70

#### % java ThreeSum 2Kints.txt



tick tick

528

#### % java ThreeSum 4Kints.txt



tick tick

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)
```

ThreeSum.java StopWatch.java



## Empirical analysis

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

Run with 1Kint.txt – how long did it take?
What do you predict will be the time it takes for 2Kint.txt?
What about 4Kint.txt?

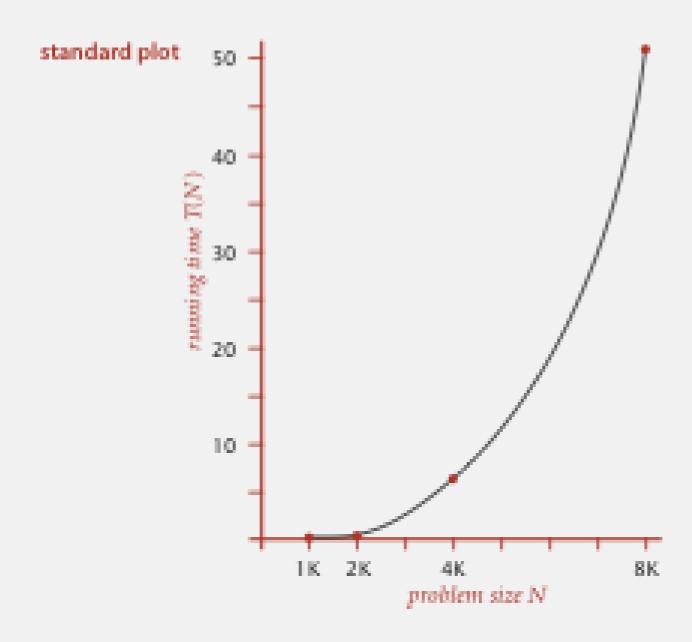
## Empirical analysis

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.



#### 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<sup>3</sup> [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.

- 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.

#### System dependent effects.

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

determines constant 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

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

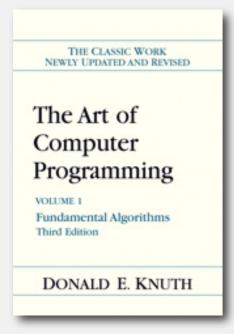
# 1.4 ANALYSIS OF ALGORITHMS

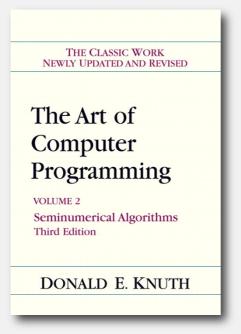
- introduction
- observations
- mathematical models
- order-of-growth classifications
- theory of algorithms
- memory

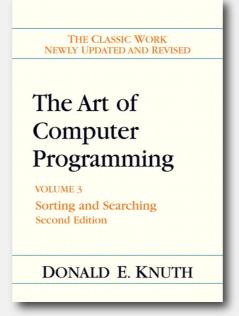
# Mathematical models for running time

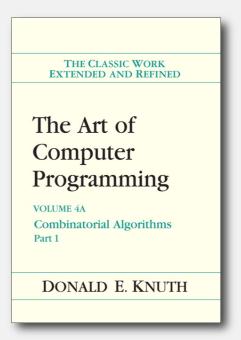
Total running time: sum of cost × frequency for all operations.

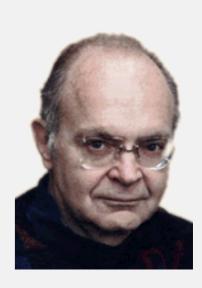
- Need to analyze program to determine set of operations.
- Cost depends on machine, compiler.
- Frequency depends on algorithm, input data.











Donald Knuth
1974 Turing Award

In principle, accurate mathematical models are available.

# Cost of basic operations

# 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
•••	•••	•••

<sup>†</sup> Running OS X on Macbook Pro 2.2GHz with 2GB RAM

# Cost of basic operations

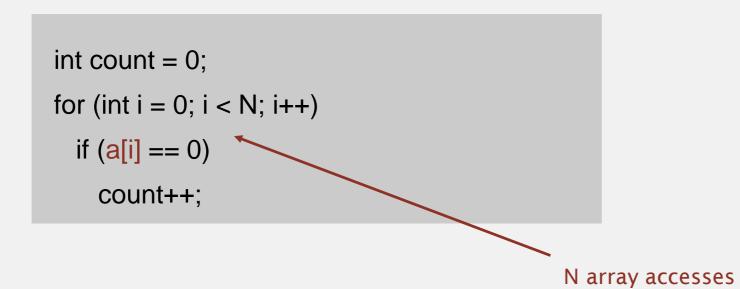
Observation. Most primitive operations take constant time.

operation	example	nanoseconds †
variable declaration	int a	$c_1$
assignment statement	a = b	$c_2$
integer compare	a < b	<i>C</i> 3
array element access	a[i]	<i>C</i> 4
array length	a.length	<i>C</i> 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.

Example: 1-Sum

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

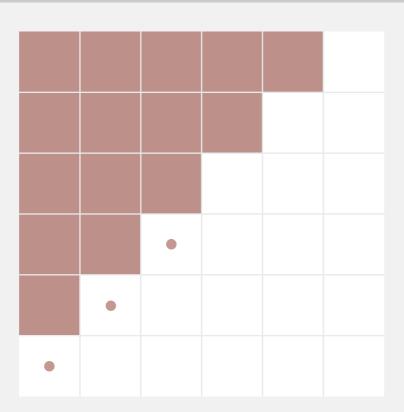


operation	frequency
variable declaration	2
assignment statement	2
less than compare	N+1
equal to compare	N
array access	N
increment	<i>N</i> to 2 <i>N</i>

# Example: 2-Sum

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

Pf. [ n even]

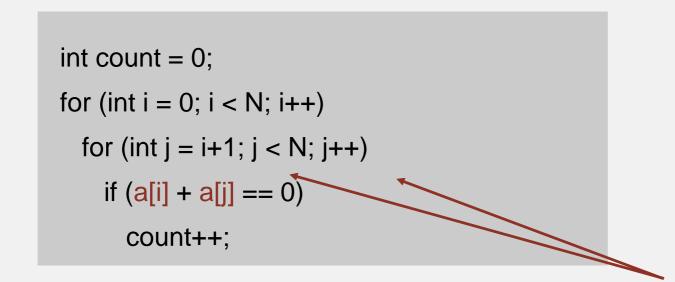


$$0+1+2+\ldots+(N-1) \;=\; \frac{1}{2}N^2\;-\;\frac{1}{2}N$$
 half of square diagonal

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

# Example: 2-Sum

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



$$0+1+2+...+(N-1) = \frac{1}{2}N(N-1)$$
$$= {N \choose 2}$$

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 $N(N-1)$

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

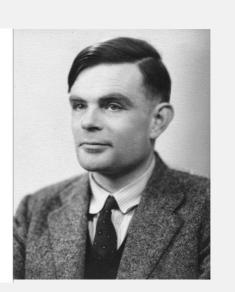
#### ROUNDING-OFF ERRORS IN MATRIX PROCESSES

By A. M. TURING

(National Physical Laboratory, Teddington, Middlesex)
[Received 4 November 1947]

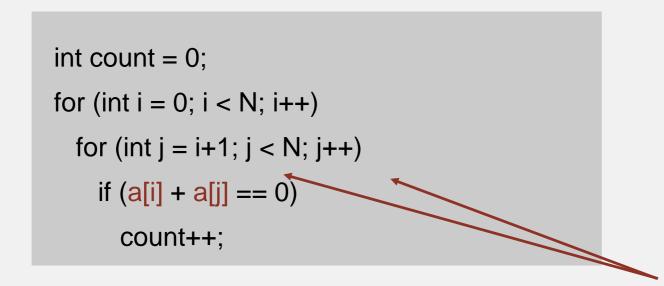
#### SUMMARY

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.



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

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

(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

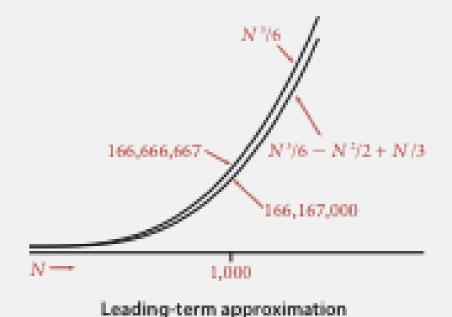
**Ex 1.** 
$$\frac{1}{6}N^3 + 20N + 16$$
  $\sim \frac{1}{6}N^3$ 

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

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)



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

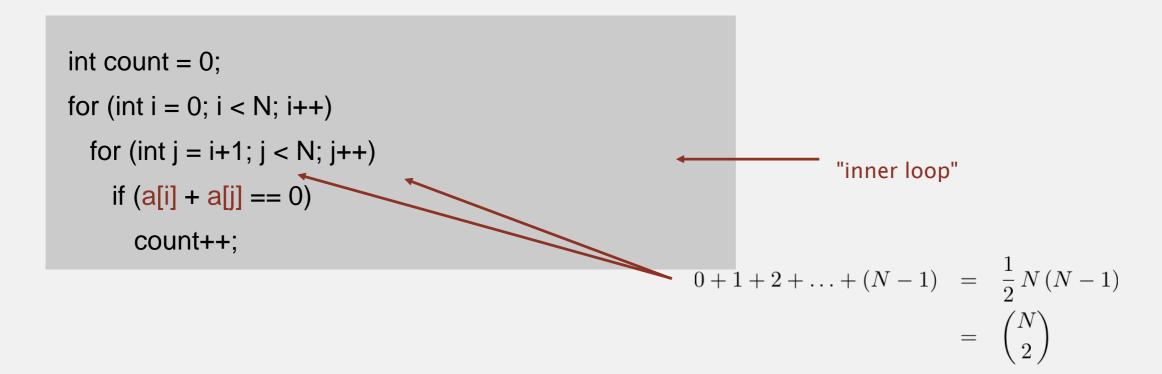
# Simplification 2: tilde notation

- 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	$\frac{1}{2}(N+1)(N+2)$	$\sim \frac{1}{2} N^2$	
equal to compare	$\frac{1}{2}N(N-1)$	$\sim \frac{1}{2} N^2$	
array access	N(N-1)	~ N <sup>2</sup>	
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?

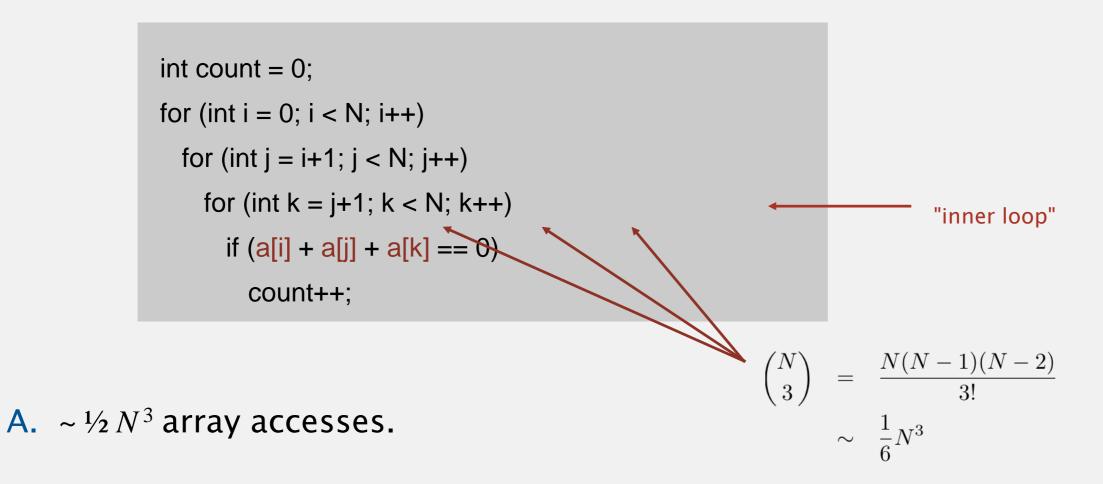


A. ~  $N^2$  array accesses.

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*?



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.** 
$$1 + 2 + ... + N$$
.

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

Ex 2. 
$$1^k + 2^k + ... + N^k$$
.

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

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

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

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

# 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 4. 
$$1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots$$

$$\sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i = 2$$

$$\int_{x=0}^{\infty} \left(\frac{1}{2}\right)^x dx = \frac{1}{\ln 2} \approx 1.4427$$

Caveat. Integral trick doesn't always work!

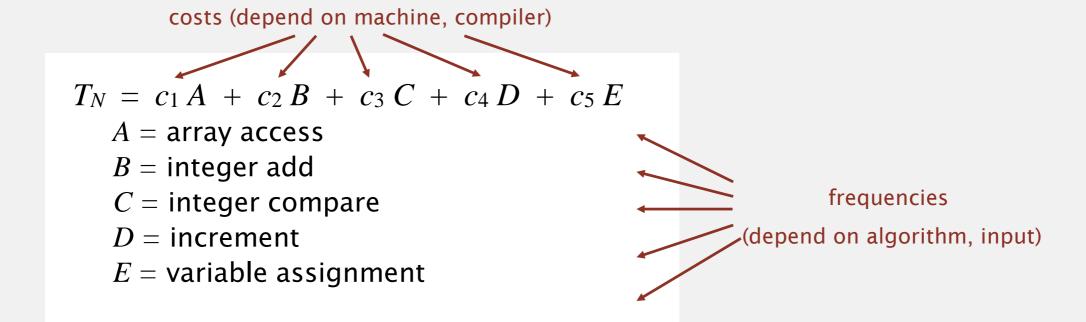
# Mathematical models for running time

In principle, accurate mathematical models are available.

#### In practice,

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





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

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

# 1.4 ANALYSIS OF ALGORITHMS

- introduction
- observations
- mathematical models
- order-of-growth classifications
- theory of algorithms
- 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++;
```

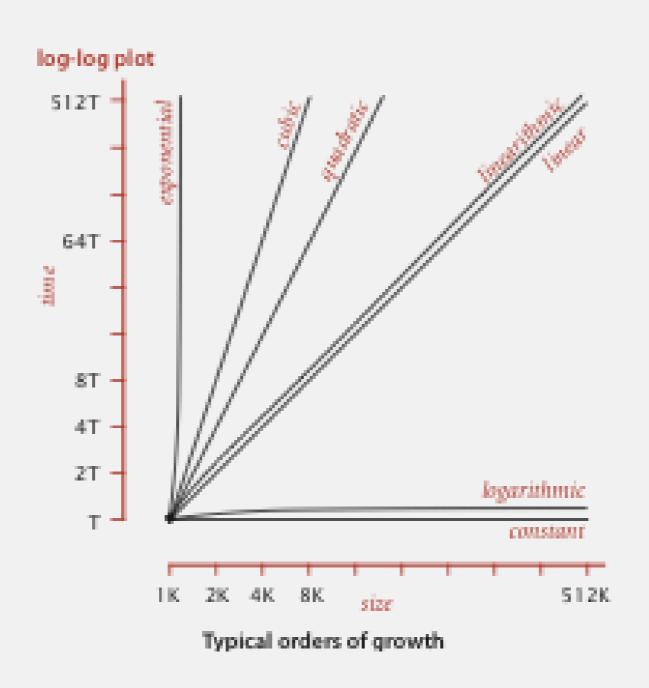
Typical usage. With running times.

# 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.



# 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	n o ma o	do a avintion	effect on a program that runs for a few seconds		
rate	name	description	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 <sup>3</sup>	cubic	not practical for medium problems	several weeks	4-5x	
2 <sup>N</sup>	exponential	useful only for tiny problems	forever	1 x	

# Binary search demo

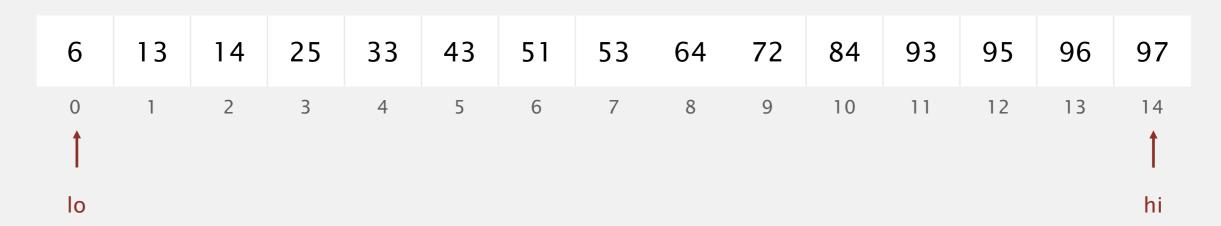
Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

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



#### successful search for 33



# Binary search: Java implementation

#### Trivial to implement?

- First binary search published in 1946.
- First bug-free one 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;
          (key < a[mid]) hi = mid - 1;
                                                                                 one "3-way compare"
    else if (key > a[mid]) lo = mid + 1;
    else return mid;
 return -1;
```

Invariant. If key appears in the array a[], then a[lo]  $\leq$  key  $\leq$  a[hi].

# 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  $\Rightarrow$  faster in practice.

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

# 1.4 ANALYSIS OF ALGORITHMS

- introduction
- observations
- mathematical models
- order-of-growth classifications
- theory of algorithms
  - 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.

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	$\Theta(N^2)$	$\frac{1}{2}N^{2}$ $10N^{2}$ $5N^{2} + 22N \log N + 3N$ $\vdots$	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	$\Omega(N^2)$	$\frac{1/2}{N^{5}}$ $N^{5}$ $N^{3} + 22 N \log N + 3 N$ $\vdots$	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?

# 1.4 ANALYSIS OF ALGORITHMS

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

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

#### Basics

Bit. 0 or 1.

NIST most computer scientists

Byte. 8 bits.

Megabyte (MB). 1 million or 2<sup>20</sup> bytes.

Gigabyte (GB). 1 billion or 2<sup>30</sup> 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

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[][]	~ 2 <i>M N</i>
int[][]	~ 4 <i>M N</i>
double[][]	~ 8 <i>M N</i>

two-dimensional arrays

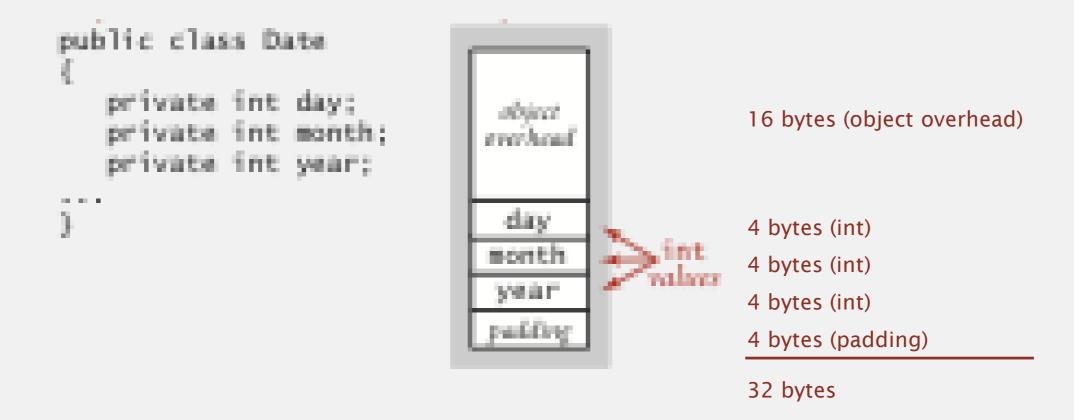
# Typical memory usage for objects in Java

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.



# 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.

# Example

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

```
16 bytes
public class WeightedQuickUnionUF
                                                                                (object overhead)
                                                                                8 + (4N + 24) bytes each
  private int[] id;
                                                                                (reference + int[] array)
  private int[] sz;
                                                                                4 bytes (int)
  private int count;
                                                                                4 bytes (padding)
                                                                                8N + 88 bytes
  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;
```

**A.**  $8N + 88 \sim 8N$  bytes.

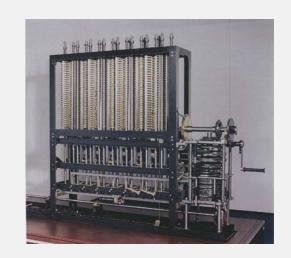
# Turning the crank: summary

#### 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.