INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

Rutgers University

1. Greatest Hits of 111

- **Memory, objects, Arrays**
- **Program stack and heap**
- **Images as 2D arrays**
- **Introduction to program analysis**
- **Running time (experimental analysis)**
- **Running time (mathematical models)**
- **Memory usage**

https://ds.cs.rutgers.edu

Last updated on 1/8/21

1. Greatest Hits of 111

- *Memory, objects, Arrays*
- *Program stack and heap*
- *Images as 2D arrays*
- *Introduction to program analysis*
- *Running time (experimental analysis)*
- *Running time (mathematical models)*
- *Memory usage*

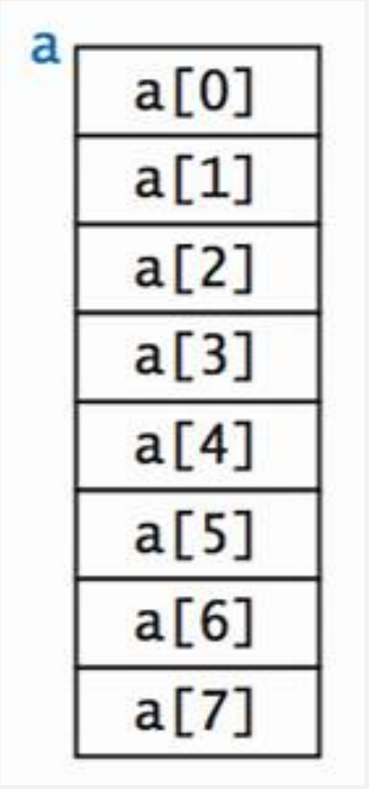
Some slides Adopted and modified from Sedgewick and Wayne

Memory, objects and arrays

LO 1.1

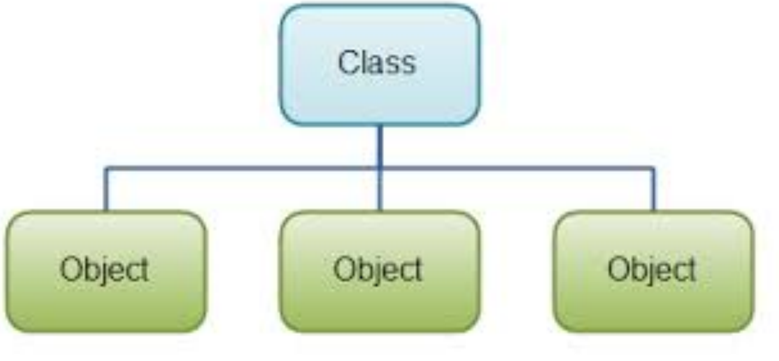
A view of physical memory storage

```
int[] a = new int[8];
```



An Array is a contiguous block of memory

```
String str1 = new String(“Welcome ”);
String str2 = new String(“to ”);
String str3 = new String(“class”);
```



Object is an instance of a class

Memory representation of an array

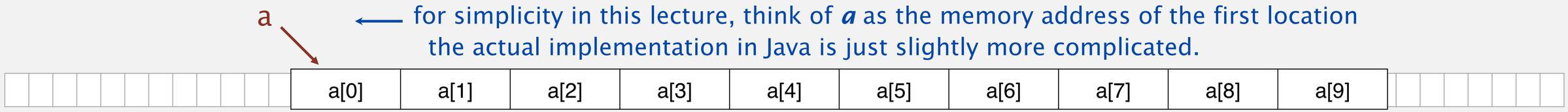
LO 1.1

An array is an indexed sequence of values of the same type.

A computer's memory is *also* an indexed sequence of memory locations.

- Each primitive type value occupies a fixed number of locations.
- *Array values are stored in contiguous locations.*

← stay tuned for many details



Critical concepts

- Indices start at 0.
- Given i, the operation of accessing the value a[i] is extremely efficient.
- The assignment b = a makes the names b and a refer to the same array.

it does *not* copy the array, as with primitive types (stay tuned for details)

Java language support for arrays

Basic support

operation	*typical code*
Declare an array reference	double[] a;
Create an array of a given length	a = new double[1000];
Refer to an array entry by index	a[i] = b[j] + c[k];
Refer to the length of an array	a.length;

Initialization options

operation	*typical code*
Default initialization to 0 for numeric types	a = new double[1000];
Declare, create and initialize in one statement	double[] a = new double[1000];
Initialize to literal values	double[] x = { 0.3, 0.6, 0.1 };

no need to use a loop like

```
for (int i = 0; i < 1000; i++)
  a[i] = 0.0;
```

BUT cost of creating an array is proportional to its length.

Programming with arrays: typical examples

create an array with random values	`double[] a = new double[n];`
	`for (int i = 0; i < n; i++)`
	`  a[i] = Math.random();`

print the array values, one per line	`for (int i = 0; i < n; i++)` `  System.out.println(a[i]);`

find the maximum of the array values	`double max = Double.NEGATIVE_INFINITY;`
	`for (int i = 0; i < n; i++)`
	`  if (a[i] > max) max = a[i];`

compute the average of the array values	`double sum = 0.0;`
	`for (int i = 0; i < n; i++)`
	`  sum += a[i];`
	`double average = sum / n;`

reverse the values within an array	`for (int i = 0; i < n/2; i++)`
	`{`
	`  double temp = a[i];`
	`  a[i] = a[n-1-i];`
	`  a[n-i-1] = temp;`
	`}`

copy sequence of values to another array	`double[] b = new double[n];`
	`for (int i = 0; i < n; i++)`
	`  b[i] = a[i];`

2D Arrays

LO 1.1, 1.2

```
double[][] a;
a = new double[m][n];
for (int i = 0; i < m; i++)
   for (int j = 0; j < n; j++)
      a[i][j] = 0;
```

2D Array Allocation

```
double[][] a = {
    { 99.0, 85.0, 98.0, 0.0 },
    { 98.0, 57.0, 79.0, 0.0 },
    { 92.0, 77.0, 74.0, 0.0 },
    { 94.0, 62.0, 81.0, 0.0 },
    { 99.0, 94.0, 92.0, 0.0 },
    { 80.0, 76.5, 67.0, 0.0 },
    { 76.0, 58.5, 90.5, 0.0 },
    { 92.0, 66.0, 91.0, 0.0 },
    { 97.0, 70.5, 66.5, 0.0 },
    { 89.0, 89.5, 81.0, 0.0 },
    {  0.0,  0.0,  0.0, 0.0 }
};
```

Initializing 2D array at compile time

Memory representation. Java represents a two-dimensional array as an array of arrays. A matrix with `m` rows and `n` columns is actually an array of length `m`, each entry of which is an array of length `n`. In a two-dimensional Java array, we can use the code `a[i]` to refer to the ith row (which is a one-dimensional array). Enables ragged arrays.

1. Greatest Hits of 111

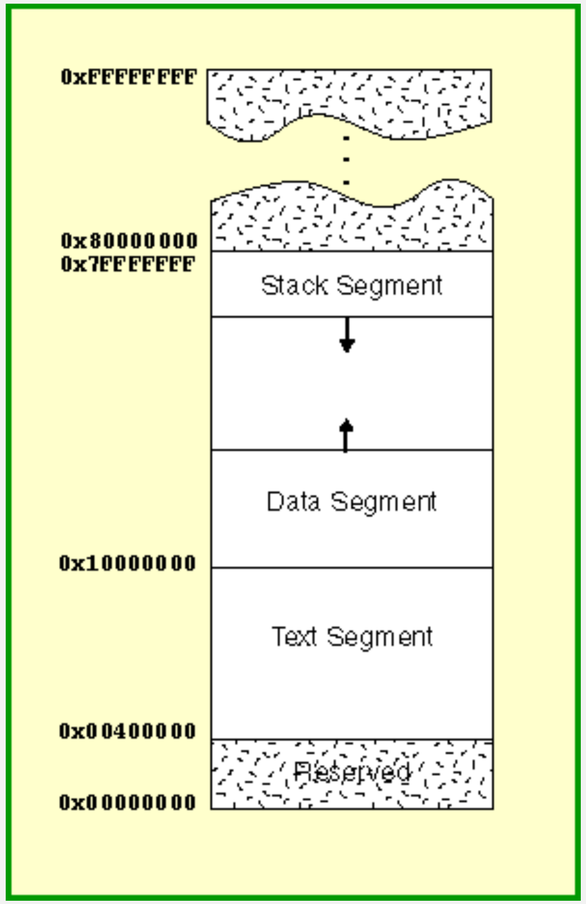
- *Memory, objects, Arrays*
- *Program stack and heap*
- *Images as 2D arrays*
- *Introduction to program analysis*
- *Running time (experimental analysis)*
- *Running time (mathematical models)*
- *Memory usage*

Some slides Adopted and modified from Sedgewick and Wayne

Program stack and heap memory

LO 1.1

- Runtime Stack - a location in the memory where **static memory** is kept



- Runtime Heap – a location in the memory where **dynamic memory** is kept

Heap memory can be fragmented as they are allocated (new) and deallocated (garbage collection) as needed

Source: https://chortle.ccsu.edu/

Quiz

LO 1.1

Identify the locations of this code where static memory (compile time) and dynamic memory (run time) are allocated?

```
public static void main(String[] args) {
    int capacity = Integer.parseInt(args[0]);
    ArrayStackOfStrings stack = new ArrayStackOfStrings(capacity);
    while (!StdIn.isEmpty()) {
        String item = StdIn.readString();
        if (!item.equals("-")) {
            stack.push(item);
        }
        else {
            StdOut.print(stack.pop() + " ");
        }
    }
    StdOut.println();
}
```

Quiz

LO 1.1

Identify the locations of this code where static memory (compile time) and dynamic memory (run time) are allocated?

```
public static void main(String[] args) {
    int capacity = Integer.parseInt(args[0]);
    ArrayStackOfStrings stack = new ArrayStackOfStrings(capacity);
    while (!StdIn.isEmpty()) {
        String item = StdIn.readString();
        if (!item.equals("-")) {
            stack.push(item);
        }
        else {
            StdOut.print(stack.pop() + " ");
        }
    }
    StdOut.println();
}
```

Answer:

Stack (static memory, size known at compile time): `int capacity`
`ArrayStackOfStrings stack`

Heap (dynamic memory, size known at run time) : `ArrayStackOfStrings(capacity)`

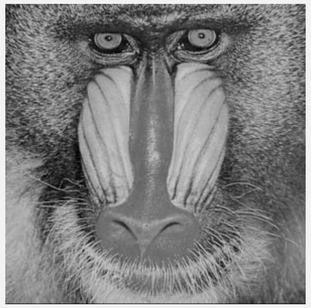
1. Greatest Hits of 111

- *Memory, objects, Arrays*
- *Program stack and heap*
- *Images as 2D arrays*
- *Introduction to analysis*
- *Running time (experimental analysis)*
- *Running time (mathematical models)*
- *Memory usage*

Some slides Adopted and modified from Sedgewick and Wayne

Images as 2D Arrays

Color image



grayscale image

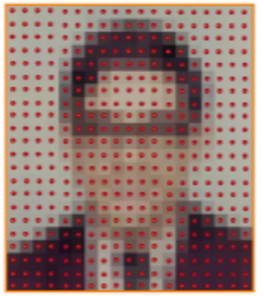
Red

Green

Blue

- An image is a 2D rectilinear array of pixels

Continuous image



Digital image

A pixel is a sample, not a little square!

(255, 255, 0)	(255, 255, 0	(255, 255, 0	(255, 255, 0			

Each entry has (R,G,B) color values
8-bit Integer between 0-255

Processing Images

Constructor and Description
`Picture(File file)` Creates a picture by reading the image from a PNG, GIF, or JPEG file.
`Picture(int width, int height)` Creates a `width`-by-`height` picture, with `width` columns and `height` rows, where each pixel is black.
`Picture(Picture picture)` Creates a new picture that is a deep copy of the argument picture.
`Picture(String name)` Creates a picture by reading an image from a file or URL.

```
int r = (rgb >> 16) & 0xFF;
int g = (rgb >>  8) & 0xFF;
int b = (rgb >>  0) & 0xFF;
```

Extracting RGB values from a 32-bit color pixel

```
int rgb = (r << 16) + (g << 8) + (b << 0);
```

Given RGB values (0-255) pack them into a 32-bit pixel

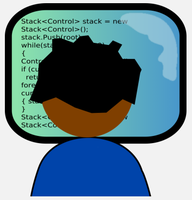
See: https://algs4.cs.princeton.edu/code/javadoc/edu/princeton/cs/algs4/Picture.html

1. Greatest Hits of 111

- *Stack and heap*
- *Memory, objects, Arrays*
- *Images as 2D arrays*
- *Introduction to program analysis*
- *Running time (experimental analysis)*
- *Running time (mathematical models)*
- *Memory usage*

Some slides Adopted and modified from Sedgewick and Wayne

Cast of characters



programmer needs to
develop a working solution

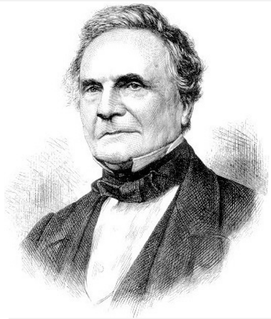
client wants to solve
problem efficiently

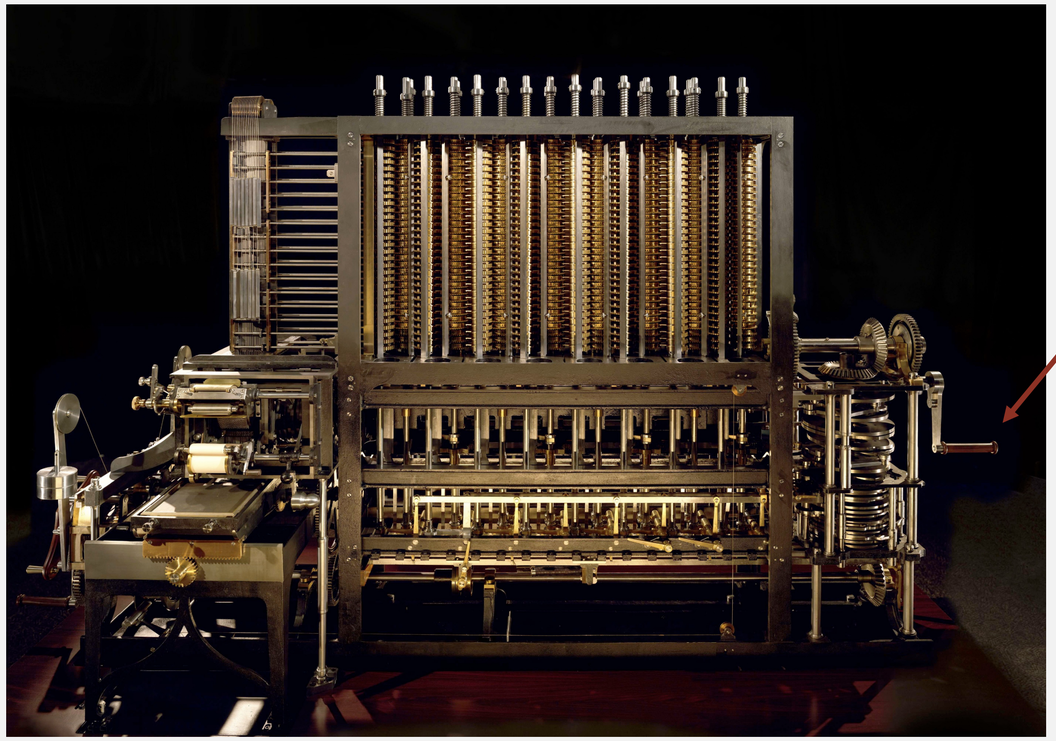
theoretician seeks
to understand

student (you)
might play all of
these roles someday

Running time

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

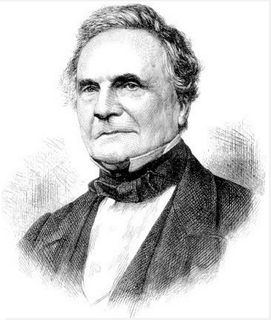




how many times do you have to turn the crank?

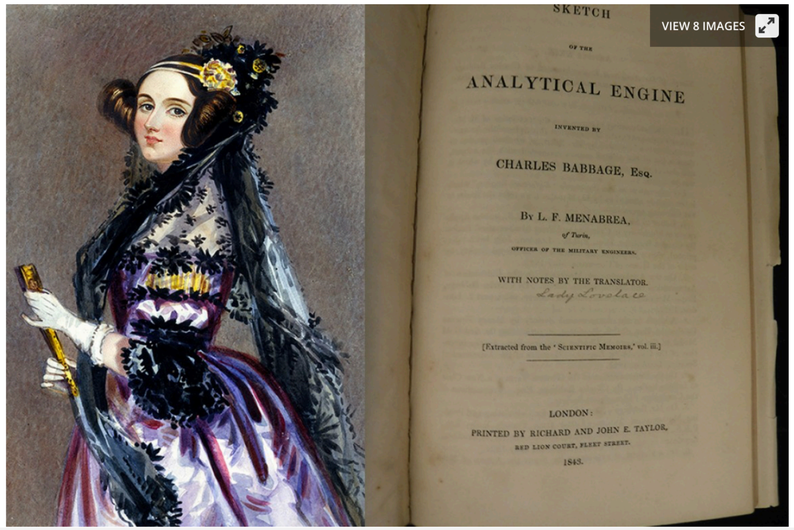
Running time

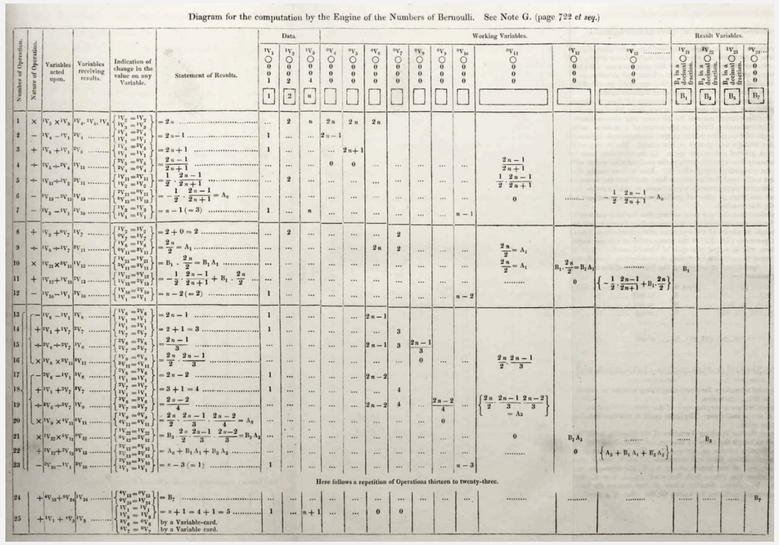
> *“ As soon as an Analytical Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise—By what course of calculation can these results be arrived at by the machine in the shortest time? ”* — *Charles Babbage (1864)*



Rare book containing the world’s first computer algorithm earns $125,000 at auction

By Matt Kennedy
July 25, 2018





Ada Lovelace’s algorithm to compute Bernoulli numbers on Analytic Engine (1843)

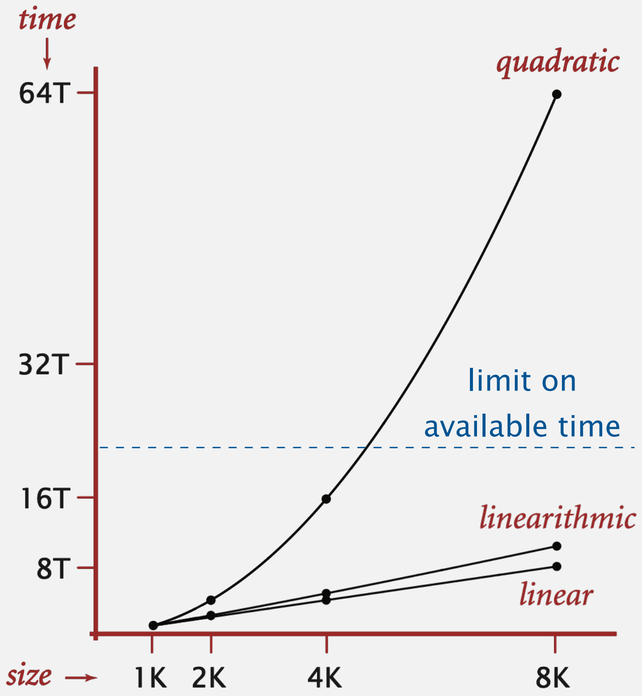
An algorithmic success story

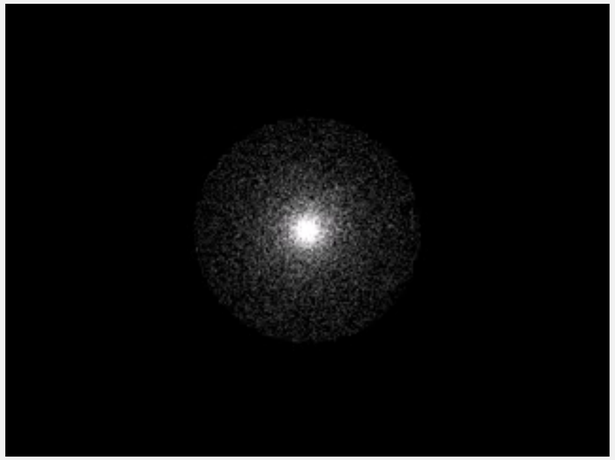
Optional Slide

N-body simulation.

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

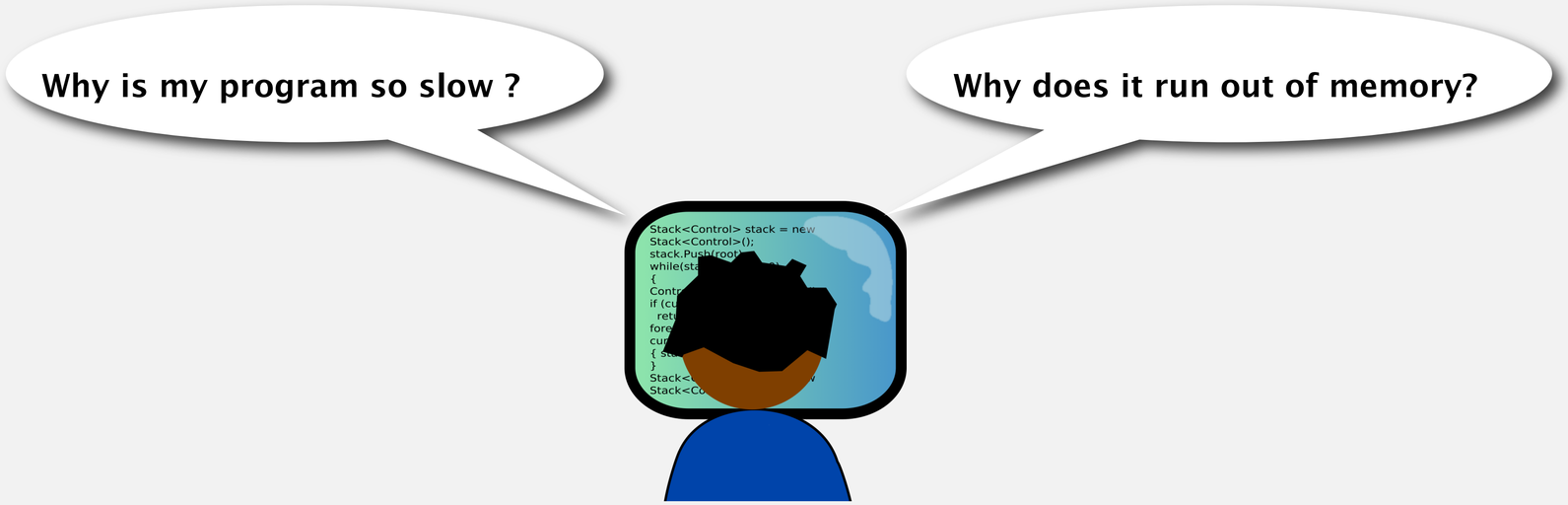
Andrew Appel
PU '81





The challenge

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



Our approach. Combination of experiments and mathematical modeling.

1. Greatest Hits of 111

- *Stack and heap*
- *Memory, objects, Arrays*
- *Images as 2D arrays*
- *Introduction to program analysis*
- ***Running time (experimental analysis)***
- *Running time (mathematical models)*
- *Memory usage*

Some slides Adopted and modified from Sedgewick and Wayne

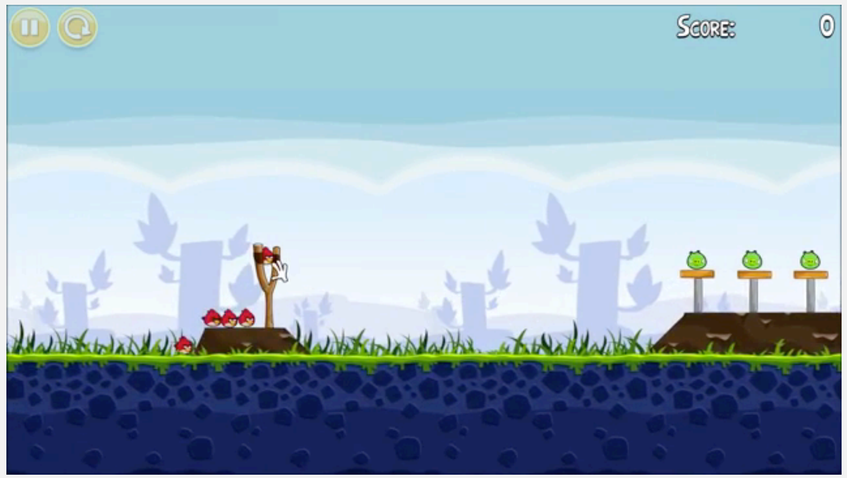
Example: 3-SUM

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

```
~/Desktop/3sum> more 8ints.txt
8
30 -40 -20 -10 40 0 10 5

~/Desktop/3sum> 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. 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)
   {
      In in = new In(args[0]);
      int[] a = in.readAllInts();
      StdOut.println(count(a));
   }
}
```

check distinct triples

for simplicity,

ignore integer overflow

Measuring the running time

Q. How to time a program?

A. Manual.

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

tick tick tick

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

tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick

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

tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick

```
4039
```

Measuring the running time

Q. How to time a program?

A. Automatic.

```
import edu.princeton.cs.algs4.StdOut;
import edu.princeton.cs.algs4.Stopwatch;

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

Empirical analysis

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

```
%
```

Empirical analysis

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

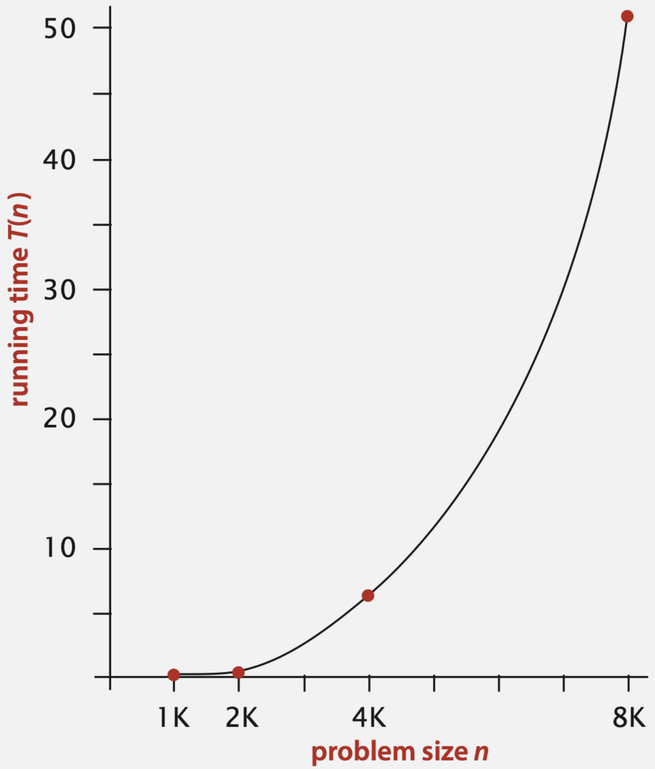
n	time (seconds) †
250	0
500	0
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1
16,000	?

How long will my program take, as a function of the input size?
To help answer this question let's plot the data.

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

Data analysis

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

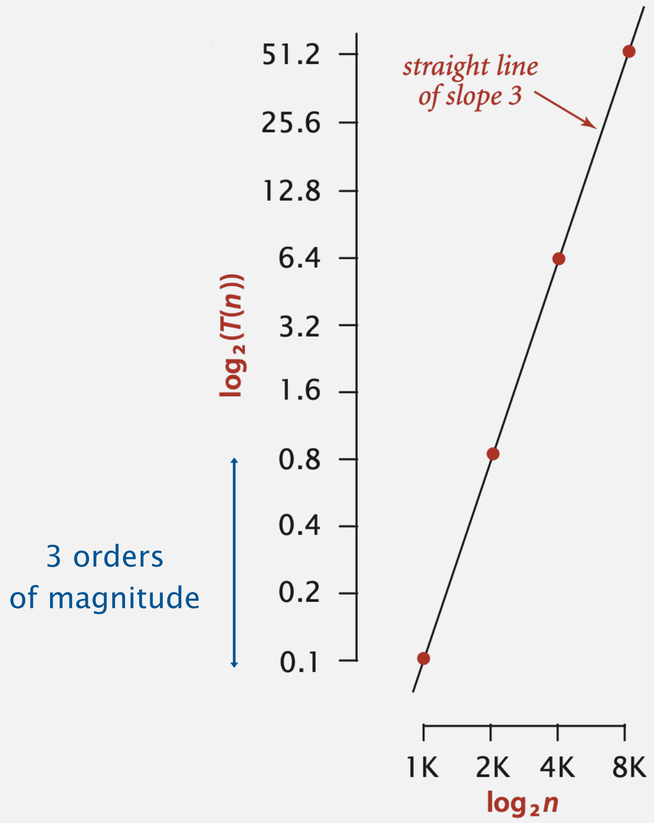


Hypothesis (power law). $T(n) = a\, n^b$

Questions. How to validate hypothesis? How to estimate a and b ?

Data analysis

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



slope

$$\log_2(T(n)) = 2.999 \log_2 n + (-33.21)$$

$$T(n) = 2^{-33.21} \times n^{2.999}$$
$$= 1.006 \times 10^{-10} \times n^{2.999}$$

Regression. Fit straight line through data points.

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

Data analysis – Logarithms (additional slide)

Before we move on to estimate *a* and *b* let's review the logarithm function.

The logarithm function is the inverse to the exponentiation function. That means that the logarithm of a number *x* is the exponent to which another number, the base *b*, must be raised to produce that number *x*.

$$\log_b x = y \; and \; b^y = x$$

In the simplest case, the logarithm counts the number of occurrences of the same factor in repeated multiplication. For example:

since $8 = 2 \times 2 \times 2 = 2^3$, the "logarithm base 2" of 8 is 3, or $\log_2(8) = 3$.

Logarithms will help estimating *a* and *b*.

Some properties of logarithms:

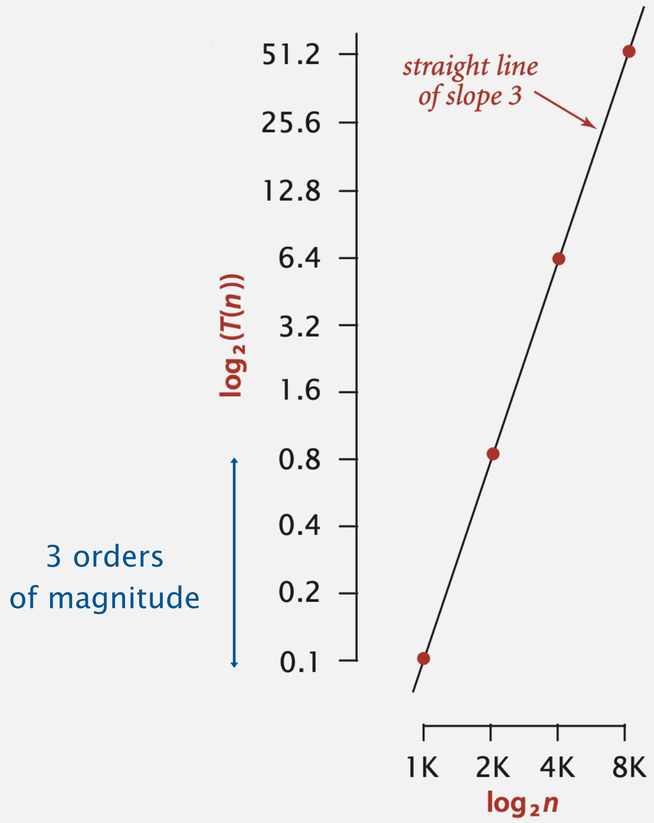
$$\log_b(xy) = \log_b x + \log_b y$$

$$\log_b(x^y) = y \log_b x$$

$$\log_a x = \log_b x \,/\, \log_b a$$

Data analysis (additional slide)

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



Taking the logarithm of the equation $T(n) = a\ n^b$ yields:

slope

$$\log_2 (T(n)) = b \log_2 n + \log_2 a$$

which is the equation of the straight line of slope 3 on the log-log plot

$\log_2 (T(n)) = 3 \log_2 n + \log_2 a$ (where a is a constant) is equivalent to:

$$T(n) = a\ n^3$$

The running time, as a function of the input size, as desired.

We can use one of our data points to solve for a. (slide 36)

Regression. Fit straight line through data points.

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

Prediction and validation, using regression

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	51.0
8,000	51.1
16,000	410.8

validates hypothesis!

Doubling hypothesis

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

Run program, doubling the size of the input.

n	time (seconds) †	ratio	$\log_2$ 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	3.0
8,000	51.1	8	3.0

$\log_2 (6.4 / 0.8) = 3.0$

seems to converge to a constant $b \approx 3$

$$\frac{T(n)}{T(n/2)} = \frac{an^b}{a(n/2)^b} = 2^b$$

$$\Longrightarrow \quad b = \log_2 \frac{T(n)}{T(n/2)}$$

Hypothesis. Running time is about $a\, n^b$ with $b = \log_2$ ratio.

Caveat. Cannot identify logarithmic factors with doubling hypothesis.

Doubling hypothesis

Doubling hypothesis. Quick way to estimate exponent 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	51.0
8,000	51.1

$$51.1 = a \times 8000^3$$

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

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

almost identical hypothesis
to one obtained via regression
(but less work)

Analysis of algorithms: quiz 1

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

A. 39 *seconds*

B. 52 *seconds*

C. 117 *seconds*

D. 350 *seconds*

n	time (seconds)
1,000	0.02
2,000	0.05
4,000	0.20
8,000	0.81
16,000	3.25
32,000	13.01

Analysis of algorithms: quiz 1

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

A. 39 *seconds*

B. 52 *seconds*

C. 117 *seconds*

D. 350 *seconds*

n	time (seconds)	ratio	$\log_2$ (ratio)
1,000	0.02	–	–
2,000	0.05	2.5	1.3
4,000	0.20	4.0	2.0
8,000	0.81	4.1	2.0
16,000	3.25	4.0	2.0
32,000	13.01	4.0	2.0

$T(n) = a\,n^2$ seconds

$T(32{,}000) = 13$ seconds

$\Rightarrow \quad a = 1.2695 * 10^{-8}$

$\Rightarrow \quad T(96{,}000) = 117$ seconds

$T(3n) = a\,(3n)^2$

$= 9\,a\,n^2$

$= 9\,T(n)$ seconds

$\Rightarrow \quad T(3 * 32{,}000) = 9 * 13 = 117$ seconds

Experimental algorithmics

System independent effects.

- Algorithm.
- Input data.

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

System dependent effects.

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

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

Bad news. Sometimes difficult to get accurate measurements.

Context: the scientific method

Experimental algorithmics is an example of the scientific method.

Chemistry
(1 experiment)

Biology
(1 experiment)

Physics
(1 experiment)

Computer Science
(1 million experiments)

Good news. Experiments are easier and cheaper than other sciences.

1. Greatest Hits of 111

- *Memory, objects, Arrays*
- *Program stack and heap*
- *Images as 2D arrays*
- *Introduction to program analysis*
- *Running time (experimental analysis)*
- *Running time (mathematical models)*
- *Memory usage*

Some slides Adopted and modified from Sedgewick and Wayne

Mathematical models for running time

LO 1.3

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.

The New York Times

Donald Knuth, master of algorithms, reflects on 50 years of his opus-in-progress, "The Art of Computer Programming."

Warning. No general-purpose method (e.g., halting problem).

Example: 1-SUM

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

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

exactly n array accesses

operation	cost (ns) †	frequency
variable declaration	2/5	2
assignment statement	1/5	2
less than compare	1/5	$n + 1$
equal to compare	1/10	n
array access	1/10	n
increment	1/10	n to $2n$

in practice, depends on caching, bounds checking, … (see COS 217)

† representative estimates (with some poetic license)

Analysis of algorithms: quiz 2

How many array accesses as a function of n?

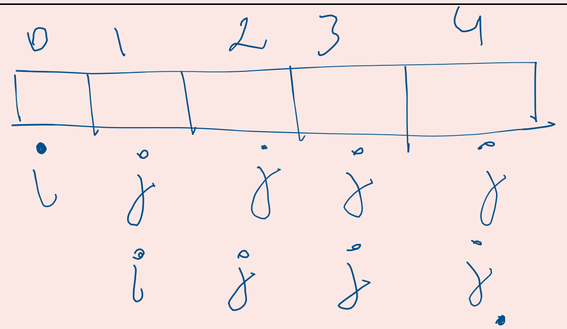
```
int count = 0;
for (int i = 0; i < n; i++)
   for (int j = i+1; j < n; j++)
      if (a[i] + a[j] == 0)
         count++;
```

A. $\frac{1}{2}\, n\,(n-1)$

B. $n\,(n-1)$

C. $2\, n^2$

D. $2\, n\,(n-1)$



$n-1 + n-2 + n-3 + \dots + 0$

$0 + 1 + 2 + 3 + \dots + n-1$

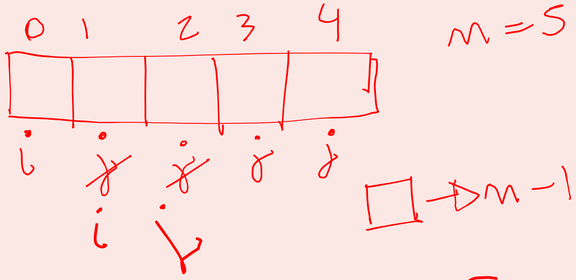
$(n-1)\left(\frac{1+n-1}{2}\right) = \frac{n^2-n}{2}$

$= n^2 - n$

Analysis of algorithms: quiz 2

How many array accesses as a function of 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++;
```



$$(n-1) + (n-2) + \ldots + 1 + 0$$

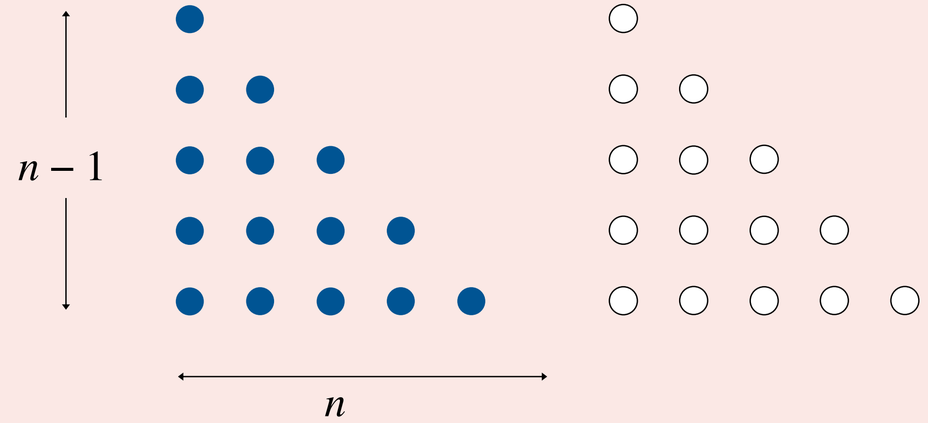
$$\tfrac{1}{2}\, n\,(n-1) \quad *2$$

A. $\tfrac{1}{2}\, n\,(n-1)$

B. $n\,(n-1)$

C. $2\, n^2$

D. $2\, n\,(n-1)$



$$2 * (1 + 2 + 3 + \ldots + n - 1) = n\,(n-1)$$

$$\Longrightarrow (1 + 2 + 3 + \ldots + n - 1) = \tfrac{1}{2}\, n\,(n-1)$$

$1 + 2 + 3 + \ldots + n-1$

$(n-1)\left(\frac{n-1+1}{2}\right) \quad \frac{n}{2}\,(n-1)$

Example: 2-SUM

Q. How many operations 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	cost (ns)	frequency
variable declaration	2/5	$n + 2$
assignment statement	1/5	$n + 2$
less than compare	1/5	$\frac{1}{2} (n + 1) (n + 2)$
equal to compare	1/10	$\frac{1}{2} n (n - 1)$
array access	1/10	$n (n - 1)$
increment	1/10	$\frac{1}{2} n (n + 1)$ to n^2

$1/4\ n^2 + 13/20\ n + 13/10$ ns

to

$3/10\ n^2 + 3/5\ n + 13/10$ ns

(tedious to count exactly)

Simplifying the calculations

Optional Slide

> *" 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

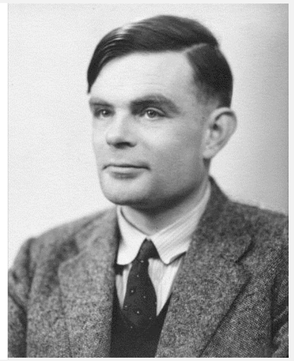
By A. M. TURING

(*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 elementary operation as a proxy for running time.

- Use the operation that cost and frequency are the highest

```
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++;
```

operation	cost (ns)	frequency
variable declaration	2/5	$n + 2$
assignment statement	1/5	$n + 2$
less than compare	1/5	$\frac{1}{2}(n + 1)(n + 2)$
equal to compare	1/10	$\frac{1}{2} n (n - 1)$
array access	1/10	$n (n - 1)$
increment	1/10	$\frac{1}{2} n (n + 1)$ to n^2

cost model = array accesses

(we're assuming compiler/JVM does not optimize any array accesses away!)

Simplification 2: asymptotic notations

LO 1.4

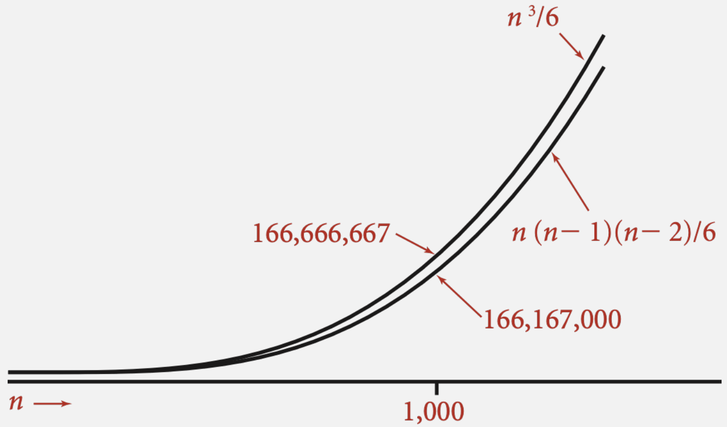
Tilde notation. Discard lower-order terms.

Big Theta notation. Also discard leading coefficient.

formal definitions involve limits

function	tilde	big Theta
$4n^5 + 20n + 16$	$\sim 4n^5$	$\Theta(n^5)$
$7n^2 + 100n^{4/3} + 56$	$\sim 7n^2$	$\Theta(n^2)$
$\frac{1}{6}n^3 - \frac{1}{2}n^2 + \frac{1}{3}n$	$\sim \frac{1}{6}n^3$	$\Theta(n^3)$

discard lower-order terms
(e.g., n = 1,000: 166.67 million vs. 166.17 million)



Leading-term approximation

Rationale.

- When n is large, lower-order terms are negligible.
- When n is small, we don't care.

Simplification 2: tilde and big Theta notations

Tilde notation. Discard lower-order terms.

Big Theta notation. Also discard leading coefficient.

Use tilde notation to:

- Count core operations (e.g., compares or array accesses).
- Make predictions about running time (in seconds).
- Measure memory usage (in bytes).

$2n$ $4n$

Use big Theta notation to:

- Analyze performance independent of machine, compiler, JVM, ...
- Understand why an algorithm doesn't scale.

$\Theta(n)$

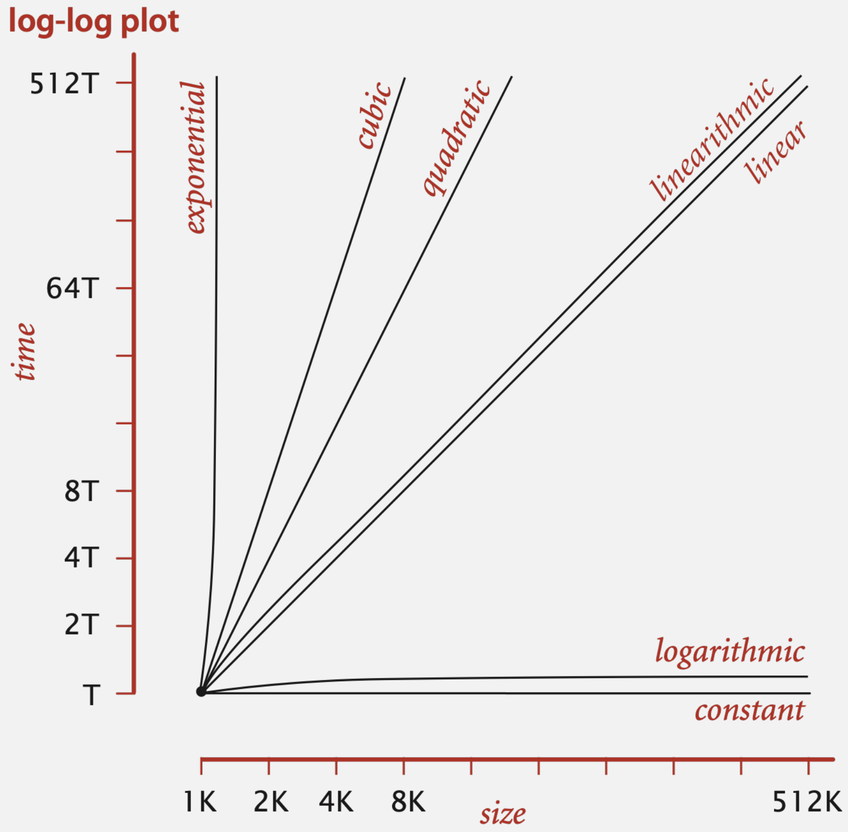
Common order-of-growth classifications

LO 1.4

Good news. The set of functions

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

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



Typical orders of growth

Commonly-used notations in the theory of algorithms

notation	provides	example	shorthand for	used to
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$ …	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 $N^3 + 22 N \log N + N$	develop lower bounds

Common order-of-growth classifications

LO 1.4

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

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"

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

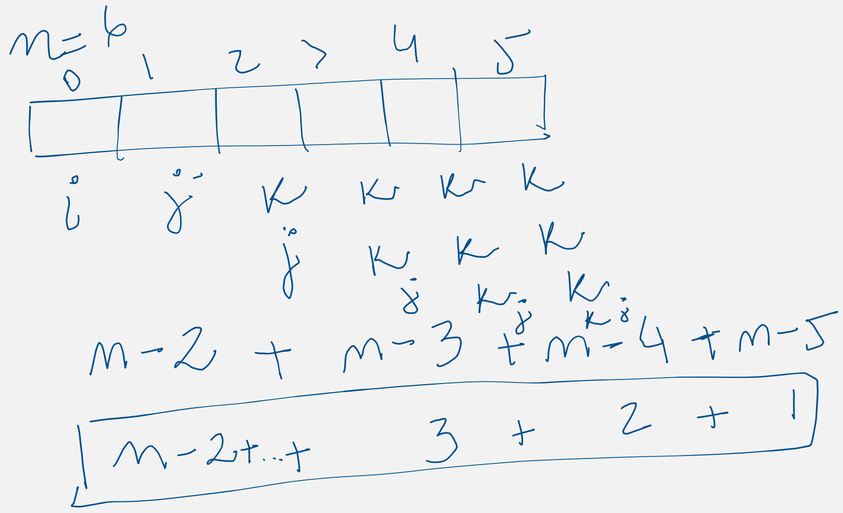
A. $\sim n^2$ array accesses.

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

$$\binom{n}{3} = \frac{n(n-1)(n-2)}{3!}$$

$$\sim \frac{1}{6} n^3 * 3$$

$n = 5$

$10 = \frac{5(4)(3)}{3*2}$

$\binom{5}{3}$

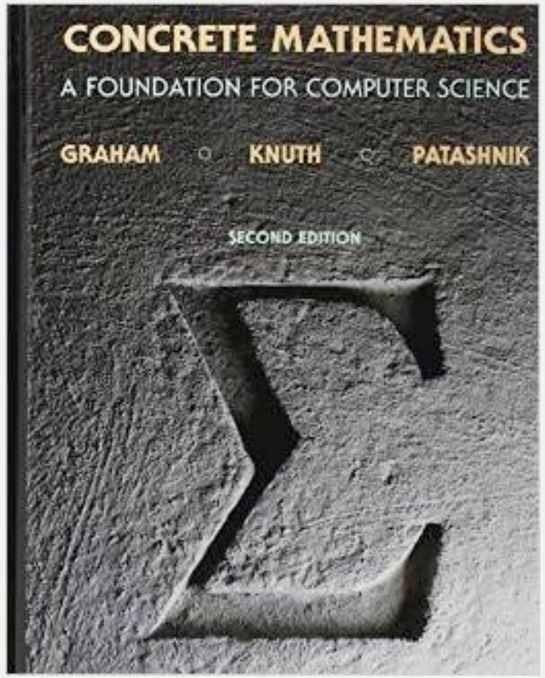
Pear apple banana kiwi orange

Bottom line. Use cost model and asymptotic notation to simplify analysis.

Estimating a discrete sum

Q. How to estimate a discrete sum?

A1. Take a discrete mathematics course (CS 205).



Estimating a discrete sum

Optional Slide

Q. How to estimate a discrete sum?

A2. Replace the sum with an integral; 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. $1 + 1/2 + 1/3 + \ldots + 1/n.$

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

Ex 3. 3-sum triple loop.

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

Ex 4. $1 + ½ + ¼ + ⅛ + \ldots$

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

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

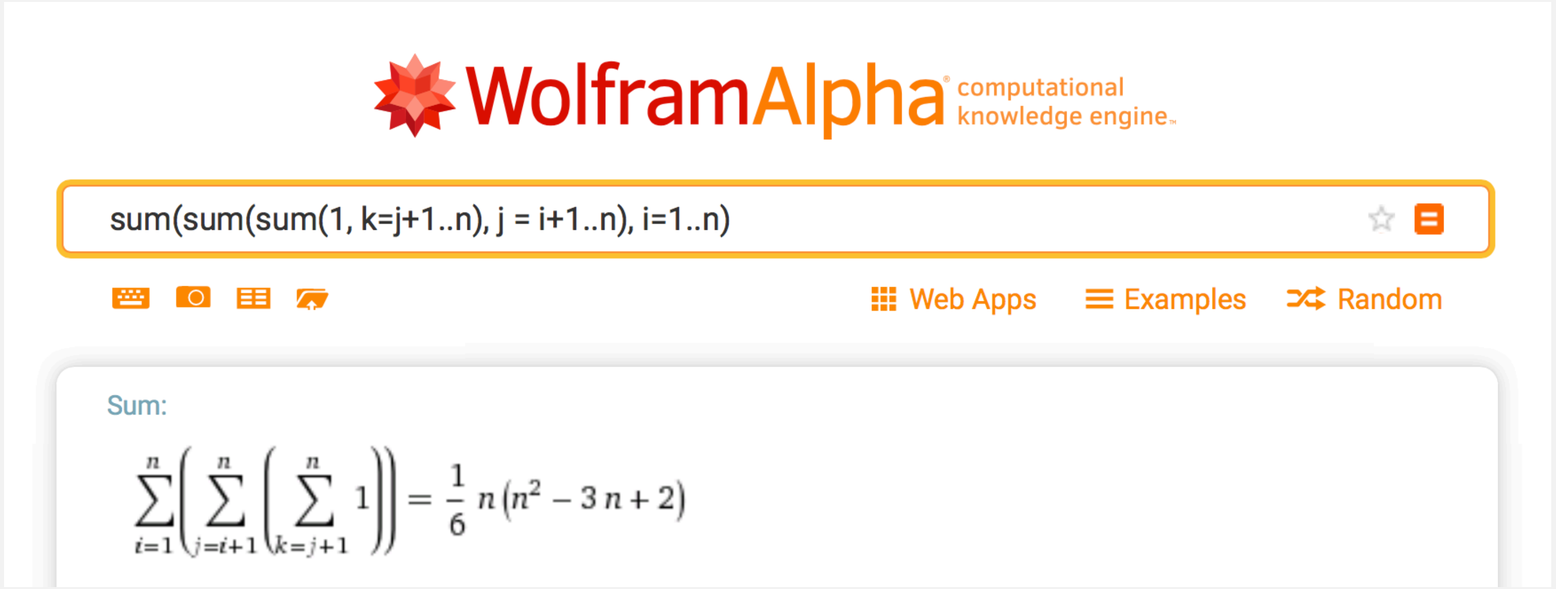
integral trick doesn't always work!

Estimating a discrete sum

Optional Slide

Q. How to estimate a discrete sum?

A3. Use Maple or Wolfram Alpha.



https://www.wolframalpha.com

Analysis of algorithms: quiz 3

How many array accesses as a function of n ?

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

$\frac{1}{2} n(n-1)$

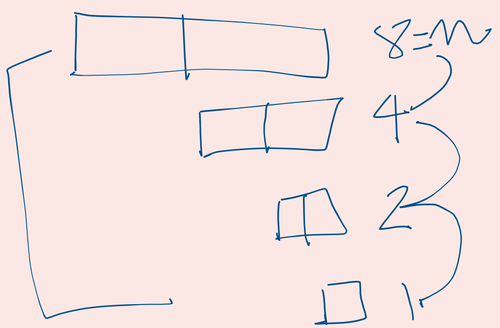
runs $\log n$ times

$T(n) = \frac{1}{2} n(n-1) * 3 \log n$

$8 = 2^3$

$n = 2$

$k = \log n$



A. $\sim n^2 \log_2 n$

B. $\sim 3/2 \; n^2 \log_2 n$

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

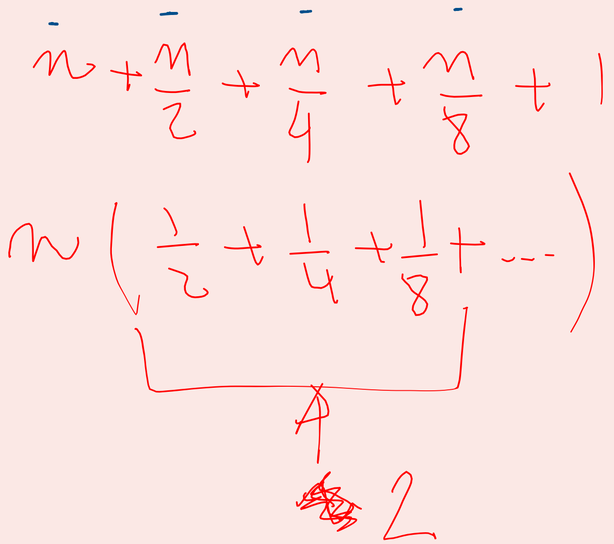
D. $\sim 3/2 \; n^3$

Analysis of algorithms: quiz 4

What is order of growth of running time as a function of n?

```
int count = 0;
for (int i = n; i >= 1; i = i/2)
   for (int j = 1; j <= i; j++)
      count++;
```

"inner loop"



A. $\Theta(n)$

B. $\Theta(n \log n)$

C. $\Theta(n^2)$

D. $\Theta(2^n)$

1. Greatest Hits of 111

- *Memory, objects, Arrays*
- *Program stack and heap*
- *Images as 2D arrays*
- *Introduction to program analysis*
- *Running time (experimental analysis)*
- *Running time (mathematical models)*
- *Memory usage*

Some slides Adopted and modified from Sedgewick and Wayne

Basics

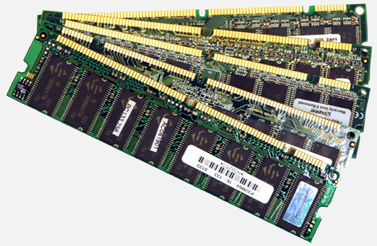
Bit. 0 or 1.

Byte. 8 bits.

NIST most computer scientists

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.

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

Typical memory usage for primitive types and arrays

LO 1.3

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

primitive types

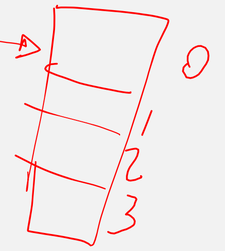
type	bytes
boolean[]	$1n + 24$
int[]	$4n + 24$
double[]	$8n + 24$

one-dimensional arrays (length n)

wasteful

(but ~ $36n$ in Python 3)

array overhead = 24 byte



type	bytes
boolean[][]	$\sim 1\ n^2$
int[][]	$\sim 4\ n^2$
double[][]	$\sim 8\ n^2$

two-dimensional arrays (n-by-n)

$4 \times 4 + 24$

$1n^2 + 24 \Rightarrow \sim 1n^2$

Typical memory usage for objects in Java

LO 1.3

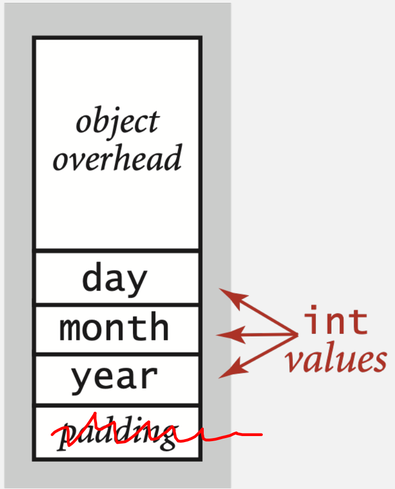
Object overhead. 16 bytes.

Reference. 8 bytes.

Padding. Memory of each object rounded up to use a multiple of 8 bytes.

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

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



16 bytes (object overhead)

4 bytes (int)

4 bytes (int)

4 bytes (int)

4 bytes (padding)

32 bytes

Typical memory usage summary

LO 1.3

Total memory usage for a data type value in Java:

- 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.

Note. Depending on application, we often count the memory for any referenced objects (recursively).

"deep memory"

Analysis of algorithms: quiz 5

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

A. $\sim 4\,n$ *bytes*

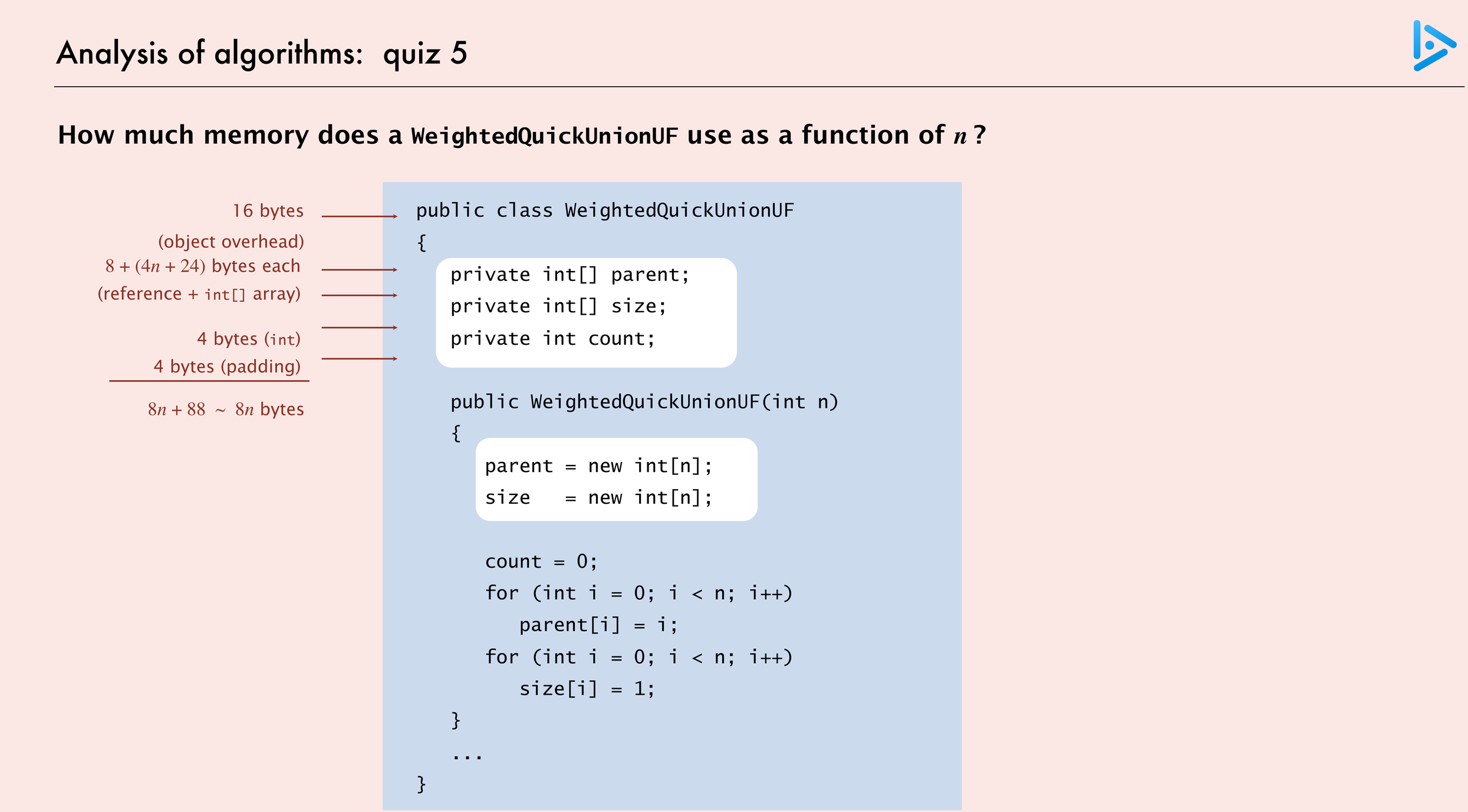
B. $\sim 8\,n$ *bytes*

C. $\sim 4\,n^2$ *bytes*

D. $\sim 8\,n^2$ *bytes*

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

   public WeightedQuickUnionUF(int n)
   {
      parent = new int[n];
      size   = new int[n];

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

Analysis of algorithms: quiz 5



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

16 bytes (object overhead)

$8 + (4n + 24)$ bytes each (reference + `int[]` array)

4 bytes (`int`)

4 bytes (padding)

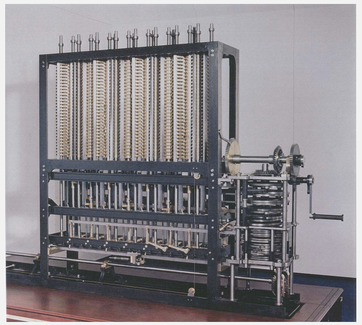
$8n + 88 \sim 8n$ bytes

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

   public WeightedQuickUnionUF(int n)
   {
      parent = new int[n];
      size   = new int[n];

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

Turning the crank: summary

Empirical analysis.

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



Mathematical analysis.

- Analyze algorithm to count frequency of operations.
- Use tilde and big-Theta notations to simplify analysis.
- Model enables us to explain behavior.

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

This course. Learn to use both.

INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

Rutgers University

1. Greatest Hits of 111

- Memory, objects, Arrays
- Program stack and heap
- Images as 2D arrays
- Introduction to program analysis
- Running time (experimental analysis)
- Running time (mathematical models)
- Memory usage

https://ds.cs.rutgers.edu

Last updated on 1/8/21