ANALYSIS OF ALGORITHMS

CS210 – Data Structures and Algorithms

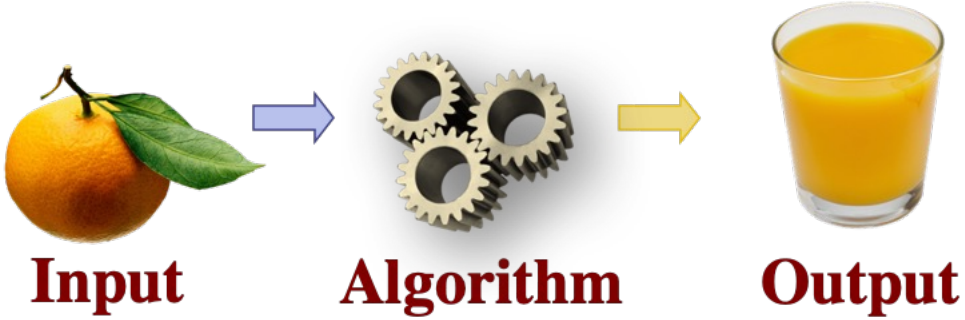
Dr. Basit Qureshi

جامعة الأمير سلطان
PRINCE SULTAN
UNIVERSITY

https://www.drbasit.org/

TOPICS

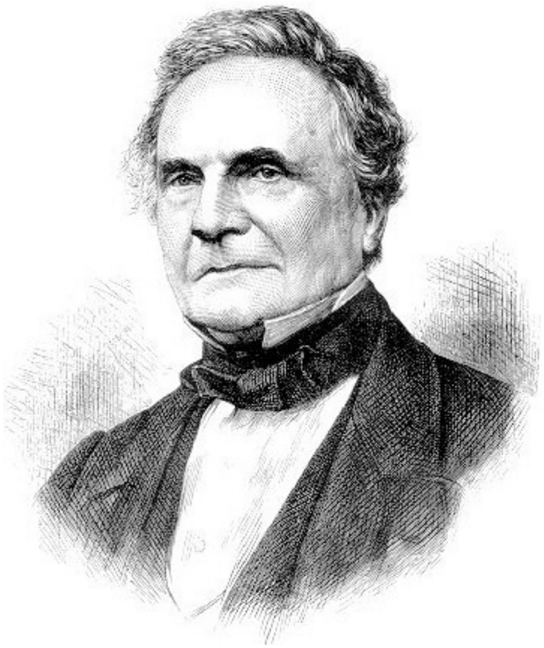
- Running Time
- Experimental Studies & challenges
- Why Algorithm Analysis?
- Estimating Runtime
- Growth functions and Asymptotic Analysis
- Comparing Algorithms
- Big Oh notation
- Analysis of Recursive Algorithms

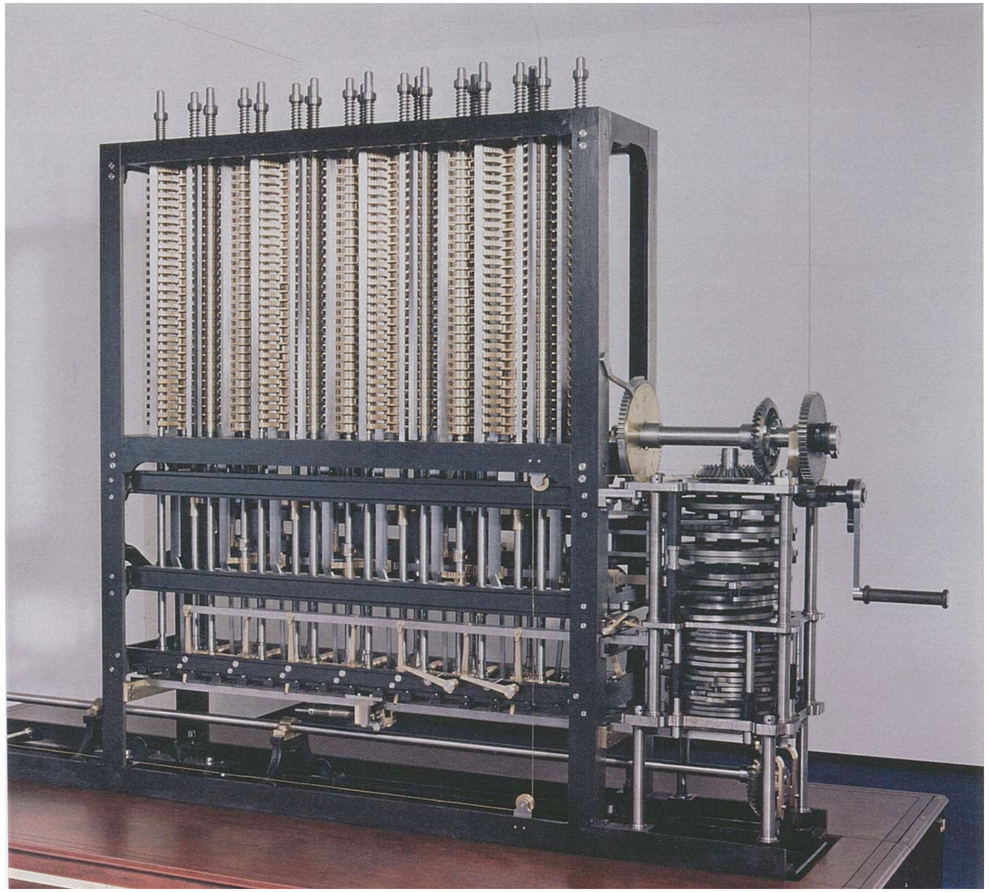


RUNNING TIME

- How to time a program?
    - Babbage Analytical Engine

" 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)*





RUNNING TIME

- How to time a program?
    - Use stopwatch!

```
% java ThreeSum 1Kints.txt
70
% java ThreeSum 2Kints.txt
528
% java ThreeSum 4Kints.txt
4039
```

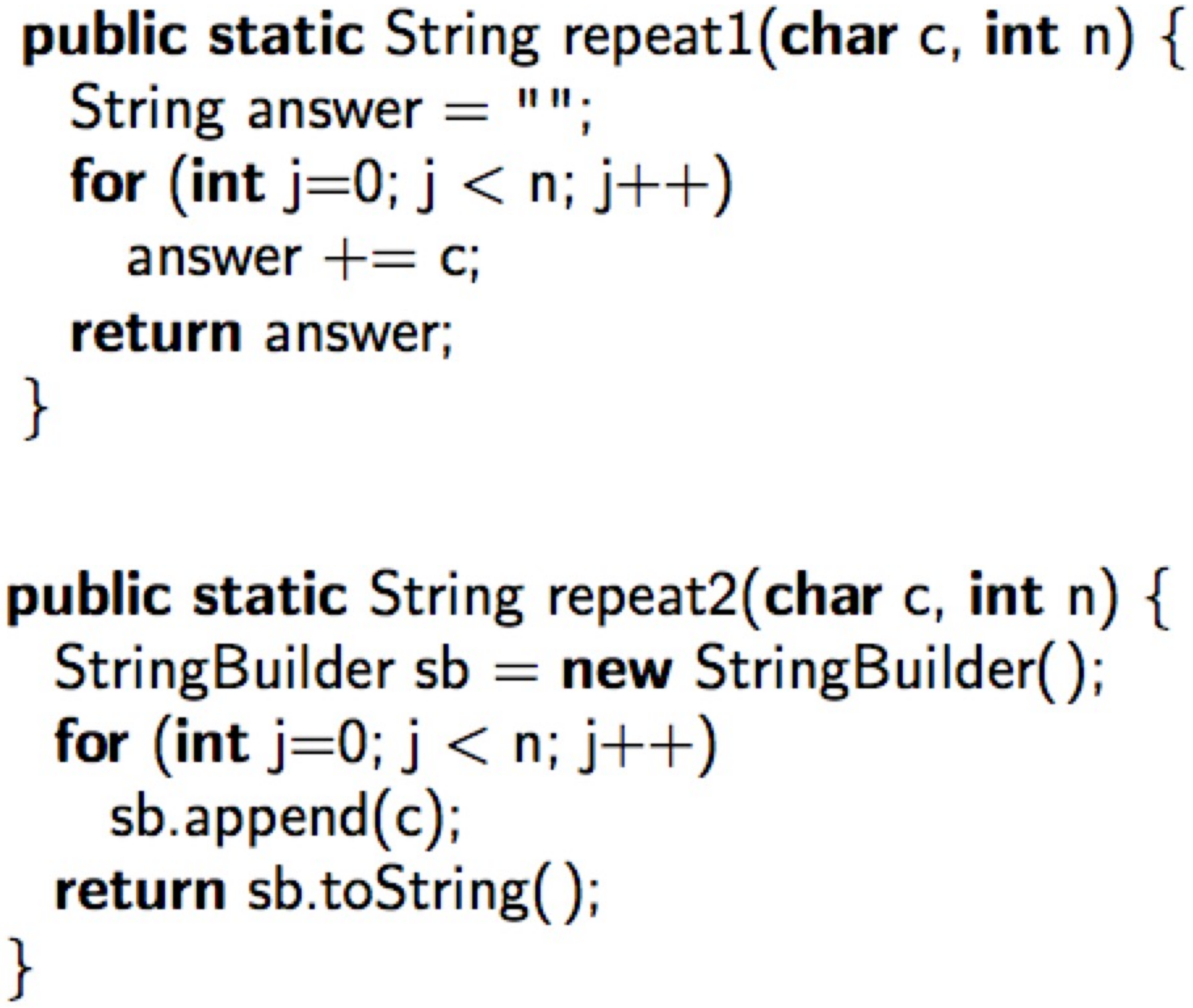
RUNNING TIME

- How to time a program?
  - Use Code?

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


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

RUNNING TIME

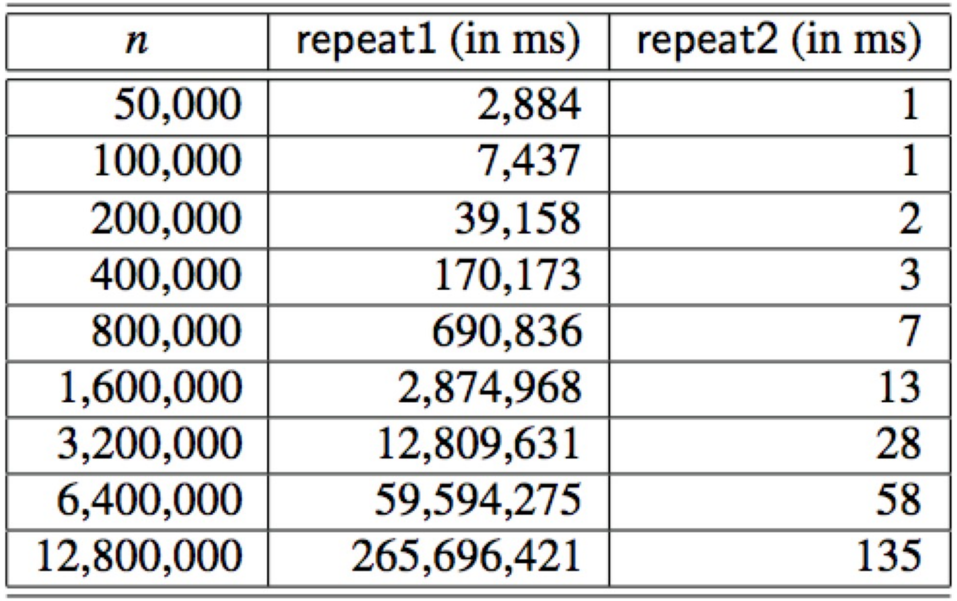
- Comparing time



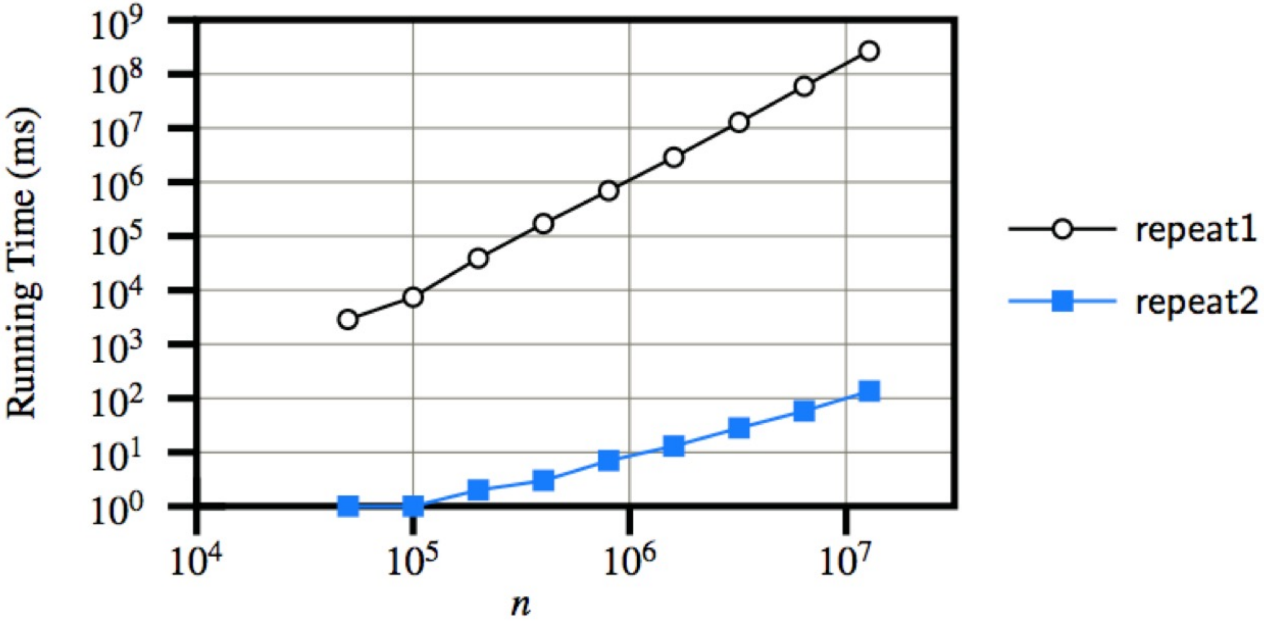
```
public static String repeat1(char c, int n) {
  String answer = "";
  for (int j=0; j < n; j++)
    answer += c;
  return answer;
}

public static String repeat2(char c, int n) {
  StringBuilder sb = new StringBuilder( );
  for (int j=0; j < n; j++)
    sb.append(c);
  return sb.toString( );
}
```

RUNNING TIME

- Comparing time

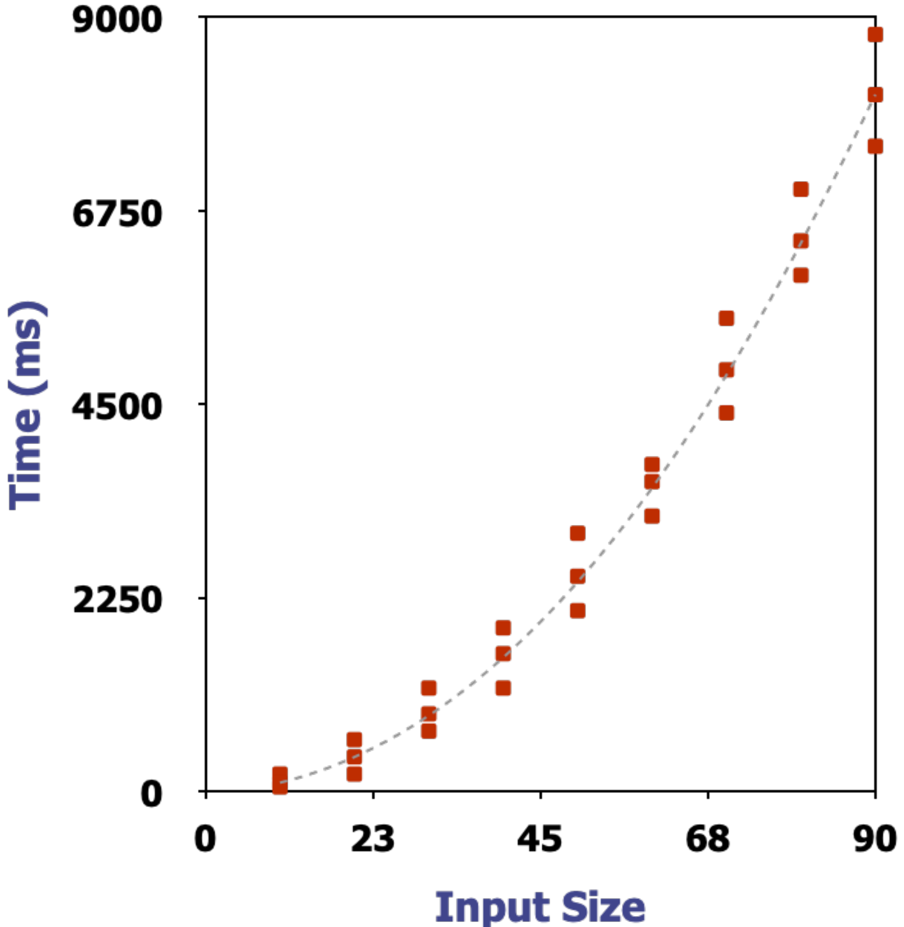


n	repeat1 (in ms)	repeat2 (in ms)
50,000	2,884	1
100,000	7,437	1
200,000	39,158	2
400,000	170,173	3
800,000	690,836	7
1,600,000	2,874,968	13
3,200,000	12,809,631	28
6,400,000	59,594,275	58
12,800,000	265,696,421	135



EXPERIMENTAL STUDIES & CHALLENGES

- Experimental study: How to?
- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition, noting the time needed
- Plot the results



EXPERIMENTAL STUDIES & CHALLENGES

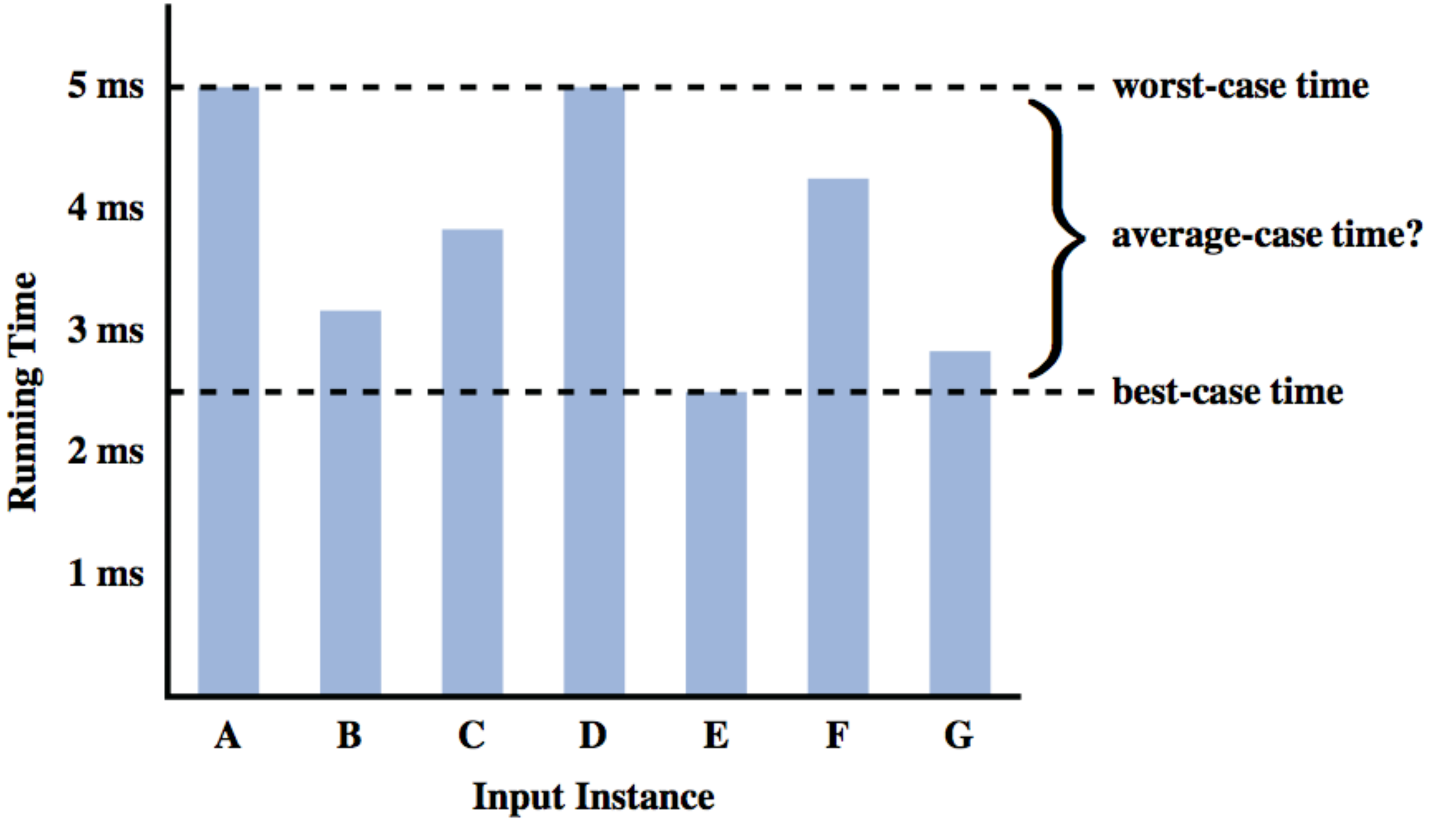
- Experimental study Challenges
- Experimental running times of two algorithms are difficult to directly compare **unless the experiments are performed in the same hardware and software environments.**
- Experiments can be done only on a **limited set of test inputs**; hence, they leave out the running times of inputs not included in the experiment (and these inputs may be important).
- An algorithm **must be fully implemented** in order to execute it to study its running time experimentally.

WHY ALGORITHM ANALYSIS

- Algorithm Analysis
- Allows us to ***evaluate the relative efficiency*** of any two algorithms in a way that is **independent of the hardware and software environment.**
- Is performed by studying ***a high-level description*** of the algorithm without need for implementation.
- Takes into account ***all possible inputs.***

WHY ALGORITHM ANALYSIS

- Understanding Run-times
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the **worst case** running time.
- Easier to analyze
- Crucial to applications such as games, finance and robotics



WHY ALGORITHM ANALYSIS

- Estimating Run-time
- Estimate the **primitive operations** : "Basic computations performed by an algorithm”
- Identifiable in pseudocode
- Largely **independent** from the programming language
- Assumed to take a constant amount of time in the RAM model

- Examples:
    - Evaluating an expression
    - Assigning a value to a variable
    - Indexing into an array
    - Calling a method
    - Returning from a method

WHY ALGORITHM ANALYSIS

Observation. **Most primitive operations take constant time**

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
...	...	...

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]	$c_7 N^2$

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

WHY ALGORITHM ANALYSIS

- Counting Primitive Operations
- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

```
                                                        1 operation
/** Returns the maximum value of a nonempty array of numbers. */
public static double arrayMax(double[ ] data) {
  int n = data.length;             2 operations; define int n; assign n a value
  double currentMax = data[0];     2 ops        // assume first entry is biggest (for now)
  for (int j=1; j < n; j++)        1 + n + n    // consider all other entries
    if (data[j] > currentMax)      2 * n        // if data[j] is biggest thus far...
      currentMax = data[j];        0 or 2 * n   // record it as the current max
  return currentMax;               1
}
```

Best case: 4n + 7 operations

Worst case: 6n + 7 operations

4n + 7 <= T(n) <= 6n + 7 operations

WHY ALGORITHM ANALYSIS

Estimating Running Time

Algorithm arrayMax executes **$5n + 5$** primitive operations in the worst case, **$4n + 5$** in the best case. Define:

Let ***a*** = Time taken by the fastest primitive operation

Let ***b*** = Time taken by the slowest primitive operation

Let ***T(n)*** be worst-case time of arrayMax.

Then

$$a\,(4n + 5) \le T(n) \le b\,(5n + 5)$$

Hence, the running time *T(n)* is bounded by two linear functions

GROWTH RATE OF RUNNING TIME

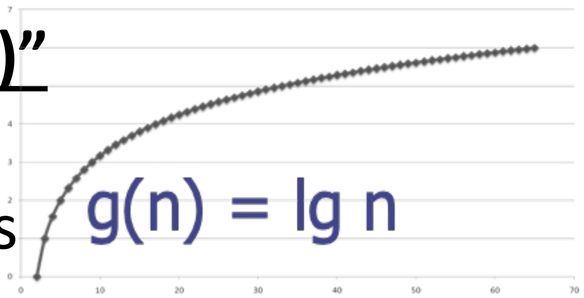
Growth rate.

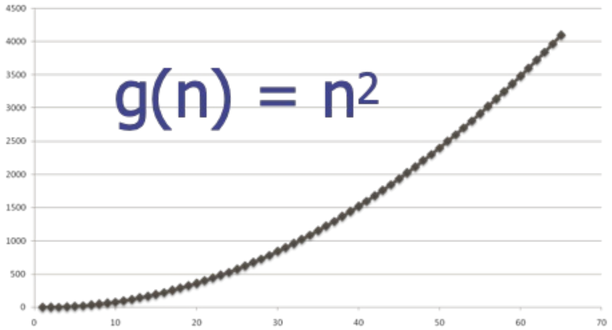
Changing the hardware/ software environment affects T(n) by a constant factor, but

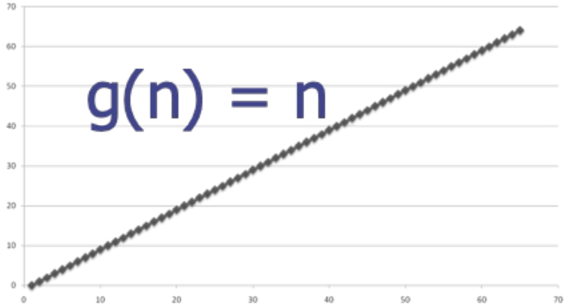
<u>“Does not alter the growth rate of T(n)”</u>

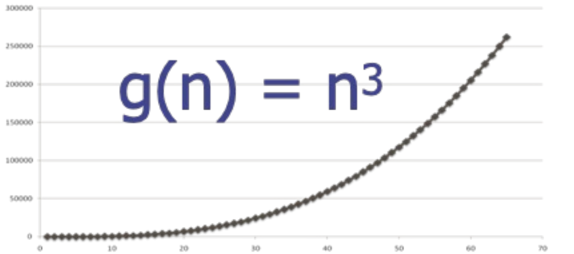
We consider Seven important functions

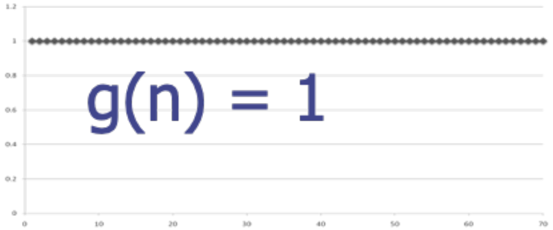
- Constant ≈ 1
- Logarithmic ≈ log n
- Linear ≈ n
- N-Log-N ≈ n log n
- Quadratic ≈ n^2
- Cubic ≈ n^3
- Exponential ≈ 2^n

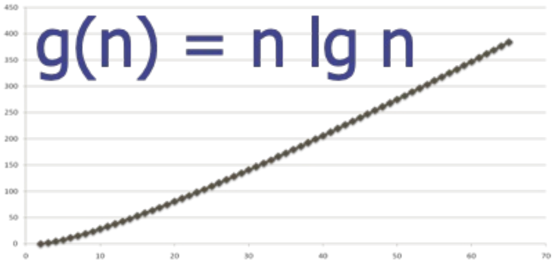


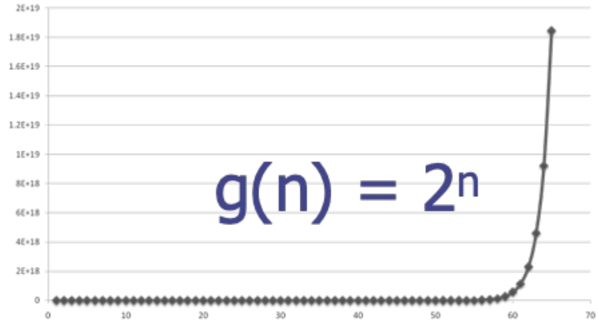












GROWTH RATE OF RUNNING TIME

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**	$T(N)$

GROWTH RATE OF RUNNING TIME

Growth rate time-perception

growth rate	time to process millions of inputs			
	1970s	1980s	1990s	2000s
1	instant	instant	instant	instant
log N	instant	instant	instant	instant
N	minutes	seconds	second	instant
N log N	hour	minutes	tens of seconds	seconds
N^2	decades	years	months	weeks
N^3	never	never	never	millennia

COMPARISON OF ALGORITHMS

Comparing two algorithms

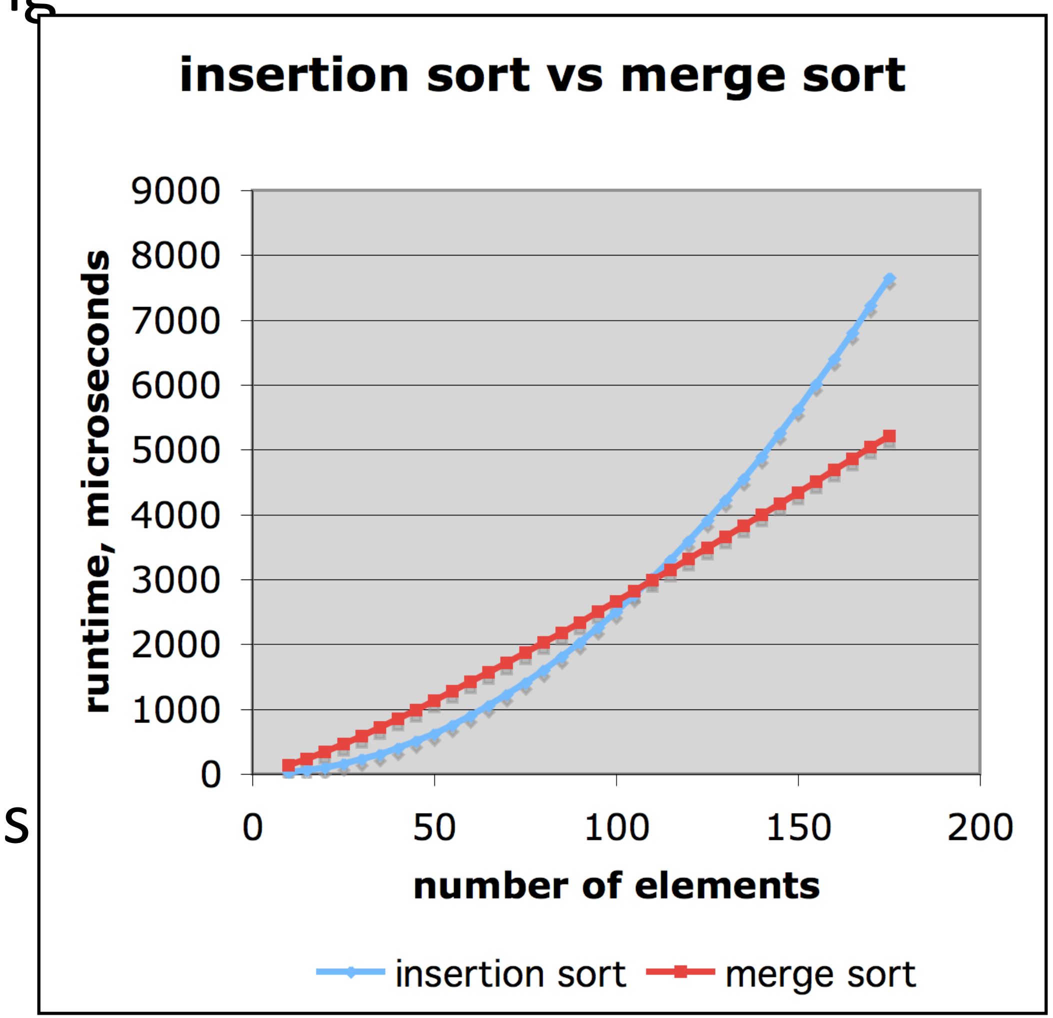
We give the runtime for two popular sorting algorithms as:

- insertion sort is $n^2 / 4$
- merge sort is $2\ n \lg n$

For a large dataset (1 million items), how long would it take to sort the data

- insertion sort takes roughly 70 hours
- merge sort takes roughly 40 seconds

For a faster machine it could be 40 minutes versus less than 0.5 seconds



COMPARISON OF ALGORITHMS

Affect of constant factors

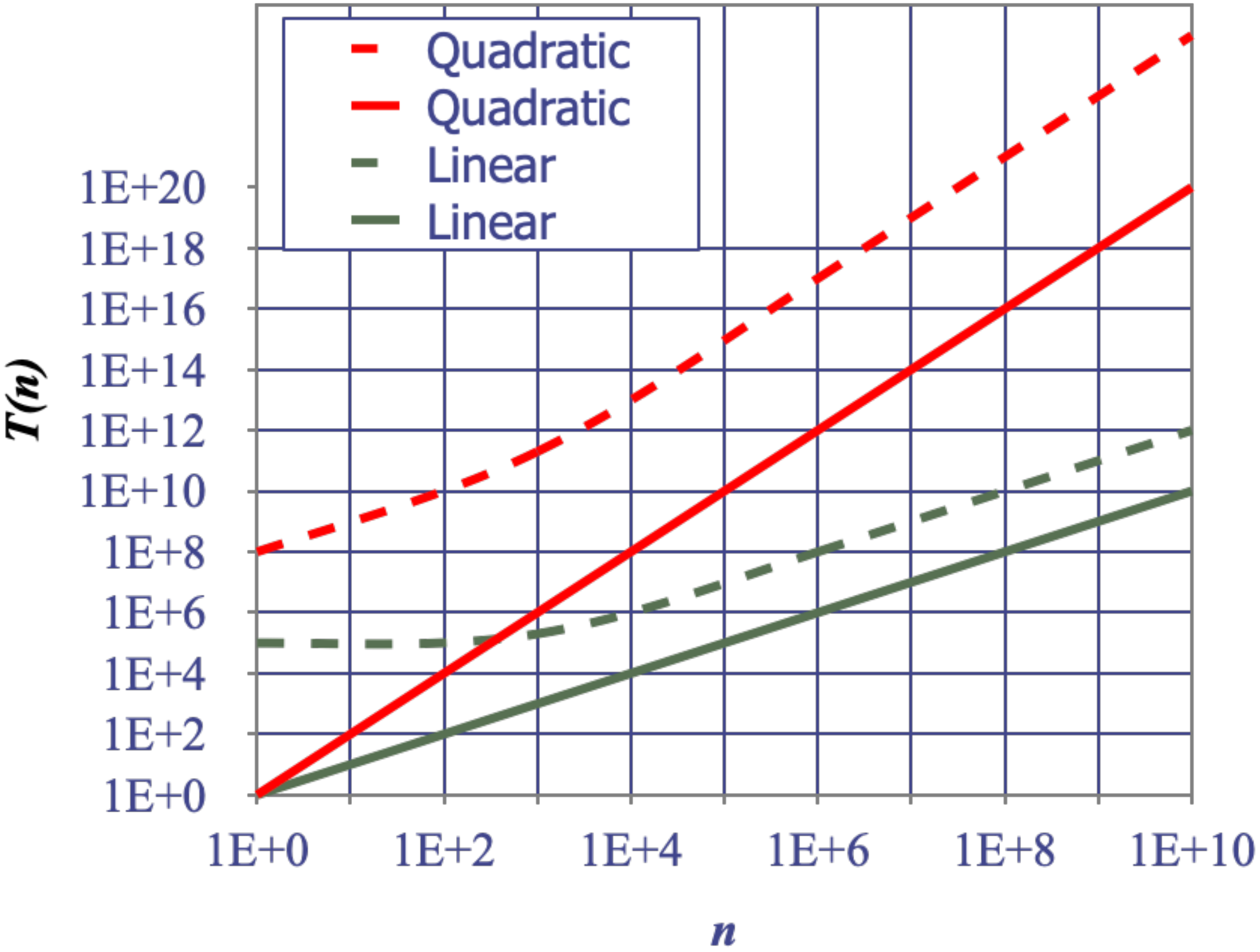
The growth rate is not affected by constant factors or

lower-order terms

Examples

10^2 **n** + 10^5 is a **linear function**

10^5 **n²** + 10^8 **n** is a **quadratic function**



BIG-OH NOTATION

The Big Oh notation

Given functions *f(n)* and *g(n),* we say that *f(n)* is *O(g(n))* if there are positive constants $\boldsymbol{c}$ and $\boldsymbol{n_0}$ such that

$$f(n) \le cg(n) \quad \text{for } n \ge n0$$

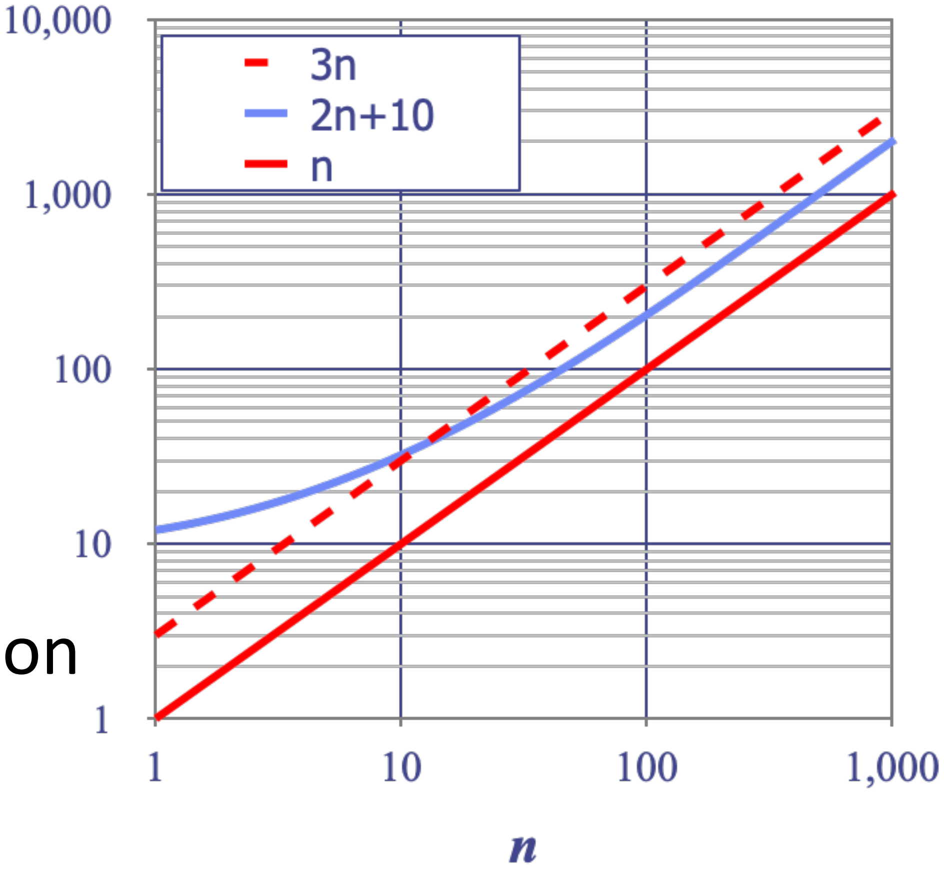
Example: Prove that $2n + 10$ is $O(n)$

$2n + 10 \le cn$

$(c - 2)\, n \ge 10$

$n \ge 10/(c - 2)$

Pick $c = 3$ and $n_0 = 10$ to satisfy the equation



BIG-OH NOTATION

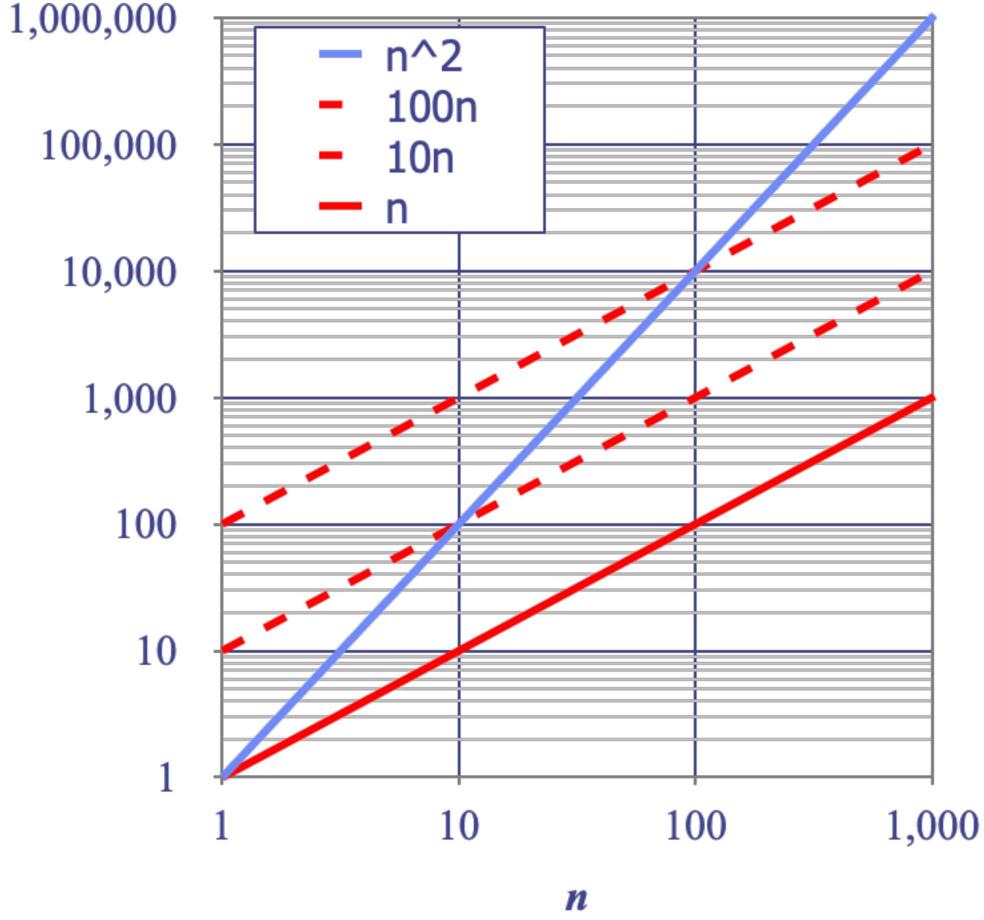
The Big Oh notation example

Example: Prove that n^2 is not $O(n)$

$n^2 \leq cn$

$n \leq c$

The above inequality cannot be satisfied since c must be a constant



BIG-OH NOTATION

The Big Oh notation example

Example: Prove that **$7n - 2$** is $O(n)$

$7 n - 2 \leq c n$

need $c > 0$ and $n_0 \geq 1$ such that for $n \geq n_0$

this is true for $c = 7$ and $n0 = 1$

BIG-OH NOTATION

The Big Oh notation example

Example: Prove that **$3 n^3 + 20 n^2 + 5$** is $O(n^3)$

$3 n^3 + 20 n^2 + 5 \le c n^3$ for $n \ge n_0$

need $c > 0$ and $n0 \ge 1$ such that

this is true for $c = 4$ and $n_0 = 21$

BIG-OH NOTATION

The Big Oh notation example

Example: Prove that **$3 \log n + 5$** is $O(\log n)$

$3 \log n + 5 \leq c \log n$

need $c > 0$ and $n_0 \geq 1$ such that for $n \geq n_0$

this is true for $c = 8$ and $n_0 = 2$

BIG-OH NOTATION

Big-Oh and Growth Rate

The big-Oh notation gives an upper bound on the growth rate of a function

The statement "f(n) is O(g(n))" means that the growth rate of f(n) is ***no more than the growth rate*** of g(n)

We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

BIG-OH NOTATION

Big-Oh rules

If is *f(n)* a polynomial of degree *d*, then *f(n)* is $O(n^d)$, i.e.,

- Drop lower-order terms
- Drop constant factors

Use the smallest possible class of functions

Say “2n is O(n)” instead of “2n is $O(n^2)$”

Use the simplest expression of the class

Say “3n + 5 is O(n)” instead of “3n + 5 is O(3 n)”

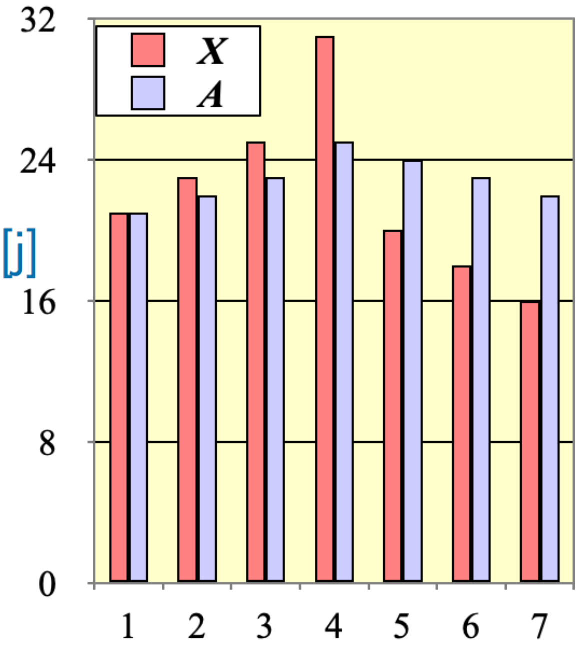
ASYMPTOTIC ALGORITHM ANALYSIS

- Asymptotic Analysis
- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the **worst-case** number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We say that algorithm arrayMax “runs in O(n) time”
- Since constant factors and lower-order terms are eventually dropped any how, we can disregard them when counting primitive operations

ASYMPTOTIC ALGORITHM ANALYSIS

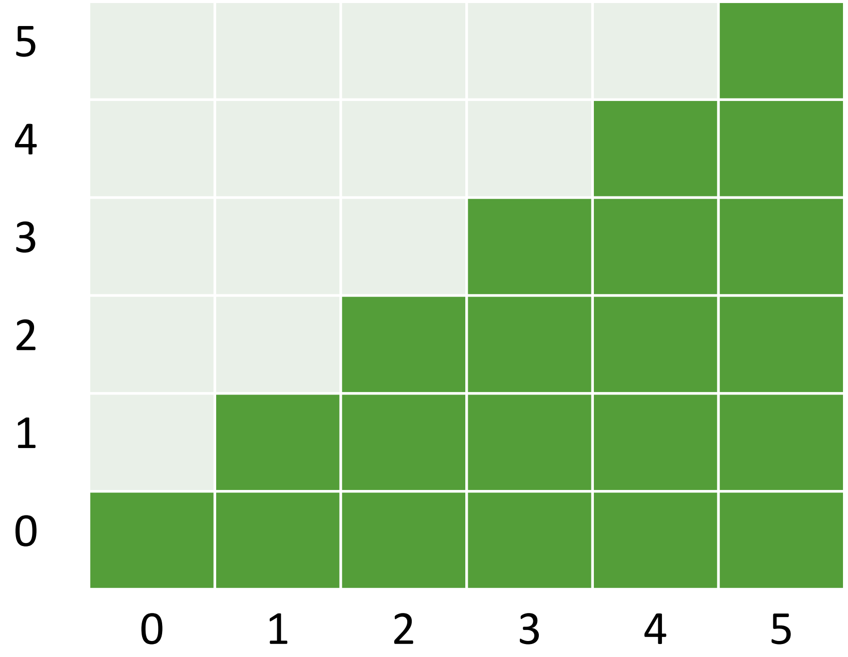
- Asymptotic Analysis - Example
- Computing Prefix Averages: The i-th prefix average of an array X is average of the first (i + 1) elements of X:
  - A[i] = (X[0] + X[1] + … + X[i])/(i+1)

```
/** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
public static double[ ] prefixAverage1(double[ ] x) {
  int n = x.length;
  double[ ] a = new double[n];                   // filled with zeros by default
  for (int j=0; j < n; j++) {
    double total = 0;                            // begin computing x[0] + ... + x[j]
    for (int i=0; i <= j; i++)
      total += x[i];
    a[j] = total / (j+1);                        // record the average
  }
  return a;
}
```



ASYMPTOTIC ALGORITHM ANALYSIS

- Asymptotic Analysis - Example
- The running time of **prefixAverage1** is $O(1 + 2 + \ldots + n)$
- The sum of the first ***n*** integers is **$n(n + 1) / 2$**
- There is a simple visual proof of this fact
- Thus, algorithm **prefixAverage1** runs in ***$O(n^2)$*** time



ASYMPTOTIC ALGORITHM ANALYSIS

- Asymptotic Analysis – Example 2
- Here is **prefixAverage2** running in ***O(n)*** time

```
/** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
public static double[ ] prefixAverage2(double[ ] x) {
  int n = x.length;
  double[ ] a = new double[n];                 // filled with zeros by default
  double total = 0;                            // compute prefix sum as x[0] + x[1] + ...
  for (int j=0; j < n; j++) {
    total += x[j];                             // update prefix sum to include x[j]
    a[j] = total / (j+1);                      // compute average based on current sum
  }
  return a;
}
```

RELATIVES OF BIG OH

- Relatives of Big Oh
- **big-Omega**
- f(n) is Ω(g(n)) if there is a constant $c > 0$ and an integer constant $n0 \geq 1$ such that

  $$f(n) \geq c\, g(n) \text{ for } n \geq n0$$

- **big-Theta**
- f(n) is Θ(g(n)) if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n0 \geq 1$ such that

  $$c' g(n) \leq f(n) \leq c'' g(n) \text{ for } n \geq n0$$

RELATIVES OF BIG OH

- Relatives of Big Oh
- **big-Oh**

  f(n) is O(g(n)) if f(n) is asymptotically **less than or equal to** g(n)
- **big-Omega**

  f(n) is Ω(g(n)) if f(n) is asymptotically **greater than or equal to** g(n)
- **big-Theta**

  f(n) is Θ(g(n)) if f(n) is asymptotically **equal to** g(n)

Math you need to Review

- Summations
- Powers
- Logarithms
- Proof techniques
- Basic probability

- Properties of powers:

  $a^{(b+c)} = a^b a^c$

  $a^{bc} = (a^b)^c$

  $a^b / a^c = a^{(b-c)}$

  $b = a^{\log_a b}$

  $b^c = a^{c*\log_a b}$

- Properties of logarithms:

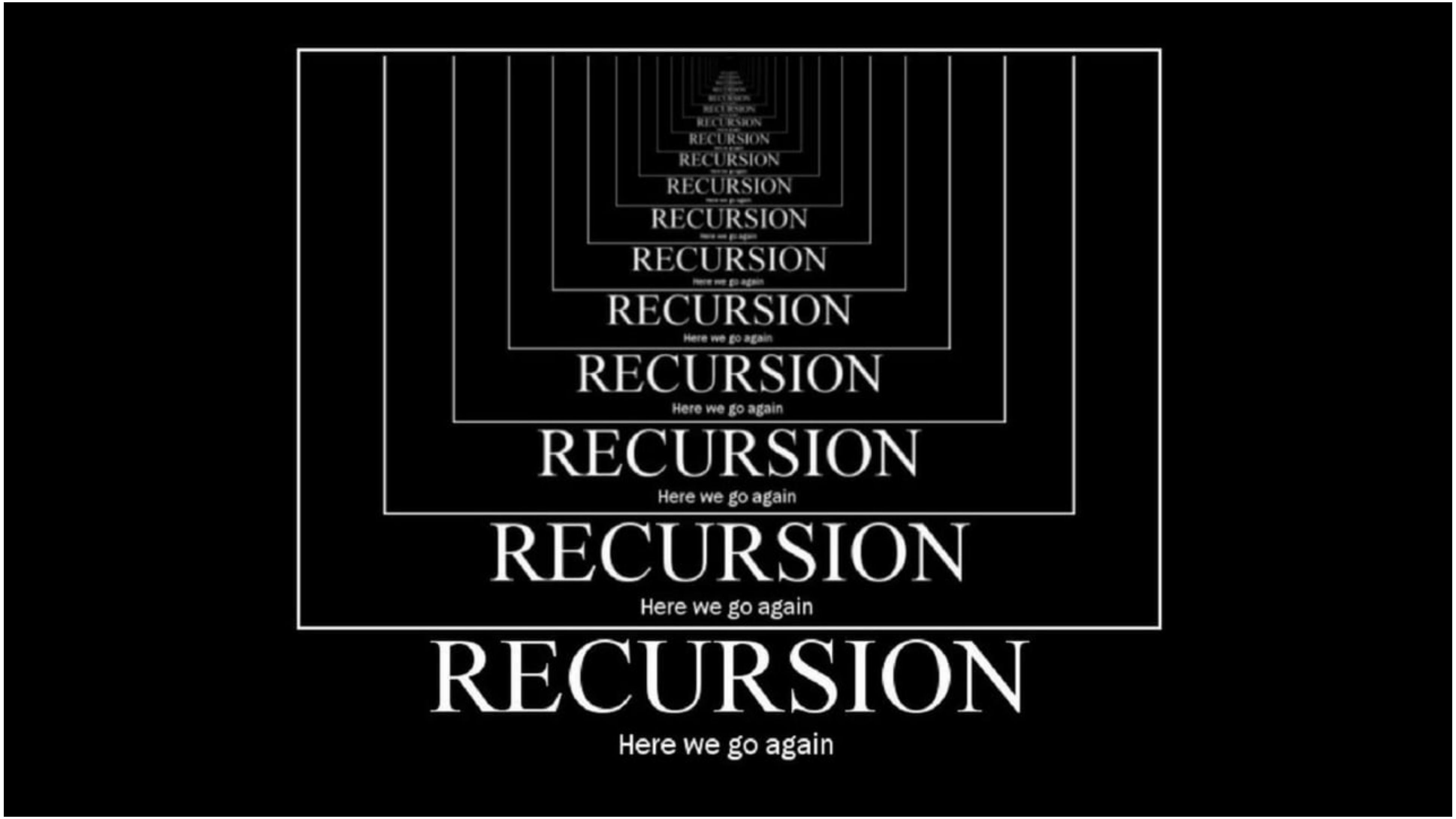
  $\log_b(xy) = \log_b x + \log_b y$

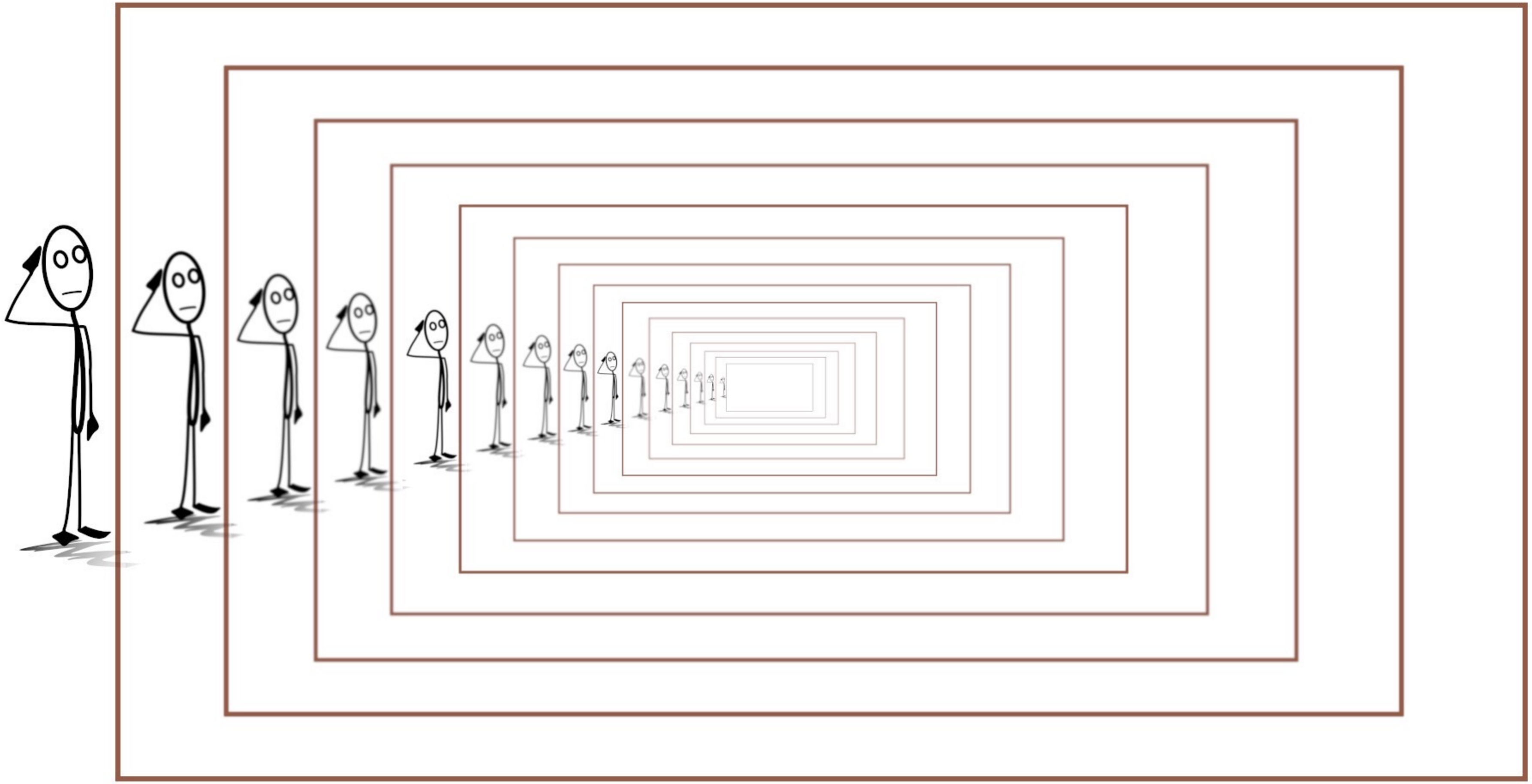
  $\log_b (x/y) = \log_b x - \log_b y$

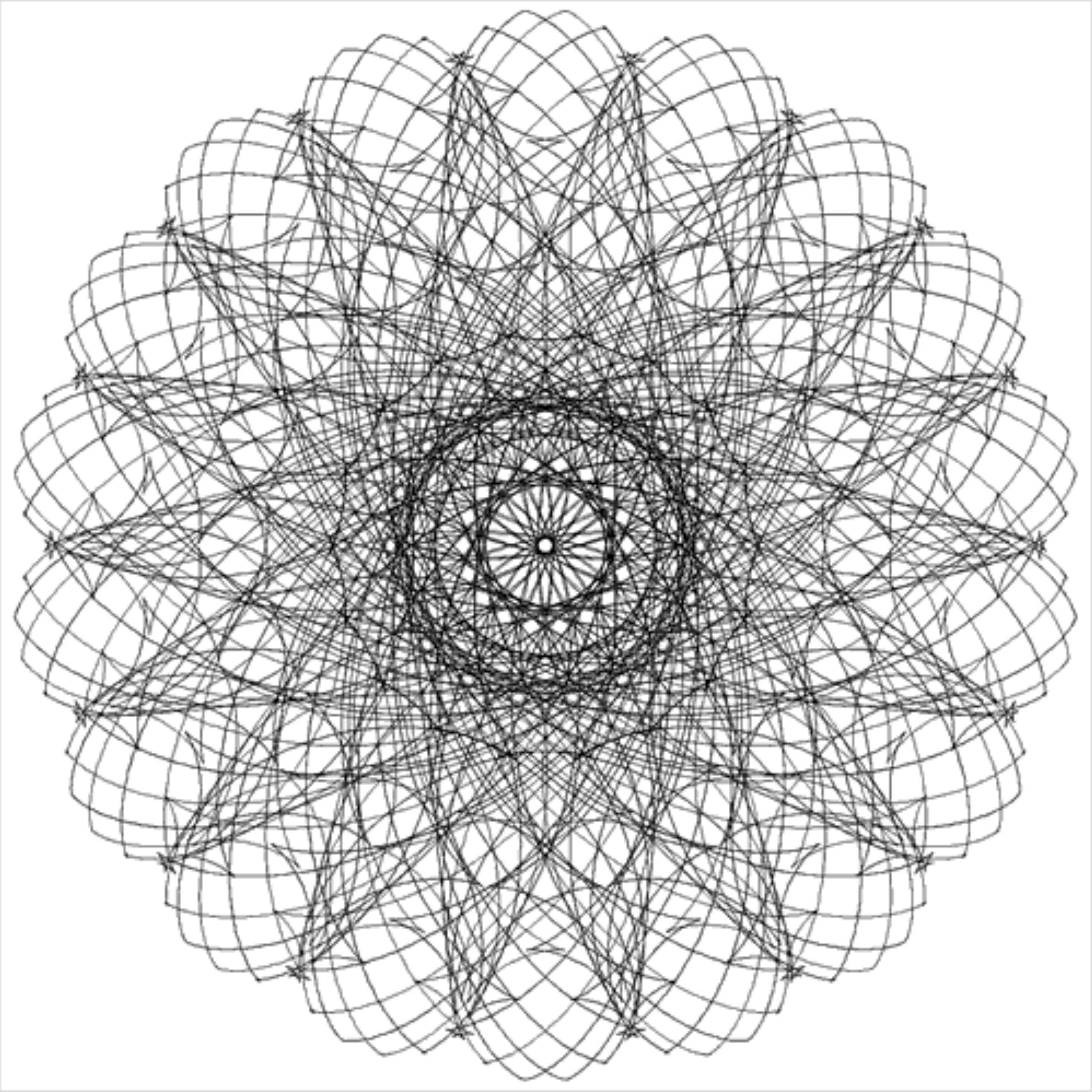
  $\log_b xa = a\log_b x$

  $\log_b a = \log_x a/\log_x b$

ANALYSIS OF RECURSIVE ALGORITHMS


RECURSION
Here we go again



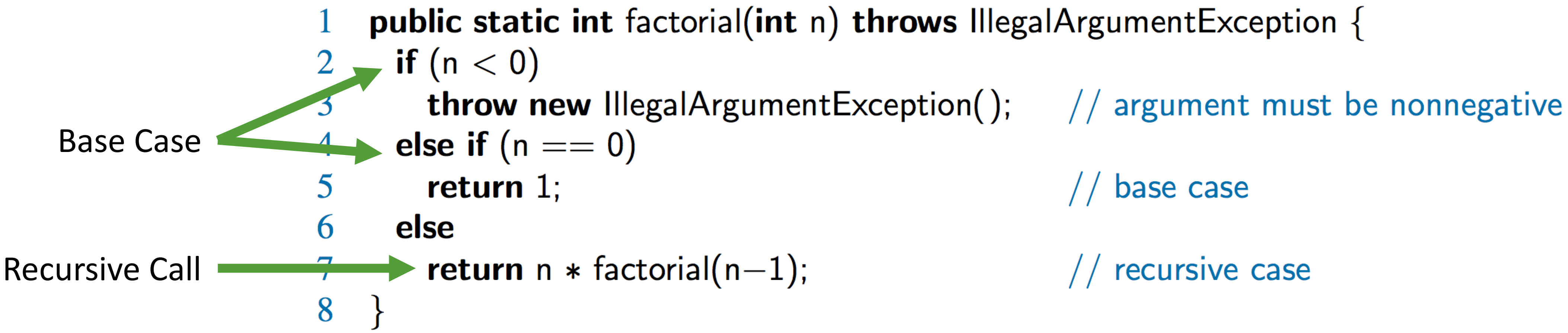


RECURSION

- Recursion: when a method calls itself
- Classic example – the factorial function:

$$n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n-1) \cdot n$$

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & else \end{cases}$$



Base Case

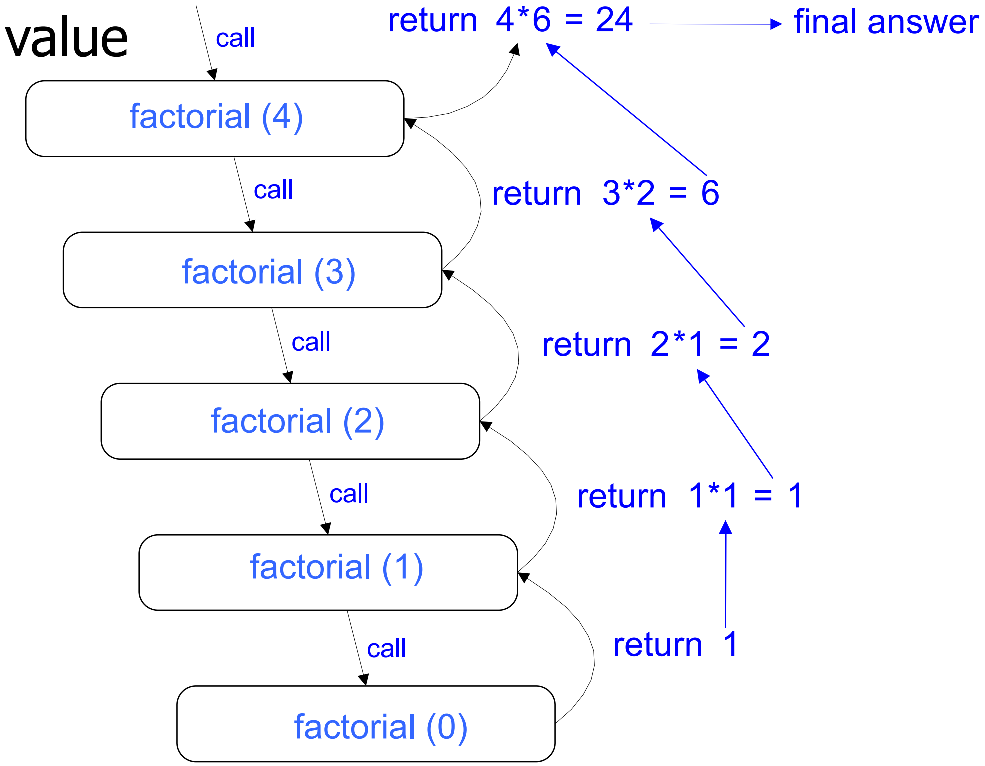
Recursive Call

```
public static int factorial(int n) throws IllegalArgumentException {
  if (n < 0)
    throw new IllegalArgumentException( );    // argument must be nonnegative
  else if (n == 0)
    return 1;                                 // base case
  else
    return n * factorial(n−1);                // recursive case
}
```

RECURSION

- **Building a recursion tree:**
  - A box for each recursive call
  - An arrow from each caller to callee
  - An arrow from each callee to caller showing return value

```
public static int factorial(int n) {
 if(n == 0)
   return 1;
 else
   return n * factorial(n - 1);
}
```

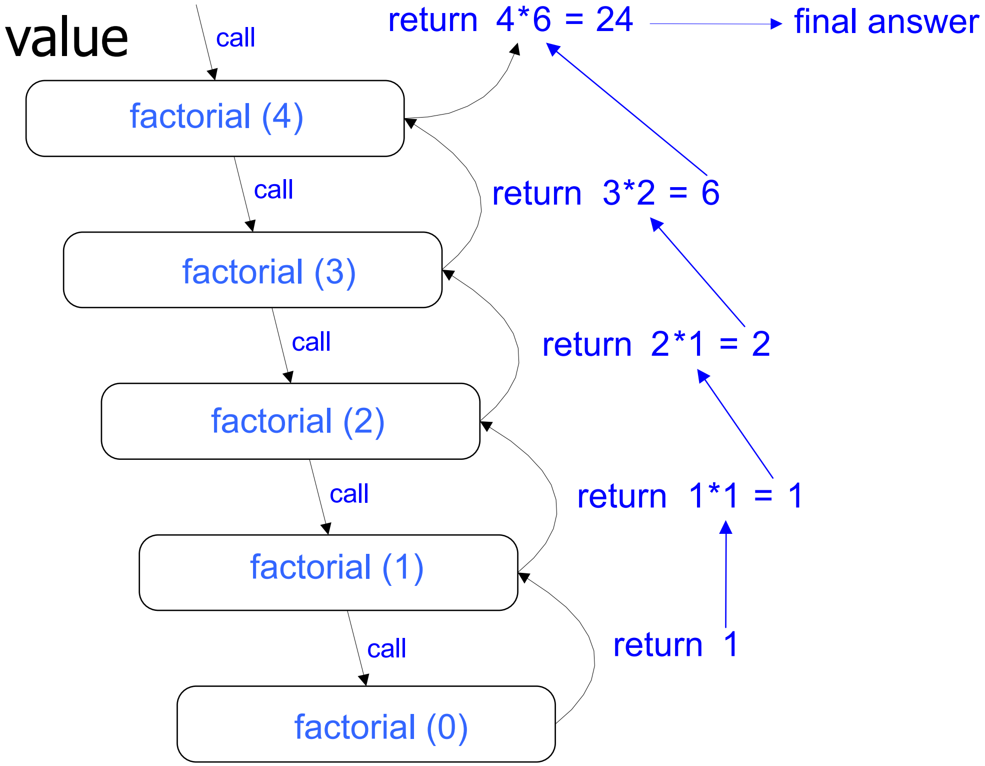


So what is the runtime for factorial ()?

RECURSION

Runtime as Big Oh

- A box for each recursive call
- An arrow from each caller to callee
- An arrow from each callee to caller showing return value



```
public static int factorial(int n) {
  if(n == 0)
    return 1;
  else
    return n * factorial(n - 1);
}
```

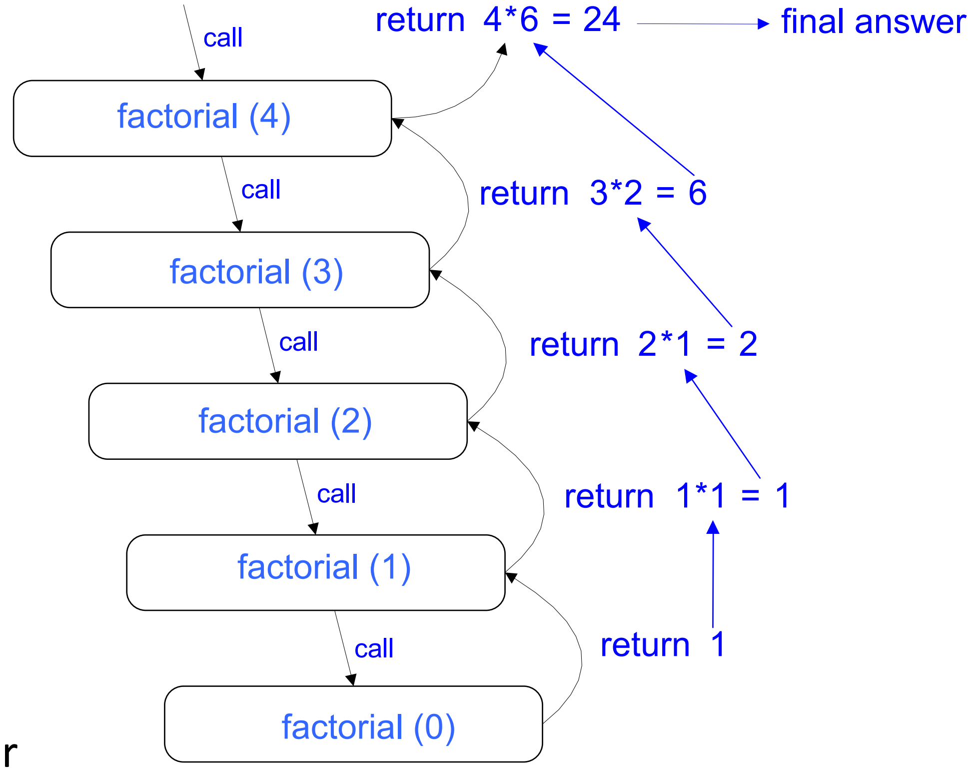
Looking at the recursion tree, we can determine

- factorial call is made for values 4, 3, 2, 1 and 0;
- 0 being the base case, there are 4 recursive calls when *n* = 4.
- for larger *n*, there would be *n* calls.
- So the runtime for factorial can be given as **O(n).**

RECURSION

- Estimating the number of operations:
  - Base call occurs only once
  - Recursive calls are made repeatedly
    - Recursion tree can help determine the order of growth.

```
1 op    public static int factorial(int n) {
1 op     if(n == 0)
1 op       return 1;
         else
4 ops      return n * factorial(n - 1);
        }
```



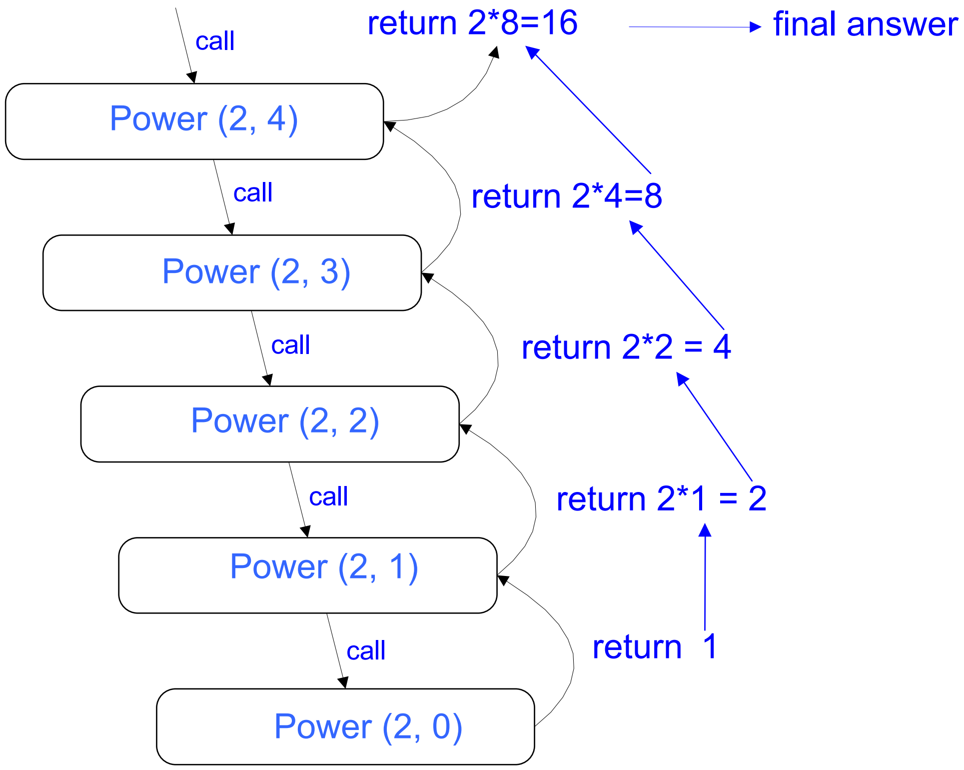
- Total recursive operations = 6 ; base-case operations is 1.
- Looking at the recursion tree, we estimate the runtime to be linear
- So **T(n) = 6n + c**
- where c is a constant time (includes base case + cost of recursion)

RECURSION

- Examples: Computing Powers

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \text{else} \end{cases}$$

```
public static int Power(int x, int n) {
 if(n == 0)
   return 1;
 else
   return x * Power(x, n - 1);
}
```


call
return 2*8=16
final answer
Power (2, 4)
call
return 2*4=8
Power (2, 3)
call
return 2*2 = 4
Power (2, 2)
call
return 2*1 = 2
Power (2, 1)
call
return 1
Power (2, 0)

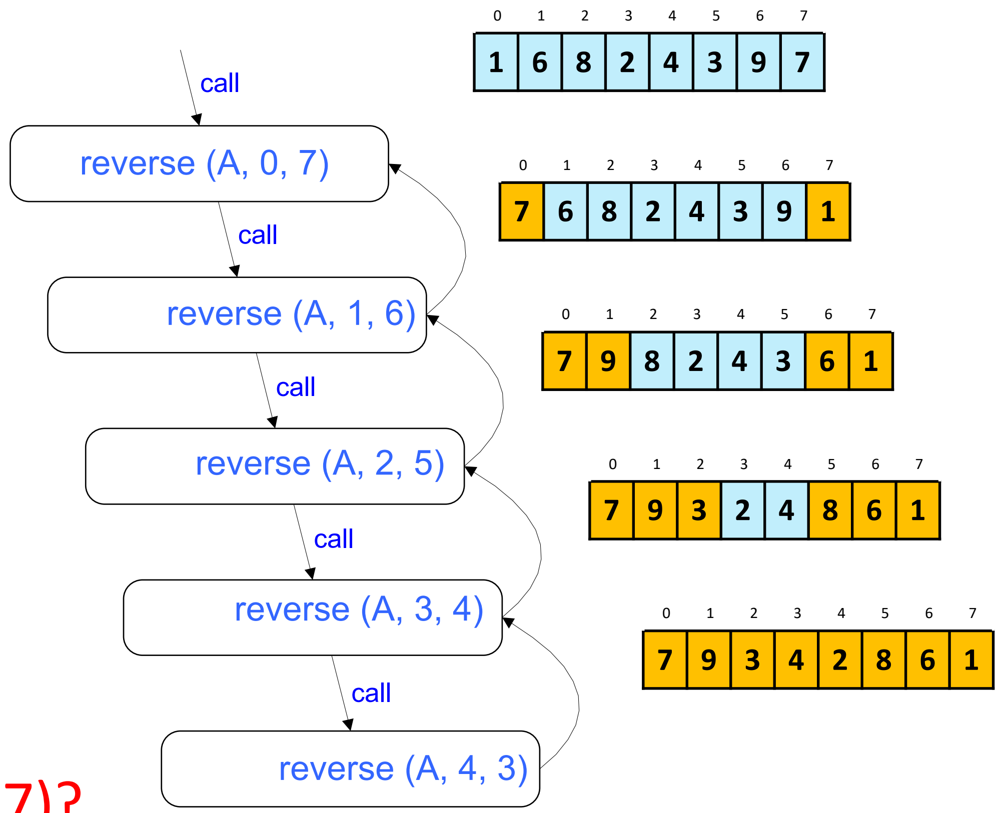
So what is the runtime for Power (2,4)?

O(n)

RECURSION

- Examples: Reversing an array

```
public static void reverse(int [] A, int i, int j){
  if(i >= j)
    return;
  else {
    int temp = A[i];
    A[i] = A[j];
    A[j] = temp;
    return reverse(A, i+1, j-1);
}
```



0	1	2	3	4	5	6	7
1	6	8	2	4	3	9	7

0	1	2	3	4	5	6	7
7	6	8	2	4	3	9	1

0	1	2	3	4	5	6	7
7	9	8	2	4	3	6	1

0	1	2	3	4	5	6	7
7	9	3	2	4	8	6	1

0	1	2	3	4	5	6	7
7	9	3	4	2	8	6	1

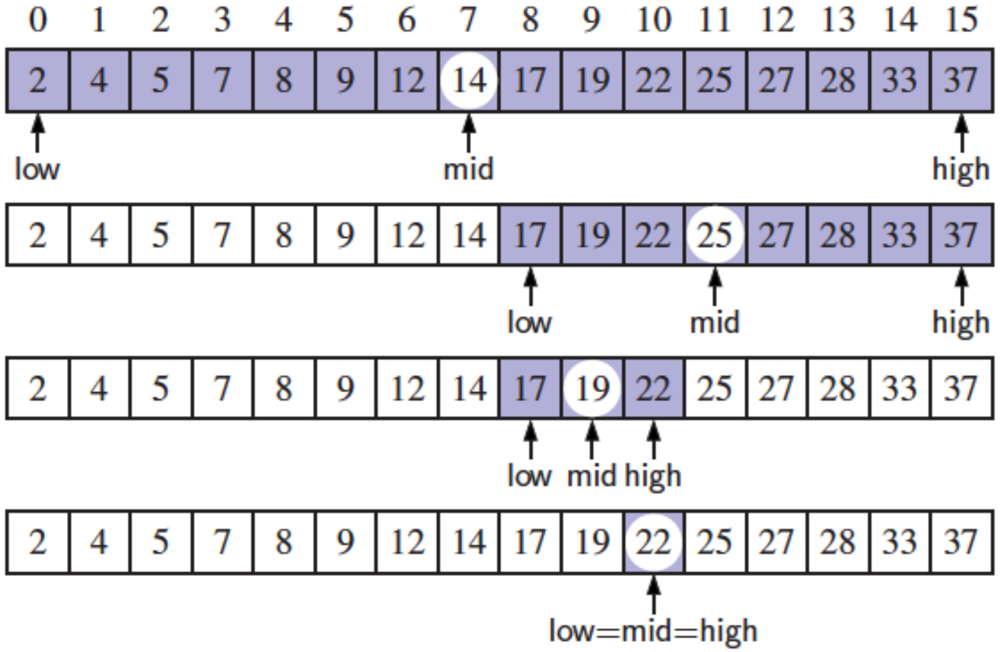
So what is the runtime for reverse (A,0,7)?

O(n/2)

If n = 7; then **n/2** calls were made to reach the middle of the array.

RECURSION

- **Examples: Binary Search:** Search for an integer in an ordered list
  - We consider three cases:
    - If the target equals data[mid], then we have found the target.
    - If target < data[mid], then we recur on the first half of the sequence.
    - If target > data[mid], then we recur on the second half of the sequence.



RECURSION

- Examples: Binary Search

```
public static boolean Bsearch(int [] A, int X, int lo, int hi){
 if(lo >= hi)
   return false;
 else {
   int mid = (lo+hi)/2;
   if (X == A[mid])
     return true;
   else if (X < A[mid])
     return Bsearch(A, X, lo, mid-1);
   else
     return Bsearch(A, X, mid+1, hi);
 }
}
```

$$(\mathrm{mid}-1)-\mathrm{low}+1=\left\lfloor\frac{\mathrm{low}+\mathrm{high}}{2}\right\rfloor-\mathrm{low}\leq\frac{\mathrm{high}-\mathrm{low}+1}{2}$$

$$\mathrm{high}-(\mathrm{mid}+1)+1=\mathrm{high}-\left\lfloor\frac{\mathrm{low}+\mathrm{high}}{2}\right\rfloor\leq\frac{\mathrm{high}-\mathrm{low}+1}{2}.$$

Each recursive call divides the search region in half; hence, there can be at most ***log n*** levels

So runtime is O(log n)

RECURSION

• Examples: Fibonacci numbers

- Fibonacci numbers are defined recursively:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2} \quad \text{for } i > 1.$$

```
public static int Fibonacci(int k){
 if(k==0)
   return 0;
 else if(k==1)
   return 1;
 else
   return Fibonacci(k-1) + Fibonacci(k-2);
}
```

RECURSION

- Examples: Fibonacci numbers
- Let n_k be the number of recursive calls by BinaryFib(k)
  - $n_0 = 1$
  - $n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
  - $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
  - $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67.$
- Note that n_k at least doubles every other time
- That is, $n_k > 2^{k/2}$. It is exponential. **$O(2^n)$**

NOTE

Materials for this set of slides were extracted from

- Goodrich, Tamassia, Goldwasser ,”Analysis of Algorithms”, 6th edition, Wiley, 2014
- Robert Sedgewick and Kevin Wayne, ”Algorithms”, 4th edition, Addison Wesley, 2011.