CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

5: Analysis of Algorithms

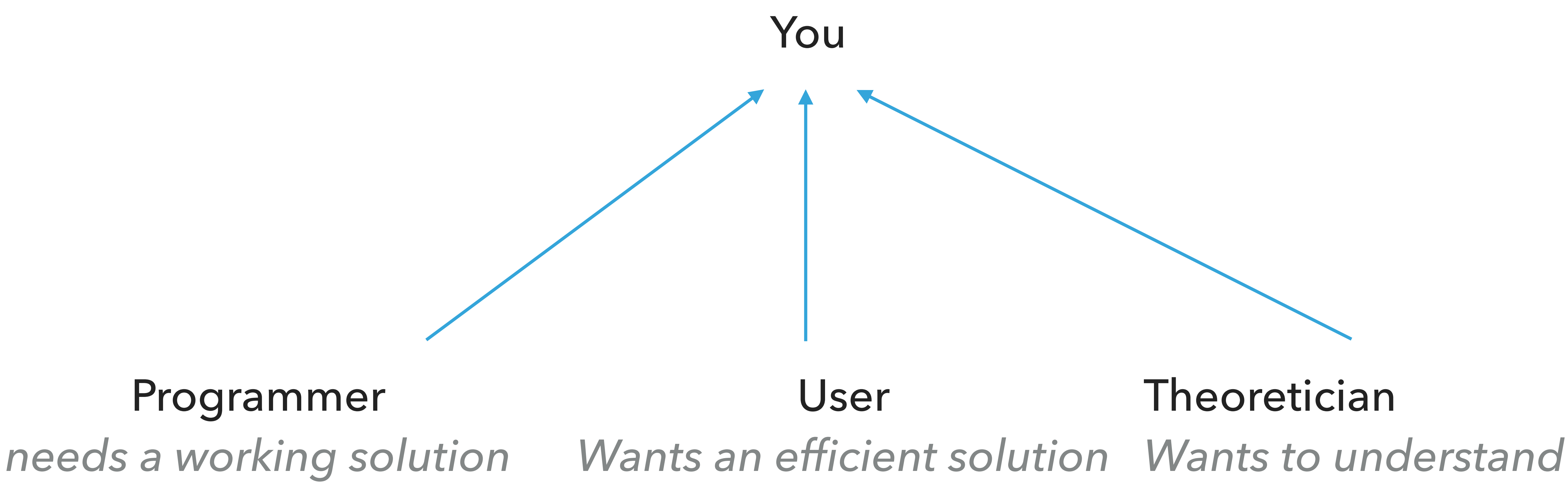
Alexandra Papoutsaki
she/her/hers

Lecture 5: Analysis of Algorithms

- Introduction
- Experimental Analysis of Running Time
- Mathematical Models of Running Time
- Order of Growth Classification
- Analysis of Memory Consumption

Some slides adopted from Algorithms 4th Edition or COS226

Different Roles



Why analyze algorithmic efficiency?

- Predict performance.
- Compare algorithms that solve the same problem.
- Provide guarantees.
- Understand theoretical basis.
- **Avoid performance bugs.**

Why is my program so slow?
Why does it run out of memory?

We can use a combination of experiments and mathematical modeling.

Lecture 5: Analysis of Algorithms

- Introduction
- Experimental Analysis of Running Time
- Mathematical Models of Running Time
- Order of Growth Classification
- Analysis of Memory Consumption

3-SUM: Given n distinct numbers, how many unordered triplets sum to 0?

- Input: 30 -40 -20 -10 40 0 10 5
- Output: 4
  - 30 -40 10
  - 30 -20 -10
  - -40 40 0
  - -10 0 10

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)  {
        String filename = args[0];
        int fileSize = Integer.parseInt(args[1]);
        try {
            Scanner scanner = new Scanner(new File(filename));
            int intList[] = new int[fileSize];
            int i=0;
            while(scanner.hasNextInt()){
                intList[i++]=scanner.nextInt();
            }
            Stopwatch timer = new Stopwatch();
            int count = count(intList);
            System.out.println("elapsed time = " + timer.elapsedTime());
            System.out.println(count);
        }
        catch (IOException ioe) {
            throw new IllegalArgumentException("Could not open " + filename, ioe);
        }
    }
```

Empirical Analysis

- Input: 8ints.txt
- Output: 4 and 0

- Input: 1Kints.txt
- Output: 70 and 0.081

- Input: 2Kints.txt
- Output: 528 and 0.38

- Input: 2Kints.txt
- Output: 528 and 0.371

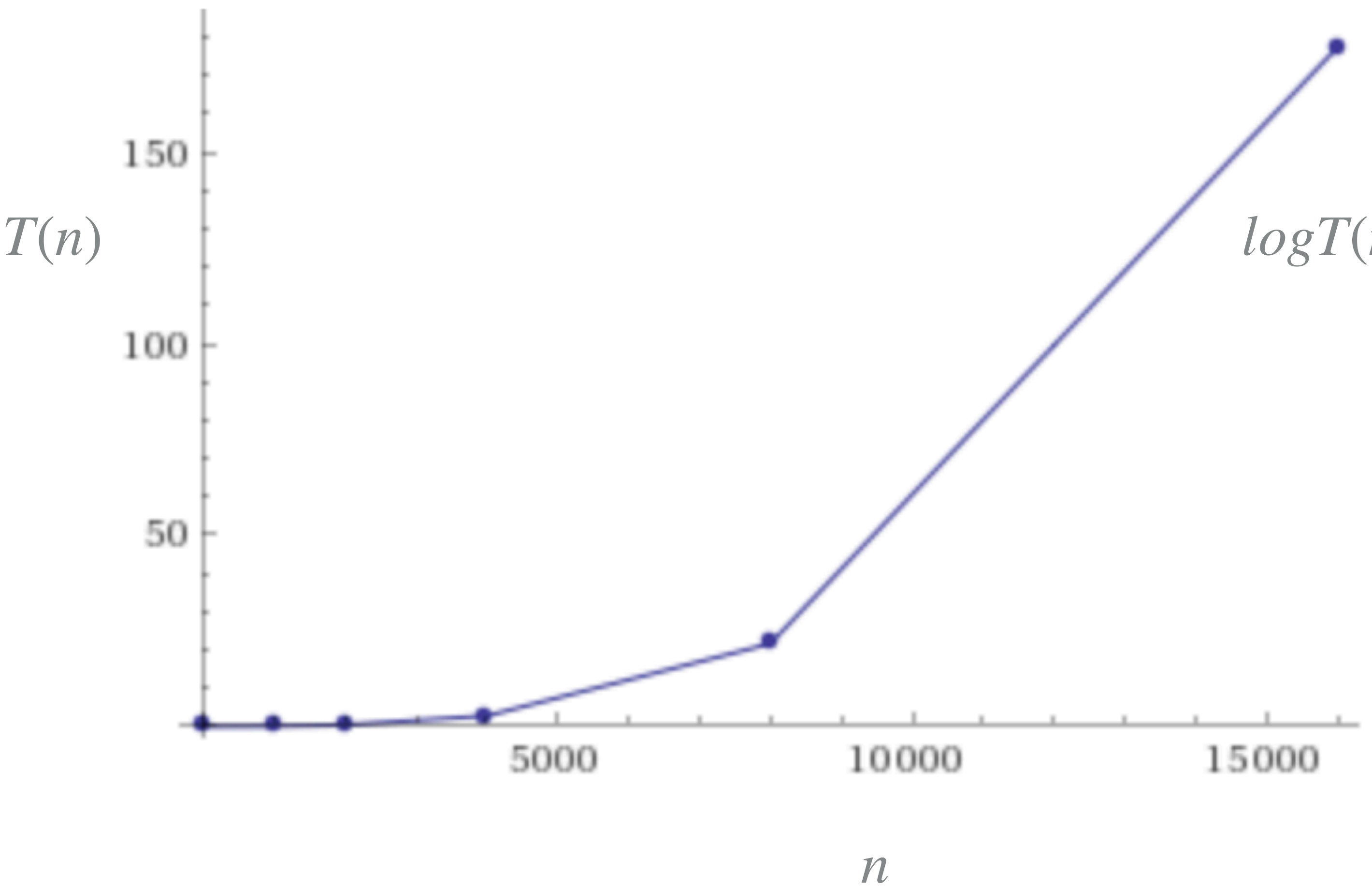
- Input: 4Kints.txt
- Output: 4039 and 2.792

- Input: 8Kints.txt
- Output: 32074 and 21.623

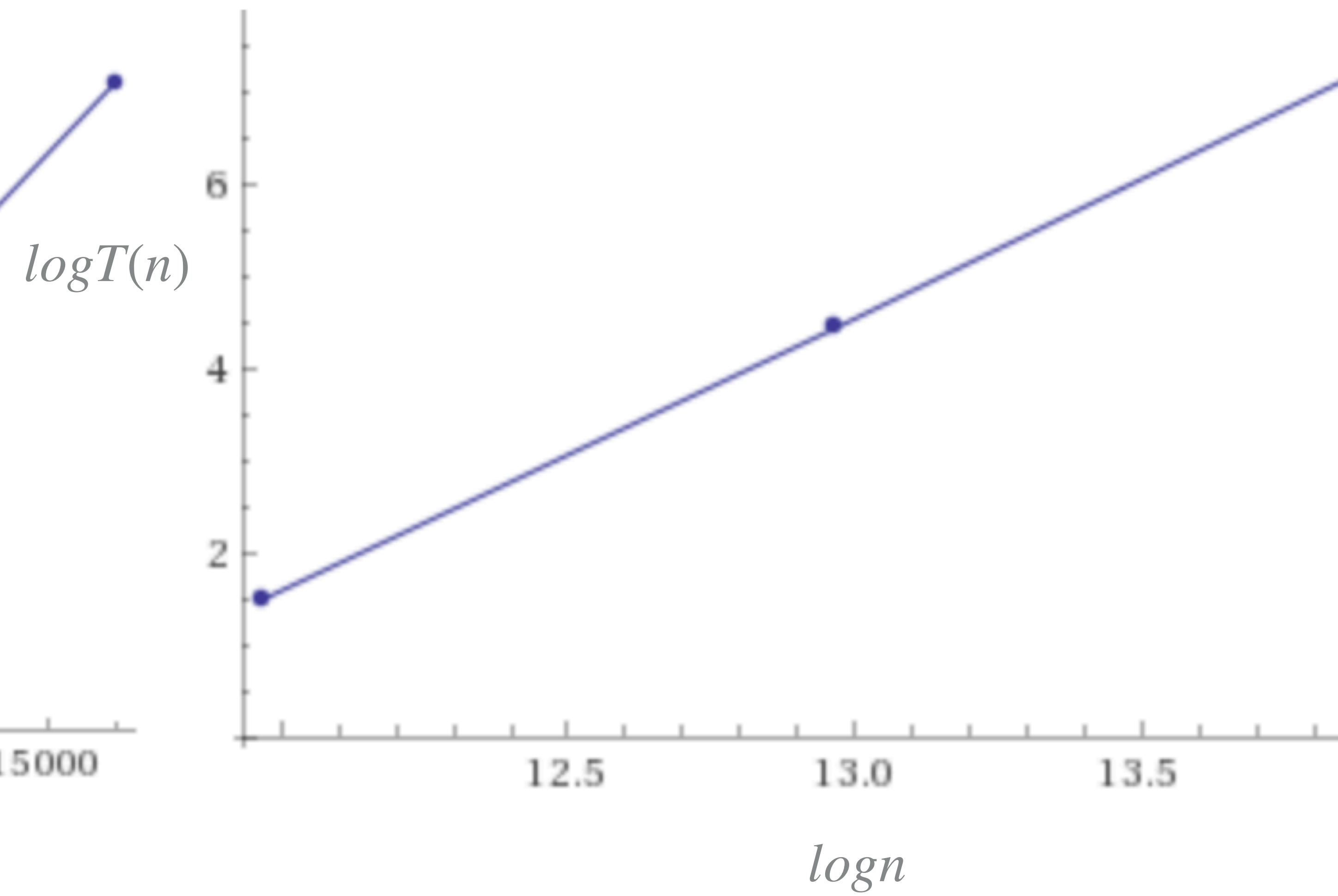
- Input: 16Kints.txt
- Output: 255181 and 177.344

Input size	Time
8	0
1000	0.081
2000	0.38
2000	0.371
4000	2.792
8000	21.623
16000	177.344

Plots and log-log plots



Straight line of slope 3



- Regression: $T(n) = an^b$ (power-law).
- $\log T(n) = b \log n + \log a$, where b is slope.
- Experimentally: ~$0.42 \times 10^{-10} n^3$, in our example for ThreeSum.

Input size	Time
8	0
1000	0.081
2000	0.38
4000	2.792
8000	21.623
16000	177.344

Doubling Hypothesis

- Doubling input size increases running time by a factor of $\frac{T(n)}{T(n/2)}$
- Run program doubling the size of input. Estimate factor of growth:
  - $\frac{T(n)}{T(n/2)} = \frac{an^b}{a(\frac{n}{2})^b} = 2^b.$
- E.g., in our example, for pair of input sizes 8000 and 16000 the ratio $(\frac{177.344}{21.623})$ is 8.2 or ~8 which can be written as 2^3, therefore b is approximately 3.
- Assuming we know b, we can figure out a.
  - E.g., in our example, $T(16000) = 177.34 = a \times 16000^3$.
    - Solving for a we get $a = 0.42 \times 10^{-10}$.

Practice Time

- Suppose you time your code and you make the following observations. Which function is the closest model of $T(n)$?

A. n^2

B. $6 \times 10^{-4} n$

C. $5 \times 10^{-9} n^2$

D. $7 \times 10^{-9} n^2$

Input size	Time
1000	0
2000	0.0
4000	0.1
8000	0.3
16000	1.3
32000	5.1

Answer

- C. $5 \times 10^{-9} n^2$
- Ratio is approximately 4, therefore $b = 2$.
- $T(32000) = 5.1 = a \times 32000^2$.
- Solving for $a = 4.98 \times 10^{-9}$.s

Input size	Time
1000	0
2000	0.0
4000	0.1
8000	0.3
16000	1.3
32000	5.1

Effects on Performance

- System independent effects: Algorithm + input data
  - Determine b in power law relationships.
- System dependent effects: Hardware (e.g., CPU, memory, cache) + Software (e.g., compiler, garbage collector) + System (E.g., operating system, network, etc).
  - Dependent and independent effects determine a in power law relationships.

- Although it is hard to get precise measurements, experiments in Computer Science are cheap to run.

Lecture 5: Analysis of Algorithms

- Introduction
- Experimental Analysis of Running Time
- Mathematical Models of Running Time
- Order of Growth Classification
- Analysis of Memory Consumption

Total Running Time

- Popularized by Donald Knuth in the 60s in the four volumes of "The Art of Computer Programming".
  - Knuth won the Turing Award (The "Nobel" in CS) in 1974.

- In principle, accurate mathematical models for performance of algorithms are available.

- Total running time = sum of cost x frequency for all operations.
- Need to analyze program to determine set of operations.
- Exact cost depends on machine, compiler.
- Frequency depends on algorithm and input data.

Cost of Basic Operations

- Add < integer multiply < integer divide < floating-point add < floating-point multiply < floating-point divide.

Operation	Example	Nanoseconds
Variable declaration	`int a`	c_1
Assignment statement	`a = b`	c_2
Integer comparison	`a < b`	c_3
Array element access	`a[i]`	c_4
Array length	`a.length`	c_5
1D array allocation	`new int[n]`	$c_6 n$
2D array allocation	`new int[n][n]`	$c_7 n^2$
string concatenation	`s+t`	$c_8 n$

Example: 1-SUM

- How many operations as a function of n?

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

Operation	Frequency
Variable declaration	2
Assignment	2
Less than	$n+1$
Equal to	n
Array access	n
Increment	n to $2n$

Example: 2-SUM

- How many operations 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++;
         }
     }
 }
```

BECOMING TOO TEDIOUS TO CALCULATE

Operation	Frequency
Variable declaration	$n+2$
Assignment	$n+2$
Less than	$1/2(n+1)(n+2)$
Equal to	$1/2n(n-1)$
Array access	$n(n-1)$
Increment	$1/2n(n+1)$ to n^2

Tilde Notation

- Estimate running time (or memory) as a function of input size n.
- Ignore lower order terms.
  - When n is large, lower order terms become negligible.

- Example 1: $\frac{1}{6}n^3 + 10n + 100 \quad \sim n^3$

- Example 2: $\frac{1}{6}n^3 + 100n^2 + 47 \quad \sim n^3$

- Example 3: $\frac{1}{6}n^3 + 100n^{\frac{2}{3}} + \frac{1/2}{n} \quad \sim n^3$

- Technically $f(n) \sim g(n)$ means that $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 1$

Simplification

- Cost model: Use some basic operation as proxy for running time.
  - E.g., array accesses
- Combine it with tilde notation.

Operation	Frequency	Tilde notation
Variable declaration	$n + 2$	$\sim n$
Assignment	$n + 2$	$\sim n$
Less than	$1/2(n+1)(n+2)$	$\sim n^2$
Equal to	$1/2n(n-1)$	$\sim n^2$
Array access	$n(n-1)$	$\sim n^2$
Increment	$1/2n(n+1)$ to n^2	$\sim n^2$

- $\sim n^2$ array accesses for the 2-SUM problem

Back to the 3-SUM problem

- 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++;
            }
        }
    }
}
```

- n^3 array accesses.

Useful approximations

- Harmonic sum: $H_n = 1 + 1/2 + 1/3 + \ldots + 1/n \quad \sim \ln n$
- Triangular sum: $1 + 2 + 3 + \ldots + n \quad \sim n^2$
- Geometric sum: $1 + 2 + 4 + 8 + \ldots + n = 2n - 1 \quad \sim n$, when n power of 2.
- Binomial coefficients: $\binom{n}{k} \quad \sim \frac{n^k}{k!}$ when k is a small constant.

- Use a tool like Wolfram alpha.

Practice Time

- How many array accesses does the following code make?

```
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++;
        }
    }
}
```

A. n^2

B. $n^2 \log n$

C. n^3

D. $n^3 \log n$

Answer

- B. $n^2 \log n$

Lecture 5: Analysis of Algorithms

- Introduction
- Experimental Analysis of Running Time
- Mathematical Models of Running Time
- Order of Growth Classification
- Analysis of Memory Consumption

Order of growth

- Definition: If $f(n) \sim cg(n)$ for some constant $c > 0$, then the order of growth of $f(n)$ is $g(n)$.
  - Ignore leading coefficients.
  - Ignore lower-order terms.

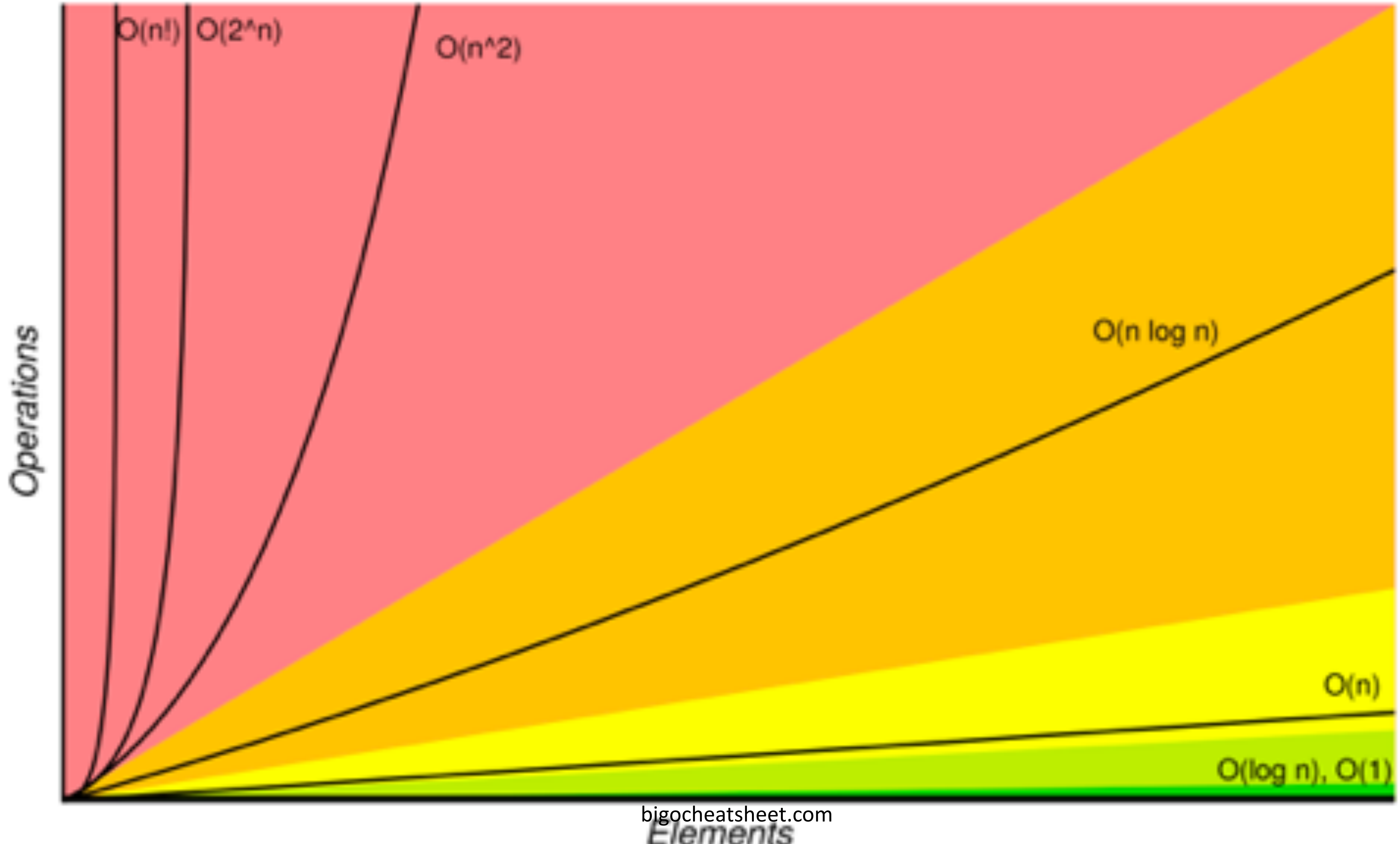
- We will use this definition in the mathematical analysis of the running time of our programs as the coefficients depend on the system.
- E.g., the order of growth of the running time of the `ThreeSum` program is n^3.

Common order of growth classifications

- Good news: only a small number of function suffice to describe the order-of-growth of typical algorithms.
- 1: constant
  - Doubling the input size, won't affect the run-time. Holy-grail
- $\log n$: logarithmic
  - Doubling the input size, will increase the runtime by a constant.
- n : linear
  - Doubling the input size, will result to double the run-time.
- $n \log n$: linearithmic
  - Doubling the input size, will result to a bit longer than double the run-time.
- n^2: quadratic
  - Doubling the input size, will result to four times as much run-time.
- n^3: cubic
  - Doubling the input size, will result to eight times as much run-time.
- 2^n: exponential
  - When you increase the input by some constant amount, the time taken is doubled.
- $n!$: factorial
  - Runtime grows exponentially with the size of the input

From slowest growing to fastest growing

- $1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n!$



Common order of growth classifications

Order-of-growth	Name	Example code	$T(n)/T(n/2)$
1	Constant	`a=b+c`	1
$\log n$	Logarithmic	`while(n>1){n=n/2;…}`	~ 1
n	Linear	`for(int i=0; i<n; i++{`	2
$n \log n$	Linearithmic	`for (i = 1; i <= n; I++){ int x = n; while (x > 0) x -= i; }`	~ 2
n^2	Quadratic	`for(int i=0; i<n; i++) { for(int j=0; j<n; j++){`	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++){`	8

Lecture 5: Analysis of Algorithms

- Introduction
- Experimental Analysis of Running Time
- Mathematical Models of Running Time
- Order of Growth Classification
- Analysis of Memory Consumption

Basics

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

Typical use of memory for primitives and arrays

- `boolean`: 1 byte
- `byte`: 1 byte
- `char`: 2 bytes
- `short`: 2 bytes
- `int`: 4 bytes
- `float`: 4 bytes
- `long`: 8 bytes
- `double`: 8 bytes
- Array overhead: 24 bytes
  - `char[n]`:2n+24 bytes
  - `int[n]`:4n+24 bytes
  - `double[n]`:8n+24 bytes

Typical use of memory for objects

- Object overhead: 16 bytes
- Reference: 8 bytes
- Padding: padded to be a multiple of 8 bytes
- Example:
  - ```
    public class Date {
            private int day;
            private int month;
            private int year;
    }
    ```
  - 16 bytes overhead + 3x4 bytes for ints + 4 bytes padding = 32 bytes
    ```

Practice Time

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

```
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;
…
}
```

A. ~$4n$ bytes

B. ~$8n$ bytes

C. ~$4n^2$ bytes

D. ~$8n^2$ bytes

Answer

B. ~$8n$ bytes

- 16 bytes for object overhead
- Each array: 8 bytes for reference + 24 overhead + 4n for integers
- 4 bytes for int
- 4 bytes for padding
- Total $88 + 8n \sim 8n$

Lecture 5: Analysis of Algorithms

- Introduction
- Experimental Analysis of Running Time
- Mathematical Models of Running Time
- Order of Growth Classification
- Analysis of Memory Consumption

Readings:

- Recommended Textbook:
  - Chapter 1.4 (pages 172-196, 200-205)
- Recommended Textbook Website:
  - Analysis of Algorithms: https://algs4.cs.princeton.edu/14analysis/

Code

- Lecture 5 code

Practice Problems:

- 1.4.1-1.4.9