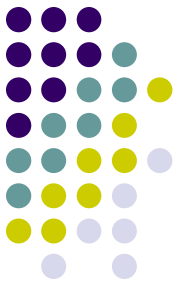


Mastering Data Structures and Algorithms: A Practical Approach

Introduction, computational and
space complexity estimation



By Dr. Juan C. Gomez
Fall 2018



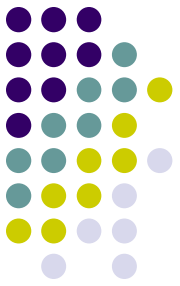
Overview

- Introduction
- What will not be taught in this class
- What? Why? And How?
- Big O notation
- Types of Analysis
- Space Complexity
- Big O simplification
 - i.e. Constants, non-dominant terms, recursion, etc
- Examples



Why are we here?

- Google-ization of tech Interviews
 - Do you know how to program?
 - Do you know your data structures?
 - Do you know the algorithms to use?
 - Are you detail oriented?
 - Do you know how to test your code?
 - Can you estimate the time and memory complexity of your programs?



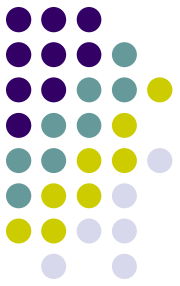
Evolution of Tech Interviews

- I hired an awesome programmer in C++, but he has no idea of how to design a large system.
- He does not know what a race condition is.
- He does not know what Virtual Memory is.
- He does not know what pipelining is.
- He does not know what a TCP segment is.
- He only works alone.



Modern Tech Interview

- System scale design
- System knowledge
- Networking knowledge
- Teamwork skills
- Object Oriented concepts
- Computer Architecture skills
- Programming skills:
 - Algorithms
 - Data structures
 - Testing
 - Estimating time and memory complexity



And if that is not enough...

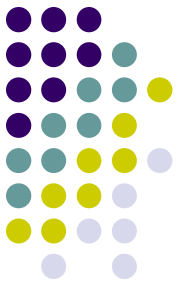
- I will give you the “Interview Laptop” to code and test your solution
 - Your program must not generate core files
 - Must pass all predefined tests
 - Oh and all this under 45 minutes!

But I do not want to work for Google!

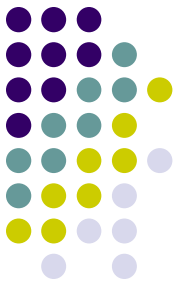


- But most tech companies are like Google these days when it comes to interviewing for most roles.
- So we must prepare well before facing these challenging interviews.

What we do not teach you here...



- System scale design
- System knowledge
- Networking knowledge
- Team work skills
- Object Oriented concepts
- Computer Architecture skills



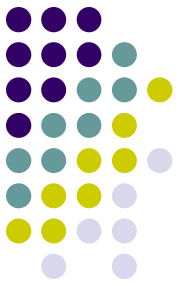
Focus of this class: What?

- Programming skills:
 - Algorithms
 - Data structures
 - Testing
 - Estimating time and memory complexity



Why?

- I am yet to hear someone being successful at a Google-ized interview without significant amount of preparation:
 - “I did 600 programming problems before passing the Google interview”
 - “My friend has taken 6 months off to prepare for the Google interview”



How?

- Practice makes perfect!
- Work on the board and on your computer.
 - Live audience correcting or challenging you.



Big O notation

- Used to measure performance and memory efficiency of your algorithms.
- Mastering this concept enables you to compare algorithms:
 - Memory requirements
 - CPU cycles requirements



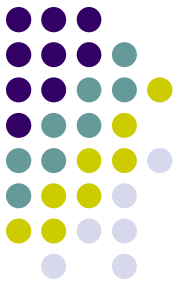
Big O: why it is important?

- Method to transfer a large file from SFO to NY:
 - Email, file transfer?
 - Hard drive on Airplane?
- Method to distribute 10 UPS parcels:
 - Distribution center to route that covers all destinations
 - Distribution center to destination n, go back.



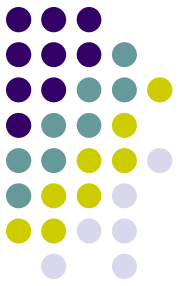
Big O: why it is important?

- Email, file transfer: time to complete has linear dependency with file size: $O(s)$
- Hard drive on Airplane?: time to complete is constant and does not depend on size of transfer: $O(1)$
- Distribution center to route that covers all destinations: time is linear function of distance to farthest destination: $O(d)$
- Distribution center to destination n , go back: function of the number of parcels (i.e. n) and distance to farthest destination: $O(d*n)$



Big O: some notation

- $T(n)$:
 - Function that defines the time taken by your algorithm for an input of size n .
- $M(n)$:
 - Function that defines the memory/space consumed by your algorithm for an input of size n .



Big O: taxonomy

- Big O: Upper bound
- Big Ω (Omega): Lower bound
- Big Θ (Theta): Tight bound



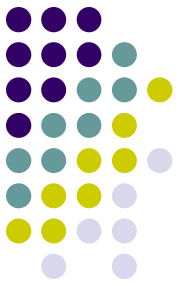
Big O: Upper bound

- $T(n)$ is said $O(f(n))$: iff
 - There is a positive constant 'c' and a positive integer n_1 for which:
 - $c \cdot f(n) \geq T(n)$ for every $n \geq n_1$



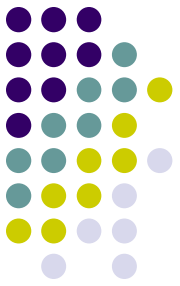
Big Ω (Omega): Lower bound

- $T(n)$ is said $\Omega(f(n))$: iff
 - There is a positive constant 'c' and a positive integer n_1 for which:
 - $c \cdot f(n) \leq T(n)$ for every $n \geq n_1$



Big Θ (Theta): Tight bound

- $T(n)$ is said $\Theta(f(n))$: iff
 - $T(n)$ is $\Omega(f(n))$
 - 'AND'
 - $T(n)$ is $O(f(n))$



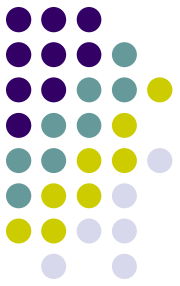
Big O: Rules!

- In industry: $\Theta(f(n))$ is overloaded to $O(f(n))$



Sample Run Times

- $O(1)$
- $O(\log(n))$
- $O(n \cdot \log(n))$
- $O(n)$
- $O(n^2)$
- $O(2^n)$
- $O(n!)$



Type of Analysis

- Best case:
 - If you get really lucky: how long will it take your algorithm to run.
- Worst case:
 - If Murphy's law holds: how long will it take your algorithm.
- Average case:
 - If you analyze running time for all possible inputs, what is the average execution time

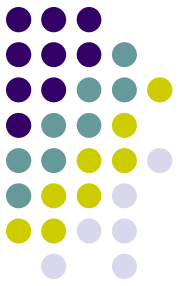


A real example: Insertion Sort

Sorted partial result		Unsorted data	
$\leq x$	$> x$	x	...

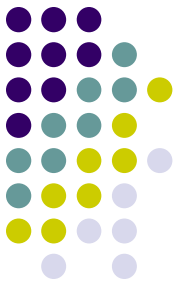
becomes

Sorted partial result			Unsorted data
$\leq x$	x	$> x$...





A real example: Insertion Sort

- Consider the following inputs to Insertion Sort:
 - {1, 2, 3, 4, 5, 6, 7}
 - {7, 6, 5, 4, 3, 2, 1}



A real example: Insertion Sort

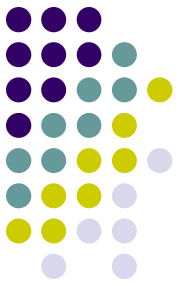
- {1, 2, 3, 4, 5, 6, 7}
 - Best Case Scenario: $O(n)$ 
- {7, 6, 5, 4, 3, 2, 1}
 - Worst Case Scenario: $O(n^2)$ 

A real example: Insertion Sort



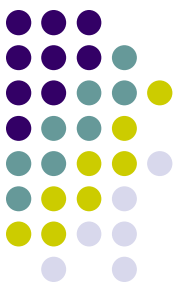
- Average Case Analysis:
 - $n!$ Possible inputs if sorting n elements
 - Compute $T(n)$ for each of those
 - Divide $\text{Sum}(T(n))/n!$
 - This is your average case analysis!





Space/Memory Complexity

- We do care about how much storage our system needs with respect to the size of the input: $M(n)$
- Memory may be:
 - Explicitly allocated
 - `int array[1000];`
 - Implicitly allocated via stack in recursive functions.



Space/Memory Complexity

```
1  int sum(int n) { /* Ex 1.*/  
2      if (n <= 0) {  
3          return 0;  
4      }  
5      return n + sum(n-1);  
6  }
```

$O(n)$ space.

Each call adds a level to the stack.

```
1  sum(4)  
2      -> sum(3)  
3          -> sum(2)  
4              -> sum(1)  
5                  -> sum(0)
```



Space/Memory Complexity

```
1  int pairSumSequence(int n) { /* Ex 2.*/  
2      int sum = 0;  
3      for (int i = 0; i < n; i++) {  
4          sum += pairSum(i, i + 1);  
5      }  
6      return sum;  
7  }  
8  
9  int pairSum(int a, int b) {  
10     return a + b;  
11 }
```

$O(1)$ space.



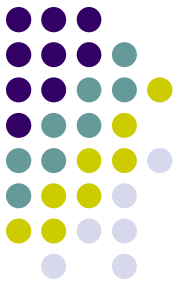
Effects of Constants in Big O

Min and Max 1

```
1  int min = Integer.MAX_VALUE;
2  int max = Integer.MIN_VALUE;
3  for (int x : array) {
4      if (x < min) min = x;
5      if (x > max) max = x;
6  }
```

Min and Max 2

```
1  int min = Integer.MAX_VALUE;
2  int max = Integer.MIN_VALUE;
3  for (int x : array) {
4      if (x < min) min = x;
5  }
6  for (int x : array) {
7      if (x > max) max = x;
8  }
```



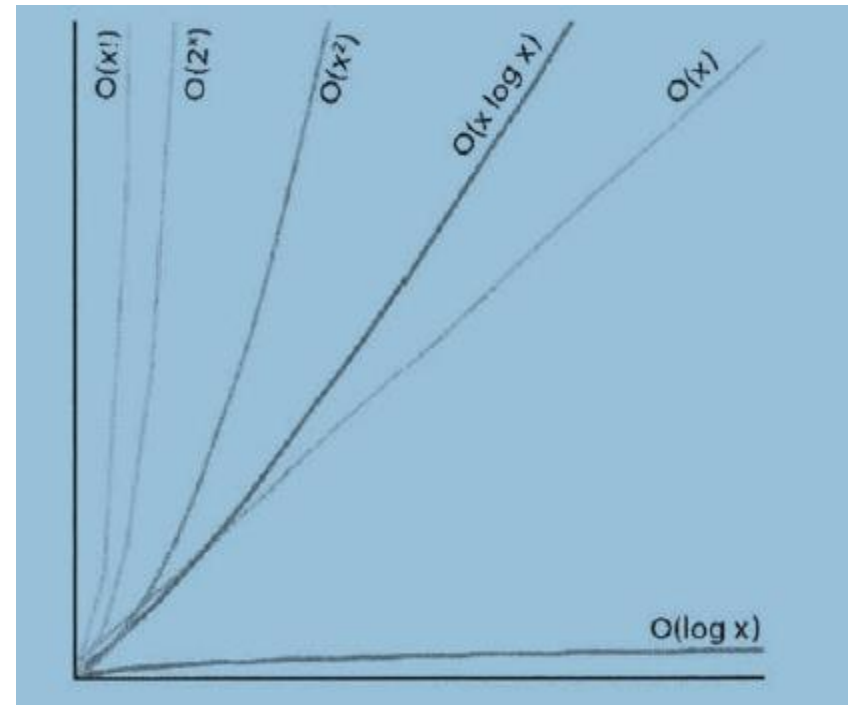
Effects of Constants in Big O

- $O(2^n) = O(n)$
- Doing work constant number of times for each input element still counts like one



Non-Dominant Terms

- $O(N^2 + N)$ becomes $O(N^2)$.
- $O(N + \log N)$ becomes $O(N)$.
- $O(5 \cdot 2^N + 1000N^{100})$ becomes $O(2^N)$.





Multi-Part Algorithms

Add the Runtimes: $O(A + B)$

```
1  for (int a : arrA) {  
2      print(a);  
3  }  
4  
5  for (int b : arrB) {  
6      print(b);  
7  }
```

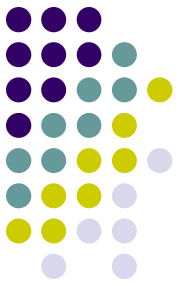
Multiply the Runtimes: $O(A*B)$

```
1  for (int a : arrA) {  
2      for (int b : arrB) {  
3          print(a + "," + b);  
4      }  
5  }
```



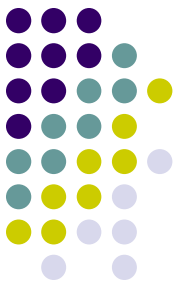
Amortized Time

- Inserting Elements into a Vector or Dynamic Array
- $O(1)$ in general
- $O(n)$ every time the library has to grow storage.
- Analysis of work needed to insert n elements:
 - n
 - $2 + 4 + 8 \dots n \rightarrow O(n)$



Log Runtimes

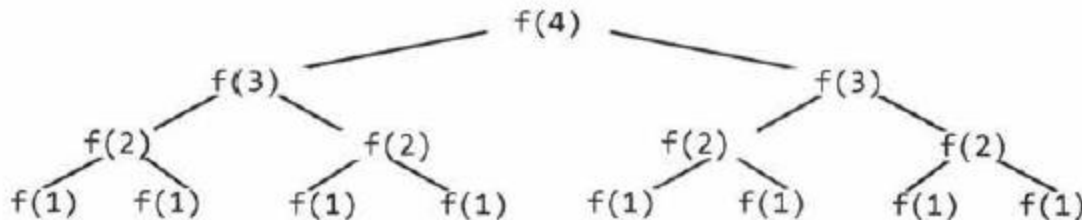
- Binary Search on a sorted array
- Search on a binary search tree
- In both cases the size of the input gets halved on each operation
- $n, n/2, n/4, \dots 2, 1$
- How many operations to process the full input:
 - $2^k = n \rightarrow k = \log_2(n)$



Recursive Runtimes

```
int f(int n) {  
    if (n <= 1) {  
        return 1;  
    }  
    return f(n - 1) + f(n - 1);  
}
```

Level	# Nodes	Also expressed as...	Or...
0	1		2^0
1	2	$2 * \text{previous level} = 2$	2^1
2	4	$2 * \text{previous level} = 2 * 2^1 = 2^2$	2^2
3	8	$2 * \text{previous level} = 2 * 2^2 = 2^3$	2^3
4	16	$2 * \text{previous level} = 2 * 2^3 = 2^4$	2^4



$O(2^n)$



Recursive Runtimes

- Recursion Tree
- Master Theorem

Handy Equations for Big O Analysis



- Sum of arithmetic series:

$$S_n = \frac{n}{2}(a_1 + a_n)$$

- Sum of geometric series:

$$\sum_{k=0}^n ar^k = \frac{a(1 - r^{n+1})}{1 - r}.$$



Examples

Example 1

What is the runtime of the below code?

```
1 void foo(int[] array) {  
2     int sum = 0;  
3     int product = 1;  
4     for (int i = 0; i < array.length; i++) {  
5         sum += array[i];  
6     }  
7     for (int i = 0; i < array.length; i++) {  
8         product *= array[i];  
9     }  
10    System.out.println(sum + ", " + product);  
11 }
```



Examples

Example 2

What is the runtime of the below code?

```
1 void printPairs(int[] array) {  
2     for (int i = 0; i < array.length; i++) {  
3         for (int j = 0; j < array.length; j++) {  
4             System.out.println(array[i] + "," + array[j]);  
5         }  
6     }  
7 }
```

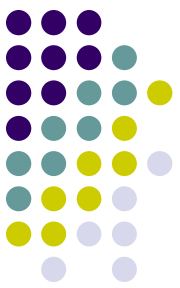



Examples

Example 3

This is very similar code to the above example, but now the inner for loop starts at $i + 1$.

```
1 void printUnorderedPairs(int[] array) {  
2     for (int i = 0; i < array.length; i++) {  
3         for (int j = i + 1; j < array.length; j++) {  
4             System.out.println(array[i] + "," + array[j]);  
5         }  
6     }  
7 }
```



Examples

Example 4

This is similar to the above, but now we have two different arrays.

```
1 void printUnorderedPairs(int[] arrayA, int[] arrayB) {
2     for (int i = 0; i < arrayA.length; i++) {
3         for (int j = 0; j < arrayB.length; j++) {
4             if (arrayA[i] < arrayB[j]) {
5                 System.out.println(arrayA[i] + "," + arrayB[j]);
6             }
7         }
8     }
9 }
```



Examples

Example 5

What about this strange bit of code?

```
1 void printUnorderedPairs(int[] arrayA, int[] arrayB) {  
2     for (int i = 0; i < arrayA.length; i++) {  
3         for (int j = 0; j < arrayB.length; j++) {  
4             for (int k = 0; k < 1000000; k++) {  
5                 System.out.println(arrayA[i] + "," + arrayB[j]);  
6             }  
7         }  
8     }  
9 }
```



Examples

Example 6

The following code reverses an array. What is its runtime?

```
1 void reverse(int[] array) {  
2     for (int i = 0; i < array.length / 2; i++) {  
3         int other = array.length - i - 1;  
4         int temp = array[i];  
5         array[i] = array[other];  
6         array[other] = temp;  
7     }  
8 }
```



Examples

Example 7

Which of the following are equivalent to $O(N)$? Why?

- $O(N + P)$, where $P < \frac{N}{2}$
- $O(2N)$
- $O(N + \log N)$
- $O(N + M)$



Examples

Example 8

Suppose we had an algorithm that took in an array of strings, sorted each string, and then sorted the full array. What would the runtime be?

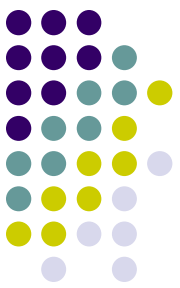


Examples

Example 9

The following simple code sums the values of all the nodes in a balanced binary search tree. What is its runtime?

```
1 int sum(Node node) {  
2     if (node == null) {  
3         return 0;  
4     }  
5     return sum(node.left) + node.value + sum(node.right);  
6 }
```



Examples (*)

Example 10

The following method checks if a number is prime by checking for divisibility on numbers less than it. It only needs to go up to the square root of n because if n is divisible by a number greater than its square root then it's divisible by something smaller than it.

For example, while 33 is divisible by 11 (which is greater than the square root of 33), the “counterpart” to 11 is 3 ($3 * 11 = 33$). 33 will have already been eliminated as a prime number by 3.

What is the time complexity of this function?

```
1  boolean isPrime(int n) {  
2      for (int x = 2; x * x <= n; x++) {  
3          if (n % x == 0) {  
4              return false;  
5          }  
6      }  
7      return true;  
}
```




Examples

Example 11

The following code computes $n!$ (n factorial). What is its time complexity?

```
1 int factorial(int n) {  
2     if (n < 0) {  
3         return -1;  
4     } else if (n == 0) {  
5         return 1;  
6     } else {  
7         return n * factorial(n - 1);  
8     }  
9 }
```



Examples (*)

Example 12

This code counts all permutations of a string.

```
1 void permutation(String str) {  
2     permutation(str, "");  
3 }  
4  
5 void permutation(String str, String prefix) {  
6     if (str.length() == 0) {  
7         System.out.println(prefix);  
8     } else {  
9         for (int i = 0; i < str.length(); i++) {  
10             String rem = str.substring(0, i) + str.substring(i + 1);  
11             permutation(rem, prefix + str.charAt(i));  
12         }  
13     }  
14 }
```



Examples

Example 13

The following code computes the Nth Fibonacci number.

```
1  int fib(int n) {  
2      if (n <= 0) return 0;  
3      else if (n == 1) return 1;  
4      return fib(n - 1) + fib(n - 2);  
5  }
```

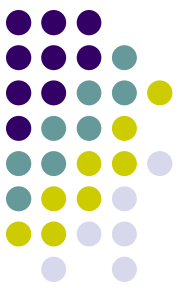


Examples

Example 14

The following code prints all Fibonacci numbers from 0 to n. What is its time complexity?

```
1 void allFib(int n) {  
2     for (int i = 0; i < n; i++) {  
3         System.out.println(i + ": " + fib(i));  
4     }  
5 }  
6  
7 int fib(int n) {  
8     if (n <= 0) return 0;  
9     else if (n == 1) return 1;  
10    return fib(n - 1) + fib(n - 2);  
11 }
```



Examples

Example 15

The following code prints all Fibonacci numbers from 0 to n. However, this time, it stores (i.e., caches) previously computed values in an integer array. If it has already been computed, it just returns the cache. What is its runtime?

```
1 void allFib(int n) {
2     int[] memo = new int[n + 1];
3     for (int i = 0; i < n; i++) {
4         System.out.println(i + ": " + fib(i, memo));
5     }
6 }
7
8 int fib(int n, int[] memo) {
9     if (n <= 0) return 0;
10    else if (n == 1) return 1;
11    else if (memo[n] > 0) return memo[n];
12
13    memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
14    return memo[n];
15 }
```



Examples

Example 16

The following function prints the powers of 2 from 1 through n (inclusive). For example, if n is 4, it would print 1, 2, and 4. What is its runtime?

```
1  int powersOf2(int n) {
2      if (n < 1) {
3          return 0;
4      } else if (n == 1) {
5          System.out.println(1);
6          return 1;
7      } else {
8          int prev = powersOf2(n / 2);
9          int curr = prev * 2;
10         System.out.println(curr);
11         return curr;
12     }
13 }
```

Questions

