



# [0] Fundamentals



# Hello!

**We are Bobby,  
Geng, Ashka,  
Simran**

We're here because we love to teach

You can find us at {bobby.wang, geng.sng, ashka.stephen, simran.singh}@duke.edu





1

# What will we cover?

And how will we cover it?



# Three Core Units

## Basics

Strings, Arrays,  
Hashing, Binary Search,  
Big O

## Data Structures

Linked Lists, Queues,  
Stacks, Trees, Graphs

## Algorithms

Divide & Conquer,  
Greedy, Recursion,  
Backtracking, Dynamic  
Programming



# Three Key Methods

## Lectures

Hear us give a quick introduction to a concept, then watch us do a mock interview with that concept

## Practice Interviews

Pair up and take turns giving each other mock technical interviews

## Coding Competitions

Using the open-source Kattis platform [<https://open.kattis.com/>], compete against other classmates to be the first to solve all problems in a set.



# Every Tuesday

8-10 PM, Soc Sci 136

Lectures, Practice Interviews,  
Programming Competitions





## Am I ready to take this class?

Some “requirements”:

- ◇ CS201 level knowledge of basic data structures (string, array, hash table, linked list, tree, graph)
- ◇ Familiarity with Java or Python
- ◇ Laptop w/ internet access for programming challenges

Emails will be sent out before class every week regarding what prior knowledge is helpful!





# Basics

- ◇ 1/16: Fundamentals, Big O, Binary Search
- ◇ 1/23: Strings, Arrays, Hashing
- ◇ 1/30: Practice Interview & Coding Competition 1

Practice interviews are done in pairs w/ other students.

Coding competitions are done via







# Data Structures

- ◇ 2/6: Singly Linked Lists, Stacks
- ◇ 2/13: Doubly Linked Lists, Queues
- ◇ 2/20: Trees
- ◇ 2/27: Graphs
- ◇ 3/6: Practice Interview & Coding Competition 2






# Algorithms (Tentative)

- ◇ 3/20: Divide & Conquer, Greedy
- ◇ 3/27: Recursion, Backtracking, Intro DP
- ◇ 4/3: Advanced DP
- ◇ 4/10: Review
- ◇ 4/17: Final Coding Competition





# What we don't do

- ◇ Advanced data structures (tries, union-find, self-balancing trees, etc.)
  - ◇ Advanced runtime analysis (master theorem, amortized analysis, etc.)
  - ◇ Advanced graph related things (Dijkstra's, network flows, MST, etc.)
  - ◇ **Rigor** - we will state things without proof, and which are not 100% correct, but which are good enough for interviews.
- 



Poll: Python vs Java vs ??????



Now that's  
out of the  
way,  
let's jump into  
our first  
topic.



A decorative pattern of hexagons in various shades of blue and teal. Some hexagons contain icons: a lightbulb, a thumbs up, a network of nodes, a smartphone, a magnifying glass, a gear, and a speech bubble. A large, solid teal hexagon is positioned in the center-left, containing the number '2'.

2

Big O Complexity



If I increase the size of the input by  $n$  times, how does the runtime/required space change proportionally? Is it proportional to  $n$ ,  $n^2$ ,  $n^3$ ,  $\log(n)$ ,  $2^n$ ,  $n!$ , or does it not change at all?

This is the fundamental question that Big O (or asymptotic) complexity asks.



# Common Time Complexities

## $O(1)$

Arithmetic operations (including comparisons between #'s), accessing memory, getting an entry from a hashmap

## $O(\log n)$

Think “binary search”

## $O(n)$

Single for loop, or an operation which hits every

## $O(n \log n)$

Seen in divide and conquer sorting algorithms. Is a minimum bound in sorting.

## $O(n^2)$ , $O(n^3)$ , etc.

Pretty common with double, triple, etc. nested for loops

## $O(n!)$ and $O(x^n)$

Common in brute force algorithms where you try out every possible combination. Don't worry about these too much since your optimal solution most likely won't have times this high





# Big O Concepts

## Only the biggest term matters

Ex:  $n^4 + 10000 \cdot n^2$ .  
The second term will dominate for values of  $n < 100$ . However at  $n = 1$  million, the first term will outweigh the second by 100 million.

## Nesting multiplies big O's together.

Example: doing a binary search for each element of an  $n$  element array will take  $n \cdot \log n$  operations, so it is  $O(n \log n)$

## Drop the constants in front of your terms.

Big O is important ONLY when  $n$  is big. Like  $n \rightarrow \infty$  big. At that point, any constant, no matter how big or small, doesn't matter.

Ex:  $9999999999 \cdot n$  and  $n/999999999$  are both still  $O(n)$

# Intro

```
int sum1(int n) {  
    int tot = 0;  
    for (int i = 0; i < n; i++) {  
        tot += i;  
    }  
    return tot;  
}  
  
int sum2(int n) {  
    int tot = 0;  
    for (int i = 0; i < n; i++) {  
        for (int j = 0; j < i; j++) {  
            tot++;  
        }  
    }  
    return tot;  
}  
  
int sum3(int n) {  
    return n * (n-1) / 2;  
}
```

**O(n)**. You are doing  $O(1)$  arithmetic operations inside a loop which runs  $O(n)$  times. By multiplication rules,  $O(1) * O(n) = O(n)$

**O(n<sup>2</sup>)**. The outer loop runs  $O(n)$  times. The inner loop runs between 0 and  $n-1$  times, so the avg # of inner loops is still  $(n-1)/2 = O(n)$ . The inner loop does  $O(1)$  arithmetic operations each iteration. By multiplication rule,  $O(n) * O(n) * O(1) = O(n^2)$

**O(1)**. There are  $O(1)$  arithmetic operations which all take  $O(1)$  time.

# Med



```
int sum1(int n) {  
    int tot = 0;  
    for (int i = 0; i < n; i++) {  
        tot += i;  
    }  
    return tot;  
}
```

```
int loopSum(int n) {  
    int tot = 0;  
    for (int i = 0; i < n; i++) {  
        tot += sum1(n);  
    }  
}
```

```
int tripleSum(int n) {  
    int tot = 0;  
    for (int i = 0; i < 3; i++) {  
        tot += sum1(n);  
    }  
    return tot;  
}
```

```
int noSum(int n) {  
    int tot = 0;  
    for (int i = 0; i < 0; i++) {  
        tot += sum2(n);  
    }  
    return tot;  
}
```

**$O(n^2)$** . Outer loop runs  $O(n)$  times,  $\text{sum1}(n)$  runs in  $O(n)$  time. By multiplication rules,  $O(n) * O(n) = O(n^2)$ .

**$O(n)$** . Outer loop runs  $3 = O(1)$  times,  $\text{sum1}(n)$  runs in  $O(n)$  time. By multiplication rules,  $O(1) * O(n) = O(n)$ .

**$O(1)$** . Loop is never entered. Function returns 0 in constant time on every call.



# Hard

```
int sum2(int n) {  
    int tot = 0;  
    for (int i = 0; i < n; i++) {  
        for (int j = 0; j < i; j++) {  
            tot++;  
        }  
    }  
    return tot;  
}  
  
int sum3(int n) {  
    return n * (n-1) / 2;  
}
```

$O(n^4)$ . `sum3` runs in  $O(1)$  time, but the RETURN VALUE of `sum3` is  $O(n^2)$ . Thus, the outer loop runs  $O(n^2)$  times. Each time the loop runs, it calls `sum2`, which runs in  $O(n^2)$  time. Thus, our overall runtime is  $O(n^2) * O(n^2) = O(n^4)$ .

```
int challengeSum(int n) {  
    int tot = 0;  
    for (int i = 0; i < sum3(n); i++) {  
        tot += sum2(n);  
    }  
    return tot;  
}
```



# Big O Reminders

**Know the difference between average and worst case runtime (and what conditions will cause those times)**

Ex: Quicksort is  $O(n \log n)$  average case,  $O(n^2)$  worst case.

NB: It's NOT the same as the difference between Big Omega and Big O

**Remember basic logarithm/series rules**

Especially:

- $\log_b(n) = O(\log n)$  for all  $b > 1$
- Sum of a geometric series is finite

**Know what you're doing big O in relation to**

For example, an algorithm which sums together every element of an  $m \times n$  matrix is  $O(a)$  if  $a$  is the total # of elements of a matrix, but it is  $O(mn)$  if we consider the number of rows and cols themselves

# Intro

```
int product1(int a, int b) {  
    int sum = 0;  
    for (int i = 0; i < b; i++) {  
        sum += a;  
    }  
    return sum;  
}  
  
int product2(int a, int b) {  
    int sum = 0;  
    for (int i = 0; i < a; i++) {  
        sum += b;  
    }  
    return sum;  
}
```

**O(b)**. The outer loop runs  $O(b)$  times, and  $O(1)$  arithmetic operations are done inside each loop iteration.

**O(a)**. The outer loop runs  $O(a)$  times, and  $O(1)$  arithmetic operations are done inside each loop iteration.



# Med

```
int binarySearch(int[] a, int target) {  
    int lo = 0;  
    int hi = a.length - 1;  
    while (lo < hi) {  
        int mid = lo + (hi - lo)/2;  
        int val = a[mid];  
        if (val == target) {  
            return mid;  
        } else if (val < target) {  
            lo = mid + 1;  
        } else {  
            hi = mid - 1;  
        }  
    }  
    return -1;  
}
```

**$O(\log n)$** . At every iteration of the while loop, an  $O(1)$  number of arithmetic operations and variable assignments are being done. The difference between lo and hi is cut in half every loop iteration, which means that eventually  $lo = hi$ . This will happen in  $\log_2(n) = O(\log n)$  steps, which is the maximum number of times that the loop will be entered. By multiplication rules,  $O(\log n) * O(1) = O(\log n)$

# Single Recursive Calls

Think of 2 things, if there is only one recursive call:

- ◇ How much work is being done in the non-recursive part of the function
- ◇ How many recursive calls it takes to get to a base case

Total work = (non recursive work) x (# of calls needed needed to reach base case)





# Intro

```
int power(int a, int b) {  
    if (b <= 0) {  
        return 1;  
    }  
    return a * power(a, b-1);  
}
```

```
int mod(int a, int b) {  
    if (b <= 0) {  
        return -1;  
    }  
    return a - (a/b) * b;  
}
```

$O(b)$ . The problem size shrinks by 1 each time, so we will hit a base case after  $O(b)$  calls. There is  $O(1)$  recursive work being done in the non-recursive part of the function, so  $O(b)$

$O(1)$ . This is NOT a recursive function. It just does  $O(1)$  arithmetic operations.

# Multiple Recursive Calls

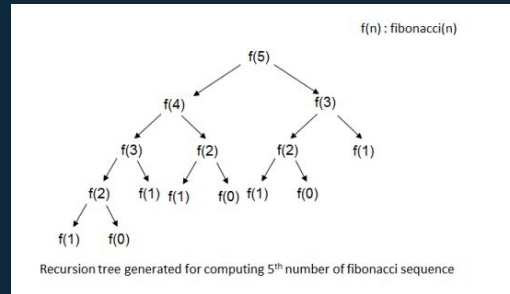
If there are multiple recursive calls per function call, then analysis is harder. The following 2 rules will cover many common situations, though.

```
int fib(int n) {  
    if (n < 0) {  
        return 0;  
    }  
    if (n < 2) {  
        return n;  
    }  
    return fib(n-1) + fib(n-2);  
}
```



# Hard

```
int fib(int n) {
    if (n < 0) {
        return 0;
    }
    if (n < 2) {
        return n;
    }
    return fib(n-1) + fib(n-2);
}
```



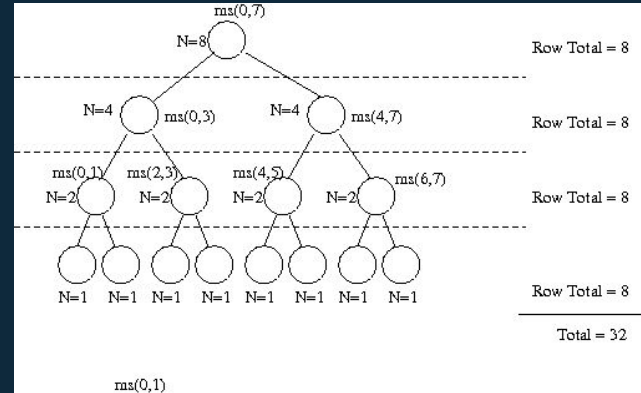
**Rule 1:** If input size decreases linearly, runtime =  $O(f * x^d)$

- $x$  = # of subcalls per function call
- $f$  = non-recursive work done per call
- $d$  = # of calls needed to hit base case (aka depth of recursion tree)

With the example here,  $x=2$ .  $f=O(1)$ ,  $d = n$ . So we get  $O(2^n)$ .

**Rule 2:** If input size decreases by the inverse of the amount of recursive calls, then the runtime =  $O(f(n) * \log n)$  where  $f(n)$  is the amount of work done at each tree “level”, and  $n$  = problem size.

Ex: mergesort. Input size decreases by a factor of 2 by each level, and each call to mergesort has two recursive calls. The work at each level remains  $O(n)$ , and the tree goes down  $O(\log n)$  levels before the calls hit the base case of a subarray of only one element. Total work =  $O(n \log n)$ .



# Hard



# Multiple Recursive Calls Caveats

**Functions with multiple recursive calls are VERY different than functions with one recursive call**

If your recursion diagram looks like a linked list and not a tree, don't try to use the rules on the previous 2 slides.


**This is an advanced topic, and we only cover generalities**

Please take CS330 and read up on master theorem for a more correct treatment of (a subset of) this difficult topic.

Ex: naive recursive fibonacci has a tighter bound of  $O(1.618^n)$ , rather than  $O(2^n)$ .

**Memorization can often work best here**

You'll rarely be asked to write a recursive function with multiple calls that you've never seen before. Most of the time it'll be implementing something common like fibonacci or quicksort, so memorizing common runtimes may be your best strategy.



# But what about space complexity?

## Usually Easier to Reason About

In iterative cases, think of “extra space” as how many things you’re stuffing in a memoization array or hashmap versus how many things you’re given as input.


## Don’t include the size of your input in space complexity

Ex: bubble sort can be done in place to sort in  $O(1)$  space. We don’t include the  $n$  element return array because that’s the SAME array as our input.

## DO include call stack space

Every recursive call takes up space in the call stack. Thus, the total amount of “extra space” is the max recursive call depth.

Ex: quicksort doesn’t use an auxiliary array, but it has an average call depth of  $\log(n)$ . Thus its extra space is  $O(\log n)$  average case.




## Side Note: What “Constant” Means

```
int sum1(int n) {  
    int tot = 0;  
    for (int i = 0; i < n; i++) {  
        tot += i;  
    }  
    return tot;  
}
```

sum1 could be considered to have an  $O(1)$  runtime in languages like Java, since Java int has a max size of  $2^{31}-1 = O(1)$ . Thus, the maximum input size and thus runtime is technically bounded.

However, not many interviewers would claim that this is  $O(1)$  since the constant is pretty big at over 4 billion.




# Side Note: What “Constant” Means

## HOWEVER

In terms of space complexity, this becomes somewhat more important. Say that you're storing a hashmap of character counts, for example, and the interviewer says that your input is an ascii string. There are only 256 =  $O(1)$  different Ascii characters. Thus, you could use an `int[256]` to store all your character counts, which is an  $O(1)$  extra space complexity solution. This is more reasonable since the constant space is smaller.





That was a  
lot to think  
about.

Let's take a  
break and  
come back in  
10 minutes.





# Big O Big Ideas

## **O(1) operations**

Memorize them: they are the “base” of every function. memory access, hashset access, math, comparison, bit manipulation.

## **Basic Concepts**

Ignoring constants, keeping biggest term, multiplying nested complexities.

## **Multiple Variables**

Make sure you are clear in defining what the “n” means in  $O(n)$ .

## **Single Recursion**

Figure out how many calls it takes to get to a base case and multiply by the amount of work done in just one call.

## **Multiple Recursion**

A difficult topic that can usually be answered with “exponential time” or “ $n \log n$ ”

## **Space Complexity**

Includes: extra variables explicitly being declared, # of recursive calls implicitly being run

Does not include: input variables themselves

A decorative pattern of hexagons in various shades of blue and teal. Some hexagons contain icons: a lightbulb, a thumbs up, a network node, a smartphone, a magnifying glass, a gear, and a speech bubble. A large, bright teal hexagon in the center-left contains the number 3.

3

# Binary Search



Less than 10% of professional programmers are able to code a bug-free binary search. Why?

<https://reprog.wordpress.com/2010/04/19/are-you-one-of-the-10-percent/>



# The Idea

You don't know the definition of "Catalyst" and want to look it up in a dictionary. How can you do this quickly?

Slow way: flip forward one page at a time until you find the word

Faster way:

- Flip to the middle of the dictionary. The words start with "j" on that page.
- Flip to a page halfway between the "j" page and the beginning. The words start with "f" on that page.
- Flip to a page halfway between the "f" page and the beginning. The words start with "b" on that page.
- Flip to a page halfway between the "b" page and the "f" page. The words on that page start with "ce..."
- Flip to a page halfway between the "b" page and the "ce..." page. The words on that page start with "ca..." And on that page, you find "catalyst."



# The Code

```
int binarySearch(int[] a, int target) {  
    int lo = 0;  
    int hi = a.length - 1;  
    while (lo < hi) {  
        int mid = lo + (hi - lo)/2;  
        int val = a[mid];  
        if (val == target) {  
            return mid;  
        } else if (val < target) {  
            lo = mid + 1;  
        } else {  
            hi = mid - 1;  
        }  
    }  
    return -1;  
}
```

Iterative

```
int binarySearch(int[] a, int target, int lo, int hi) {  
    if (hi < lo) {  
        return -1;  
    }  
    int mid = lo + (hi - lo)/2;  
    int val = a[mid];  
    if (val == target) {  
        return mid;  
    } else if (val < target) {  
        return binarySearch(a, target, mid + 1, hi);  
    } else {  
        return binarySearch(a, target, lo, mid - 1);  
    }  
}
```

Recursive



# Why is binary search hard?

## Off By One Errors

It's hard to remember in an interview if there's a  $+1$  or  $-1$  or neither, if it's  $\leq$  or just  $<$ , whether to return mid or hi or lo, etc.

My solution: code binary search the same every time.

## Disguised Problems

Not all binary search problems will initially seem like a binary search problem. See the dice example to follow.

## Icarus Syndrome

Because the concept of binary search is more intuitive for many people than dynamic programming or bit manipulation, they may choose to skip studying for this topic in interviews, thinking that they'll remember the details when prompted.



# Intro

## “Bug Hunting”

You're at your awesome internship at facegoogsoftzon and one day, you find that the codebase has a bug! You want to find which version number the bug first appeared in. Your version numbers are consecutive integers from 1 to  $n$ , where  $n$  is the current version.

You have a function, `isGood(x)` that will tell you whether a certain version has the bug. However, `isGood()` takes a LONG time to run (it re-compiles your entire codebase so each call takes several hours!!) so you want to minimize the # of calls to `isGood()`. Write a function that will find the first version with the bug.





# Interviewing Tactics I

## Testing 1,2,3

Even on examples like this, where you're not given an explicit test input, come up with your own.

For sure know how to write assert statements and a short test suite for your code. If all else fails, write print statements.

## Language Familiarity

Related to testing: make sure that you know how to compile/run your code in that particular IDE. In particular, Java's weird with the static keyword. Don't get stuck with not knowing how to do text input/output, either.

## Talk It Through

Make sure you understand your problem completely before starting to code - it's very time consuming and stressful to retype your code.



Med

## “Peak Performance”

You work for the astronomy department in a 2d universe and are given a `double[] heights`, where `heights[i]` is the height of the terrain at point `i`. Your goal is to find a place for the astronomy department to build their new telescope. This place should be at a peak - that is, an `i` such that `heights[i - 1] < heights[i] > heights[i + 1]`. There may be multiple peaks - just pick any one of them.



# Interviewing Tactics II

## Twisting the Problem

Many problems will be variations of the problems you see in this class, but the core algorithm will remain the same. When in doubt, you should brainstorm how you can apply classic CS concepts to an unfamiliar problem

## Attention to Detail

Be especially careful when you're doing math with doubles and integers in the same problem. This could lead to some very hard to find bugs down the road. Similarly, do a quick mental check to make sure that all variable types are what they're supposed to be - especially in duck typed languages like Python.



## “Don’t Tell Me The Odds”



I give you an array of doubles `pdf[]` that represent the pdf of a random number generator. Each `pdf[i]` represents the chance of generating the integer `i`. For example,  $[\frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}]$  would represent a fair six-sided dice, and  $[0.5, 0.5]$  would represent a fair coin. If `pdf = [0.1, 0.2, 0.3, 0.4]`, then 0 would show with probability 0.1, 1 would show with probability 0.2, etc.

Given a `double[]` and a function `rand()` which generates a random float between 0 and 1, write a function `generate()` that will return an int based on the inputted pdf. In addition, each generator is initialized once but will be rolled billions of times, so preprocessing is OK as long as it reduces the runtime of `generate()`.



# Interviewing Tactics III

## Trade-offs

There's often no solution that will minimize both space and time complexity all the time. Talk with your interviewer about how preprocessing, adding an auxiliary array/hashmap, etc. can be beneficial, and also how it may detract. This is a great chance to show what you know.

## Helper Methods/Classes

Oftentimes, as problems get more complex, you'll need to write helper functions, or even define a custom class to solve a problem effectively. Code should be as simple as possible, so make sure you verbally justify to your interviewer why you're adding more complex code.





# When NOT to use Binary Search

## When element access isn't $O(1)$

Ex: don't binary search a sorted linked list, since accessing an element in the middle of a linked list is  $O(n)$ . You may find your element after  $O(\log n)$  queries, but each query takes  $O(n)$  time, so binary search would take  $O(n \log n)$  time, slower than linear search.

## Multidimensional Arrays

Unless you can easily convert a 2d or 3d sorted array into a 1d sorted array, binary search will NOT work.

Ex: peak finding in 2d, searching a column and row wise sorted matrix





# Thanks!

## (M)Any questions?

Special thanks to all the people who made and released these awesome resources for free:

- ◇ Presentation template by [SlidesCarnival](#)
- ◇ Photographs by [Unsplash](#)

This template is free to use under [Creative Commons Attribution license](#).

You can find us at: {

bobby.wang,  
geng.sng,  
ashka.stephen,  
simran.singh

}@duke.edu

