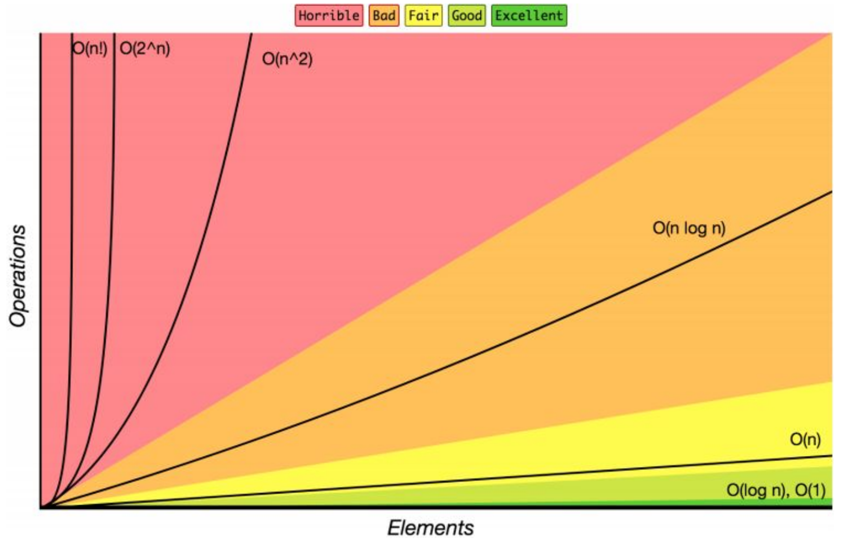
Week 6 – Asymptotic Analysis

Review: Algorithms Complexity Class

complexity class: A category of algorithm efficiency based on the algorithm's relationship to the input size N.

Complexity Class	Big-O	Runtime if you double N	Example Algorithm
constant	O(1)	unchanged	Accessing an index of an array
logarithmic	$O(\log_2 N)$	increases slightly	Binary search
linear	O(N)	doubles	Looping over an array
log-linear	$O(N \log_2 N)$	slightly more than doubles	Merge sort algorithm
quadratic	$O(N^2)$	quadruples	Nested loops!
…	…	…	…
exponential	$O(2^N)$	multiplies drastically	Fibonacci with recursion

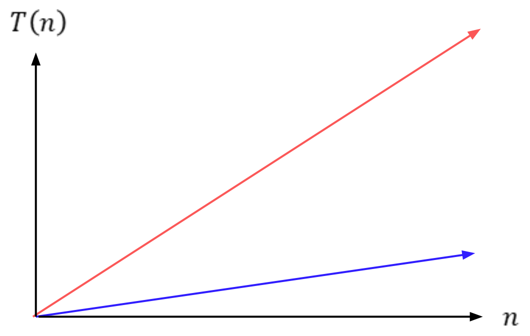


https://www.bigocheatsheet.com/

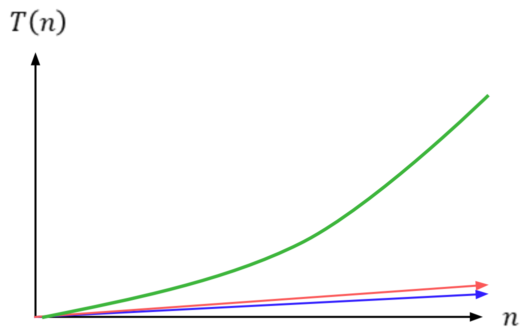
Function growth

Imagine you have three possible algorithms to choose between.
Each has already been reduced to its mathematical model

$f(n) = n$ $g(n) = 4n$ $h(n) = n^2$

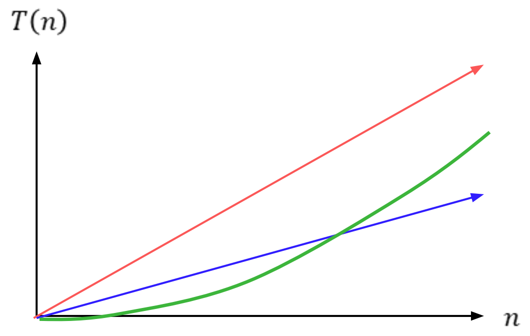


The growth rate for f(n) and g(n) looks very different for small numbers of input



...but since both are linear eventually look similar at large input sizes

whereas h(n) has a distinctly different growth rate



But for very small input values h(n) actually has a slower growth rate than either f(n) or g(n)

Definition: Big-O

We wanted to find an upper bound on our algorithm's running time, but

- We don't want to care about constant factors.
- We only care about what happens as n gets large.

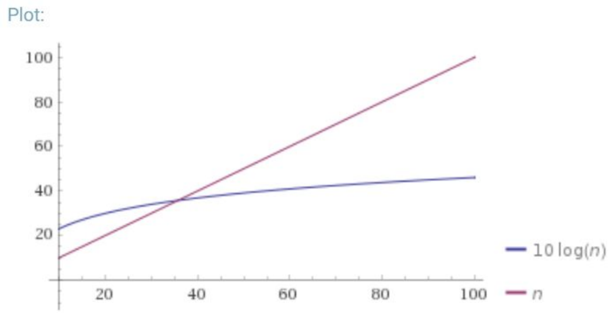
Big-O

$f(n)$ is $O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,

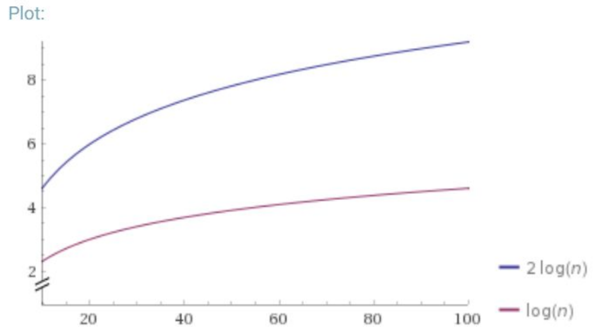
$$f(n) \leq c \cdot g(n)$$

We also say that $g(n)$ "dominates" $f(n)$

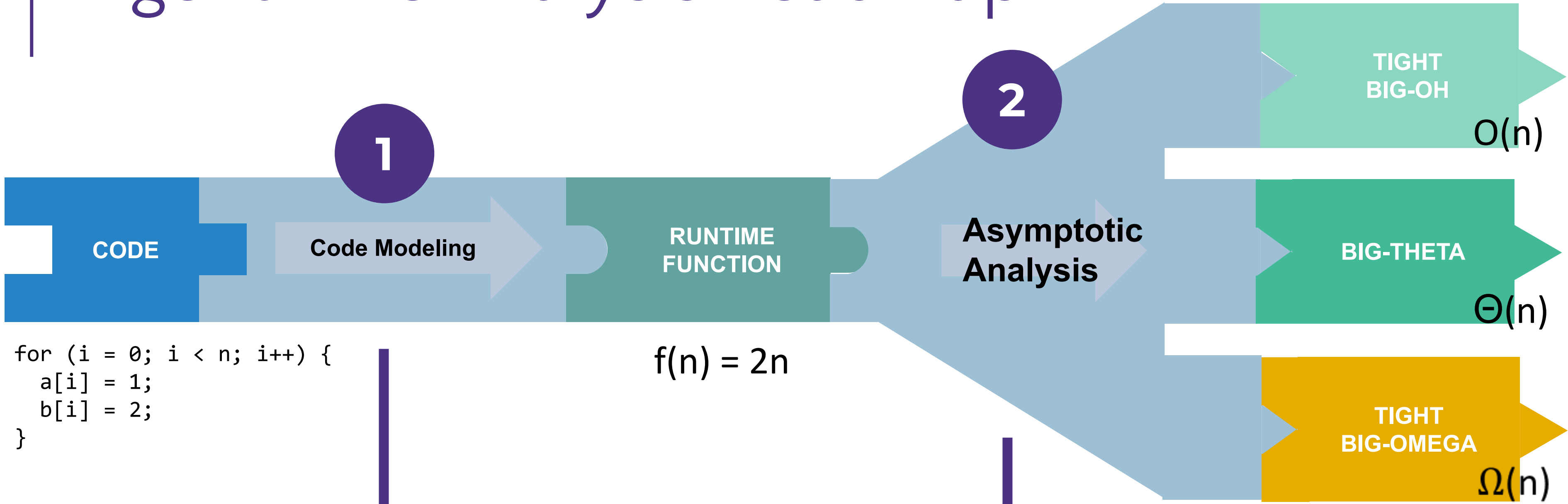
Why n_0?



Why c?



Algorithmic Analysis Roadmap



```
for (i = 0; i < n; i++) {
  a[i] = 1;
  b[i] = 2;
}
```

$f(n) = 2n$

Now, let's look at this tool in more depth. How exactly are we coming up with that function?

We just finished building this tool to characterize a function in terms of some useful bounds!

Review: Data Structures Complexity Class – **Access at Index Operation**

Data Structure	Average Case Runtime	Worst Case Runtime	Space Complexity
Array	O(1)	O(1)	O(N)
Stack	O(N)	O(N)	O(N)
Queue	O(N)	O(N)	O(N)
Linked-List	O(N)	O(N)	O(N)
HashTable	N/A	N/A	O(N)
Binary Search Tree	O(logN)	O(N)	O(N)
AVL Tree	O(logN)	O(logN)	O(N)

https://www.bigocheatsheet.com/

Review: Data Structures Complexity Class – **Search for Entry Operation**

Data Structure	Average Case Runtime	Worst Case Runtime	Space Complexity
Array	O(N)	O(N)	O(N)
Stack	O(N)	O(N)	O(N)
Queue	O(N)	O(N)	O(N)
Linked-List	O(N)	O(N)	O(N)
HashTable	O(1)	O(N)	O(N)
Binary Search Tree	O(logN)	O(N)	O(N)
AVL Tree	O(logN)	O(logN)	O(N)

https://www.bigocheatsheet.com/

Review: Data Structures Complexity Class – **Insertion new Entry Operation**

Data Structure	Average Case Runtime	Worst Case Runtime	Space Complexity
Array	O(N)	O(N)	O(N)
Stack	O(1)	O(1)	O(N)
Queue	O(1)	O(1)	O(N)
Linked-List	O(1)	O(1)	O(N)
HashTable	O(1)	O(N)	O(N)
Binary Search Tree	O(logN)	O(N)	O(N)
AVL Tree	O(logN)	O(logN)	O(N)

https://www.bigocheatsheet.com/

Review: Data Structures Complexity Class – **Deletion Operation**

Data Structure	Average Case Runtime	Worst Case Runtime	Space Complexity
Array	O(N)	O(N)	O(N)
Stack	O(1)	O(1)	O(N)
Queue	O(1)	O(1)	O(N)
Linked-List	O(1)	O(1)	O(N)
HashTable	O(1)	O(N)	O(N)
Binary Search Tree	O(logN)	O(N)	O(N)
AVL Tree	O(logN)	O(logN)	O(N)

https://www.bigoc

Big-O Patterns to look out for

- `for (int i = 0; i < n; i++)` = single for-loop with respect to input n
  - O(n)
- `for (int i = 0; i < n; i++)`

  `    for (int j = 0; j < i; j++)` = double for-loop with respect to input n
  - O(n^2)
- traversing through trees
  - O(logN) for balanced trees and O(N) in the worst case for non-balanced trees (i.e. BSTs)
- Traversing through graphs using DFS/BFS
  - O(V+E) where V = vertices/nodes and E = edges/links
  - Notice Graphs have 2 input values with respect to our runtime, edges and vertices instead of just using one variable N

Scenario #1

You are going to Disneyland for spring break! You've never been, so you want to make sure you hit ALL the rides.

Is there a graph algorithm that would help?

BFS or DFS

How would you draw the graph?

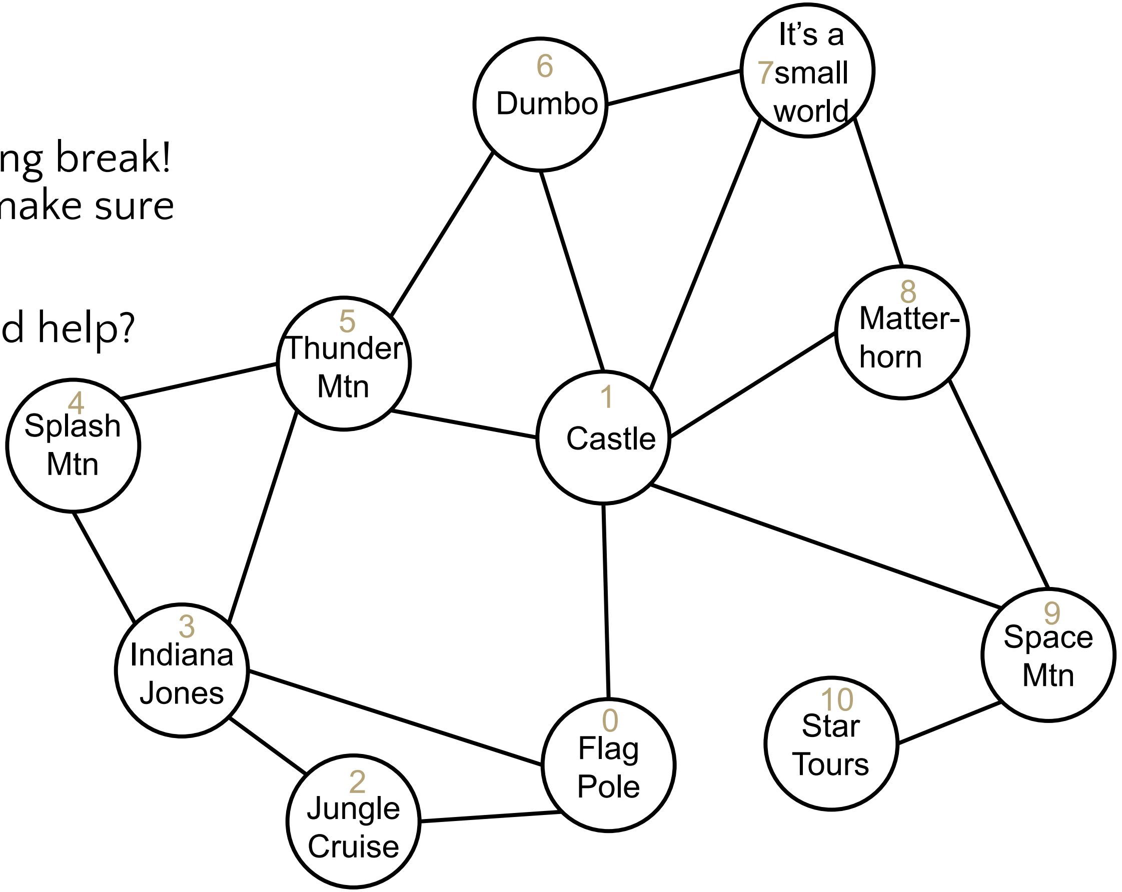
- What are the vertices?

Rides

- What are the edges?

Walkways

BFS = 0 1 2 3 5 6 7 8 9 4 10
DFS = 0 3 5 6 7 8 9 10 1 4 2



Scenario #1 continued

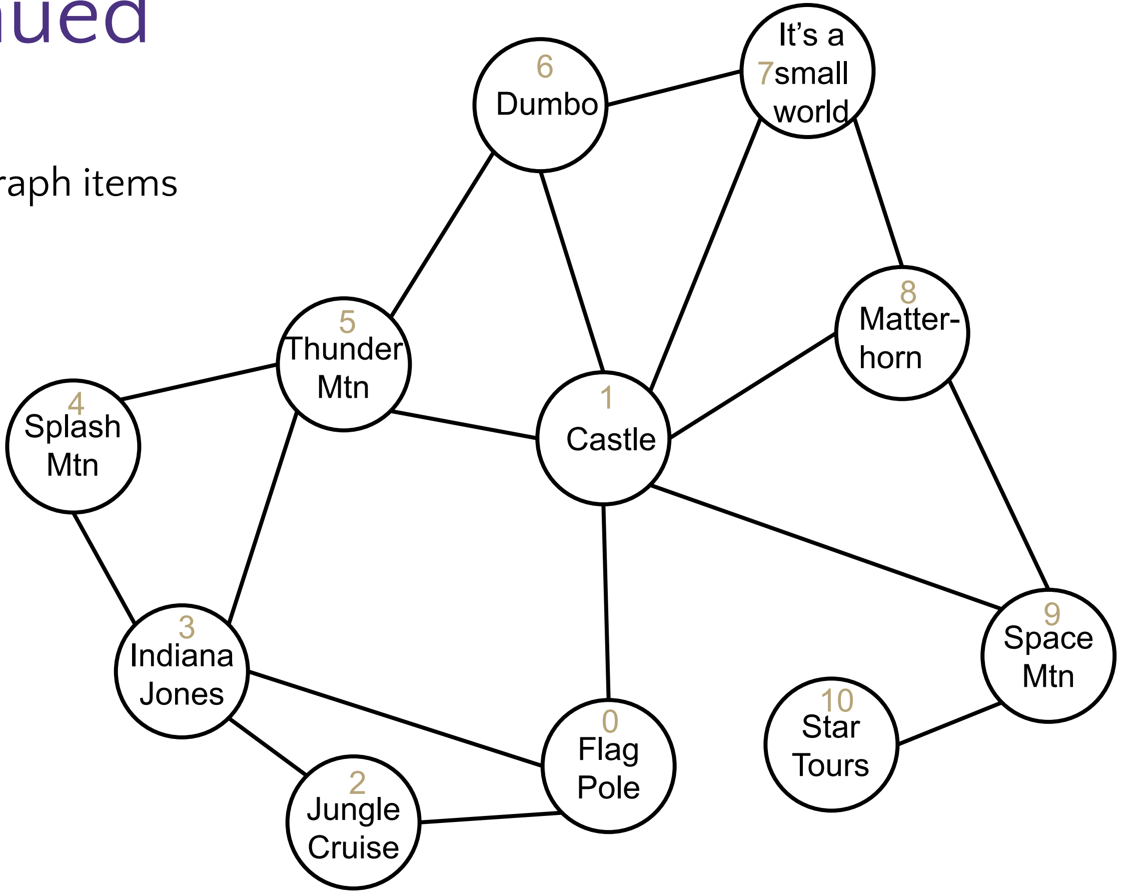
Now that you have your basic graph of Disneyland what might the following graph items represent in this context?

Weighted edges
- Walkway distances
- Walking times
- Foot traffic

Directed edges
- Entrances and exits
- Crowd control for fireworks
- Parade routes

Self Loops
- Looping a ride

Parallel Edges
- Foot traffic at different times of day
- Walkways and train routes

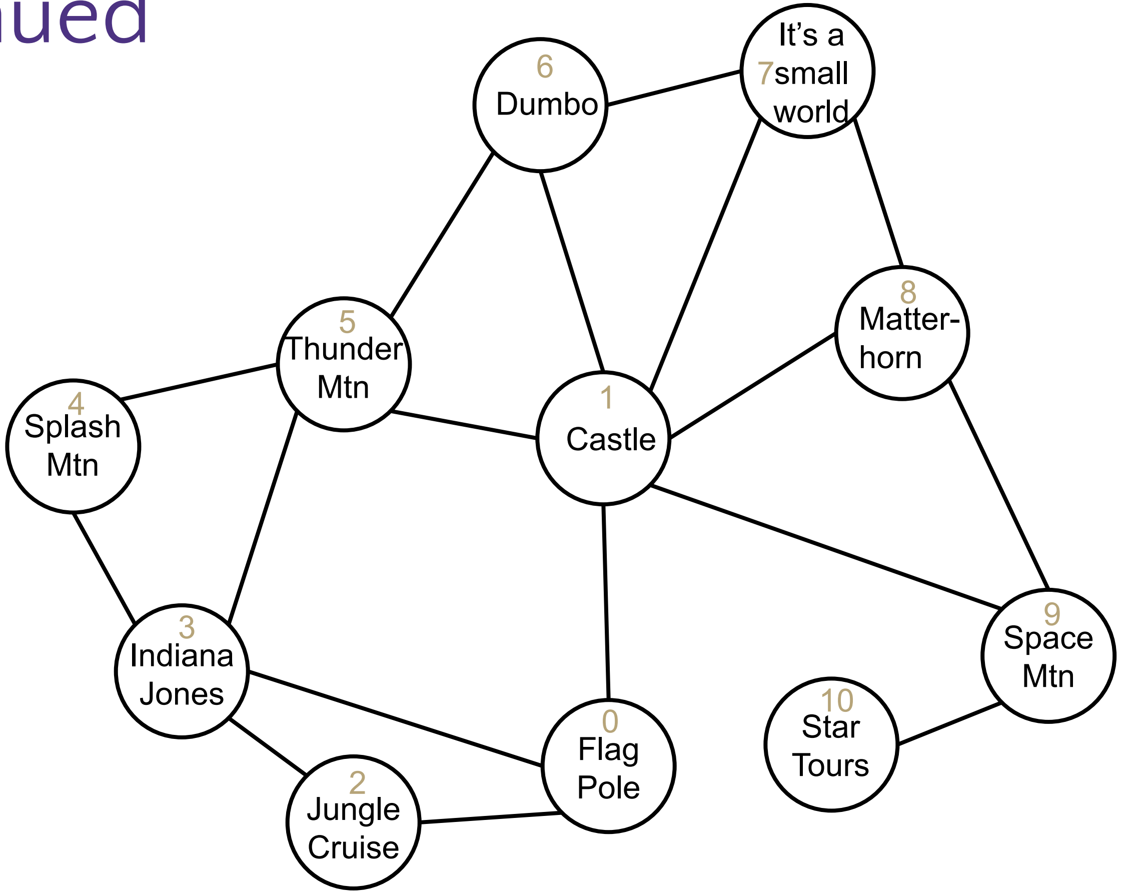


Scenario #1 continued

What would be the runtime of going through every ride using a BFS/DFS algorithm?

- We start at a starting Node (Castle)
- Go through every unvisited node that is our current neighbor through a connected edge
- Continue until we have no undiscovered nodes left in our graph

O(V+E)



Scenario #2

You have a list (of length n) of lists with all the same length of m containing different letters to make up different words of length m. (Ex. list of lists on right)

You want to find out the total count of a certain input character (in this case, we'll say it is the letter 'w') across every character of your list of lists.

What is the total runtime in terms of n and m for finding every count of 'w' and returning the count?

[m, o, o]	[c, o, w]	[m, e, w]	[c, a, t]

n = 4
m = 3

O(m+n)

Let’s go through a problem together…

<u>Infamous Two-Sum Problem:</u>

Given an array of integers `nums` and an integer `target`, return indices of the two numbers such that they add up to `target`.

You may assume that each input would have <u>exactly one solution</u>, and you may not use the *same* element twice.

You can return the answer in any order.

Ex.

Input: `nums = [2,7,11,15]; target = 9`

Output: `[0,1]`

Explanation: `Because nums[0] + nums[1] == 9, we return [0, 1].`

Clarify & Examples

Do we need to return every possible solution or only one?

- Typically you can assume only one solution exists

Can you use the same element twice in the sum?

- No

Input: nums = `[1,4,10,-3]`, target = `14`

Output: `[1,2]` or `[2,1] # 4 + 10 = 14`

Input: nums = `[9,5,1,23]`, target = `10`

Output: `[0,2]` or `[2,0] # 9 + 1 = 10`

Input: nums = `[1,-2,5,10]`, target = `-1`

Output: `[0,1]` or `[1,0] # 1 + -2 = -1`

Approach

Brute Force: Use two nested for-loops so that for every element we check every other element to see if they both add up to the target value.

Code:

```
for (int i = 0; i < nums.length; i++) {
    for (int j = i + 1; j < nums.length; j++) {
        if (nums[i] + nums[j] == target) {
            return new int[] {i, j};
        }
    }
}
```

Runtime: ?

O(N^2)

Is there a faster way?

Optimize

Optimized Solution: Use a HashMap that stores the value we need somewhere later in our list (starting from our current value) to hit the target value as the key, and stores the index of our two values as the values.

Code:

```
    HashMap<Integer, Integer> need = new HashMap<>();
    for (int i = 0; i < nums.length; i++) {
         if (!need.contains(nums[i])) {
              need.put(target-nums[i], i);
         } else {
              return new int[] {need.get(nums[i]), i}
  }
```

Runtime: ?

O(N)

Space Complexity: O(N)

Practice Problems of the Week

Go through each problem and talk through your solution(s) runtimes. See if you can find a more optimal solution for each problem than your initial solution.

https://leetcode.com/problems/add-two-numbers

https://leetcode.com/problems/median-of-two-sorted-arrays

https://leetcode.com/problems/longest-palindromic-substring

https://leetcode.com/problems/3sum

https://leetcode.com/problems/valid-parentheses

https://leetcode.com/problems/merge-k-sorted-lists

https://leetcode.com/problems/remove-duplicates-from-sorted-array

https://leetcode.com/problems/remove-element