Data Structures and Algorithms

But First…Time and Space Complexity

- Time complexity: The time it takes for an algorithm to run, given a certain input (n)
- Space complexity: The amount of memory space used by an algorithm/program, including the space of input values for execution (n)

→ IMPORTANCE: Time and memory are critical to make programming efficient, and often memory has an inverse relationship with time (worse memory → better time complexity)

Measuring Runtime Performance

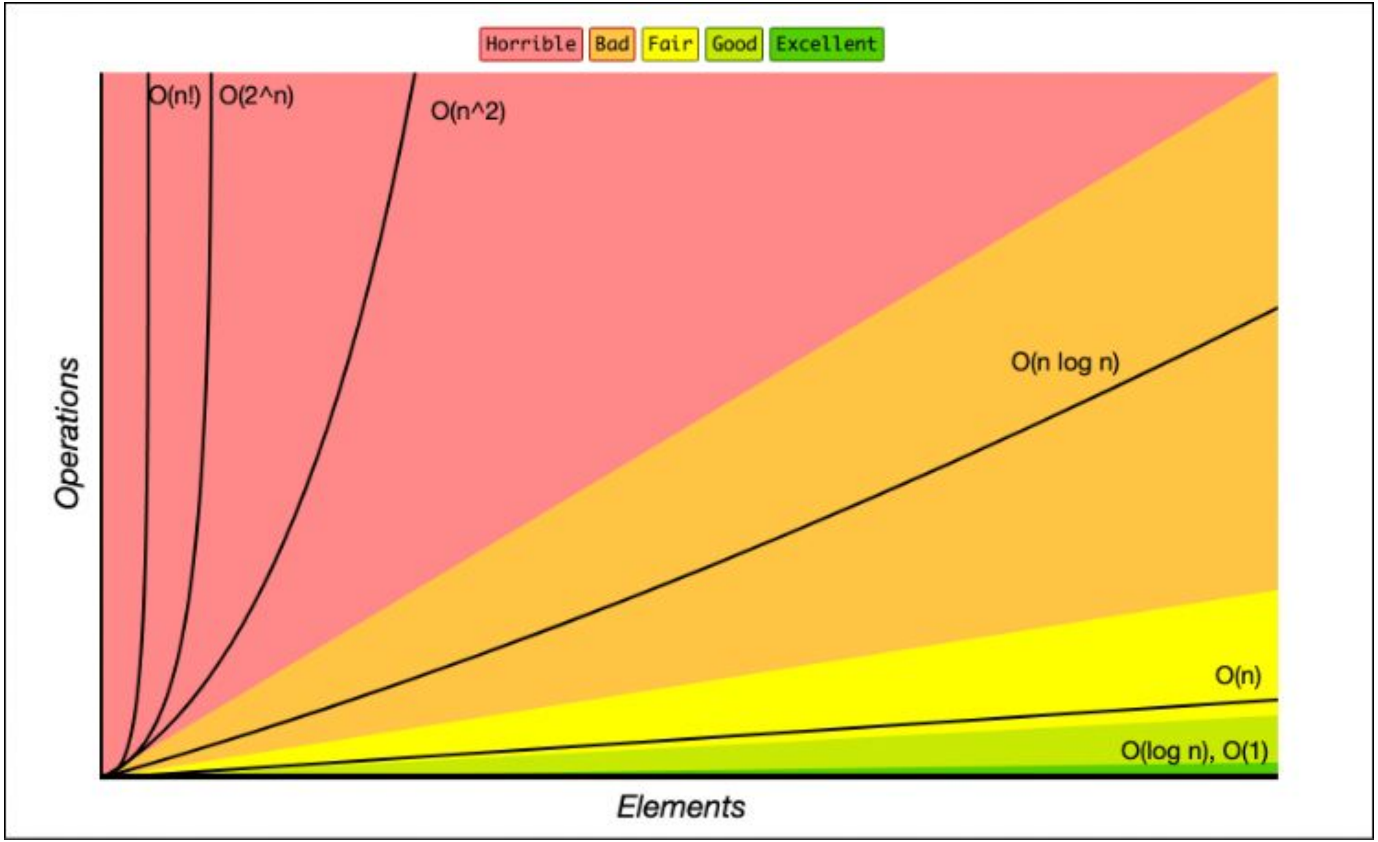
O-notation: The upper-bound of an algorithm (the worst case scenario)

Θ-notation: The tight-bound of an algorithm (the average case scenario)

Ω-notation: The lower-bound of an algorithm (the best case scenario)

→ A lot of times when people talk about $O(n)$, $O(n^2)$, they are actually talking about Θ

Time Complexity Chart



Time Complexity Further Explained

What does O(n), O(n^2), O(logn)... really mean?

What runtime complexity means is the relationship between how slow/fast our program runs in respect to the input size

Example:

O(n) means that the runtime complexity of our program has a linear relationship with relationship to our input size

- If we were to increase our data size by 2 times, then our runtime complexity of our program would also be increased by 2 (showing a linear relationship)

Question: What does O(1) mean?

Measuring Runtime Performance

EX 1:

```
print('Hello')
```

EX 2:

```
X = 'Hello'
print(x)
```

Measuring Runtime Performance

EX 3:

```
if a > b:
    return True
else:
    return False
```

EX 4:

```
for value in data:
    print(value)
```

Measuring Runtime Performance

EX 5:

```
for index in range(0, len(data), 3):
    print(data[index])
```

EX 6:

BINARY SEARCH (https://www.geeksforgeeks.org/binary-search/)

Measuring Runtime Performance

EX 7:

```
For x in data:
    For y in data:
        print(‘Hello’)
```

EX 8:

```
For x in data:
    print(‘Hello’)
For y data:
    For z in data:
        Print(‘Goodbye”)
```

What are Data Structures and Algorithms?

A data structure is a container that holds a collection of items, used for data organization, management, and efficient access

- Example: Arrays/Vectors

An algorithm is a procedure used for solving a problem

- Example: Cooking Recipe

Python Dictionaries (Hash Tables)

A dictionary is a collection of key-value pairs (Associative data structure)

- Maps the key to its associated value

Why is it so powerful?

- Dictionaries are very good at finding an element → O(1)
- Typically when a question is based around searching, a dictionary is the way to go

NOTE: This is an example of sacrificing memory for runtime (Hash tables use an excess of memory - unused space)

Dictionary Syntax

Creating a dictionary:

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
```

Returning each item in a dictionary:

```
x = thisdict.items()
```

Dictionary Syntax

Checking if a key exists:

```
if "model" in thisdict:
    print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

Inserting a new key-value pair:

dictName[KEY] = VALUE

→ KEY THING TO NOTE:

DO NOT CHECK IF A VALUE EXISTS BY USING THIS METHOD (BY DEFAULT, A DICTIONARY WILL CREATE A NEW KEY-VALUE PAIR IN THE DICTIONARY)

How to make an empty dictionary:

dictName = {}

Contains Duplicate Example

https://leetcode.com/problems/contains-duplicate/description/

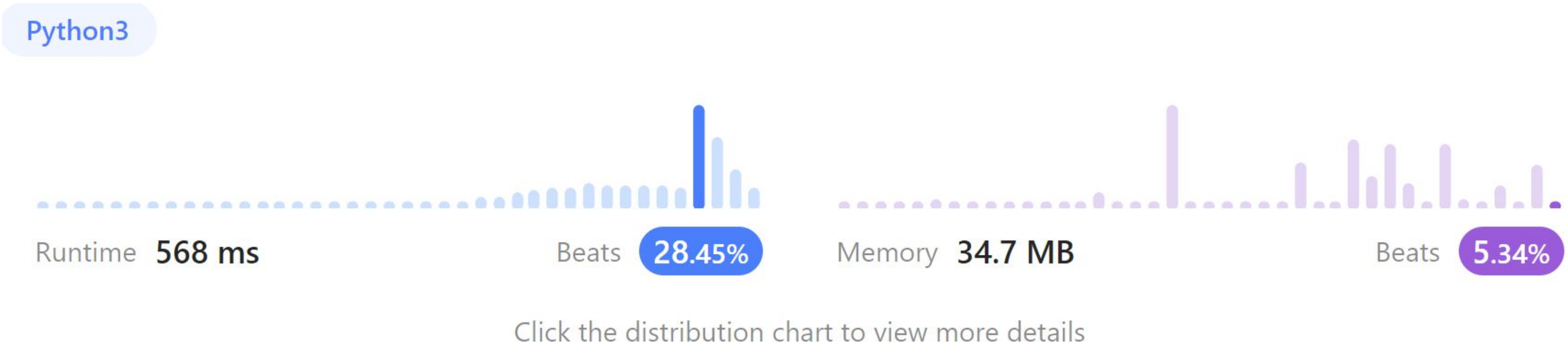
Contains Duplicate Hash Table Solution

```
class Solution:
    def containsDuplicate(self, nums: List[int]) -> bool:
        hashTable = {}
        for x in nums:
            if x in hashTable:
                return True
            else:
                hashTable[x] = x
        return False
```

Runtime?

KEYNOTE: Realistically, a set would be better (a set is like a hash table, but its actual value is used as its key)

Hash Table Runtime and Memory

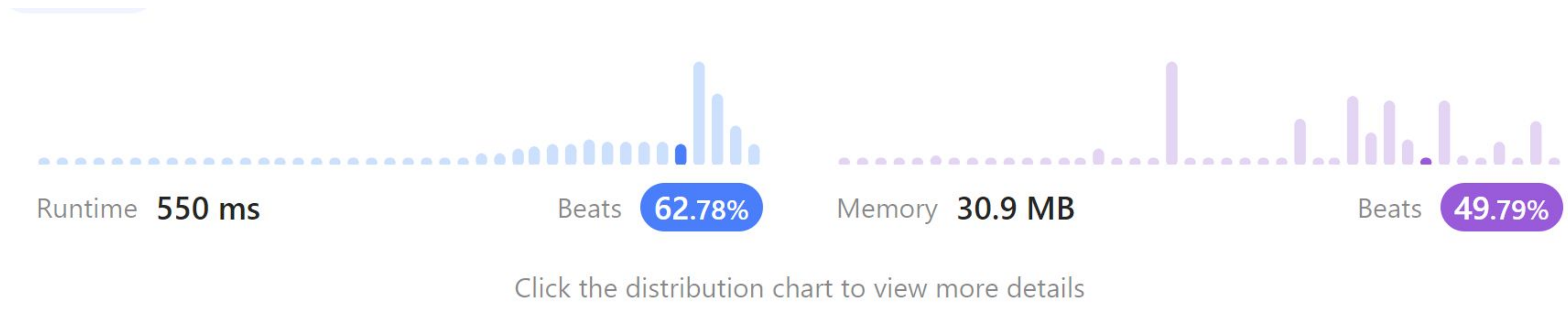


Contains Duplicate Set Solution

```
class Solution:
    def containsDuplicate(self, nums: List[int]) -> bool:
        hashset = set()
        for x in nums:
            if x in hashset:
                return True
            else:
                hashset.add(x)
        return False
```

Runtime?

Set Runtime and Memory

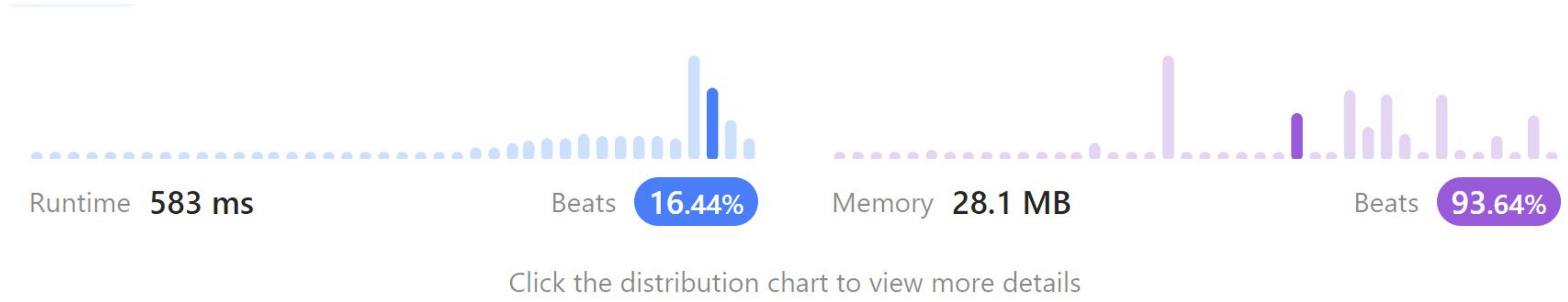


Contains Duplicate Array/Vector Solution

```
class Solution:
    def containsDuplicate(self, nums: List[int]) -> bool:
        nums.sort()
        for i in range(len(nums)-1):
            if nums[i] == nums[i+1]:
                return True
        return False
```

Runtime?

Array/Vector Runtime and Memory



Contains Duplicate Another Array/Vector Solution

```
class Solution:
    def containsDuplicate(self, nums: List[int]) -> bool:
        for i in range(len(nums)):
            for j in range(len(nums)):
                if nums[i] == nums[j] and i != j:
                    return True
        return False
```

Runtime?

Another example to look at…Two Sum

https://leetcode.com/problems/two-sum/description/

Two Sum Solution

```
class Solution:

    def twoSum(self, nums: List[int], target: int) -> List[int]:

        values = {}

        for idx, value in enumerate(nums):

            if target - value in values:

                return [values[target - value], idx]

            else:

                values[value] = idx
```

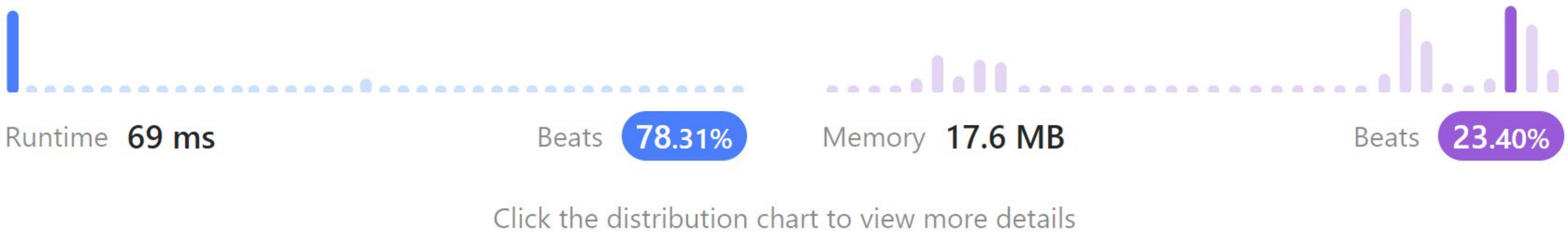
Runtime?

KEYNOTE: Enumerate is used to retrieve the position index and corresponding value (idx is the index and value is the corresponding value while iterating through the list)

https://www.askpython.com/python/built-in-methods/for-loop-two-variables

(Look for “Example 2: “**for**” Loop using **enumerate()** function”)

Two Sum Runtime and Memory



Leetcode Problems to Practice (Labeled ‘easy’)

Contains Duplicate

Valid Anagram

Two Sum

Sign of the Product of an Array (General Array/Vector problem)

Roman to Int (General Array/Vector problem)

Stack

A stack is a LIFO data structure (LIFO - Last In, First Out)

→ Think of a ‘stack’ of pancakes (Typically you eat the pancake on top first, the last pancake you put on)

Real-world example:

- Browser history

Stack Syntax

Stacks can be utilized using lists in Python (Lists in Python are ordered, changeable, and allow duplicate values)

- Ordered: Items have a defined order (when adding an element to a list, it will be added to the end of the list by default)

Creating an empty stack in Python

```
myStack = []
```

Adding an element to a stack

```
myStack.append(ELEMENT)
```

Popping an element in a stack

```
myStack.pop()
```

Stack Example Problem (Balanced Parentheses)

Given an expression string, write a python program to find whether a given string has balanced parentheses or not.

Examples:

```
Input : {[]{()}}
Output : Balanced

Input : [{}{}(]
Output : Unbalanced
```

Balanced Parentheses Solution

```python
# Python3 code to Check for
# balanced parentheses in an expression
open_list = ["[","{","("]
close_list = ["]","}",")"]

# Function to check parentheses
def check(myStr):
    stack = []
    for i in myStr:
        if i in open_list:
            stack.append(i)
        elif i in close_list:
            pos = close_list.index(i)
            if ((len(stack) > 0) and
                (open_list[pos] == stack[len(stack)-1])):
                stack.pop()
            else:
                return "Unbalanced"
    if len(stack) == 0:
        return "Balanced"
    else:
        return "Unbalanced"
```

Runtime?

Practice Problem

Valid Parentheses (https://leetcode.com/problems/valid-parentheses/)

A Common Stack Use Case

Stacks are widely used for Depth First Search (DFS)

DFS is used to search through a tree or graph to find a node far away from the root node that meets a specific criteria (DFS does not always finds the optimal solution, but a trade-off is that it uses less memory, since it does not need to store a lot of nodes)

If you know that the node that you are looking for is near the ‘bottom’ of the graph, DFS is typically the way to go

Queue

A queue is a FIFO data structure (First-in, First-out)

→ Can be seen as the opposite of a stack

Think of a real-life line (People that get in line first, get to go first)

Real-world example:

- Hospital sign up
- Restaurant reservations

Queue Syntax

Queues can also be implemented using a list

Creating an empty queue in Python

myQueue = []

Adding an element to a queue

myStack.append(ELEMENT)

Popping an element in a queue

myStack.pop(0) → The pop function by default pops the last element, but by putting in 0 as an argument, you can pop the first element

A Common Queue Use Case

Queues are widely used for Breadth First Search (BFS)

BFS is used to search through a graph to find a node that meets specific criteria (BFS always finds the optimal solution, but a trade-off is that it uses more memory, since it needs to store more nodes)

- However, when dealing with a weighted graph, you want to use a priority queue instead of a regular queue
- A priority queue is a type of queue that arranges elements in the queue based on some sort of priority value (Like a priority queue based on which value is the lowest/greatest)

When finding the shortest path to a node, you typically want to use BFS to find it

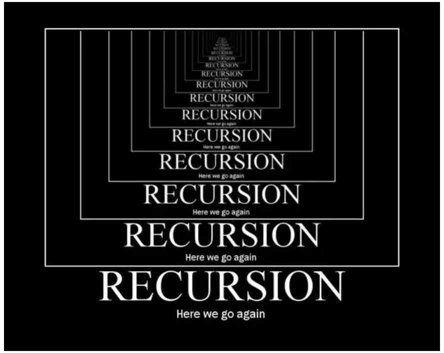
For more information on priority queues:
https://www.geeksforgeeks.org/priority-queue-set-1-introduction/

For more information about Graph Traversal Methods

https://www.youtube.com/watch?v=pcKY4hjDrxk

Goes over DFS and BFS with some useful walkovers and visual aids

Recursion



Recursion is when a function calls itself directly or indirectly

What makes recursion so powerful?

→ It can reduce code length dramatically (Abstracts a lot of the work done)

Recursion is a ‘leap of faith’ (You have to assume that the function already works)

General idea of recursion: Every time you call your recursive function, the input that you put in the function should be getting smaller and smaller (or reach a certain point like a nullptr)

Requirements of a recursive function

1. You MUST have a base case(s)
   a. Without a base case(s), you will have infinite recursion (There will be no point where you stop the recursion)
2. Recursive step
   a. Calling the function again with a smaller and smaller input size

→ **KEYNOTE**: Generally, finding the base case(s) first will help you determine the recursive step much easier

Recursion Example (Factorial)

```
def factorial_recursion (n):
    if n == 1:  ← Base case
        return n
    else:
        return n * factorial_recursion( n - 1) ← Recursive step]
```

To help it all make sense, let’s do an example where we have n = 3:

1. First, we return 3 * factorial_recursion (2)
2. Then, we return 2 * factorial_recursion (1)
3. Then, because the input is 1, we have reached the base case, so we just return 1
4. Because we have the value of factorial_recusion(1), we know that 2 * factorial_recursion(1) = 2
5. Lastly, because we know the value of factorial_recursion(2), we know that 3 * factorial_recursion(2) = 6

Another Example…Fibonacci Sequence

A fibonacci sequence is where each fibonacci number is the sum of the two previous preceding fibonacci numbers (This does not include negative numbers)

→ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, … (Fibonacci Sequence)

Questions:

- What is the base case(s)?
- What is the recursive step?

KEYNOTE: When trying to find the base case, think what numbers/conditions can I no longer recurse back

Building the Fibonacci Function

```
def Fibonacci (n):
        …(Time to solve it)
```

Fibonacci Sequence Solution

```
def Fibonacci (n):
    if n <= 1:
        return n
    else:
        return Fibonacci(n - 1) + Fibonacci (n - 2)
```

Recursion Practice Problem

Power function

→ Implement this function using recursion

→ What this function does is the pow() built into Python

- pow(4,3) = 4 * 4 * 4 = 64
- https://www.w3schools.com/python/ref_func_pow.asp (Reference of the pow() function in python)
-

HINT: Sometimes, when doing recursion, you want to pass two things into your recursive function call

Power Function Solution

```python
# Python3 code to recursively find
# the power of a number

# Recursive function to find N^P.
def power(N, P):

    # If power is 0 then return 1
    # if condition is true
    # only then it will enter it,
    # otherwise not
    if P == 0:
        return 1

    # Recurrence relation
    return (N*power(N, P-1))
```

A Common Recursion Use Case

Recursion is widely used when dealing with tree traversal

- However, unlike with trees, there is no sense of dealing with a smaller and smaller input, so instead, we say if it reaches a nullptr

Example:

If root is None: ← This is the base case commonly used for tree traversal, signifying that we have reached a leaf node (the end of a 'path' for a tree)

return

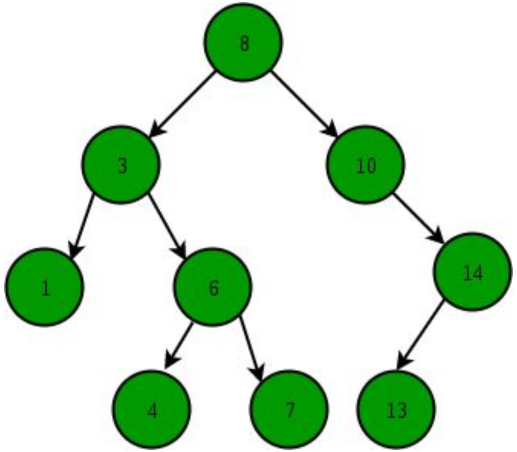


Diagram of a binary search tree (BST)

Additional Readings (Written in C++)

https://ajzhou.gitlab.io/eecs281/notes/

Topics Covered

- Chapter 5: Recursion and Recurrence Relations
- Chapter 9: Stacks and Queues
- Chapter 17: Hash Tables and Collision Resolution

NOTE: This textbook goes incredibly in-depth and provides a lot of context to what we are learning; it also covers a lot of useful examples