Developed by: Linda Perez
rosmery.p.p@gmail.com

Algorithms and Data Structures

Project: Problems vs Algorithms

Decision Made and Complexity

Problem-1

Finding the Square Root of an Integer

 # The expected time complexity is O(log(n))

Decision Made: Create a list from 0 to the middle of the number and use binary search to identify the square root of the number.

***def** sqrt(number: int)-> int:*

***def** binary_search(arr, target) -> int*

"A little modification of the algorithm is that the target is the square of the number"

Complexity Time: Binary Search O(log(n))

Complexity Memory = List(0,n//2) : O(n) *n: is the size of the list n/2*

Problem-2

Search in a Rotated Sorted Array

 # The expected time complexity is O(log(n))

Decision Made: We know we can find an element in a sorted array in an efficient way using binary search, in this case, what we need to do is find a pivot or index where the min element is, and search the target.

***def** rotated_array_search(input_list, number) -> int:*

***def** binary_search(arr, target, left, right) -> int: "is used to find the target inside the array rotated"*

***def** find_pivot(arr, left: int, right: int) "we used binary search to get pivot because the array is Sorted"*

Complexity Time: Binary Search O(log(n))

Complexity Memory = O(1) *"In Place "*

Problem-3

Rearrange Array Elements

 # The expected time complexity is O(nlog(n))

Decision Made: To find two elements where the sum is maximum. What We do is sort the array and the from the end of the array and put each value to their correspondence position.

def rearrange_digits(input_list):

def merge_divide(arr):

def conquer_merge(arr, left, right):

Complexity Time: MergeSort(O(nlog(n)) + O(n) -> O(nlog(n))

Complexity Memory = O(#numbers) - > O(2) -> O(1)

Problem-4

Dutch National Flag Problem

 # The expected time complexity is O(n)

Decision Made: If we have just 3 numbers we can divide the array into three parts, when we traverse the array we can swapping 0's to the first part and 2's to the third part keeping the 1's in the center or second part of the array.

***def** sort_012(input_list)*

Complexity Time: O(n)

Complexity Memory = O(1) *"In Place"*

Problem-5

Building a Trie in Python

Decision Made: As we did with Tries in Basis Algorithms Part of Nanodegree. We can keeping the same structure and calculate suffixes traversing the Trie.

```
class TrieNode:

  def insert(self, char):

## The Trie itself containing the root node and insert/find functions

class Trie:

  def insert(self, word):

  def find(self, prefix):

  def suffixes(self, suffix = ''):

  def suff(self, current_node: TrieNode, output:str, ans):
```

*Complexity Time: find an element inside a dictionary O(1) * traverse the trie structure O(n)*

Where n: is the number of characters of a word → O(n)

*Complexity Memory = O(dictionary*character)*

Problem-6

Max and Min in an Unsorted Array

 # The expected time complexity is O(n)

Decision Made: traverse the array getting the min and max element, comparing with every element.

```
def get_min_max(ints):
```

Complexity Time: O(n)

Complexity Memory = O(1) "In Place"

Problem-7

Router Using a Trie

Decision Made: Applied knowledge from problem-5 just changing the character for parts from a path and adding a handler attribute to identify the handler of any Path.

```
class RouteTrieNode:
  def insert(self, word: str, handler=None):
```

Complexity Time: O(1)

Complexity Memory = O(dictionary)

```
# A RouteTrie will store our routes and their associated handlers
class RouteTrie:
  def insert(self, elements: List[str], handler=None):
  def find(self, elements: List[str], handler=None) -> str:
```

Complexity Time: ***Insert*** *and* ***find*** *O(n)*

Complexity Memory = ***insert and find*** *O(dictionary*word) + O(list of part-path) => O(1)*

```
# The Router class will wrap the Trie and handle
class Router:
  def add_handler(self, path: str, handler=None):
  def lookup(self, path: str):
  def split_path(self, path) -> List[str]:
```

Complexity Time: ***add_handler***, ***lookup*** *O(n)*

*Complexity Memory = O(dictionary*word) + O(list of part-path) => O(1)*