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: We can create a list from 0 to the middle of the number and use binary search to identify the square root of the number. Checking in every iteration of the Binary search if the square of a particular number of a list is the number what we are looking for of their floor.

***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

BinarySearch is O(log(n)) as SquareRoot. We can demonstrate this counting the number of iterations we need to find a number in the array [0,number/2].

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

We just has generated an additional data structure, a list to find the square root.

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 that 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

Rotated Array Search is O(log(n)) as BinarySearch because we have determined Pivot using BinarySearch and we have got the number under the array using Binary Search. Then, We did several Binary Search c(log(n)) where c is the number of times that we traverse the array with Binary Search. Then it is the same as O(c*log(n)) → O (log(n)) because in a worst case we don't need to appreciate the constants.*

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

We have processed the array without any additional Structure.

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 from the beginning to the end of the array and put each value to the correspondence position.

def rearrange_digits(input_list):

def merge_divide(arr):

def conquer_merge(arr, left, right):

Complexity

To rearrange the numbers of an array we can use the MergeSort algorithm where in the worst case their complexity is O(nlog(n)) additionally after have sorted the array we need to distribute the numbers in two lists then it is O(n). Resulting O(nlog(n))+O(n) → O(n+1log(n)) → O(nlog(n))

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

We have need just 2 additial arrays of size 1 to generate the result

Complexity Memory = O(size of the result array) - > O(constant) -> 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

We just have traversed the array one time then we have made n comparisons.

Complexity Time: O(n)

We have processed the array without any additional Structure.

Complexity Memory = O(1)   "In Place"

Problem-5

Building a Trie in Python

Decision Made: As we did with Tries in The Basis Algorithms Part of Nanodegree. We can keep 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

Complexity Time

Insert an element in a Trie in a worst case is O(n) because what we should add every character in the trie structure.

Find a prefix element in a Trie is O(1) per every character of a Word. Then is O(n)

To Generate Suffixes we find the prefix of the word O(n) and then we're getting ('backtracking") the whole word recursively. It is O(n) + O(n^2) -> O(n^2)

Complexity Memory

*To insert an element we need a new dictionary O(size of dictionary*number of elements)*

To Find a prefix element in a Trie we don't need an additional structure O(1)

To Generate Suffixes we just need the result list when we have concatenated the word.

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

What we need is just traverse the array and do n-comparisons. Resulting O(n) Even if we have the maximum or minimum at the end of the unsorted array.

Complexity Time: O(n)

There is not an additional structure.

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

Problem-7

Router Using a Trie

Decision Made: We can apply 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

Complexity Time:

Insert an element in a Trie in a worst case is O(n) because what we should add every word in the Trie structure.

Find a prefix element in a Trie is O(1) per every word of a Path. Then is O(n)

Complexity Memory

*To insert an element we need a new dictionary O(size of dictionary*size of the word)*

To Find a prefix element in a Trie we don't need an additional structure 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

Complexity Time:

Split Path is divide the list in parts O(n) .

Add a Handle in a Trie in a Trie in a worst case is O(n) + O(n) because firstly we split the path then we find per every word from the beginning of the Path to the end. Resulting in O(2n) → O(n)

Lookup a prefix element in a Trie is O(2n) because we initially split the path and then we find every Word. Then is O(n) too.

Complexity Memory

To Add an element we need a new dictionary O(size of dictionary size of the word)*

To Find a prefix element in a Trie we don't need an additional structure O(1)