COMP 6481

Programming and Problem Solving: Assignment 1

Submitted To: Kaustubha Mendhurwar

Submission Date: October 15 2023

Iymen Abdella

Student No. 40218280

PART 1

QUESTION 1A

```
SwapOccurrences( list, A, B):
    For each element in the list:
        If element = A:
            Set element to B
        Else If element = B:
            Set element to A
```

QUESTION 1B

The time complexity of the algorithm for swapping all occurrences of two numbers A and B in a list is O(n), where n is the number of elements in the list. This is because the algorithm iterates through the list once, performing constant-time operations to check and swap for each element.

QUESTION 1C

The space complexity of the algorithm for swapping all occurrences of two numbers A and B in a list is O(1), since it uses constant additional space regardless of the size of the input list. Our algorithm only uses a fixed amount of memory to store temporary variables for index tracking and the values of A and B.

QUESTION 2A

```
RearrangeString(inputString):
    lowercaseLetters = []
    digits = []
    uppercaseLetters = []
    For each character in inputString:
        If character is a lowercase letter:
            Append character to lowercaseLetters
        Else If character is a digit:
            Append character to digits
        Else If character is an uppercase letter:
            Append character to uppercaseLetters
    Sort digits in ascending order
    Sort lowercaseLetters in alphabetical order
    Sort uppercaseLetters in alphabetical order
    resultString = Concatenate(lowercaseLetters, digits, uppercaseLetters)
    Return resultString
```

QUESTION 2B

The algorithm has time complexity of **O(n*log(n)).** List creation can be done in constant time. Iterating through input string of size n can be done in O(n), each append and check can be done in constant time as well. Finally, the sort is done in **O(n*log(n))**, and concatentation is done in linear time complexity.

QUESTION 2C:

The space complexity is **O(1)** since the memory size remains constant regardless of the size of the input string. We need only to account for the list creation which can be done in O(1) space complexity.

QUESTION 3i

```
FindPrimes(arr, target) :
    For i = 0 to length(arr) - 2:
        If IsPrime(arr[i]) and IsPrime(arr[i + 1])
         and arr[i] + arr[i + 1] = target:
            Display "Two consecutive prime numbers with a sum of", target,
            "are", arr[i], "and", arr[i + 1],
                    "found at indices", i, "and", i + 1, "respectively."
            Return  // Exit the loop after the first pair is found

    Display "No two consecutive prime numbers with a sum of", target, "were
found in the array."


IsPrime(n):
    If n <= 1:
        Return False
    ElseIf n <= 3:
        Return True
    ElseIf n is divisible by 2 or 3:
        Return False
    End If
    i = 5
    While i * i <= n:
        If n is divisible by i or n is divisible by i + 2:
            Return False
        i = i + 6


    Return True
```

QUESTION 3ii

The algorithm is designed with the following motives:

Efficiency: It efficiently checks consecutive elements, stopping after finding the first valid pair, thus optimizing time complexity.

Accuracy: The IsPrime function accurately determines prime numbers.

Readability: It uses clear variable names and comments for easy comprehension.

User-Friendly Output: Provides a user-friendly message with prime numbers, indices, and the target sum.

Robustness: Handles cases where no valid pair exists and communicates this clearly in the output.

QUESTION 3iii

The time complexity of the solution is O(n * sqrt(m)), where "n" is the length of the input array, and "m" is the maximum value in the array. Iterating through the Array is O(n) because it goes through each element once. Checking for Primality is done for each element within for loop. Checking if a number "m" is prime involves iterating up to the square root of "m" to check for divisors and has time complexity: O(sqrt(m)).

Combining these, we get O(n * sqrt(m)).

QUESTION 3iv

The maximum size of stack growth in the algorithm is determined by the depth of recursive calls during primality checks in the IsPrime function. The maximum depth occurs when the input "m" is a large prime number, making it proportional to the square root of "m." Therefore, the maximum stack growth is O(sqrt(m)), where "m" is the largest element in the input array.