GIT Department of Computer Engineering
CSE 222/505 - Spring 2022
Homework 4 Time Complexity Report

Emircan Demirel
1901042674

Question 1:

```java
private static int SubStringCount(String mainString, String query, int currentOccur, int targetOccur, int index){
    int mainLength = mainString.length();
    int queryLength = query.length();

    /*Base Case*/
    if (mainLength == 0 || mainLength < queryLength
            || queryLength == 0 || targetOccur <= 0)
        return -1;

    /*
    check given query String
    if string is found increase return value,
    then recursively traverse in mainString
    */
    if (mainString.substring(0, queryLength).equals(query)){
        currentOccur++;
        if(currentOccur == targetOccur)
            return index;
    }
    /*
    continue to search recursively
    */
    return SubStringCount(mainString.substring(1), query, currentOccur, targetOccur, index + 1);
}
```

Recursive calls n times

$T(n) = T(n-1) + \theta(1)$

$T(n-1) = T(n-2) + \theta(1)$

.

.

.

$T(0) = \theta(1)$ ----> sum of all lines $T(n) = \theta(n)$

Time Complexity: $\theta(n)$

Question 2:

```
private static void BinarySearch(int[] arr, int left, int right, int belowLimit, int topLimit){
    if(right < left || belowLimit >= topLimit){
        return;
    } else {
        int mid = left + (right - left) / 2;
        if (arr[mid] <= topLimit && arr[mid] >= belowLimit)
            System.out.printf(arr[mid] + " ");
        if(arr[mid] >= belowLimit)
            BinarySearch(arr, left, mid - 1, belowLimit, topLimit);
        if(arr[mid] <= topLimit)
            BinarySearch(arr, mid + 1, right, belowLimit, topLimit);
    }
}
```

This method has θ(log n) complexity for positive values and another θ(logn) complexity for negative values. It gives us θ(log n) value in general.

BinarySearch works for a range of size zero:

Proof: If the belowLimit is bigger than topLimit or right key is smaller than left key the method return null, which is correct

Assuming BinarySearch works for a range of elements of any size from 0 to n, then binary search works for a range of elements of size n+1.

Proof: If arr[mid] >= belowLimit, this means we need to search the range [left, mid). The range is [left, mid) is necessarily smaller than the range (mid, right] and we assumed that binary search works for all ranges of size smaller than n+1, so it will work for [left, mid) and (mid, right] with similar logic applies.

If belowLimit <= arr[mid] <= topLimit, then the function print the element to the console, which is correct.

Time Complexity: θ(log n)

Question 3:

```
private static void Pair(int[] arr, int sum, int firstIndex, int nextIndex, int currentSum, ArrayList<Integer> subArray){
    if(firstIndex >= arr.length - 1 || arr.length == 0)
        return;
    else if(nextIndex >= arr.length){
        subArray = new ArrayList<>();
        Pair(arr, sum, firstIndex+1, firstIndex+1, 0, subArray);
        return;
    } else {
        currentSum += arr[nextIndex];
        if(sum >= 0){
            if(currentSum <= sum){
                subArray.add(arr[nextIndex]);
                if(currentSum == sum){
                    currentSum = 0;
                    System.out.println(subArray);
                    subArray = new ArrayList<>();
                }
            } else if(currentSum > sum){
                subArray = new ArrayList<>();
                Pair(arr, sum, firstIndex+1, firstIndex+1, 0, subArray);
                return;
            }
        }
    }
    Pair(arr, sum, firstIndex, nextIndex+1, currentSum, subArray);
}
```

FirstIndex increases n times and depended to it, nextIndex increases n times in every increment. This gives us $\theta(n^2)$ as a worst case and $\theta(n)$ as a best case.

Time Complexity: $O(n^2)$

Question 4 is on the next page -------->

Question 4:

```
# foo (integer1, integer2)

    if (integer1 < 10) or (integer2 < 10)        base case: constant time
      return integer1 * integer2

    //number_of_digit returns the number of digits in an integer
    n = max(number_of_digits(integer1), number_of_digits(integer2))
    half = int(n/2)                                          -> constant time x2

    // split_integer splits the integer into returns two integers
    //               from the digit at position half. i.e.,
```

```
    // first integer = integer / 2^half
    // second integer = integer % 2^half
    int1, int2 = split_integer (integer1, half)   constant time
    int3, int4 = split_integer (integer2, half)   constant time

    sub0 = foo (int2, int4) recursive call: θ(log n)
    sub1 = foo ((int2 + int1), (int4 + int3)) recursive call: θ(log n)
    sub2 = foo (int1, int3)  recursive call: θ(log n)

    return (sub2*10^(2*half))+((sub1-sub2-sub0)*10^(half))+(sub0)constant time
```

This function takes two integers as parameters. In base case, if one of these integers is smaller than 10, the function return multiplication of these two numbers. On the other hand, return value of the max method, which compared number of digits for integers, initialized as n variable. Then half of n assigned to half integer variable. split_integer method splits the integer into returns two integers from the digit at position half. At the end, the function makes recursive calls to assign sub values and return an operation with using this sub values.

Time Complexity: O(log n)