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

Emre YILMAZ
1901042606

1. SYSTEM REQUIREMENTS

Task 1.)

- There has to be a recursive function that searches a given string in another given bigger string. The function should return the index of the *ith* occurrence of the query string and return -1 when the query string doesn't occur in the big string or the number of occurences is less than *i*. The *i* value comes from user.

Task 2.)

- It is assumed that we have a sorted integer array. We must implement a recursive algorithm to find the number of items in the array between two given integer values. The searching process must be done as ***binary search***

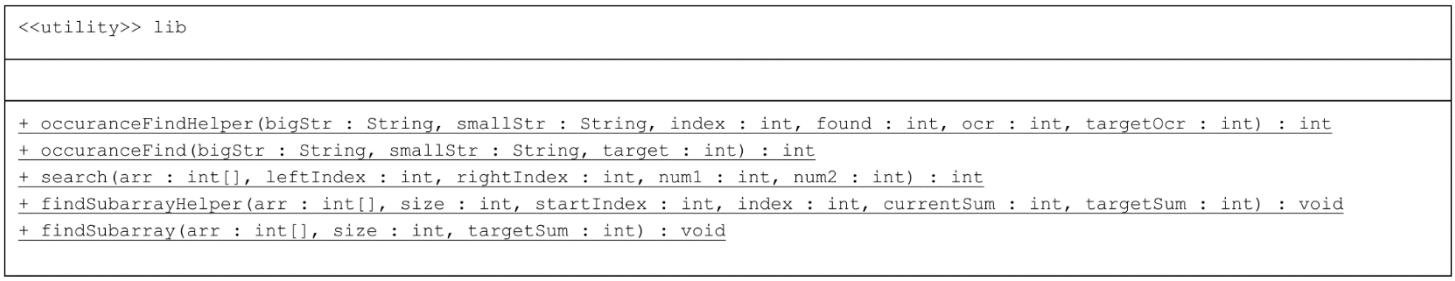
Task 3.)

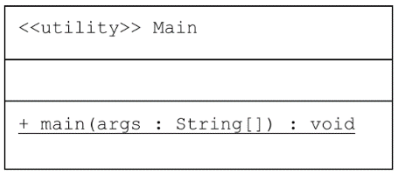
- It is assumed that we have an unsorted integer array. We must implement a recursive algorithm to find contiguous subarray/s that the sum of its/their items is equal to a given integer value.

Task 4.)

- There is a part of code, that is recursive function, we have to explain the algorithm and how it works in detail. We must analyze the time complexity of this function by analyzing the number of multiplication operations (which is the basic operation) at the base case.

2. CLASS DIAGRAMS





3. PROBLEM SOLUTION APPROACH

Task 1.)

- The function should return the index of the *ith* occurrence of the query string. Firstly, we have 2 index parameters *index (index of bigger string)* and *found(index of smaller string).* We also have current occurrence and the target occurrence as parameters.
- Base case is when the index of the large string exceeds the size of the string.
- There is a final condition. If the *currentOccurence* and the target *occurrence* are the same, we have to return the current index.
- There are more 3 recursive conditions.
    - The first of these conditions is when the current indexes of the bigger and smaller strings are equal and the smaller string comes to the end. When this condition is true, it means that we found an occurrence. Now, we will continue. We will have a recursive call. So, we have to change some parameters of new call. We must increase *currentOccurence* variable, then we have to make zero the index of the smaller string. Then we will increase the index of bigger string.
    - The second condition is when current indexes of the bigger and smaller strings are equal and the smaller string does NOT come to the end yet. When this condition is true, it means that we have not found an occurrence yet but there is a chance to find. We have to go on. Now, we will have a recursive call. We have to change some parameters. The current index of the big string must NOT be changed. The current index of the small string must be increased by one. The rest of parameters are the same.

- The third condition is there is no match. When this condition is true, we will have a new recursive call. We have to change some parameters. We must make the index of the small string ZERO since there is no match, yet. We must increase the index of the bigger string by one. The rest of parameters are the same.

Task 2.)

- We will make a binary search.
- Firstly, we will find the middle index.
- The base case occurs when there is no middle index anymore (the search is done). It returns 0.
- There 3 recursive conditions
  - The first condition is if the middle element is bigger than upper bound. If this condition is true, we will have a recursive call and change some parameters. The middle index is going to be *right index,* the left index does not change.
  - The second condition is if the middle element is smaller than lower bound. If this condition is true, we will have a recursive call and change some parameters. The middle index is going to be *left index,* the right index does not change.
  - The third condition is if the middle element is between upper and lower bounds. Then, we must increase return value by one and add it with 2 recursive calls. There may be some elements between upper bound and lower bound both the left of the found middle index element or right of the foud middle index element. So, we have a recursive call *1+recursiveCall(toLeft)+recursiveCall(toRight)* recursive call.

Task 3.)

- We are going to search contiguous subarray. We have 1 *array* and 2 *index*(*startIndex* and *index*) parameters. We have *currentSum* and *targetSum* parameters
- StartIndex => The starting index of the found subarray.
- Index => The current index that we are checking.
- The base case is occurs when we exceeds the size of the array.
- If the element of the array at [*index]* is smaller than *targetSum*, it means that we have a chance to find a subarray. We will add the value of the element at [index] position to *currentSum*.
  - If it is equal to *targetSum* it means that we have found a subarray. Then, we will print the information of this subarray and continue to search with a new recursive call. In this new recursion call, we start to search a new subarray. So, we will increase the *startIndex* by one and make the index=startIndex+1

since we are starting to search a new subarray. We also will make *currentSum*=0 since we start to a new subarray searching

- If it is smaller than *targetSum*, it means that we have a change to find a subarray. We will make a recursive call. In this recursive call we will NOT change the *startIndex* since we are continue searching that already exists. We will increase the *index* by one. We will increase the *currentSum* by the element [index] of the array.

- If the element of the array at [*index]* is greater than *targetSum*, it means that there are no match. We will start a new position to search. We will make a recursive call and increase the startIndex by one. We will make the index=*startIndex* since we are starting to search a new subarray.
- ***NOTE FOR T.A***. -> In this function, the subarrays are printed as "from [3] to [5]" that means "3. index of the array to 5. index of the array"

4. TEST CASES

TASK 1.)

- 4 different case is run:
  - Looking for a word in the Big String's middle of somewhere and the occurrence number is smaller than target occurrence
  - Looking for a word in the Big String's end and the occurrence number is equals to target occurrence.
  - Looking for a target i value that greater than the occurance exists in the big string.
  - Looking for a word in the Big String's beginning and smaller than target occurrence.

TASK 2.)

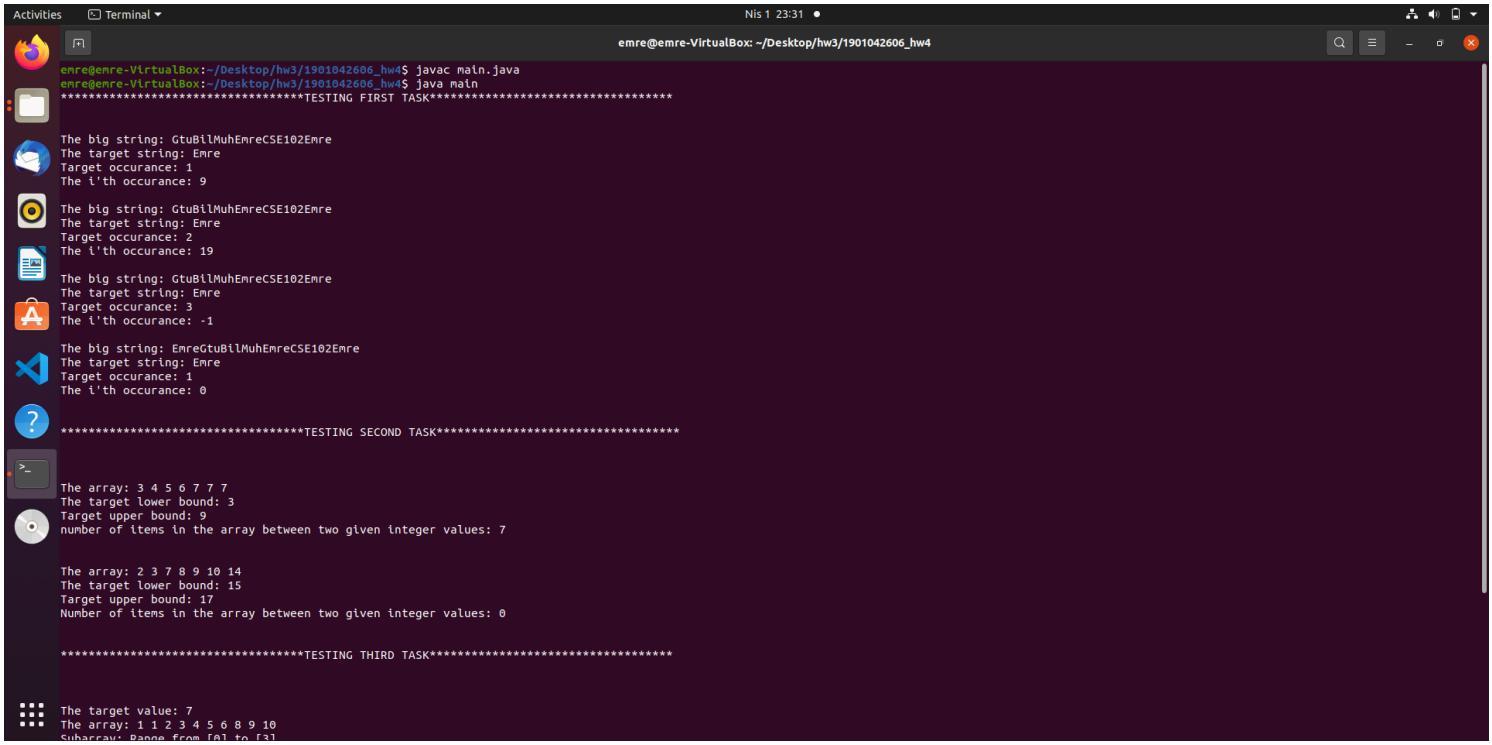
- The target numbers (between upper and lower bound) is put to head, tail and middle of the array. These conditions is run successfully.
- Then, a test case that there is no number between upper and lower bound is set. This condition is run successfully.

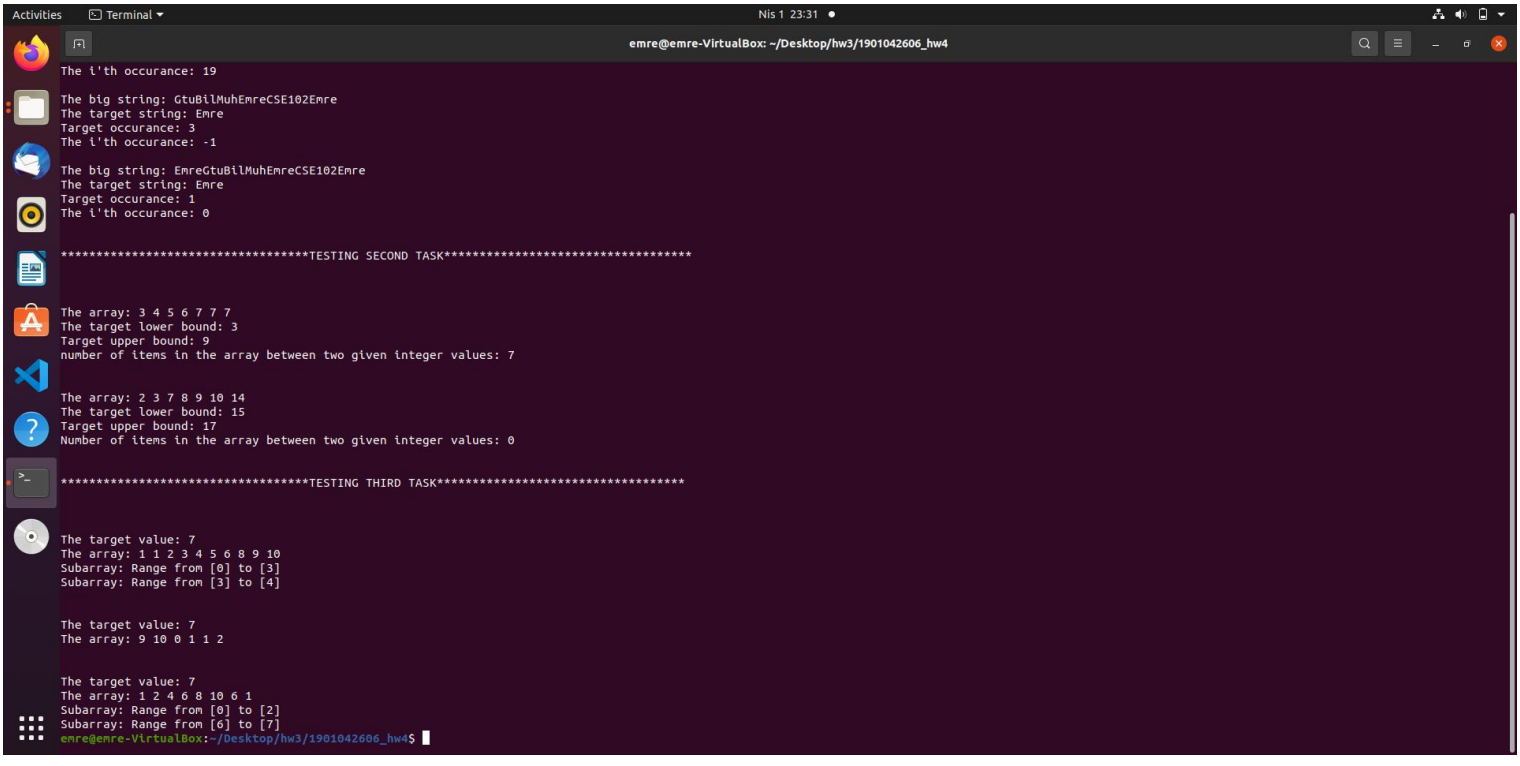
TASK 3.)

- Subarrays are set to be contiguous. It runs successfuly

- There are subarrays at the beginning and end of the array. This condition runs successfully.
- The condition that there is no subarray is run successfully

5. RUNNING AND RESULTS





QUESTION 1 TIME COMPLEXITY

```
public static int occuranceString(String bigStr, String smallStr,int index, int found, int ocr, int targetOcr)
{
    if(ocr==targetOcr) Θ(1)
        return index-smallStr.length(); Θ(1)

    else if(index==bigStr.length()) Θ(1)
        return -1; Θ(1)

    if(bigStr.charAt(index)==smallStr.charAt(found) && found==smallStr.length()-1) Θ(1)
        return occuranceString(bigStr, smallStr, index: index+1, found: 0, ocr: ocr+1, targetOcr); T(n) = T(n-1) + Θ(1) = Θ(n)

    else if(bigStr.charAt(index)==smallStr.charAt(found)) Θ(1)
        return occuranceString(bigStr, smallStr, index: index+1, found: found+1, ocr, targetOcr); T(n) = T(n-1) + Θ(1) = Θ(n)

    else
        return occuranceString(bigStr, smallStr, index: index+1, found: 0, ocr, targetOcr); T(n) = T(n-1) + Θ(1) = Θ(n)
}
```

Best Case = Θ(1)
Worst Case = Θ(n)
Time complexity = O(n)

QUESTION 2 TIME COMPLEXITY

```
public static int search(int [] arr, int leftIndex, int rightIndex, int num1, int num2)
{
    int midElement =( rightIndex + leftIndex) / 2; Θ(1)

    if(rightIndex-leftIndex==1) Θ(1)
        return 0;

    if(arr[midElement] > num2) Θ(1)
        return search(arr, leftIndex, midElement,num1,num2); T(n) = T(n/2) + Θ(1) =Θ(logn)

    else if(arr[midElement] < num1) Θ(1)
        return search(arr,midElement, rightIndex, num1, num2); T(n) = T(n/2) + Θ(1) = Θ(logn)

    else
    {
        return 1 + search(arr,midElement, rightIndex, num1, num2)
        + search(arr, leftIndex, midElement,num1,num2);
    }
}
```

T(n) = T(n/2) + T(n/2) + Θ(1) = 2T(n/2) + Θ(1) = Θ(logn)

The best case = Θ(1)
The worst case = Θ(logn)

The time complexity of the function is: O(logn)

QUESTION 3 TIME COMPLEXITY

```
public static void findSubarrayHelper(int [] arr, int size, int startIndex, int index, int currentSum, int targetSum)
{
    if(index>=size || startIndex>=size) Θ(1)
        return; Θ(1)

    if(arr[index]<=targetSum) Θ(1)
    {
        if(currentSum+arr[index]==targetSum) Θ(1)
            System.out.printf("Subarray: Range from [%d] to [%d]\n",startIndex, index); Θ(1)

        else if(currentSum+arr[index] <= targetSum)Θ(1)
        {
            findSubarrayHelper(arr, size, startIndex, index: index+1, currentSum: currentSum+arr[index], targetSum); T(n) = T(n-1) + Θ(1) = Θ(n)
            return;
        }
    }

    findSubarrayHelper(arr, size, startIndex: startIndex+1, index: startIndex+1, currentSum: 0, targetSum); T(n) = T(n-1) + O(n) = O(n²)
}
```

Best case Θ(1)
Worst Case Θ(n)
T(n) = O(n)

Best Case = Θ(1)
Worst Case = O(n²)
Time complexity = O(n²)

QUESTION 4 EXPLANATION and TIME COMPLEXITY

The purpose is to divide the digits of at least one of the numbers with more than one digit so that one digit remains, while multiplying 2 numbers with each other, and do the operation in this way.

The base case checks whether at least one number have less than 2 digits. If it is, return the multiplication

The maximum digit number of numbers is found, then it is divided by 2 to get ***half*** variable that splits the integer teturns two integers.

After that we pass the new *splitted smaller numbers* to function recursively. So, we are getting close to base case.

When we reach the base case finally, we do the last operation to get multiplication result of two numbers.

```
# foo (integer1, integer2)

    if (integer1 < 10) or (integer2 < 10)  Θ(1)
      return integer1 * integer2

    //number_of_digit returns the number of digits in an integer
    n = max(number_of_digits(integer1), number_of_digits(integer2))  Θ(1)
    half = int(n/2)  Θ(1)

    // 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)  Θ(1)
    int3, int4 = split_integer (integer2, half)  Θ(1)

    sub0 = foo (int2, int4)  T(n) = T(n/ ) + O(1) = O(logn)
    sub1 = foo ((int2 + int1), (int4 + int3))  T(n) = T(n/ ) + O(logn) = O(log²n)
    sub2 = foo (int1, int3)  T(n) = T(n/ ) + O(log²n) = O(log³n)

    return (sub2*10^(2*half))+((sub1-sub2-sub0)*10^(half))+(sub0)  Θ(1)
```

Best Case = Θ(1)
Worst Case = Θ(Log³n)
T(n) = O(Log³n)

INDUCTION PROOFS

Proof Task-1

STEP 1 Checking base case:

if index is equal to length of the string, it means that we exceed the string. We cannot go anymore. Moreover, if we reach the end it means that we could NOT find an occurance. So, the base case is true.

STEP 2

The all recursive calls in the conditions get close to base case. All of them increases , which index variable that can be able to reach the base case, by one

STEP 3

There are 3 recursive conditions that checks whether there is an occurance. We proved that they are getting closer to base case. If the recursive calls find an occurance they return it. So, if a recursive call finds an occurance, the base case does NOT work and the result is returned. We can say that original problem is solved.

Proof Task-2

STEP 1 Checking base case:

If the difference between the left index and the right index is 1, it means that there is no middle index anymore. So, we have done with binary search and we must stop.

STEP 2

There are 3 recursive conditions and all of them and each reduces the difference between the right and left index. It adds the right and left index and divides it into two and passes it to the function as a new limit. When this recursive process continues, eventually the difference between the right and left index will be 1, that is, there will be no elements between the right and left indexes. So we will arrive at the base case at the end of the program.

STEP 3

We have proved that we reach the base case eventually. If the third condition, which the condition where we find a number between the upper and lower limits, occurs in any of the recursive calls, we will add 1 to the result. If this smaller problem occurs, we will add 1 to the return value. This way, every correct target we find will add 1 to our result. We will always be able to return the correct result since we can always reach the base case including the condition where we can't find a number.