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

Hasan Mutlu
1801042673

1. SYSTEM REQUIREMENTS

Functional Requirements:

- ➔ User should be able to see the result of hard-coded test cases.

Non-Functional Requirements:

- ➔ Implementation: The programs shall be implemented using VSCode, Ubuntu 18.04 WSL and Java 11.
- ➔ Compiling an Running: The programs should be compiled and run with following commands:
- ➔ -$ javac *.java PartX.java
- ➔ -$ java PartX
- ➔ Efficiency: The algorithms must run as efficiently as possible.
- ➔ Relaibility: The algorithms and the programs must run reliably, should handle every possible valid usage.

2. CLASS DIAGRAMS

There isn't any object oriented design. Here are the functions and their parameters in every part.

3. PROBLEM SOLUTION APPROACH

I started with part1. It was easiest one. I covered whole string by getting next substring of characters in required length starting from every index and comparing it with the searched string.

In part2, I first implemented it in linear time, like how I did in part1. But it wasn't the most efficient way, obviously. So I converted it to a binary search. At first, I was trying to find the solution by one recursive call but things were getting too complicated. Later, I decided to find appropriate positions of the given two numbers in the array one by one and count number of elements between them by simply subtracting found indexes. But one being

upper bound and one being lower bound was causing contradictions in output. So I decided to introduce one more parameter to indicate their bounds and solved this problem.

In part3, the return type wasn't specified. So I decided to print the index interval of the found substring. This problem requires nested recursive calls, obviously. It would be very complicated for me if I tried to implement it directly recursively. But instead, I decide to first implement it with regular loops. There was two nested loops and I carefully converted them into two recursive calls. One problem was, when no substring was found, it was printing nothing. So I decided to return number of substrings found to the wrapper function and print it there as well.

In part4, I didn't understand anything from the code by looking at it. So I write it and run. When I run, I realized that it multiplies two numbers. I didn't understand how and decided to search for recursive multiplication algorithms in Google and found out that it is Karatsuba algorithm.

In part5, I firstly analyzed the problem. I decided that I need several recursive functions. Counting from start, one for generating blocks with different length, one for generating different number of blocks, one for controlling current block, one for inserting a blocks at all possible positions, and one for inserting the block at a position is needed, with additional helper functions, of course. But as I developed the program, I needed much more functions than these. At the end, the program is finished with a chain of 6 recursive functions.

4. TEST CASES

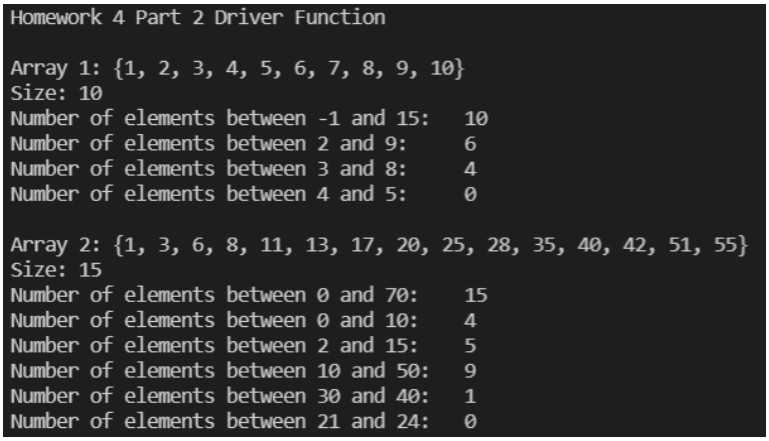
Test Case #	Test Case Description	Test Data	Expected Result	Actual Result	Pass/Fail
1	Part1 Test1	String: "hereheretemptemptempheretemphere" Searched String: "here" Searched Occurances: 0,1,2,3,4,5	"-1, 0, 4, 20, 28, -1"	As Expected	Pass
2	Part1 Test2	String: "123456789123456789123456789" Searched String: "2345" Searched Occurances: 0,1,2,3,4	"1, 10, 19, -1"	As Expected	Pass
3	Part1 Test3	String: "aaaaaa" Searched String: "aa" Searched Occurances: 1,2,3,4,5,6	"0, 1, 2, 3, 4, -1"	As Expected	Pass
4	Part2 Test1	Array: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} Number of elements between: -1 and 15 2 and 9 3 and 8 4 and 5	"10, 6, 4, 0"	As Expected	Pass

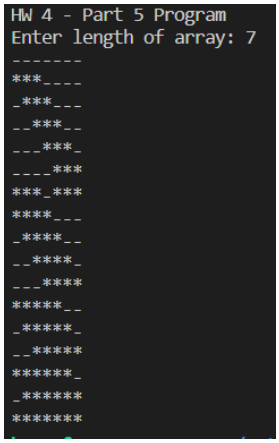
5	Part2 Test2	Array: {1, 3, 6, 8, 11, 13, 17, 20, 25, 28, 35, 40, 42, 51, 55} Number of elements between: 0 and 70 0 and 10 2 and 15 10 and 50 30 and 40 21 and 24	"15, 4, 5, 9, 1, 0"	As Expected	Pass
6	Part3 Test1	Array: {1, 9, 2, 8, 3, 7, 6, 4, 5, 5} Searched sum: 10	0 and 1 2 and 3 4 and 5 6 and 7 8 and 9 5 subarray found.	As Expected	Pass
7	Part3 Test2	Array: {1, 2, 3, 4, 5, 4, 3, 2, 1, 7, 3, 5} Searched sum: 15	0 and 4 4 and 8 9 and 11 3 subarray found.	As Expected	Pass
8	Part5 Test1	Array Length: 7	Same as PDF	As Expected	Pass
9	Part5 Test2	Array Length: 13	Correct Output	As Expected	Pass

5. RUNNING AND RESULTS

```
Homework 4 Part 1 Driver Function

String: herehereteemptemptempheretemphere
Searched String: here
Index of 0th occurance: -1
Index of 1st occurance: 0
Index of 2nd occurance: 4
Index of 3th occurance: 20
Index of 4th occurance: 28
Index of 5th occurance: -1

String: 123456789123456789123456789
Searched String: 2345
Index of 1st occurance: 1
Index of 2nd occurance: 10
Index of 3th occurance: 19
Index of 4th occurance: -1

String: aaaaaa
Searched String: aa
Index of 1st occurance: 0
Index of 2nd occurance: 1
Index of 3th occurance: 2
Index of 4th occurance: 3
Index of 5th occurance: 4
Index of 6th occurance: -1
```

```
Homework 4 Part 3 Driver Function

Array: {1, 9, 2, 8, 3, 7, 6, 4, 5, 5}
Searched sum: 10
Sum found in indexes 0 and 1: {1, 9}
Sum found in indexes 2 and 3: {2, 8}
Sum found in indexes 4 and 5: {3, 7}
Sum found in indexes 6 and 7: {6, 4}
Sum found in indexes 8 and 9: {5, 5}
5 subarray found.

Array: {1, 2, 3, 4, 5, 4, 3, 2, 1, 7, 3, 5}
Searched sum: 15
Sum found in indexes 0 and 4: {1, 2, 3, 4, 5}
Sum found in indexes 4 and 8: {5, 4, 3, 2, 1}
Sum found in indexes 9 and 11: {7, 3, 5}
3 subarray found.

Array: {11,-1,12,-1,-1}
Searched sum: 10
Sum found in indexes 0 and 1: {11, -1}
Sum found in indexes 1 and 3: {-1, 12, -1}
Sum found in indexes 2 and 4: {12, -1, -1}
3 subarray found.
```





6. COMPLEXITY ANALYSIS

Q1)

```
public static int part1Recursive(String biggerString, String givenString, int ith, int nth,
int index){
        if(index + givenString.length() -1 >= biggerString.length())
            return -1;
        String buf = biggerString.subSequence(index, index + givenString.length()).toString();
        if(givenString.equals(buf)){
            nth++;
            if(ith == nth)
                return index;
            else
                return part1Recursive(biggerString, givenString, ith, nth, index + 1);
        }
        return part1Recursive(biggerString, givenString, ith, nth, index + 1);
   }
```

Best case: When givenString's first occurance is searched and it is in the beginning of the biggerString. In this scenario, (`givenString.equals(buf)`) and (`ith == nth`) conditions are satisfied and time complexity is:

$T(n)_{best} = Teta(1)$

Worst case: When givenString doesn't exist as much as searched occurance amount in the biggerString. In this scenario, the biggerString is covered as much as its length until base case is satisfied and the time complexity is:

$T(n)_{worst} = Teta(n)$

so, $T(n)_{overall} = O(n)$

Note: I assume `subSequence(int, int)` function works in Teta(1) time. Because I think it accesses biggerString's characters in constant time and the substring is created by copying constant amount of characters. If it works in linear or some other time, all results above shall be multiplied by that time complexity.

Q2)

```
public static int part2(int[] arr, int size, int num, int upper, int lower,
int boundType){
        int index = (upper + lower)/2;
        if(index == 0)
            if(num >= arr[index])
                return 0;
            return -1;
        if(index == size - 1)
            if(num > arr[index])
                return size;
            return index;
        if(num == arr[index])
            return index;
        if(num < arr[index] && arr[index - 1] < num)
            if(boundType == 1)
                return index;
            return index - 1;
        if(num > arr[index] && arr[index + 1] > num)
            if(boundType == 1)
                return index + 1;
            return index;
        if(num < arr[index])
            upper = index;
        else if(num > arr[index])
            lower = index;
        return part2(arr, size, num, upper, lower, boundType);
    }
```

Best case: Both two given integer values' appropriate positions are in the middle of the list and that index value is returned immiadetaly. In this case, either (`num < arr[index] && arr[index - 1] < num`) or (`num > arr[index] && arr[index + 1] > num`) conditions are satisfied in the first calls. So time complexity is:

$T(n)_{best}$ = Teta(1)

Worst case: At least one given integer value's appropriate position in the array is at a 'leaf like' position in the array. In this scenario, time complexity is:

$T(n)_{worst}$ = Teta(logn)

so, $T(n)_{overall}$ = O(logn)

Q3)

```
public static int part3_1(int[] arr, int size, int sum, int index){
        int found = 0;
        if(index == size)
            return found;
        int currentSum = arr[index];
        found += part3_2(arr, size, sum, index, index + 1, currentSum, 0);
        return found + part3_1(arr, size, sum, index + 1);
    }

public static int part3_2(int[] arr, int size, int sum, int startIndex, int index,
int currentSum, int found){
        if(index > size)
            return found;
        if (currentSum == sum) {
            int prevIndex = index - 1;
            System.out.print("Sum found in indexes " + startIndex+ " and " + prevIndex +
": {");
            printSubArray(arr, startIndex, prevIndex);
            System.out.println("}");
            found++;
        }
        if (index == size)
            return found;
        currentSum += arr[index];
        return part3_2(arr, size, sum, startIndex, index+1, currentSum, found);
    }

    public static void printSubArray(int[] arr, int index, int endIndex){
        if(index == endIndex){
            System.out.print(arr[index]);
            return;
        }
        System.out.print(arr[index] + ", ");
        printSubArray(arr, index+1, endIndex);
    }
```

Best and worst cases are only related to `printSubArray` function in this program. In p3_1 and p3_2 functions, only base cases are indexes reaching at the end of the array. Entire array is covered by starting from every index and checking the remaining of the array. So it is executed n + n(-1) + (n-2) + …. + 2 + 1 times which is:

n*(n-1)/2 = n^2/2 – 1/2

so, T(n) = Teta(n^2) in first two functions without taking `printSubArray` into account

Best case: When condition that calls `printSubArray` is never satisfied. In this scenario, time complexity is:

$T(n)_{best} = Teta(n^2)$

Worst case: When entire array is the subarray with the given sum, in this case, `printSubArray` function covers entire array, so its time complexity of that function is Teta(n) as itself. In this scenario, total time complexity is:

$T(n)_{worst} = Teta(n^2)*Teta(n) = Teta(n^3)$

$T(n)_{overall} = O(n^3)$

Q4)

This function multiplies two numbers. It is a divide and conquer algorithm. It works faster than regular multiplication approach we normally do in our daily life. Normally, we multiply two numbers by multiplying every digit with each other and summing the results. For example, if we calculate 1234*5678, we make 16 multiplications.

In this foo() function in the PDF, the numbers are divided as '12*10^2 + 34' and '56*10^2+78'. If you multiply these two, you get:

➔ '12*56*10^4 + (12*78 + 34*56)*10^2 + 34*78.'

(12*78 + 34*56) part in the middle is actually equal to (12+34)*(56+78) – (12*56) – (34*78). When put in place:

➔ 12*56*10^4 + [(12+34)*(56+78) – (12*56) – (34*78)]*10^2 + 34*78.

Now this looks familliar. To relate this example with the function in PDF:

➔ int1 = 12, int2 = 34, int3 = 56, int4 = 78,
➔ sub0 = (34*78), sub1 = (12+34)*(56+78), sub2 = (56*78)
➔ n = 4, max number of digits.

I have shown that this function multiplies two numbers so far. Let's continue with performance.

➔ 12*56*10^4 + [(12+34)*(56+78) – (12*56) – (34*78)]*10^2 + 34*78

In here, there are 3 distinct multiplications of input numbers' elements. (12*56) and (34*78) repeat themselves. So we make 1 less multiplication, instead of multiplying every half of these numbers with each other, which would make 4 multiplications. (Multiplications with powers of 10 aren't multiplications, really. They are only shift operations for addition.)

If we continued with foo(12,34), it would call the foo() function 3 times more and all of them would hit the base case, doing one multiplication each. So, for n=2 digits, we would make 3 multiplications instead of 2^n = 4 multiplications. In other words, we would multiply two numbers faster than Teta(n^2) time. This is why it is a faster multiplication algorithm.

➔ It's recurrence relation is T(n) = 3T(n/2) + cn + d, according to the Wikipedia page.
➔ It's time complexity is Teta(n^log2(3)) = Teta(n^1,58), according to the Wikipedia page.