Gebze Technical University
Computer Engineering

CSE 222 - 2019 Spring

HOMEWORK 4 REPORT

GÖKHAN HAS
161044067

Course Assistant: Ayşe Şerbetçi TURAN

1 INTRODUCTION

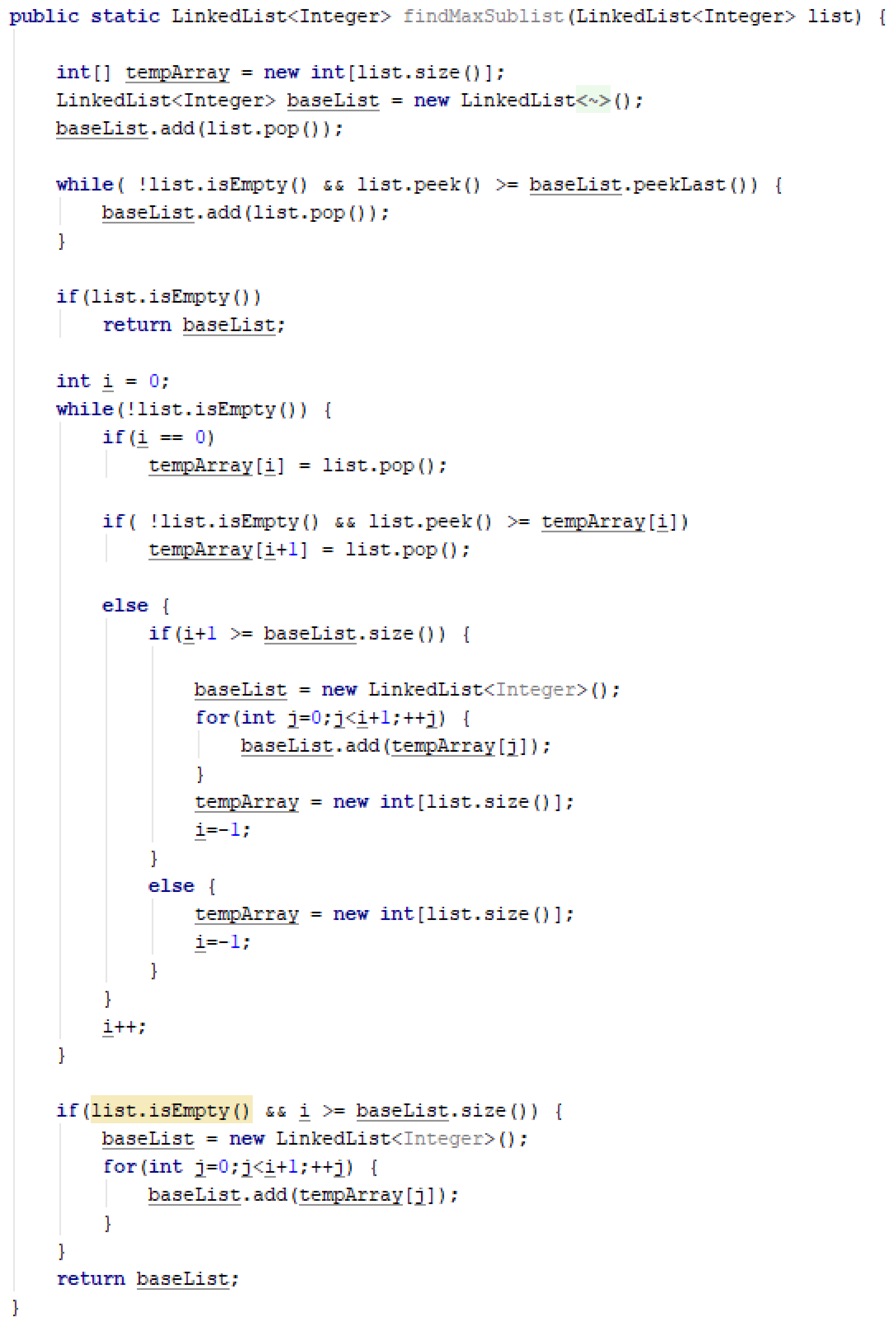
1.1 Problem Definition

In this assignment, 5 different parties are given and the results of these parties are requested. In some questions time complextiy calculation is asked and in some questions, code is requested according to a certain time complexity. Our work is relatively easy when we don't have time complexities. We can behave generously, but in other questions we have to plan and do everything. Those questions take a little more time. I had different solutions but I had to delete a lot of code since I had to write time according to complextiy.

2 QUESTIONS

2.1 Question 1

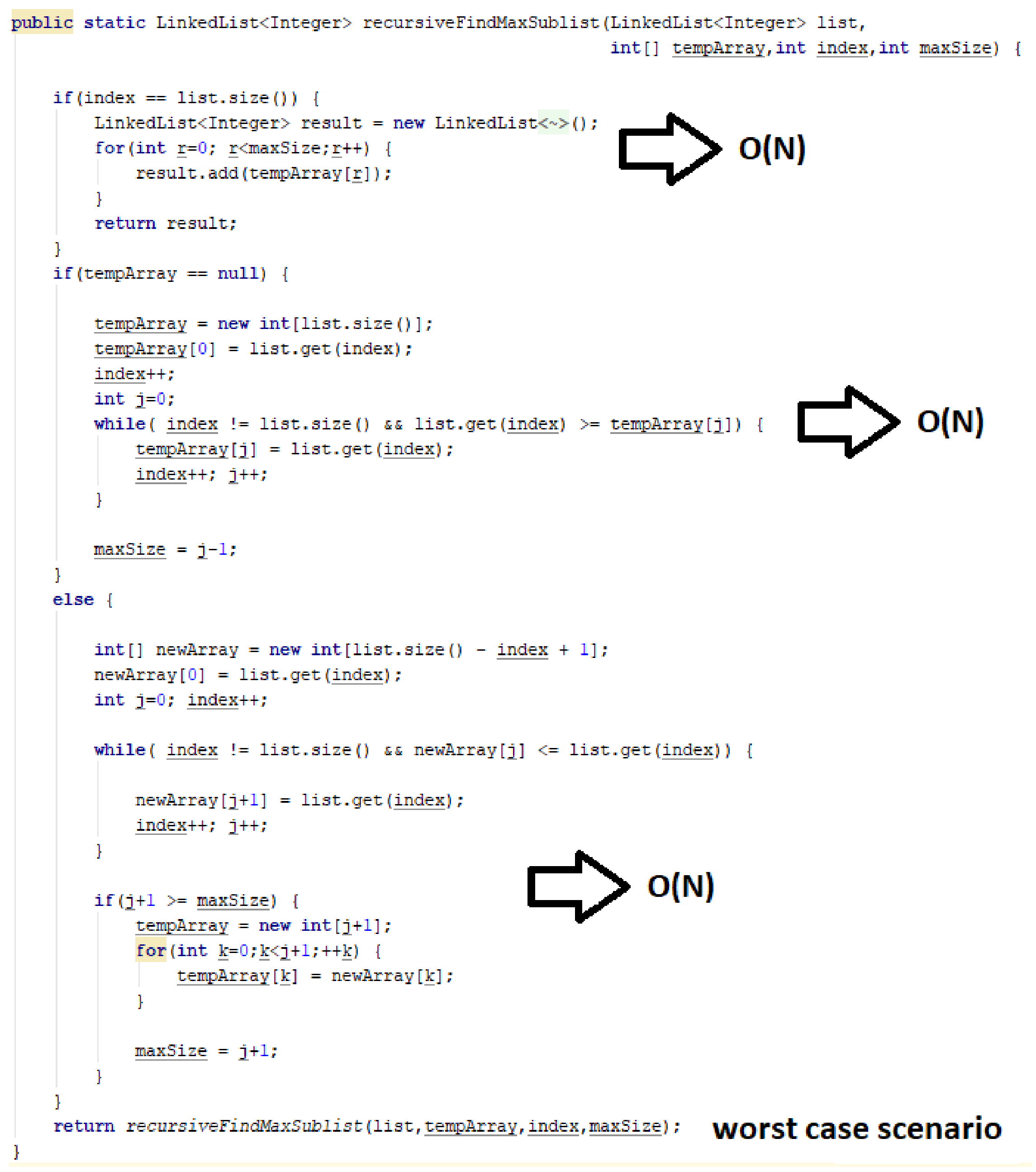
a-) In this case, we are asked to return the maximum sublist in a linkedlist in the form of a linkedlist. I start by creating a one-dimensional array. The size of the Array will be up to the size of the linkedlist. Then I create a linkedList. This linkedliste sent the linkedlist to pop, I'm throwing that element. These operations are O (1). Then I get the first sublist I found in the linkedlist. Then I checked if the linkedlist sent to me is empty. This is O (1). Then comes a while loop rotating up to the size of the list. In the loop there is another loop in the else, but this loop will never return more than or equal to the size of the list. So time complexity will be O (n). The time complexity is O (n) because there is no more time in the function. The last time, it takes as much as O (n).



```java
public static LinkedList<Integer> findMaxSublist(LinkedList<Integer> list) {

    int[] tempArray = new int[list.size()];
    LinkedList<Integer> baseList = new LinkedList<~>();
    baseList.add(list.pop());

    while( !list.isEmpty() && list.peek() >= baseList.peekLast()) {
        baseList.add(list.pop());
    }

    if(list.isEmpty())
        return baseList;

    int i = 0;
    while(!list.isEmpty()) {
        if(i == 0)
            tempArray[i] = list.pop();

        if( !list.isEmpty() && list.peek() >= tempArray[i])
            tempArray[i+1] = list.pop();

        else {
            if(i+1 >= baseList.size()) {

                baseList = new LinkedList<Integer>();
                for(int j=0;j<i+1;++j) {
                    baseList.add(tempArray[j]);
                }
                tempArray = new int[list.size()];
                i=-1;
            }
            else {
                tempArray = new int[list.size()];
                i=-1;
            }
        }
        i++;
    }

    if(list.isEmpty() && i >= baseList.size()) {
        baseList = new LinkedList<Integer>();
        for(int j=0;j<i+1;++j) {
            baseList.add(tempArray[j]);
        }
    }
    return baseList;
}
```

b-) The master theorem cannot be used for this function. This is due to the fact that this function cannot be shown with the master theorem. Master Theorem can be applied to the problems, divide and conquere approach can be solved and the problem that works on each of the area of influence (and of course, later combine) are the types of algorithms. This function I wrote, unfortunately, does not have such a recursion as to divide the problem into pieces.

For this function T (n) = c + T (n-c) will be correct. Recursive call is made 1 time. And each time the index variable is increasing. Base case is given the status of the index variable equal to the size of the list. (c is a first sublist size in the list.)



```java
public static LinkedList<Integer> recursiveFindMaxSublist(LinkedList<Integer> list,
                                                           int[] tempArray,int index,int maxSize) {

    if(index == list.size()) {
        LinkedList<Integer> result = new LinkedList<~>();
        for(int r=0; r<maxSize;r++) {
            result.add(tempArray[r]);
        }
        return result;
    }
    if(tempArray == null) {

        tempArray = new int[list.size()];
        tempArray[0] = list.get(index);
        index++;
        int j=0;
        while( index != list.size() && list.get(index) >= tempArray[j]) {
            tempArray[j] = list.get(index);
            index++; j++;
        }

        maxSize = j-1;
    }
    else {

        int[] newArray = new int[list.size() - index + 1];
        newArray[0] = list.get(index);
        int j=0; index++;

        while( index != list.size() && newArray[j] <= list.get(index)) {

            newArray[j+1] = list.get(index);
            index++; j++;
        }

        if(j+1 >= maxSize) {
            tempArray = new int[j+1];
            for(int k=0;k<j+1;++k) {
                tempArray[k] = newArray[k];
            }

            maxSize = j+1;
        }
    }
    return recursiveFindMaxSublist(list,tempArray,index,maxSize);
}
```

T(n) = c + T(n-c) → c is constant

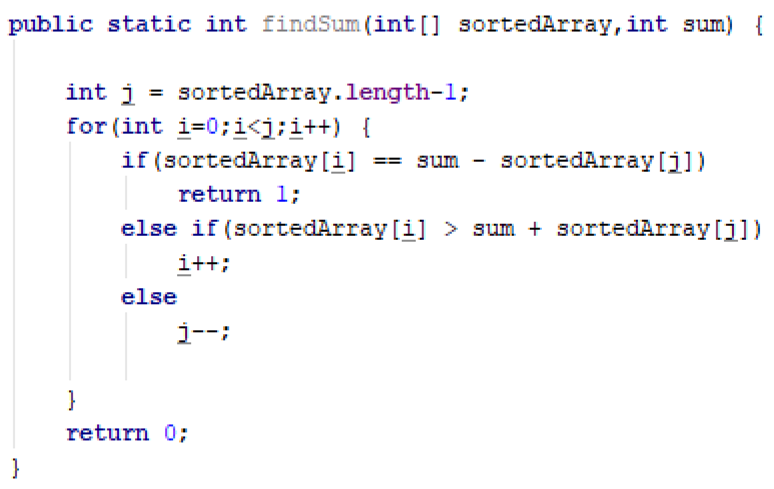
n = 1 → T(n) = n. The statement is true.

Assume that n=k → T(k) = c + T(k-c) is true. That result is O(k) so, O(n)

For n = k+1 → T(k+1) = c + T(k+1-c). That result is O(k+1)

Finally the result is O(n).

2.2 Question 2



```java
public static int findSum(int[] sortedArray,int sum) {

    int j = sortedArray.length-1;
    for(int i=0;i<j;i++) {
        if(sortedArray[i] == sum - sortedArray[j])
            return 1;
        else if(sortedArray[i] > sum + sortedArray[j])
            i++;
        else
            j--;

    }
    return 0;
}
```

Take the worst case scenario when analyzing this code. In the case of the Worst case, the sum of any two numbers cannot be written in any way. Thus the for loop will have to return to the size of the array. Because i was started from 0. The variable j starts from the size of array 1. And the looping condition is specified as i <j. If the result is found 1 will be return, the other kind will be return 0. So the loop will go up to the size of the array. If we say the size of the array, time complextiy, θ(n) will be found as.

2.3 Question 3

```
for (i=2*n; i>=1; i=i-1)                      → This loops times 2n
        for (j=1; j<=i; j=j+1)                → This loops depend on i. So its 2n.
              for (k=1; k<=j; k=k*3)          → This loops depend on log3(i+1).So,
                    print("hello")              log3(2n+1)
```

In the nested loops, the results are multiplied. So 2n x 2n x log3(2n+1) = **O(n^2logn)**

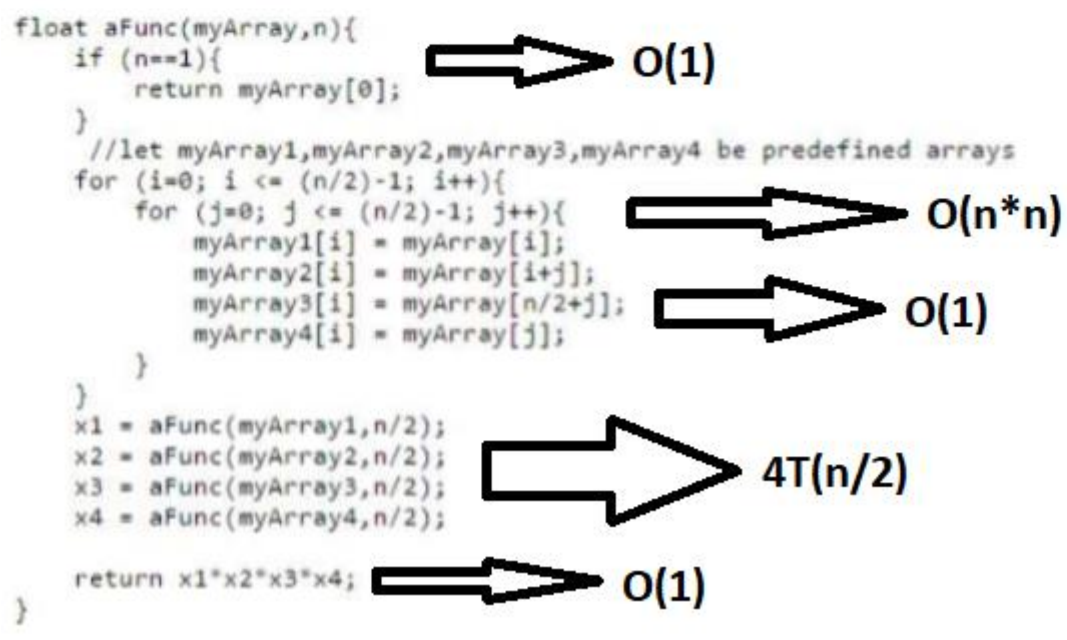
2.4 Question 4

When we examine this function, we first come across a base case. When you enter into the if you are doing a constant time process. Just made the return. So that expression is O (1). Then comes the nested for loops. The first cycle goes up to (n / 2) -1, the second going up to (n / 2) - 1. Since only assignments are made within the loops, the inside expressions can be accepted as O (1). ((n / 2) -1) * ((n / 2) -1) = n^2 / 4 ... as it comes. The result is O (n^2). Then the function is called 4 times and n / 2 elements are sent. So we can express it in 4T (n / 2).

As a result, our main equation is T (n) = n^2 + 4T (n / 2). We can apply master theorem to this equation.

According to master theorem, g(n) = n^2 and c = 2, a = 4 and b = 2

$\log_b{}^a = \log_2{}^4 = 2 = c = 2$. So its result is O($n^c$ logn). Since **O(n^2 logn),** c is equals 2.



```
float aFunc(myArray,n){
    if (n==1){
        return myArray[0];
    }
     //let myArray1,myArray2,myArray3,myArray4 be predefined arrays
    for (i=0; i <= (n/2)-1; i++){
        for (j=0; j <= (n/2)-1; j++){
            myArray1[i] = myArray[i];
            myArray2[i] = myArray[i+j];
            myArray3[i] = myArray[n/2+j];
            myArray4[i] = myArray[j];
        }
    }
    x1 = aFunc(myArray1,n/2);
    x2 = aFunc(myArray2,n/2);
    x3 = aFunc(myArray3,n/2);
    x4 = aFunc(myArray4,n/2);

    return x1*x2*x3*x4;
}
```