Gebze Technical University
Computer Engineering

CSE 222 - 2019 Spring

HOMEWORK 04
1-4 ANSWER OF QUESTIONS

Muhammed ÖZKAN
151044084

Course Assistant: Ayşe ŞERBETÇİ TURAN

1.

A)

```
public LinkedList subListIterative(LinkedList list) {
    int lastLenght = 1;
    int maxLenght = 1;
    int total = 1;
    int index = 0;

    LinkedList.Node temp = list.head;

    do {
        if (temp.next != null) {
            if (temp.data < temp.next.data) {
                lastLenght++;
            } else {
                if (lastLenght > maxLenght) {
                    maxLenght = lastLenght;
                    index = total - lastLenght;
                }
                lastLenght = 1;
            }
            total++;
        }
        temp = temp.next;
    } while (temp != null);

    if (lastLenght > maxLenght) {
        maxLenght = lastLenght;
        index = total - maxLenght;
    }
    return makeLinkedList(list.head, index, maxLenght);
}
```

This algorithm traverse every element of the linked list from start to finish then it find the maximum length sorted sublist in this list and returns the head of the list and the number of elements. The function returned returns the new list. Each element in the linked list is visited at most once. Therefore the complexity of this function is **O (n).**

B)

```
public static LinkedList subListRecursive(LinkedList list) {
    int[] index = new int[4];//[0]lastLenght,[1]maxLenght,[2]total,[3]index
    index[0] = 1;
    index[1] = 1;
    index[2] = 1;
    index[3] = 0;

    index = subListRecursiveTail(list.head, index);


    if (index[0] > index[1]) {
        index[1] = index[0];
        index[3] = index[2] - index[1];
    }

    return makeLinkedList(list.head, index[3], index[1]);
}


public static int[] subListRecursiveTail(LinkedList.Node temp, int arr[]) {

    //[0]lastLenght,[1]maxLenght,[2]total,[3]index
    if (temp.next != null) {

        if (temp.data < temp.next.data) {
            arr[0]++;
        } else {

            if (arr[0] > arr[1]) {
                arr[1] = arr[0];
                arr[3] = arr[2] - arr[0];
            }

            arr[0] = 1;
        }
        arr[2]++;

        arr = subListRecursiveTail(temp.next, arr);
    }

    return arr;
}
```

This algorithm works with the same logic as iterative and the only difference is that the tail recursively coded. We will look at the tail for complexity this algorithm. The one-dimensional array in the function was used to easily transfer the variables to other sub-iterations. The wrapping function of already has a constant complexity. We can now write the recurrence relation. $\boldsymbol{T(n) = T(n-1) + O(8)}$ the $\boldsymbol{O(8)}$ is for inside constant code. Now we can the master theorem apply to this expression.

Let a ≥ 1 and b > 1 be constants, let f(*n*) be a function, and let T(*n*) be a function over the positive numbers defined by the recurrence

$T(n) = aT(n/b) + f(n)$.

If $f(n) = \Theta(n^d)$, where $d \geq 0$, then

- $T(n) = \Theta(n^d)$ if $a < b^d$,
- $T(n) = \Theta(n^d \log n)$ if $a = b^d$,
- $T(n) = \Theta(n^{\log_b a})$ if $a > b^d$.

a, b and d in our function a=1, b=2 and d=1

$a \ b^d \rightarrow 1=2^1 \rightarrow 1<2$ this equality fits the first rule.

$T(n) = \Theta(n^d)$ if $a < b^d$, from here

As a result $\boldsymbol{\Theta(n)}$

Now we can the induction apply to this expression.

T(1) = 8

T(n) = 8 + T(n-1)

T(n) = 8 + 8 + T(n-2)

T(n) = 8 + 8 + 8 + T(n-3)

.....

T(n) = k*8 + T(n-k)

k= n-1 => T(n) = (n-1)*8 + T(n-(n-1))

T(n) = (n-1)*8 + T(1)

T(n) = (n-1)*8 + 8

T(n) = 8n-8 + 8

T(n) = θ (n)

2.

```
boolean checkTwoElements(int arr[], int sum) {
    int first = 0;
    int last = arr.length - 1;

    while (first < last)
        if (arr[first] + arr[last] < sum)
            first++;
        else if (arr[first] + arr[last] > sum)
            last--;
        else {
            System.out.printf("Found it %d + %d = %d",
arr[first], arr[last], sum);
            return true;
        }

    System.out.printf("Doesn't exist two elements with given
sum %d", sum);
    return false;
}
```

The operation logic of the above function is as follows: this function sums the last element of the array with the first element. If the sum is greater than the desired value, the last element comes back, if the sum is less than the desired value, the first element goes forward. The loop end conditions; If function finds the desired value or first and last index intersect, the loop ends. The complexity of the function is **O (n).** Each element in the array is visited at most once.

3.

```
for(i=2*n;i>=1;i=i-1)
    for(j=1;j<=i;j=j+1)
        for(k=1;k<=j;k=k*3)
            print("hello");
```

The complexity of the inner loop is $\log_3 k$. *k* depends on *j*. *j* depends on *i*, we can write *n* instead of *k*. The final form of complexity $\log_3 2n$. The other two outer loops are interconnected. When we examine this connection, $\frac{2n(2n+1)}{2} = n(2n+1) = 2n^2 + n$ complexity comes. We have to multiply the complexities, because these three loops are running together. As a result of running time is $2n^2 + n \log_3 2n$ if we switch from here to big o notation, we find this $\boldsymbol{O(n^2 \log\ n)}$

4.

```
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;
}
```

The complexity of two loops in this function $\left(\left(\frac{n}{2}\right)-1\right)*\left(\left(\frac{n}{2}\right)-1\right)=\frac{n^2}{4}-n+1$ here comes $O(n^2)$. We can now write the recurrence relation. $\boldsymbol{T(n)=4T\left(\frac{n}{2}\right)+O(n^2)}$ the $O(n^2)$ is for inner 2 for loops. Now we can the master theorem apply to this expression.

Let a ≥ 1 and b > 1 be constants, let f(*n*) be a function, and let T(*n*) be a function over the positive numbers defined by the recurrence

T(*n*) = aT(*n*/b) + f(*n*).

If f(*n*) = Θ(n^d), where d ≥ 0, then

- T(*n*) = Θ(n^d) if a < b^d,
- T(*n*) = Θ(n^dlog *n*) if a = b^d,
- T(*n*) = Θ($n^{\log_b a}$) if a > b^d.

a, b and d in our function a=4, b=2 and d=2

a b^d → $4=2^2$ → 4=4 this equality fits the second rule.

T(*n*) = Θ(n^dlog *n*) if a = b^d, from here

As a result $\boldsymbol{\Theta(n^2 \log n)}$