Gebze Technical University
Computer Engineering

CSE 222 - 2018 Spring

HOMEWORK 4 (Part 1-4)

ALI HAYDAR KURBAN
151044058

Course Assistant: Ayse Serbetci Turan

PART 1

Part 1 a)

To solve this problem I implement my own linked list data structor. Its important and useful parts are “head” it represents head of my linked list, “data” it represents the item which is stored in my linked list, “next” it represent the next Node in my linked list, “add” it is a method which adds an element the last position in the my linked list.

My iterative method:

```
private static part1 part1Iterative(part1 list){
        part1 returnList = new part1();
        Node temp = list.head;
        Node returnTemp = list.head;
        Node currentTemp = list.head;

        int sizeMax = 0;
        int currentSize = 1;

        while(temp.next != null){
            if(temp.data <= temp.next.data){
                currentSize++;
                temp = temp.next;
            }

            else{
                if(currentSize > sizeMax){
                    sizeMax = currentSize;
                    returnTemp = currentTemp;
                }

                currentTemp = temp.next;
                temp = temp.next;
                currentSize = 1;
            }
        }

        if(currentSize > sizeMax){
            sizeMax = currentSize;
            returnTemp = currentTemp;
        }

        for(int i = 0; i < sizeMax; ++i){
            returnList.add(returnTemp.data);

            returnTemp = returnTemp.next;
        }
        return returnList;
    }
```

First of all I create an new linked list object to return maximum length sorted sublist, then I create three temp temporary Node which are temp, returnTemp, currentTemp. temp is used to reach each element of the list and its initial reference is head of the list. currentTemp is used to store start of sublist of at a moment and its initial reference is head of the list. returnTemp is used to return maximum length sorted sublist and its initial reference is head of my list. Then a create two variables which are sizeMax and currentSize. sizeMax represent the lenght of maximum length sorted sublist, currentSize represents the leght of sublist of at a moment.

Then I start a loop, which reaches all element in the list. If an element is smaller than the next of itself then increase the currentSize value and skip the next element. Otherewise check the sizeMax and currentSize which is bigger, if currentSize is bigger then make it sizeMax and make the returnTemp currentTemp and make the currentSize one. Then make the currentTemp and temp next element.

Finally with a outside loop add all elemts the maximum length sorted sublist in my linked list which was created at the top. Then return the my linked list.

It has **O(n)** complexity “n” means that number of element inside of the list, the reaseon of being O(n) is controling the each element with next element of itself.

Part 1 b)

To solve this problem I use my own linked list data structor which is mentioned Part1 a. There are three methods which are part1Recursive, copyList, helperpart1Recursive. part1Recursive only calls helperpart1Recursive method to solve given problem, copyList is called by helperpart1Recursive, it add all elemts the maximum length sorted sublist in my linked list.

My recursive method:

```
private static part1 part1Recursive(part1 list){
    return helper_part1Recursive(list.head, list.head, list.head,0,1);
}
private static part1 copyList(part1 returnList ,Node temp, int sizeValue) {
    if(sizeValue == 0)
        return returnList;
    else{
        sizeValue--;
        returnList.add(temp.data);
        return copyList(returnList,temp.next, sizeValue);
    }
}
```

```
private static part1 helper_part1Recursive(Node temp, Node returnTemp,
                                                                 Node currentTemp, int sizeMax, int currentSize){

        part1 returnList = new part1();

        if(temp.next == null){
            if(currentSize > sizeMax){
                sizeMax = currentSize;
                returnTemp = currentTemp;
            }

            return copyList(returnList,returnTemp,sizeMax);
        }

        else {
            if(temp.data <= temp.next.data){
                currentSize++;
                temp = temp.next;
            return helper_part1Recursive(temp,returnTemp,currentTemp,sizeMax,currentSize);
            }

            else{
                if(currentSize > sizeMax){
                    sizeMax = currentSize;
                    returnTemp = currentTemp;
                }
                currentTemp = temp.next;
                temp = temp.next;
                currentSize = 1;

            return helper_part1Recursive(temp,returnTemp,currentTemp,sizeMax,currentSize);
            }
        }
    }
```

The best case is the element is last element of the list. If currentSize is bigger than sizeMax make it sizeMax and make the returnTemp currentTemp. Then call the copyList method.

Other case, if an element is smaller than the next of itself then increase the currentSize value and skip the next element then recall itself. Otherwise check the sizeMax and currentSize which is bigger, if currentSize is bigger then make it sizeMax and make the returnTemp currentTemp and make the currentSize one. Then make the currentTemp and temp next element. Then recall itself.

It has O(n) complexity. Lets analyze it

Complexity by using induction:

$$T(n) = \begin{cases} \text{Theta}(m), & \text{if } (n == 1) \\ A + 2*T(n - 1), & \text{if } (n != 1) \end{cases}$$

A is a constant, it is if else and other instruction. It is not relevant with size of linked list. So I can call it A.

Theta(m) is copyList complexity, worst senario it is Theta(n) otherwise smaller than Theta(n).

n means that the number of element from temp(mentioned Part1 a) to null. 2*T(n - 1) means that there are two recursive call.

T(n) = A + 2*T(n - 1)
= A + A + 2*T(n - 2)
= A + A + A + 2*T(n - 3)

T(n) = k*A + 2*T(n - k)

Lets make k = n -1.

T(n) = (n - 1)*A + 2*T(1)

T(1) = O(n)

T(n) = n*A – A + 2*n so that it is **O(n).**

Complexity by using Master Theorem:

T(n) = A + 2*T(n - 1)

Master theorem works

T(n) = a*T(n / b) + f(n) when a >= 1 and b > 1.

In my station b is 1 so that Master Theorem can not be used.

PART 2

```
private static boolean part2Method(int A[], int sum)
{
    int startArray = 0;
    int endArray = A.length - 1;

    for(int i = 0; i < A.length; ++i){
        if (A[startArray] + A[endArray] == sum)
            return true;
        else if (A[startArray] + A[endArray] < sum)
            startArray++;
        else if(A[startArray] + A[endArray] > sum)
            endArray--;
    }
    return false;
}
```

The given array is sorted, it means that minimum element is in the index of 0, maximum element is in the index of array.length – 1. To solve this problem I create two variable which are startArray its initial valuse is 0, endArray its initial value is array.length – 1.

My method has startArray and endArray variable. Inside my method there is a for loop “for(int i = 0; i < array.length; ++i)”. Inside the loop there are three if statements which are make the method Theta(n) complexity.

Fist statement:

If number1 which is startArray index inside the array plus number2 which is endArray index inside the array are equal the given x value, return true.

Second statement:

If number1 which is startArray index inside the array plus number2 which is endArray index inside the array are smaller than the given x value, lets increase the startArray. The array is sorted when you increase the index you will get bigger number.

Third statement:

If number1 which is startArray index inside the array plus number2 which is endArray index inside the array are bigger than the given x value, lets decrease the startArray. The array is sorted when you decrease the index you will get smaller number.

The method in best case has O(n) complexity and in the worst case has Omega(n) complexity. Cuase of these and three statements the method has **Theta(n)** complexity.

PART 3

2*n + (2*n – 1) + (2*n - 2) + (2*n - 3) + …. + (2*n – 2*n)

This sum = ((2*n) * (2*n)) + (2*n * (2*n + 1)) / 2

It is 2*n^2 – n.

Final loop works log(2*n).

So that it is (2*n^2 – n) * log(2*n). It is **O(n^2 * log(n)).**

PART 4

First of all calculate the loop number in given aFunc function to solve this problem. There is a nested for loop. Totally it work ((n / 2) – 1) * ((n / 2) – 1) times. It means O(n^2).

$$T(n) = \begin{cases} O(1), \text{ if } (n == 1) \\ f(n) + 4*T(n / 2), \text{ if } (n > 1) \end{cases}$$

If(n == 1), it it best case, it only return so that O(1) complexity.

This 4, “4*T(n / 2)” means that there are 4 recursive calls.

Lets find f(n), inside the nested for loop there are four instruction. Each of them works ((n / 2) – 1) * ((n / 2) – 1) times, and there are some constant operation such as addition division, let call them c. so that totally 4 * (((n / 2) – 1) * ((n / 2) – 1) + c). We can throw constant and lower level terms.

So that f(n) = O(n^2).

T(n) = 4*T(n / 2) + n^2

Lets use Master Theorem

T(n) = a*T(n / b) + f(n)

a = 4, b = 2, d = 2

a = b^d, so that O(n^d * log(n))

So that the complexity of given problem is **O(n^2 * log(n)).**