GTU Department of Computer Engineering
CSE 222/505 – Spring 2019
Homework 04
Solutions 1-4

171044039
Ahmed Semih ÖZMEKİK

1. Given a single linked list of integers, we want to find the maximum length sorted sublist in this list. For example for the list L= {1, 9, 2, 7, 20, 13} the returned list should be S = {2, 7, 20}.

a) Iterative method which performs this task:

```
private static LinkedList<Integer> iterative(LinkedList<Integer> list)
   {
      // Maximum length sorted sublist.
      LinkedList<LinkedList<Integer>> subLists = new LinkedList<>();

      int index = 0;
      int prevNum = list.get(index);

      subLists.add(new LinkedList<>());
      for (Integer num:list){
         if (num>=prevNum)
            subLists.get(index).add(num);
         else{
            ++index;
            subLists.add(new LinkedList<>());
            subLists.get(index).add(num);
         }
         prevNum = num;
      }

      int maxLength = subLists.get(0).size();
      index = 0;
      int i=0;
      for (LinkedList<Integer> sub :subLists){
         if (maxLength<sub.size()){
            maxLength = sub.size();
            index = i;
         }
         ++i;
      }

      return subLists.get(index);

   }
```

Let’s analyze the complexity of this method:

We will try to find T(n) for “n” is the number of elements in the linked list. We must keep in mind that “index” variable is the number of sorted sub lists inside the list. In this algorithm, we find all the sorted sub list in the list and add them to new temporary list of sorted sub list. And then, among these lists we just picked the longest one.

And we will be considering the worst case scenario which can make the “index” is “n” at most. In essence, this is the case which there is not any sorted sub lists in the list or there is at least one but exists at the tail or close to tail. As we want to underestimate, I pick the worst case. That makes “index” is “n”.

$T(n) = n^* \max(T_{if}(n), T_{else}(n)) + n$
$T_{if}(n) = n$
$T_{else}(n) = n$
$T(n) = n^2+n$
$T(n) = \Theta(n^2)$

b) Recursive method which performs this task:

```
private static LinkedList<Integer> recursive(LinkedList<Integer> list)
   {
      // List of sorted sublist.
      LinkedList<Integer> subList1 = new LinkedList<>();
      LinkedList<Integer> subList2 = new LinkedList<>();

      int index = 0;
      int prevNum = list.get(index);

      Iterator<Integer> it = list.iterator();
      while(it.hasNext()){
         int num = it.next();
         if (num>=prevNum){
            subList1.add(num);
            it.remove();
         }
         else
            subList2 = recursive(list);
         prevNum = num;
         ++index;
      }

      if (subList1.size()>subList2.size())
         return subList1;
      else
         return subList2;

   }
```

Let’s analyze this method’s time complexity:

In our recursion algorithm, we consider two lists. For each sub routine, the one with bigger size is the longest sorted list in that sub routine. If we say for each recursion call we have “x” variable which is the length of the sorted sub list, our T(n) becomes:

$T(n) = x + T(n-x)$

In first “if” block, we construct our first sub list which is constant time complexity. And we have came up with recurrence relation now, which is quite tough to deal with. Because we are not sure of the value of “x” which can vary. But let’s say for the worst case scenario we don’t have any sub sorted lists in the list. For this case, our “n” many sub lists have only one value which corresponds to one element in the real list. In essence, “x” is being “1” for this case.

$T(n) = 1 + T(n-1)$
$T(n) = \Theta (n)$

2. Describe and analyze a Θ (n) time algorithm that given a sorted array searches two numbers in the array whose sum is exactly x.

```
private static void search(int[] arr, int sum)
  {
     int head = 0;
     int tail = arr.length-1;

     while(head<tail){
        int tempSum = arr[head]+arr[tail];
        if (tempSum == sum)
           break;
        else if (tempSum<sum)
           ++head;
        else
           --tail;

     }

     System.out.printf("(%d, %d)", arr[head],arr[tail]);
  }
```

Let’s analyze this algorithm:

We will try to find T(n) for “n” is the number of elements in the array. First of all, our algorithm relies on the fact that the array is sorted. Array being sorted means that, when we are trying to find the sum of two numbers inside it, we can and have to search wisely to visit each element only once at most. We iterate over the array both from the beginning and the end of the array. The number we are going to compare with “x” is the sum of those two points. Let’s say the number we have came up with is less than the “x”. Since, from the beginning to end, number increases and we can go one step further in the beginning and try again with a greater number. Or else, when our guess is smaller, we can come one step back from end, and try with smaller number.

And since each element is visited only and only once:
T(n) = Θ (n)

3. Calculate the running time of the code snippet below:

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

[1] n
[2] n*(n+1)/2
[3] logn

T(n) = $n^2\log(n)$

4. Write a recurrence relation for the following function and analyze its time complexity T(n).

T(1) = 1
T(n) = (n/2-1)*(n/2-1) + 4*T(n/2)
T(n) = $(n^2/4)$ + 4T(n/2)
Using Master Theorem: (T(n) = a*T(n/b) + f(n))
d = 2, a = 4, b = 2.
T(n) = Θ $(n^{2\log n})$