Algorithm Design & Analysis - Assignment 6

Michele Fattoruso - Z1840898

November 13, 2018

1 Longest monotonically increasing subsequence

Give an $\mathcal{O}(n^2)$ time algorithm to find the longest monotonically increasing subsequence of a sequence of n integers.

1.1 Explanation

The naive approach to the problem, could be to generate every possible subsequence of the n numbers given in input, which would require $\mathcal{O}(2^n)$ time. To improve the runtime of the algorithm we will try to solve the problem with a dynamic programming approach.

What we know from the problem, is that we need to find a subsequence of numbers where given a position i in the array, we add 1 to the length of the longest subsequence, if the element in position j, with j¿i is bigger than the element in position i.

To solve this problem we will use an additional array, which we will use to save the longest subsequence found for the numbers from position 0 to position i.

We will initialize the first element of the array to 1, as the subsequence of length 1 has max length 1, and the rest of the array to zero. From this longest subsequence of the array of size 1, so we can continue analyzing the longest subsequence for the array of size 2, so composed of the elements in position 0 and 1. if we find that the element in position 1 is bigger than the element in position 0,then the current iteration value will be equalto the value of the longest subsequence of max index 0, with the current maximum length.

Iterating this approach out algorithm will build the maximum length of the subsequence.

To solve this problem in $\mathcal{O}(n^2)$ time, we will proceed with a dynamic programming approach to the problem.

1.2 Code

```
def LongestMonotoneSubstring(self, nums):
    n = len(nums)
    if(n==0):
        return 0;
    #Initialize array to zeros, first element to 1
    values = [0] * n
    values[0] = 1

    longest=1
    for i in range(1,n):
        currStepValue=0
        for j in range(i):
            if(nums[i] > nums[j]):
                currStepValue=max(currStepValue,values[j])
        values[i] = currStepValue + 1
        longest = max(values[i],longest)
    return longest
```

1.3 Complexity

It was asked to solve the problem with a time complexity of $\mathcal{O}(n^2)$, and the proposed algorithm satisfy the requirements, as it performs a double linear scan over the whole array. Additionaly, as no constraint was set for the space complexity, we used an additional array of size n to solve the problem, so with a space complexity $\mathcal{O}(n)$

2 Smallest set of unit-length closed intervals

Describe an efficient algorithm that, given a set {x1, x2, . . . , xn} of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct.

2.1 Introduction

I was not sure if the problem required to calculate the size of the smallest set, or to calculate the set itself, So I've coded both solutions.

2.2 Explanation

As my understanding of the problem, the received input is already sorted, and that allowed me to solve the problem through a greedy approach. The Idea behind the greedy approach solution is that, given the first element in the array X, we can define a unit length closed interval, by simply adding 1 to the element, which gives us the interval [X,X+1]. From that point, if the next element fits in the range, then we continue analyzing the input array, if the element instead is outside of the interval, so if it's bigger than X+1, then we know that we found the starting point of another interval.

2.3 Code - Return size

```
def longestUnitInterval(nums):
    if(len(nums)==0):
        return 0
    setmax = nums[0]+1
    setsize=1
    for x in nums:
        if x > setmax:
            setmax = x+1
            setsize+=1
    return setsize
```

2.4 Output

```
Longest set Interval:  4
```

2.5 Code - Return a set

```
def getLongestUnitInterval(nums):
    if(len(nums)==0):
        return 0
    setmax = nums[0]+1
    tempArr = []
    finalList = []
    for x in nums:
        if x > setmax:
            finalList.append(tempArr)
            tempArr = [x]
            setmax = x+1
        else:
            tempArr.append(x)
    finalList.append(tempArr)
    return finalList
```

2.6 Output

```
[[0.7, 1.0, 1.5], [2.0, 2.3, 2.6], [3.1, 3.6, 3.9], [4.2, 4.7, 5.2]]
```

2.7 Complexity

The proposed algorithm is able to return the correct answer by simply performing a linear scan on the input. This will give us a time complexity of $\mathcal{O}(n)$ for both the proposed algorithms. The space complexity of the two algorithms instead is different, as in the first algorithm which returns just the minimum size of the set, we don't use any additional data structure, so the space complexity is $\mathcal{O}(1)$, instead in the algorithm which returns the devided set instead, we allocate extra memory to build the new set, so the space complexity is $\mathcal{O}(n)$.

Also, we should note that this algorithm works only for ordered sequence of numbers. If an unordered sequence of numbers is to be received, then we will need to first perform a sort of the sequence. In that case, the cost of the overall procedure would be dominated by the cost of the sort, so the time complexity would be $\mathcal{O}(Sort)$.