HW 6 Solution

CS/ECE 374 B: Algorithms & Models of Computation, Spring 2020 Version: **1.0**

1 Problem 16:

Solution 16.A:

Algorithm 1 Select top kth element

Input : An array of number of posts, kth smallest number
Output: The kth smallest number in array

```
Function selectTop(A[1...n], k):
    if n ≤ 100 then
        use brute force
    else
        t = ⌈n/100⌉
        for i ← 0 to t do
            T[i] ⟵ TopKPer(A [100i − 99...100i])
        end
        temp ⟵ selectTop(T[1...t], tk/n)
        r ⟵ partition(A [1...n], temp)
        if k < r then
            return selectTop(A [1...r − 1], k)
        if k > r then
            return selectTop(A [r + 1...n], k − r)
        else
            return r
        end
    end
r1 ⟵ selectTop(A[1...n], 0.95n)
r2 ⟵ selectTop(A[1...n], 0.9n)
total1 ⟵ sum(A[r1...n])
total2 ⟵ sum(A[1...r2])
if total1 ≥ 10 ∗ total2 then
    return True
else
    return False
end
```

In the selectTop(A, k) function, we used the same idea as momselect covered in the lecture that allow us to compute the kth smallest element in the array with the time complexity of $O(n)$. Therefore, with the same idea, we can use this algorithm to choose the 0.95n^{th} and 0.9n^{th} largest element in the array in $O(n)$. Then, without the loss of generality, we can assume that the array is partitioned after selecting. We can directly add up the elements on the one side of the selected position to calculate the total value, which is $O(n)$. In this way, we would be able to check if the top 5% students' posts are more than the bottom 90% of the posts with the time complexity of $O(n)$.

Solution 16.B:

Algorithm 2 Compute the total p% students' posts

```
Input  : An n × l matrix where n = $2^l - 1$
Output: The missing bit string in the matrix
1  Function selectTopK(p[1...k], A[1...n], output[1...k]):
2      if k = 1 then
3          r ⟵ selectTop(A[1...n], $(1 - p[k])n$)
4          output[$k$] ⟵ sum(A[$r$...n])
5          return
6      mid ⟵ $\lceil \frac{k}{2} \rceil$
7      r ⟵ selectTop(A[1...n], $(1 - p[mid])n$])
8      output[mid] ⟵ sum(A[$r$...n])
9      selectTopK(p[1...$mid$], A[1...$r$], output[1...k])
10     selectTopK(p[$mid + 1$...k], A[$r + 1$...n], output[1...k])
```

In this algorithm, we utilize the idea of binary search on the array p. We can split both the array A and p, where A contains the number of posts and p contains the percent that we need to calculate, into two parts, one part larger than the mid percentile, for both A and p, and the other part is the smaller ones. Then we start a modified binary search on these two parts.

Since we already know that for given percentile, we can calculate the result in $O(n)$. In this way, we can solve this problem by recursion tree. First of all, the recurrence of this algorithm is $T(n, k) = T(n_1, \frac{k}{2}) + T(n_2, \frac{k}{2}) + O(n)$, where $n_1 + n_2 = n$. Therefore, for each level of recursion tree, we need to traverse through the whole array to solve the subproblem. And there are $\log_2 k$ levels in total since we are dividing the length of array p into half in every recursion. Therefore, the time complexity of this algorithm is $O(n) * \log_2 k = O(n \log k)$.

2 Problem 17:

Solution 17:

Algorithm 3 Determine if w is in L

```
Input  : A string w
Output: Whether the string is in L
Function IswInL(w[1...]):
    for $i \leftarrow 1$ to $n$ do
        if IsStringInL(w[$1...i$], $L_1$) & IsStringInL(w[$i+1...n$], $L_2$) then
            return True
    end
    return False
Function IswInL*(w[$1...n$]):
    boolean ISL [$1...(n+1)$]
    ISL [$n+1$] $\longleftarrow$ $True$
    for $i \leftarrow n$ to $1$ do
        ISL [$i$] $\longleftarrow$ $False$
        for $j \leftarrow i+1$ to $n+1$ do
            if ISL[$j$] & IswInL(w[$i...j-1$]) then
                ISL [$i$] $\longleftarrow$ True
                Break
        end
    end
    if ISL[1] then
        return True
    else
        return False
    end
```

We can solve this problem through text segmentation in the textbook. First in the function IswInL, we detect if w can be split into two parts where each part belongs to L1 and L2. Then we can use this as a subroutine to further construct an iterative algorithm to determine whether the string is in (L1 • L2)*. With the same reasoning with what covered in lecture, if w is empty string or w is in (L1 • L2), then it is automatically true. Then we need to iterate over w to determine if w can be split into two parts where the first part is in (L1 • L2) and the second part is in (L1 • L2)*.

For the analysis of this algorithm, given the length n of the string, there are $O(n)$ subproblems in total. And for each subproblem, we need $O(n^2)$ to solve it since for every check if the substring is in (L1 • L2) would require $O(n)$. Therefore, the time complexity of this algorithm is $O(n^3)$. And since we only need a one-dimensional array to store the values in iteration, the space complexity of this algorithm is $O(n)$.

3 Problem 18:

Solution 18:

Let **maxProfits(i, k)** represent the optimal k-shot strategy for max return algorithm give k intervals through day 1 to i. In this case, we can obviously get our base case as: if i < 2 or there are 0 intervals, the profits would automatically be 0. Now we will consider the case at day i, which is the last day of our algorithm. There are two cases on the last day, we can choose to sell on that day or not. If we do not sell on that day, then the result would be equal to **maxProfits(i - 1, k)**. On the other hand, we would sell on the last day. Then we need to find another optimal day j to buy the stock and consider the rest of the days from 1 to $j-1$. Therefore, we can write our recurrence for this algorithm as:

$$\textbf{maxProfits(i, k)} \begin{cases} 0, & \text{if } k = 0 \\ 0, & \text{if } i < 2 \\ \textbf{max} \begin{cases} \textbf{maxProfits(i - 1, k)} \\ \textbf{max}_{\text{ for } 1 \le j \le i-1}, 1000(p[i] - p[j]) + \textbf{maxProfits(j - 1, k - 1)} \end{cases} & \text{Otherwise} \end{cases}$$

Algorithm 4 Set up k-shot strategy for max return

Input : A string w

Output: Whether the string is in L

```
Function maxProfits(p[1...n]):
    MPT ⟵[0...n ][0...k ]
    for $i \leftarrow 0$ to n do
        for $j \leftarrow 0$ to k do
            if $i < 2$ $j = 0$ then
                MPT $[i][j]$ ⟵ 0else
                    MPT$[i][j]$ ⟵ MPT$[i-1][j]$
                    for $m \leftarrow i-1$ to 1 do
                        large ⟵ 1000(p$[i]$ − p$[m]$) + MPT$[m][j-1]$
                        if MPT$[i][j]$ < large then
                            MPT$[i][j]$ = large
                    end
                end
            end
        end
        return MPT[n][k]
```

As we can see from this algorithm, we store the value of this strategy for given day i and k intervals into a two-dimensional array. Since we have already known the recurrence is the right algorithm. Further proof of correctness of this iterative algorithm would not be necessary.

For the analysis of complexity, we can see that there are nk subproblems in total and the solution to each subproblem would require $O(n)$. Therefore, the time complexity for this algorithm is $O(n^2k)$.

For space complexity, the result is trivial since the only data structure we need is a two-dimensional array. Thus the space complexity for this algorithm is $O(nk)$.