HW 7 Solution

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

1 Problem 19:

Solution 19:

When subdividing the rectangle slab, there are three possible choices: no cut, divide horizontally and divide vertically. Therefore, at each step, we need to find the best subdividing method. If we choose to subdivide, we can recursively decide the maximum profits for the rest two parts. In this case, we can write the recurrence relation as:

$$\begin{aligned}\text{maxProfit(n, m)} = \max\{\ & p[n, m],\\ & max\{\text{for } 0 < i < \lfloor \tfrac{n}{2} \rfloor, \text{maxProfit(n - i, m) + maxProfit(i, m)}\}\\ & max\{\text{for } 0 < j < \lfloor \tfrac{m}{2} \rfloor, \text{maxProfit(n, m - j) + maxProfit(n, j)}\}\ \}\end{aligned}$$

With this recurrence, we can first write a program to find the maximum amount of profits using dynamic programming. Our base case here should be maxProfit(1,1) where we cannot make any cut to the rectangle slab. Therefore, the maximum profit should automatically be p[1][1]. From this base case, we can traverse the whole output array from 1 to n and calculate the maximum profit at each step. Since for every time traversing, we need to compare the values among horizontal cut, vertical cut and no cut, this algorithm requires a $O(m + n)$ complexity at each loop to calculate the maximum value. And for each cut, we store the method of subdividing and the point of cut with this value. Therefore, at each point in output array, we store a pair with the first element as current maximum profits and the second element as (cutting method, cutting point).

After calculating the maximum profits of a n × m rectangle slab, we need to walk backwards to find the whole subdividing sequence. Since we don't know exactly how many cuts we have made for a n × m rectangle slab, the best way for us to reach base case, where base case is no need for any more cut, is recursion. In this way, starting with the whole rectangle, along with the output array and an empty list to store the sequence, the moment we make a subdivide, we split the whole slab into two parts. After storing this cut into the sequence, we need to do recursion on these two rectangles to find the rest of the cuts.

For the analysis of this algorithm, as we can see, in the first part calculating the maximum profit, we have to traverse the whole n × m array and for each loop we need $O(m + n)$ to calculate the maximum value. Therefore, the time complexity for the first part is $O(mn(m + n))$. And for the second part recursion, we can only consider the worst case. In the worst scenario. we need n × m cuts to subdivide the whole rectangle into 1 × 1 square. Therefore, the maximum times of recursion is mn. And inside every recursion, we can consider these comparing calculations and append operation on list to be $O(1)$. Therefore, in the worst case, the time complexity of recursion is $O(mn)$.

In conclusion, the time complexity for this algorithm is $O(mn(m + n) + mn) = O(mn(m + n))$. Since we only need one n × m array to store the maximum profits of each step, the space complexity of this algorithm is $O(mn)$.

Algorithm 1 Find the optimal strategy for slicing a rectangle slab

```
Input  : An array containing the prices of rectangle of certain height and width
Output: The max strategy for subdividing the rectangle
Function maxProfit(Rec[0...$n$][0...$m$]):
    output ⟵ [1...$n$][1...$m$]
    for $i \leftarrow 1$ to $n$ do
        for $j \leftarrow 1$ to $m$ do
            if $i = j = 1$ then
                output [i][j] = (p [1][1], None)
            else
                All ⟵ p[$i$][$j$]
                H ⟵ 0
                for $a \leftarrow 1$ to $\lfloor \frac{n}{2} \rfloor$ do
                    if output[$a$][$j$] + output[$i-a$][$j$] > H then
                        H ⟵ output[$a$][$j$][0] + output[$i-a$][$j$][0]
                        h ⟵ $a$
                end
                V ⟵ 0
                for $b \leftarrow 1$ to $\lfloor \frac{m}{2} \rfloor$ do
                    if output[$i$][$j-b$] + output[$i$][$b$] > V then
                        V ⟵ output[$i$][$j-b$][0] + output[$i$][$b$][0]
                        v ⟵ $b$
                end
                max ⟵ max($All$, H, V)
                method ⟵ $max = H?max = V?max = All?(H,h) : (V,v) : None$
                output [i][j] ⟵ $(max, method)$
            end
        end
    end
    return output [n][m]
sequence ⟵ []
Function divideSeq(output[1...$n$][1...$m$], Rec[$x_{min}$...$x_{max}$][$y_{min}$...$y_{max}$], sequence):
    if output[$x_{max} - x_{min}$][$y_{max} - y_{min}$][1][0] = $None$ then
        return sequence
    if output[$x_{max} - x_{min}$][$y_{max} - y_{min}$][1][0] = $H$ then
        h ⟵ output[$x_{max} - x_{min}$][$y_{max} - y_{min}$][1][1]
        sequence.append(($H, h, x_{min}, x_{max}$))
        return divideSeq(output[1...$n$][1...$m$], Rec[$x_{min}$...$x_{max}$][$y_{min}, h$], sequence) +
         divideSeq(output[1...$n$][1...$m$], Rec[$x_{min}$...$x_{max}$][$h+1, y_{max}$], sequence)
    else
        v ⟵ output[$x_{max} - x_{min}$][$y_{max} - y_{min}$][1][1]
        sequence.append(($V, v, y_{min}, y_{max}$))
        return divideSeq(output[1...$n$][1...$m$], Rec[$x_{min}$...$v$][$y_{min}, y_{max}$], sequence) +
         divideSeq(output[1...$n$][1...$m$], Rec[$v+1$...$x_{max}$][$y_{min}, y_{max}$], sequence)
    end
return divideSeq(output[1...$n$][1...$m$], Rec[0...$n$][0...$m$], [])
```

2 Problem 20:

Solution 20:

As described in the context, we can get that if A[i] < A[i+1], then it is possible that after A[i], the rocks can contain some air. And with each step forward, the increased amount of air is A[j] - A[i] if there exists some k that A[k] ≤ A[i] and i < j < k. In this case, we can divide the problem into two cases. Generally, we can use two iterators to represent the positions we are at when finding the two boundaries that can contain air. Therefore, given the array A, we would be able to have a recurrence relation as follows:

$$\text{maxAir(i, j)} = \begin{cases} 0, & \text{if } i = j \text{ or } j > n \text{ or } j \leq 0 \\ \text{maxAir(i, j + 1)}, & \text{if } A[i] < A[j] \\ \text{maxAir(j, j + 1)} + (A[k] - A[i]) \text{for } i < k < j, & \text{if } A[j] \leq A[i] \end{cases}$$

With this recurrence relation, we can minimize the unnecessary calculations using dynamic programming to store the possible air and the maximum contained air at each step. Therefore, we can have our algorithm for this problem.

Algorithm 2 Find maxAir under rock algorithm

Input : An array containing the height of the rocks
Output: The total amount of air contained under the rocks

```
Function maxAir(A[1...n]):
    OPT ⟵[1...n]
    i ⟵ 0
    j ⟵ 0
    OPT [1] ⟵ 0
    temp ⟵ 0
    for j ← 0 to n do
        OPT[j] ⟵OPT [ j- 1]
        if A[i] < A[j] then
            temp ⟵ temp+ A [j ] - A [i ]
            j ⟵ j + 1
            continue
        if A[i] ≥ A[j] then
            OPT [j ] ⟵ OPT[j] + temp
            temp ⟵ 0
            i ⟵ j
            j ⟵ j + 1
    end
    return OPT [n]
```

With the analysis of the complexity of this algorithm, we can easily see that the time complexity is $O(n)$ since there is only one iterator j that traverses the array and i is just changing as j changes. For the space complexity, since all the values at each step is stored in a one-dimensional array. Thus, it is trivial to conclude that the space complexity of this algorithm is $O(n)$.

However, we can optimize this algorithm by not utilizing memoization since we would only want the final result of given array without the need of recording every step. In this case, the algorithm would be like:

Algorithm 3 Find maxAir under rock algorithm

Input : An array containing the height of the rocks

Output: The total amount of air contained under the rocks

```
Function maxAir(A[1...$n$]):
    result ⟵ 0
    temp ⟵ 0
    i ⟵ 0
    j ⟵ 0
    for j ← 0 to $n$ do
        if A[i] < A[j] then
            temp ⟵ temp+ A [j ] - A [i ]
            j ⟵ j + 1
            continue
        if A[i] ≥ A[j] then
            result ⟵ result + temp
            temp ⟵ 0
            i ⟵ j
            j ⟵ j + 1
    end
    return result
```

As we can see from this algorithm, with basically the same idea, we can replace the original output array with a single variable result. In this way, without increasing runtime complexity, we can decrease the space complexity of this algorithm to $O(1)$.

3 Problem 21:

Solution 21:

There are three moves at each position in total, where two for A and E. Thus, we can consider these five positions independently. Since we are given the positions of apples of each stage, we can construct an array with length n + 1 corresponding to each stage that 1 represents there is an apple at this position. For instance, for the example in the context, we can construct stage A as $[0, 0, 0, 0, 0, 1, 0]$, with a starting 0 to deal with cat-first-moving strategy. Therefore, after constructing the array in $O(n)$, we would trivially have the following recurrence relations:

$$
\begin{aligned}
&opt(K, i) = K[i] && \text{if i = n}\\
&opt(A, i) = max\{opt(A, i+1), opt(B, i+1)\} + A[i]\\
&opt(B, i) = max\{opt(A, i+1), opt(B, i+1), opt(C, i+1)\} + B[i]\\
&opt(C, i) = max\{opt(B, i+1), opt(C, i+1), opt(D, i+1)\} + C[i]\\
&opt(D, i) = max\{opt(C, i+1), opt(D, i+1), opt(E, i+1)\} + D[i]\\
&opt(E, i) = max\{opt(D, i+1), opt(E, i+1)\} + E[i]
\end{aligned}
$$

Algorithm 4 Find the most apples cat can catch

Input : Five stages with apples' positions as constructed above
Output: The max number of apples the cat can catch starting from stage C

```
Function opt(stage[A[0...n], B[0...n], C[0...n], D[0...n], E[0...n]]):
    output ⟵ [A...E][0...n]
    for $i \leftarrow n$ to 0 do
        if $i = n$ then
            output [A ][n] = A [n]
            output [B ][n] = B [n]
            output [C ][n] = C [n]
            output [D ][n] = D [n]
            output [E ][n] = E [n]
        else
            output [A ][i] = max {output [A ][i + 1], output [B ][i + 1]}
            output [B ][i] = max {output [A ][i + 1], output [B ][i + 1], output [C ][i + 1]}
            output [C ][i] = max {output [B ][i + 1], output [C ][i + 1], output [D ][i + 1]}
            output [D ][i] = max {output [C ][i + 1], output [D ][i + 1], output [E ][i + 1]}
            output [E ][i] = max {output [D ][i + 1], output [E ][i + 1]}
        end
    end
    return output [C ][0]
```

In this algorithm, each stage would be considered separately at each level so that the dependencies of the next step is filled before we move to the next step. And for the complexity of this algorithm, we can easily see that the only iteration here is i from n to 0, and inside each loop, the operation could be done in $O(1)$. Therefore, the time complexity of this algorithm is $O(n)$. Also, for the space complexity, the only space this algorithm requires is a two-dimensional array whose size is 5n. Thus, the space complexity of this algorithm is $O(n)$.