SUN Yilin
520030910361

Homework 4

Algorithms
April 30, 2022

1

1.1

The idea is to scan the array from left to right and update the maximum revenue meanwhile. See algorithm below for details.

Algorithm 1 Max Revenue-1,1($\boldsymbol{a}$)

```
1: procedure MAX REVENUE-1,1($\boldsymbol{a}$):                ▷ $\boldsymbol{a} = \{a_1, a_2, a_3, \ldots, a_n\}$
2:     result$\leftarrow 0$, sum$\leftarrow 0$, $i \leftarrow 1$      ▷ sum is our current optimal to maintain
3:     while $i < n + 1$ do
4:         sum$\leftarrow$sum$+a_i$
5:         if sum>result then
6:             result$\leftarrow$sum                          ▷ Update the optimal subsequence
7:         if sum<0 then
8:             sum$\leftarrow$0                               ▷ Discard the bad subsequence we do not want
9:         $i \leftarrow i + 1$
10:    return result
```

Since our algorithm only requires us to scan the array once, clearly the time complexity is $O(n)$.

1.2

The idea is still scanning the array from left to right once, but we will use an array $\boldsymbol{s}[n]$ to store the maximum revenue of a subsequence ending at position $i, i \in [n]$.
First we initialize $s_1, s_2, s_3, \ldots, s_L$. Then for each element $a_i, i \in [n]$, we need to update $R - L + 1$ elements in $\boldsymbol{s}[n]$, which are $s_{i+L}, s_{i+L+1}, \ldots, s_{i+R}$. The initialize rule and update rule are given in the algorithm below.

Algorithm 2 Max Revenue$(L, R, \boldsymbol{a})$

```
 1: procedure MAX REVENUE($L, R, \boldsymbol{a}$):                      ▷ $\boldsymbol{a} = \{a_1, a_2, a_3, \ldots, a_n\}$
 2:     result$\leftarrow 0$, $i \leftarrow 1$ $s_i \leftarrow 0, \forall i \in [n]$          ▷ $\boldsymbol{s}$ is described above
 3:     while $i < L + 1$ do
 4:         if $a_i > 0$ then
 5:             $s_i \leftarrow a_i$                          ▷ Initialization for the first $L$ elements
 6:         if $a_i >$result then
 7:             result$\leftarrow a_i$                          ▷ Update the result
 8:         $i \leftarrow i + 1$
 9:     $i \leftarrow 1$
10:     while $i + L < n + 1$ do
11:         step$\leftarrow L$
12:         while $i$+step$< n + 1$ do
13:             if $s_i + a[i$+step$] > s[i$+step$]$ then
14:                 $s[i$+step$] = s_i + a[i$+step$]$          ▷ update the optimal result at $i$
15:             if $s[i$+step$] >$result then
16:                 result$\leftarrow s[i$+step$]$
17:             step$\leftarrow$step$+1$
18:         $i \leftarrow i + 1$
19:     return result
```

We need to do $R - L + 1$ updates at each round and there are n rounds in total, so the time complexity is $O((R - L + 1)n)$. So with the difference between L and R approaching n, our algorithm actually becomes $O(n^2)$.

1.3

We still need to scan the array from left to right once, but instead of updating $R - L + 1$ elements at each iteration, we use a better strategy.
For each a_i, we look for the largest $s_j, i-R \leq j \leq i-L$ and use this largerst result to update s_i. Then by our algorithm in class, we can find all the largest s_j for our current a_i with only $O(n)$ time. A detailed algorithm is given below.

Algorithm 3 Max Revenue($L, R, \boldsymbol{a}$)

```
procedure MAX REVENUE($L, R, \boldsymbol{a}$):                          ▷ $\boldsymbol{a} = \{a_1, a_2, a_3, \dots, a_n\}$
    result$\leftarrow$ 0, $i \leftarrow 1$ $s_i \leftarrow 0, \forall i \in [n]$
    while $i < L + 1$ do
        if $a_i > 0$ then
            $s_i \leftarrow a_i$                    ▷ Initialization is the same as algorithm 2
        if $a_i >$result then
            result$\leftarrow a_i$                                   ▷ Update the result
        $i \leftarrow i + 1$
    $i \leftarrow L + 1$
    while $i < n + 1$ do
        max$\leftarrow$ k-Largest($\boldsymbol{s}, i - R, i - L$)    ▷ Algorithm in class to find max in $O(1)$ time
        if max$+a_i > s_i$ then
            $s_i \leftarrow$max$+a_i$
        if $s_i >$result then
            result$\leftarrow s_i$
        $i \leftarrow i + 1$
    return result
```

The reason why this algorithm is faster is that by using the k-Largest algorithm, we do not have to update $R-L+1$ times each round. We only need to look up the largest result before a certain element and the look up process is $O(1)$ for each element. As a result, the total running time of our algorithm can be reduced to $O(n)$.

2 Optimal Indexing for A Dictionary

Description of state transition equation:
In the algorithm below, we use $f(i,j), 1 \leq i \leq j \leq n$ to denote the minimum of total number of comparisons of the best binary search tree consisting of $a_i, a_{i+1}, \ldots, a_j$.
We then use dynamic programming to calculate all the $f(i,j)$ and our optimal result is $f(1,n)$ after we finish the calculations.
Note that we define $f(i,j) = 0$ if $i > j$.
The algorithm can be described by the folloing state transition equation:

$$f(i,j) = \min_{i \leq r \leq j} \left\{ f(i,r-1) + f(r+1,j) + \sum_{k=i}^{j} w_k \right\} \quad 1 \leq i \leq j \leq n$$

How we construct the optimal BST:
This equation only gives the criteria by which we pick the optimal BST each step, below is a detailed description of how the BST is constructed, for the completeness of our algorithm. We can always let T_{ij} be a sub-BST generated upon the process we calculate $f(i,j)$.

For starters, $T_{ii}, i \in [n]$ denote a sub-BST with only one vertex a_i as the root.
Then as we calculate the state transition equation, by finding the minimun

$$\min_{i \leq r \leq j} \left\{ f(i,r-1) + f(r+1,j) + \sum_{k=i}^{j} w_k \right\}$$

What we are actually doing is to find a new root a_r to serve as the father of sub-BST $T_{i,r-1}$ and $T_{r+1,j}$. By the time we calculate $f(i,j)$ and construct T_{ij}, no matter which a_r we choose as the root, the left sub-BST $T_{i,r-1}$ and right sub-BST $T_{r+1,j}$ must have already been constructed. Also, their optimal value must already have been stored in $f(i,r-1)$ and $f(r+1,j)$. Although in some cases the sub-BST might be empty, these cases do not affect our process of construction.
To sum up, as our DP algorithm finished, an optimal binary search tree T_{1n} is generated by our description above and its minimal number of comparisons is given by $f(1,n)$ meanwhile.

3

Description of state transition equation:
In the algorithm below, we use $f(i,j), 1 \leq i \leq j \leq n$ to denote the maximum length of palindrome in the interval $[i,j]$ (not necessarily all elements in $[i,j]$ are used).

For starters, we initialize $f(i,i) = 1, \forall i \in [n]$ because all subsequence of length 1 are palindromes.
Then we calculate $f(i,j), 1 \leq i < j \leq n$ by the following state transition equation.

$$f(i,j) = \max\{2 \cdot \mathbb{1}(s_i = s_j) + f(i+1, j-1), f(i+1,j), f(i,j-1)\} \quad 1 \leq i < j \leq n$$

where $\boldsymbol{s} = s_1, s_2, s_3, \ldots, s_n$ is the given input string.

How the longest palindrome is given:
We use a_{ij} to denote the palindrome subsequence upon interval $[i,j]$ and describe how we update the sequence.
For starters, $a_{ii} = s_i, \forall i \in [n]$ as we initialize $f(i,i)$. Then when we calculate $f(i,j)$, when we choose the maximum of the three terms, we modify a_{ij}.
$\forall i,j$ such that $1 \leq i < j \leq n$
If

$$f(i,j) = 2 \cdot \mathbb{1}(s_i = s_j) + f(i+1, j-1)$$

then $a_{ij} = s_i a_{i+1,j-1} s_j$, i.e. we generate a nested palindrome subsequence a_{ij} by adding two letters at the beginning and at the end of $a_{i+1,j-1}$.
If

$$f(i,j) = f(i+1,j)$$

then $a_{ij} = a_{i+1,j}$.
If

$$f(i,j) = f(i,j-1)$$

then $a_{ij} = a_{i,j-1}$.
Finally the optimal palindrome subsequence is given by a_{1n} with length $f(1,n)$.
Running time:
Our algorithm requires us to fill in a table of $n * n$ where each calculation can be finished by comparing three numbers, which can be done in $O(1)$ time. So the total running time is $O(n^2)$.

4

4.1

Algorithm Description:
Denote the number of independent sets in a subtree rooted at v by $f(v)$, we can calculate by the following state transition equation.

$$f(v) = \prod_{w\text{:grandchild of } v} f(w) + \prod_{w\text{:child of } v} f(w)$$

Then let the root of given tree G be r, our result is given by $f(r)$. By our definition of algorithm, we can calculate $f(r)$ recursively.
As for the initial states of recursion, all the leaves l of original graph G should have $f(l) = 1$.
Running Time:
We are going to calculate $f(r)$ recursively. Since all the sub trees are independent of each other, their calculation process will not need information of each other. As a result, for any single vertex v of G, $f(v)$ need and only need calculating once. Since we assume it takes $O(1)$ time to store and multiply two integers, the total running time is constant times total number of calculations of $f(v)$. So the running time is $O(n)$.
Proof of correctness:
We prove by induction.
Clearly the base step is correct. We only need to show at each induction step, our state transition equation can actually give us a correct result.
Suppose the current root of our subtree T_v is v, we now want to calculate the number of independent sets in T_v. There are two kinds of indepent sets, we can calculate them respectively and add them together.
First are the independent sets which include the root v. Since v is in those independent sets, their children cannot be in the sets, but we can choose from the independent sets of the grandchildren of v freely. Each grandchild w of v has $f(w)$ independent sets, so the total number is multiplying all those $f(w)$ together, as shown by our equation.
Second are the independent sets which do not include the root v. Since v is not in those independent sets, we can choose from the independent sets of the children of v freely, similarly, the total number is multiplying $f(w)$ where w is a child of v.
Finally by adding up these two kinds of independent sets together, we get the total number of independent sets rooted at v.
So the induction process can actually give us the correct result at each iteration, finally $f(r)$ is the result we want.

4.2

Algorithm Description:
Denote the number of maximum independent sets in a subtree rooted at v by $f(v)$, denote the size of maximum independent set in the same subtree by $g(v)$. We initialize $f(l) = g(l) = 1$ for all the leaves l in the tree G and calculate $f(v), g(v)$ according to the following state transition equations.

$$g(v) = \max\left\{\sum_{w:\text{child of } v} g(w), \quad 1 + \sum_{w:\text{grandchild of } v} g(w)\right\}$$

$$f(v) = \begin{cases} \prod_{w:\text{child of } v} f(w), \quad \text{if} \quad \sum_{w:\text{child of } v} g(w) > 1 + \sum_{w:\text{grandchild of } v} g(w) \\ \prod_{w:\text{grandchild of } v} f(w), \quad \text{if} \quad \sum_{w:\text{child of } v} g(w) < 1 + \sum_{w:\text{grandchild of } v} g(w) \\ \prod_{w:\text{child of } v} f(w) + \prod_{w:\text{grandchild of } v} f(w), \quad \text{otherwise} \end{cases}$$

Then let the root of given tree G be r, our result is given by $f(r)$.
Running Time:
Likewise, we are going to calculate $f(r)$ and $g(r)$ recursively. Since all the sub trees are independent of each other, their calculation process will not need information of each other. As a result, for any single vertex v of G, we first calculate $g(v)$ and $g(v)$ need and only need calculating once. During the process we calculate $g(v)$, we can know the larger of the two numbers we are comparing. This means that although there are three cases of $f(v)$, when we are actually calculating $f(v)$, we do not need to check which case it is because we already know when we were calculating $g(v)$ for the same v. Since we assume it takes $O(1)$ time to store and multiply two integers, the total running time is constant times total number of calculations of $f(v)$ plus $g(v)$, which remains $O(n)$.
Proof of correctness:
The method of calculation of number of maximum independent sets is exactly the same as the case where we calculate independent sets, we only need to prove that at each iteration, we are picking the actual maximum independent set in the subtree T_v rooted at v.
The size of maximum independent set at T_v defined as $g(v)$ and $g(v)$ is given by

$$\max\left\{\sum_{w:\text{child of } v} g(w), \quad 1 + \sum_{w:\text{grandchild of } v} g(w)\right\}$$

This equation means that the size of maximum independent set is either $\sum_{w:\text{child of } v} g(w)$ if we do not choose r in the set, or $1 + \sum_{w:\text{grandchild of } v} g(w)$ if we choose r in the set.
By picking the larger of these two sizes we guarantee that our outcome is optimal. So we can make sure that the final size of maximum independent set is correct. Then the counting of number of such sets is also correct by our proof in problem 4.1.

5 Comments

5.1

Although the DP algorithms are more difficult than the greedy algorithms, this assignment does not require proof of correctness for most of the problems. As it turns out, this time I took about a whole day to finish the assignment.

5.2

Q1→3. The intuition is straightforward and we can use the hint. By applying the algorithm from class, it is not hard to figure out this problem.
Q2→5. I myself spent great time on thinking about the state transition equation of this dynamic programming. It is difficult, from my perspective, because the relying relationship between the states is not that easy to figure out.
Q3→4. This problem is also a difficult one. But based on the algorithm from class about DP on intervals, it is not that hard as question 2.
Q4→3. Since we have learnt the algorithm to find the size of maximum independent set in class, the algorithm for this problem can be built from that algorithm and it is not very demanding to figure out solution for this problem.

5.3

I got all the ideas of these algorithms from class or our lecture notes, but I indeed talked with some of my classmates for the simplicity of description of algorithm and proof of correctness. But the actual work is finished on my own.