SUN Yilin
520030910361

Homework 4

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($\boldsymbol{a}$)

procedure MAX REVENUE-1,1($\boldsymbol{a}$): ▷ $\boldsymbol{a} = \{a_1, a_2, a_3, \ldots, a_n\}$
    result← 0, sum← 0, $i \leftarrow 1$ ▷ sum is our current optimal to maintain
    while $i < n + 1$ **do**
        sum←sum+a_i
        if sum>result **then**
            result←sum ▷ Update the optimal subsequence
        if sum<0 **then**
            sum←0 ▷ Discard the bad subsequence we do not want
        $i \leftarrow i + 1$
    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$(L, R, \boldsymbol{a})$

procedure MAX REVENUE$(L, R, \boldsymbol{a})$: ▷ $\boldsymbol{a} = \{a_1, a_2, a_3, \ldots, a_n\}$
    result$\leftarrow 0$, $i \leftarrow 1$ $s_i \leftarrow 0, \forall i \in [n]$ ▷ $\boldsymbol{s}$ is described above
    while $i < L + 1$ **do**
        if $a_i > 0$ **then**
            $s_i \leftarrow a_i$ ▷ Initialization for the first L elements
        if $a_i >$result **then**
            result$\leftarrow a_i$ ▷ Update the result
        $i \leftarrow i + 1$
    $i \leftarrow 1$
    while $i + L < n + 1$ **do**
        step$\leftarrow L$
        while i+step$< n + 1$ **do**
            if $s_i + a[i\text{+step}] > s[i\text{+step}]$ **then**
                $s[i\text{+step}] = s_i + a[i\text{+step}]$ ▷ update the optimal result at i
            if $s[i\text{+step}] >$result **then**
                result$\leftarrow s[i\text{+step}]$
            step$\leftarrow$step+1
        $i \leftarrow i + 1$
    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($L, R, \boldsymbol{a}$): ▷ $\boldsymbol{a} = \{a_1, a_2, a_3, \ldots, a_n\}$
    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 $O(n^2)$.