# 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(a):                ▷ a = {a_1, a_2, a_3, ..., a_n}
    result ← 0, sum ← 0, i ← 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 ← 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** $s_i + a[i$+step$] > s[i$+step$]$ **then**
                $s[i$+step$] = s_i + a[i$+step$]$ ▷ update the optimal result at $i$
            **if** $s[i$+step$] >$result **then**
                result$\leftarrow s[i$+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($L, R, \boldsymbol{a}$): ▷ $\boldsymbol{a} = \{a_1, a_2, a_3, \dots, a_n\}$
  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 $O(n)$.