CS 341: Assignment 4

x969li

Q2. [10 marks] Treasure Collector.

Let $A = [(v_1, f_1), (v_2, f_2), ..., (v_n, f_n)]$ denotes the input array.

Subproblem:
$M_i[j+1, k]$ is the maximum value of collection of size at most j, on floor at most k at the end of i^{th} round.
For all $i \in [1, n], j \in [0, m], k \in [1, h-1]$,
Base case: $M_0[j+1, k] = 0$.
At i-th round, we can either collect the treasure or not. Thus, we have
$M_i[j+1, k] = \max(M_{i-1}[j+1, k], M_{i-1}[j, k] + v_i)$.
Also, $M_i[j+1, k] \geq M_i[j, k]$ and $M_i[j+1, k+1] \geq M_i[j+1, k]$. We will prove that these two invariants hold below.

$M_n[m+1][h]$ has the desired result.

Description of Algorithm:

1. [For better efficiency] If $h >> n$, we can create a temporary array A' from A, where $A' = [(f_1, 1), (f_2, 2), ..., (f_n, n)]$.
   Then we sort A' based on the value of f. We should use merge sort algorithm, which is stable. Denote the result array as $[(f'_1, i_1), (f'_2, i_2), ..., (f'_n, i_n)]$. The resulting array should satisfy $f'_1 \leq f'_2 \leq ... \leq f'_n$.
   At last, we assign a new index j for each (f'_j, i_j), and modify A such that $A[i_j] = (v_{i_j}, j)$ for all $j \in [1, n]$. This allows us to bound the value of h and set h to be n.
2. Initialize the $(m+1) \times h$ 2D-array M and set all of its entries to be zero.
3. We have to start checking the treasure appeared in the first round, and update the maximum value for $M[j][k]$ for all $j \in [1, m], k \in [1, h]$, until we have iterated through all the rounds.
4. At each round i, the treasure appeared in this round might affect all $M[j+1][k]$ where $j \in [1, m], k \in [f_i, h]$.
   Since $M_i[j+1, k] = \max(M_{i-1}[j+1, k], M_{i-1}[j, k] + v_i)$, we need to update $M[j+1, k]$ before $M[j, k]$.
   Hence, the nested loop should start from $j = m$, and end at $j = 1$, with stepping to be -1.
   Inside this loop,
   1. we first check $M[j+1, k] = \max(M[j+1, k], M[j, k] + v_i)$.
   2. Then, we need to update the maximum value that can be collected on at most $f_i + 1, ..., h$ floor. If the i-th round does not change $M[j+1, k]$ for some $k \in [f_i + 1, h]$, we can then stop checking because the invariant ensures that $M_i[j+1, k+1] \geq M_i[j+1, k]$.
      We do not need to care about $M[j+1][1], \cdots, M[j+1][f_i - 1]$ because we cannot go back to the lower floors if we collect the treasure on f_i-th floor.
5. After we iterate over all the rounds, we can then return $M[m+1][h]$.

Proof of Correctness:

The invariants always hold:
For all $i \in [1, n], j \in [0, m], k \in [1, h-1]$,

1. $M_i[j+1, k] \geq M_{i-1}[j+1, k]$
2. $M_i[j+1, k] \geq M_i[j, k]$
3. $M_i[j+1, k+1] \geq M_i[j+1, k]$

The base case, which is $M_0[j+1][k]$ for all $j \in [0, m], k \in [1, h-1]$ is 0 since all entries of the 2D-array are initialized to zero. The three invariants definitely hold.
Assume we are checking the i-th round.
Line 5 of the pseudocode will update $M[j+1][f_i]$.
If $M_{i-1}[j+1][f_i] < M_{i-1}[j][f_i]$, then $M_i[j+1][f_i] = M[j+1][f_i] \leftarrow M_{i-1}[j][f_i] + v_i$.
So after line 5 we have $M_i[j+1, k] \geq M_{i-1}[j+1, k]$ and $M_i[j+1, k] \geq M_{i-1}[j, k]$.
Since $M_i[j+1, k] = \max(M_{i-1}[j+1, k], M_{i-1}[j, k] + v_i)$ and $M_i[j, k] = \max(M_{i-1}[j, k], M_{i-1}[j-1, k] + v_i)$ or 0 if $j = 0$, we also have $M_i[j+1, k] \geq M_i[j, k]$.
Hence, the invariant $(1), (2)$ hold.

From line 6 to 8, we are updating $M[j+1][l]$ for all $l \in [\min(f_i + 1, h), h]$ such that $M_i[j+1][h] \geq M_i[j+1][h-1] \geq \cdots \geq M_i[j+1][f_i+1] \geq M_i[j+1][f_i]$.
Thus the invariant (3) holds.

Therefore, at the end of each rounds, the three invariants always hold.
Thus, according to the definition of $M[j+1][k]$, we have that $M[m+1][h] = M_n[m+1][h]$ contains the correct result, which is the maximum total value of treasure of at most m collections, on or beneath h-th floor.

Pseudocode:

$\textbf{Algorithm} \quad \textbf{collectTreasure}(n, m, h, A)$

> $A = [(v_1, f_1), (v_2, f_2), \ldots, (v_n, f_n)]$

1. Optional: $A \leftarrow \texttt{re-assign}(A)\ \texttt{if}\ \ h > 1.5n$
2. Initialize the $(m+1) \times h$ 2D-array M and set all of its entries to be zero
3. $\textbf{for } i \textbf{ from } 1 \textbf{ to } n \textbf{ do}$
4. $\quad \textbf{for } j \textbf{ from } m \textbf{ to } 1 \textbf{ do}$ // stepping $= -1$
5. $\quad\quad M[j+1][f_i] \leftarrow \max(M[j+1][f_i], M[j][f_i] + v_i)$
6. $\quad\quad \textbf{for } l \textbf{ from } f_i + 1 \textbf{ to } h \textbf{ do}$
7. $\quad\quad\quad M[j+1][l] \leftarrow \max(M[j+1][l], M[j+1][l-1])$
8. $\quad\quad\quad$ // if $M[j+1][l]$ is not changed, we can break this loop
9. $\textbf{return } M[m+1][h]$

Algorithm re-assign(A)

> $A = [(v_1, f_1), (v_2, f_2), ..., (v_n, f_n)]$

1. $A' \leftarrow [(f_1, 1), (f_2, 2), ..., (f_n, n)]$
2. $A'' \leftarrow$ `MergeSort`(A') `//` stable sort based on the value of f
3. `//` $A'' = [(f_1', i_1), (f_2', i_2), ..., (f_n', i_n)]$ where $f_1' \leq f_2' \leq ... \leq f_n'$
4. **for** j **from** 1 **to** n **do**
5. $\quad A[i_j] \leftarrow (v_{i_j}, j)$
6. `return` A

Run-time Analysis:

Without calling `re-assign`, the runtime is $O(nmh)$ because

1. The initialization of the 2D-array takes $O(mh)$ time.
2. The innermost for-loop takes $O(h)$ time.
3. The second innermost for-loop iterates m time.
4. The outermost for-loop iterates n time.

Hence, the overall run-time is $O(nmh + mh) = O(nmh)$. The space complexity is $O(mh)$.

If `re-assign` is called, the runtime is $O(n^2m)$ because h is bound to be n, creating A' takes $O(n)$ time, sorting A' using MergeSort takes $O(n \log n)$ time, modifying A takes $O(n)$ time. The rest is the same.
Thus, the total run-time is $O(n \log n + n + n + n^2m) = O(n^2m)$. The space complexity is $O(mh + n)$.