CSCI 570 : Homework 7

1 Graded Problems

(1). You are given an integer array $a[1], \ldots, a[n]$, find the contiguous subarray (containing at least one number) which has the largest sum and **only return its sum**. The optimal subarray is **not** required to return or compute.
Taking $a = [5, 4, -1, 7, 8]$ as an example: the subarray $[5]$ is considered as a valid subarray with sum 5, though it only has one single element; the subarray $[5, 4, -1, 7, 8]$ achieves the largest sum 23; on the other hand, $[5, 4, 7, 8]$ is not a valid subarray as the numbers 4 and 7 are not contiguous.
Solution: Let $f[1], \ldots, f[n]$ be the array where $f[i]$ is the largest sum of all contiguous sub-arrays ending at index i. Considering the recurrence relation of $f[i]$: if the subarray $[a[i]]$ achieves the largest sum, then we have $f[i] = a[i]$; otherwise, the optimal subarray ending at index i should consist of the optimal subarray achieving $f[i-1]$ at index $i-1$, therefore, $f[i] = a[i] + f[i-1]$ in this case due to the contiguous requirement. Combining these two cases together, we have the following equality for all $i \geq 2$:

$$f[i] = \max\{f[i-1], 0\} + a[i],$$

with the boundary condition $f[1] = a[1]$. Finally, the largest sum achievable is exactly $max(f)$, which can be achieved by traversing all n entries of array f. The time complexity of this proposed algorithm is $\mathcal{O}(n)$ as there are n subproblems and each subproblem costs $O(1)$ to compute. The pseudo code is presented in Algorithm 1.

Algorithm 1

Input: an integer array a and the length of this array n.
Initialize: set $f[1] = a[1]$ and $Ans = f[1]$.
for $i = 2, \ldots, n$ **do**
  Compute $f[i]$ as

$$f[i] = \max\{f[i-1], 0\} + a[i]$$

  if $f[i] > Ans$ **then** ▷ update answer

$$Ans = f[i]$$

Return: the largest sum stored in Ans.

(2). Solve Kleinberg and Tardos, Chapter 6, Exercise 10.
Solution:
(a) Consider the following example: there are a total 4 minutes, the numbers of steps that can be done respectively on the two machines in the 4

minutes are listed as follows (in time order): Machine A: 2, 1, 1, 200; Machine B: 1, 1, 20, 100. The given algorithm will choose A, then move, then stay on B for the final two steps. The optimal solution will stay on A for the four steps.

(b) An observation is that, in the optimal solution for the time interval from minute 1 to minute i, you should not move in minute i, because otherwise, you can keep staying on the current machine and get a better solution ($a_i > 0$ and $b_i > 0$). For the time interval from minute 1 to minute i, if you are on machine A in minute i, you either stay on machine A in minute $i-1$ or are in the process of moving from machine B to A in minute $i-1$. Now let $OPT_A(i)$ represent the maximum value of a plan in minute 1 through i that ends on machine A, and define $OPT_B(i)$ analogously for B. If the first case is the best action to make for minute $i-1$, we have $OPT_A(i) = a_i + OPT_A(i-1)$; otherwise, we have $OPT_A(i) = a_i + OPT_B(i-2)$. In sum, we have

$$OPT_A(i) = a_i + \max\{OPT_A(i-1), OPT_B(i-2)\},$$

and similarly

$$OPT_B(i) = b_i + \max\{OPT_A(i-2), OPT_B(i-1)\}.$$

It takes $O(1)$ time to complete the operations in each iteration; there are $O(n)$ iterations; the tracing backs takes $O(n)$ time. Thus, the overall complexity is $O(n)$.

Algorithm 2

Input: the number of minutes n, the number of steps can be complete in each minute $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$.

Initialize: set $OPT_A(0) = OPT_B(0) = 0$,$OPT_A(1) = a_1$ and $OPT_B(1) = b_1$ as boundary conditions; set $D_A(1) =$ STAY and $D_B(1) =$ STAY to record actions.

for $i = 2, \ldots, n$ **do**

Compute $OPT_A(i)$ and $OPT_B(i)$ as

$$OPT_A(i) = \max\{OPT_A(i-1), OPT_B(i-2)\} + a_i,$$
$$OPT_B(i) = \max\{OPT_B(i-1), OPT_A(i-2)\} + b_i.$$

Record the actions (either STAY or MOVE) that achieve $OPT_A(i)$ and $OPT_B(i)$ in $D_A(i)$ and $D_B(i)$.

Initialize the optimal solution (the sequence of actions) as $Sol = \{\}$, and set

$$\textit{curTime} = n, \textit{curMachine} = \underset{\textit{machine} \in \{A,B\}}{\operatorname{argmax}} \{OPT_{\textit{machine}}(n)\}$$

for back-tracing.

while *curTime* > 0 **do**

if $D_{\textit{curMachine}}(\textit{curTime}) =$ STAY **then**

Update *curTime* $\leftarrow$ *curTime* $- 1$.

Add STAY at the front of Sol.

else

Switch *curMachine* and update *curTime* $\leftarrow$ *curTime* $- 2$.

Add MOVE at the front of Sol.

Return: the largest number of steps $\max\{OPT_A(n), OPT_B(n)\}$, and the optimal action sequence Sol.

---

(3). You are given an array of positive numbers $a[1], \ldots, a[n]$.
For a sub-sequence $a[i_1], a[i_2], \ldots, a[i_t]$ of array a (that is, $i_1 < i_2 <, \ldots, < i_t$): if it is an increasing sequence of numbers, that is, $a[i_1] < a[i_2] < \ldots < a[i_t]$, its happiness score is given by

$$\sum_{k=1}^{t} k \times a[i_k].$$

Otherwise, the happiness score of this array is zero.
For example, for the input $a = [22, 44, 33, 66, 55]$, the increasing subsequence $[22, 44, 55]$ has happiness score $(1) \times (22) + (2) \times (44) + (3) \times (55) = 275$; the increasing subsequence $[22, 33, 55]$ has happiness score $(1) \times (22) + (2) \times (33) + (3) \times (55) = 253$; the subsequence $[33, 66, 55]$ has happiness score 0 as this sequence is not increasing.
Please design an efficient algorithm to **only** return the highest happiness score over all the subsequences. The optimal subsequence is **not** required to return or compute.
Solution: Let $L(i, k)$ be the cumulative value of the happiest subsequence that ends at the first i items and using the i-th element for a subsequence of length k. Therefore, we have the following equality

$$L(i, k) = \max_{j<i, a[j]<a[i]} L(j, k-1) + k \times a[i],$$

as the recurrence equation, which is similar to that of the longest increasing subsequence. Obviously, the happiest subsequence ending at index i of length k must be consisted of some subsequence ending at index j of length $k-1$ for some $j < i$ that $a[j] < a[i]$. Therefore, the subsequence eding at index j of length $k-1$ must be the optimal subsequence for $(j, k-1)$ which achieves the happiness score $f(j, k-1)$. Finally, $f(i, k)$ can be computed via taking the maximum over all valid $(j, k-1)$ and adding $k * a[i]$. Clearly, the algorithm runs in $O(n^3)$ since there are at most $O(n^2)$ entries and each entry take $O(n)$ time to compute.

Algorithm 3

Input: an integer array a and the length of this array n.
Initialize: set $L(i, j) = -\infty$ for all (i, j) that $i, j \in 1, \ldots n$, and $Ans = -\infty$.
for $i = 1, \ldots, n$ **do**
  Set $L(i, 1) = a[i]$ and update $Ans = \max\{Ans, L(i, 1)\}$.
  for $k = 2, \ldots, i$ **do**
    for $j = 1, \ldots, i-1$ **do**
      if $a[j] < a[i]$ *and* $k - 1 \leq j$ **then** ▷ update $L(i, k)$
        $L(i, k) \leftarrow \max\{L(i, k), L(j, k-1) + k \times a[i]\}$
    if $L(i, k) > Ans$ **then** ▷ update answer
      $Ans = L(i, k)$
Return: the highest happiness score stored in Ans.

(4). You are given an $m \times n$ binary matrix $g \in \{0, 1\}^{m \times n}$. Each cell either contains a "0" or a "1". Give an efficient algorithm that takes the binay matrix g, and returns the largest side length of a square that only contains

1's. You are **not** required to give the optimal solution.

Solution: We define the $m \times n$ dynamic programming matrix f with $f(i,j)$ being the side length of the maximum square matrix of 1's whose right bottom cell is (i,j). That is, the sub matrix whose left top cell is $(i - f(i,j) + 1, j - f(i,j) + 1)$is the largest square matrix that is filled with 1's and ending at cell (i,j), and either $f(i,j) = \min\{i,j\}$ or the matrix $(i - f(i,j), j - f(i,j))$ to (i,j) contains some 0's.

Starting from cell $(1,1)$, we compute the dynamic programming matrix f at cell (i,j) as

$$f(i,j) = \begin{cases} \min\{f(i-1,j), f(i-1,j-1), f(i,j-1)\} + 1, \\ \qquad\qquad i \geq 2 \ \ \& \ \ j \geq 2 \ \ \& \ \ g(i,j) = 1, \\ g(i,j), \quad \text{Otherwise.} \end{cases}$$

The recurrence equation can be verified directly. Suppose the largest 1's square ended at (i,j) has the length of k, then it is ensured that $\min\{f(i-1,j), f(i-1,j-1), f(i,j-1)\} \geq k-1$ as these three sub matrices of size $k-1$ (that is, the matrices ended at $(i-1,j)$, $(i,j-1)$ and $(i-1,j-1)$) are all filled with 1's.

On the other hand, suppose that $\min\{f(i-1,j), f(i-1,j-1), f(i,j-1)\} = k$ and $g(i,j) = 1$ hold, then we have $f(i,j) \geq k+1$ as the square matrix of size $k+1$ ending at (i,j) only contains 1's. Combining these two inequalities ensure the correctness of the aforementioned recurrence equation.

We also remember the largest side length of a square found so far. In this way, we traverse the dynamic programming matrix once and find out the required maximum size. This gives the side length of the square (say $maxsqlen$).

Algorithm 4

Input: a binary matrix g, the number of rows m and the number of columns n.
Initialize: set $f(i,j) = 0$ for all cells (i,j) that $i \in 1,\ldots,m, j \in 1,\ldots,n$. Set $Ans = 0$ for answer.
for $i = 1,\ldots,m$ **do**
  for $j = 1,\ldots,n$ **do**
    if $g(i,j) = 0$ *or* $i = 1$ *or* $j = 1$ **then** ▷ boundary condition
      $f(i,j) = g(i,j)$
    else
      $f(i,j) = \min\{f(i-1,j), f(i,j-1), f(i-1,j-1)\} + 1$
    if $f(i,j) > Ans$ **then** ▷ update answer
      $Ans = f(i,j)$
Return: the largest side length stored in Ans.

---

2 Practice Problems

(1). Given n sand piles of weights $w[1],\ldots,w[n]$ on a line, we want to merge all the sand piles together. At each step we can only merge adjacent sand piles, with cost equal to the weight of the merged sand pile. In particular, merging sand piles i and $i+1$ has cost $w_i + w_{i+1}$. Furthermore, one gets a single sand pile of weight $w_i + w_{i+1}$ in its place which is now adjacent to sand piles $i-1$ and $i+2$. Note that different merging orders will result in

different final costs, please find the minimum cost to merge all the sand piles. The optimal merging order is **not** required.

- Sample Input: $n = 4, w[1, \ldots, n] = [100, 1, 1, 100]$.
- Sample Output: the minimum cost is $1+1+2+100+102+100 = 306$ by first merging $(1, 1)$, then $(100, 2)$ and $(102, 100)$.

Solution: We define a subproblem $f(i, j)$ to be the minimum possible cost of merging all the piles from i to j. Then, we have

$$f(i, j) = \min_{i \le k < j} \{f(i, k) + f(k + 1, j)\} + \sum_{k=i}^{j} w_k$$

for all $i < j$. For any solution that merges the sand piles from i to j must have a final merge which merges piles from i to k and from $k + 1$ to j. Then those sub-intervals of sand piles should be merged at minimum cost which is defined recursively. The recurrence equation chooses the minimum cost solution of this form.
Since there are at most n^2 subproblems, the space complexity is $O(n^2)$. The runtime is $O(n^3)$ since the computation of subproblem takes the minimum over $O(n)$ terms with at most $O(n^2)$ subproblems.

Algorithm 5

Input: the number of sand piles n, and the weights $w[1], \ldots, w[n]$.
Initialize: set all entries of f as

$$f(i, j) = \begin{cases} +\infty, & 1 \le i < j \le n \\ w[i], & i = j \end{cases}.$$

for $s = 2, \ldots, n$ **do** ▷ Iterate the number of sand piles
  for $i = 1, \ldots, n - s + 1$ **do** ▷ Iterate all pairs (i, j) that $j - i + 1 = s$
    Compute $j = i + s - 1$ and $w(i, j) = \sum_{k=i}^{j} w_k$.
    for $k = i, \ldots, j - 1$ **do**
      Update $f(i, j)$ as

$$f(i, j) \leftarrow \min \{f(i, k) + f(k + 1, j) + w(i, j), f(i, j)\}$$

Return: $f(1, n)$ as the minimum cost.
