Problem 1 (20 Points)

Usually, we want to improve the time complexity. But now, let us talk about space complexity.

(a) Recall the knapsack DP algorithm in the lecture, which has nW subproblems totally. We need to maintain a subproblem table with size $n \times W$. Can we use only $O(W)$ space (e.g., maintain only a $1 \times W$ size subproblem table) to implement the DP algorithm (still runs in $O(nW)$)?

(b) Recall the Edit Distance DP algorithm in the lecture, which has nm subproblems totally. Can we use only $O(\min\{n, m\})$ space to implement the DP algorithm (still runs in $O(nm)$)?

Solution. (a) Recall the DP transition function

$$f[i, w] = \max\left\{f[i-1, w], f\left[i, w - c_i\right] + v_i\right\}.$$

To determine $f[i, a]$, we only need to know $f[i-1, \cdot]$ and $f[i, b]$ for $b < a$. Therefore, maintaining a $2 \times W = O(W)$ table is enough.

(b) Without loss of generality, assuming $m < n$. Recall the DP transition function

$$ED[i, j] = \min\left\{ED[i-1, j-1] + \mathbb{1}\{x_i \neq y_j\}, ED[i, j-1] + 1, ED[i-1, j] + 1\right\}.$$

To determine $ED[i, a]$, we only need to know $ED[i-1, \cdot]$ and $ED[i, b]$ for $b < a$. Therefore, maintaining a $2 \times m = O(\min\{n, m\})$ table is enough.

This solution is provided by ***T.A. Kangrui Mao****.* $\square$

Problem 2 (25 points)

Given a sequence of integers $a_1, a_2, ..., a_n$, a lower bound and an upper bound $1 \leq L \leq R \leq n$. An (L, R)-step subsequence is a subsequence $a_{i_1}, a_{i_2}, ..., a_{i_\ell}$, such that $\forall 1 \leq j \leq \ell - 1$, $L \leq i_{j+1} - i_j \leq R$. The revenue of the subsequence is $\sum_{j=1}^{\ell} a_{i_j}$. Design a DP algorithm to output the maximum revenue we can get from a (L, R)-step subsequence.

(a) Suppose $L = R = 1$. Design a DP algorithm in $O(n)$ to find the maximum $(1, 1)$-step subsequence.

(b) Design a DP algorithm in $O(n^2)$ to find the maximum (L, R)-step subsequence for any L and R.

(c) Design a DP algorithm in $O(n)$ to find the maximum (L, R)-step subsequence for any L and R.

Solution. In fact we can design an algorithm in $O(n)$ to find the maximum (L, R)-step subsequence for any L and R to solve this problem in once, but for completeness of this reference we do it one by one.

(a) When $L = R = 1$ the sequence is successive. Let $f(k) = max a_k, f(k-1) + a_k$ denote the maximum revenue of a subsequence ending with a_k. Then the answer is $max_{1 \leq k \leq n} f(k)$. *Correctness*: we will prove it by induction.

- For $k = 1$, $f(a) = a_1$;
- Suppose that for any $k = m$ we have the maximum revenue $f(k)$;
- For $k = m + 1$, $f(k) = a_k$ or $f(k) = f(k-1) + a_k$. Thus all the subrevenue can be calculated and $f(k)$ given by the state transition equation is the maximum revenue. Therefore, this algorithms is always correct.

Time complexity: Since $f(k)$ only depends on $f(k-1)$, the process is linear on the sequence, so the complexity is $O(n)$.

(b) The same with $2(a)$, assuming Let $f(k) = max a_k, max_{k-R \leq p \leq k-L} f(p) + a_k$ denote the maximum revenue of a subsequence ending with a_k, where $p > 0$.

Correctness: we will prove it by induction.

- For $k = 1$, $f(a) = a_1$;
- Suppose that for any $k = m$ we have the maximum revenue $f(k)$;
- For $k = m+1$, $f(k) = a_k$ or $f(k) = max_{k-R \leq p \leq k-L} f(p) + a_k$. Thus all the subrevenue can be calculated and $f(k)$ given by the state transition equation is the maximum revenue. Therefore, this algorithms is always correct.

Time complexity: For each k wen need $O(R - L)$ time for inner loop, and each loop will run n times. So the complexity is $O((R - L) \cdot n) = O(n^2)$.

(c) By applying PLL(potential Largest List) to the solution in $2(b)$, we obtain an improved algorithm in $O(n)$.

Algorithm 1 Maximum (L,R)-step Subsequence

Input: Integer squence $a_1, a_2, ..., a_n$, lowwer bound L and upper bound R

Output: Maximum revenue from a (L,R)-step Subsequence

function UPDATE(k)
  while $Q.front().index < k - R$ **do**
    $Q.pop_front()$
  while $Q.back().value \leq f(i - L)$ **do**
    $Q.pop_back()$
  $Q.push_back(f(i - L))$ **return** $Q.front()$
  Construct a double-ended queue Q
  for $k \leftarrow 1$ to n **do**
    $f(k) = max a_k, UPDATE(k)$
  return $max_{1 \leq k \leq n} f(k)$

When updating, if we record the previous position p_k for $f(k)$, we can trace back to find out the exact sbsequence required.

Correctness: As the first element in PLL is the maximum value we wanted, accordingly, the output of this algorithm is always correct.

Time complexity: As each a_i will be pushed and popped only once, so the updating process runs in $O(1)$. As we update n times, total complexity is $O(n)$.

This solution is provided by ***T.A. Panfeng Liu*** *and* ***Stu. Xiangyuan Xue***. □

Problem 3 (25 points)

Optimal Indexing for A Dictionary

Consider a dictionary with n different words $a_1, a_2, ..., a_n$ sorted by the alphabetical order. We have already known the number of search times of each word a_i, which is represented by w_i. Suppose that the dictionary stores all words in a binary search tree T, i.e., each node's word is alphabetically larger than the words stored in its left subtree and smaller than the words stored in its right subtree. Then, to look up a word in the dictionary, we have to do $\ell_i(T)$ comparisons on the binary search tree, where $\ell_i(T)$ is exactly the level of the node that stores a_i (root has level 1). We evaluate the search tree by the total number of comparisons for searching the n words, i.e., $\sum_{i=1}^{n} w_i \ell_i(T)$. Design a DP algorithm to find the best binary search tree for the n words to minimize the total number of comparisons.

Solution. Use $dp[i][j]$ to represent the comparison times of the optimal binary search tree from a_i to a_j.

$$dp[i][j] = \begin{cases} 0 & i > j \\ w_i & i = j \\ min_{i \le r \le j}\, dp[i][r-1] + dp[r+1][j] + \sum_{k=i}^{j} w_k & i < j \end{cases}$$

Correctness We use induction to prove $dp[i][j]$ is the cost of the optimal binary tree. Assume that the claim is true for every $[i][j]$ where $j - i \le n - 1$. We prove it is also true for $j - i = n$. Consider the optimal binary tree for $[i][j]$. Assuming its root is r, the critical point is that its left part is an optimal binary tree of $[i][r-1]$ and its right part is an optimal binary tree of $[r+1][j]$. Then, by enumerating all choices of root r, we can get the optimal binary tree of $[i][j]$.

Complexity $O(n^3)$ for there are $O(n^2)$ states and each update operation costs $O(n)$. □

This solution is provided by ***T.A. Xiaolin Bu***.

Problem 4 (30 points)

Collecting Gift On a Grid

Given n gifts located on a $(m \times m)$ grid. The i-th gift is located at some point (x_i, y_i) (integers chosen in $1 \cdots m$) on the grid, with value $v_i \geq 0$. A player at $(1, 1)$ is going to collect gifts by several *Upper-Right Move*. In particular, assuming the player is currently located at (x, y), he can make one *Upper-Right Move* to another point (x', y') where $x' \geq x$ and $y' \geq y$. The cost of this movement is $(x' - x)^2 + (y' - y)^2$. The player will collect the i-th gift when he is at point (x_i, y_i). There is no restriction for the number of *Upper-Right Move* and the final location of the player.

(a) Design an $O(m^2)$ algorithm to maximize the player's profit, i.e., the sum of value he collects minus the sum of cost he pays for his *Upper-Right Move*.

(b) Sometimes, n can be much smaller than m. Can you design another algorithm that runs in $O(n^2)$ for this situation?

(c) (0 Points. It is for fun, you can discuss your idea with me.) Is there any difference if the player can only make *Upper-Right Move* among gifts? Can we still design efficient algorithm runs in $O(n^2)$, $O(nm)$, and $O(m^2)$?

Solution. Proof for problem 4 here.

(a)

First, consider the minimal cost of all routes to move from (x, y) to (x', y') $(x \leq x', y \leq y')$. Easy to know that the minimal cost is (x'-x)+(y'-y) by moving to the adjacent grid up/right each step.

algorithm: assume f(i, j) to be the maximal profit the player could get when end at grid (i, j).

$$f(i,j) = \begin{cases} v(1,1) & i = 1, j = 1 \\ f(i-1,j) + v(i,j) - 1 & i > 1, j = 1 \\ f(i,j-1) + v(i,j) - 1 & i = 1, j > 1 \\ max\{f(i-1,j), f(i,j-1)\} + v(i,j) - 1 & i > 1, j > 1 \end{cases}$$

The maximal profit = $max_{1 \leq i \leq m, 1 \leq j \leq m} f(i,j)$

correctness: We use induction to prove that f(i, j) is the maximal profit for each grid. For i = j = 1, the player is at the starting point and it cost nothing. For i,j >1, since the way to have minimal cost for any route on the map is to move to adjacent up/right grid each step, by considering the down/left grids' profit, we can calculate f(i, j) by induction.

time complexity: The time cost for each step is O(1), and there are m^2 grids on the map and consequently m^2 steps, so the total time complexity is $O(m^2)$. The time cost for choosing the maximal profit of all grids is $O(mlogm)$, which is smaller than $O(m^2)$ and is omitted.

(b) Since the player gets value only when it enters a grid with a gift, but pays cost for every move, it is obvious that the maximal-profit ending grid must be a grid with a gift. Considering the optimal moving strategy we discussed in (a), we can only discuss the move in the rows and columns that contains gifts in them to optimize our algorithm.

algorithm: Pick all the columns and rows of grids with gifts and form a new map. The values for the gifts remain the same, while the cost for moving to adjacent up/right grids is converted according to their original locations. Assume g(i, j) to be the maximal profit the player could get when ending at grid (i, j) of the new map.

$$g(i,j) = \begin{cases} v(1,1) & i=1, j=1 \\ g(i-1,j)+v(i,j)-cost & i>1, j=1 \\ g(i,j-1)+v(i,j)-cost & i=1, j>1 \\ max\{g(i-1,j), g(i,j-1)\}+v(i,j)-cost & i>1, j>1 \end{cases}$$

The maximal profit = $max_{1\leq i, 1\leq j} g(i,j)$

correctness: We use induction to prove that g(i, j) is the maximal profit for each grid. For i = j = 1, the player is at the starting point and it cost nothing. For i,j >1, since the way to have minimal cost for any route on the map is to move to adjacent up/right grid each step, by considering the down/left grids' profit, we can calculate g(i, j) by induction.

time complexity: The time cost for each step is O(1). Since there are n gifts, the size of the new map is no lager than n^2 and consequently we have no larger than n^2 steps, so the total time complexity is $O(n^2)$. The time cost for choosing the maximal profit of all grids is $O(nlogn)$, which is smaller than $O(n^2)$ and is omitted.

□