

CS 344: Design and Analysis of Computer Algorithms

Instructor: Kangning Wang

Homework 3: Due on **March 14, 2025**

Acknowledgment: Our team members are Alexander Lin (al1655), Pranav Tikkawar (pt422), and Ivan Zheng (iz72) .

Problem 1

We are given an $n \times n$ matrix, where each entry a_{ij} is a number. We start from the top left corner, and in each step, we move one square either downward or to the right. We end when we reach the bottom right corner. We aim to maximize the sum of the numbers that we have visited.

1. Give an $O(n^2)$ -time DP algorithm for the problem.
2. Now suppose that we do this process twice, but each number is only counted once even if we have visited it twice. Give an $O(n^3)$ -time DP algorithm for the new problem.

Solution.

1. We start by initializing an array $dp[i][j]$. The base case is $dp[0][0] = a[0][0]$. We then iterate through the matrix. For each cell (i, j) , the maximum sum can be obtained by choosing the better of the two possible moves: from the cell above $\rightarrow dp[i-1][j]$ and from the cell to the left $\rightarrow dp[i][j-1]$. Therefore, the recurrence is $dp[i][j] = a[i][j] + \max(dp[i-1][j], dp[i][j-1])$. The maximum sum is at $dp[n-1][n-1]$. The time complexity is $O(n^2)$ to store the dp array.
2. Let $dp[i_1][j_1][i_2][j_2]$ represent the maximum sum achievable when the first path is at (i_1, j_1) and the second path is at (i_2, j_2) . However, this would lead to $O(n^4)$ complexity, which is not efficient. Let k be the total number of steps taken so far. Since both paths start at $(0, 0)$ and end at $(n-1, n-1)$, the total number of steps for each path is $2(n-1)$. At step k , the positions of the two paths can be represented as (i_1, j_1) and (i_2, j_2) , where $i_1 + j_1 = k$ and $i_2 + j_2 = k$. This reduces the state space to $dp[k][i_1][i_2]$, where k ranges from 0 to $2(n-1)$, and i_1, i_2 range from 0 to $n-1$. For each step k , and for each possible position (i_1, j_1) and (i_2, j_2) of the two paths: if the two paths are at the same cell, the value is counted only once. Otherwise, the values at both cells are added. The recurrence is $dp[k][i_1][i_2] = \text{value at } (i_1, j_1) + \text{value at } (i_2, j_2) + \max(\text{possible moves})$. The possible moves are: both paths move right, both paths move down, or one path moves right, and the other moves down. The algorithm fills a table of size $O(n^3)$, so the time complexity is $O(n^3)$.

Problem 2

We are given two number sequences (a_1, a_2, \dots, a_n) and (b_1, b_2, \dots, b_n) . Among their common subsequences (not necessarily consecutive), find one with the maximum sum. Give an $O(n^2)$ -time DP algorithm for this problem.

Solution.

To solve this problem, we can use dynamic programming. We will create a 2D array dp where $dp[i][j]$ represents the maximum sum of common subsequences between the first i elements of sequence a and the first j elements of sequence b . We can initialize the array with all zeros and to fill it up, then iterate through each element of the two sequences and update the dp array based on the following conditions:

- If $a_i = b_j$, then we can add the value of a_i (or b_j) to the maximum sum of common subsequences of the previous elements, i.e., $dp[i][j] = dp[i-1][j-1] + a[i]$.
- If $a_i \neq b_j$, then we take the maximum of the previous sums, i.e., $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$.

The final answer will be stored in $dp[n][n]$. The time complexity of this algorithm is $O(n^2)$ because we are iterating through each element of the two sequences and updating the dp array in constant time.

Problem 3

We have n identical magic eggs. We can drop a magic egg from a building with m floors. There is an unknown threshold $t \in \{1, 2, \dots, m\}$, so that dropping a magic egg from floor $t-1$ has no impact on the egg, but dropping a magic egg from floor t breaks it and makes it disappear.

We want to know the answer to the following question. What is the minimum number $f(n, m)$ so that we can always deduce the threshold t with at most $f(n, m)$ egg drops? For example, if $n = 1$, then the optimal strategy is to drop the egg from floor 1, then floor 2, and so on, until the egg survives floor $m-1$ or breaks. Therefore, $f(1, m) = m-1$.

Give a DP algorithm for this problem, and write down its running time.

Solution.

We first need to build the function $f(a, b)$ which gives us the minimum number of drops needed to determine the threshold t given a eggs and b floors.

Consider that we drop the egg from some floor x , $1 < x < b$. We then have two cases:

1. The egg breaks, in which case we have one less egg to work with but know that the threshold t is below x

2. The egg survives, in which case we have the same number of eggs and know that the threshold t is above x

Case 1 tells us that we must then search for $f(a - 1, x - 1)$, as we know floor x is above the threshold. Case 2 tells us that we must search for $f(a, b - x)$ since floor x is below the threshold, so there are $b - x$ floors remaining above floor x to test.

Since we must always be able to deduce the threshold t in $f(a, b)$ drops, we need to account for whichever case between 1 and 2 that actually gives us the greatest number of remaining tests we must conduct, so we want:

$$\max\{f(a - 1, x - 1), f(a, b - x)\}$$

We want to choose x to *minimize* the value of $f(a, b)$, so therefore we have:

$$f(a, b) = 1 + \min_{x \in [1, b]} \{\max\{f(a - 1, x - 1), f(a, b - x)\}\}$$

Here, we need to add 1 no matter what since when we drop the egg at floor x , we add one drop.

Now that we have a formula for $f(a, b)$, we must consider how to implement it into an algorithm. Considering that we likely need to store $f(a, b)$ for each value of a and b possible, we initialize a $n + 1 \times m + 1$ matrix, adding 1 to each dimension to account for having 0 eggs and/or 0 floors. We note that $f(1, b) = b - 1$ as given in the problem, base cases $f(a, 0) = 0$ and $f(a, 1) = 1$ for when we have 0 floors and 1 floor left, and edge case $f(0, b) = 0$ for when we have 0 eggs left. This is chosen deliberately to be 0 since we don't want to end up with 0 eggs left when there are multiple floors left, since that means we are not able to determine t . Setting $f(0, b) = 0$ ensures that the $\max\{\dots\}$ portion of the formula will always avoid said scenario unless there are also 0 floors left, in which case we have 0 tests left to run anyway.

We can now construct the full matrix describing the problem by looping over every set of indices ij of the matrix, then looping over all possible values of x for each ij (ie since $j = 1$ corresponds to $f(a, 0)$, $j = \beta$ corresponds to $f(a, \beta + 1)$, we need to loop x through the values 1 through $j + 1$), calculating the optimal x , and storing the value of $f(a, b)$. Our final answer the minimum number of drops needed will be stored in $f(n, m)$, which is the $(n+1)(m+1)$ matrix element.

Looping over all elements of the matrix is $O(nm)$, and since x can range up to m , calculating the optimal x is worst-case $O(m)$, giving us an overall worst-case runtime of $O(nm^2)$

Problem 4

We need to build a binary search tree with n nodes whose keys are $1, 2, \dots, n$ in this order. Let p_i be the frequency with which node i will be queried. Each time node i is queried, we need to spend time equal to one plus the depth of node i . Give an $O(n^3)$ -time (or better) DP algorithm to find an optimal binary search tree that minimizes the total time spent on all queries.

Solution.

To solve this we will create a 2d DP array which represents the minimum cost of an optimal BST containing keys from i to j . We will denote this array as $dp[i][j]$.

For the case when $i = j$, the cost is just the frequency of that node, p_i .

The case for $i \neq j$ is a bit more complicated. We will need to consider all possible roots k from i to j . For each root, we will calculate the cost of the left and right subtrees and add the cost of the root itself. The cost of the left subtree is given by $dp[i][k - 1]$ and the cost of the right subtree is given by $dp[k + 1][j]$. The cost of the root is given by the sum of all frequencies from i to j .

Therefore, we can write the recurrence relation as follows:

$$dp[i][j] = \min_{i \leq k \leq j} \{ dp[i][k - 1] + dp[k + 1][j] + \sum_{m=i}^j p_m \}$$

Problem 5

We covered the 0/1 knapsack and unbounded knapsack problems in class. The bounded knapsack problem is another variant. In this problem, there are n types of items, and type i has c_i identical copies. An item of type i has an integer weight $w_i \geq 1$ and a value v_i . Given a weight limit W , the goal is to find the maximum sum of values using a subset of items whose sum of weights is at most W .

Give an $O(nW^2)$ -time DP algorithm for the bounded knapsack problem. As a voluntary challenge, improve the running time to $O(nW \log W)$ or $O(nW)$. (Hint: One can use the monotonic queue to achieve an $O(nW)$ running time. Alternatively, for an $O(nW \log W)$ running time, consider the binary representation of each c_i .)

Solution.

The basic 0/1 knapsack relies on dynamic programming with the function $f(k, W)$, where k denotes the sum of values using a subset $\{a_1, a_2, \dots, a_k\}$ of the n items and W denotes the weight limit of those items. Specifically, the formula is the following:

$$f(k, W) = \max\{f(k - 1, W), f(k, W - w_k) + v_k\}$$

The idea here is that when we encounter a new item a_k , we can either choose to not add it, which gives us the first element of the max function, or add it, which gives us the second element of the max function, depending on whichever is most optimal.

When we expanded to an unbounded knapsack, we used essentially the same formula, but instead of a_k denoting an item, it denotes an item type. When implementing the unbounded knapsack, we must then vary the number of items of type a_k we add from 1 to the maximum allowed by

the weight limit W .

Our solution essentially follows the unbounded knapsack problem, except we now limit the number of copies of item type k we can add to the knapsack based on the number of copies c_k that exist. We initialize an array of size $W + 1$ with the first element being 0 since a weight limit of 0 corresponds to no items and thus no value. We then loop through the item types from 1 to n . Inside this loop, we loop through all possible weights W , and for each, we loop a variable p from 1 to $\min\{c_i, \lfloor \frac{W}{w_i} \rfloor\}$ and use the following function:

$$f(i, W) = \max\{f(i - 1, W), f(i, W - p * w_i) + p * v_i\}$$

This will loop through all possible number of item i we can add and tell us what our best case value is. Our final answer will be stored in $f(n, W)$.

This algorithm needs to loop through all n items, and for each item, it needs to go through the entire array of size on order of W in the . It also may need to revisit each weight W again in the process of looping through all the possible number of copies of an item we add. This means our final runtime is $O(nW^2)$.