CS 3510: Design & Analysis of Algorithms October 10th 2024

Problem Set 5: Dynamic Programming

Jinseo Lee Due: October 18th 2024

- Please type your solutions using LaTeX or any other software. Handwritten solutions will not be accepted.
- Your algorithms must be in plain English & mathematical expressions, and the pseudo-code is optional. Pseudo-code, without sufficient explanation, will receive no credit.
- If we ask for a specific running time, a correct solution achieving it will receive full credit even if a faster solution exists.
- Unless a question explicitly states that no work is required to be shown, you must provide an explanation or justification for your answer

Problem 1: Max Op

(30) Points

Suppose we are given a sequence of n positive integers interspersed with $n-1$ arithmetic operators, each of which is either addition or multiplication. In other words, we are given

$$y = x_1 \,\mathrm{op}_1\, x_2 \,\mathrm{op}_2 \cdots \mathrm{op}_{n-1}\, x_n$$

where each x_i is a positive integer and op_i is either $+$ or $*$. We want to fully parenthesize this expression so that the resulting arithmetic value y is as large as possible.

For example, if the input is $2 + 4 * 1 + 5$, then the desired answer is $(2 + 4) * (1 + 5) = 36$. In contrast, the parenthesization $((2 + (4 * 1)) + 5)$ only evaluates to 11.

Design a dynamic programming algorithm to return the maximum possible value as described above.

1. Give the dimensions of your DP table and what each cell in the table stores/represents. No justification needed.

   Solution:

   The DP table has n x n dimensions, where n is the number of numerical values in the given input. DP[i][j] represents the maximum output of the sub-expression from the i^{th} variable to the j^{th} variable, where $1 \le i \le j \le n$.

2. Give the base case(s). Justify your answer.

   Solution:

   Since DP[1][1], DP[2][2], ... DP[n][n] contains only one number without any operator, the maximum value is the number itself.

   Therefore, DP[i][j], where i = j, equals i^{th} variable of the input.

3. Give the recurrence. Justify why this recurrence correctly fills in your DP table.

   Solution:

   DP[i][j] = max(DP[i][k] op_k DP[k+1][j]) for all $i \le k \le j - 1$, where op_k is the operator between k^{th} variable and $k + 1^{th}$ variable.

   In order to retrieve the maximum value of the sub-expression ranging from i to j index, it is necessary to split the expression into two subpart with the split point 'k' and combine the result. More specifically, op_k splits the expression and considers the maximum value of max(sub-expression from i^{th} to k^{th} variable) $+/*$ max(sub-expression from $k + 1^{th}$ to j^{th} variable), for all possible k values within the range. The left part could be retrieved by looking at DP[i][k], and the right part could be done with DP[k+1][j]. However, only the upper triangular part of the dp table (where $i \le j$) needs to be filled; The cells below the main diagonal represent cases where $i > j$, which do not correspond to any valid sub-expression.

4. How do we recover the maximum possible value that can be created? Justify your answer.

Solution:

Retrieving the value at DP[1][n] will return the maximum value that can be created with the whole expression.

Since the recurrence above takes the expression ranging from i^{th} variable to j^{th} variable into an account, the boundary variables must include both the first variable and the last variable for it to consider the whole expression. This makes i value of 1, the first element, and j value of n, the last element.

5. Give the runtime. Justify your answer.

Solution:

The runtime is $O(n^3)$.

It takes O(n) time to fill in each cell of the dp table as there are at most (n-1) k values (O(n-1) = O(n)). Also, it takes $O(n^2)$ times to fill in the dp table as it has a size of n times n dimension. (Not necessarily n^2 times but is upper-bounded as it does not fill in the cells below the main diagonal) Therefore, the overall runtime is $O(n^3)$ $(O(n) \cdot O(n^2) = O(n^3))$

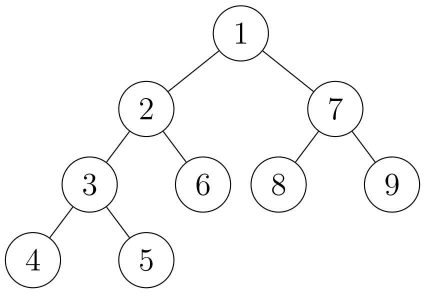
Problem 2: National Park Trees

(30 points)

The National Park spans over $|V|$ forest sections, connected by trails that resemble a tree $T = (V, E)$. Park rangers want to ensure that the entire park is monitored to prevent forest fires. Watchtowers can be installed at any section $S \in V$, and when done so, all trails/edges connected to S are under surveillance.

Your task is to develop a strategy using Dynamic Programming to find the minimum number of watchtowers needed to monitor the entire park. The park is modeled as an unweighted undirected tree structure with n vertices and m edges, where each node represents a forest section and the edges represent the trails. The running time of your algorithm should be $\mathcal{O}(|V|)$.

Example Input:



Example Output: 3. The buildings that need to be watchtowers are {2, 3, 7}.

Design an efficient algorithm to count the minimum number of watchtowers needed.

1. Give the dimensions of your DP table and what each cell in the table stores/represents. No justification needed.

    Solution:

    The DP table has the size of 2 x V, where V is the number of vertices. The horizontal variables of the DP table represent the corresponding vertices (if park = 1, then 1). The first row, DP[0][Vertex], represents the minimum number of watchtowers required if the watchtower is not installed at the current park whereas the second row, DP[1][vertex], represents those if the watchtower is installed at the current park.

2. Give the base case(s). Justify your answer.

    Solution:

    DP[0][leaf] = 0, DP[1][leaf] = 1

    In the leaf node, the minimum number of watchtowers required depends solely on the current park as there are no children parks to consider. Therefore, the minimum number of watchtower equals 0 if it is not installed at the current park. Similarly, it is

1 if the watchtower is installed.

3. Give the recurrence. Justify why this recurrence correctly fills in your DP table.

Solution:

$\text{DP}[0][\text{vertex}] = \sum_{\textit{child}} \text{DP}[1][\text{child}]$

: If watchtower is not installed at the current park, all of its children must have the watchtower installed. Therefore, DP[0][vertex] equals the total sum of its children with the watchtower installed (second row).

$\text{DP}[1][\text{vertex}] = 1 + \sum_{\textit{child}} \min(\text{DP}[0][\text{child}], \text{DP}[1][\text{child}])$

: If watchtower is installed at the current park, both the current park and all of its children are covered. Therefore, for each child, we take the minimum between the cases of the child having a watchtower or not as we are aiming for the minimum number of the watchtower possible. Finally, we add 1 as watchtower is installed at the current park.

The DP table is filled from the leaves to the root. This ensures that each node's value is optimally built from its children, leading to the minimum number of watchtowers overall.

4. How do we recover the minimum number of watchtowers needed to monitor the entire park? Justify your answer.

Solution:

The minimum number of watchtowers needed to monitor the entire park could be retrieved by the following: min(DP[0][root], DP[1][root])

The expression above evaluates the two possible scenarios for the root node, which represents the entire park. DP[0][root] gives the minimum number of watchtowers needed if no watchtower is installed at the root, meaning all its children must cover it. DP[1][root] gives the minimum if a watchtower is installed at the root, covering both the root and its children. By taking the minimum of these two values, we ensure that the overall solution is optimal, as it accounts for both covering strategies and selects the one requiring fewer watchtowers.

5. Give the runtime. Justify your answer.

Solution:

The runtime is O($|V|$).

In order to fill the DP table from the leave nodes to the root nodes, we need to run DFS and sort the vertices on the increasing post order, which as a whole takes O($|V|$) time. Filling in each cell in the DP table takes constant O(1) time, and filling in the DP table size of 2V takes another O($|V|$) (O(2V)=O(V)). Therefore, the overall runtime is: O($|V|$) + (O(1) $\cdot$ O($|V|$)) = O($|V|$)

Problem 3: Will's Thanksgiving

(20 points)

Will's trying to host Thanksgiving dinner for the family, but he's never hosted it before. He plans to cook all the food, but with his busy lifestyle, he can't cook all the Thanksgiving dinner food. Then, he has the family vote on what they would most like to eat, and he estimates how long it would take to cook the food. He's fine with only pleasing some portion of the family.

In other words, given a list of N dishes, the total votes v_i for each dish, and how long each dish takes to prepare t_i, figure out a set of dishes that can be prepared in the least amount of time and still meet or exceed the vote threshold V. Note that only one dish can be made at a time and each dish can only be made once.

The matrix $M(i, j)$ represents the minimum amount of time it takes to get j votes from the first i dishes.

No work is required. For the correct option(s), replace the '\item' with '\item[\checkmark]' below.

1. Give the base case(s).

   ☐ $M(0, i) = 0,$ for $i \leq N$

   ✓ $M(i, 0) = 0,$ for $i \leq N$

   ☐ $M(i, j) = -\infty$ for $i \leq N$, j < 0

   ✓ $M(i, j) = \infty$ for $i \leq N$, j < 0

2. Give the recurrence relation(s).

   ✓ $M(i, j) = \min(M(i-1, j), M(i-1, j-v_i) + t_i)$

   ☐ $M(i, j) = \max(M(i-1, j), M(i, j-v_i) + t_i)$

   ☐ $M(i, j) = \min(M(i-1, j), M(i-1, j-v_i) - t_i)$

   ☐ $M(i, j) = \min(M(i-1, j), t_i - M(i-1, j-v_i))$

3. Give the runtime.

   ☐ $\mathcal{O}(N)$

   ✓ $\mathcal{O}(NV)$

   ☐ $\mathcal{O}(V)$

   ☐ $\mathcal{O}(V^2)$

Problem 4: Another Knapsack

(20 points)

The knapsack problem, defined in the course textbook in Section 6.4, involves maximizing the value of items in a knapsack with a capacity. The textbook describes an $\mathcal{O}(nW)$ time algorithm for the knapsack problem without repetition, where we have n items (each of value v_i and weight w_i) and knapsack capacity of W. Now we design an $\mathcal{O}(n^2 \cdot v_{\max})$ time algorithm, where $v_{\max}$ denotes the largest value v_i and n again denotes the number of items.

To solve this problem, we create a 2-dimensional array M with dimensions n by $n \cdot v_{\max}$. This means it has in total $n^2 \cdot v_{\max}$ entries. Each entry $M(i, v)$ in the matrix M represents the minimum possible weight contained in the knapsack to achieve a total value of exactly v when selecting a subset of items from the set $\{1, 2, \ldots, i\}$. If it's impossible to achieve a value of v by selecting a subset of items from $\{1, 2, \ldots, i\}$, we set $M(i, v)$ to infinity. Given this structure, answer the following questions about the associated DP algorithm.

No work is required. For the correct option(s), replace the '\item' with '\item[\checkmark]' below.

1. Give the base case(s).

   - ☐ $M(1, v) = w_i$ if $v < v_i$ else 0
   - ✓ $M(1, 0) = 0$
   - ✓ $M(1, v) = w_1$ if $v = v_1, \infty$ otherwise.
   - ☐ $M(1, v) = w_1 + v_1$ if $v = v_1, \infty$ otherwise.

2. Give the recurrence relation(s).

   - ☐ $M(i, v) = M(i-1, v) + \min(w_i, M(i-1, v-v_i))$
   - ☐ $M(i, v) = w_i + \min\{M(i-1, v), M(i-1, v-v_i)\}$
   - ☐ $M(i, v) = \min(M(i-1, v), w_i \times M(i-1, v-v_i))$
   - ✓ $M(i, v) = \min(M(i-1, v), w_i + M(i-1, v-v_i))$