Algorithms and Data Structures: Homework #11

Due on April 27, 2020 at 23:00

Henri Sota

Problem 11.1

Consider the array $A = (a_1, a_2, \ldots, a_n)$. We call a subarray a succession of numbers in A where the position of any value in the subarray in the array is always bigger than the value of the previous one. Use dynamic programming to find a subarray of maximal ordered length of the array A. You can assume there are no duplicate values in the array. Your algorithm should find one optimal solution if multiple exist.

In order to construct the subarray with the most elements up to a certain position, the dynamic programming approach can build upon previous subproblems of the smaller positions. Building each solution to the subproblems can start from the first element, whose only subarray has solution only itself, and consider adding the elements if they are compatible and if the size of the subproblem up to that position is bigger than the solution built upon.

```
bool Longest(const std::vector<int> &first, const std::vector<int> &second) {
   return first.size() < second.size();
}

std::vector<int> LOS(std::vector<int> sequence) {
    int size = (int) sequence.size();

    // Dynamic Programming Table to store solutions to previous subproblems
    std::vector<std::vector<int>> dp(size);

    // Store the first subarray which is only the first element
    dp[0].push_back(sequence[0]);

    // Build upon the first subarray to find solutions to other subproblems
    for (int i = 1; i < size; ++i) {
        // Build subarray out of solved subproblems for the current i
        for (int j = 0; j < i; ++j) {
            // Check if number at position j can be added and the size of
            // the subproblem solved for subarray up to position j is bigger
            // than the one we have already built upon now
            if ((sequence[i] > sequence[j]) && (dp[i].size() < dp[j].size()))
                // Set the solution to the subproblem of the subarray up to
                // i to the subproblem of the subarray up to j when the size
                // of subproblem j is bigger than subproblem i we already know
                dp[i] = dp[j];
        }
        // Subproblem solved for subarray up to position i will involve the
        // element at position i
        dp[i].push_back(sequence[i]);
    }

    // Find the biggest subarray in dp using comparator based on vector size
    std::vector<std::vector<int>>::iterator los = std::max_element(
        dp.begin(), dp.end(), Longest
    );

    return *los;
}
```

Listing 1: Dynamic Programming Solution for Longest Ordered Subarray Implemented in C++

* In order for the solution to work with the input, there is a need for a negative value to act as a terminator to the sequence as we haven't been given the size of the sequence before. Solution based on implementation here.

Problem 11.2

Consider a triangle formed from n lines ($1 < n \leq 100$), each line containing natural numbers in the interval $[0, 10000]$.

a) Use dynamic programming to determine the biggest sum of numbers existent on the path from the number in the first line and a number from the last line and print the respective path to the output. Each number in this path is seated to the left or to the right of the other value above it.

In order to solve this problem using dynamic programming, the optimal substructure property and overlapping problems should be present. We can deduce that the path with maximum sum includes the first element of the input as every path starts from the top. This means that the path with maximum sum subtracting the first element is one of the paths starting from the second or the third element in the input, known as the left and right choice of the first elements. Expanding this argument further until the last row of the triangle, we have enumerated all possible subproblems. During this enumeration, many of the subproblems coincide to be the same and therefore are calculated over and over again. With the dynamic programming approach, by building bottom-up from the smallest subproblems to the biggest one, the following implementation is given:

```
enum Choice {Left, Right};

std::vector<int> reconstructPath(std::vector<std::vector<int>> &input,
                                 std::vector<std::vector<Choice>> &paths) {
    int pathLocation = 0;
    std::vector<int> path;

    // Traverse each level from the top to reconstruct path from the Choices
    for (int i = 0; i < (int) input.size(); ++i) {
        path.push_back(input[i][pathLocation]);
        // Move on the column to the right if Choice is Right for the next read
        if (paths[i][pathLocation] == Right)
            pathLocation++;
    }

    return path;
}

int SIT(std::vector<std::vector<int>> dp,
        std::vector<std::vector<Choice>> &paths) {
    // Solve subproblems starting from the second to last row and build upon
    // these subproblems while going towards the top
    for (int i = (int) dp.size() - 2; i > -1; --i) {
        // Calculate the maximum path of element j in row i based on the maximum
        // value of the paths of the elements on row i+1, and columns j and j+1
        for (int j = 0; j <= i; ++j) {
            // Check if left path has strictly bigger sum than the one on the
            // right and set the Choice for element in paths depending on it
            if (dp[i + 1][j] > dp[i + 1][j + 1])
                dp[i][j] = dp[i][j] + dp[i + 1][j];
            else dp[i][j] = dp[i][j] + dp[i + 1][j + 1];
            paths[i][j] = dp[i + 1][j] > dp[i + 1][j + 1] ? Left : Right;
        }
    }

    return dp[0][0];
}
```

Listing 2: Dynamic Programming Solution for Sum In Triangles Implemented in C++

* Solution based on implementation here.

b) Analyze the runtime of your solution and compare it to the brute force approach.

The dynamic programming approach calculates all of the subproblems by traversing each node from the second to last row to the top of the triangle. Given a triangle with n rows, it will traverse and calculate the maximum of the left and right choice subpaths for $\frac{n(n-1)}{2}$ elements. The path reconstruction procedure will only take n steps, but it uses an additional $O(n)$, bringing the space complexity to $O(2n)$ with the copy of the input, which could be avoided by just backtracking the solution from the top of the dp matrix. The time complexity is $T(n) = O(n^2) + O(n) = O(n^2)$.

The brute force approach would calculate the sum of all the possible paths and find the path with the biggest sum. Given a triangle with n rows, the number of choice combinations, possible paths, would be 2^{n-1}, which grows large really fast. The brute force approach has time complexity $O(2^n)$.

Therefore, the dynamic programming approach is far more superior than the brute force approach, by reducing the time complexity from exponential order to polynomial of second order.

c) Explain why a greedy algorithm does not work for this problem.

A greedy algorithm would make the most optimal choice locally that maximizes the sum. Therefore, for each decision, it will make the most optimal one in order to hopefully obtain the global optimal solution. Given the input example of the input, the greedy choice solution would be the path $(7, 8, 1, 7, 5)$ with sum 28. This is not the global optimal solution, which is the path $(7, 3, 8, 7, 5)$ with sum 30.

Greedy Choice Solution

7

3 **8**

8 **1** 0

2 **7** 4 4

4 **5** 2 6 5

Global Optimal Solution

7

3 8

8 1 0

2 **7** 4 4

4 **5** 2 6 5

The reason why the greedy choice solution fails at achieving the global optimal solution is due to narrowing down the choice of the best solution by removing local suboptimal choices going further down the triangle.