Algorithms and Data Structures

Spring 2019

Assignment 10

Date: May 02, 2019

Taiyr Begeyev

Problem 10.1 Longest Ordered Subarray

Consider the array $A = (a_1, a_2, ..., a_n)$. 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. Example For $A = (8, 3, 6, 50, 10, 8, 100, 30, 60, 40, 80)$ the solution is : $(3, 6, 10, 30, 60, 80)$

```
/*
__________           .__
\__    ___/ _____    |__|  ___.__. _______
  |    |    \__  \   |  | <   |  | \_  __ \
  |    |     / __ \_ |  |  \___  |  |  | \/
  |____|    (____  / |__|  / ____|  |__|
                 \/        \/
*/
#include <iostream>
#include <vector>
using namespace std;

void print(vector<int> arr) {
    for (int i: arr) {
        cout << i << " ";
    }
    cout << endl;
}

/*
    @brief Find and print Longest Ordered Array

    Create a vector container, which consists of vector containers.
    arr[i] is a vector itself that stores some subsequence for each
    element of the array a. Subsequence of each element is composed of
    Longest Ordered Array of this specific element. That's where Dynamic
    Programming Concept comes in handy. Go from left to right towards the
    current element and try to find LIS based on previous elements' LIS
*/

void findLIS(vector<int> a) {
    int n = a.size();
    // create 2d array
    vector<vector<int>> arr(n);

    // first element of the array a is already LIS of itself
    arr[0].push_back(a[0]);
    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if ( (a[i] > a[j]) && (arr[j].size() + 1 > arr[i].size()) ) {
                // find the max(arr[j])
                arr[i] = arr[j];
            }
            // push the current element
        }
        arr[i].push_back(a[i]);
    }

    // find the vector container with the longest length
    vector<int> max = arr[0];
    for (vector<int> i: arr) {
        if (i.size() > max.size()) {
            max = i;
        }
    }
    print(max);
}

int main() {
    int n;
    cout << "Enter the size of the array: ";
    cin >> n;

    vector<int> arr(n);

    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    findLIS(arr);

    return 0;
}
```

Problem 10.2 Sum in triangles

(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.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

/*
    @brief finds and prints the max sum from top to bottom
    of the Triangle

    At each level we have to choose whether to pick the number from
    left or right on our path. In order to make the optimal choice, which maximizes
    the sum, we need to know how large the sum is if we go either way.
    We start from the row before the last row. We use the following formula: a + max(b, c),
    where a is the current element and b and c are "adjacent element" (left and right element
    on the next row). We keep using this formula until we reach the top of the triangle
    (the first row), which will be equal to the max sum
    We dynamically allo
*/

void sumInTriangle(int** arr, int n) {
    // dynamically allocate memory
    int** arr2 = new int*[n];
    for (int i = 0; i < n; i++) {
        arr2[i] = new int[n];
    }

    // copy the arr to it
    for(int i = 0; i < n; i++) {
        for (int j = 0; j <= i; j++) {
            arr2[i][j] = arr[i][j];
        }
    }

    // use the formula
    for (int i = n - 2; i >= 0; i--) {
        for (int j = 0; j <= i; j++) {
            arr[i][j] += max(arr[i + 1][j], arr[i + 1][j + 1]);
        }
    }

    cout << "Sum = " << arr[0][0] << endl;

    // determine the path. That's where our second array comes in handy.
    // Since arr doesn't contain values of the triangle anymore, but rather sums
    int max_pos = 0;
    for (int i = 0; i < n; i++) {
        if (arr[i][max_pos] < arr[i][max_pos + 1]) {
            max_pos++;
        }

        cout << arr2[i][max_pos] << " ";
    }
    cout << endl;

    // deallocate memory
    for (int i = 0; i < n; i++) {
        delete[] arr2[i];
    }
    delete[] arr2;
}

int main() {
    while(true) {
        try {
            int n;
            cout << "Enter the number of lines: ";
            cin >> n;
            if ( (n <= 1) || (n > 100) ) {
                throw "Wrong range of n";
            }

            // dynamically allocate memory
            int **arr = new int*[n];
            for (int i = 0; i < n; i++) {
                arr[i] = new int[n];
            }

            // input
            for(int i = 0; i < n; i++) {
                for (int j = 0; j <= i; j++) {
                    cin >> arr[i][j];
                    if (arr[i][j] < 0 || arr[i][j] > 10000) {
                        throw "Wrong value of the triangle element";
                    }
                }
            }

            sumInTriangle(arr, n);

            // deallocate memory
            for (int i = 0; i < n; i++) {
                delete[] arr[i];
            }
            delete[] arr;
            break;
        }
        catch(const char* msg) {
            cerr << "Exception: " << msg << endl;
        }
    }
    return 0;
}
```

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

In light of the fact that the total number of elements in the triangle is the following $g(n) = \frac{n(n+1)}{2}$, where n is the number of rows, it yields that the time complexity of the dynamic programming approach equals $T(n) = O(g(n)) = O(\frac{n(n+1)}{2}) = O(n^2)$. Since we iterate through our "matrix", it requires quadratic time to make the job done. In the contrast, the brute force approach has a running time of 2^k, where $k = \frac{n(n-1)}{2}$. Due to the fact that that every number has two possible paths, apart from the last row.

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

The problem with the greedy approach is that it has a local choice of the sub-problems, while dynamic programming would solve the all sub-problems and then select one that would lead to a globally optimal solution. If we apply the greedy approach, we end with the completely different answer, which is not the largest. So we try to pick as max number as possible. It follows that the path is $7 \to 8 \to 1 \to 7 \to 5$, which gives 28 in the sum. That's proves that the greedy approach is not always the best.

(c) A scuba diver uses a special equipment for diving. He has a cylinder with two containers: one with oxygen and the other with nitrogen. Depending on the time he wants to stay under water and the depth of diving the scuba diver needs various amount of oxygen and nitrogen. The scuba diver has at his disposal a certain number of cylinders. Each cylinder can be described by its weight and the volume of gas it contains. In order to complete his task the scuba diver needs specific amounts of oxygen and nitrogen. Theoretically, the diver can take as many cylinders as he wants/needs. Use dynamic programming to find the minimal total weight of cylinders he has to take to complete the task and which those cylinders are. In case of several acceptable solutions, printing just one of them is enough.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    // number of testcases
    int c;
    cin >> c;

    for (int i = 0; i < c; i++) {
        while(true) {
            // execute unless exception is thrown
            try {
                // volume of oxygen and nitrogen needed to compete task
                int t, a;
                cin >> t >> a;
                if (t < 1 || t > 21)
                    throw "t is out of range";
                if (a < 1 || t > 79)
                    throw "a is out of range";

                // number of cylinders
                int n;
                cin >> n;
                if (n < 1 || n > 1000)
                    throw "n is out of range";

                int oxygen[1000], nitrogen[1000], weight[1000];
                for (int i = 0; i < n; i++) {
                    // enter data for each cylinder
                    cin >> oxygen[i] >> nitrogen[i] >> weight[i];
                }


                // dynamic programming
                int dp[1000][22][80];
                for (int i = 0; i < n; i++) {
                    for (int j = 0; j <= t; j++) {
                        for (int k = 0; k <= a; k++) {
                            dp[i][j][k] = __INT_MAX__;
                            if (j == 0 && k == 0) {
                                dp[i][j][k] = 0;
                            }
                        }
                    }
                }

                for (int j = 0; j <= t; j++) {
                    for (int k = 0; k <= a; k++) {
                        if (j <= oxygen[0] && k <= nitrogen[0] && (j > 0 || k > 0)) {
                            dp[0][j][k] = weight[0];
                        }
                    }
                }

                for (int i = 1; i < n; i++) {
                    for (int j = 0; j <= t; j++) {
                        for (int k = 0; k <= a; k++) {
                            dp[i][j][k] = dp[i - 1][j][k];

                            if (j <= oxygen[i] && k <= nitrogen[i])
                                dp[i][j][k] = min(dp[i][j][k], weight[i]);
                            else
                                dp[i][j][k] = min(dp[i][j][k],
                                    weight[i] + dp[i-1][max(0,j-oxygen[i])][max(0,k-nitrogen[i])]
                        }
                    }
                }

                // find the path
                int i = n - 1, x = t, y = a;
                vector<int> path;
                while(i >= 0 && (x > 0 || y > 0)) {
                    if (i == 0) {
                        path.push_back(i + 1);
                        break;
                    }
                    else if (dp[i][x][y] != dp[i-1][x][y]) {
                        x -= oxygen[i];
```

```cpp
                        x = max(x, 0);
                        y -= nitrogen[i];
                        y = max(y, 0);
                        path.push_back(i + 1);
                        i--;
                    }
                    else {
                        i--;
                    }
                }
                reverse(path.begin(), path.end());

                // print min weight
                cout << dp[n - 1][t][a] << endl;

                // print the cylinders needed to complete task
                for (int x: path) {
                    cout << x << " ";
                }
                cout << endl;

                // if no exceptions are thrown, then get out of the infinite loop
                break;
            }
            catch(const char* e) {
                cerr << e << endl;
                cout << "Start Over!" << endl;
            }
        }
    }

    return 0;
}
```