CMS 460
Algorithms
Sprint 5
Dynamic Programming

Kenneth Shortrede, Alex Wehr, Olliver Luo, Sanghun Park

Matrix - Shortest path to end

Introduction

We are given a matrix of size M x N, with each position of the matrix containing a number, and a starting point of (0,0). Then, by moving either one unit down, or one unit right, we need to get to the last position of the matrix (M-1, N-1). For example, from position (i, j) we have a choice to make to go either to (i, j+1) or to (i+1, j). We need to go to the last position by having the lowest possible sum of numbers (every number we cross on our path, adds to our total)

Example

Find the least costly route possible in…

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

Shortest:

$$\begin{bmatrix} 0 & 1 & 2 \\ & & 5 \\ & & 8 \end{bmatrix}$$

Solution

The main idea to solve the problem is to use recursion in order to find the best optimal path. As each step of the way only presents 2 possible solutions (right or down), it is not hard to understand how recursion could prove very helpful in solving this. We break the problem into subproblems continually, until we reach the base case where the solution becomes atomic and gets returned.

Sum to reach a certain position (m, n) = Sum to reach (m,n) + Min (sum to reach (m, n-1), Sum to each (m-1, n))

This is a relatively simple and straightforward approach, but as you may have realized already, it is very resource consuming. It is not using Dynamic Programming. On this simple approach, everytime a minimum value is found, it is not saved in any way, and since that same value is going to be used multiple times, it is calculated over and over. This results in a very costly program, that when the matrix gets large enough, becomes very resource consuming.

For example, we start at (0,0) and we can move to (0, 1) or (1, 0)
(0,1) → Calculate minimum of going to (0, 2) and (1, 1)
(1,0) → Calculate minimum of going to (1, 1) and (2,0)

As we can see here, even in the second step of the movements, we are already repeating the same calculations (highlighted). As we move forward in the matrix, the amount of steps that get repeated will highly increment.

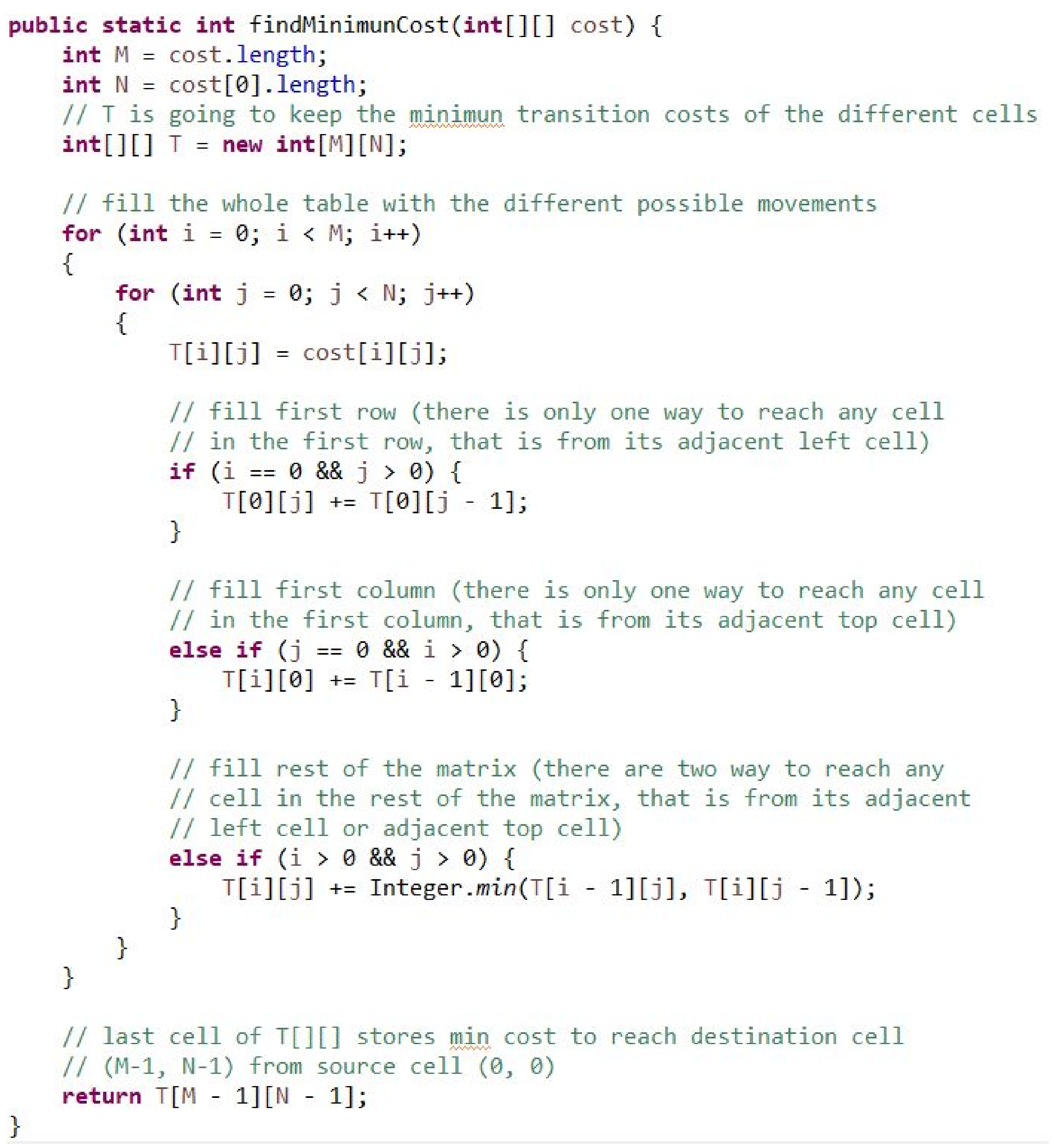
In order to avoid this repetition, we are going to create a table at the beginning, with the total values associated with each step we decide to take. Then, before making any calculations, we are going to make sure if the result has already been found, and if so, we will obtain it from the table.

Steps to Complete the Problem:

1) Fill the 2 base cases first. We can start by filling the first row, in which there is only one way of getting to each position, and then filling the first column, in which (as well) there is only one way of getting to each.
2) After we have the base cases for column 0 and row 0, we can start filling the rest of the table. In each remaining position of the table, we will now have only 2 possible paths (coming from the position up or the position left). Thus, we only need to find the minimum value of those 2 choices, and sum them up to the actual value of the position we are moving towards.

Below we can see the code, where we use 2 for-loops (for rows and columns), we fill the first row and column first, and then we find the minimum of the two possible paths, in each iteration of the loop.

Programming Example Method



```java
public static int findMinimunCost(int[][] cost) {
    int M = cost.length;
    int N = cost[0].length;
    // T is going to keep the minimun transition costs of the different cells
    int[][] T = new int[M][N];

    // fill the whole table with the different possible movements
    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < N; j++)
        {
            T[i][j] = cost[i][j];

            // fill first row (there is only one way to reach any cell
            // in the first row, that is from its adjacent left cell)
            if (i == 0 && j > 0) {
                T[0][j] += T[0][j - 1];
            }

            // fill first column (there is only one way to reach any cell
            // in the first column, that is from its adjacent top cell)
            else if (j == 0 && i > 0) {
                T[i][0] += T[i - 1][0];
            }

            // fill rest of the matrix (there are two way to reach any
            // cell in the rest of the matrix, that is from its adjacent
            // left cell or adjacent top cell)
            else if (i > 0 && j > 0) {
                T[i][j] += Integer.min(T[i - 1][j], T[i][j - 1]);
            }
        }
    }

    // last cell of T[][] stores min cost to reach destination cell
    // (M-1, N-1) from source cell (0, 0)
    return T[M - 1][N - 1];
}
```

Review:

We went over two different implementations of the problem. First, we used a certain type of recursion that allowed us to get the correct result, but at the expense of more resources. The second attempt used a table and a few for loops in order to get to that same results without repeating processes.

- Review Question: Using normal code or pseudocode, can you think of a way that we could use the recursive option but adding a table to not repeat operations?