Lab Exercises: Methodology of Problem Solving

This document defines the **in-class exercise** assignments for the "Algorithms" course @ Software University.

For the following exercises you are given a Visual Studio solution "**Problem-Solving-Lab**" holding portions of the source code. You can download it from the course's page. You can **test your solutions in the Judge system here**.

Part II – Zig-Zag Matrix

You are given a **matrix of positive integer numbers**. A **zig-zag path** in the matrix starts from some cell in the first column, goes to some cell **up** in the second column, then to some cell **down** in the third column, etc. until the last column is reached. Your task is to write a program that finds the zigzag path with the **maximal sum**. Example:

If multiple maximal zig-zag paths exist, print the first one which uses the upper-most cell possible at each column (from left to right).

On the first line of input you'll receive the number of rows N. On the second line you'll receive the number of columns M. On each of the next N rows you'll receive M positive integer numbers separated by a comma (',').

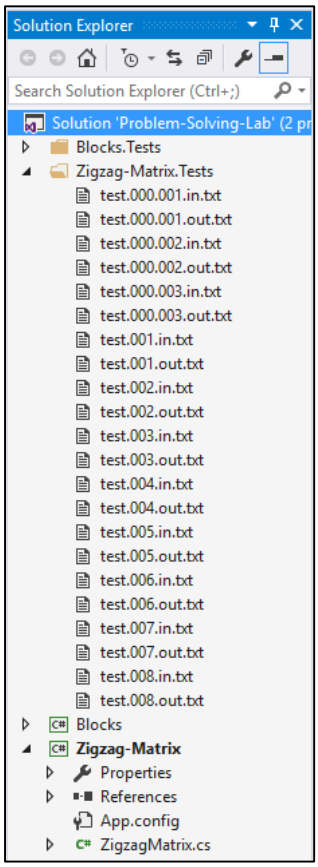
Print the maximal sum along with the path followed in format: **{maxSum} = {cell1} + {cell2} + … + {cellM}**. You can test your solution to the problem here.

Input	Output
4 4 2,4,5,6 9,7,1,5 8,7,7,9 8,2,6,4	30 = 8 + 7 + 6 + 9
17 4 714,52,415,740 102,321,549,697 44,830,171,952 414,58,309,16 956,596,667,526 711,691,776,214 617,919,924,536 102,637,758,360 446,315,243,132 856,313,794,920 732,566,376,314 891,869,999,456	3761 = 891 + 919 + 999 + 952

363,869,471,137 650,108,393,24 277,201,124,184 397,13,596,408 73,811,506,100	
5 10 339,575,789,846,979,801,574,337,95,863 612,383,154,963,796,733,748,281,370,854 675,164,992,998,38,958,856,214,567,348 857,709,774,768,270,798,663,440,506,66 458,172,785,558,953,312,854,131,222,250	7919 = 857 + 575 + 992 + 963 + 953 + 958 + 854 + 337 + 567 + 863

Provided Assets

For this problem you're given a project **Zigzag-Matrix** and a solution folder called **Zigzag-Matrix.Tests** which holds all tests from the Judge system.



Analyze the Problem

We're looking for a path in a matrix. At first, it might occur to you that this can be solved by an algorithm like Dijkstra's. We need, however, just one cell from each column and based on the cells we choose we'll have a different number of available next cells moving forward.

We can try a greedy approach by taking the biggest cell from each column. But the problem restricts the available cells we can take because once we decide to take a cell on a given column, on the next column we should check either above or below it. What if in the first column we have the largest value at cell (0, 0)? We are only allowed to move up afterwards, but we're at the first row, so we'll have nowhere to go. A greedy algorithm will not work.

The best option we have based on what we've learned so far is to use **dynamic programming** – for each cell find the maximum path and then recover the path when we find the global maximum.

Break Down the Problem

An outline of a dynamic programming solution is pretty straightforward:

1) Read the input and fill the matrix we'll be working with
2) Using DP find the maximal path leading to each cell
3) Recover the path after the DP algorithm is finished
4) Print the output

Choose Appropriate Data Structures

At first glance, we'll obviously need the following structures:

- A matrix of integers which we receive as input
- A matrix to hold the max path for each cell (we'll fill this using a DP approach)
- An array to keep track of the path we need to take once we're done (this is trickier though, an array won't do, we'll see why later)
- At some point, we'll need a list for the path when we recover it, but we'll create it when we reach that point

Solve the Problem Step by Step

Step 1. Read the Input

It's up to you whether you prefer using a two-dimensional matrix or a jagged array. Here, we'll use a jagged array to simplify the parsing of the input.

Read the dimensions, create the matrix and then fill it:

```
int numberOfRows = int.Parse(Console.ReadLine());
int numberOfColumns = int.Parse(Console.ReadLine());

int[][] matrix = new int[numberOfRows][];
ReadMatrix(numberOfRows, matrix);
```

We'll separate the parsing of the matrix in a method. Since we have a jagged array (array of arrays), we can simply split each row and convert it to an array of integers using LINQ:

```
private static void ReadMatrix(int numberOfRows, int[][] matrix)
{
    for (int i = 0; i < numberOfRows; i++)
    {
        matrix[i] = Console.ReadLine()
            .Split(',')
            .Select(int.Parse)
            .ToArray();
    }
}
```

Step 2. Dynamic Programming – Setup

It's not that hard to see that keeping the max path for each cell will require another matrix with the same dimensions as the one we get at the input.

It's much harder to see what we'll need to recover the path though. We take one cell from each column, so why not keep the row index for each column in an array?

If you think about it, you'll see this won't work. Dynamic programming works by checking each path to find the optimum. At any given column though, there may be better paths (up to that point) than the global maximum. If we keep row indices in an array, we run the risk of overwriting the values in it based on local maximums. Once we find a better path later on, we'll have no way of recovering the path that led to it, as the array holds a different path, one that was better up to a point, but turned out to be non-optimal.

So, the way to approach this is to **hold the path in yet another matrix** – for each cell we'll keep the row index of the cell which led to it in order to produce the maximal path for that cell.

Having all this in mind, we'll have the following setup:

```
int[,] maxPaths = new int[numberOfRows, numberOfColumns];
int[,] previousRowIndex = new int[numberOfRows, numberOfColumns];
```

Another thing we need to do before we start with the DP algorithm, is to ensure we have a starting point. We'll be traversing the columns first, then the cells at each row, therefore, we need to initialize the first column. As pointed out before, cell (0, 0) is impossible to reach, but all others are and the maximal value for each is just the value of the cell:

```
// Initialize first column
for (int row = 1; row < numberOfRows; row++)
{
    maxPaths[row, 0] = matrix[row][0];
}
```

Step 3. Dynamic Programming – Implementation

Once we have the first column initialized, we can start filling all the others – first the columns and then the rows.

```
// Fill max paths
for (int col = 1; col < numberOfColumns; col++)
{
    for (int row = 0; row < numberOfRows; row++)
    {
        // TODO
    }
}
```

Note that this will take care of the requirement – " If multiple maximal zigzag paths exist, print the first one which uses the upper-most cell possible at each column (from right to left)."

We'll only fill the value in each cell with the best path which uses the upper-most cell from the previous column; this is guaranteed because of the way we traverse the rows – from row 0 to the last.

Next, for each cell we need to find the best path. We do this by **adding the cell's value to the best path from the previous column**. Because of the zigzag requirement, **we need to check if the column is even or odd**. If the column is odd, this means the path needs to come from a cell below the current one; if the column is even we'll check only rows which are above the current cell's row. The algorithm is the same, we just loop different parts of the column.

We can declare the maximal path of the previous column in a separate variable:

```
int previousMax = 0;

// On odd columns we check cells below and one column to the left
if (col % 2 != 0)
{
    for (int i = row + 1; i < numberOfRows; i++)
    {
        if (maxPaths[i, col - 1] > previousMax)
        {
            // TODO: update previousMax
            // TODO: mark the best path to cell in the previousRowIndex matrix
        }
    }
}
else // on even columns we check cells above and one column to the left
{
    for (int i = 0; i <= row - 1; i++)
    {
        if (maxPaths[i, col - 1] > previousMax)
        {
            // TODO: update previousMax
            // TODO: mark the best path to cell in the previousRowIndex matrix
        }
    }
}
```

Note: again, we traverse the rows in increasing order to ensure the final result will contain the max path with upper-most cells.

Once we have the previous maximum, the maximum for the current cell is just the sum of the previous max and its value:

```
maxPaths[row, col] = previousMax + matrix[row][col];
```

Step 4. Recover the Path

So far we didn't keep the global maximum anywhere, so we'll need to manually check what is the row index of the last cell in the path:

```
var currentRowIndex = GetLastRowIndexOfPath(maxPaths, numberOfColumns);
```

We just traverse the last column and get the row index of the max value contained in the **`maxPaths`** matrix:

```
private static int GetLastRowIndexOfPath(int[,] maxPaths, int numberOfColumns)
{
    int currentRowIndex = -1;
    int globalMax = 0;
    for (int row = 0; row < maxPaths.GetLength(0); row++)
    {
        if (maxPaths[row, numberOfColumns - 1] > globalMax)
        {
            globalMax = maxPaths[row, numberOfColumns - 1];
            currentRowIndex = row;
        }
    }

    return currentRowIndex;
}
```

Once we have the row, we can recover the path:

```
var path = RecoverMaxPath(numberOfColumns, matrix, currentRowIndex, previousRowIndex);
Console.WriteLine("{0} = {1}", path.Sum(), string.Join(" + ", path));
```

The **`RecoverMaxPath()`** method will start at the **`currentRowIndex`** and the last column and follow the path kept in **`previousRowIndex`** for that cell. It will put each cell value in a list and reverse it before returning it:

```
private static List<int> RecoverMaxPath(
    int numberOfColumns,
    int[][] matrix,
    int rowIndex,
    int[,] previousRowIndex)
{
    List<int> path = new List<int>();
    int columnIndex = numberOfColumns - 1;

    while (columnIndex >= 0)
    {
        // TODO: add cell to path (found at rowIndex)
        // TODO: update rowIndex using the info in previousRowIndex
        columnIndex--;
    }

    path.Reverse();

    return path;
}
```

Complete the TODOs to recover the maximal path.

Test the Solution

Test your solution in the Judge system here. If your implementation is correct all tests should pass:

07. Zigzag Matrix

Allowed working time: 0.10 sec.
Allowed memory: 16.00 MB
Size limit: 16.00 KB
Checker: Trim

C# code ▾ Submit

	Submissions
Points	Time and memory used
✔✔✔✔✔✔✔✔ 100 / 100	Memory: 9.28 MB Time: 0.020 s