Lab Exercises: Methodology of Problem Solving

This document defines the **in-class exercise** assignments for the ["Algorithms" course @ Software University](https://softuni.bg).

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](#)**.

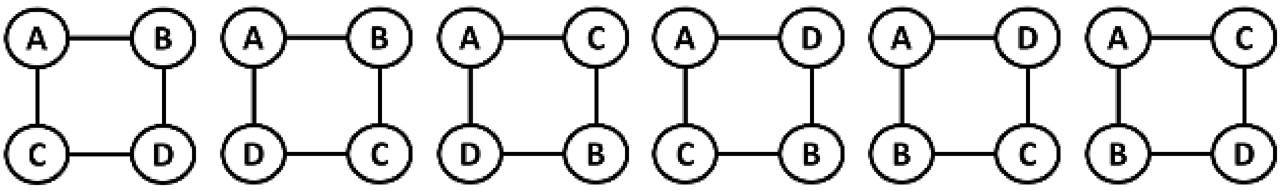
Part I – Blocks

This first problem is **combinatorial**: generating all **2 x 2 blocks** holding **n** letters.

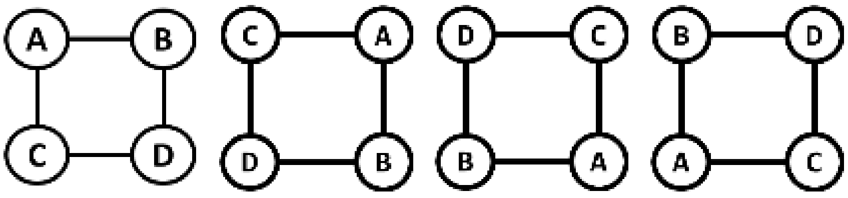
Problem Description

We are given an integer **n** (4 ≤ **n** ≤ 10). Using the **first n capital Latin letters**, generate all distinct **blocks of 2 x 2 letters**. Use each letter inside a block at most once (no repeating letters are allowed in the blocks). We assume that blocks obtained by **rotating** another block are the same and should be skipped.

Example: **n = 4**. The letters used in the blocks are: **A**, **B**, **C**, and **D**. The expected generated blocks are as follows:



Note that the below blocks are the same (after **rotation**):



You can **represent blocks as strings**, e.g. the first block above is **ABCD** (take the corners from top to bottom and from left to right).

Input

The input holds a single integer number **n** (4 ≤ **n** ≤ 10).

Output

At the first line in the output, print the number of unique blocks in format:

```
Number of blocks: {count}
```

At the next lines **print each unique block** on a single line. The ordering of the lines is not strictly defined, but you should first generate all blocks starting with '**A**', then non-duplicate block starting with '**B**', etc.

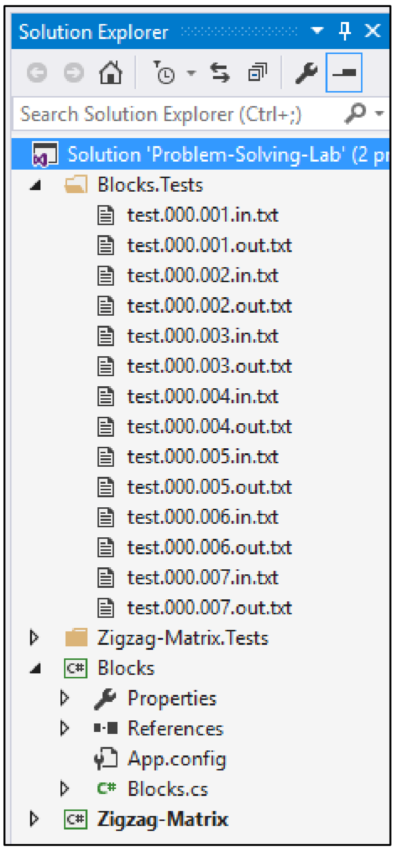
Sample Input and Output

Input	Output
4	`Number of blocks: 6`

	ABCD ABDC ACBD ACDB ADBC ADCB
5	Number of blocks: 30 ABCD ABCE ABDC ABDE ABEC ABED ACBD ACBE ACDB ACDE ACEB ACED ADBC ADBE ADCB ADCE ADEB ADEC AEBC AEBD AECB AECD AEDB AEDC BCDE BCED BDCE BDEC BECD BEDC

Provided Assets

For this problem you are given a Visual Studio project called "**Blocks**" along with all tests from the Judge system in a solution folder called **Blocks.Tests**. You can use the **in.txt** files to test your program and compare the output with the contents of the respective **out.txt** file.


Solution Explorer

Search Solution Explorer (Ctrl+;)

```
Solution 'Problem-Solving-Lab' (2 pr
  Blocks.Tests
    test.000.001.in.txt
    test.000.001.out.txt
    test.000.002.in.txt
    test.000.002.out.txt
    test.000.003.in.txt
    test.000.003.out.txt
    test.000.004.in.txt
    test.000.004.out.txt
    test.000.005.in.txt
    test.000.005.out.txt
    test.000.006.in.txt
    test.000.006.out.txt
    test.000.007.in.txt
    test.000.007.out.txt
  Zigzag-Matrix.Tests
  Blocks
    Properties
    References
    App.config
    Blocks.cs
  Zigzag-Matrix
```

Analyze the Problem

What type of problem are we dealing with? We have some elements and we need to choose some of them and combine them to obtain some unique combination. This is obviously a **combinatorics** problem.

Since the order of elements matters and we always pick 4 elements out of a set with 4+ elements, what we need to generate are **variations of 4 elements out of n elements**. We don't take the same letter more than once, so we **won't be interested in repetitions**.

Knowing the type of problem and what needs to happen, the solution becomes much clearer. We can use the algorithm for generating variations we've used before in the course. Experience with solving such problems is most helpful at this stage.

Break Down the Problem

Basically, the problem boils down to the following:

1) We receive an integer from the console
2) Based on the received number we need to generate the letters we'll use for building the blocks
3) Generate all variations of 4 symbols using an algorithm you already know or one you can quickly find online
4) Save all blocks obtained in a collection
5) Keep track of rotated blocks – blocks obtained from other blocks by rotation
6) Output the results in the required format

All steps are trivial except steps 3 and 5. At step 3 we can use an algorithm we've used before and modify it. At step 5 we just need to save in a collection all blocks obtained after a rotation; once we have a block we need to rotate it three times and save each rotated block in the collection.

Choose Appropriate Data Structures

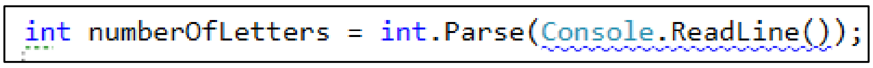
There are several structures we'll need based on what we've outlined so far:

- A structure to hold the set of letters – an array of chars will do nicely
- A structure to hold the currently obtained block – it should be of fixed size (4) and we should be able to modify/swap elements, so, again, an array of chars is appropriate
- A structure to hold the results – since each block will be represented by a string and blocks should be unique, we can use a HashSet<string> for the results
- A structure to hold the rotated blocks – again, blocks will be kept as strings, so a HashSet<string> will do

Solve the Problem Step by Step

Step 1. Get the Input Data

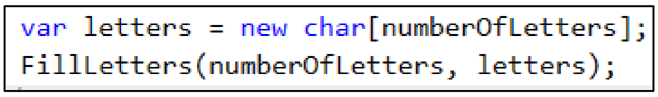
This is easy, read a number from the console:



```
int numberOfLetters = int.Parse(Console.ReadLine());
```

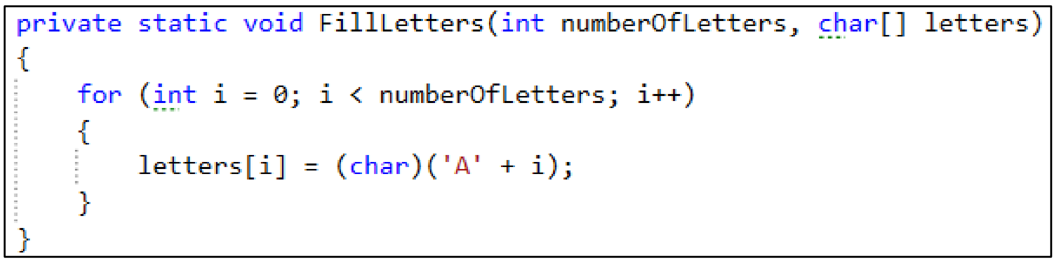
Step 2. Generate the Set of Letters to Use

We have the number **n**; we need to get **n** letters starting from 'A'. We can declare an array of chars to hold the letters and write a method to fill the letters in it:



```
var letters = new char[numberOfLetters];
FillLetters(numberOfLetters, letters);
```

The **`FillLetters()`** method is a simple loop which will traverse the array and place a letter in it. At each step we increment both the index we'll fill and the letter we'll use – the loop's iteration variable can be used to access the array's elements by index and obtain the letter:



```
private static void FillLetters(int numberOfLetters, char[] letters)
{
    for (int i = 0; i < numberOfLetters; i++)
    {
        letters[i] = (char)('A' + i);
    }
}
```

Run the code to make sure we have the correct set of letters.

Step 3. Generate Variations

What we'll have in the end is a series of strings we'll keep in a collection. To make the code testable, it is a good idea to create a public method which accepts the size of the initial set as an argument and returns or fills the results in a HashSet. We can include the letter generation we just wrote in it:

```
public static HashSet<string> FindBlocks(int numberOfLetters)
{
    var letters = new char[numberOfLetters];
    FillLetters(numberOfLetters, letters);

    // TODO

    return results;
}
```

If we decide to write unit tests for our application, this method is easy to use as it receives the number of letters and returns the resulting blocks.

When generating variations, we need to have: the set of letters, a place to hold the current combination, a way to check whether an element is already taken (since we need variations without repetition), the collection to hold the results, the collection to hold the rotated blocks and the index to start at. Quite a lot of things.

We can declare the collection containing the rotated blocks as static; alternatively, we can pass it as a parameter to all methods which need it. Here, we'll declare it as static above the **`Main()`** method.

```
private static readonly HashSet<string> UsedCombinations = new HashSet<string>();
```

We could do the same with the results, but in this case we'll create it inside the **`FindBlocks()`** method and return it.

Let's declare the variables we'll need and pass them to the method which generates the variations:

```
public static HashSet<string> FindBlocks(int numberOfLetters)
{
    var letters = new char[numberOfLetters];
    FillLetters(numberOfLetters, letters);

    bool[] used = new bool[numberOfLetters];
    char[] currentCombination = new char[LettersToChoose];
    HashSet<string> results = new HashSet<string>();

    GenerateVariations(letters, currentCombination, used, results);

    return results;
}
```

The current combination is always of length 4, so we've used a constant.

```
private const int LettersToChoose = 4;
```

The **`GenerateVariations()`** method is a modification of the algorithm to generate variations you probably know:

```
private static void GenerateVariations(
    char[] letters,
    char[] currentCombination,
    bool[] used,
    HashSet<string> results,
    int index = 0)
{
    if (index >= currentCombination.Length)
    {
        // TODO: Add result to resulting set
    }
    else
    {
        for (int i = 0; i < letters.Length; i++)
        {
            if (!used[i])
            {
                // TODO: mark the element as used
                currentCombination[index] = letters[i];
                // TODO: Generate variations from current index onward
                // TODO: unmark the element as used
            }
        }
    }
}
```

You can **complete the TODOs in the else clause**. As for adding the result to the resulting collection – that will require a separate method.

Step 4. Add Unique Blocks to Result

Let's create an **`AddResult()`** method and call it in the **`if`** clause above in place of the TODO.

The method should receive the current combination and the result collection; it can access the rotated blocks since they are kept in a static collection.

First, we'll obviously check if the combination is in the rotated blocks; if not – we add it to the result along with all rotated equivalents to the rotated blocks collection. This isn't too hard; having the array, just think about how the indices change when rotating the block:

```
private static void AddResult(char[] result, HashSet<string> results)
{
    string currentCombination = new string(result);
    if (!UsedCombinations.Contains(currentCombination))
    {
        results.Add(currentCombination);

        UsedCombinations.Add(currentCombination);
        UsedCombinations.Add(new string(new[] { result[3], result[0], result[2], result[1] }));
        UsedCombinations.Add(new string(new[] { result[2], result[3], result[1], result[0] }));
        UsedCombinations.Add(new string(new[] { result[1], result[2], result[0], result[3] }));
    }
}
```

Step 5. Print Result

Now that we have the results, we can print them. Let's use a method for this task as well; the **`Main()`** method will look like this:

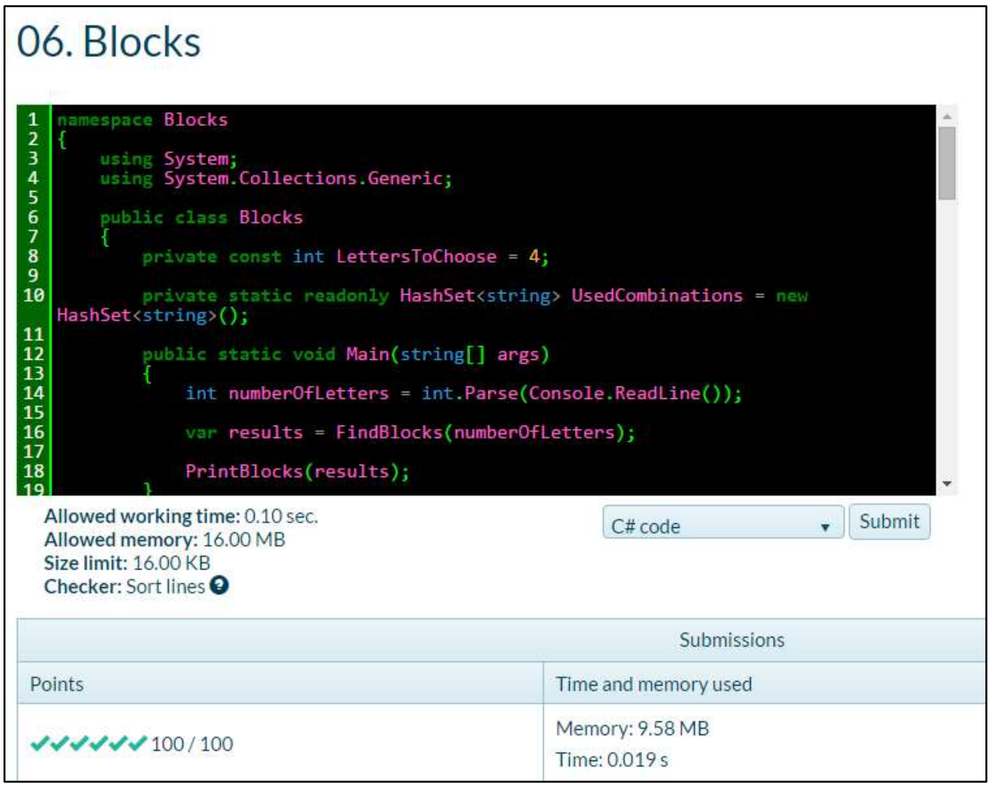
```
public static void Main(string[] args)
{
    int numberOfLetters = int.Parse(Console.ReadLine());

    var results = FindBlocks(numberOfLetters);

    PrintBlocks(results);
}
```

The **`PrintBlocks()`** method is nothing special, it just prints the number of elements in the HashSet and then outputs each string on a separate line:

```
private static void PrintBlocks(HashSet<string> results)
{
    Console.WriteLine("Number of blocks: {0}", results.Count);
    foreach (var combination in results)
    {
        Console.WriteLine(combination);
    }
}
```

Test the Solution

Instead of unit tests, you can submit your code in the Judge system here. You can write your own tests if you'd like, of course. If you did everything correctly, all tests should pass:



06. Blocks

```
namespace Blocks
{
    using System;
    using System.Collections.Generic;

    public class Blocks
    {
        private const int LettersToChoose = 4;

        private static readonly HashSet<string> UsedCombinations = new
HashSet<string>();

        public static void Main(string[] args)
        {
            int numberOfLetters = int.Parse(Console.ReadLine());

            var results = FindBlocks(numberOfLetters);

            PrintBlocks(results);
        }
```

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

Submissions

Points	Time and memory used
✓✓✓✓✓✓ 100 / 100	Memory: 9.58 MB Time: 0.019 s

Simplifying the Solution

We used non-optimal **problem solving strategy** for the above problem. We implemented the **first idea we had** to solve this problem, without thinking about **efficiency**, **simplicity**, etc. Now, we can think about a better solution.

We have two points to optimize in our solution:

- We **hold all generated blocks** and **rotate** each new block to **check for duplicates**. This takes too much memory and also slows down the solution. Can we check for duplicates in better way? Can we **avoid generating duplicated blocks**? Can we generate all blocks without duplicates directly?
- Can we **simplify the algorithm to generate the variations** of 4 elements? We have a fixed number of elements – 4. Maybe we can use **4 nested `for`-loops**?

A Simpler Solution – 4 Nested Loops

There is a much simpler way to arrive at the answers having in mind we always select 4 elements. Generating variations with recursion is easier to implement when we have an unknown number **k**, but when **k** is fixed, **we can solve the problem using k nested loops**. Each loop will take a letter; we then need to check for repetitions and if we have 4 different letters we print them. So, the problem can be solved like this:

```
var lastLetter = 'A' + n - 1;
for (char l1 = 'A'; l1 <= lastLetter; l1++)
{
    for (char l2 = (char)(l1 + 1); l2 <= lastLetter; l2++)
    {
        for (char l3 = (char)(l1 + 1); l3 <= lastLetter; l3++)
        {
            if (l3 != l2)
            {
                for (char l4 = (char)(l1 + 1); l4 <= lastLetter; l4++)
                {
                    if (l4 != l3 && l4 != l2)
                    {
                        Console.WriteLine("{0}{1}{2}{3}", l1, l2, l3, l4);
                    }
                }
            }
        }
    }
}
```

To **avoid checking for equal rotated blocks** we use the following consideration: each block can be rotated in such a way, so **its smallest letter is at the first position** (at the top-left corner). For example, if we have a block **DCBA**, it can be rotated so that its smallest letter '**A**' comes at the first position. Thus, it is the same as **ABCD**. We may conclude, that all blocks can be obtained by putting the smallest letter at the first position and require that all other letters are bigger than the first. In the above code: the letter **l1** changes from [**A…lastLetter**] and **l2** > **l1**, **l3** > **l1** and **l4** > **l1**.

You can test the above code to make sure it's correct.

Calculating the Number of Blocks

To calculate the **number of blocks**, you can use a **counter in the innermost loop**. On the output though, we need the number of blocks to come first which leaves us two choices, both of which don't seem very efficient:

1. Store the results in a collection and then print it (takes up more memory and iterates the blocks twice)

2. First iterate to count the blocks (4 nested loops) and then iterate to print them (4 nested loops again, double the work).

The next thing to ask ourselves is: isn't there a **formula to calculate the number of blocks** we'll obtain?

We can check. Once we have a working solution we can print the number of blocks for several consecutive inputs and write down the results. There is an online encyclopedia of integer sequences; you can input a sequence of numbers and you can **find a formula** for calculating what the result would be based on the input number. You'll probably get more than one match; the more numbers you enter, the better the chance you'll find what you're looking for. For instance, enter the first 5 results (for n = 4, 5, 6, 7 and 8): **6, 30, 90, 210, 420**. The online encyclopedia prints a formula matching these numbers: **`a(n) = n * (n + 1) * (n + 2) * (n + 3) / 4`**.

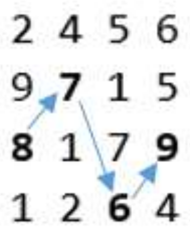
Have in mind that for this function **`a(1) = 6`**, which is what we want for input 4. Basically, when we get, say 8 as input, we actually need **`a(5)`** which is **`a(8 - 3)`**. So, the actual **formula** we need would turn into:

```
(n - 3) * (n - 2) * (n - 1) * n / 4
```

You can print the result of this expression above the nested loops and check it in the Judge system, it turns out it is correct! So, now we calculate the number of blocks with a simple formula and we only iterate once to print them which is a great improvement over the first idea we explored – using the recursive algorithm for generating variations.

Part II – Zig-Zag Matrix

You are given a **matrix of positive integer numbers**. A **sig-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 zigzag 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 (',').

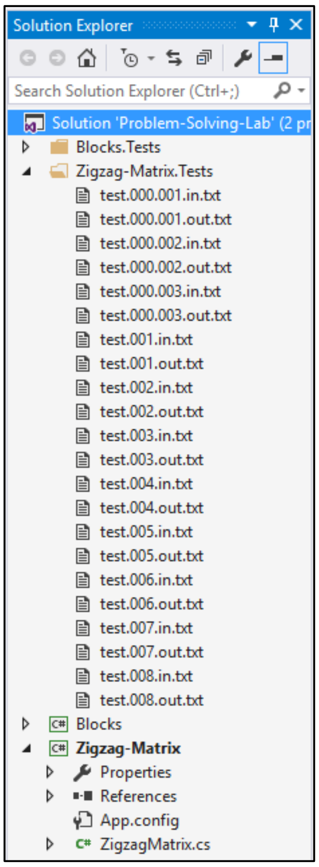
Print the maximal sum along with the path followed in format: **`{maxSum} = {cell1} + {cell2} + … + {cellM}`**. You can test your solution to the problem here.

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	3761 = 891 + 919 + 999 + 952

4 714,52,415,740 102,321,549,697 44,830,171,952 414,58,309,16 956,596,667,526 711,691,776,214 617,919,924,536 102,637,758,360 446,315,243,132 856,313,794,920 732,566,376,314 891,869,999,456 363,869,471,137 650,108,393,24 277,201,124,184 397,13,596,408 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.



```
Solution 'Problem-Solving-Lab' (2 pr
  Blocks.Tests
  Zigzag-Matrix.Tests
    test.000.001.in.txt
    test.000.001.out.txt
    test.000.002.in.txt
    test.000.002.out.txt
    test.000.003.in.txt
    test.000.003.out.txt
    test.001.in.txt
    test.001.out.txt
    test.002.in.txt
    test.002.out.txt
    test.003.in.txt
    test.003.out.txt
    test.004.in.txt
    test.004.out.txt
    test.005.in.txt
    test.005.out.txt
    test.006.in.txt
    test.006.out.txt
    test.007.in.txt
    test.007.out.txt
    test.008.in.txt
    test.008.out.txt
  Blocks
  Zigzag-Matrix
    Properties
    References
    App.config
    ZigzagMatrix.cs
```

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:

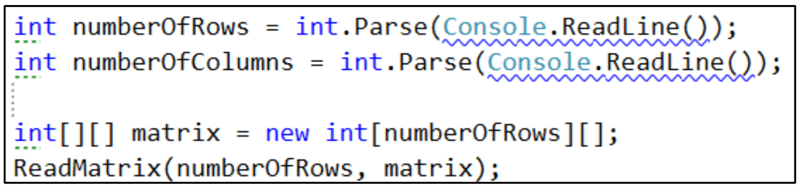
- A matrix of integers which we receive as input
- A matrix to hold the max path for each cell (we'll fill this using a DP approach)
- An array to keep track of the path we need to take once we're done (this is trickier though, an array won't do, we'll see why later)
- At some point, we'll need a list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

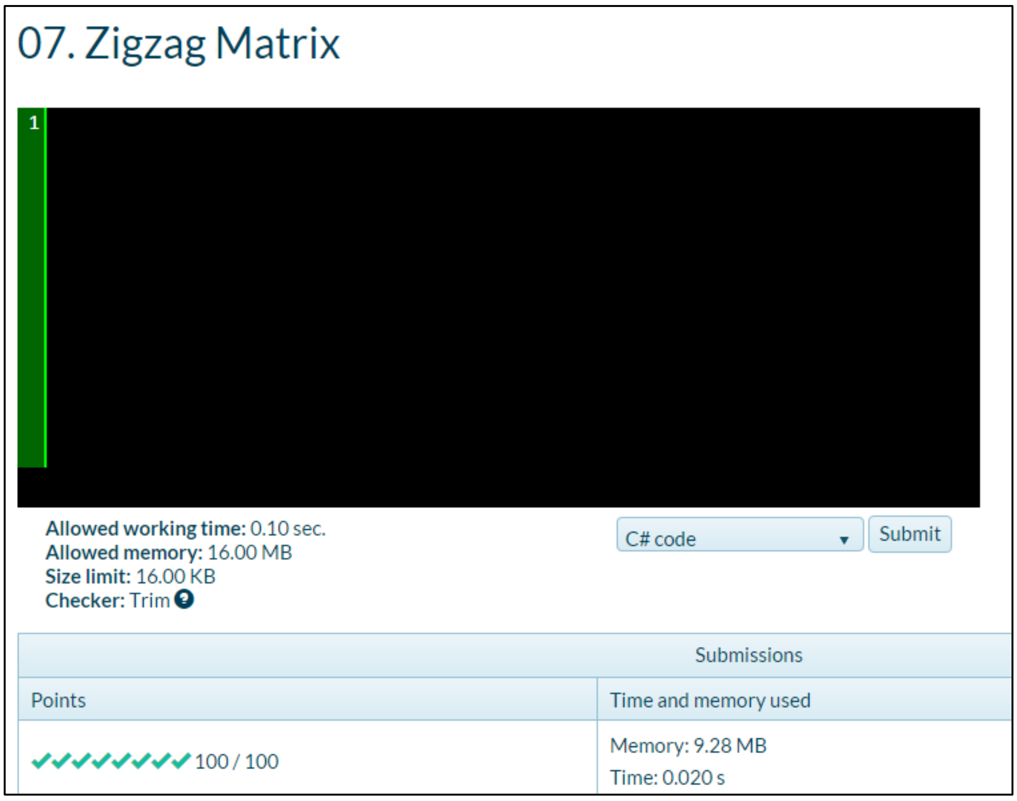
The **RecoverMaxPath()** method will start at the **currentRowIndex** and the last column and follow the path kept in **previousRowIndex** for that cell. It will put each cell value in a list and reverse it before returning it:

```
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

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