Exercises: Print Variations and Combinations

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

Part I – Generate Variations with Repetition

A variation (V^n_k) is a way of selecting **k** elements from a set of **n** elements in which the order of elements matters. For instance, if we have the set n = {1, 2, 3} and k = 2, this means we select two elements out of the three. If we allow repetitions, we can select the same number more than once, therefore, we'll have the following variations:

(1, 1) – since repetitions are acceptable

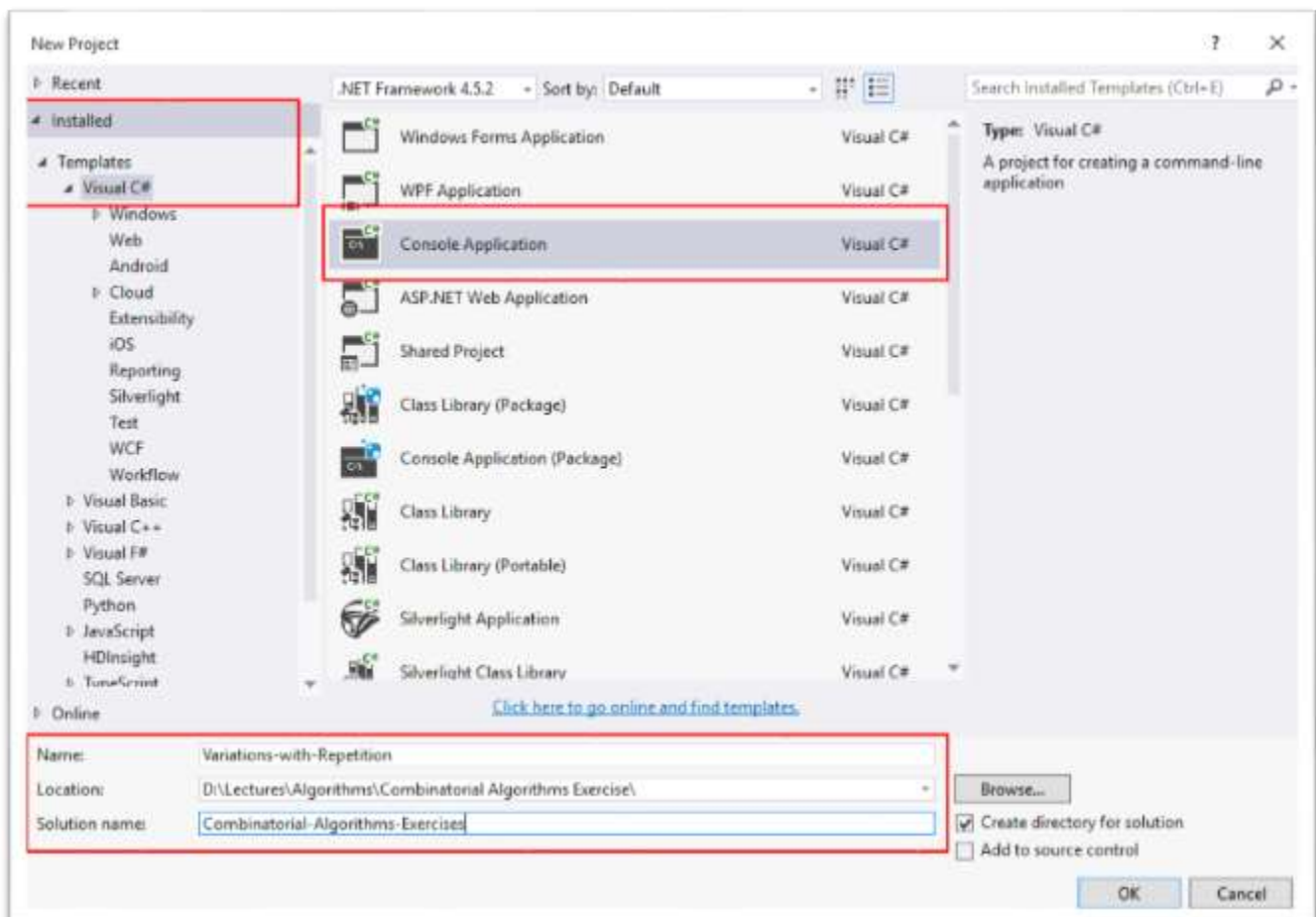(1, 2), (2, 1) – since order matters (unlike combinations)

(2, 2)

(1, 3), (3, 1)

(3, 3)

(2, 3), (3, 2)

Problem 1. Create a C# Console Application

For this exercise, you need to create a new Visual Studio solution to hold all of the problems below. Open Visual Studio (we recommend 2015 Community edition). Select **New Project** from the **Start Page** or the **File** menu.


Follow us:

In the pop-up window (see above), select **Visual C#** from the menu to the right and find the **Console Application** template.

At the bottom, there are three fields:

1) **Project name** – give a meaningful name to your project, e.g. Variations-with-Repetition
2) **Location** – choose an appropriate location for the project (the place where you keep all course materials)
3) **Solution name** – a solution is a collection of projects. We'll use a single solution to hold all projects for this exercise, so an appropriate name would be **Combinatorial-Algorithms-Exercises** or something similar

When you're ready, click OK to finish. A solution with a single project will be created.
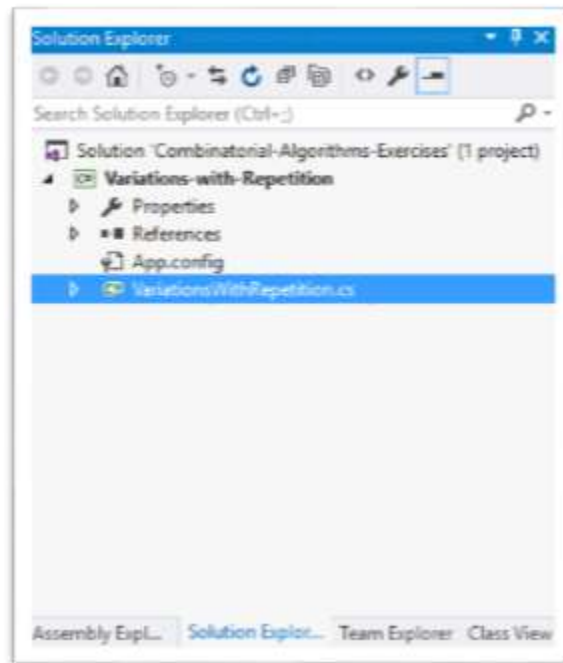
Rename the class and the auto-generated file, it is by default called Program.cs. Remove all unnecessary using directives:

```csharp
namespace Variations_with_Repetition
{
    public class VariationsWithRepetition
    {
        public static void Main(string[] args)
        {
        }
    }
}
```

You are now ready to start coding.

Problem 2. Setup

To simplify things, we'll use hardcoded values for now. Selecting two elements out of 3 is easy to keep track of and it produces just 9 variations. We'll need an **array of integers** to hold each variation we obtain, therefore, it should be of size **k**.

```
int n = 3;
int k = 2;

int[] array = new int[k];
```

Problem 3. Write the Recursive Method to Generate Variations

The method that will generate the variations, let's call it **GenerateVariations**, should accept as parameters: the **array** to hold the obtained variations, the **size of the set** and the **current index**:

```
private static void GenerateVariations(int[] array, int sizeOfSet, int index = 0)
{
```

The optional parameter index will allow us to call the method in our Main method like this:

```
GenerateVariations(array, n);
```

It will loop through all numbers from 1 to n (inclusive) and will place the current number at the current index. Next, it will generate all variations for the next index onward.

The bottom of the recursion is reached when we've filled **k** elements (we can use the size of the array instead of passing k to the method). So, the bottom of the recursion is simple, stop when the current index is outside the valid range. When we reach it, we print the result:

```
if (index >= array.Length)
{
    Print(array);
}
```

In the else clause, we have the loop from 1 to n:

```
else
{
    for (int i = 1; i <= sizeOfSet; i++)
    {
        array[index] = i;
        // TODO: Continue generating variations starting with index + 1
    }
}
```

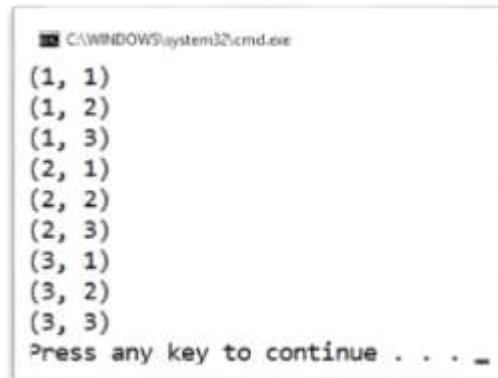Complete the TODO by calling the GenerateVariations method.

Problem 4. Write the Print() Method

In the print method, we need to print the currently obtained variation. **string.Join** makes this a one-liner:

```csharp
private static void Print(int[] array)
{
    Console.WriteLine("({0})", string.Join(", ", array));
}
```

Problem 5.   Test the Code

Run the code. It should produce the following result:

```
C:\WINDOWS\system32\cmd.exe
(1, 1)
(1, 2)
(1, 3)
(2, 1)
(2, 2)
(2, 3)
(3, 1)
(3, 2)
(3, 3)
Press any key to continue . . . _
```

Use the debugger to follow the process step by step and see how each of the results is produced.

Problem 6.   Remove Hardcoded Values

Go back to the Main method and replace the hardcoded values for n and k with input from the console:

```csharp
int n = int.Parse(Console.ReadLine());
int k = int.Parse(Console.ReadLine());

int[] array = new int[k];

GenerateVariations(array, n);
```

Problem 7.   Test the Code Again

Test with several different values for n and k. Stick to smaller numbers as it will get much harder to check whether the variations are correct.

Part II – Generate Variations without Repetition

The second problem is to generate variations **without** repetition, meaning that (1, 1) would no longer be valid. It's a simple modification of the previous program.

Problem 8.   Create a C# Console Application

In **Solution Explorer**, right-click the solution and select **Add** -> **New Project**.
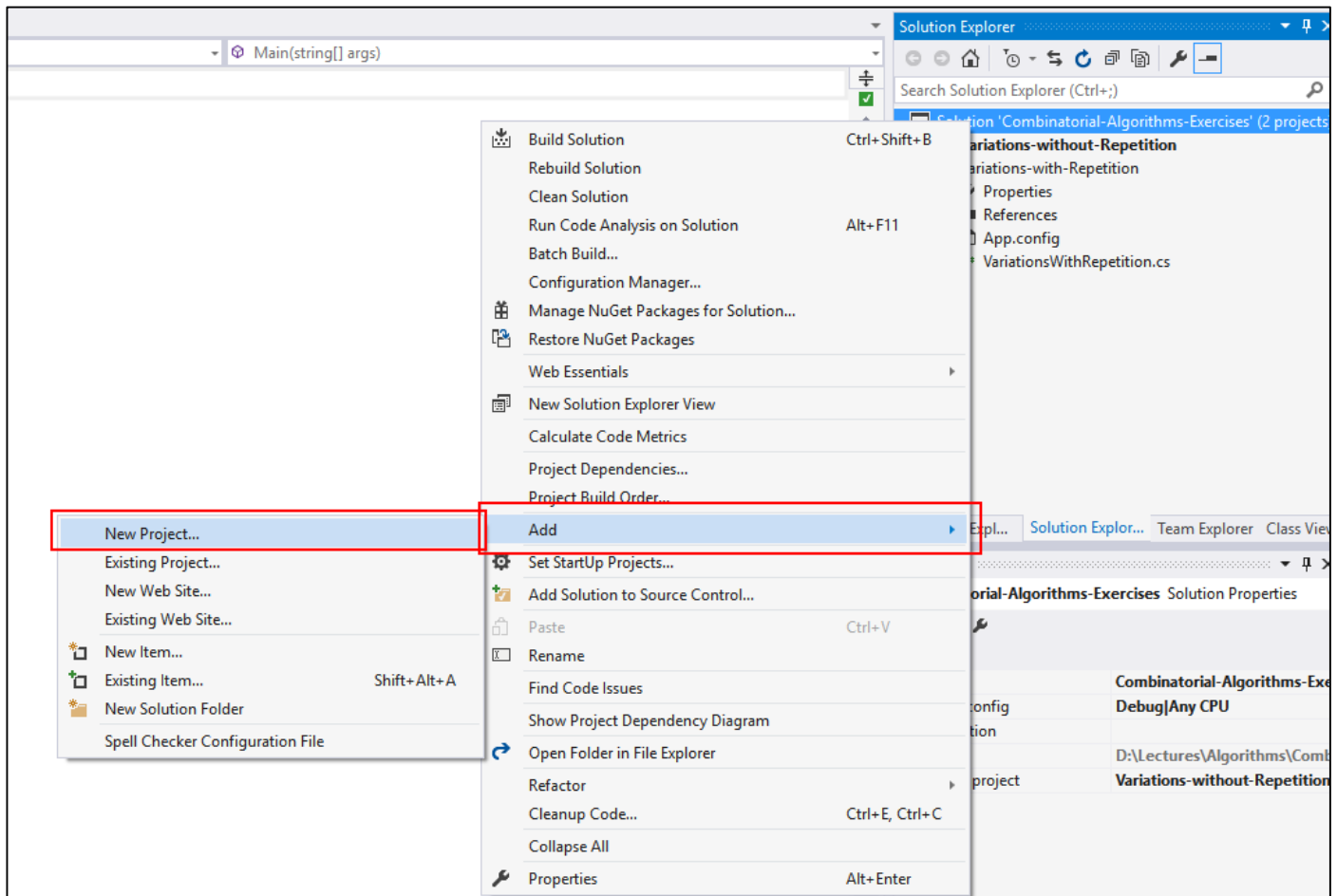
Since the problem is solved in almost the same manner like the previous one, **copy and paste the code from Part I** to the new project.

Problem 9. Keep Track of Used Numbers

To avoid repetitions, we just need to keep track of all the numbers we've already used. This can be done by creating an **array of Boolean values** which will simply tell us whether the number was used or not. To keep a correspondence between the number itself and its index in the new array, we can declare it with size **n + 1**, thus index 0 will never be used.

```
int[] array = new int[k];
bool[] used = new bool[n + 1];
```

The GenerateVariations method will have to take the new array as a parameter as well:

```
private static void GenerateVariations(int[] array, int sizeOfSet, bool[] used, int index = 0)
{
```

The idea is the following – we use only numbers which haven't been used, so we need a conditional statement inside the for-loop. As soon as we use a number, we mark it as used, then continue generating variations. After the recursive call, we need to unmark the number.

```
for (int i = 1; i <= sizeOfSet; i++)
{
    if (!used[i])
    {
        // TODO: Mark i as used
        array[index] = i;
        // TODO: Recursive call
        // TODO: Unmark i as used
    }
}
```

Problem 10. Test the Modified Code

Testing the code for n = 2 and k = 2 should produce the following (the same as the previous program, but repetitions are omitted):

```
C:\WINDOWS\system32\cmd.exe
3
2
(1, 2)
(1, 3)
(2, 1)
(2, 3)
(3, 1)
(3, 2)
Press any key to continue . . .
```

Part III – Generate Combinations with Repetition

In combinations, **order does not matter** (unlike variations). This means that the combinations (1, 2) and (2, 1) are considered the same and once we obtain one of them, the other is no longer valid.

Problem 11. Create a C# Console Application

Create a new project for the problem. Follow the steps outlined in problem 8.

Problem 12. Setup

Let's generate all combinations of 2 numbers from a set of 3 numbers.

```
int k = 2;
int n = 3;

int[] array = new int[k];
```

Problem 13. Write the Recursive Method

The method to generate combinations is almost the same as the one to generate variations. The difference is that once we use a given number we can no longer use it afterwards. To achieve this, we'll declare a new variable **start**

which will tell us at each recursive step where we should start the loop. E.g. after we've exhausted all combinations starting with 1, we'll just start with 2 the next time and won't go back to 1:

```
private static void GenerateCombinations(int[] array, int sizeOfSet, int index, int start = 1)
{
```

The only adjustments you need to make compared to the first problem (variations with repetition) is to pass the current number as start when doing the recursive call, and adjusting the loop so that it starts not from 1 every time, but from the number passed as start:

```
else
{
    // TODO: for-loop with modified start value of i
    {
        array[index] = i;
        // TODO: call the method recursively, incrementing the index and passing i as start
    }
}
```

Problem 14. Write the Print() Method

You may copy the Print() method from any of the previous problems.

Problem 15. Test the Code

Testing the code should produce the following results (remember, n = 3, k = 2):

```
C:\WINDOWS\system32\cmd.exe
(1, 1)
(1, 2)
(1, 3)
(2, 2)
(2, 3)
(3, 3)
Press any key to continue . . .
```

Problem 16. Remove Hardcoded Values and Retest

This step is the same as problems 6 and 7. Read n and k from the console and run the program several times with different values to make sure it is correct.

Part IV – Generate Combinations without Repetition

Problem 17. Create a C# Console Application

To keep your work organized, create a new project for the last problem for this exercise. **Copy and paste** the code from the previous problem as you'll only need to make a slight modification.

Follow us:

SOFTWARE UNIVERSITY
FOUNDATION

Page 7 of 8

Problem 18. Modify the Code

The only difference between this problem and the previous one is that you cannot repeat a number. You achieve this by **incrementing the start variable along with the current number** when you call the method recursively (i.e. the new start should be i + 1, instead of i).

Problem 19. Test

Selecting 2 elements out of 3 should produce the following result:

```
C:\WINDOWS\system32\cmd.exe
2
3
(1, 2)
(1, 3)
(2, 3)
Press any key to continue . . .
```

When repetitions aren't allowed, the number of combination decreases, so you can test with larger values, e.g. 3 elements out of 5 should produce this result:

```
C:\WINDOWS\system32\cmd.exe
3
5
(1, 2, 3)
(1, 2, 4)
(1, 2, 5)
(1, 3, 4)
(1, 3, 5)
(1, 4, 5)
(2, 3, 4)
(2, 3, 5)
(2, 4, 5)
(3, 4, 5)
Press any key to continue . . .
```

This is it! You've successfully written the recursive algorithms to generate variations and combinations (with and without repetition)!