Homework: Combinatorial Algorithms

This document defines the **homework assignments** for the "Algortihms" course @ Software University. Please submit a single **zip** / **rar** / **7z** archive holding the solutions (source code) of all below described problems.

Problem 1. Permutations

Write a recursive program for generating and printing all **permutations** (without repetition) of the numbers 1, 2, ..., n for a given integer number **n** (n > 0). The number of permutations is found by calculating n!

Step 1. Choose Appropriate Data Structures

We'll be working with the numbers from **1 to n**. To find all their permutations, we'll be swapping the numbers two by two. We know the number of elements we'll be working with and we'll be swapping elements in-place (we won't be using a separate structure). Therefore, we can use an **array of integers**.

Step 2. Setup

For now, let's find the permutations of the first three natural numbers – **1, 2, and 3**. We can easily replace all hardcoded values after we make sure the program works with a small enough number. We need an array containing the values 1, 2, 3. We can use the object initializer in C#, but that will pose a problem when we start testing with user input. We can use a for-loop or generate a sequence of numbers using the **Enumerable.Range** method by providing a start value and count of elements. Since the method returns IEnumerable<int>, we must call **ToArray()** to convert it to a regular array:

```
int n = 3;
var array = Enumerable.Range(1, n).ToArray();
```

Later, instead of 3, we'll read a number from the console and **array** will be generated automatically.

Since the number of permutations for larger values of **n** can be hard to keep track of, declare a static count variable and print it in the end. We know that if the program works correctly, the count should be **n!** when we're done.

```
private static int countOfPermutations = 0;

static void Main()
{
    int n = 3;
    var array = Enumerable.Range(1, n).ToArray();

    // TODO: Generate permutations and print them

    Console.WriteLine($"Total permutations: {countOfPermutations}");
}
```

Step 3. Understanding the Algorithm

The algorithm for generating permutations is readily available in the Internet. The hard part though is understanding what it does and why. This is by far the most important step of solving this problem, so use any means necessary to grasp the concept – pen and paper, the Visual Studio debugger, or any other tool that can help you.

The algorithm works like this:

1. Start at a given index **i** in the array
2. For each index **k** (starting with i), swap the two numbers at indices **i** and **k** and continue permuting from **i + 1** (this is the recursive call)
3. Stop the process when you reach the last element and print the array.

In essence, to find all permutations of the numbers 1, 2 and 3, we do the following steps:

- Loop from 1 to 3
- At each step, we have a different number at index 0
    - When at index 0 we have 1, we obtain all permutations starting with 1 by calling the Permute method for the rest of the numbers (indices 1 to n – 1). This will produce the permutations [1, 2, 3] and [1, 3, 2]
    - We swap 1 and 2, now 2 is at index 0. Again, we continue by finding all permutations of the numbers from index 1 to index n – 1. This will produce the permutations [2, 1, 3] and [2, 3, 1]. Before we continue, we swap back 1 and 2
    - We swap 1 and 3, now 3 is at index 0. We find all permutations starting with 3 by generating the permutations of the other two numbers, so we obtain [3, 2, 1] and [3, 1, 2].

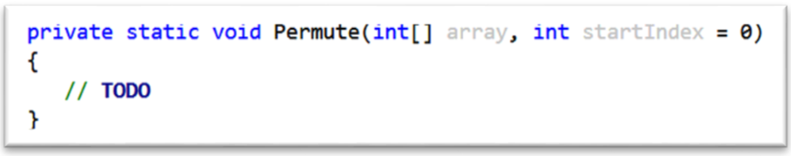
In short, we try each number at the first position and generate recursively all permutations of the rest of the numbers.

Go through the steps with pen and paper; with 3 elements it is manageable to keep track of each step and if you understand the process in principle, you'll be able to solve the problem with larger values for n.

Step 4. Coding the Solution

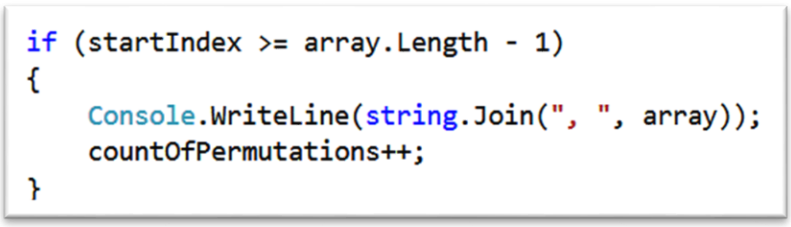
Go through the steps above until you understand how the program should work. When you know what needs to be done, coding the solution is the easier task.

We'll need a method which permutes the numbers starting from a given index. It should work on the array, so we can declare it as follows:



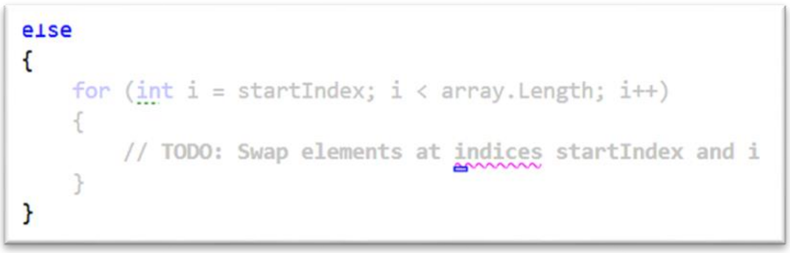
```
private static void Permute(int[] array, int startIndex = 0)
{
    // TODO
}
```

The easiest part is the bottom of the recursion. When start index is the last index, print the array and increment the count variable:



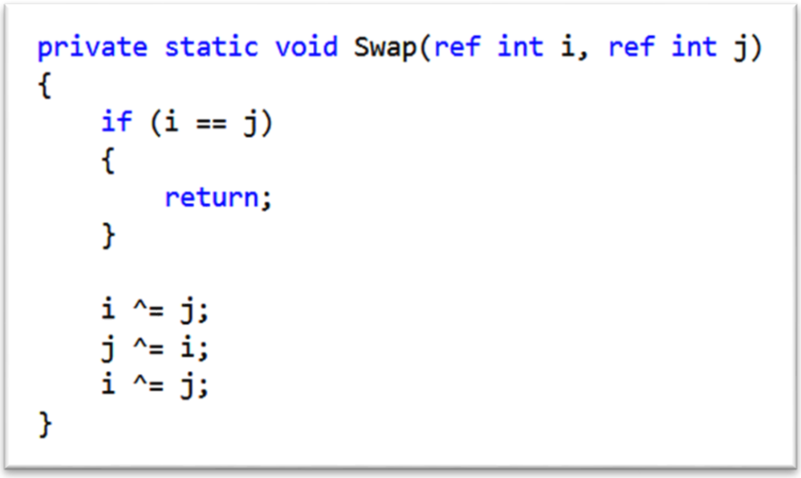
```
if (startIndex >= array.Length - 1)
{
    Console.WriteLine(string.Join(", ", array));
    countOfPermutations++;
}
```

Now, in the else clause we need to loop through all elements and swap each of them with the element at the startIndex:



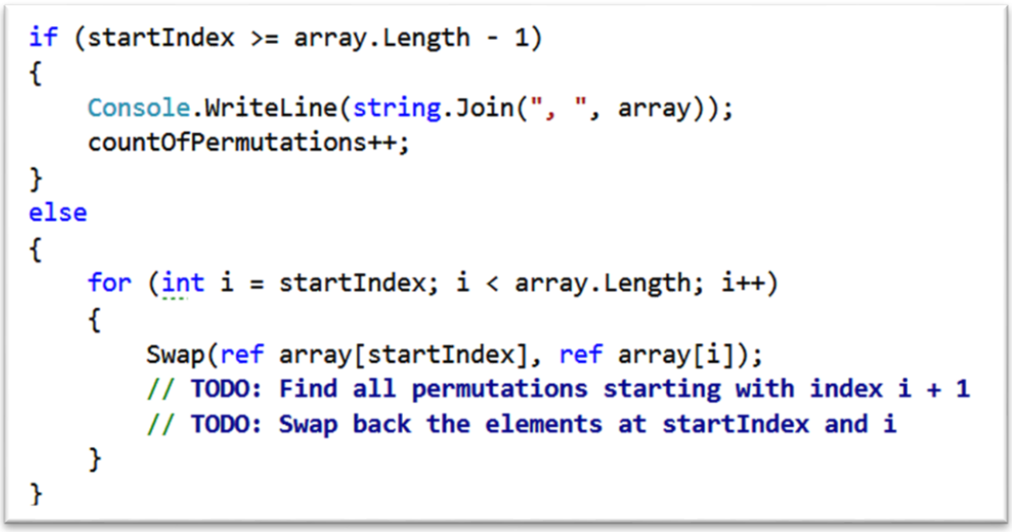
```
else
{
    for (int i = startIndex; i < array.Length; i++)
    {
        // TODO: Swap elements at indices startIndex and i
    }
}
```

Swapping two integer numbers can be done in several ways. Create a **Swap** method and use an implementation of your choice. Here is one using the XOR operator to swap two variable values without the use of a temporary variable:



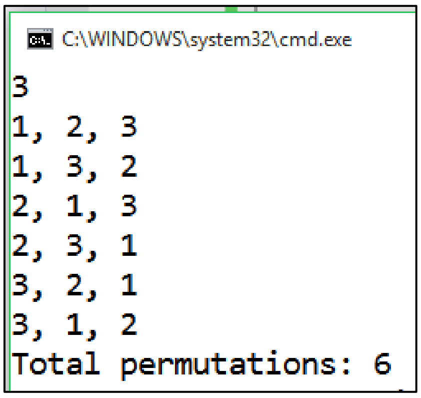
```
private static void Swap(ref int i, ref int j)
{
    if (i == j)
    {
        return;
    }

    i ^= j;
    j ^= i;
    i ^= j;
}
```

So, we need to call Swap, then find all permutations from i + 1 and swap back the two elements by calling Swap again with the same numbers like this:



```
if (startIndex >= array.Length - 1)
{
    Console.WriteLine(string.Join(", ", array));
    countOfPermutations++;
}
else
{
    for (int i = startIndex; i < array.Length; i++)
    {
        Swap(ref array[startIndex], ref array[i]);
        // TODO: Find all permutations starting with index i + 1
        // TODO: Swap back the elements at startIndex and i
    }
}
```

Complete the TODOs and test. This should be the result:

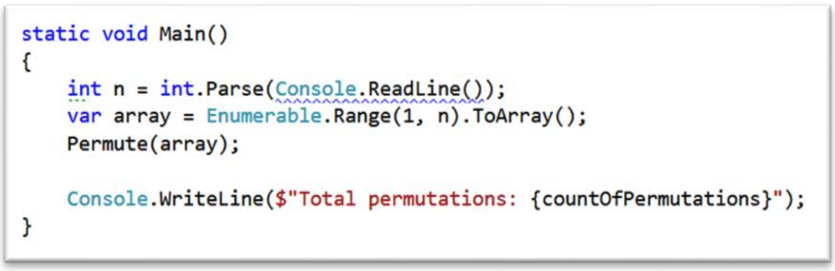


C:\WINDOWS\system32\cmd.exe

```
3
1, 2, 3
1, 3, 2
2, 1, 3
2, 3, 1
3, 2, 1
3, 1, 2
Total permutations: 6
```

Step 5. Remove Hardcoded Values and Retest

Finally, just modify the Main method to receive n from the console:



```
static void Main()
{
    int n = int.Parse(Console.ReadLine());
    var array = Enumerable.Range(1, n).ToArray();
    Permute(array);

    Console.WriteLine($"Total permutations: {countOfPermutations}");
}
```

Test with several different values. With n > 4 it will be hard to manually check if all permutations obtained are correct, but you can at least make sure they are as many as they should be – n!

Examples:

Input	Output

n=3	`1, 2, 3`
	`1, 3, 2`
	`2, 1, 3`
	`2, 3, 1`
	`3, 2, 1`
	`3, 1, 2`
	`Total permutations: 6`

n=5	`1, 2, 3, 4, 5`
	`1, 2, 3, 5, 4`
	`1, 2, 4, 3, 5`
	`1, 2, 4, 5, 3`
	`1, 2, 5, 4, 3`
	`1, 2, 5, 3, 4`
	`1, 3, 2, 4, 5`
	`1, 3, 2, 5, 4`
	`1, 3, 4, 2, 5`
	`1, 3, 4, 5, 2`
	…
	`5, 1, 2, 4, 3`
	`5, 1, 2, 3, 4`
	`Total permutations: 120`

Problem 2. Generate Permutations Iteratively

The above problem presented a recursive solution for generating all permutations (without repeating elements) of a collection. Your task is to write a **non-recursive algorithm** to achieve the same goal. There shouldn't be any recursive calls in your program (only loops). You may use the examples for problem 1 to check whether your solution is correct.

Hint: http://www.quickperm.org/

Problem 3. Generate Combinations Iteratively

Write an **iterative** program to generate all combinations (without repetition) of **k** elements from a set of **n** elements. Remember, in combinations, the order of elements doesn't matter – (1 2) and (2 1) are considered the same combination. You are not allowed to use recursion. Search the Internet for a suitable algorithm.

Input	Output

n=3
k=2
	1 2
	1 3
	2 3

n=5
k=3
	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

Problem 4. Generate Subsets of String Array

Write a recursive program for generating and printing all **subsets** of **k** strings from given set of strings **s**.

Input	Output	Solution with nested loops

s = {test, rock, fun}
k = 2
	(test rock)
	(test fun)
	(rock fun)
		see below

```
var set = new[] { "test", "rock", "fun" };
int k = 2;

for (int i1 = 0; i1 < set.Length; i1++)
{
    for (int i2 = i1 + 1; i2 < set.Length; i2++)
    {
        Console.WriteLine($"({set[i1]} {set[i2]})");
    }
}
```

Problem 5. Permutations with Repetition

Write a program to generate all **permutations with repetition** of a given multi-set. Ensure your program **efficiently** avoids duplicated permutations. Test it with { 1, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 }.

Input	Output	Comments
`s = {1, 3, 5, 5}`	`{ 1, 3, 5, 5 }`	You need to obtain only **unique** permutations. Note that in the set {1, 3, 5, 5} you can exchange the two 5s, but this produces essentially the same permutation. When there are many repeated elements this will cause significant perfomance issues.
	`{ 1, 5, 3, 5 }`	
	`{ 1, 5, 5, 3 }`	
	`{ 3, 1, 5, 5 }`	
	`{ 3, 5, 1, 5 }`	
	`{ 3, 5, 5, 1 }`	
	`{ 5, 1, 3, 5 }`	
	`{ 5, 1, 5, 3 }`	
	`{ 5, 3, 1, 5 }`	
	`{ 5, 3, 5, 1 }`	
	`{ 5, 5, 1, 3 }`	
	`{ 5, 5, 3, 1 }`	

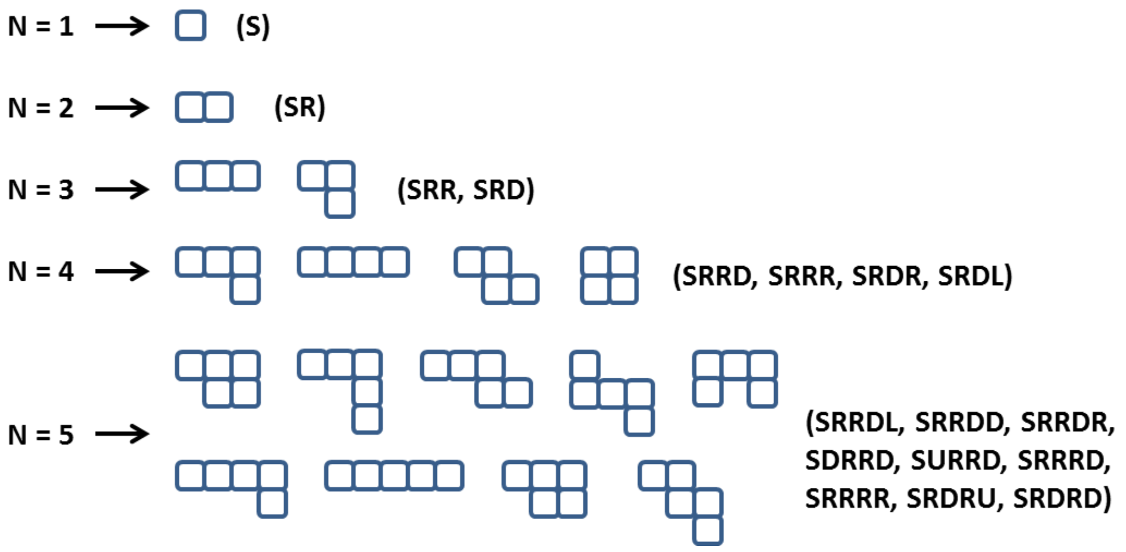
Hint: http://hardprogrammer.blogspot.bg/2006/11/permutaciones-con-repeticin.html

Problem 6. *Snakes

You can test your solution to the problem in the Judge system here.

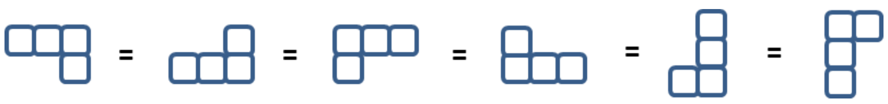
A **snake** is a sequence of several square blocks, attached one after another. A snake starts with a block at some position and continues with another block to the left, right, up or down, then again with another block to the left, right, up or down, etc. A snake of size **N** consists of a sequence of **N** blocks and is not allowed to cross itself.

You are given a number **N** and you should find all possible snakes of **N** blocks, represented as sequences of moves denoted as: **S** (start), **L** (move left), **R** (move right), **U** (move up) and **D** (move down). Examples (for N = 1, 2, 3, 4, and 5):

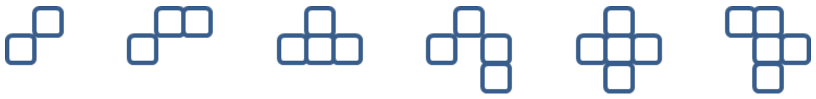

N = 1 → (S)
N = 2 → (SR)
N = 3 → (SRR, SRD)
N = 4 → (SRRD, SRRR, SRDR, SRDL)
N = 5 → (SRRDL, SRRDD, SRRDR, SDRRD, SURRD, SRRRD, SRRRR, SRDRU, SRDRD)

Note: some figures could look visually the same but represent different snakes, e.g. **SRRDL** and **SRDRU**.

Some snakes (sequences of blocks) are the same and should be printed only once. If after a number of rotations and/or flips two snakes are equal they are considered the same and should be printed only once. For example the snakes **SRRD**, **SRRU**, **SLLD, SLLU**, **SRUU** and **SUUR** are the same:



Not all forms consisting of N blocks are snakes of size N. Examples of non-snake forms:



Note: When generating the snakes, there may be different correct answers. When testing your solution, priority should be as follows: R -> D -> L -> U. The visual example above for n = 5 does NOT follow this priority.

Input

- The input should be read from the console.
- It will contain an integer number **N** in the range [1 ... 15].
- The input data will always be valid and in the format described. There is no need to check it explicitly.

Output

- The output should be printed on the console. It should consist of a variable number of lines:
- Each line should hold a snake represented as a sequence of moves.
- On the last line, print the number of snakes in format: **"Snakes count = {0}"**.

Constraints

- Allowed working time for your program: 10 seconds. Allowed memory: 512 MB.

Examples

Input	Sample Output	Comments

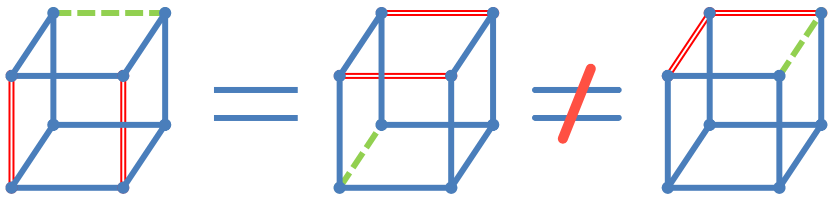
2	`SR`
	`Snakes count = 1`
		`Note that` **`SU`**`,` **`SL`** `and` **`SD`** `are also correct outputs. However, SR takes precedence because R has priority over all other directions.`

4	`SRRR`
	`SRRD`
	`SRDR`
	`SRDL`
	`Snakes count = 4`
		`Note that there are many other correct outputs for N = 4, but this is the expected output according to the priority of directions (right, down, left, up).`

Problem 7. *Cubes

You can test your solution to the problem in the Judge system here.

You are given 12 sticks of the same length, each colored with a color in the range [1 … 4]. Write a program to find the number of different cubes that can be built using these sticks. Note that two cubes are equal if a sequence of rotations exists that transforms the first cube to the second. For example, the first two cubes below are equal (after two rotations) but are different from the third cube:



Print on the console the number of different cubes that can be obtained using the sticks provided.

Input

- The input data should be read from the console. It will consist of a single line that contains 12 numbers in the range [1 … 4] separated by spaces.
- The input data will always be valid and in the format described. There is no need to check it explicitly.

Output

- The output should be printed on the console. It should consist of 1 line:
- On the only output line print a single integer number, representing the number of cubes that can be created using the provided sticks.

Constraints

- Allowed working time for your program: 0.75 seconds. Allowed memory: 128 MB.

Examples

Input	Output
`1 2 2 2 2 2 2 2 2 2 2 2`	`1`
`1 1 2 2 2 3 3 3 3 3 3 3`	`340`