Homework: Recursion

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. Reverse Array

Write a program that reverses and prints an array. Use **recursion**.

Input	Output
1 2 3 4 5 6	6 5 4 3 2 1

Problem 2. Nested Loops To Recursion

Write a program that simulates the execution of n nested loops **from 1 to n** which prints the values of all its iteration variables at any given time on a single line. **Use recursion.**

Examples:

Input	Output	Solution with nested loops (assuming n is positive)

n=2	1 1
	1 2
	2 1
	2 2
		`int n = 2;`
		
		`for (int i1 = 1; i1 <= n; i1++)`
		`{`
		`  for (int i2 = 1; i2 <= n; i2++)`
		`  {`
		`  Console.WriteLine($"{i1} {i2}");`
		`  }`
		`}`

n=3	1 1 1
	1 1 2
	1 1 3
	1 2 1
	1 2 2
	…
	3 2 3
	3 3 1
	3 3 2
	3 3 3
		`int n = 3;`
		
		`for (int i1 = 1; i1 <= n; i1++)`
		`{`
		`  for (int i2 = 1; i2 <= n; i2++)`
		`  {`
		`  for (int i3 = 1; i3 <= n; i3++)`
		`  {`
		`  Console.WriteLine($"{i1} {i2} {i3}");`
		`  }`
		`  }`
		`}`

Problem 3. Combinations with Repetition

Write a **recursive** program for generating and printing all combinations **with duplicates** of **k** elements from a set of **n** elements (k <= n). In combinations, the **order of elements doesn't matter**, therefore (1 2) and (2 1) are the same combination, meaning that once you print/obtain (1 2), (2 1) is no longer valid.

Examples:

Input	Output	Comments	Solution with nested loops

n=3
k=2
	(1 1)
	(1 2)
	(1 3)
		• n=3 => we have a set of three elements {1, 2, 3}	`int n = 3;`
			`int k = 2;`
			
			`// k == 2 => 2 nested for-loops`

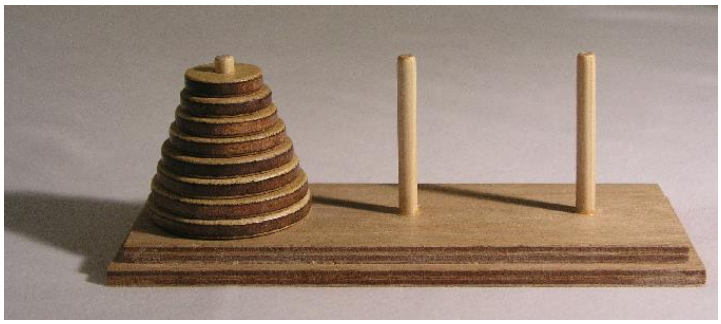
	(2 2)
	(2 3)
	(3 3)
		• k=2 => we select two elements out of the three each time
		• Duplicates are allowed, meaning (1 1) is a valid combination.
			`for (int i1 = 1; i1 <= n; i1++)`
			`{`
			`  for (int i2 = i1; i2 <= n; i2++)`
			`  {`
			`  Console.WriteLine($"({i1} {i2})");`
			`  }`
			`}`

n=5
k=3
	(1 1 1)
	(1 1 2)
	(1 1 3)
	(1 1 4)
	(1 1 5)
	(1 2 2)
	…
	(3 5 5)
	(4 4 4)
	(4 4 5)
	(4 5 5)
	(5 5 5)
		Select 3 elements out of 5 - {1, 2, 3, 4, 5}, a total of 35 combinations
		(1 2 1) is not valid as it's the same as (1 1 2)
			`int n = 5;`
			`int k = 3;`
			
			`// k == 3 => 3 nested for-loops`
			`for (int i1 = 1; i1 <= n; i1++)`
			`{`
			`  for (int i2 = i1; i2 <= n; i2++)`
			`  {`
			`  for (int i3 = i2; i3 <= n; i3++)`
			`  {`
			`  Console.WriteLine($"({i1} {i2} {i3})");`
			`  }`
			`  }`
			`}`

Problem 4. Tower of Hanoi

Your task is to solve the famous Tower of Hanoi puzzle using recursion.

In this problem, you have three rods (let's call them **source**, **destination** and **spare**). Initially, there are **n disks**, all placed on the source rod like in the picture below:



Your objective is to move all disks from the source rod to the destination rod. There are several **rules**:

1) Only one disk can be moved at a time
2) Only the topmost disk on a rod can be moved
3) A disk can only be placed on top of a larger disk or on an empty rod

Step 1. Choose Appropriate Data Structures

First, we need to decide how to model the problem in our program. The size of a disk can be represented by an **integer number** – the larger the number, the larger the disk.

How about the rods? According to the rules outlined above, we can either take a disk from the top of the rod or place a disk on top of it. This is an example of **Last-In-First-Out (LIFO)**, therefore, an appropriate structure to represent a rod would be **Stack<T>**. Since we'll be storing integers to represent the disks, we need **Stack<int>**, three of them to be precise – the **source**, the **destination** and the **spare**.

Step 2. Setup

Now that we have an idea what structures we'll be using, it's time for the initial setup. Before solving the puzzle for any number of disks, let's solve it with 3 and use hardcoded values. With 3 disks, it will be easier to keep track of the steps we'll take.

Initially, destination and spare are empty. In source, we need to have the numbers 1, 2, and 3, 1 being on top. We can use the **Enumerable.Range** method to obtain a sequence of integer numbers by providing a start value and count:

```
var range = Enumerable.Range(1, 3); // 1, 2, 3
```

The constructor of Stack<T> allows us to pass a collection which will be used to create the stack. If we pass the variable **range** to the constructor, the largest disk will be on top, which is not what we want, so we can call the **Reverse** method from LINQ to reverse the numbers. We can omit the range variable altogether and pass the result directly like this:

```
Stack<int> source = new Stack<int>(Enumerable.Range(1, 3).Reverse());
Stack<int> destination = new Stack<int>();
Stack<int> spare = new Stack<int>();
```

Step 3. Breaking down the Problem

The Tower of Hanoi is solved by breaking it down to sub-problems. What we'll try to do is:

1) Move all disks from source to destination starting with the largest (bottom disk)
    a) If the bottom disk is equal to 1, we can simply move it
    b) If the bottom disk is larger than 1
        I. move all disks above it (starting from bottom – 1) to the spare rod
        II. move the bottom disk to destination
        III. finally, move the disks now on spare to destination (back on top of the bottom disk)

In essence, steps **1.b.i** and **1.b.iii** repeat step 1, the only difference is that we're viewing different rods as source, destination and spare.

Step 4. Solution

Looking at step 3 above, it's apparent that we'll need a method which takes 4 arguments: the value of the bottom disk and the three rods (stacks).

```
private static void MoveDisks(int bottomDisk, Stack<int> source, Stack<int> destination, Stack<int> spare)
{
    // TODO
}
```

We need an if-statement to check if bottomDisk == 1 (the bottom of our recursion). If that's the case, we'll pop an element from the source and push it to the destination. We can do it on a single line like this:

```
if (bottomDisk == 1)
{
    destination.Push(source.Pop());
}
else
{
    // TODO
}
```

In the else clause, we need to do three things: 1) move all disks from bottomDisk - 1 from source to spare; 2) move the bottomDisk from source to destination; 3) move all disks from bottomDisk – 1 from spare to destination.

```
if (bottomDisk == 1)
{
    destination.Push(source.Pop());
}
else
{
    // TODO: Move disks on top of bottomDisk from source to spare
    destination.Push(source.Pop());
    // TODO: Move disks previously moved to spare to destination
}
```

Complete the TODOs in the above picture, by calling MoveDisks recursively. If you did everything correctly, this should be it! Now it's time to test it.

Step 5. Check Solution with Hardcoded Value

In order to check this solution, let's make the three stacks static and declare an additional variable which will keep track of the current number of steps taken.

```
private static int stepsTaken = 0;

private static Stack<int> source;
private static readonly Stack<int> destination = new Stack<int>();
private static readonly Stack<int> spare = new Stack<int>();
```

We'll need a method that prints the contents of all stacks, so we know which disk is where after each step:

```
private static void PrintRods()
{
    Console.WriteLine("Source: {0}", string.Join(", ", source.Reverse()));
    Console.WriteLine("Destination: {0}", string.Join(", ", destination.Reverse()));
    Console.WriteLine("Spare: {0}", string.Join(", ", spare.Reverse()));
    Console.WriteLine();
}
```

Having the needed variables and the PrintRods method, we can modify the Main method like this:

```
public static void Main(string[] args)
{
    int numberOfDisks = 3;
    source = new Stack<int>(Enumerable.Range(1, numberOfDisks).Reverse());
    PrintPegs();
    MoveDisks(numberOfDisks, source, destination, spare);
}
```

In this case, we make the stacks static, because from within the MoveDisks method we don't know which stack is which. Since the stacks are now static, check for a collision of variable names and rename the parameters of MoveDisks if necessary; here, we'll just append Rod to distinguish the static stacks from the method parameters. Now, in both the if and the else clause of MoveDisks, we need to increment the steps counter, print which disk has been moved and print the contents of the three stacks like this:

```
if (bottomDisk == 1)
{
    stepsTaken++;
    destinationRod.Push(sourceRod.Pop());
    Console.WriteLine($"Step #{stepsTaken}: Moved disk {bottomDisk}");
    PrintRods();
}
```

The same is repeated in the else clause, the difference being the recursive calls we make before and after the move.

After running the program you should now see each step of the process like this:

```
C:\WINDOWS\system32\cmd.exe
Source: 3, 2, 1
Destination:
Spare:

Step #1: Moved disk 1
Source: 3, 2
Destination: 1
Spare:

Step #2: Moved disk 2
Source: 3
Destination: 1
Spare: 2
```

The Tower of Hanoi puzzle always takes exactly **$2^n - 1$ steps**. With n == 3, all seven steps should be shown and in the end all disks should end up on the destination rod.

Using the output of your program and the debugger, follow each step and try to understand how this recursive algorithm works. It's much easier to see this with three disks.

Step 6. Remove Hardcoded Values and Retest

If everything went well and you're confident you've understood the process, you can replace 3 with input from the user, just read a number from the console.

Test with several different values, and make sure that the steps taken are $2^n - 1$ and that all disks are successfully moved from source to destination.

Here is the full example with 3 disks:

Input	Output
3	Source: 3, 2, 1 Destination: Spare: Step #1: Moved disk 1 Source: 3, 2 Destination: 1 Spare: Step #2: Moved disk 2 Source: 3 Destination: 1 Spare: 2 Step #3: Moved disk 1 Source: 3 Destination: Spare: 2, 1 Step #4: Moved disk 3 Source: Destination: 3 Spare: 2, 1 Step #5: Moved disk 1 Source: 1 Destination: 3 Spare: 2 Step #6: Moved disk 2 Source: 1 Destination: 3, 2 Spare: Step #7: Moved disk 1 Source: Destination: 3, 2, 1 Spare:

Congratulations, you just solved The Tower of Hanoi puzzle using recursion!

Problem 5. Combinations without Repetition

Modify the previous program to **skip duplicates, e.g. (1 1) is not valid.**

Examples:

Input	Output	Comments	Solution with nested loops

n=3
k=2
	(1 2)
	(1 3)
	(2 3)
		• n=3 => we have a set of three elements {1, 2, 3}
		• k=2 => we select two elements out of the three each time
		• Duplicates are not allowed, meaning (1 1) is not a valid combination.
			`int n = 3;`
			`int k = 2;`
			
			`// k == 2 => 2 nested for-loops`
			`for (int i1 = 1; i1 <= n; i1++)`
			`{`
			`  for (int i2 = i1 + 1; i2 <= n; i2++)`
			`  {`
			`  Console.WriteLine($"({i1} {i2})");`
			`  }`
			`}`

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)
		Select 3 elements out of 5 – {1, 2, 3, 4, 5}, a total of 10 combinations	`int n = 5;`
			`int k = 3;`
			
			`// k == 3 => 3 nested for-loops`
			`for (int i1 = 1; i1 <= n; i1++)`
			`{`
			`  for (int i2 = i1 + 1; i2 <= n; i2++)`
			`  {`
			`  for (int i3 = i2 + 1; i3 <= n; i3++)`
			`  {`
			`  Console.WriteLine($"({i1} {i2} {i3})");`
			`  }`
			`  }`
			`}`

Problem 6. Paths between Cells in Matrix

We are given a matrix of passable and non-passable cells. Write a recursive program for finding all paths between two cells in the matrix. The matrix can be represented by a two-dimensional char array or a string array, passable cells are represented by a **space (' ')**, non-passable cells are represented by **asterisks ('*')**, the start cell is represented by the symbol **'s'** and the exit cell is represented by **'e'**. Movement is allowed in all four directions (up, down, left, right) and a cell can be passed only once in a given path.

Print on the console all paths and on the last line the count of paths found. You can represent the directions with symbols, e.g. 'D' for down, 'U' for up, etc. The ordering of the paths is not relevant.

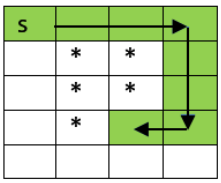
Examples:

Let's consider the following layout:

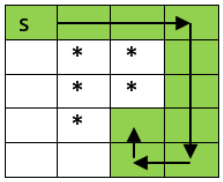
s			
	*	*	
	*	*	
	*	e	

There are several paths between the start point and the exit:

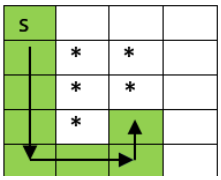
Path 1



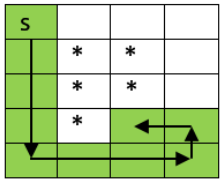
Path 2



Path 3



Path 4



Input Layout	Sample Output
s * * * * * e	RRRDDDL RRRDDDDLU DDDDRRRUL DDDDRRU Total paths found: 4
s * * * * * * * e *	RRRRRDDDLLL RRRRRDDDDLULL RRRDDDL DDDDRRU Total paths found: 4

Problem 7. Connected Areas in a Matrix

Let's define a connected area in a matrix as an area of cells in which there is a path between every two cells. Write a program to find **all** connected areas in a matrix. On the console, print the total number of areas found, and on a separate line some info about each of the areas – its position (top-left corner) and size. Order the areas by size (in descending order) so that the largest area is printed first. If several areas have the same size, order them by their position, first by the row, then by the column of the top-left corner. So, if there are two connected areas with the same size, the one which is above and/or to the left of the other will be printed first.

Examples:

Input Layout	Sample Output
1 * 2 * 3 * * * * * * * * *	Total areas found: 3 Area #1 at (0, 0), size: 13 Area #2 at (0, 4), size: 10 Area #3 at (0, 8), size: 5
* 1 * 3 * 2 * * * * * * * * * * * 4 * * * *	Total areas found: 4 Area #1 at (0, 1), size: 10 Area #2 at (0, 8), size: 10 Area #3 at (0, 4), size: 6 Area #4 at (3, 4), size: 6

Hints:

- Create a method to find the first traversable cell which hasn't been visited. This would be the top-left corner of a connected area. If there is no such cell, this means all areas have been found.
- You can create a class to hold info about a connected area (its position and size). Additionally, you can implement IComparable and store all areas found in a SortedSet.