# Homework: Recursion

This document defines the **homework assignments** for the ["Algortihms" course @ Software University](https://softuni.bg/trainings/1194/Algorithms-September-2015). Please submit a single zip / rar / 7z archive holding the solutions (source code) of all below described problems.

## Tower of Hanoi

Your task is to solve the famous [Tower of Hanoi](https://en.wikipedia.org/wiki/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**.

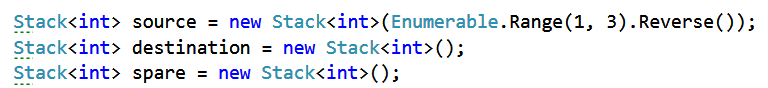
### Step 2. Setup

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.

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:



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:



### Step 3. Breaking down the Problem

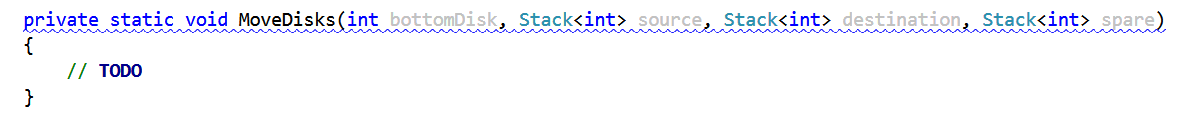
The Tower of Hanoi is solved by breaking it down to sub-problems. What we’ll try to do is:

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

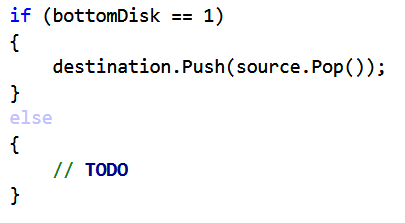
In essence, steps **1.b.i** and **1.b.iii** repeat step 1, the only difference is that we’re viewing different rods as source, destination and spare.

### Step 4. Solution

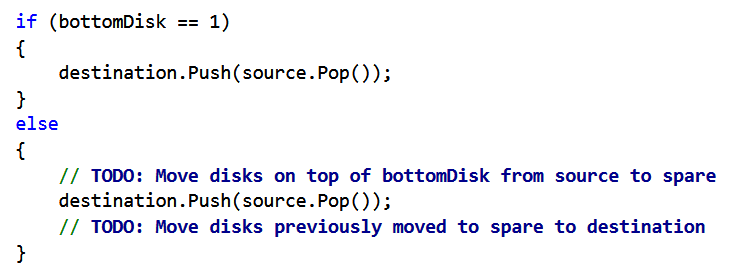
Looking at step 3, it’s apparent that we’ll need a method which takes 4 arguments: the value of the bottom disk and the three rods (stacks).



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:



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.

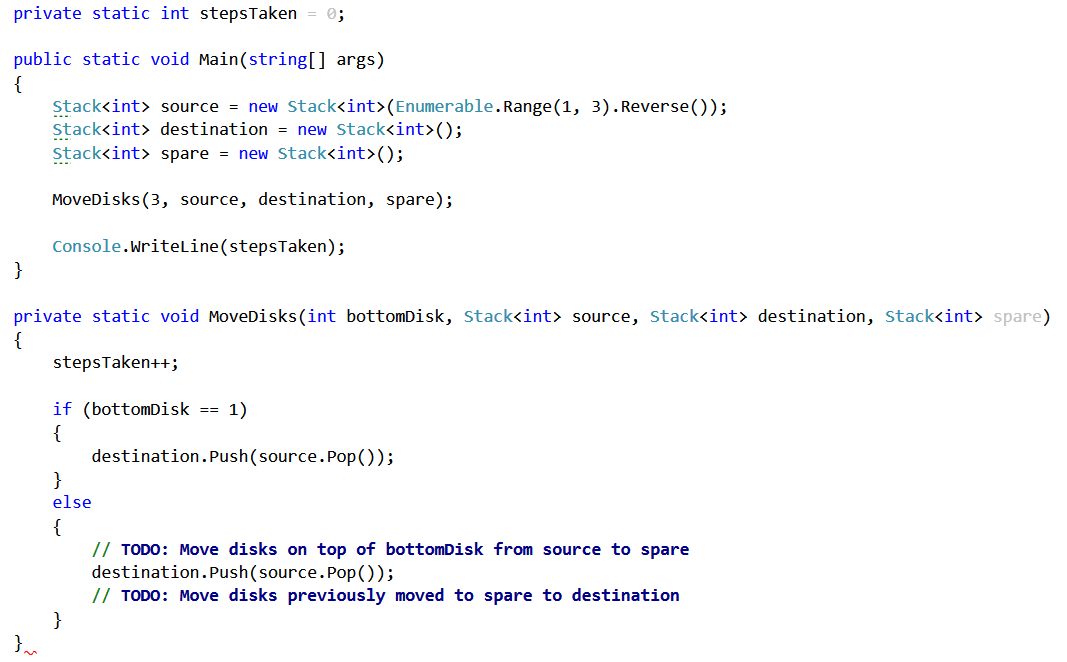


Complete the TODOs in the above picture, by calling MoveDisks recursively. If you did everything correctly, this should be it!

### Step 5. Check Solution with Hardcoded Value

Call your method with bottomDisk == 3, place a breakpoint after it and check out the values in all three stacks. If your algorithm is correct, source and spare should be empty and destination should contain the numbers 3, 2 and 1.

Another way to check is to count the number of steps taken. The Tower of Hanoi takes exactly **2n – 1 steps** to solve, so if n == 3, you should finish in 7 steps. You can declare a static count variable and print it:



### Step 6. Remove Hardcoded Values and Retest

If everything went well, you can replace 3 with input from the user. Test with several different values, check out the stacks and make sure that stepsTaken is equal to 2n – 1.

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

## 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}");  }  }  } |

## 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)  (2 2)  (2 3)  (3 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 allowed, meaning (1 1) is 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; 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})");  }  }  } |

## 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})");  }  }  } |

## 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

|  |  |  |  |
| --- | --- | --- | --- |
| s |  |  |  |
|  | \* | \* |  |
|  | \* | \* |  |
|  | \* |  |  |
|  |  |  |  |

Path 3

|  |  |  |  |
| --- | --- | --- | --- |
| s |  |  |  |
|  | \* | \* |  |
|  | \* | \* |  |
|  | \* |  |  |
|  |  |  |  |

Path 2

|  |  |  |  |
| --- | --- | --- | --- |
| s |  |  |  |
|  | \* | \* |  |
|  | \* | \* |  |
|  | \* |  |  |
|  |  |  |  |

Path 4

|  |  |  |  |
| --- | --- | --- | --- |
| s |  |  |  |
|  | \* | \* |  |
|  | \* | \* |  |
|  | \* |  |  |
|  |  |  |  |

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

## 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.