# Exercise: Recursion and Combinatorial Problems

This document defines the lab for the "Algorithms – Fundamentals (Java)" course @ Software University.

Please submit your solutions (source code) to all below-described problems in Judge.

## 1. Reverse Array

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

### Examples

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

## 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<br>(assuming n is positive) |
|---|---|---|
| 2 | 1 1<br>1 2<br>2 1<br>2 2 | `int limit = 2;`<br><br>`for (int i1 = 1; i1 <= limit; i1++) {`<br>    `for (int i2 = 1; i2 <= limit; i2++) {`<br>        `System.out.println(i1 + " " + i2);`<br>    `}`<br>`}` |
| 3 | 1 1 1<br>1 1 2<br>1 1 3<br>1 2 1<br>1 2 2<br>…<br>3 2 3<br>3 3 1<br>3 3 2<br>3 3 3 | `int limit = 3;`<br><br>`for (int i1 = 1; i1 <= limit; i1++) {`<br>    `for (int i2 = 1; i2 <= limit; i2++) {`<br>        `for (int i3 = 1; i3 <= limit; i3++) {`<br>            `System.out.println(i1 + " " + i2 + " " + i3);`<br>        `}`<br>    `}`<br>`}` |

## 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matter**, therefore (1 2) and (2 1) are the same combination, meaning that once you print/obtain (1 2), (2 1) is no longer valid.

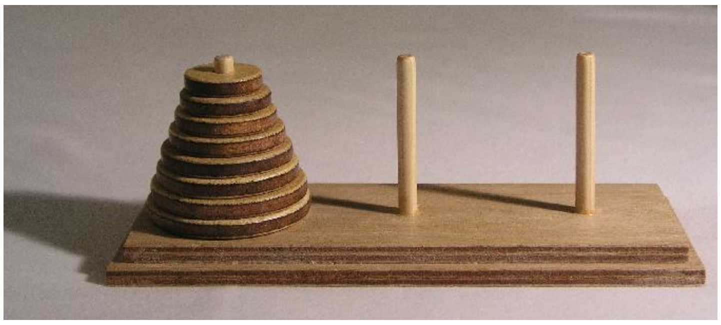
### Examples

| Input | Output | Comments |
|---|---|---|
| 3<br>2 | 1 1<br>1 2<br>1 3<br>2 2<br>2 3<br>3 3 | • n=3 => we have a set of three elements {1, 2, 3}<br>• k=2 => we select two elements out of the three each time<br>• Duplicates are allowed, meaning (1 1) is a valid combination. |
| 5<br>3 | 1 1 1<br>1 1 2<br>1 1 3<br>1 1 4<br>1 1 5<br>1 2 2<br>…<br>3 5 5<br>4 4 4<br>4 4 5<br>4 5 5<br>5 5 5 | Select 3 elements out of 5 – {1, 2, 3, 4, 5}, a total of 35 combinations<br>(1 2 1) is not valid as it's the same as (1 1 2) |

## 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** we need three of them to be precise – the **source**, the **destination,** and the **spare**.

## Step 2. Setup

Now that we have an idea of 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, the destination and spare are empty. In the source, we need to have the numbers 1, 2, and 3, 1 being on top.

## Step 3. Breaking down the Problem

The Tower of Hanoi is solved by breaking it down into 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 the 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 a source, destination, and spare.

## Step 4. Solution

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

```java
private static void solve(int disk, Deque<Integer> source, Deque<Integer> destination, Deque<Integer> spare) {
    // TODO: Add implementation
}
```

We need an if-statement to check if disk == 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:

```java
private static void solve(int disk, Deque<Integer> source, Deque<Integer> destination, Deque<Integer> spare) {
    if (disk == 1) {
        destination.push(source.pop());
    } else {
        // TODO: Add recursive call
    }
}
```

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.

```java
private static void solve(int disk, Deque<Integer> source, Deque<Integer> destination, Deque<Integer> spare) {
    if (disk == 1) {
        destination.push(source.pop());
    } else {
        // TODO: Move disk on top of disk from source to spare
        // TODO: Move top disk from source to destination
        // TODO: Move disk 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 that will keep track of the current number of steps taken.

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

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

```
Source: 3, 2, 1
Destination:
Spare:

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

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

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

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:

## Examples

| Input | Output |
|---|---|
| 3 | `Source: 3, 2, 1`<br>`Destination:`<br>`Spare:`<br><br>`Step #1: Moved disk`<br>`Source: 3, 2`<br>`Destination: 1`<br>`Spare:`<br><br>`Step #2: Moved disk`<br>`Source: 3`<br>`Destination: 1`<br>`Spare: 2`<br><br>`Step #3: Moved disk`<br>`Source: 3`<br>`Destination:`<br>`Spare: 2, 1`<br><br>`Step #4: Moved disk`<br>`Source:`<br>`Destination: 3`<br>`Spare: 2, 1`<br><br>`Step #5: Moved disk`<br>`Source: 1`<br>`Destination: 3`<br>`Spare: 2`<br><br>`Step #6: Moved disk`<br>`Source: 1`<br>`Destination: 3, 2`<br>`Spare:`<br><br>`Step #7: Moved disk` |

| | |
|---|---|
| | `Source:`<br>`Destination: 3, 2, 1`<br>`Spare:` |

## 5. Combinations without Repetition

Modify the solution from the **problem 3** program to **skip duplicates, e.g. (1 1) is not valid.**

### Examples

| Input | Output | Comments |
|---|---|---|
| 3<br>2 | 1 2<br>1 3<br>2 3 | • n=3 => we have a set of three elements {1, 2, 3}<br>• k=2 => we select two elements out of the three each time<br>• Duplicates are not allowed, meaning (1 1) is not a valid combination. |
| 5<br>3 | 1 2 3<br>1 2 4<br>1 2 5<br>1 3 4<br>1 3 5<br>1 4 5<br>2 3 4<br>2 3 5<br>2 4 5<br>3 4 5 | Select 3 elements out of 5 – {1, 2, 3, 4, 5},<br>a total of 10 combinations |

## 6. 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.

On the first line, you will get the **number of rows**.

On the second line, you will get the **number of columns**.

The rest of the input will be the **actual matrix**.

### Examples

| Example Layout | Output |
|---|---|

| | |
|---|---|
| 4<br>9<br>---*---*-<br>---*---*-<br>---*---*-<br>----*-*-- | Total areas found: 3<br>Area #1 at (0, 0), size: 13<br>Area #2 at (0, 4), size: 10<br>Area #3 at (0, 8), size: 5 |
| 5<br>10<br>*--*---*--<br>*--*---*--<br>*--*****--<br>*--*---*--<br>*--*---*-- | Total areas found: 4<br>Area #1 at (0, 1), size: 10<br>Area #2 at (0, 8), size: 10<br>Area #3 at (0, 4), size: 6<br>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 Comparable and store all areas found in a TreeSet.

## 7. Cinema

Write a program that prints all of the possible **distributions** of a group of friends in a **cinema hall**. In the **first line,** you will be given all of the friends' **names** separated by a **comma and space**. Until you receive the command **`"generate"`** you will be given some of those friends's **names** and a **number of the place** that they want to have. (format: **`"{name} - {place}"`**) So here comes the tricky part. Those friends **want** only to **sit** in the place that they **have chosen**. They **cannot sit in other places**. For more clarification see the examples below.

### Output

Print all the **possible ways to distribute the friends** having in mind that some of them want a particular place and they will sit there only. The **order** of the output does **not matter**.

### Constrains

- The friend's **names** and the **number** of the place will always be valid.

### Examples

| **Input** | **Output** | **Comments** |
|---|---|---|

| | | |
|---|---|---|
| `Peter, Amy, George, Rick`<br>`Amy - 1`<br>`Rick - 4`<br>`generate` | `Amy Peter George Rick`<br>`Amy George Peter Rick` | Amy only wants to sit on the first seat and Rick wants to sit on the fourth, so we only switch the other friends |
| `Garry, Liam, Teddy, Anna, Buddy, Simon`<br>`Buddy - 3`<br>`Liam - 5`<br>`Simon - 1`<br>`generate` | `Simon Garry Buddy Teddy Liam Anna`<br>`Simon Garry Buddy Anna Liam Teddy`<br>`Simon Teddy Buddy Garry Liam Anna`<br>`Simon Teddy Buddy Anna Liam Garry`<br>`Simon Anna Buddy Garry Liam Teddy`<br>`Simon Anna Buddy Teddy Liam Garry` | |

## 8. Word Cruncher

Write a program that receives some **strings** and **forms another** string that is required. In the **first line,** you will be given **all of the strings** separated by a **comma and space**. On the next line, you will be given the **string** that needs to be **formed from the given strings**. For more clarification see the examples below.

### Input

- On the first line you will receive the **strings** (separated by comma and space `", "`).
- On the next line you will receive the **target string.**

### Output

- Print each of them found ways to form the required string as shown in the examples.

### Constrains

- There might be **repeating elements** in the input.

### Examples

| Input | Output |
|---|---|
| `text, me, so, do, m, ran`<br>`somerandomtext` | `so me ran do m text` |

| | |
|---|---|
| `this, th, is, Word, cruncher, cr, h, unch, c, r, un, ch, er`<br>`Wordcruncher` | `Word c r un ch er`<br>`Word c r unch er`<br>`Word cr un c h er`<br>`Word cr un ch er`<br>`Word cr unch er`<br>`Word cruncher` |

## 9. School Teams

Write a program that receives the **names** of **girls** and **boys** in a class and generates **all possible ways** to create **teams** with **3 girls** and **2 boys**. Print each team on a **separate line** separated by a comma and space **", "** (**first** the **girls then** the **boys**). For more clarification see the examples below

***Note***: **"Linda, Amy, Katty, John, Bill"** is the **same as "Linda, Amy, Katty, Bill, John"**; so print only **the first case**

### Input

- On the **first line** you will receive the **girl's** names separated by a comma and space **", ".**
- On the **second line** you will receive the **boy's** names separated by a comma and space **", ".**

### Output

- On **separate lines** print all the possible **teams** with exactly **3 girls** and **2 boys** separated by comma and space and starting with the girls.

### Constrains

- There will always be **at least 3 girls** and **2 boys** in the input.

### Examples

| Input | Output |
|---|---|
| `Linda, Amy, Katty`<br>`John, Bill` | `Linda, Amy, Katty, John, Bill` |
| `Lisa, Yoana, Marta, Rachel`<br>`George, Garry, Bob` | `Lisa, Yoana, Marta, George, Garry`<br>`Lisa, Yoana, Marta, George, Bob`<br>`Lisa, Yoana, Marta, Garry, Bob`<br>`Lisa, Yoana, Rachel, George, Garry`<br>`Lisa, Yoana, Rachel, George, Bob`<br>`Lisa, Yoana, Rachel, Garry, Bob`<br>`Lisa, Marta, Rachel, George, Garry`<br>`Lisa, Marta, Rachel, George, Bob`<br>`Lisa, Marta, Rachel, Garry, Bob`<br>`Yoana, Marta, Rachel, George, Garry`<br>`Yoana, Marta, Rachel, George, Bob`<br>`Yoana, Marta, Rachel, Garry, Bob` |