# PART 1: RECURSIVE

## Recursive Drawing

Write a program that draws the figure below depending on n. Use **recursion**.

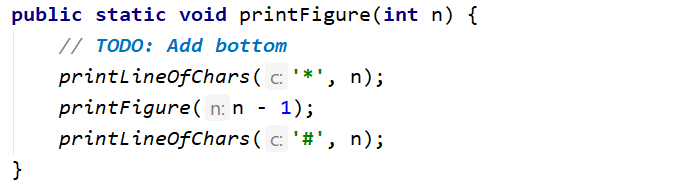
### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 2 | \*\*  \*  #  ## |
| 5 | \*\*\*\*\*  \*\*\*\*  \*\*\*  \*\*  \*  #  ##  ###  ####  ##### |

### Hints

Set the bottom of the recursion

Define pre- and post- recursive behavior



## Generating 0/1 Vectors

Generate all n-bit vectors of zeroes and ones in lexicographic order.

### Examples

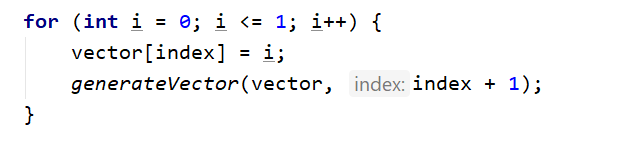
|  |  |
| --- | --- |
| **Input** | **Output** |
| 3 | 000  001  010  011  100  101  110  111 |
| 5 | 00000  00001  00010  …  11110  11111 |

### Hints

The method should receive as parameters the array which will be our vector and a current index

Bottom of recursion should be when the index is outside of the vector

To generate all combinations, create a for loop with a recursive call



## Recursive Fibonacci

Each member of the **Fibonacci sequence** is calculated from the **sum of the two previous members**. The first two elements are 1, 1. Therefore the sequence goes as 1, 1, 2, 3, 5, 8, 13, 21, 34…

The following sequence can be generated with an array, but that’s easy, so **your task is to implement it recursively**.

If the function **getFibonacci(n)** returns the nth Fibonacci number, we can express it using **getFibonacci(n) = getFibonacci(n-1) + getFibonacci(n-2)**.

However, this will never end and in a few seconds a Stack Overflow Exception is thrown. In order for the recursion to stop it has to have a "bottom". The bottom of the recursion is getFibonacci(1), and should return 1. The same goes for getFibonacci(0).

### Input

* On the only line in the input the user should enter the wanted Fibonacci number N where 1 ≤ N ≤ 49

### Output

* The output should be the nth Fibonacci number counting from 0

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5 | 8 |
| 10 | 89 |
| 21 | 17711 |

## Computing Powers

Computing a positive integer power of a number is easily seen as a recursive process.

Consider an example:

an : ­ If n = 0, an is 1 (by definition) ­

If n > 0, an is a \* an–1

Write a program to compute powers based on base as a and exponent as n with recursive

### Examples

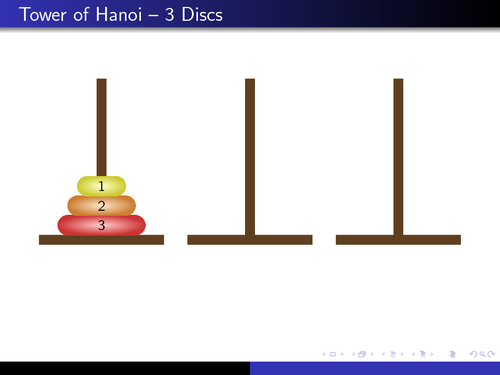
|  |  |
| --- | --- |
| **Input** | **Output** |
| **3**  **3** |  |
| **4**  **7** |  |

## Hanoi Tower

This is a fun puzzle game where the objective is to move an entire stack of disks from the source position to another position. Three simple rules are followed:

* Only one disk can be moved at a time.
* Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack. In other words, a disk can only be moved if it is the uppermost disk on a stack.
* No larger disk may be placed on top of a smaller disk.

With n is a number of disks as an input.





**Examples**

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3 | 23 1 0  3 1 2  3 0 12  0 3 12  1 3 2  1 23 0  0 123 0 |

1. **Option 1: Solve it with recursive**

***Hint: Use your SinglyLinkedList as a tower to store disks.***

1. **Option 2: Solve it with non-recursive**

# PART 2: RECURSIVE WITH BACKTRACKING

## Find All Paths in a Labyrinth

You are given a labyrinth. Your goal is to find all paths from the start (cell 0, 0) to the exit, marked with 'e'.

* Empty cells are marked with a dash '-'
* Walls are marked with a star '\*'

On the first line, you will receive the dimensions of the labyrinth. Next you will receive the actual labyrinth.

The order of the paths does not matter.

**Examples**

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3  3  ---  -\*-  --e | RRDD  DDRR |
| 3  5  -\*\*-e  -----  \*\*\*\*\* | DRRRRU  DRRRUR |

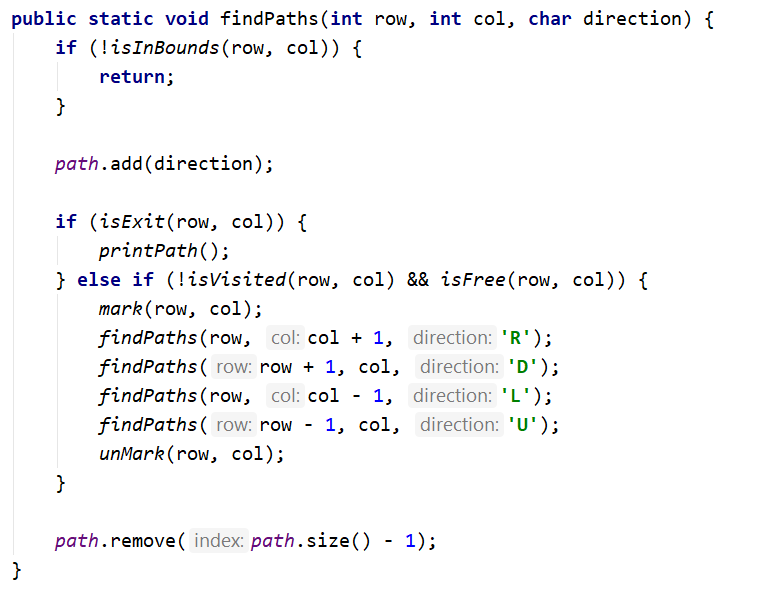
**Hints**

Create methods for reading and finding all paths in the labyrinth.

Create a static list that will hold every direction (basically the path)



Finding all paths should be recursive



Implement all helper methods that are present in the code above.

## Queens Puzzle

In this lab we will implement a recursive algorithm to solve the **"8 Queens" puzzle**. Our goal is to write a program to **find all possible placements of 8 chess queens** on a chessboard, so that no two queens can attack each other (on a row, column or diagonal).

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| *(no input)* | \* - - - - - - -  - - - - \* - - -  - - - - - - - \*  - - - - - \* - -  - - \* - - - - -  - - - - - - \* -  - \* - - - - - -  - - - \* - - - -  \* - - - - - - -  - - - - - \* - -  - - - - - - - \*  - - \* - - - - -  - - - - - - \* -  - - - \* - - - -  - \* - - - - - -  - - - - \* - - -  …  *(90 solutions more)* |

## Learn about the "8 Queens" Puzzle

Learn about the "8 Queens" puzzle, e.g. from Wikipedia: <http://en.wikipedia.org/wiki/Eight_queens_puzzle>.

## Define a Data Structure to Hold the Chessboard

First, let’s define a data structure to hold the **chessboard**. It should consist of 8 x 8 cells, each either occupied by a queen or empty. Let’s also define the size of the chessboard as a constant:

## Define a Data Structure to Hold the Attacked Positions

We need to **hold the attacked positions** in some data structure. At any moment during the execution of the program, we need to know **whether a certain** **position** **{row, col} is under attack** by a queen or not.

There are many ways to **store the attacked positions**:

* By keeping **all currently placed queens** and checking whether the new position conflicts with some of them.
* By keeping an int[][] **matrix of all attacked positions** and checking the new position directly in it. This will be complex to maintain because the matrix should change many positions after each queen placement/removal.
* By keeping **sets of all attacked rows, columns and diagonals**. Let’s try this idea:

The above definitions have the following assumptions:

* **The Rows** are 8, numbered from 0 to 7.
* **The Columns** are 8, numbered from 0 to 7.
* The **left diagonals** are 15, numbered from -7 to 7. We can use the following formula to calculate the left diagonal number by row and column: leftDiag = col - row.
* The **right diagonals** are 15, numbered from 0 to 14 by the formula: rightDiag = col + row.

Let’s take as an **example** the following chessboard with 8 queens placed on it at the following positions:

* {0, 0}; {1, 6}; {2, 4}; {3, 7}; {4, 1}; {5, 3}; {6, 5}; {7, 2}



Following the definitions above for our example the **queen {4, 1}** occupies the **row 4**, **column 1**, **left diagonal -3** and **right diagonal 5**.

## Write the Backtracking Algorithm

Now, it is time to write the recursive **backtracking algorithm** for placing the 8 queens.

The algorithm starts from row 0 and tries to place a queen at some column at row 0. On success, it tries to place the next queen at row 1, then the next queen at row 2, etc. until the last row is passed.

## Check if a Position is Free

Now, let’s write **the code to check whether a certain position is free**. A position is free when it is not under attack by any other queen. This means that if some of the rows, columns or diagonals is already occupied by another queen, the position is occupied. Otherwise it is free.

Recall that col-row is the number of the left diagonal and row+col is the number of the right diagonal.

## Mark / Unmark Attacked Positions

After a queen is placed, we need to **mark as occupied all rows, columns and diagonals** that it can attack.

On removal of a queen, we will need a method to mark as free all rows, columns and diagonals that were attacked by it.

## Print Solutions

When a solution is found, it should be printed at the console. First, introduce a solutions counter to simplify checking whether the found solutions are correct.

Next, pass through all rows and through all columns at each row and **print the chessboard cells**: