Lesson:

Backtracking

Pre Requisites:

- Basic Java
- Recursion in Java

List of concepts involved :

- Introduction to Backtracking
- Permutations of string
- N-Queen Interview Problem
- Sudoko Solver interview problem
- Rat in a maze interview problem

Introduction to Backtracking

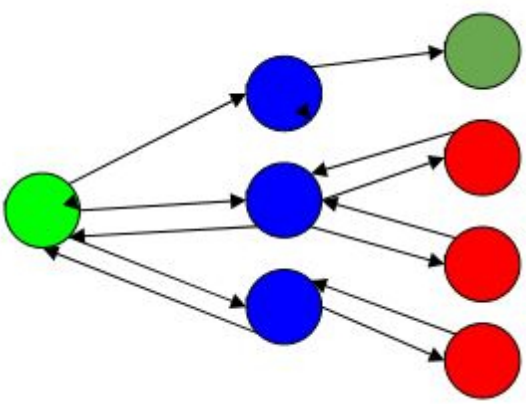
Return is a method based problem-solving algorithm. It uses recursive calls to find a solution, builds the solution step by step, and increments the value over time. Eliminate solutions that do not lead to a solution to the problem based on constraints set to solve the problem.

Backtracking algorithms apply to some specific types of problems.

Solution problems are used to find valid solutions to problems.

Optimization problem used to find the best acceptable solution.

In a backtracking problem, the algorithm tries to find a sequential path to a solution with a few small breakpoints where the problem can return if no acceptable solution can be found.



Example here,
green is the starting point, blue is the midpoint, red is the point for no valid solution, and dark green is the final solution.

Here, if the algorithm reaches the end and checks whether it is a solution or not, it returns a solution, otherwise it returns to a point one step later and finds a solution by finding a path to the next point.

Permutations of String

Finding a permutation of a string in Java means computing all possible new positions in a string by swapping characters. Also, in Java, the total number of permutations of a string is equal to the factorial of the length of a given string.

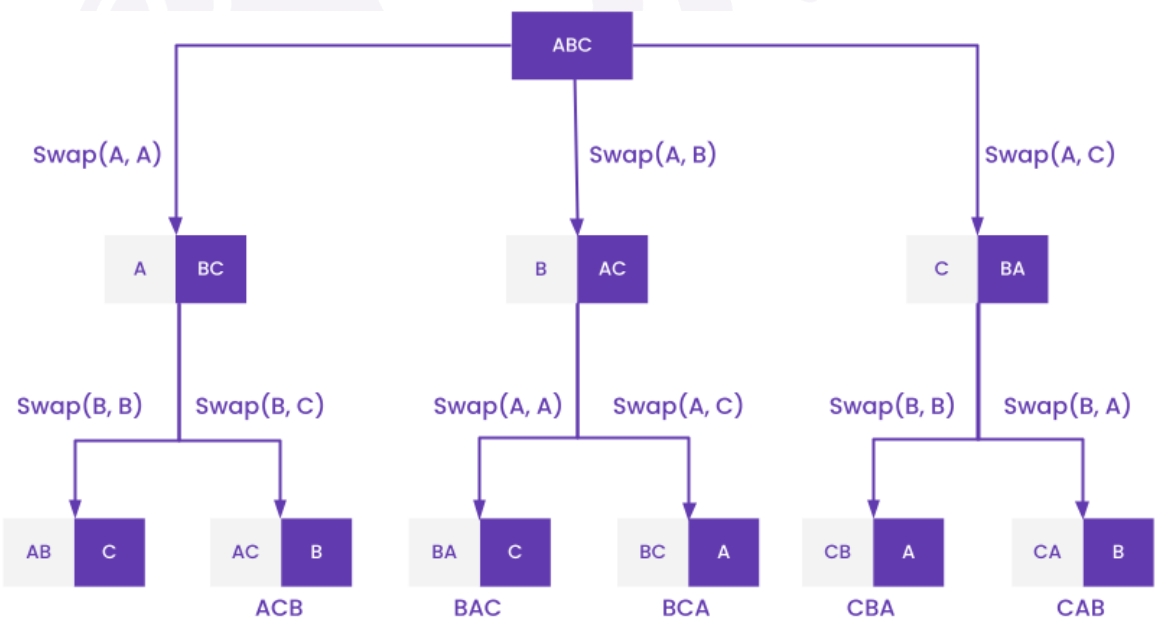
Example:
row XYZ has 3! that
6 permutation lines - [XYZ, XZY, YXZ, YZX, ZXY, ZYX]

Strings in Java are immutable and cannot be changed or modified, so a simple idea is to convert a string to an array of characters to generate permutations. You can use the backtracking concept by replacing each remaining character in the string with the first character and generating all permutations of the remaining characters with a recursive call.

This Java program uses a recursive or backtracking approach to print all possible string permutations.
The input string is converted to a character array using the toCharArray() function and passed to the resolver function. The simple idea is to replace the value in the character array with the index character element passed to keep track of the position of the string, then recursively call the solve function again to find the value (index+1).

The default condition fires when the currently passed index value equals (arr.length - 1), so prints the resulting array as one of the possible permutations of the string.

Replace back to return original value in for loop after recursive call to print all possible permutations. Since this is a return, the current character element can be used in any other possible string scheme.



Recursion Tree of String 'ABC'

Code: LP_CODE1.java

Output:

```
Enter string to generate its permutations: ABC
ABC ACB BAC BCA CBA CAB
```

Interview Problem : N-Queen

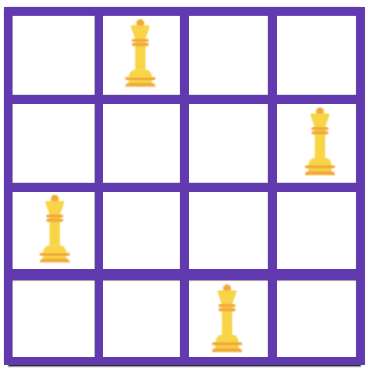
What is the N-Queens Problem?

How can N queens be placed on an NxN chessboard so that no two of them attack each other?

Let's took an example of N=4.

Movement of Queen in a chessboard: A Queen can move in any direction of board but while moving, it can't change the direction.
Therefore two queens shouldn't be in same row and column so that they can't attack each other.

When N=4, the solution looks like :



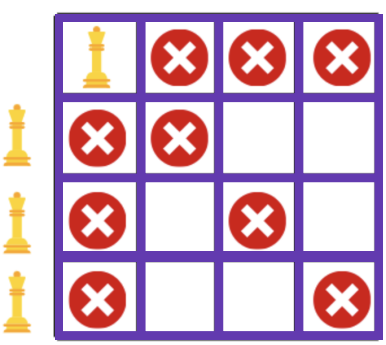
Solution to the N-Queen Problem

We're trying to fix this by putting the queen in place and removing the possibility of an attack on her. We place one queen in each row/column.

If you see that the queen is under attack at the selected location, try the next location.

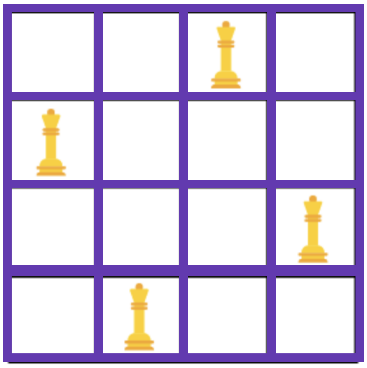
If the queen is under attack at every position in a row, go back and reposition the queen that was in front of the current position.

Repeat this process of placing and returning queens until all N queens have been successfully placed.



The red cross marks the positions which are under attack from a queen. Whenever we reach a state where we have a queen to place but all the positions in the rows are under attack, we backtrack.

This is not the only possible solution to the problem. If you move each queen one step forward in a clockwise manner, we get another solution.:



Code : LP_CODE2.java

Output.

```
0  0  1  0
1  0  0  0
0  0  0  1
0  1  0  0
```

Interview problem: Sudoku Solver

What is Sudoku

Simply put, Sudoku is a combinatorial number placement puzzle with a 9 x 9 grid of cells partially filled with numbers from 1 to 9. The goal is to fill the remaining empty fields with the remaining numbers so that there is only one. Numbers in each row and column. Quantity by type.

Also, each subsection of the 3 x 3 grid cannot have duplicate numbers.
Difficulty increases naturally with the number of empty fields on each board.

For example this is the unsolved Sudoku puzzle

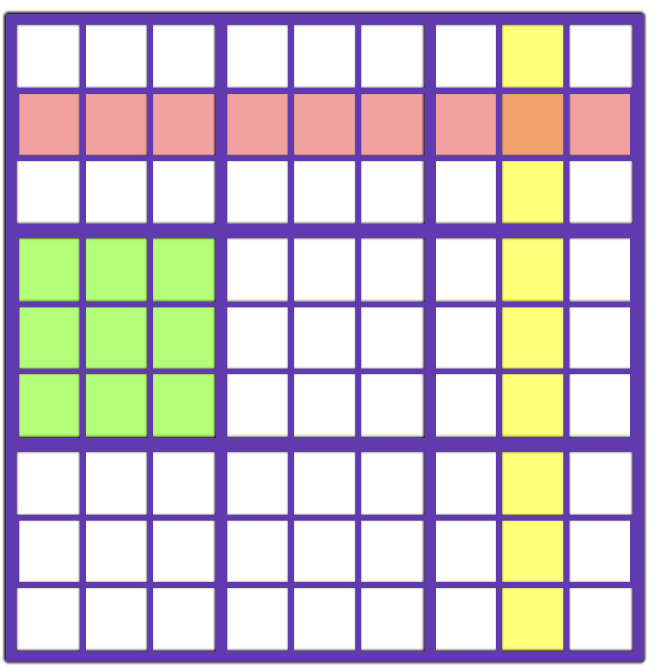
```
8 . . . . . . . .
. . 3 6 . . . . .
. 7 . . 9 . 2 . .
. 5 . . . 7 . . .
. . . . 4 5 7 . .
. . . 1 . . . 3 .
. . 1 . . . . 6 8
. . 8 5 . . . 1 .
. 9 . . . . 4 . .
```

And this is the solved version of it.
8 1 2 7 5 3 6 4 9
9 4 3 6 8 2 1 7 5
6 7 5 4 9 1 2 8 3
1 5 4 2 3 7 8 9 6
3 6 9 8 4 5 7 2 1
2 8 7 1 6 9 5 3 4
5 2 1 9 7 4 3 6 8
4 3 8 5 2 6 9 1 7
7 9 6 3 1 8 4 5 2

Approach to Solve Sudoku
Sudoku Solving Using Recursive Backtracking Algorithm
As with all other backtracking problems, you can solve Sudoku by sequentially assigning numbers to empty cells.

- Before assigning numbers, we need to check that the current row, current column, and current 3X3 subgrid do not have the same number.
- If there are no numbers in that row, column, or subgrid, you can assign a number and recursively check whether the result of the assignment is a solution. If the operation does not lead to a solution, the next number is tried for the currently empty cell. Returns false if none of the numbers (1 through 9) lead to a solution.
- When we assign a number to a location we need to perform multiple checks on it so that the solution is valid and doesn't break sudoku rules.
- **Row Check:** The following assigned number should not be present in the row
- **Column Check:** The following assigned number should not be present in the column.
- **3X3 box check:** We also need to perform a check the following assigned number shouldn't be present in the enclosing 3X3 box.

To keep the simplicity of code we will check all these constraints in a separate function isValid(). The function will return true, if all the above 3 checks are true else will return false.



Code Link: LP_CODE3.java

Output:

```
3 1 6 5 7 8 4 9 2
5 2 9 1 3 4 7 6 8
4 8 7 6 2 9 5 3 1
2 6 3 4 1 5 9 8 7
9 7 4 8 6 3 1 2 5
8 5 1 7 9 2 6 4 3
1 3 8 9 4 7 2 5 6
6 9 2 3 5 1 8 7 4
7 4 5 2 8 6 3 1 9
```

Time Complexity: O(9^(n*n))

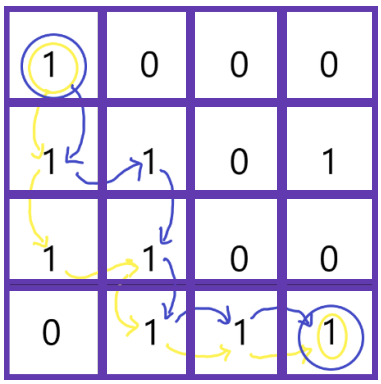
Space complexity : O(1)

Interview Problem: Rat in a Maze

Problem Statement: Consider the rat at (0, 0) in a square matrix of order N * N. The rat must reach its destination at (N - 1, N - 1). Find all possible paths for the rat to get from its source to its destination. The direction the mouse can move is "U" (up), "D" (down), "L" (left), and "R" (right). A matrix cell with a value of 0 means it is blocked and rats cannot pass through it, and a matrix cell with a value of 1 means that rats can pass through it.
Note: In a path, no cell can be visited more than one time.
Print the answer in lexicographical(sorted) order

Example 1:

Input:
N = 4
m[][] = {{1, 0, 0, 0},
{1, 1, 0, 1},
{1, 1, 0, 0},
{0, 1, 1, 1}}
Output: DDRDRR DRDDRR



The rat can reach the destination at (3, 3) from (0, 0) by two paths - DRDDRR and DDRDRR, when printed in sorted order we get DDRDRR DRDDRR.

Approach to Solve the problem
So we have a matrix. Next, find the path from the source cell (0,0) to the target cell(N-1, N-1) and do the following steps:

- Check the current cell. If this is the target cell, the puzzle is solved.
- If not, move down to see if it is possible to move in the downward direction (to move from a cell, the cell must not be free and in progress).
- We will continue the selected path down the next cell if we can move there.
- If not, let's move to the right cell if this is vacant and not already present in path.
- If not Move up when blocked or removed,
- Move to the cell on the left if it cannot move up.

If none of the four moves (down, right, up, or left) are possible, simply go back and change the current path (return).
Thus, the summary is that we try to move to the other cell (down, right, up, and left) from the current cell and if no movement is possible, then just come back and change the direction of the path to another cell.

Code: LP_CODE5.java

Output:

```
1       0       0       0       0
1       1       1       1       0
0       0       0       1       0
0       0       0       1       1
0       0       0       0       1
```

This is one of the valid paths for the above problem.

Next Class Teasers:

- Introduction to LinkedList
- Insertion and Deletion
- Reverse a linked list
- Cycle detection in a Linked List