Introduction to Backtracking

Backtracking is a famous algorithmic-technique for solving/resolving problems recursively by trying to build a solution incrementally. Solving one piece at a time, and removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to as the time elapsed till reaching any level of the search tree) is the process of backtracking.

In other words, **Backtracking** can be termed as a general algorithmic technique that considers searching **every possible combination** in order to solve a computational problem.

There are generally three types of problems in backtracking -

- **Decision Problems:** In these types of problems, we search for **any feasible** solution.
- **Optimization Problems:** In these types of problems, we search for the **best** possible solution.
- **Enumerations Problems:** In these types of problems, we find **all feasible** solutions.

Backtracking and recursion

Backtracking is based on recursion, where the algorithm makes an effort to build a solution to a computational problem incrementally. Whenever the algorithm needs to choose between multiple possible alternatives, it simply tries **all possible options** for the solution recursively step-by-step. The usual scenario is that you are faced with a number of options, and you must choose one of these. After you make your choice you will get a new set of options; just what set of options you get depends on what choice you made. This procedure is repeated over and over until you reach a final state. If you made a good sequence of choices, your final state is a ***goal state****;* if you didn't, it isn't.

In recursion, the function calls itself until it reaches a base case. In backtracking, we use recursion in order to explore all the possibilities until we get the desired solution for the problem. Backtracking works right after the recursive step, i.e if the recursive step results in a solution that is not desired, it retraces back and looks for other possible options.

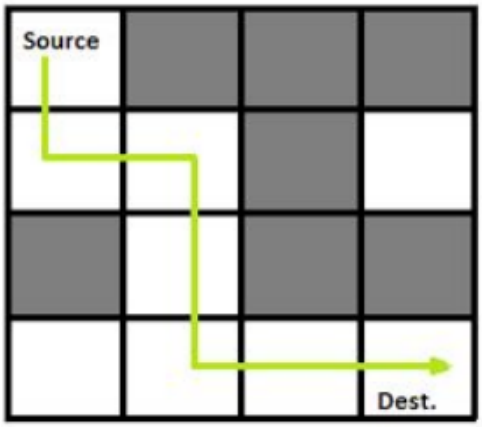
Implementation

- **Rat Maze problem**
  Given a 2D-grid, for simplicity let us assume that the grid is a **square** matrix of size **N**, having some cells as free and some as blocked. Given source **S** and destination **D**, we need to find whether there exists a path from source to destination in the maze, by traversing through **free cells** only.

  Let us assume that source S is at the **top-left corner** of the maze and destination D is at the **bottom-right corner** of the maze, and the movements allowed are to either move to the **right** or to **down**.

The **black-colored** cells represent the **blocked** cells and **white-colored** cells represent the **free** cells.

The possible path from **source** to **destination** is -



NOTE: There may be **multiple** possible paths from source to destination.

We will use backtracking to find whether there exists a path from source to destination in the given maze.

Steps:

- If the destination point i.e **N** is reached, return true.
- Else
    - Check if the current position is a **valid** position i.e the position is within the **bounds** of the maze as well as it is a **free** position.
    - Mark the current position as 1, denoting that the position can lead to a possible solution.
    - Move **forward**, and recursively check if this move leads to reach to the destination.
    - If the above move fails to reach the destination, then move **downward**, and recursively check if this move leads to reach the destination.
    - If none of the above moves leads to the destination, then mark the current position as 0, denoting that it is **not possible** to reach the destination from this position.

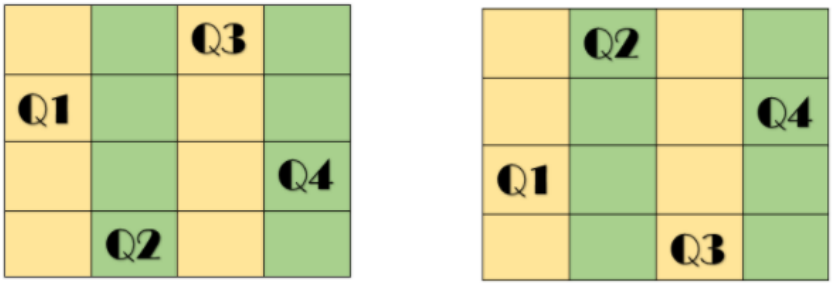
Pseudocode:

```
function isValid(x, y, N)

        /*
                Check if the position is within the bounds of the maze and the
                position does not contain a blocked cell.
        */

        if(x <= N and y <= N and x, y is not blocked)
                return true
        else
                return false

function RatMaze(maze[][], x, y, N)

        /*
                x, y is the current position of the rat in the maze.
                Check if the current position is a valid position.
        */

        if isValid(x, y, N)
                mark[x][y] = 1
        else
                return false

        /*
                If the current position is the bottom-right corner, i.e N, then we
                have found a solution
        */

        if x equals N and y equals N
                mark[x][y] = 1
                return true

        /*
                Otherwise, try moving forward or downward to look for other
                possible solutions.
        */

        bool found = RatMaze(maze,x+1,y,N) OR RatMaze(maze,x,y+1,N)

        /*
```

```
            If a solution is found from the current position by moving forward
            or downward, then return true, otherwise mark the current
            position as 0, as it is not possible to reach the end of the maze.
*/

if(found)
        return true
else
        mark[x][y] = 0
        return false
```

- **N-Queens Problem**

  Given an **NxN** chessboard, we need to arrange **N queens** on the board in such a way that **no two queens** attack each other. A queen can attack **horizontally, vertically,** or **diagonally**.
  Below is a diagram showing how **4 queens** can be placed on the chessboard of size **4x4**, such that no two queens attack each other.



  In the above diagram, the 4 queens are represented by **Q1**, **Q2**, **Q3**, **Q4** and it shows the possible **valid arrangements** of the 4 queens on the chessboard.
  NOTE: There can be **multiple** valid arrangements for the queens on the chessboard.
  We will use backtracking to find a valid arrangement of the queens on the given chessboard. We place the first queen arbitrarily anywhere within the board, and then place the next queen in a position that is not attacked by any other queens placed so far, if no such position is present we backtrack and change the position of the previous queens. A solution is found if we are able to place all the queens on the chessboard.
  Steps:
  - Place a queen **arbitrarily** at any position on the chessboard.
  - Check if this position is **safe**, i.e it is not attacked by any other queens.
  - If the position is not safe, then look for other positions on the board, and if no such position is found, then return false as we cannot place any more queens.
  - If the position is safe, then recursively check for Q-1 queens, if the function returns true, in other words, all queens were placed successfully on the board, then return true.

- **NOTE:** Q denoting the number of queens to be placed, N is passed as the value in the function call, representing the value of Q initially, as we need to place N queens on the board.

Pseudocode:

```
function isValid(x,y,board[][],N)

    /*
        Check if the position at x,y is not attacked by any other
        queen.
            1. Check if no position is marked 1 in the same row.
            2. Check if no position is marked 1 in the same column.
            3. Check if no position is marked 1 in the diagonals.
    */

    for i = 0 to N - 1
        if board[x][i] equals 1 or board[i][y] equals 1
            return false

    tempx = x
    tempy = y

    while tempx >= 0 and tempy < N
        if board[tempx][tempy] equals 1
            return false
        tempx -= 1
        tempy += 1

    tempx = x
    tempy = y

    while tempx < N and tempy >= 0
        if board[tempx][tempy] equals 1
            return false
        tempx += 1
        tempy -= 1

    tempx = x
    tempy = y

    while tempx >= 0 and tempy >= 0
        if board[tempx][tempy] equals 1
            return false
```

```
                tempx -= 1
                tempy -= 1

        tempx = x
        tempy = y

        while tempx < N and tempy < N
                if board[tempx][tempy] equals 1
                        return false
                tempx += 1
                tempy += 1


        //  The position is a safe position, return true
        return true

function N-Queens(Q, board[][], N)

        //  Q represents the number of queens to be placed on the board.

        //  Base Case, when all queens have been placed
        if Q equals 0
                return true

        /*
                For each possible position on the board, check if the position is
                safe i.e it is not attacked by any other queen placed so far.
        */

        for i = 0 to N - 1
                for j = 0 to N - 1
                        bool can = isValid(i, j, board, N)

                        /*
                                If the position is safe, then mark the position on the
                                board as 1, and check recursively for Q-1 queens if
                                they can be placed successfully.
                        */

                        if isValid is true
                                board[i][j] = 1
                                bool solve = N - Queens(Q - 1,board,N)
```

```
                        /*
                                The remaining queens can be placed
                                successfully, return true, otherwise, unmark
                                the position on the board, and check for
                                other possible options.
                        */

                        if solve is true
                                return true
                        else
                                board[i][j] = 0
                else
                        continue
        /*
                Since there was no possible option to place a queen on the board,
                so return false.
        */

        return false
```

Applications of backtracking

- Backtracking is useful in solving puzzles such as the **Eight queen puzzle, Crosswords, Verbal arithmetic, Sudoku,** and **Peg Solitaire.**
- The technique is also useful in solving combinatorial optimization problems such as parsing and the knapsack.
- The backtracking search algorithm is used in load frequency control of multi-area interconnected power systems.

Advantages of backtracking

- It is a step-by-step representation of a solution to a given problem, which is very easy to understand.
- It is easy to first develop an algorithm, and then convert it into a flowchart and into a computer program.
- It is very easy to implement and contains fewer lines of code, almost all of them being generally few lines of recursive function code.

Disadvantages of backtracking

- More optimal algorithms for the given problem may exist.
- Very time inefficient in a lot of cases when the branching factor is large.
- Can lead to large space complexities because of the recursive function call stack.