BACKTRACKING ALGORITHMS

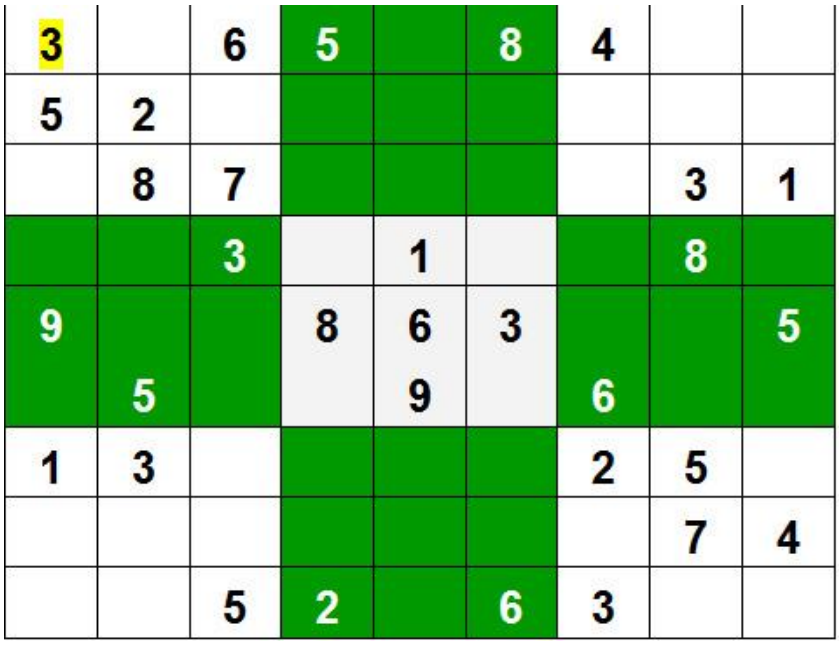
Definition

Backtracking is an algorithmic technique where the goal is to get all solutions to a problem using the brute force approach. It consists of building a set of all the solutions incrementally. Since a problem would have constraints, the solutions that fail to satisfy them will be removed.

The term backtracking suggests that if the current solution is not suitable, then backtrack and try other solutions. Thus, recursion is used in this approach.

This approach is used to solve problems that have multiple solutions.

For example, consider the Sudoku solving Problem, we try filling digits one by one. Whenever we find that the current digit cannot lead to a solution, we remove it (backtrack) and try the next digit. This is better than the naive approach (generating all possible combinations of digits and then trying every combination one by one) as it drops a set of permutations whenever it backtracks.



There are three types of problems in backtracking –

1. Decision Problem – In this, we search for a feasible solution. For example, puzzles such as eight queens puzzle, crosswords, verbal arithmetic, Sudoku, and Peg Solitaire.

2. Combinatorial Optimization Problem – In this, we search for the best solution, such as parsing and the knapsack problem.
3. Enumeration Problem – In this, we find all feasible solutions. For example, logic programming languages such as Icon, Planner and Prolog, which use backtracking internally to generate answers.

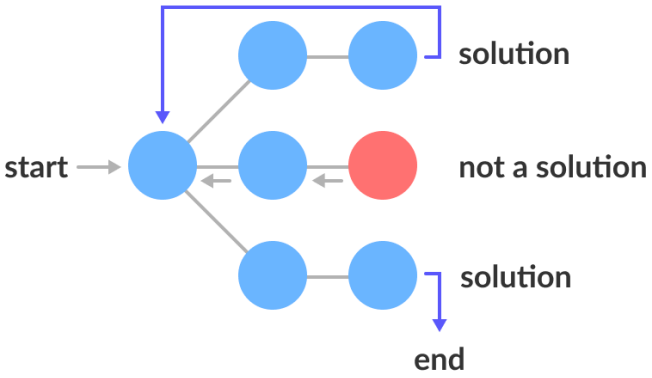
Explanation

Consider a situation that you have three boxes in front of you and only one of them has a gold coin in it but you do not know which one. So, to get the coin, you will have to open all of the boxes one by one. You will first check the first box, if it does not contain the coin, you will have to close it and check the second box and so on until you find the coin. This is what backtracking is, which is solving all sub-problems one by one to reach the best possible solution.

The backtracking algorithm uses recursive calling to find a solution set by building a solution step by step, increasing levels with time. To find these solutions, a search tree named state-space tree is used. In a state-space tree, each branch is a variable, and each level represents a solution.

State-Space Tree

A space state tree is a tree representing all the possible states (solution or nonsolution) of the problem from the root as an initial state to the leaf as a terminal state.



Example of state-space tree

Problem: You want to find all the possible ways of arranging 2 boys and 1 girl on 3 benches.

Constraint: Girl should not be on the middle bench.

Solution: There are a total of 3! = 6 possibilities. We will try all the possibilities and get the possible solutions. We recursively try all the possibilities.

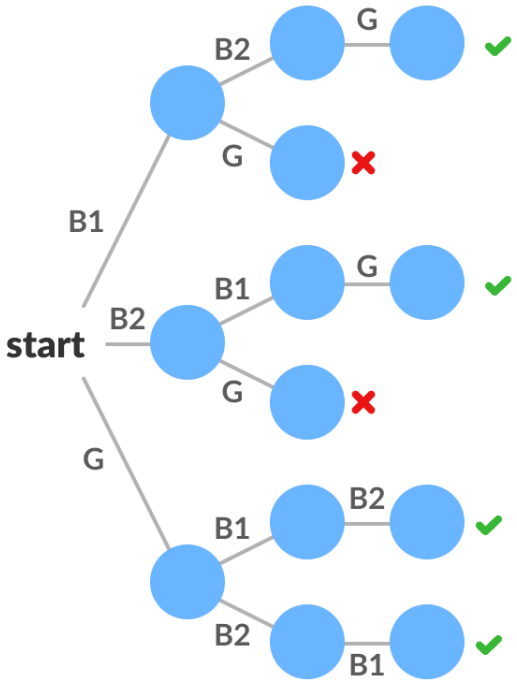
All the possibilities are:

(B represents boys and G represents girls)

B1	B2	G
B1	G	B2
B2	B1	G

B2	G	B1
G	B1	B2
G	B2	B1

The following state space tree shows the possible solutions:



Backtracking Algorithm

```
Backtrack(x)
   if x is not a solution
      return false
   if x is a new solution
      add to list of solutions
   backtrack(expand x)
```

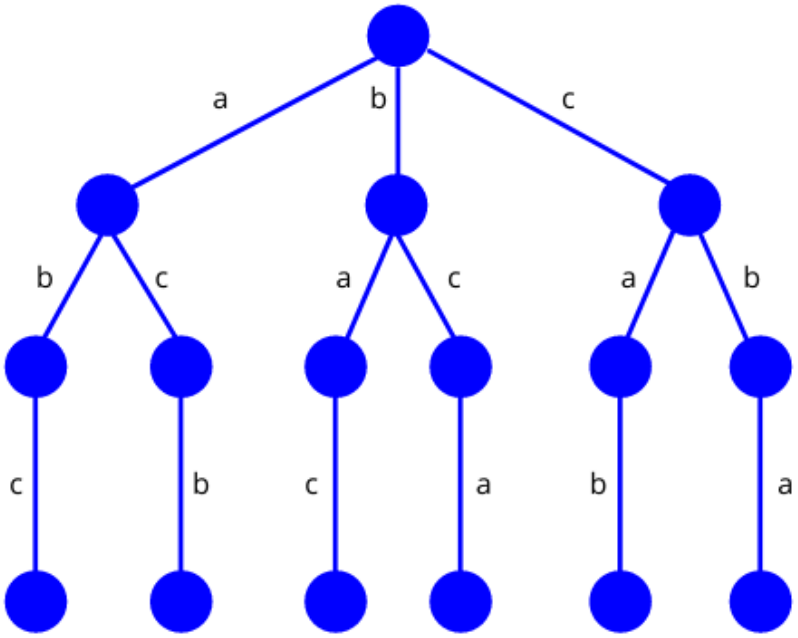
A backtracking algorithm uses the depth-first search method. When it starts exploring the solutions, a bounding-function is applied so that the algorithm can check if the so-far built solution satisfies the constraints. If it does, it

continues searching. If it doesn't, the branch would be eliminated, and the algorithm goes back to the level before.

An Example

We're taking a very simple example here to explain the theory behind a backtracking process. We want to arrange the three letters *a, b, c* in such *a* way that *c* cannot be beside *a*.

According to the backtracking, first, we'll build a state-space tree. We'll find all the possible solutions and check them with the given constraint. We'll only keep those solutions that satisfy the given constraint:



The possible solutions of the problems would be: *(a,b,c), (a,c,b), (b,a,c), (b,c,a), (c,a,b), (c,b,a)*.

Nevertheless, the valid solutions to this problem would be the ones that satisfy the constraint, which keeps only ***(a,b,c)*** and ***(c,b,a)*** in the final solution set.

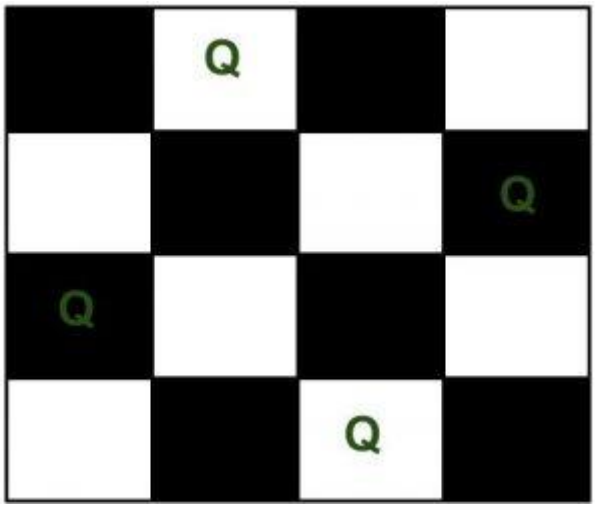
Examples

N-Queens Problem

A classic example of backtracking is the *n*-Queens problem, first proposed by German chess enthusiast Max Bezzel in 1848. Given a chessboard of size *n*, the problem is to place *n* queens on the *n X n* chessboard, so no two queens are attacking each other.

It is clear that for this problem, **we need to find all the arrangements of the positions of the *n* queens on the chessboard, but there is a constraint: no queen should be able to attack another queen.**

For example, the following is a solution for the 4 Queen problem.



Input:
(The size of the chessboard: *n*)

n = 4

Output:
(The expected output is a binary matrix that has 1s for the blocks where queens are placed.)

{0, 1, 0, 0}

{0, 0, 0, 1}

{1, 0, 0, 0}

{0, 0, 1, 0}

Solution:

A **brute-force solution** would be to generate all possible configurations of queens on board and print a configuration that satisfies the given constraints.

The **backtracking solution** is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the

solution. If we do not find such a row due to clashes then we backtrack and return false.

Pseudocode:

Algorithm 1: Backtracking Algorithm for solving the N-Queens Problem

```
Data: Q[n]: an array contains the positions of n queens; k: index
      of the first empty row
Result: All the possible placement of n non-attacking queens on a
        chessboard
Procedure NQueen(Q[n], k)
    if k == n + 1 then
        return Q;
    end
    for j = 1 to n do
        valid = True;
        for i = 1 to k-1 do
            if (Q[i]=j) or (Q[i]=j+k-i) or (Q[i]=j-k+i) then
                valid = False;
            end
            if valid = True then
                Q[k] = j;
                NQueen(Q[n], k+1);
            end
        end
    end
end
```

Implementation:

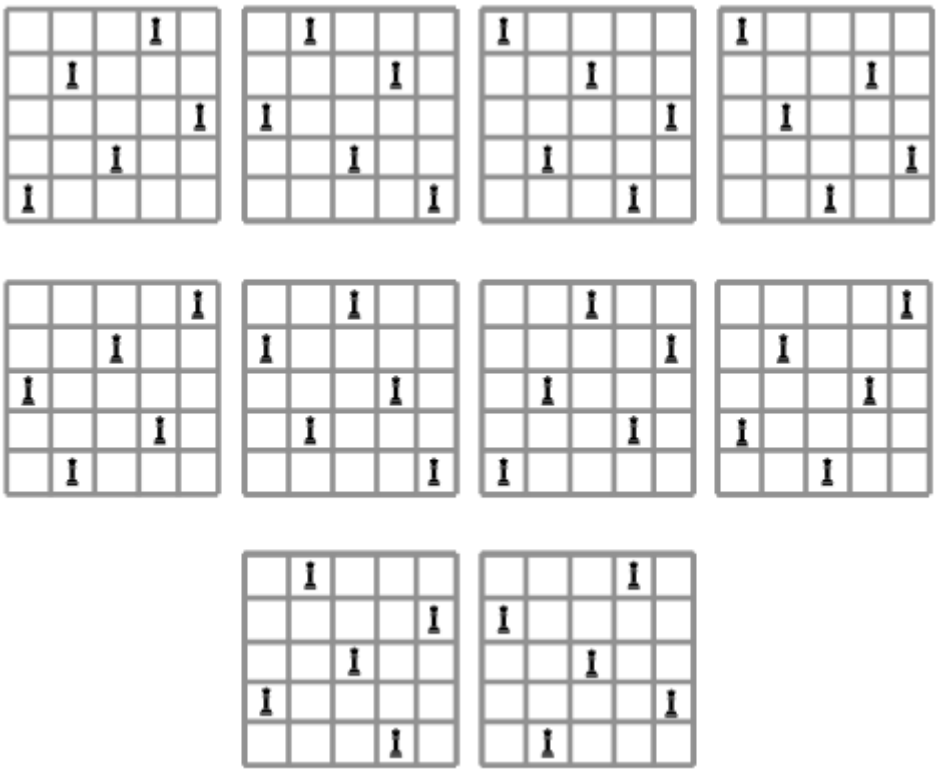
```
public class NQueenProblem {
   final int N = 4;

   /* A utility function to print solution */
   void printSolution(int board[][])
   {
      for (int i = 0; i < N; i++) {
         for (int j = 0; j < N; j++)
            System.out.print(" " + board[i][j]
                             + " ");
         System.out.println();
      }
   }

   /* A utility function to check if a queen can
     be placed on board[row][col]. Note that this
```

```
  function is called when "col" queens are already
  placed in columns from 0 to col -1. So we need
  to check only the left side for attacking queens */
boolean isSafe(int board[][], int row, int col)
{
   int i, j;

   /* Check this row on left side */
   for (i = 0; i < col; i++)
      if (board[row][i] == 1)
         return false;

   /* Check upper diagonal on left side */
   for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
      if (board[i][j] == 1)
         return false;

   /* Check lower diagonal on left side */
   for (i = row, j = col; j >= 0 && i < N; i++, j--)
      if (board[i][j] == 1)
         return false;

   return true;
}

/* A recursive utility function to solve N
  Queen problem */
boolean solveNQUtil(int board[][], int col)
{
   /* base case: If all queens are placed
     then return true */
   if (col >= N)
      return true;

   /* Consider this column and try placing
     this queen in all rows one by one */
   for (int i = 0; i < N; i++) {
      /* Check if the queen can be placed on
        board[i][col] */
      if (isSafe(board, i, col)) {
```

```
            /* Place this queen in board[i][col] */
            board[i][col] = 1;

            /* recur to place rest of the queens */
            if (solveNQUtil(board, col + 1) == true)
               return true;

            /* If placing queen in board[i][col]
              doesn't lead to a solution then
              remove queen from board[i][col] */
            board[i][col] = 0; // BACKTRACK
         }
      }

      /* If the queen can not be placed in any row in
        this column col, then return false */
      return false;
   }

   /* This function solves the N Queen problem using
     Backtracking.  It mainly uses solveNQUtil () to
     solve the problem. It returns false if queens
     cannot be placed, otherwise, return true and
     prints placement of queens in the form of 1s.
     Please note that there may be more than one
     solution, this function prints one of the
     feasible solutions.*/
   boolean solveNQ()
   {
      int board[][] = { { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 } };

      if (solveNQUtil(board, 0) == false) {
         System.out.print("Solution does not exist");
         return false;
      }

      printSolution(board);
```

```
        return true;
    }

    // driver program to test above function
    public static void main(String args[])
    {
        NQueenProblem Queen = new NQueenProblem();
        Queen.solveNQ();
    }
}
```

If we take a *5 X 5* chessboard as an example, solving the problem results in 10 solutions, which leads us to use the backtracking algorithm to retrieve all these solutions:

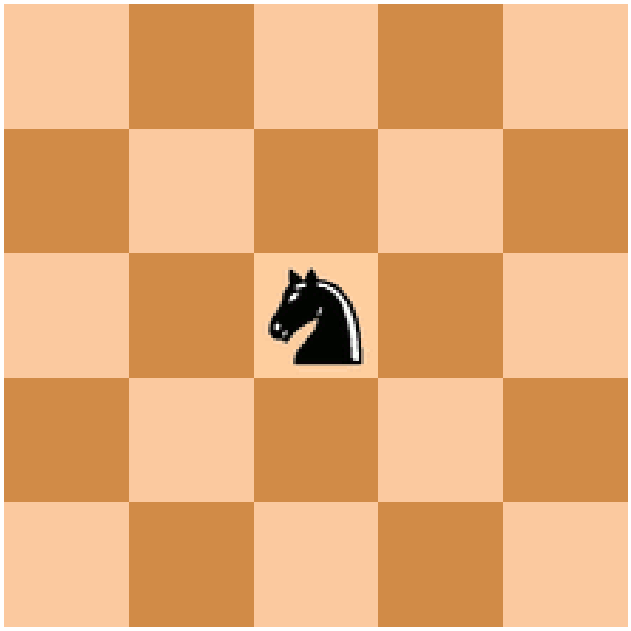


It is true that for this problem, the solutions found are valid, but still, a backtracking algorithm for the n-Queens problem presents a time complexity equal to **O(2^n)**.

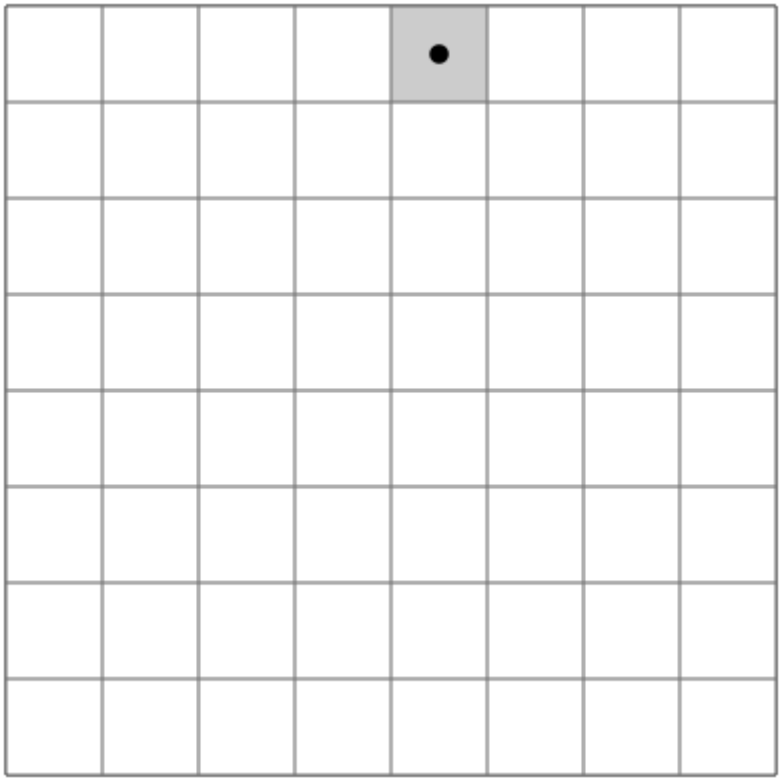
A Knight's Tour Problem

The "knight's tour" is a classic problem in graph theory, first posed over 1,000 years ago and pondered by legendary mathematicians including Leonhard Euler before finally being solved in 1823.

The knight's tour puzzle is played on a chessboard with a single chess piece, the knight. The object of the puzzle is to find a sequence of moves that allow the knight to visit every square on the board exactly once, like so:



One such sequence is called a "tour." The upper bound on the number of possible legal tours for an eight-by-eight chessboard is known to be 1.305×10^{35}; however, there are even more possible dead ends. Clearly, this is a problem that requires some real brains, some real computing power, or both.



Problem Statement:

Given a *N X N* board with the Knight placed on the first block of an empty board. Moving according to the rules of chess knight must visit each square exactly once. Print the order of each cell in which they are visited.

Examples:

Input: N = 8

Output: The path followed by Knight to cover all the cells

0 59 38 33 30 17 8 63

37 34 31 60 9 62 29 16

58 1 36 39 32 27 18 7

35 48 41 26 61 10 15 28

42 57 2 49 40 23 6 19

47 50 45 54 25 20 11 14

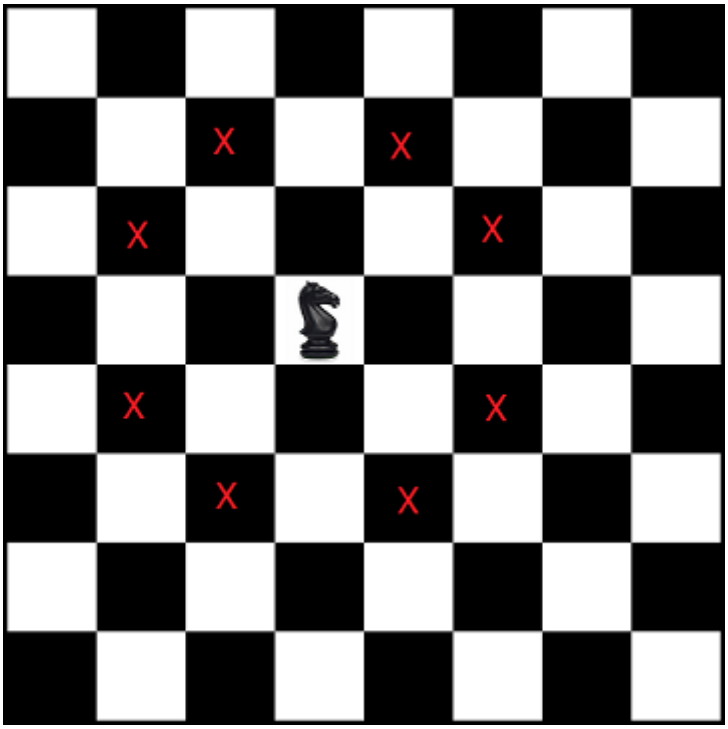
56 43 52 3 22 13 24 5

51 46 55 44 53 4 21 12

Algorithm:

The knight should search for a path from the starting position until it visits every square or exhausts all possibilities. We can easily achieve this with the help of backtracking. We start from the given source square in the chessboard and recursively explore all eight paths possible to check if they lead to the solution or not. If the current path doesn't reach the destination or explore all possible routes from the current square, backtrack. To make sure that the path is simple and doesn't contain any cycles, keep track of squares involved in the current path in a chessboard, and before exploring any square, ignore the square if it is already covered in the current path.

We know that a knight can move in 8 possible directions from a given square, as illustrated in the following figure:



We can find all the possible locations the knight can move to from the given location by using the array that stores the knight movement's relative position from any location. For example,

```
If the current location is (x, y), we can move to position (x + row[k],
y + col[k]) for 0 <= k <=7 using the following arrays:

row[] = [ 2, 1, -1, -2, -2, -1, 1, 2, 2 ]
col[] = [ 1, 2, 2, 1, -1, -2, -2, -1, 1 ]

So, from a position (x, y) in the chessboard, the valid moves are:

(x + 2, y + 1)
(x + 1, y + 2)
(x – 1, y + 2)
(x – 2, y + 1)
(x – 2, y – 1)
(x – 1, y – 2)
(x + 1, y – 2)
(x + 2, y – 1)
```

Important Note: Please avoid changing the sequence of the above arrays. The order in which the knight will move is circular and will be optimum. Using the above order, we will get to a vacant position in a few moves. Also, it is always better to start backtracking from any corner of the chessboard. If we start from somewhere middle, the knight can go in 8 different directions. If we start from

the corner, the knight can go to only two points from there. Since the algorithm is exponential, optimized input to it can make a huge difference.

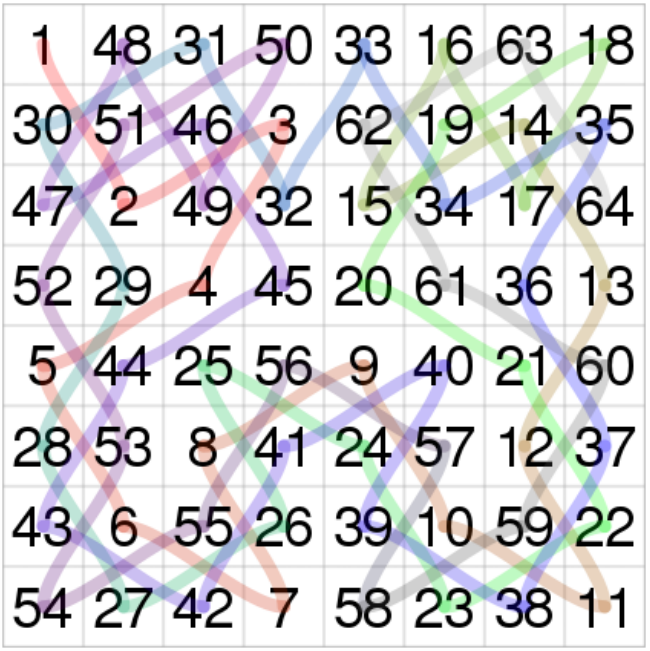
Implementation:

```
import java.util.Arrays;
class KnightsTour
{
   // `N × N` chessboard
   public static final int N = 8;

   // Below arrays detail all eight possible movements for a knight.
   // Don't change the sequence of the below arrays
   public static final int[] row = { 2, 1, -1, -2, -2, -1, 1, 2, 2 };
   public static final int[] col = { 1, 2, 2, 1, -1, -2, -2, -1, 1 };

   // Check if `(x, y)` is valid chessboard coordinates.
   // Note that a knight cannot go out of the chessboard
   private static boolean isValid(int x, int y)
   {
      if (x < 0 || y < 0 || x >= N || y >= N) {
         return false;
      }

      return true;
   }

   private static void print(int[][] visited)
   {
      for (var r: visited) {
         System.out.println(Arrays.toString(r));
      }
      System.out.println();
   }

   // Recursive function to perform the knight's tour using backtracking
   public static void knightTour(int[][] visited, int x, int y, int pos)
   {
      // mark the current square as visited
```

```
    visited[x][y] = pos;

    // if all squares are visited, print the solution
    if (pos >= N*N)
    {
        print(visited);
        // backtrack before returning
        visited[x][y] = 0;
        return;
    }

    // check for all eight possible movements for a knight
    // and recur for each valid movement
    for (int k = 0; k < 8; k++)
    {
        // get the new position of the knight from the current
        // position on the chessboard
        int newX = x + row[k];
        int newY = y + col[k];

        // if the new position is valid and not visited yet
        if (isValid(newX, newY) && visited[newX][newY] == 0) {
            knightTour(visited, newX, newY, pos + 1);
        }
    }

    // backtrack from the current square and remove it from the current path
    visited[x][y] = 0;
}

public static void main(String[] args)
{
    // `visited[][]` serves two purposes:
    // 1. It keeps track of squares involved in the knight's tour.
    // 2. It stores the order in which the squares are visited.
    int[][] visited = new int[N][N];
    int pos = 1;
```

```
        // start knight tour from corner square `(0, 0)`
        knightTour(visited, 0, 0, pos);
    }
}
```

Output:



1	48	31	50	33	16	63	18
30	51	46	3	62	19	14	35
47	2	49	32	15	34	17	64
52	29	4	45	20	61	36	13
5	44	25	56	9	40	21	60
28	53	8	41	24	57	12	37
43	6	55	26	39	10	59	22
54	27	42	7	58	23	38	11

The problem with this backtracking method is its time complexity. The time complexity of the above backtracking solution is exponential. There are N^2 Cells and for each, we have a maximum of 8 possible moves to choose from, so the worst running time is $O(8^{N^2})$, which is impractical on larger boards. Each time a specific path doesn't work we're backtracking, which means the number of steps keeps going up with each wrong move.

This shouldn't be a problem for an 8 x 8 or a 64 x 64 chessboard but make it a 1,000,000 x 1,000,000 dataset and the computer might not be very happy with us. For larger N values, it is well beyond modern computers' capacity (or networks of computers) to perform operations on such a large set.

Problem Statements

Rat in a Maze

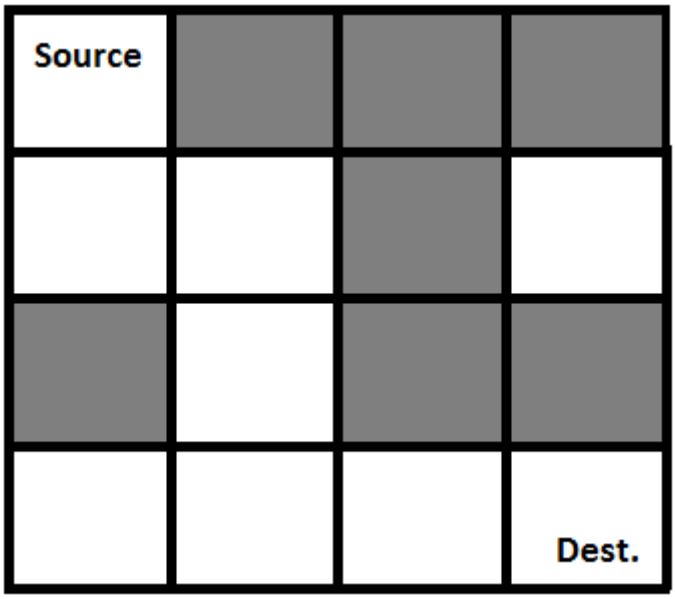
Explanation:

A Maze is given as N*N binary matrix of blocks where source block is the upper left most block i.e., maze[0][0], and destination block is lower rightmost block i.e., maze[N-1][N-1]. A rat starts from the source and has to reach its destination. The rat can move only in two directions: forward and down.

In the maze matrix, 0 means the block is a dead end, and 1 means the block can be used in the path from source to destination. Note that this is a simple version of the typical Maze problem. For example, a more complex version can be that the rat can move in 4 directions and a more complex version can be with a limited number of moves.

Example:

Following is an example maze.:

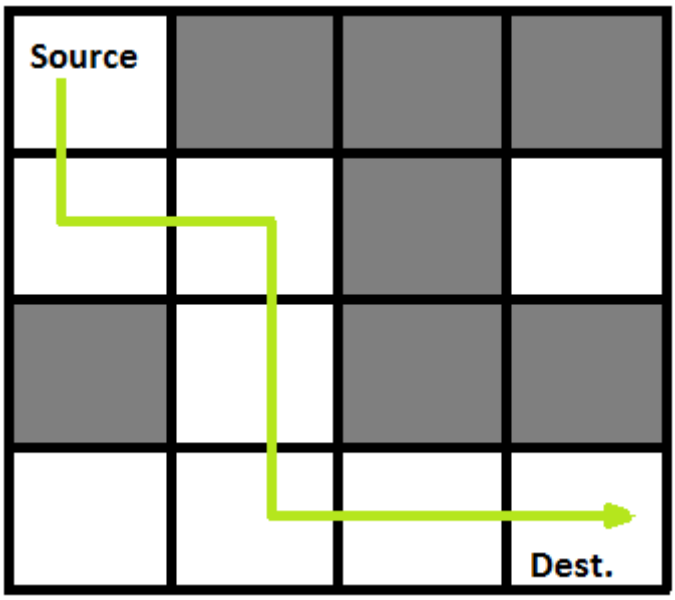


Grey blocks are dead ends (value = 0)

Following is a binary matrix representation of the above maze:

{1, 0, 0, 0}
{1, 1, 0, 1}
{0, 1, 0, 0}
{1, 1, 1, 1}

Following is a maze with highlighted solution path:



Following is the expected solution matrix (output of program) for the above input matrix:

{1, 0, 0, 0}
{1, 1, 0, 0}
{0, 1, 0, 0}
{0, 1, 1, 1}

Subset Sum

Explanation:

Subset sum problem is to find a subset of elements that are selected from a given set whose sum adds up to a given number K. Subset sum problem is the problem of finding a subset such that the sum of elements equals a given number. We are considering the set contains non-negative values. It is assumed that the input set is unique (no duplicates are presented).

Example:

Given the following set of positive numbers:

{2, 9, 10, 1, 99, 3}

We need to find if there is a subset for a given sum say 4:

{1, 3 }

For another value say 5, there is another subset:

{2, 3}

Similarly, for 6, we have {2, 1, 3} as the subset.

For 7, there is no subset where the sum of elements is equal to 7.

Find all paths from the first cell to the last cell of a matrix

Explanation:

Given an M × N integer matrix, find all paths from the first cell to the last cell. We can only move down or to the right from the current cell.

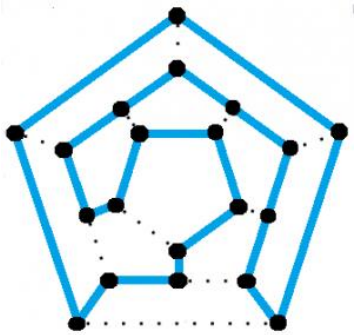
Example:

```
Input:

[ 1  2  3 ]
[ 4  5  6 ]
[ 7  8  9 ]

Output:

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

Hamiltonian Cycle

Explanation:

Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in the graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains the Hamiltonian Cycle or not. If it contains, then prints the path.



The input to the program is a 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.

The output of the program should be an array path[V] that should contain the Hamiltonian Path. path[i] should represent the ith vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.

Example:

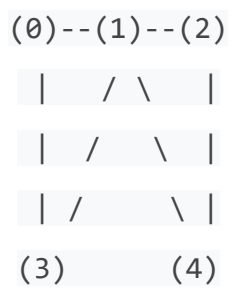
Input:


```
{{0, 1, 0, 1, 0},
 {1, 0, 1, 1, 1},
 {0, 1, 0, 0, 1},
 {1, 1, 0, 0, 1},
 {0, 1, 1, 1, 0}}
```

Output: Solution Exists: Following is one Hamiltonian Cycle

```
0  1  2  4  3  0
```

Input:




```
{{0, 1, 0, 1, 0},
```

```
{1, 0, 1, 1, 1},
{0, 1, 0, 0, 1},
{1, 1, 0, 0, 0},
{0, 1, 1, 0, 0}}
```

Output: `Solution does not exist`

Assessment

1) Backtracking algorithm is implemented by constructing a tree of choices called as?
   a) State-space tree
   b) State-chart tree
   c) Node tree
   d) Backtracking tree

2) What happens when the backtracking algorithm reaches a complete solution?
   a) It backtracks to the root
   b) It continues searching for other possible solutions
   c) It traverses from a different route
   d) Recursively traverses through the same route

3) In what manner is a state-space tree for a backtracking algorithm constructed?
   a) Depth-first search
   b) Breadth-first search
   c) Twice around the tree
   d) Nearest neighbour first

4) Which one of the following is an application of the backtracking algorithm?
   a) Finding the shortest path
   b) Finding the efficient quantity to shop
   c) Ludo
   d) Crossword

5) Backtracking algorithm is faster than the brute force technique
   a) true
   b) false

6) The problem of finding a subset of positive integers whose sum is equal to a given positive integer is called as?
   a) n- queen problem
   b) subset sum problem

c) knapsack problem
d) hamiltonian circuit problem

7) The problem of placing n queens in a chessboard such that no two queens attack each other is called as?
a) n-queen problem
b) eight queens puzzle
c) four queens puzzle
d) 1-queen problem

8) How many directions do queens attack each other?
a) 1
b) 2
c) 3
d) 4

9) How many solutions are there for 8 queens on 8*8 board?
a) 12
b) 91
c) 92
d) 93

10) How many fundamental solutions are there for the eight queen puzzle?
a) 92
b) 10
c) 11
d) 12