Backtracking Algorithm

Backtracking can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.

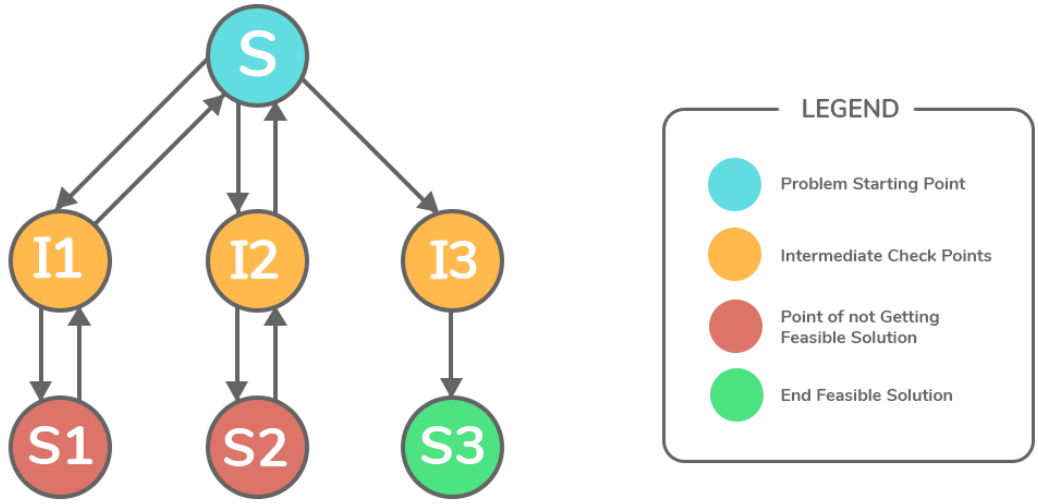
Backtracking is an algorithmic technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any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

How Backtracking Algorithms Works?

Backtracking is the technique to solve programs recursively. In backtracking problems, you will have a number of options and you must choose one of these options.

After you make your choice you will explore a new set of options, if you reach the feasible solution through those choices you will print the solution otherwise you will backtrack and return to explore the other paths and choices that can possibly lead to the solution.

For Instance: Consider the following space state tree. With the help of the below space state tree let's understand how the backtracking algorithm actually works.

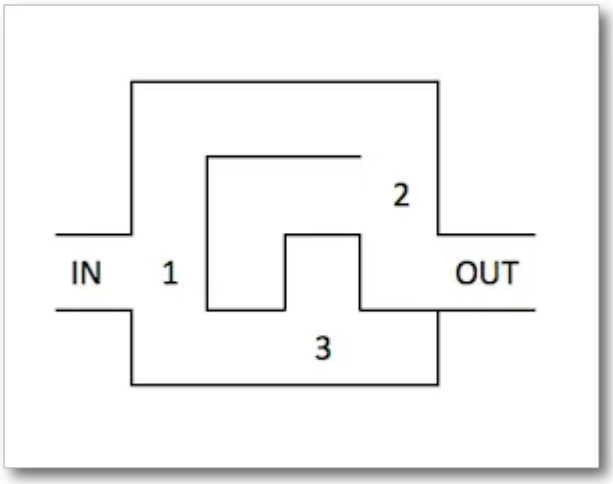


So we can summarise the backtracking algorithm as follows:

1. We select one possible path and try to move forward towards finding a feasible solution.
2. If we will reach a point from where we can't move towards the solution then we will backtrack.
3. Again we try to find out the feasible solution by following other possible paths.

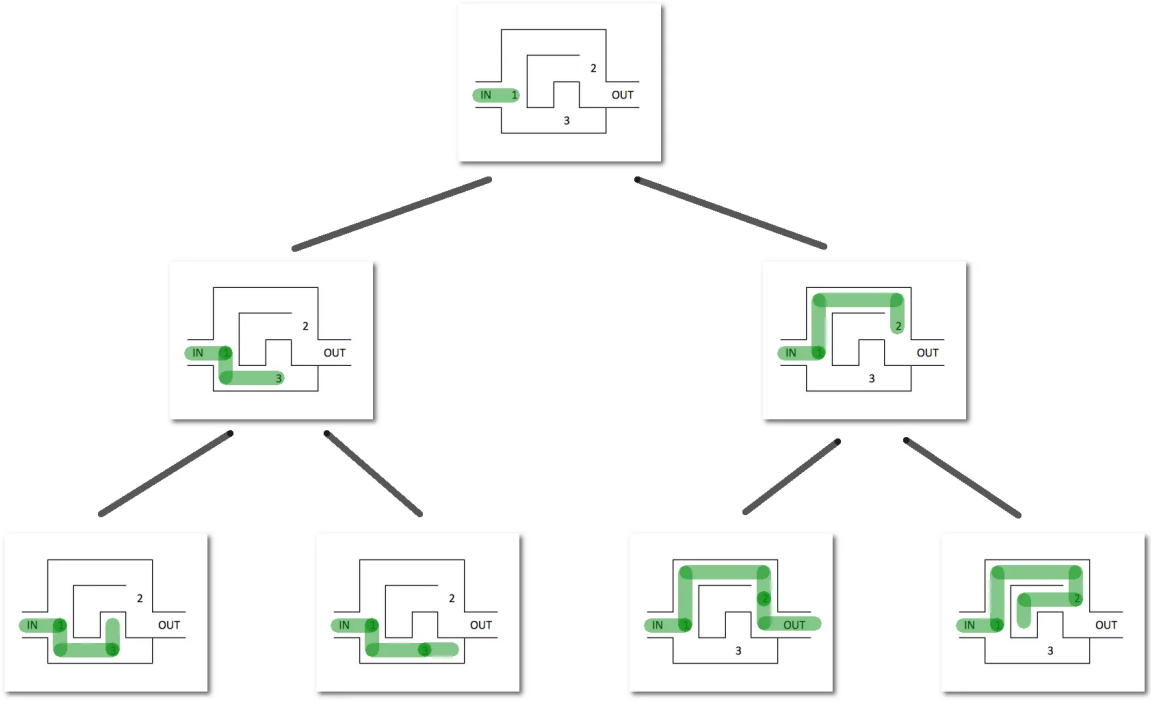
Algorithm

```
Step 1 – if current_position is goal, return success
Step 2 – else,
Step 3 – if current_position is an end point, return failed.
Step 4 – else, if current_position is not end point, explore and repeat above steps.



Backtrak(n)
            if n is not the solution
                        return false
            if n is new solution
                        add the list of soliution to the list

backtrack(expdn n)
```



where we have labeled the junctions as **1, 2** and **3**.

If we want to check every possible path in the maze, we can have a look at the tree of paths, split for every junctions stop:



Pseudo Code

```
function backtrack(junction):

  if is_exit:
    return true  for each direction of junction:
    if backtrack(next_junction):
      return true
```

```
  return false
```

```
- at junction 1 chooses down       (possible values: [down, up])
    - at junction 3 chooses right  (possible values: [right, up])
        no junctions/exit          (return false)
    - at junction 3 chooses up     (possible values: [right, up])
        no junctions/exit          (return false)
- at junction 1 chooses up         (possible values: [down, up])
    - at junction 2 chooses down   (possible values: [down, left])
        the exit was found!        (return true)
```

Please note that every time a line is indented, it means that there was a recursive call. So, when a `no junctions/exit` is found, the function returns a `false` value and goes back to the caller, that resumes to loop on the possible paths starting from the junction. If the loop arrives to the end, that
means that from that junction on there's no exit, and so it returns `false`.

The idea is that we can build a solution step by step using recursion; if during the process we realise that is not going to be a valid solution, then we stop computing that solution and we return back to the step before (*backtrack*).
In the case of the maze, when we are in a dead-end we are forced to backtrack, but there are other cases in which we could realise that we're heading to a non valid (or not good) solution before having reached it. And that's exactly what we're going to see now.

When to Use a Backtracking Algorithm?

Backtracking algorithms are ideally used when solving problems that involve making a sequence of choices, with each choice leading towards a potential solution. They are best suited for problems where the solution space is large and a brute-force approach is computationally impractical.

Backtracking is normally used when we are faced with a multiple number of options and we have to choose one among them based on the constraints given. After the choice we will be having a new set of options which is where the recursion comes to the rescue. The procedure is repeated untill we get a feasible solution.

1.puzzles like Sudoku.

2. N-Queens problem.

3.decision-making problems(finding shortest paths, subsets, or combinations).

4.Graph coloring.

5.Hamilton cycle.

Backtracking may not be optimal for problems with smaller decision spaces

Types of Backtracking Problems

1. **Decision Problem:** In this type of problem we always search for a feasible solution.
2. **Optimization Problem:** In this type of problem we always search for the best possible solution.
3. **Enumeration Problem:** In this type of problem we try to find all feasible solutions.

Difference between the Backtracking and Recursion

Recursion is a technique that calls the same function again and again until you reach the base case. Backtracking is an algorithm that finds all the possible solutions and selects the desired solution from the given set of solutions.

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

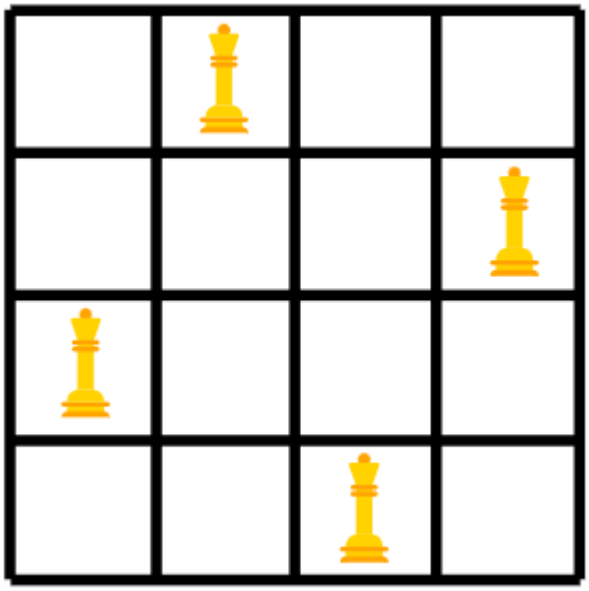
This problem is commonly seen for N=4 and N=8.

Let's look at an example where **N=4**

A queen can move any number of steps in any direction. The only constraint is that it can't change its direction while it's moving.

One thing that is clear by looking at the queen's movement is that no two queens can be in the same row or column.

That allows us to place only one queen in each row and each column.



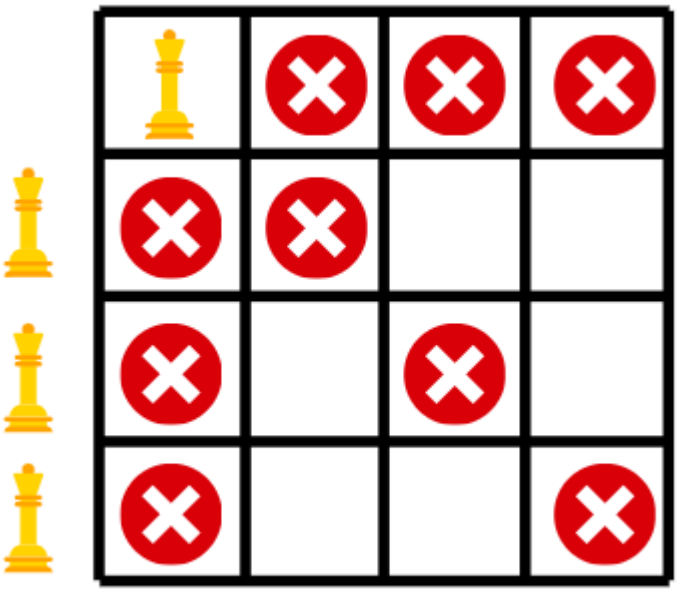
Solution to the N-Queens Problem

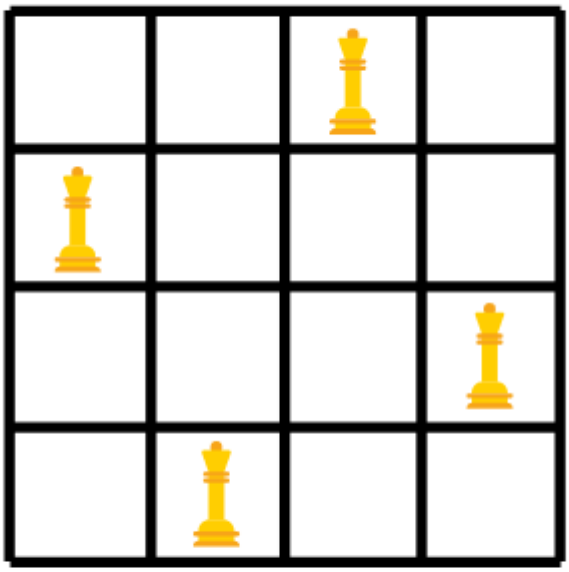
The way we try to solve this is by placing a queen at a position and trying to rule out the possibility of it being under attack. We place one queen in each row/column.

If we see that the queen is under attack at its chosen position, we try the next position.

If a queen is under attack at all the positions in a row, we backtrack and change the position of the queen placed prior to the current position.

We repeat this process of placing a queen and backtracking until all the N queens are placed successfully.





```
package Backtracking;
public class NQueen {
  static final int N = 4;

  static 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();
    }
  }

  // function to check whether the position is safe or not
  static boolean isSafe(int board[][], int row, int col) {
    int i, j;
    for (i = 0; i < col; i++)
      if (board[row][i] == 1)
        return false;

    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
      if (board[i][j] == 1)
        return false;

    for (i = row, j = col; j >= 0 && i < N; i++, j--)
      if (board[i][j] == 1)
        return false;

    return true;
  }

  // The function that solves the problem using backtracking
  public static boolean solveNQueen(int board[][], int col) {
    if (col >= N)
      return true;

    for (int i = 0; i < N; i++) {
      // if it is safe to place the queen at position i,col -> place it
      if (isSafe(board, i, col)) {
        board[i][col] = 1;

        if (solveNQueen(board, col + 1))
          return true;

        // backtrack if the above condition is false
        board[i][col] = 0;
      }
    }
    return false;
  }

  public static void main(String args[]) {
    int board[][] = {
        { 0, 0, 0, 0 },
        { 0, 0, 0, 0 },
        { 0, 0, 0, 0 },
        { 0, 0, 0, 0 } };
```

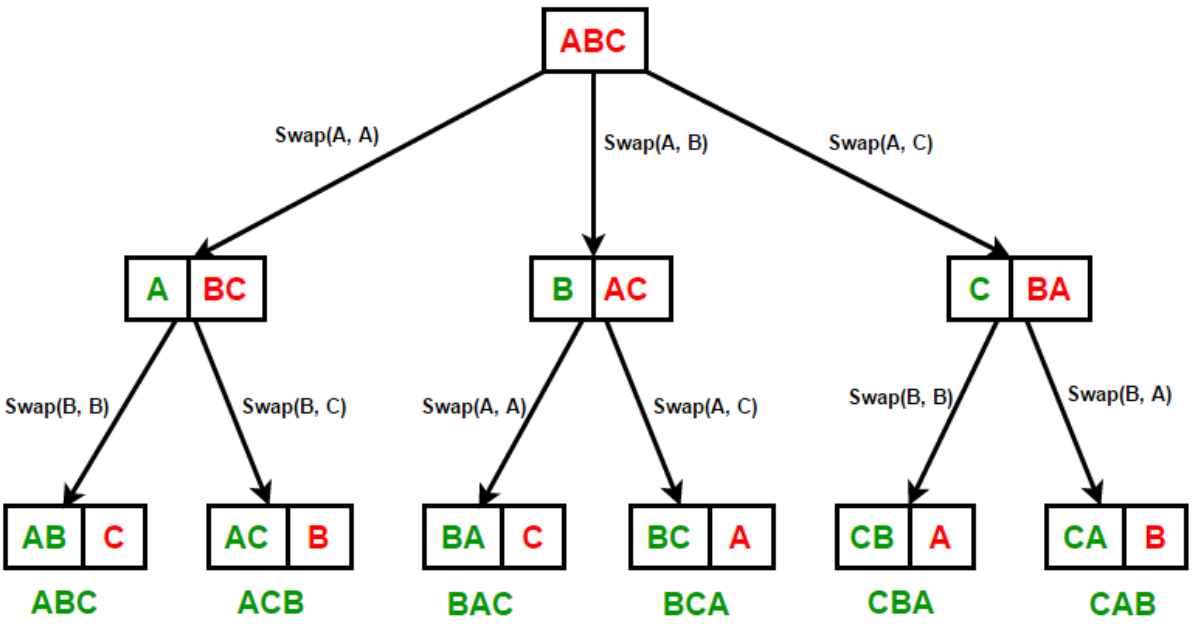
```
    if (!solveNQueen(board, 0)) {
      System.out.print("Solution does not exist");
      return;
    }

    printSolution(board);

  }
}
```

Write a Java program to generate all permutations of a string.

`ABC` has 6 permutations, i.e., `ABC, ACB, BAC, BCA, CBA, CAB`.

SOL:

Since the string is immutable in Java, the idea is to convert the string into a character array. Then we can in-place generate all permutations of the given string using backtracking by swapping each of the remaining characters in the string with its first character and then generating all the permutations of the remaining characters using a recursive call.



Recursion Tree for string "ABC"

```
package Backtracking;

public class All_Permutations_of_String {
  private static void swap(char[] chars, int i, int j) {
    char temp = chars[i];
    chars[i] = chars[j];
    chars[j] = temp;
  }

  // Recursive function to generate all permutations of a string
  private static void permutations(char[] chars, int currentIndex) {
    if (currentIndex == chars.length - 1) {
      System.out.println(String.valueOf(chars));
    }

    for (int i = currentIndex; i < chars.length; i++) {
      swap(chars, currentIndex, i);
      permutations(chars, currentIndex + 1);
      swap(chars, currentIndex, i);
    }
  }

  public static void findPermutations(String str) {

    // base case
    if (str == null || str.length() == 0) {
      return;
    }

    permutations(str.toCharArray(), 0);
  }

  // generate all permutations of a string in Java
  public static void main(String[] args) {
    String str = "ABC";
    findPermutations(str);
  }
}
```

Find the total number of unique paths in a maze from source to destination

Find the total number of unique paths that the robot can take in a given maze to reach a given destination from a given source

Positions in the maze will either be open or blocked with an obstacle. The robot can only move to positions without obstacles, i.e., the solution should find paths that

contain only open cells. Retracing the one or more cells back and forth is not considered a new path. The set of all cells covered in a single path should be unique from other paths. At any given moment, the robot can only move one step in either of the four directions. The valid moves are:

Go North: (x, y) ——> (x – 1, y)

Go West: (x, y) ——> (x, y – 1)

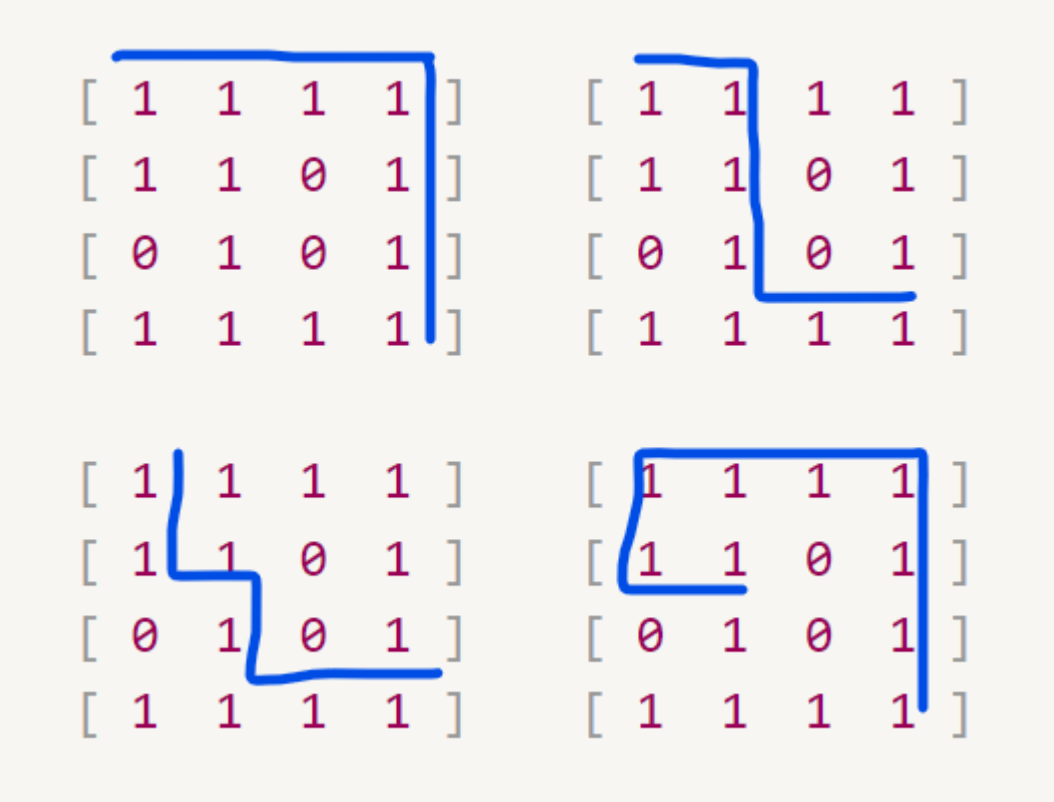
Go South: (x, y) ——> (x + 1, y)

Go East: (x, y) ——> (x, y + 1)

consider the following maze in the form of a binary matrix where 0 represents a blocked cell and 1 represents an open cell:

[1 1 1 1]
[1 1 0 1]
[0 1 0 1]
[1 1 1 1]

We have to find the total number of unique paths from source to destination. The above maze contains 4 unique paths (marked in blue color).




```
[ 1  1  1  1 ]   [ 1  1  1  1 ]
[ 1  1  0  1 ]   [ 1  1  0  1 ]
[ 0  1  0  1 ]   [ 0  1  0  1 ]
[ 1  1  1  1 ]   [ 1  1  1  1 ]

[ 1  1  1  1 ]   [ 1  1  1  1 ]
[ 1  1  0  1 ]   [ 1  1  0  1 ]
[ 0  1  0  1 ]   [ 0  1  0  1 ]
[ 1  1  1  1 ]   [ 1  1  1  1 ]
```

The robot should search for a path from the starting position to the goal position until it finds one or until it exhausts all possibilities.

We can easily achieve this with the help of Backtracking.

We start from the given source cell in the matrix and explore all four paths possible and recursively check if they will lead to the destination or not. We update the unique path count whenever the destination cell is reached. If a path doesn't reach the destination or explored all possible routes from the current cell, we backtrack. To make sure that the path is simple and doesn't contain any cycles, keep track of cells involved in the current path in a matrix, and before exploring any cell, ignore the cell if it is already covered in the current path

```
package Backtracking;
public class Maze_Unique_Paths {
  // Check if cell (x, y) is valid or not
  private static boolean isValidCell(int x, int y, int N) {
    return !(x < 0 || y < 0 || x >= N || y >= N);
  }

  private static int countPaths(int[][] maze, int i, int j,
int x, int y, boolean visited[][]) {

    // if destination (x, y) is found, return 1
    if (i == x && j == y) {
      return 1;
    }

    // stores number of unique paths from source to destination
    int count = 0;

    // mark the current cell as visited
    visited[i][j] = true;

    // `N × N` matrix
    int N = maze.length;

    // if the current cell is a valid and open cell
    if (isValidCell(i, j, N) && maze[i][j] == 1) {
      // go down (i, j) ⟶ (i + 1, j)
      if (i + 1 < N && !visited[i + 1][j]) {
        count += countPaths(maze, i + 1, j, x, y, visited);
      }

      // go up (i, j) ⟶ (i - 1, j)
      if (i - 1 >= 0 && !visited[i - 1][j]) {
        count += countPaths(maze, i - 1, j, x, y, visited);
      }

      // go right (i, j) ⟶ (i, j + 1)
      if (j + 1 < N && !visited[i][j + 1]) {
        count += countPaths(maze, i, j + 1, x, y, visited);
      }

      // go left (i, j) ⟶ (i, j - 1)
      if (j - 1 >= 0 && !visited[i][j - 1]) {
        count += countPaths(maze, i, j - 1, x, y, visited);
      }
    }

    // backtrack from the current cell and remove it from the current path
    visited[i][j] = false;

    return count;
  }

  public static int findCount(int[][] maze, int i, int j, int x, int y) {
    // base case: invalid input
    if (maze == null || maze.length == 0 || maze[i][j] == 0 || maze[x][y] == 0) {
```

```
      return 0;
    }

    // `N × N` matrix
    int N = maze.length;

    // 2D matrix to keep track of cells involved in the current path
    boolean[][] visited = new boolean[N][N];

    // start from source cell (i, j)
    return countPaths(maze, i, j, x, y, visited);
  }

  public static void main(String[] args) {
    int[][] maze = {
        { 1, 1, 1, 1 },
        { 1, 1, 0, 1 },
        { 0, 1, 0, 1 },
        { 1, 1, 1, 1 } };

    // source cell (0, 0), destination cell (3, 3)
    int count = findCount(maze, 0, 0, 3, 3);

    System.out.println("The total number of unique paths are " + count);
  }
}
```

Print all shortest routes in a rectangular grid

Given an `M × N` rectangular grid, print all routes in the grid that start at the first cell `(0, 0)` and ends at the last cell `(M-1, N-1)`. We can move down or right or diagonally (down-right), but not up or left.

```
Input:

{ 1, 2, 3 }
{ 4, 5, 6 }
{ 7, 8, 9 }

Output:

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

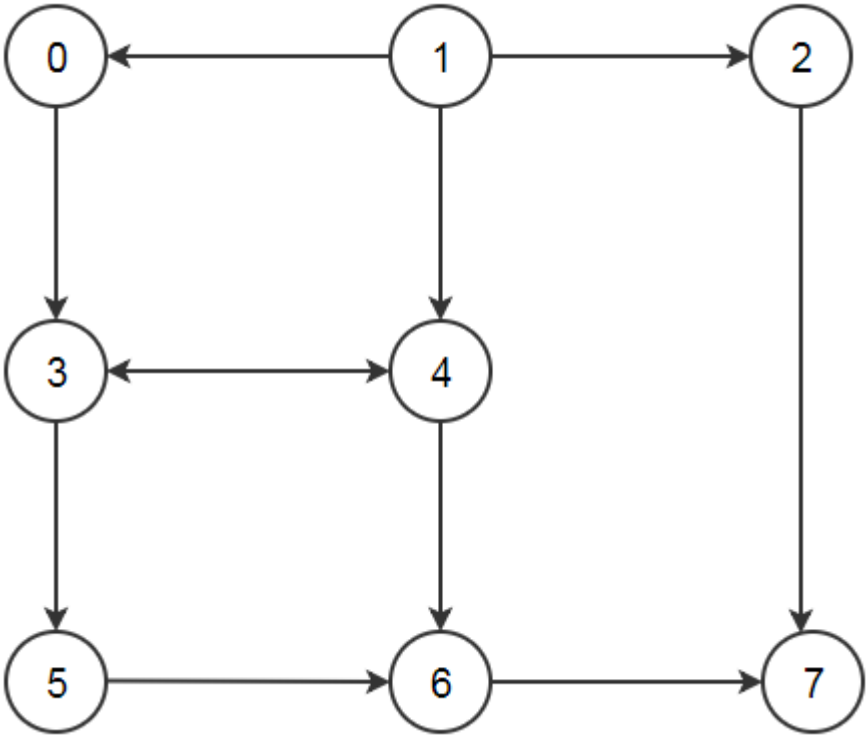
```
[ 1, 5, 6, 9 ]
[ 1, 5, 9 ]
```

```
package Backtracking;
import java.util.Stack;
public class ShortestRoutes {
  // Recursive function to get all routes in a rectangular grid
  // that start at cell (i, j) and ends at the last cell (M-1, N-1).
  public static void printPaths(int[][] mat, Stack<Integer> route, int i, int j) {
    // base case
    if (mat == null || mat.length == 0) {
      return;
    }

    int M = mat.length;
    int N = mat[0].length;

    // include current cell in route
    route.add(mat[i][j]);

    // if the last cell is reached
    if (i == M - 1 && j == N - 1) {
      System.out.println(route);
    } else {
      // move down
      if (i + 1 < M) {
        printPaths(mat, route, i + 1, j);
      }

      // move right
      if (j + 1 < N) {
        printPaths(mat, route, i, j + 1);
      }

      // move diagonally
      if (i + 1 < M && j + 1 < N) {
        printPaths(mat, route, i + 1, j + 1);
      }
    }

    // backtrack: remove the current cell from the route
    route.pop();
  }

  // Print all routes in a rectangular grid
  public static void printPaths(int[][] mat) {
    // list to store the current route
    Stack<Integer> route = new Stack<>();

    // start from the first cell (0, 0)
    printPaths(mat, route, 0, 0);
  }

  public static void main(String[] args) {
    int[][] mat = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

```
    printPaths(mat);
  }
}
```

Find the path between given vertices in a directed graph

Given a directed graph and two vertices (say source and destination vertex), determine if the destination vertex is reachable from the source vertex or not. If a path exists from the
source vertex to the destination vertex, print it.



For example, there exist two paths `[0–3–4–6–7]` and `[0–3–5–6–7]` from vertex `0` to vertex `7` in the following graph. In contrast, there is no path from vertex `7` to any other vertex.

```
package Backtracking;
import java.util.*;
//A class to store a graph edge
class Edge {
  public final int source, dest;

  private Edge(int source, int dest) {
```

```
    this.source = source;
    this.dest = dest;
  }

  // Factory method for creating an immutable instance of `Edge`
  public static Edge of(int a, int b) {
    return new Edge(a, b); // calls private constructor
  }
}

//A class to represent a graph object
class Graph {
  // A list of lists to represent an adjacency list
  List<List<Integer>> adjList = null;

  // Constructor
  Graph(List<Edge> edges, int n) {
    adjList = new ArrayList<>();

    for (int i = 0; i < n; i++) {
      adjList.add(new ArrayList<>());
    }

    // add edges to the directed graph
    for (Edge edge : edges) {
      adjList.get(edge.source).add(edge.dest);
    }
  }
}

public class Path_Directed_Graph {
  // Function to perform BFS traversal from a given source vertex in a graph to
  // determine if a destination vertex is reachable from the source or not
  public static boolean isReachable(Graph graph, int src, int dest) {
    // get the total number of nodes in the graph
    int n = graph.adjList.size();

    // to keep track of whether a vertex is discovered or not
    boolean[] discovered = new boolean[n];

    // create a queue for doing BFS
    Queue<Integer> q = new ArrayDeque<>();

    // mark the source vertex as discovered
    discovered[src] = true;

    // enqueue source vertex
    q.add(src);

    // loop till queue is empty
    while (!q.isEmpty()) {
      // dequeue front node and print it
      int v = q.poll();

      // if destination vertex is found
      if (v == dest) {
        return true;
      }
```

```
      // do for every edge (v, u)
      for (int u : graph.adjList.get(v)) {
        if (!discovered[u]) {
          // mark it as discovered and enqueue it
          discovered[u] = true;
          q.add(u);
        }
      }
    }

    return false;
  }

  public static void main(String[] args) {
    // List of graph edges as per the above diagram
    List<Edge> edges = Arrays.asList(
        Edge.of(0, 3),
        Edge.of(1, 0),
        Edge.of(1, 2),
        Edge.of(1, 4),
        Edge.of(2, 7),
        Edge.of(3, 4),
        Edge.of(3, 5),
        Edge.of(4, 3),
        Edge.of(4, 6),
        Edge.of(5, 6),
        Edge.of(6, 7));

    // total number of nodes in the graph (labeled from 0 to 7)
    int n = 8;

    // build a graph from the given edges
    Graph graph = new Graph(edges, n);

    // source and destination vertex
    int src = 0, dest = 7;

    // perform BFS traversal from the source vertex to check the connectivity
    if (isReachable(graph, src, dest)) {
      System.out.println("Path exists from vertex " + src + " to vertex " + dest);
    } else {
      System.out.println("No path exists between vertices " + src + " and " + dest);
    }
  }
}
```

Find ways to calculate a target from elements of the specified array

Given an integer array, return the total number of ways to calculate the specified target from array elements using only the addition and subtraction operator. The use of any other operator is forbidden.

```
Consider the array { 5, 3, -6, 2 }.

The total number of ways to reach a target of 6 using only + and – operators is 4 as:

(-)-6 = 6
(+) 5 (+) 3 (-) 2 = 6
(+) 5 (-) 3 (-) -6 (-) 2 = 6
(-) 5 (+) 3 (-) -6 (+) 2 = 6

Similarly, there are 4 ways to calculate the target of 4:

(-)-6 (-) 2 = 4
(-) 5 (+) 3 (-)-6 = 4
(+) 5 (-) 3 (+) 2 = 4
(+) 5 (+) 3 (+)-6 (+) 2 = 4
```

```
package Backtracking;

public class Calculate_Taget_Element {
  // Count ways to calculate a target from elements of a specified array
  public static int countWays(int[] nums, int i, int target) {
    // base case: if a target is found
    if (target == 0 && i == nums.length) {
      return 1;
    }

    // base case: no elements are left
    if (i == nums.length) {
      return 0;
    }

    // 1. ignore the current element
    int exclude = countWays(nums, i + 1, target);

    // 2. Consider the current element

    // 2.1. Subtract the current element from the target
    // 2.2. Add the current element to the target
    int include = countWays(nums, i + 1, target - nums[i])
+ countWays(nums, i + 1, target + nums[i]);

    // Return total count
    return exclude + include;
  }

  public static void main(String[] args) {
    // input array and target number
    int[] nums = { 5, 3, -6, 2 };
    int target = 6;

    System.out.println(countWays(nums, 0, target) + " ways");
```

```
  }
}
```

```
package Backtracking;

import java.util.ArrayDeque;
import java.util.Deque;

class Pair {
  Integer num;
  Character sign;

  Pair(Integer num, Character sign) {
    this.num = num;
    this.sign = sign;
  }

  @Override
  public String toString() {
    return "(" + this.sign + ")" + this.num + " ";
  }
}

public class Calculate_Target_Element_print_all_pair {
  private static void printList(Deque<Pair> list) {
    for (Pair p : list) {
      System.out.print(p);
    }
    System.out.println();
  }

  // Print all ways to calculate a target from elements of a specified array
  public static void printWays(int[] nums, int i, int target, Deque<Pair> auxlist) {
    // base case: if a target is found, print the result list.
    if (target == 0 && i == nums.length) {
      printList(auxlist);
    }

    // base case: no elements are left
    if (i == nums.length) {
      return;
    }

    // ignore the current element
    printWays(nums, i + 1, target, auxlist);

    // consider the current element and subtract it from the target
    auxlist.addLast(new Pair(nums[i], '+'));
    printWays(nums, i + 1, target + nums[i], auxlist);
```

```
    auxlist.pollLast(); // backtrack

    // consider the current element and add it to the target
    auxlist.addLast(new Pair(nums[i], '-'));
    printWays(nums, i + 1, target - nums[i], auxlist);
    auxlist.pollLast(); // backtrack
  }

  public static void main(String[] args) {
    // input array and target number
    int[] nums = { 5, 3, -6, 2 };
    int target = 6;

    printWays(nums, 0, target, new ArrayDeque<>());
  }
}
```