BACKTRACKING

Backtracking is an algorithmic technique for solving recursive problems to build every solution incrementally and removing those solutions that fail to satisfy the constraints of the problem at any point of time.

It's often used to solve constraint satisfaction problems like Sudoku, N-Queens, or maze-solving problems.

KEY CONCEPTS OF BACKTRACKING:

1. **Choice:** At each step, make a choice from a set of available options.
2. **Constraints:** Check if the current partial solution satisfies the problem's constraints.
3. **Goal:** If a solution is reached, return the solution.
4. **Backtrack:** If a choice doesn't lead to a solution, undo the last choice (backtrack) and try another.

N-QUEENS PROBLEM

The N-Queens Problem is a classic backtracking problem where the goal is to place N queens on an N×NN \times NN×N chessboard such that no two queens attack each other. Queens can attack horizontally, vertically, and diagonally, so the solution must ensure that no two queens are on the same row, column, or diagonal.

APPROACH

- We place queens column by column, starting from the leftmost column.
- For each column, we try placing the queen in each row.
- Before placing a queen in a cell, we check whether it is safe (i.e., no other queen threatens it).
- If placing the queen results in a conflict, we backtrack by removing the queen and trying the next row.
- If we successfully place queens in all columns, the solution is found.

CODE

```
#include <iostream>

#include <vector>

using namespace std;
```

```
// Function to check if it's safe to place a queen at board[row][col]
bool isSafe(const vector<vector<int>>& board, int row, int col, int N) {
    // Check the same row on the left side
    for (int i = 0; i < col; i++)
        if (board[row][i])
            return false;

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

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

    return true;
}

// Recursive utility function to solve N-Queens problem
bool placeQueen(vector<vector<int>>& board, int col, int N) {
    // If all queens are placed, return true
    if (col >= N)
        return true;
```

```
    // Consider this column and try placing a queen in all rows one by one
    for (int i = 0; i < N; i++) {
        // Check if it's safe to place the queen at board[i][col]
        if (isSafe(board, i, col, N)) {
            // Place the queen
            board[i][col] = 1;

            // Recur to place the rest of the queens
            if (placeQueen(board, col + 1, N))
                return true;

            // If placing the queen doesn't lead to a solution, backtrack
            board[i][col] = 0; // Remove the queen
        }
    }

    // If the queen cannot be placed in any row in this column, return false
    return false;
}

// Function to solve the N-Queens problem
bool solveNQueens(int N) {
    // Create an N x N chessboard initialized with 0
    vector<vector<int>> board(N, vector<int>(N, 0));

    // Use the utility function to solve the problem
    if (!placeQueen(board, 0, N)) {
```

```
        cout << "No solution exists" << endl;
        return false;
    }

    // Print the solution
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            cout << (board[i][j] ? "Q " : ". ");
        cout << endl;
    }

    return true;
}

int main() {
    int N = 8; // Example for 8-Queens
    solveNQueens(N);
    return 0;
}
```

Output:

```
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
. . Q . . . . .
```

EXPLANATION

- **isSafe function**: Checks whether placing a queen at board[row][col] is valid. It ensures no queen is already placed in the same row, or on the upper and lower diagonals.
- **placeQueen function**: This is a recursive function that places queens column by column. If it successfully places queens in all columns, it returns true. Otherwise, it backtracks and tries other rows.
- **solveNQueens function**: This initializes the chessboard and calls the recursive placeQueen to solve the N-Queens problem. If a solution exists, it prints the chessboard with the queens placed.

TIME COMPLEXITY

The time complexity of this backtracking approach is O(N!) where N is the number of queens. This is because, in the worst case, we try to place a queen in each row of every column, and each placement might lead to backtracking.

RAT IN A MAZE PROBLEM

The rat starts at the top-left corner of the maze (position (0, 0)) and tries to reach the bottom-right corner (position (N-1, N-1)). The rat can only move in four possible directions: left, right, up, and down. The goal is to find a path from the start to the destination.

APPROACH

1. Start from the top-left corner (0, 0) of the maze.
2. At each step, check if moving in any of the four possible directions (left, right, up, down) leads to the solution.
3. If moving in a certain direction is safe (i.e., within the maze boundaries and on an open path), recursively try to solve the problem from that point.
4. If the destination is reached, the solution is found.
5. If no solution is found, backtrack by marking the current cell as not part of the solution and try other directions.

CODE

```
#include <iostream>

using namespace std;


#define N 4
```

```
// Function to print the solution matrix
void printSolution(int sol[N][N]) {
   for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++)
         cout << sol[i][j] << " ";
      cout << endl;
   }
}

// A utility function to check if the move is safe
bool isSafe(int maze[N][N], int x, int y) {
   // Check if (x, y) is within bounds and the cell is open (i.e., equals 1)
   return (x >= 0 && x < N && y >= 0 && y < N && maze[x][y] == 1);
}

// Recursive utility function to solve the Rat in a Maze problem
bool findPath(int maze[N][N], int x, int y, int sol[N][N]) {
   // Base case: if (x, y) is the bottom-right corner, return true
   if (x == N - 1 && y == N - 1 && maze[x][y] == 1) {
      sol[x][y] = 1;
      return true;
   }

   // Check if maze[x][y] is valid
   if (isSafe(maze, x, y)) {
      // Mark the current cell as part of the solution path
      sol[x][y] = 1;
```

```
    // Move forward in the x direction
    if (findPath(maze, x + 1, y, sol))
        return true;

    // If moving in x direction doesn't work, try moving in the y direction
    if (findPath(maze, x, y + 1, sol))
        return true;

    // If neither direction works, backtrack: unmark the current cell
    sol[x][y] = 0;
    return false;
  }

  return false;
}

// Function to solve the Rat in a Maze problem using backtracking
bool solveMaze(int maze[N][N]) {
  int sol[N][N] = { { 0, 0, 0, 0 }, { 0, 0, 0, 0 }, { 0, 0, 0, 0 }, { 0, 0, 0, 0 } };

  if (!findPath(maze, 0, 0, sol)) {
    cout << "No solution exists" << endl;
    return false;
  }

  // Print the solution path
```

```
    printSolution(sol);
    return true;
}

int main() {
    int maze[N][N] = { { 1, 0, 0, 0 },
                       { 1, 1, 0, 1 },
                       { 0, 1, 0, 0 },
                       { 1, 1, 1, 1 } };

    solveMaze(maze);
    return 0;
}
```

Output:

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

EXPLANATION

- **isSafe function:** This checks if the current cell (x, y) is within the maze bounds and is an open path (1).
- **findPath function**: This is the recursive function that attempts to find a path for the rat from the current cell (x, y) to the destination (N-1, N-1). It tries to move the rat in both the x and y directions. If either move leads to a solution, it returns true. If both moves fail, it backtracks by unmarking the current cell.
- **solveMaze function**: Initializes the solution matrix, calls the recursive findPath function, and prints the solution if it exists.

TIME COMPLEXITY

The time complexity is O(2N2)O(2^{N^2})O(2N2) in the worst case because at every cell, the rat has two options to move either in the x-direction or the y-direction, and there are N2N^2N2 cells.