Day 28

Backtracking

In this topic, we will learn about the backtracking, which is a very important skill set to solve recursive solutions. Recursive functions are those that calls itself more than once. Consider an example of Palindrome:

Initially, the function isPalindrome(S, 0, 8) is called once with the parameters isPalindrome(S, 1, 7). The recursive call isPalindrome(S, 1, 7) is called once with the parameters isPalindrome(S, 2, 6).

Backtracking is one of the techniques that can be used to solve the problem. We can write the algorithm using this strategy. It uses the Brute force search to solve the problem, and the brute force search says that for the given problem, we try to make all the possible solutions and pick out the best solution from all the desired solutions. This rule is also followed in dynamic programming, but dynamic programming is used for solving optimization problems. In contrast, backtracking is not used in solving optimization problems. Backtracking is used when we have multiple solutions, and we require all those solutions.

Backtracking name itself suggests that we are going back and coming forward; if it satisfies the condition, then return success, else we go back again. It is used to solve a problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criteria.

When to use a Backtracking algorithm?

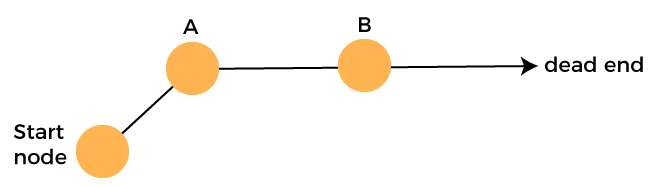
When we have multiple choices, then we make the decisions from the available choices. In the following cases, we need to use the backtracking algorithm:

* A piece of sufficient information is not available to make the best choice, so we use the backtracking strategy to try out all the possible solutions.
* Each decision leads to a new set of choices. Then again, we backtrack to make new decisions. In this case, we need to use the backtracking strategy.

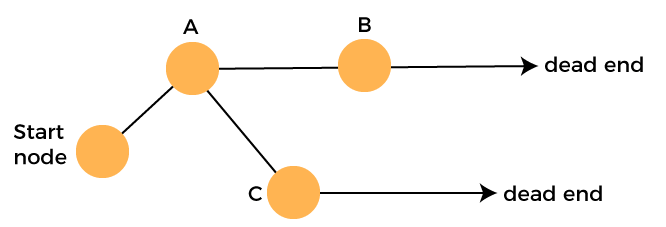
How does Backtracking work?

Backtracking is a systematic method of trying out various sequences of decisions until you find out that works. Let's understand through an example.

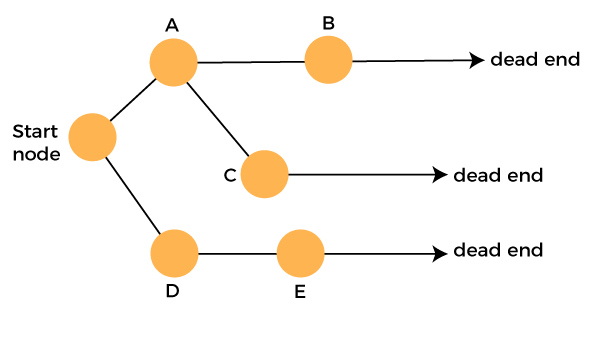
We start with a start node. First, we move to node A. Since it is not a feasible solution so we move to the next node, i.e., B. B is also not a feasible solution, and it is a dead-end so we backtrack from node B to node A.



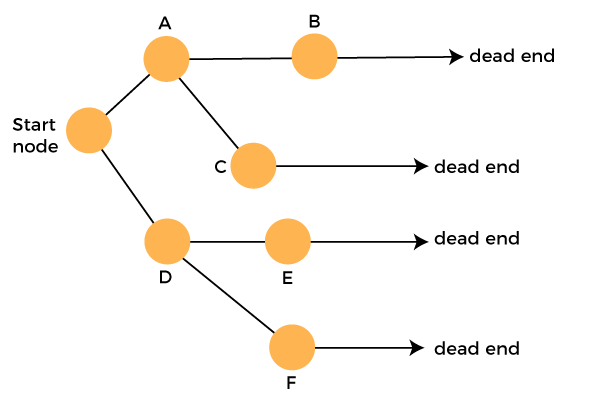
Suppose another path exists from node A to node C. So, we move from node A to node C. It is also a dead-end, so again backtrack from node C to node A. We move from node A to the starting node.



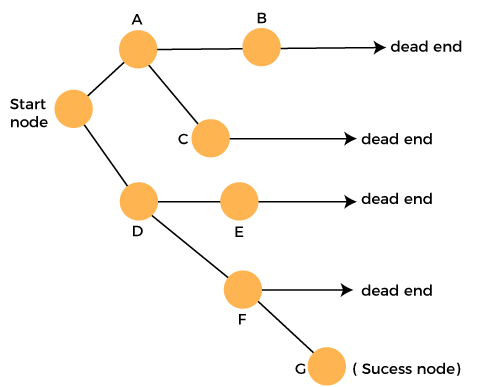
Now we will check any other path exists from the starting node. So, we move from start node to the node D. Since it is not a feasible solution so we move from node D to node E. The node E is also not a feasible solution. It is a dead end so we backtrack from node E to node D.



Suppose another path exists from node D to node F. So, we move from node D to node F. Since it is not a feasible solution and it's a dead-end, we check for another path from node F.



Suppose there is another path exists from the node F to node G so move from node F to node G. The node G is a success node.



**The terms related to the backtracking are:**

* **Live node:** The nodes that can be further generated are known as live nodes.
* **E node:** The nodes whose children are being generated and become a success node.
* **Success node:** The node is said to be a success node if it provides a feasible solution.
* **Dead node:** The node which cannot be further generated and also does not provide a feasible solution is known as a dead node.

Many problems can be solved by backtracking strategy, and that problems satisfy complex set of constraints, and these constraints are of two types:

* **Implicit constraint:** It is a rule in which how each element in a tuple is related.
* **Explicit constraint:** The rules that restrict each element to be chosen from the given set.

Applications of Backtracking

* N-queen problem
* Sum of subset problem
* Graph coloring
* Hamiliton cycle

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.

**Print all Permutations**

Time complexity - O(n\*n!)

public class Recursion3 {

   public static void printPermutation(String str, int idx, String perm) {

       if(str.length() == 0) {

           System.out.println(perm);

           return;

       }

       for(int i=0; i<str.length(); i++) {

           char currChar = str.charAt(i);

           String newStr = str.substring(0, i) + str.substring(i+1);

           printPermutation(newStr, idx+1, perm+currChar);

       }

   }

   public static void main(String args[]) {

       String str = "abc";

       printPermutation(str, 0, "");

   }

}

**N-Queens**

Time complexity - O(n^n)

class Solution {

   public boolean isSafe(int row, int col, char[][] board) {

       //horizontal

       for(int j=0; j<board.length; j++) {

           if(board[row][j] == 'Q') {

               return false;

           }

       }

       //vertical

       for(int i=0; i<board.length; i++) {

           if(board[i][col] == 'Q') {

               return false;

           }

       }

       //upper left

       int r = row;

       for(int c=col; c>=0 && r>=0; c--, r--) {

           if(board[r][c] == 'Q') {

               return false;

           }

       }

       //upper right

       r = row;

       for(int c=col; c<board.length && r>=0; r--, c++) {

           if(board[r][c] == 'Q') {

               return false;

           }

       }

       //lower left

       r = row;

       for(int c=col; c>=0 && r<board.length; r++, c--) {

           if(board[r][c] == 'Q') {

               return false;

           }

       }

       //lower right

       for(int c=col; c<board.length && r<board.length; c++, r++) {

           if(board[r][c] == 'Q') {

               return false;

           }

       }

       return true;

   }

   public void saveBoard(char[][] board, List<List<String>> allBoards) {

       String row = "";

       List<String> newBoard = new ArrayList<>();

       for(int i=0; i<board.length; i++) {

           row = "";

           for(int j=0; j<board[0].length; j++) {

               if(board[i][j] == 'Q')

                   row += 'Q';

               else

                   row += '.';

           }

           newBoard.add(row);

       }

       allBoards.add(newBoard);

   }

   public void helper(char[][] board, List<List<String>> allBoards, int col) {

       if(col == board.length) {

           saveBoard(board, allBoards);

           return;

       }

       for(int row=0; row<board.length; row++) {

           if(isSafe(row, col, board)) {

               board[row][col] = 'Q';

               helper(board, allBoards, col+1);

               board[row][col] = '.';

           }

       }

   }

   public List<List<String>> solveNQueens(int n) {

       List<List<String>> allBoards = new ArrayList<>();

       char[][] board = new char[n][n];

       helper(board, allBoards, 0);

       return allBoards;

   }

}