Lesson:

Backtracking

PW SKILLS

Pre-Requisites

- Basics of Cpp
- Recursion in Cpp

Today's Checklist:

- Introduction to Backtracking
- Working of Backtracking
- Problem Solving on Backtracking

Introduction to Backtracking

Whenever Recursion is applied to any problem , the problem is broken into smaller ones and logic is applied to them. In Backtracking, we try to eliminate solutions that fail to fulfill the conditions of the problem. It follows the Recursion process along with eliminating solutions that do not satisfy the conditions.

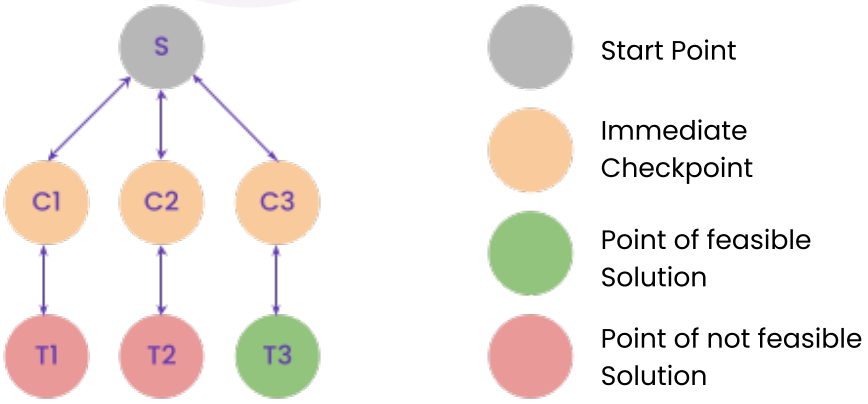
In Recursion, the recursive function calls itself until it reaches the base case. While Backtracking, we apply Recursion to check all possibilities until we get our final solution.

Steps -

- If the current solution satisfies the given condition, then return success.
- If the current solution is the final point, then the return failed.
- If the current solution is not the final point, check for other solutions(explore and repeat above steps).

Working of Backtracking

The Backtracking algorithm tries to figure out the path of the most feasible solution. In the process of searching for the solution,checkpoints are created. So, if the solution is not available at one path, it reverts to the checkpoint and then moves to the next path, and so on. Let us understand this with an example with a 'space state tree' representation of the problem:



Following is the flow of the above graph:

- Start from node S.
- Move to node C1 and mark it as a checkpoint.
- Traverse to the path under C1 till T1 to look for a feasible solution.

- The solution is not found at T1, so move back to the recent checkpoint, i.e., C1.
- Traverse to the next path, i.e., C2, and perform the same operations from 2 to 3.
- Move to C3 and mark it as a checkpoint.
- Traver to the path from C3 to the solution, which is found at T3.
- Return from S to T3 as a feasible solution.

So, to summarize the overall flow of the algorithm, we will perform:
Step 1: If the current point is detected as a feasible solution, we will return success.
Step 2: However, if no path exists to traverse, then return failure, indicating that no solution exists.
Step 3: But if the path exists, then backtrack and explore the solution.
Let us now explore that theory and do a bit of programming using Backtracking.

Problem 1: Permutation with backtracking

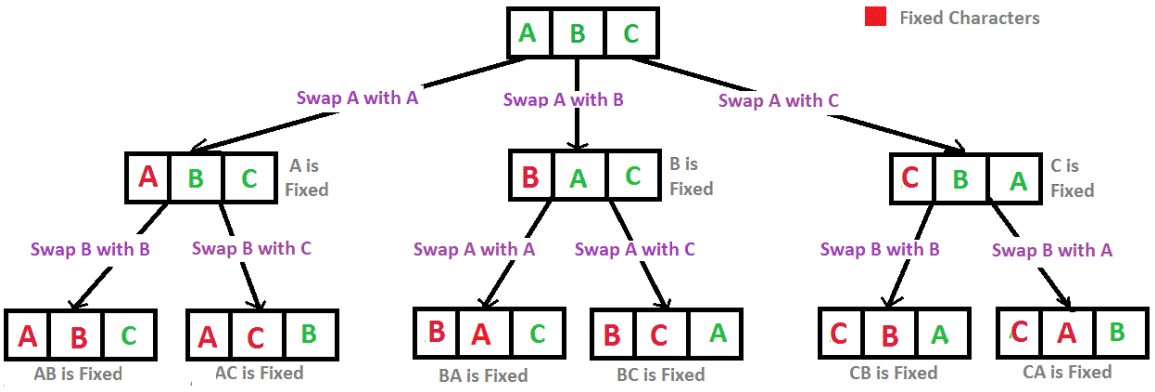
Write a program to print all permutations of the given string s in lexicographically sorted order.

Input: PQR

Output:
PQR PRQ QPR QRP RPQ RQP

Explanation:
Given string PQR has permutations in 6
forms as PQR, PRQ, QPR, QRP, RPQ and RQP.

Explanation: So, in this question, you are required to return all the possible permutations of the given string s. What does that mean? This means that whatever string you are provided with, you need to return all the possible arrangements of it in the form of a string.



Recursion Tree for Permutations of String "ABC"

Explanation of the above diagram

- We'll fix one character at every step then permutations of the remaining characters are written next to them one by one.
- Next, we'll fix two characters and so on. These steps are followed by writing the permutation of the remaining characters next to the fixed characters

Approach:
Code link Cpp :

Explanation:

- The backtrack function takes two arguments: the input string s and an index i indicating the current position in the string.
- The base case occurs when i reaches the length of the string, at which point we have generated a complete permutation and print it.
- Otherwise, we iterate over all the characters in the string starting from index i, swapping each character with the one at index i and then recursively calling backtrack with the updated string and the index i+1. After the recursive call, we swap the characters back to their original positions to restore the string and continue with the next iteration.
- The main function prompts the user to enter a string, sorts it in lexicographically increasing order, and calls backtrack with the initial index of 0.

The time complexity of the previous solution is O(n*n!), where n is the length of the string s. This is because there are n! permutations of a string of length n, and generating each permutation requires n swaps and a sort operation, which takes O(n log n) time. However, we can improve the time complexity to O(n!) by avoiding the sort operation.
Here's the updated C++ program:

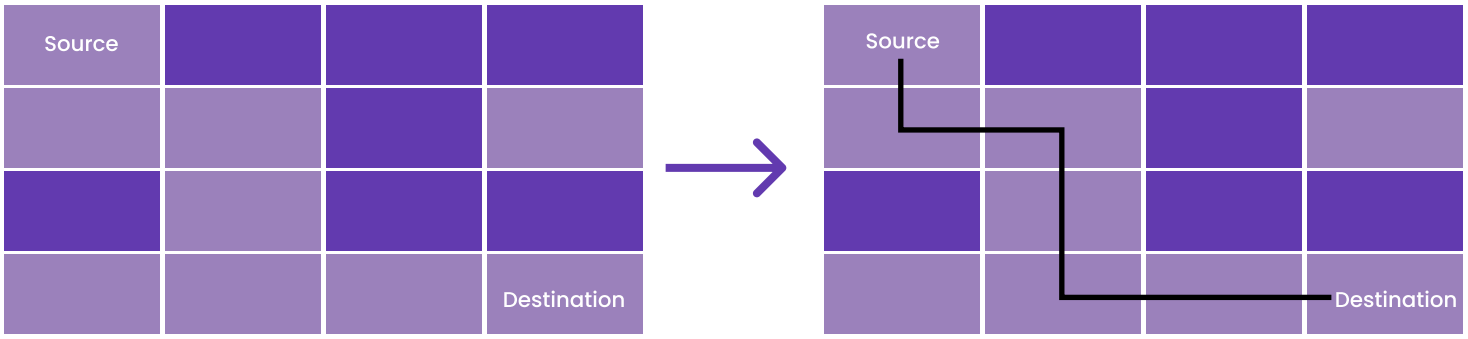
Code Link

Explanation:

- This implementation is similar to the previous one, except that we remove the sort operation. Instead, we rely on the fact that the input string is already sorted in lexicographically increasing order.
- We swap the characters at indices i and j, generate all permutations starting from index i+1, and then swap the characters back to their original positions before moving to the next iteration of the loop.
- This approach generates the same set of permutations as the previous implementation, but without the sort operation, it has a time complexity of O(n!), which is optimal for generating all permutations of a string of length n.
- The space complexity of the above approach is O(n), where n is the length of the input string s. This is because we use a single string variable to store the input string, and the backtracking algorithm modifies the string in place without creating any additional data structures.
- The space required for the call stack is also O(n) due to the recursive nature of the backtracking algorithm. Each recursive call creates a new activation record on the stack, which contains the current state of the string and the current index i. Since the maximum depth of the recursion tree is n, the space required for the call stack is O(n).
- Therefore, the total space complexity of the above approach is O(n) + O(n) = O(n).

Problem 2: Rat in a Maze ()

Rat in a Maze – A maze is an N*N binary matrix of blocks where the upper left block is known as the Source block, and the lower rightmost block is known as the Destination block. If we consider the maze, then maze[0][0] is the source, and maze[N-1][N-1] is the destination. Our main task is to reach the destination from the source. We have considered a rat as a character that can move either forward or downwards.
In the maze matrix, a few dead blocks will be denoted by 0 and active blocks will be denoted by 1. A rat can move only in the active blocks.



Binary Representation –

Input -

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

Output – (We will print 1 for our source to destination path)

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

Intuition – We can solve this problem using Recursion. We need a path from source to destination to try multiple paths. When we use any random path and it fails to reach the destination, we can backtrack and try another path. Since we are trying the same logic multiple times, we can use the recursive technique. We call this technique of Backtracking and trying another path a *'Backtracking Algorithm'.*

Approach – Create a recursive function that will follow a path, and if that path fails, then Backtrack and try another path. Let us see each step –

- Create an output matrix having all values as 0.
- Create a recursive function which will take input matrix, output matrix, and rat position (i,j).
- If the position is not valid, then return.
- Make output[i][j] as one and verify if the current position is the destination or not. If yes, return the output matrix.
- Recursively call the function for position (i+1, j) and (i,j+1).
- Make output[i][j] as 0.

Cpp solution:

Time Complexity –

If there are N rows and N columns, then complexity will be O(2^(N^2)).
There are N^2 total cells, and for each cell, we have 2 options (right direction and down direction). Hence, complexity is 2^(N^2).

Space Complexity –

If there are N rows and N columns, then complexity will be O(N^2).

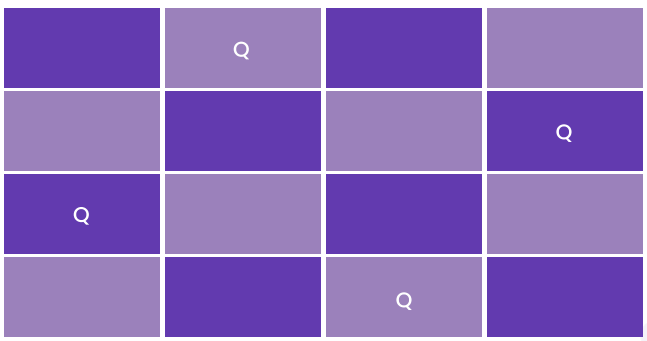
Problem 3: N Queens ()

Consider an N*N chessboard. N Queen Problem is to accommodate N queens on the N*N chessboard such that no 2 queens can attack each other.

Sample Input 1

Input	N=4
Output	{ 0, 1, 0, 0} { 0, 0, 0, 1} { 1, 0, 0, 0} { 0, 0, 1, 0}

Explanation:



Naive Approach – Try to generate all possible combinations and print the combination that satisfies the condition.

Efficient Approach – This can be solved using Recursion.
Intuition - Our main task is to place a queen with no clash with another queen. We will check this condition for all columns one by one. We will backtrack and check for another column if there is any clash. This technique is a backtracking algorithm.

Steps -

- Start from the left-most column.
- If all queens are placed, then return true. (base case)
- Iterate through every row for the current column.
- If the queen is safely placed in the current row, mark [row, column] as a solution.
- Verify if placing the queen in her current position is safe. Then return true.
- If placing queen is not safe, unmark [row, column], then backtrack and go to step 4 to check other rows.
- If all the rows have been checked and are not fulfilling the condition, return false and backtrack agai

CPP CODE

Time Complexity –
If there are N Queens, the complexity will be O(N!).
Space Complexity –
If there are N Queens, the complexity will be O(N).

Upcoming Class Teasers:

- Backtracking-2