Lesson:

Backtracking

PW SKILLS

Pre-Requisites

- Basics of Cpp
- Recursion in Cpp

Today's Checklist:

1. Introduction to Backtracking
2. Working of Backtracking
3. 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.

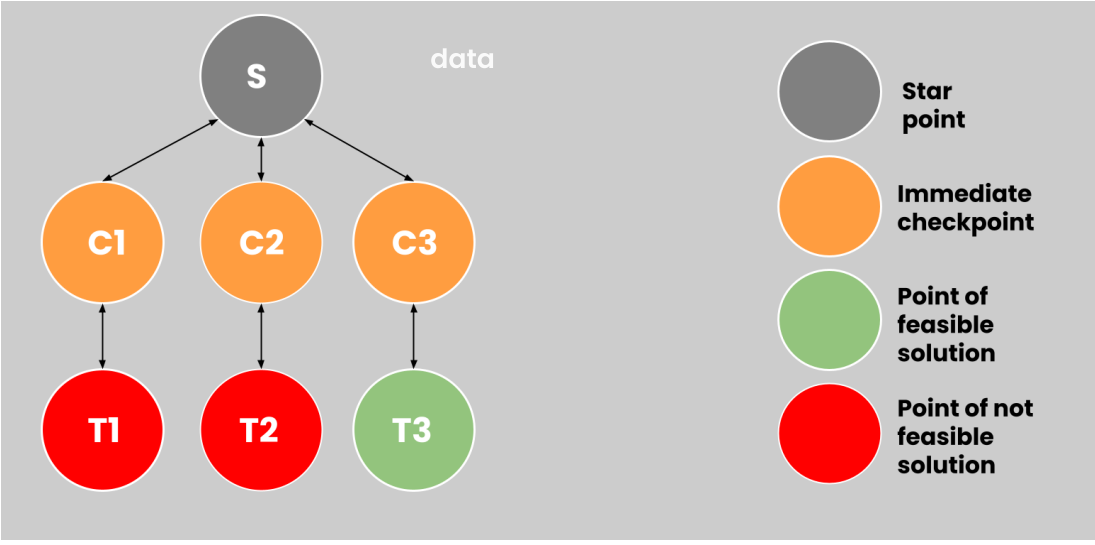
In Recursion, the recursive function calls itself until it reaches the base case. While Backtracking, we apply Recursion to check all possibilities until we get our final solution.

Steps -

1. If the current solution satisfies the given condition, then return success.
2. If the current solution is the final point, then the return failed.
3. 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:

1. Start from node S.
2. Move to node C1 and mark it as a checkpoint.
3. Traverse to the path under C1 till T1 to look for a feasible solution.
4. The solution is not found at T1, so move back to the recent checkpoint, i.e., C1.
5. Traverse to the next path, i.e., C2, and perform the same operations from 2 to 3.
6. Move to C3 and mark it as a checkpoint.
7. Traver to the path from C3 to the solution, which is found at T3.
8. Return from S to T3 as a feasible solution.

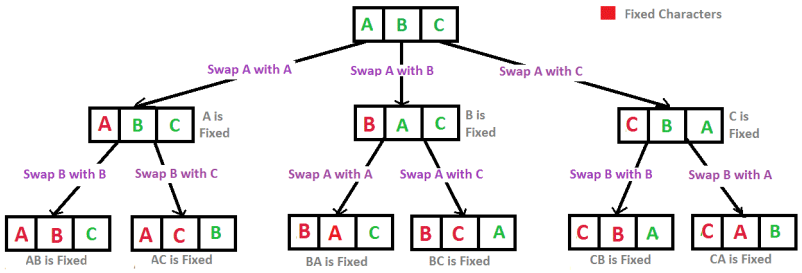
So, to summarize the overall flow of the algorithm, we will perform:
Step 1: If the current point is detected as a feasible solution, we will return success.
Step 2: However, if no path exists to traverse, then return failure, indicating that no solution exists.
Step 3: But if the path exists, then backtrack 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

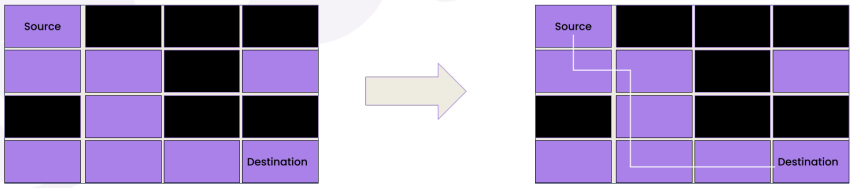
Code link:

Explanation:

1. The main method simply calls the printPermutations method with a sample string.
2. The printPermutations method converts the input string s to a character array arr , then sort the character array in increasing order and calls the generatePermutations method with arr and an initial index of 0.
3. The generatePermutations method is the recursive function that generates all permutations. It takes two parameters:
   - arr is the character array representing the current permutation.
   - index is the current index being considered for the permutation.
4. The base case of the recursion is when index is equal to the length of arr, which means that all characters have been placed in a permutation. In this case, the current permutation is printed and the function returns.
5. In the recursive case, the function loops through each subsequent index starting from index 0. For each index, it swaps the character at that index with the character at the provided index, effectively placing that character in the current permutation. It then recursively calls itself with the updated arr and the next index, generating permutations for the remaining indices. After the recursive call, the function swaps the characters back to restore the original order and continue the loop.
6. Once all recursive calls have returned, the function has generated all permutations of the input string.

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)

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'.**

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 –

1. Create an output matrix having all values as 0.
2. Create a recursive function which will take input matrix, output matrix, and rat position (i,j).
3. If the position is not valid, then return.
4. Make output[i][j] as one and verify if the current position is the destination or not. If yes, return the output matrix.
5. Recursively call the function for position (i+1, j) and (i,j+1).
6. 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).

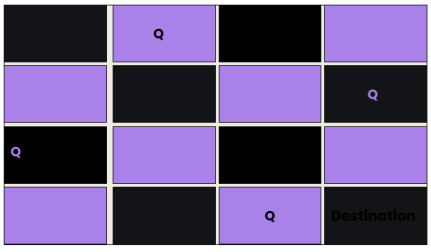
Problem 3: N Queens ()

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 -

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

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).

Problem 4:Sudoku Solver ()

Consider a 9*9 2D array grid that is partially filled with numbers from 1 to 9. The Sudoku Solver problem is to fill remaining blocks with numbers from 1 to 9 so that every row, column and subgrid (3*3) contains exactly one instance of digits (1 to 9).

Example:

Input - (Unfilled cells are denoted as 0).

```
{ {3, 0, 6, 5, 0, 8, 4, 0, 0},
  {5, 2, 0, 0, 0, 0, 0, 0, 0},
  {0, 8, 7, 0, 0, 0, 0, 3, 1},
  {0, 0, 3, 0, 1, 0, 0, 8, 0},
  {9, 0, 0, 8, 6, 3, 0, 0, 5},
  {0, 5, 0, 0, 9, 0, 6, 0, 0},
  {1, 3, 0, 0, 0, 0, 2, 5, 0},
  {0, 0, 0, 0, 0, 0, 0, 7, 4},
  {0, 0, 5, 2, 0, 6, 3, 0, 0} }
```

Output -

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

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

Efficient Approach – We can solve this problem using Recursion:

1. Create a function that will check if the assignment of the number to the cell is safe or not.
2. Create a recursive function that will take the grid as input.
3. Assign a number to an unfilled cell, after which one must check if it is safe to assign. If it is safe to assign, then call the function for all safe cases. If any recursive call returns true, simply return true. If no recursive call returns true, then return false.
4. If there is no unassigned cell, return true.

Code link

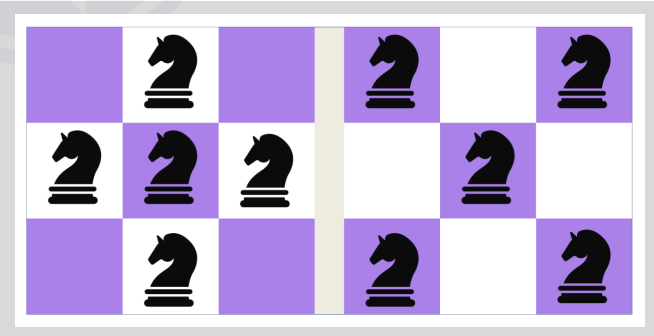
Time Complexity – Time complexity will be O(9^(N*N)).
Space Complexity – Space complexity will be O(N*N).

Output:

Problem 5:Place K-knights such that they do not attack each other

Given integers M, N and K, the task is to place K knights on an M*N chessboard such that they don't attack each other. The knights are expected to be placed on different squares on the board. A knight can move two squares vertically and one square horizontally or two squares horizontally and one square vertically. The knights attack each other if one of them can reach the other in single move. There are multiple ways of placing K knights on an M*N board or sometimes, no way of placing them. We are expected to list out all the possible solutions.

Examples:
Input: M = 3, N = 3, K = 5 **Output:** K A K A K A K A K A K K K A K A **Total number of solutions** : 2



Input: M = 5, N = 5, K = 13 **Output:** K A K A K A K A K A K A K A K A K A K A K A K A K **Total number of solutions :** 1

Approach:

- This problem can be solved using backtracking. The idea is to place the knights one by one starting from first row and first column and moving forward to first row and second column such that they don't attack each other.
- When one row gets over, we move to the next row. Before placing a knight, we always check if the block is safe i.e. it is not an attacking position of some other knight.

- If it is safe, we place the knight and mark it's attacking position on the board else we move forward and check for other blocks.
- While following this procedure, we make a new board every time we insert a new knight into our board.
- This is done because if we get one solution and we need other solutions, then we can backtrack on our old board with the old configuration of knights which can then be checked for other possible solutions.
- The process of backtracking is continued till we get all our possible solutions.

Below is the implementation of the above approach:

Code link

Problem 6: Given a list of integers and a target integer, we have to return a list of all unique combinations of candidates where the chosen numbers sum to the target. We can return the combinations in any order. The same number may be chosen from the list an unlimited number of times. Two combinations are unique if the frequency of at least one of the chosen numbers is different.

Sample Input: nums = [2,3,5], target = 8
Sample Output: [[2,2,2,2], [2,3,3], [3,5]]

This problem statement can be solved in a number of ways using various approaches. We are going to solve this using the backtracking technique since we have to find all the possible combinations as the solution. So let's get an insight about backtracking before moving on to the approach to solving the problem.

Algorithm

1. First, sort the input list in ascending orderif you want to get the output in increasing order.
2. Then, remove all the recurring values from the list, since duplicate values in the input list can produce duplicates in the output list. But we want output without any duplicates.
3. We try to find the possible solutions with the recursive backtracking approach. We have two base conditions to check. One condition is to check whether the target value becomes negative at any point in time. If the target value becomes negative then it denotes that the particular solution is not suitable and we can ignore searching through it.
4. And another one is checking whether the target value became zero. If the target value becomes zero then it denotes that we have attained the required target through searching. So we have to add that temporary list values to the output list.
5. Else add the present index in that list to the current list and recursively call the function with target = target - cand[i] and index as the current i value, then remove that element.

Example

Consider the input list [2, 3, 5] for our example and the target sum is 8. We have to find all the possible combinations with this input. First sort all the elements in the list to make the further process easy.
Then for the first element 2, check whether the target value is negative or zero by subtracting 2 from the target value. If both are not true then again call the function recursively until either of the condition is true. By repeating the recursive call we can reach a point where 2+2+2+2 equals 8. Thus we got one combination sum which is equal to the required target.
In a similar way, we get two more combinations 2+3+3 and 3+5 which also sums up the target. The process is more clearly explained with the following visual representation.

Output: [[1,2,2],[5]]

Explanation: These are the unique combinations whose sum is equal to target.

Approach:

- The backtrack function recursively tries all possible combinations of candidates that sum up to the target number.
- It keeps track of the current combination in the temp vector and adds it to the result vector when the target is reached. To avoid duplicate combinations, it skips candidates that are the same as the previous one.
- The combinationSum2 function first sorts the candidates in non-decreasing order, and then calls the backtrack function with the initial values of temp, result, and start set to 0. Finally, it returns the result vector.
- In the main function, we create a sample input vector candidates and a target number target, and call the combinationSum2 function to find all unique combinations that sum up to the target. We then print the result vector to the console.

Code link

Problem 6: Knight's Tour

Consider an N*N chessboard. The Knight's Tour problem is to print order when the Knight visits that block of the chessboard. Initially, the knight will be placed at the first block of the chessboard. The rule is that the Knight visits each block exactly once.

Example - Input - N = 8
Output -

```
 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
```

Naive Approach – Try to generate all possible tours and print the tour that is satisfying the condition.

Efficient Approach – We can solve this problem using Recursion.

Intuition – We need outputs such that the Knight visits the block exactly once. So, we will take an empty vector that will take Knight moves, and if that move is violating the rule, then we will Backtrack and try for another move.

Steps –

1. Create a solution vector.
2. Verify if all the blocks are visited and print the solution
3. If not, add the next move to the solution vector and recursively check if this move satisfies the condition.
4. If the above move is not valid, remove this from the solution vector and try another move.
5. If none of the moves work, then return false.

Time Complexity – If there are N rows and N columns, complexity will be O(8^(N^2)).

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

CODE Link:

Output:

```
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
```