Back Tracking

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

→ Backtracking is a algorithm technique for solving problem 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 search tree.)

Types of Backtracking Algorithms

there are three types of problems in backtracking:—

(a) Decision problem — In this, we search for a feasible solution.
(b) optimization Problem — " " " " " the best solution.
(c) Enumeration problem — " " " " " all feasible solution.

When can be Backtracking Algorithm used?

→ Consider the Soduku solving problem, we try filling digit one by one. Whenever we find that current digit can't lead to a solution, we remove it (backtrack) and try next digit.

→ This is better than naive approach (generating all possible combination of digits and then trying every combination one by one) as it drops a set of permutation whenever it backtracks.

Given a string print all those permutation which doesn't contain "AB" as a substring.

i/p : str = "ABC"
o/p : 'ACB', 'BAC', 'BCA', 'CBA' ✓
'ABC' 'CAB' ×

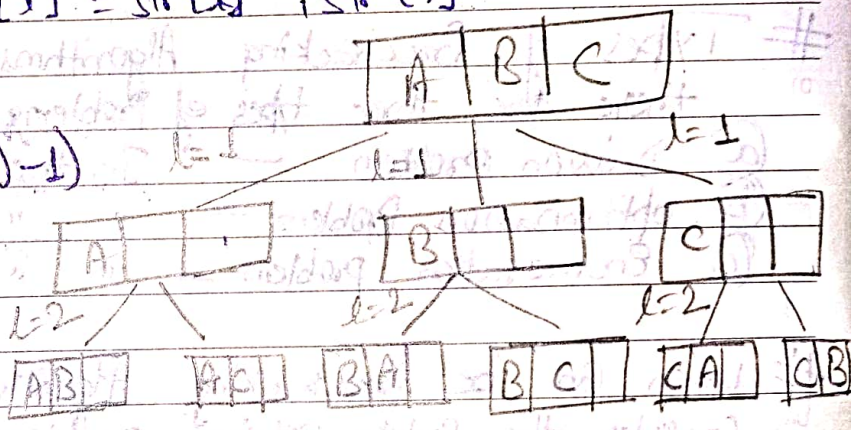
NOVEMBER
03 '24
45th Week
308-058
SUNDAY

S	M	T	W	T	F	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

NOV 2024

M-1

```
def per(str, l, r):
    if l==r:
        if "AB" not in ''.join(str):
            print(*str, sep="", end=" ")
        return
    else:
        for i in range(l, r+1):
            str[i], str[l] = str[l], str[i]
            per(str, l+1, r)
            str[i], str[l] = str[l], str[i]

str = "ABC"
per(list(str), 0, len(str)-1)
```



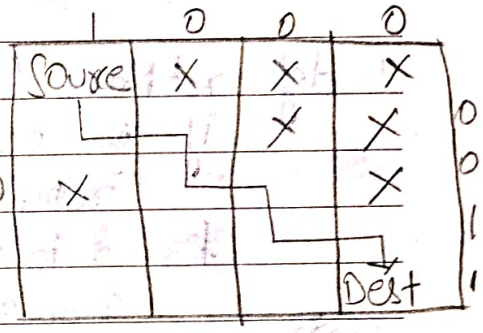
M-2

```
def issafe(str, l, i, r):
    if l!=0 and str[l-1]=='A' and str[i]=='B': return False
    if r==l+1 and str[i]=='A' and str[l]=='B': return False
    return True

def permute(str, l, r):
    if l==r: print(*str, sep="", end=" ") return
    else: for i in range(l, r+1):
            if issafe(str, l, i, r):
                str[i], str[l] = str[l], str[i]
                permute(str, l+1, r)
                str[i], str[l] = str[l], str[i]

str = "ABC"
permute(list(str), 0, len(str)-1)
```

O(n! * n)

Rat in a Maze

A Maze is given as N x N binary matrix of blocks where source block is the upper left most i.e maze[0][0] and destination block is lower rightmost block i.e, maze[N-1][N-1]. A rat starts from source and has to reach the destination. The rat can move only in two direction forward and down.

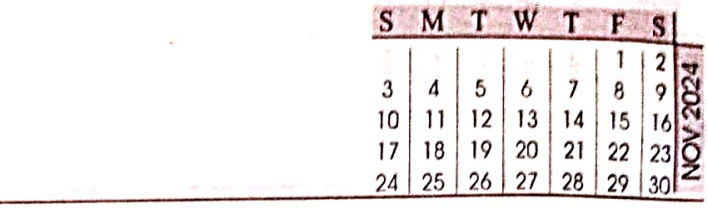


In the maze, 0 means the block is a dead end and 1 means the block can be used in the path from source to destination. Note that this is a simple version of the typical Maze Problem. For example, a more complex version can be that the rat can move in 4 directions and a more complex version can be with limited moves.

Approach: Form a recursive function, which will follow a path & check if the path reaches the destination (or) not. If the path doesn't reach the destination then backtrack and try other paths.

Algorithms:—

1. Create a solution matrix, initially filled with 0's.
2. Create a recursive function, which takes initial matrix, output matrix and positional of rat (i, j).
3. If the position is out of the matrix (or) the position is not valid then return.
4. Mark the positions output[i][j] as 1 and check if the current position is destination (or) not. if destination is reached Print the output matrix and return.
5. Recursively call for position (i+1, j) and (i, j+1)
6. unmark position (i, j). i.e output[i][j] = 0.



n=4

```
def isValid(n, maze, x, y, res):
    if x>=0 and y>=0 and x<n and y<n and maze[x][y]==1
            and res[x][y]==0:
        return True
    return False

def ratMaze(n, maze, move_x, move_y, x, y, res):
    if x==n-1 and y==n-1:
        return True
    for i in range(4):
        x_new = x + move_x[i]
        y_new = y + move_y[i]
        if isValid(n, maze, x_new, y_new, res):
            res[x_new][y_new] = 1
            if ratMaze(n, maze, move_x, move_y, x_new, y_new, res)
                return True
            res[x_new][y_new] = 0
    return False
```

TC = $O(4^{n^2})$
SC = $O(n^2)$

Sudoku Problem

Given a Partially filled 9x9 2d array, the goal is to assign digit (from 1 to 9) to the empty cells so that every row, column and subgrid of size 3x3 can contain exactly one instance of the digit from 1 to 9.

For Code use ChatGPT.

Naive Approach!

The naive approach is to generate all possible configuration of number from 1 to 9 to fill the empty cell. Try every configuration one by one until the correct configuration is found.

i.e., For every unassigned position fill the position with a no. from 1 to 9. After filling all the unassigned position check if the matrix is safe (or) not. If the safe print else recurs for other cases.

Back tracking

Sudoku can be solved by assigning number one by one to empty cells. Before assigning a number, check whether it is safe to assign.

Check that the same no is not present in the current row, current column and current 3×3 subgrid. After checking for safety, assign the number and recursively check whether this assignment leads to a solution (or) not. If the assignment doesn't lead to a solution, then try the next number for the current empty cell. And if none of the number (1 to 9) leads to a solution, return false and print no solution exists.

N Queen's Problem

We are given a no. 'N', we need to consider N×N chess board and we need to place n Queens on the chart ~~board~~ such that no two queens attach each other. Please note that a queen can move horizontally anywhere, vertically anywhere, diagonally anywhere.

i/p = 4
o/p = yes

0	0	1	0
1	0	0	0
0	0	0	1
0	1	0	0

✓

n = 5
o/p: yes

NOVEMBER
07
'24
45th Week
312-054
THURSDAY

S	M	T	W	T	F	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

NOV 2024

Back tracking Algorithms. → TC = O(n!)

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we need to find a row for which there is no clash, we mark this row and column as part of solution.

If we do not find such a row due to clashes, then we backtrack and return false