Algorithms

10. Backtracking

Handwritten by pankaj kumar

Backtracking

Date

Page No. 22.

2. Backtracking

Date
Page No. 23.

* What is backtracking?

- Backtracking is an algorithm technique that considers searching in every possible combination for solving a computational problem
- It is known for solving problems recursively one step at a time and removing those solutions that do not satisfy the problem constraints at any point of time
- It is a refined brute force approach that tries out all the possible solutions and chooses the best possible ones out of them
- The backtracking approach is generally used in the cases where there are possibilities of multiple solutions.

Note:— The term backtracking implies— if the current solution is not suitable, then eliminate that and backtrack (go back) and check for other solutions.

* How it work?

- In any backtracking problems, the algorithm tries to find a path to the feasible solution which has some intermediary checkpoints. In case they don't lead to the feasible solution, the problem can backtrack from the checkpoints and take another path in search of the solution.

* Types of Backtracking Problems

1. Decision Problems— Here, we search for feasible solution.
2. Optimization Problems— For this type, we search for the best solution.
3. Enumeration Problems— We find set of all possible feasible solutions to the problem of this type.

* Backtracking Problem Identification

- Every problem that has clear and well established constraints on any objective solution which incrementally aids candidate to the solution and abandons a candidate ("backtracks") whenever it determines that the candidate is not able to reach a feasible soution. Such problem can be solved by Backtracking. It is generally exponential in nature with

regards to both time and space.

- However, most of the commonly discussed problems, can be solved using other popular algorithm like Dynamic programming or greedy algorithm in O(n), O(logn), or O(n*logn) time complexities in order of I/p size. Hence, in such cases, usage of backtracking becomes an overkill.
- But there still remain some problems that only have backtracking algorithm as the means of solving them till date.

*for example :- we have three boxes and we are told that only one of them has a gold coin in it and we do not know exactly which box has it. So, in order to identify which box has the coin, we will have no other option than opening all the boxes one by one.

[*] Application :-

→ N-Queens Problem
→ Maze Problem
→ Graph coloring problem
→ Hamiliton cycle

[*] Problems :-

⇒ ① Rat In a Maze - (Amazon, microsoft)

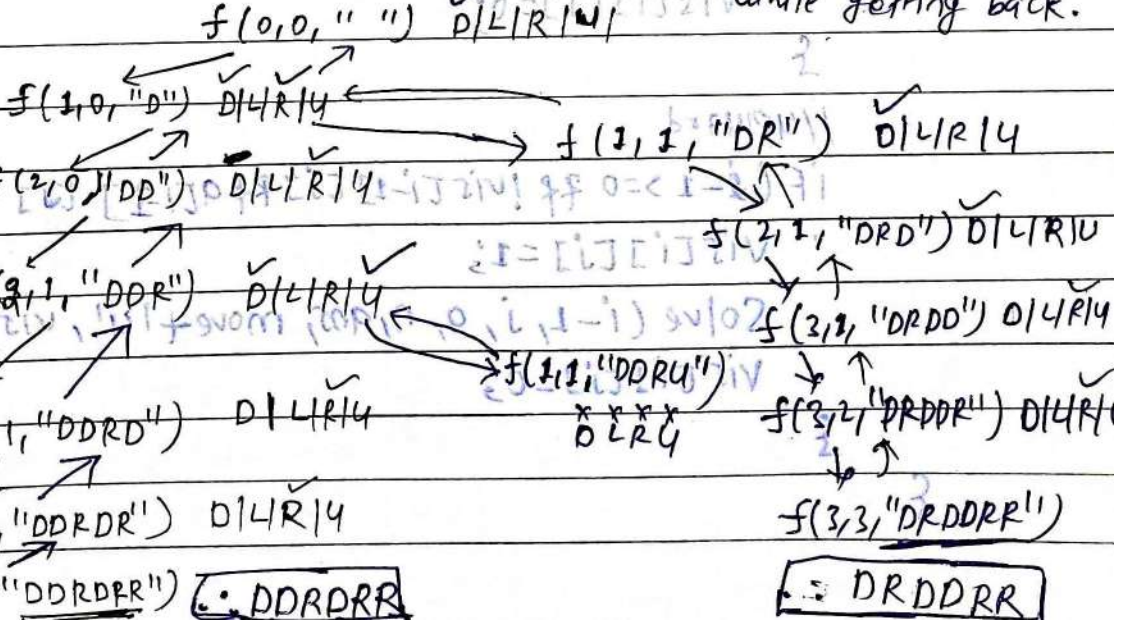
Consider a rat placed at (0,0) in a square matrix of order N*N. It has to reach the destination at (N-1, N-1). Find all possible paths that the rat can take to reach from source to destination. The directions in which the rat can move are up/down, left, right. Value 0 at a cell in the matrix represent that it is blocked and rat cannot move to it while value 1 at a cell in the matrix represent that rat can be travel through it.

Note:- In a path, no cell can be visited more than one time.

Ex 1. Input: N=4	Ex 2. Input N=2
m[][] = {{1,0,0,0}, {1,1,0,1}, {1,1,0,0}, {0,1,1,1}}	m[][] = {{1,0}, {1,0}}
Output: DDRDRR, DRDDRR	Output: -1

Approach :-

- Start at the source (0,0) with an empty string and try every possible path i.e, upwards(U), downward(D), leftwards(L), rightward(R).
- As the answer should be in lexicographical order so it's better to try the directions in lexicographical order i.e (D,L,R,U)
- Declare a 2D-array named visited. because the question states that a single cell should be included only once in the path. so it is important to keep track of the visited cell in array.
- Also check "out of bound" condition while going in particular diren.
- Whenever you reach the destination it's very important to get back as shown in the recursion tree.
- While getting back, keep on unmarking the visited array for the respective direction. also check whether there is diff path possible while getting back.



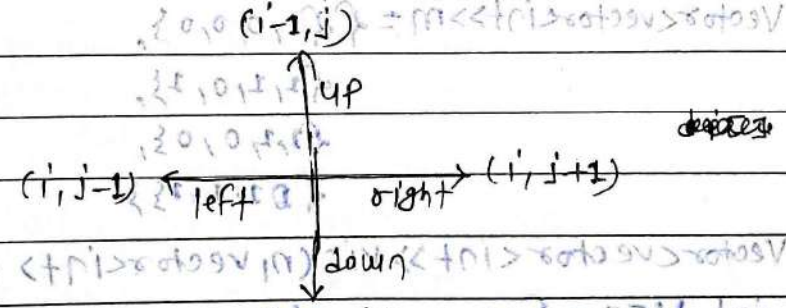
```
void solve(int i, int j, vector<vector<int>>& a, int n,
           vector<string>& ans, string move, vector<vector<int>>& vis){
    if(i==n-1 && j==n-1){
        ans.push_back(move);
        return;
    }
    //downward
    if(i+1<n && !vis[i+1][j] && a[i][j-1]==1){
        vis[i][j]=1;
        solve(i+1, j, a, n, ans, move+'D', vis);
        vis[i][j]=0;
    }
    //Left
    if(j-1>=0 && !vis[i][j-1] && a[i][j-1]==1){
        vis[i][j]=1;
        solve(i, j-1, a, n, ans, move+'L', vis);
        vis[i][j]=0;
    }
    //right
    if(j+1<n && !vis[i][j+1] && a[i][j+1]==1){
        vis[i][j]=1;
        solve(i, j+1, a, n, ans, move+'R', vis);
        vis[i][j]=0;
    }
    //upward
    if(i-1>=0 && !vis[i-1][j] && a[i-1][j]==1){
        vis[i][j]=1;
        solve(i-1, j, a, n, ans, move+'U', vis);
        vis[i][j]=0;
    }
}
```

```
int main(){
    int n=4;
    vector<vector<int>> m = {{1,0,0,0},
                             {1,1,0,1},
                             {1,1,0,0},
                             {0,1,1,1}};
    vector<string> ans;
    vector<vector<int>> vis(n,vector<int>(n,0));
    solve(0,0,m,n,ans,"",vis);
    if(ans.size()==0) cout<<-1;
    else
        for(auto i:ans) cout<<i<<endl;
    return 0;
}
```

Time complexity: $O(4^{(m*n)})$, bcz on every cell we need to try 4 different directions.

Space complexity: $O(m*n)$, maximum depth of recurrence tree (auxiliary space).

Note :— writing an individual code for every direction is a lengthy process therefore we truncate the 4 "if statement" into a single for loop using the following approach.



	D	L	R	U
di[]	+1	+0	+0	-1
dj[]	+0	-1	+1	+0

```
void solve(int i, int j, vector<vector<int>>& a, int n,
    vector<string>& ans, string move, vector<vector<int>>& vis,
                                    int di[], int dj[]){
    if(i==n-1 && j==n-1){
        ans.push_back(move);
        return;
    }
    string dir="DLRU";
    for(int ind=0; ind<4; ind++){
        int nexti = i+di[ind];
        int nextj = j+dj[ind];
        if(nexti>=0 && nextj>=0 && nexti<n && nextj<n &&
            a[nexti][nextj]==1 && !vis[nexti][nextj]){
            vis[i][j]=1;
            solve(nexti, nextj, a, n, ans, move+dir[ind], vis, di, dj);
            vis[i][j]=0;
        }
    }
}
int main(){
    int n=4;
    vector<vector<int>> m={{1,0,0,0},
                           {1,1,0,1},
                           {1,1,0,0},
                           {0,1,1,1}};
    vector<vector<int>> vis(n, vector<int>(n,0));
    int di[]={+1, 0, 0, -1};
    int dj[]={0, -1, 1, 0};
    vector<string> ans;
    solve(0,0,m,n,ans,"",vis,di,dj);
    if(ans.size()==0) cout<<-1;
    else
        for(auto i: ans) cout<<i<<endl;
}
```

② N Queen Problem :-

- Return all distinct solutions to the N-Queenr puzzle

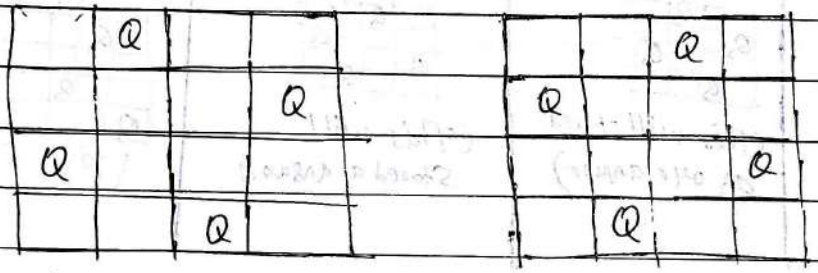
→ The n-queens puzzle is the problem of placing n queens on an n×n chessboard such that no two queenr attack each other.

→ Given an integer 'n', return all distinct solutions to the n-queens puzzle. you may return the answer in any order.

→ 'Q' and '.' both indicate a queen and an empty space.

Input : n = 4

O/P: [["..Q.","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]



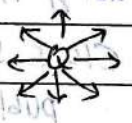
Input : n = 1

O/P : [["Q"]]

*Rules For n-Queens in chessboard

1. Every row should have one Queen
2. Every column should have one Queen
3. No two queens can attack each other

Queen attacks



* Algorithm

(1) Start in the leftmost column

(2) if all queens are placed
return true

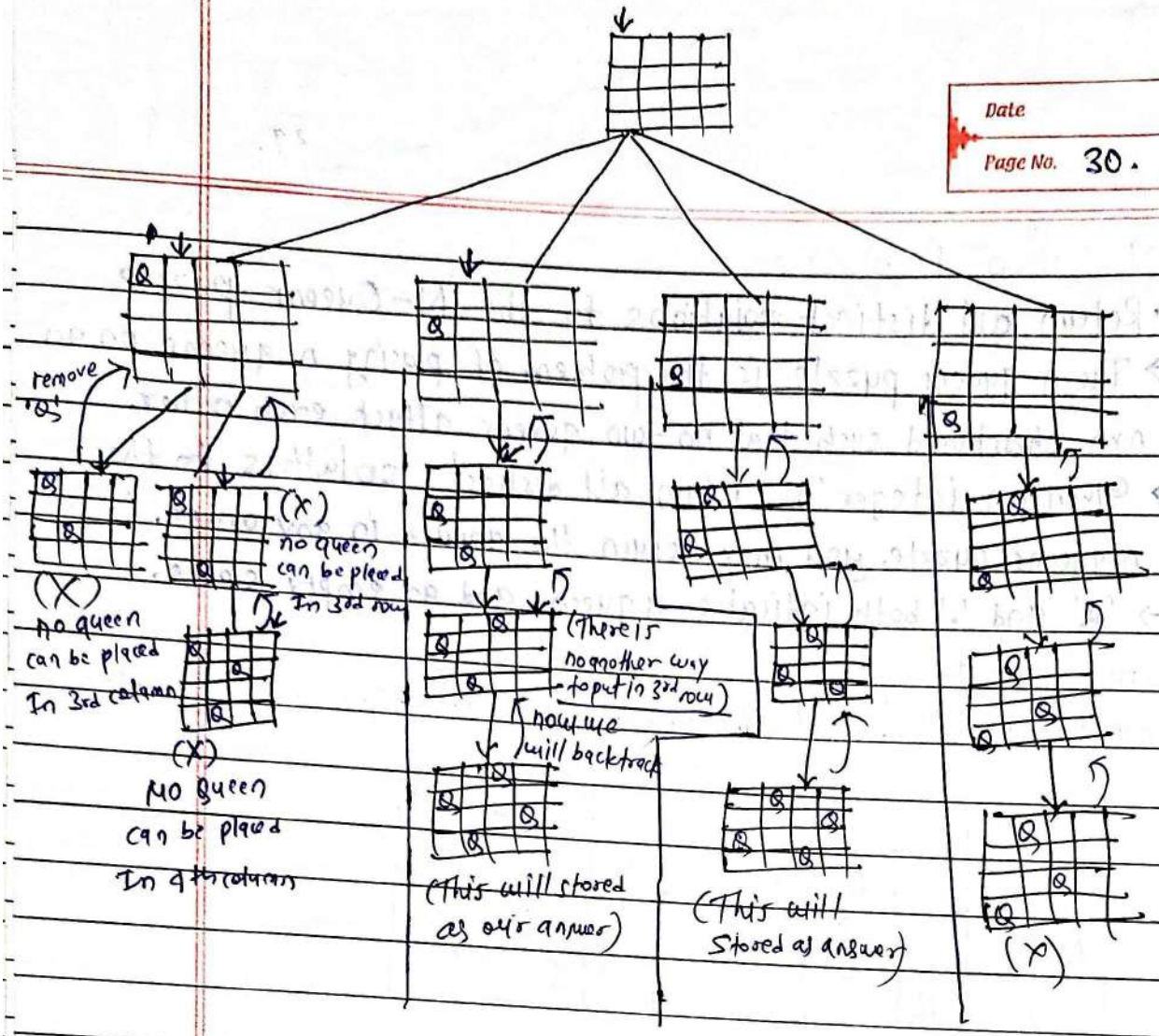
(3) Try all rows in the current column
Do following for every tried row.

(a) if the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution

(b) if it lead to solution return true

(c) if not unmark this [row, column] (backtrack) and go to step (a) to try other rows.

(4) if all rows have been try and nothing worked return false to trigger backtrack



Program :-

```
#include <bits/stdc++.h>
using namespace std;
class Solution{
    public:
    bool isSafe(int row, int col, vector<string> board, int n){
        //Check the row on left side
        for(int i=0; i<col; i++)
            if(board[row][i]=='Q') return false;
        //Check Upper diagonal on left side
        for(int i=row, j=col; i>=0 && j>=0; i--, j--)
            if(board[i][j]=='Q') return false;
        //Check Lower diagonal on left side
        for(int i=row, j=col; j>=0 && i<n; i++, j--)
            if(board[i][j]=='Q') return false;
        return true;
    }
```

```
public:
  void solve(int col, vector<string> &board, vector<vector<string>>
                                                &ans, int n){
    if(col==n){
      ans.push_back(board);
      return;
    }
    for(int row=0; row<n; row++){
      if(isSafe(row, col, board, n)){
        board[row][col]='Q';
        solve(col+1, board, ans, n);
        board[row][col]='.';
      }
    }
  }
public:
  vector<vector<string>> solveNQueens(int n){
    vector<vector<string>> ans;
    vector<string> board(n);
    string s(n,'.');
    for(int i=0; i<n; i++){
      board[i]=s;
    }
    solve(0, board, ans, n);
    return ans;
  }
};
int main(){
  int n=4;
  Solution obj;
  vector<vector<string>> ans = obj.solveNQueens(n);
```

```
for(int i=0; i<ans.size(); i++){
    for(int j=0; j<ans[0].size(); j++){
        cout<<ans[i][j]<<endl;
    }
    cout<<endl;
}
return 0;
}
```

Time complexity: $O(n^3)$

Space complexity: $O(n^2)$

* Optimization in isSafe function

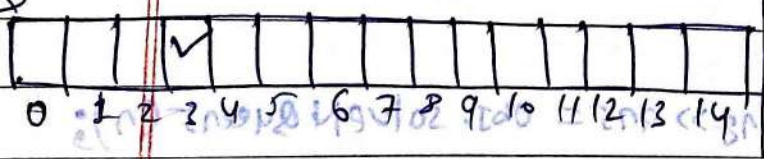
The idea is not to check every element in right and left diagonal instead use property of diagonals.

1. The sum of i and j is constant and unique for each right diagonal (i=row, j=col)
2. The difference of i and j is constant and unique for each left diagonal (i=row, j=col)

lower upper diagonal

	0	1	2	3	4	5	6	7
0	0	1	2	(3)	4	5	6	7
1	1	2	(3)	4	5	6	7	8
2	2	(3)	4	5	6	7	8	9
3	(3)	4	5	6	7	8	9	10
4	4	5	6	7	8	9	10	11
5	5	6	7	8	9	10	11	12
6	6	7	8	9	10	11	12	13
7	7	8	9	10	11	12	13	14

- we are filling as `row+co` → hash function
- for n*n grid maxm val is `2*n-1`
- means hash size is 15

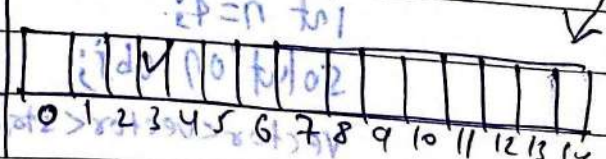


upper lower diagonal

	0	1	2	3	4	5	6	7
0	7	8	9	10	11	12	13	14
1	6	7	8	9	10	11	12	13
2	5	6	7	8	9	10	11	12
3	4	5	6	7	8	9	10	11
4	(3)	4	5	6	7	8	9	10
5	2	(3)	4	5	6	7	8	9
6	1	2	(3)	4	5	6	7	8
7	0	1	2	(3)	4	5	6	7

→ hash function

- Filling as. `(n-1)+(row-col)`
- maxm val. `2*n-1`
- means hash size is 15



Program :-

Date
Page No. 33,

```
#include <bits/stdc++.h>
using namespace std;
class Solution{
 public:
  void Solve(int col, vector<string>& board, vector<vector<string>>& ans,
             vector<int>& leftrow, vector<int>& upperDiagonal,
             vector<int>& lowerDiagonal, int n){
    if(col==n){
      ans.push_back(board);
      return;
    }
    for(int row=0; row<n; row++){
      if(leftrow[row]==0 && lowerDiagonal[row+col]==0 &&
                upperDiagonal[n-1+col-row]==0){
        board[row][col]='Q';
        leftrow[row]=1;
        lowerDiagonal[row+col]=1;
        upperDiagonal[n-1+col-row]=1;
        Solve(col+1, board, ans, leftrow, upperDiagonal, lowerDiagonal, n);
        board[row][col]='.';
        leftrow[row]=0;
        lowerDiagonal[row+col]=0;
        upperDiagonal[n-1+col-row]=0;
      }
    }
  }
 public:
  vector<vector<string>> solveNQueens(int n){
    vector<vector<string>> ans;
    vector<string> board(n);
    string s(n,'.');
    for(int i=0; i<n; i++){
      board[i]=s;
    }
```

```
vector<int> leftrow(n,0), upperDiagonal(2*n-1,0), lowerDiagonal
                                        (2*n-1,0);
solve(0, board, ans, leftrow, upperDiagonal, lowerDiagonal, n);
return ans;
}
};
int main(){
int n=4;
solution obj;
vector<vector<string>> ans = obj.solveNQueens(n);
for(int i=0; i<ans.size(); i++){
for(int j=0; j<ans[0].size(); j++){
cout<<ans[i][j]<<endl;
}
cout<<endl;
}
return 0;
}
```

Time complexity: $O(N^2)$

Space complexity: $O(N^2)$

(3) Sudoku Solver

Given a 9x9 incomplete sudoku, solve it such that it becomes valid sudoku. valid sudoku has following properties.

1. All the rows should be filled with numbers (1-9) exactly once.
2. All the columns should be filled with numbers (1-9) exactly once.
3. Each 3x3 submatrix should be filled with numbers (1-9) exactly once.

Note :- character '.' indicate empty cell.

Ex:- Input

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

Output

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

Note :- The can exist many solution. The above solution is one of them.

Intution :-
since we have to fill the empty cells with available possible numbers and we can also have multiple solutions, the main instruction is to try every possible way of filling the empty cells. And the correct way to try all possible solutions is to use recurrion. In each cell to the recursive function, we just try all the possible numbers for a particular cell and transfer the updated board to the next recurrive call.

Approach :-
- Let's see the step by step approach. our main recursive function (solve()) is going to just do a plain matrix traversal of the sudoku board. when we find an empty cell, we pause and try to put all available numbers (1-9) in that particular empty cell.
- we need another loop to do that. But wait, we forgot one thing. the board has to satisfy all the condition, right? so, For that we have another function (isValid()) which will check whether the number we have inserted into that empty cell will not violate any condition.
- if it is violating, we try with the next number. if it is not, we call the same function recursively, but this time with the updated state of the board. Now, as usual it tries to fill the remaining cells in the board in the same way.
- if at any point ~~[illegible]~~ we cannot insert any number from 1-9 in a particular cell, it means the current state of the board is wrong and we need to backtract. and we need to return false to let the parent function know that we cannot fill this way.
- If a recursive call return true, we can assume that we found one possible way of filling and we simply do an early return.

Validating board

- Now, After determining a number for a cell (at i'th row, j'th col), we try to check the validity. since we have 3 condition to check. we can do this withing a single loop from 0 to 8 to check the validy as.
- we loop from 0 to 8 and check the values.

(1) board[i][col], 1st condition, checking in row
(2) board[row][i], 2nd condition, checking in col
(3) For row. expression $(3*(row/3) + i/3)$ evaluates to the row for col. expression $(3*(col/3) + i\%3)$

- For example, if row=5 and col=3 the cells visited ar.

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

, if i=0, row= $3*(3/3)+0/3$ = (3, col=3)
i=1, row= $3*(3/3)+1/3$ = (3, col=4)
i=2, row= (3, col=5)
i=3, row= (4, col=3)
i=4, row= (4, col=4)
i=5, (row=4, col=5)
i=6, (row=5, col=3)
i=7, (row=5, col=4)
i=8, (row=5, col=5)

Program:-

```
#include <iostream>
#include <vector>
using namespace std;
bool isValid (vector<vector<char>>& board, int row, int col, char c){
    for (int i=0; i<9; i++){
        if (board[i][col]==c)
            return false;
        if (board[row][i]==c)
            return false;
        if (board[3*(row/3)+i/3][3*(col/3)+i%3]==c)
            return false;
    }
    return true;
}
```

```
bool solve (vector<vector<char>>& board) {
    for(int i=0; i<board.size(); i++){
        for(int j=0; j<board[0].size(); j++){
            if (board[i][j]=='.'){
                for(char c='1'; c<='9'; c++){
                    if(isValid(board, i, j, c)){
                        board[i][j]=c;
                        if(solve(board))
                            return true;
                        else
                            board[i][j]='.';
                    }
                }
                return false; //we have no choice to put any 1-9
            }
        }
    }
    return true;
}
int main(){
    vector<vector<char>> board{
        {'9','5','7','.','1','3','.','8','4'},
        {'4','8','3','.','5','7','1','.','6'},
        {'.','1','2','.','4','9','5','3','7'},
        {'1','7','.','3','.','4','9','.','2'},
        {'5','.','4','9','7','.','3','6','.'},
        {'.','-','-','-','-','-','-','-','-'}};
    solve(board);
    for(int i=0; i<9; i++){
        for(int j=0; j<9; j++)
            cout<<board[i][j]<<" ";
        cout<<"\n";
    }
    return 0;
}
```

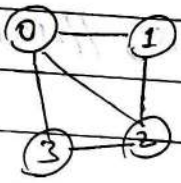
Time complexity: $O(9^{(n^2)})$, in the worst case, for each cell in the n^2 board, we have 9 possible numbers.
Space complexity: $O(1)$, $O(1)$, since we are filling the given board itself, there is no extra space required.

(4.) M-Coloring Problem :—

Given an undirected graph and a number m, determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with the same color.

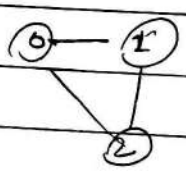
Ex:-

Input: N=4, M=3, E=5
Edges[] = {
(0,1),
(1,2),
(2,3),
(3,0),
(0,2)
}



O/P: 1, i.e, Possible

I/P: N=3, M=2, E=3
Edges[] = {
(0,1),
(1,2),
(0,2)
}



O/P: 0, i.e, not possible

Approach:-

Basically starting from vertex 0. color one by one the different vertices.

Base condition:-

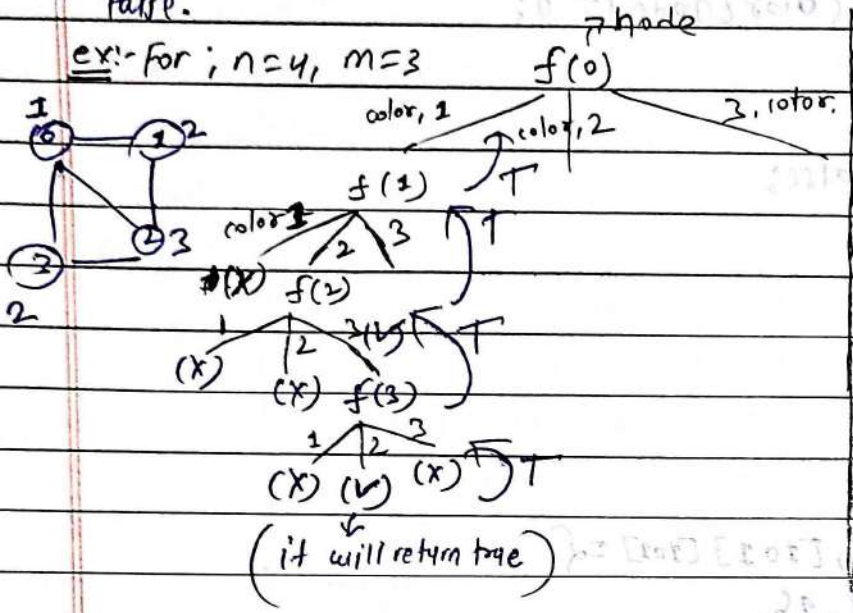
If I have colored all the N nodes, return true.

Recurrion:-

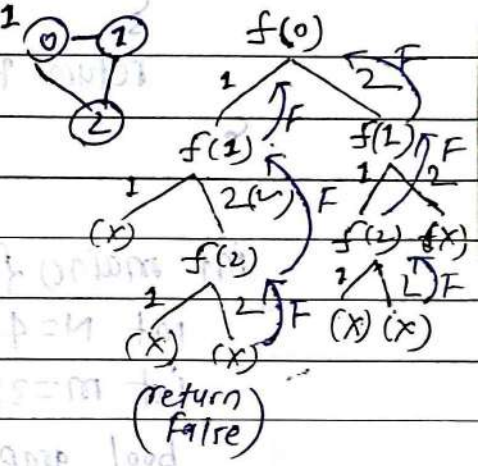
- Trying every color from 1 to m with the help of a for loop
- isSafe function returns true if it is possible to color it with that color i.e none of the adjacent node have the same color.
- color it with color i then call the recursive function for the

next node if it returns true we will return true
- And if it is false then take off the color.
- now if we have tried out every color from 1 to m and it was not possible to color it with any of the m colors then return false.

ex:- For ; n=4, m=3



ex:- For, n=3, m=2

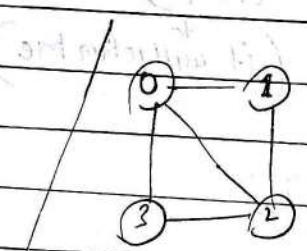


Program:-

```
#include <bits/stdc++.h>
using namespace std;
bool isSafe(int node, int color[], bool graph[101][101], int n, int col){
    for(int k=0; k<n; k++){
        if(k!=node && graph[k][node]==1 && color[k]==col){
            return false;
        }
    }
    return true;
}
bool solve(int node, int color[], int m, int N, bool graph[101][101]){
    if(node==N){
        return true;
    }
```

```
for(int i=1; i<=m;i++){
    if(isSafe(node, color, graph, N, i)){
        color[node]=i;
        if(solve(node+1, color, m, N, graph)) return true;
        color[node]= 0;
    }
}
return false;
}

int main(){
    int N=4;
    int m=3;
    bool graph[101][101]={
        {0,1,1,1},
        {1,0,1,0},
        {1,1,0,1},
        {1,0,1,0},
        (0,2) };
    int color[N]={0};
    cout<< solve( 0, color, m, N, graph) <<endl;
}
```



Time complexity: O(N^M) (N raised to M)
Space complexity: O(N)

(5) The knight's tour Problem :-

- Given a N*N board with the knight placed on the first block of an empty board. Moving according to the rules of chess knight must visit each square exactly once. Print the order of each cell in which they are visited.

* Example :-

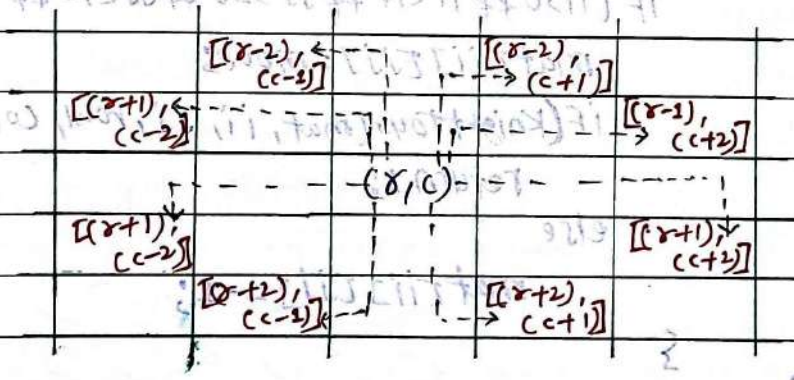
Input : N=8

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

* Rules of chess knight for every cell (r,c) :-

- For every cell (r,c) we have 8 possibilities.



The knight can move in the following specified block.

raw[8] = {2, 1, -1, -2, -2, -1, 1, 2};

column[8] = {1, 2, 2, 1, -1, -2, -2, -1};

* Algorithm :-

- Travel the matrix to only valid cell in the row, column vector Given
- if the step is valid then mark the current move in that cell of the matrix
- if the current cell is not valid or does not help to complete traversal of the matrix then recur back and change the current value of the cell which was updated before the recursive step and goes for other cell given co-ordinate in (row, column) vector
- Recur until move equals to NXN (For zero index move NXN-1)

* Code :-

```
#include <bits/stdc++.h>
using namespace std;
int KnightTour(vector<vector<int>> &mat, int i, int j, int row[8],
               int col[8], int move, int N){
    if(move==N*N)
        return 1;
    int ii, jj;
    for(int k=0; k<8; k++){
        ii = i + row[k];
        jj = j + col[k];
        if(ii>0 && ii<N && jj>=0 && jj<N && mat[ii][jj]==-1){
            mat[ii][jj]=move;
            if(KnightTour(mat, ii, jj, row, col, move+1, N)==1)
                return 1;
            else
                mat[ii][jj]=-1;
        }
    }
    return 0;
}

int main(){
    int N;
```

```
cin>>N;
vector<vector<int>> mat(N, vector<int>(N,-1));
int row[8]={2, 1, -1, -2, -2, -1, 1, 2};
int col[8]={1, 2, 2, 1, -1, -2, -2, -1};
mat[0][0]=0
if(KnightTour(mat, 0, 0, row, col, 1, N)){
    for(auto i: mat){
        for(auto j: i)
            cout<<j<<" ";
        cout<<"\n";
    }
}
else
  cout<<"-1";
return 0;
}
```

Time :- $O(8^{N^2})$
Space :- $O(N^2)$

(6) Letter combination of a phone number

- Given a digit string (digits=2-9), Generate all possible letter combinations that the number could represent. A mapping of digits to letter (just like on the telephone buttons) is given below.

1	2 abc	3 def
4 ghi	5 jkl	6 mno
7 pqrs	8 tuv	9 wxyz
*	0	#

- The digit 0 maps to '0' itself
- The digit 1 maps to '1' itself

Date 1
Page No. 44

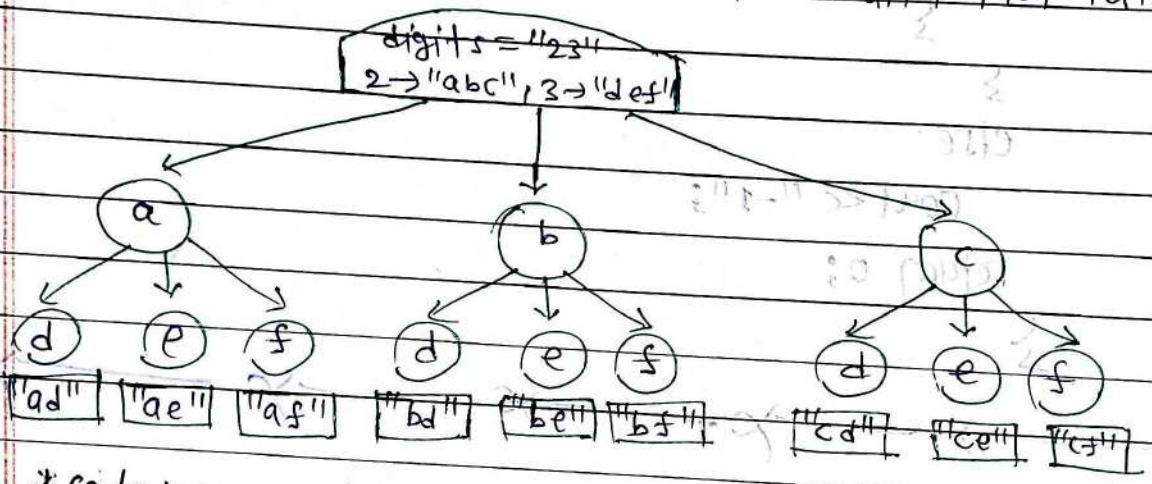
*Examples:-

• Input : "23"
Output : ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

• Input : "2"
Output : ["a", "b", "c"]

*Approach: Backtracking

• The idea is to consider a digit as the starting point and generate all possible combination with that letter.



*code:

```
#include <bits/stdc++.h>
using namespace std;
void findAll (map<char,string> mp, string digits, string temp,
              vector<string> &ans, int i){
    if (i>=digits.size()){
        ans.push_back(temp);
        return;
    }
    char c = digit[i];
    for(auto ch:mp[c]){
        temp.push_back(ch);
```

```
        FindAll(mp, digits, temp, ans, i+1);
        temp.pop_back();
    }
}
int main(){
    string digits;
    cin >> digits;
    map<char, string> mp{
        {'0', "0"},
        {'1', "1"},
        {'2', "abc"},
        {'3', "def"},
        {'4', "ghi"},
        {'5', "jkl"},
        {'6', "mno"},
        {'7', "pqrs"},
        {'8', "tuv"},
        {'9', "wxyz"},
    };
    string temp="";
    vector<string> ans;
    if(digit.size()==0)
        cout<<"";
    FindAll(mp, digits, temp, ans, 0);
    for(auto i:ans)
        cout<<i<<" ";
    return 0;
}
```

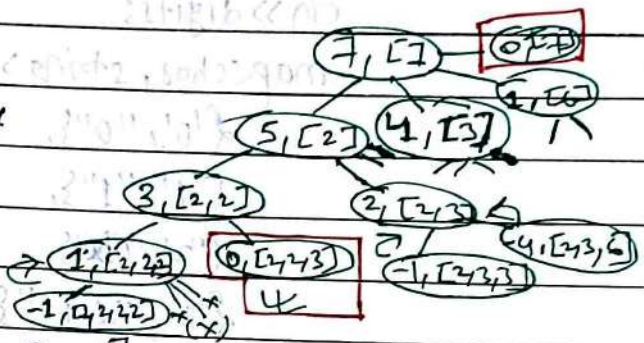
* Time complexity: $O(4^n)$
* Space complexity: $O(4^n)$

(7) Combination Sum :-

- You are given an arraylist of N distinct positive integers. You are also given a non-negative integer B.
- Your task is to find all unique combination in the array whose sum is equal to B. A number can be chosen any number of times from arraylist

*Example!

(1) Arraylist = [2, 3, 6, 7], B = 7
Output = [[2, 2, 3], [7]]



(2) Arraylist = [1, 2], B = 4
Output = [[1, 1, 1, 1], [1, 1, 2], [2, 2]]

* Approach :

- Since the problem is to get all the possible results, not the best or the number of result, thus we don't need to consider DP (Dynamic programming.
- The approach is to use naive-backtracking-based recursive approach

* Algorithm :

1. Sort the Array
2. remove duplicate
3. Then use recursion & backtracking to solve the problem
   (A.) If at any time sub-problem sum == 0 then add that array to the result (vector of vectors).
   (B.) Else if sum is -ve then ignore that sub-Problem
   (C.) Else insert the present index of that array to the current vector and call the Function with sum = sum - arr[idx] and index = index, then pop that element from the current index (backtrack) and call the function with sum = sum and index = index + 1

*Code:

```
void CombinationSum(vector<int> arr, vector<vector<int>> &ans,
                    vector<int> curr, int i, int sum, int n){
    if(sum==0){
        ans.push_back(curr);
        return;
    }
    if(i>=n or sum<0)
        return;
    curr.push_back(arr[i]);
    CombinationSum(arr, ans, curr, i, sum-arr[i], n);
    curr.pop_back();
    CombinationSum(arr, ans, curr, i+1, sum, n);
    return;
}

int main(){
    vector<int> arr = {1,2};
    int sum = 4;
    int n = 2;
    vector<vector> ans;
    vector<int> curr;
    CombinationSum(arr, ans, curr, 0, sum, n);
    for(auto i: ans){
        for(auto j:i)
            cout<<j<<" ";
        cout<<"\n";
    }
    return 0;
}
```

we can also use this as iterative see in next ques. combination sum II

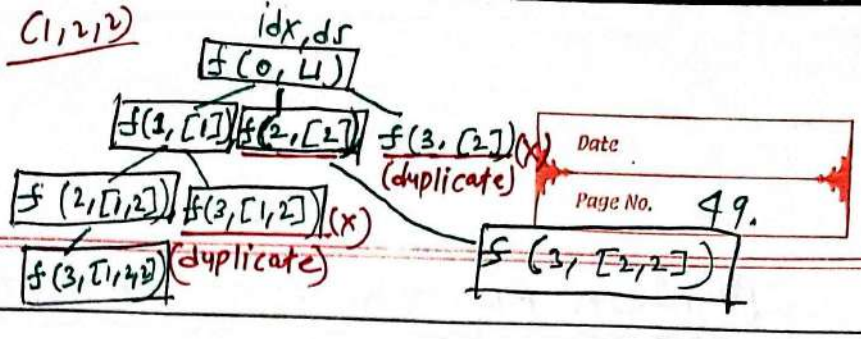
8. Combination Sum 2 :-

- Given a collection of candidate numbers (candidates) and a target number (target), find all unique combination in candidates where the candidate numbers sum to target.
- Each number in candidates may only be used once in the combination.

* Example :-
* Input : candidates = [10,1,2,7,6,1,5], target = 8
Output: [[1,1,6], [1,2,5], [1,7], [2,6]]

* Code :-

```
void FindCombination (vector<vector<int>> &ans, int i, int target,
                      vector<int> & curr, vector<int> arr){
    if (target==0){
        ans.push_back(curr);
        return;
    }
    for(int j=i; j<arr.size(); j++){
        if (arr[j] >target) return;
remove duplicate ⟹ if (j>i && arr[i]==arr[i-1]) continue;
        curr.push_back(arr[i]);
        FindCombination(ans, j+1, target-arr[i], curr, arr);
        curr.pop_back();
    }
}
int main(){
    vector<vector<int>>ans;
    vector<int> arr={10,1,2,7,6,1,5};
    vector<int> curr;
    sort(arr.begin(), arr.end());
    FindCombination(ans, 0, target, curr, arr);
    //Print the vector ans
    return 0;
}
```



Date
Page No. 49.

(9) Subset II

- Given an integer array nums that may contain duplicates, return all possible subsets (the power set).
- The solution set must not contain duplicate subsets return the solution in any order.

Example:-

Input: nums = [1,2,2]

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

code:-

```
class Solution{
public:
    void allsubs(vector<int>& nums, int curr, vector<int>& ds,
                 vector<vector<int>>& res){
        res.push_back(ds);
        for(int i=curr; i<nums.size(); i++){
            if(i>curr && nums[i]==nums[i-1])
                continue; //avoiding duplicates
            ds.push_back(nums[i]);
            allsubs(nums, i+1, ds, res);
            ds.pop_back();
        }
        return;
    }
    vector<vector<int>> subsetWithDup(vector<int>& nums){
        vector<vector<int>> res;
        vector<int> ds;
        sort(nums.begin(), nums.end());
        allsubs(nums, 0, ds, res);
        return res;
    }
};
```

Another Approach Using set (Naive Approach)

Code :-

```
class Solution{
    public:
        void allsubs(vector<int>& nums, int curr, vector<int>&ds,
                     set<vector<int>> &ans){
            if (curr >= nums.size()){
                ans.insert(ds);
                return;
            }
            int currval = nums[curr];
            ds.push_back(currval);
            allsubs(nums, curr+1, ds, ans);
            ds.pop_back();
            allsubs(nums, curr+1, ds, ans);
        }

        vector<vector<int>> subsetWithDup(vector<int>& nums){
            Set<vector<int>> ans;
            Vector<int> vec;
            Sort(nums.begin(), nums.end());
            allsubs(num, 0, vec, ans);
            Vector<vector<int>> res(ans.begin(), ans.end());
            return res;
        }
};
```

(10) Given n pairs of parantheses, write a function to generate all combinations of well-formed parantheses.

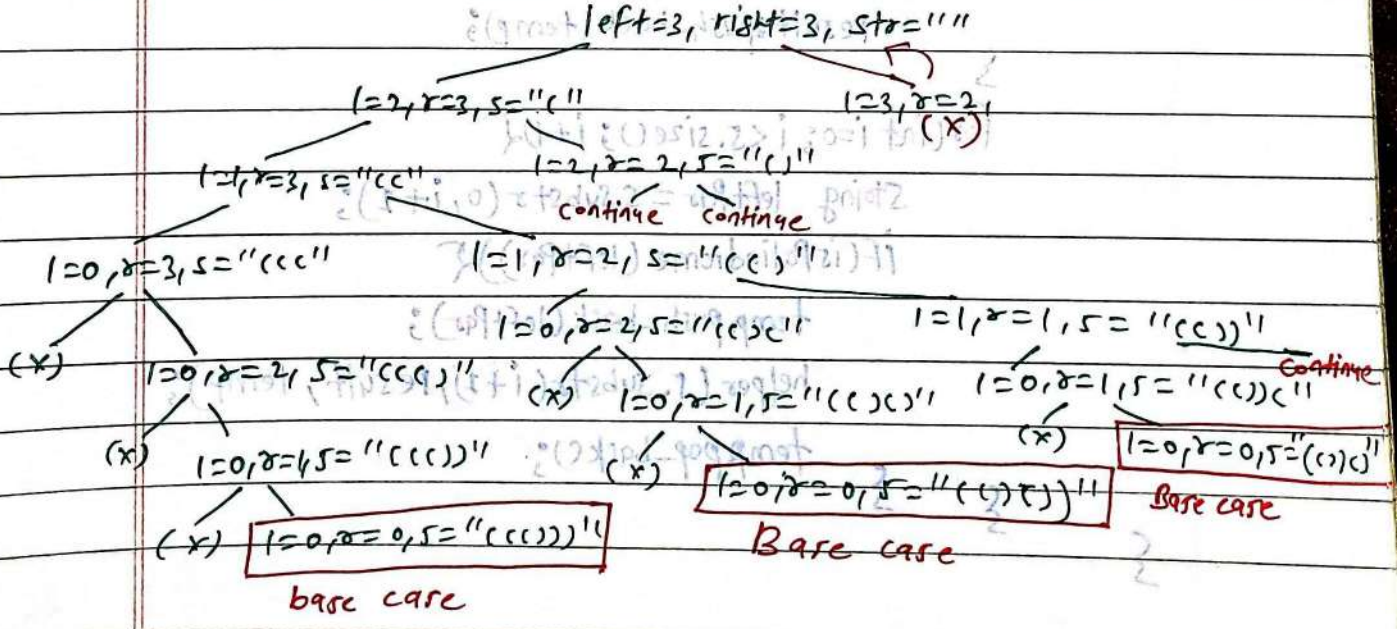
Example:

Input: n=3

Output: ["((()))","(()())","(())()","()(())","()()()"]

Code:

```
Class Solution{
    public:
        vector<string> generateParenthesis(int n){
            vector<string> res;
            helper(res, "", n, n);
            return res;
        }
        void helper(vector<string>&res, string str, int left, int right){
            if(left==0 && right==0){
                res.push_back(str);
                return;
            }
            if(left>0) helper(res, str+"(", left-1, right);
            if(right>left) helper(res, str+")", left, right-1);
        }
};
```



(11) Palindrome partition :—

Given a string s, partition s such that every substring of the partition is a palindrome. Return all possible palindrome partitioning of s.

- A palindrome string is a string that reads the same backward as forward.

*Example :-

Input: s = "aab"

Output: [["a","a","b"],["aa","b"]]

*code :—

```
Class Solution{
public:
bool isPalindrome(string s){
        int l=0;
        int r=s.size()-1;
        while (l<r){
            if (s[l++] != s[r--])
                return false
        }
        return true;
}
void helper(string s, vector<vector<string>> &result, vector<string>
            temp){
    if (s.size()==0){
            result.push_back(temp);
    }
    for(int i=0; i<s.size(); i++){
        string leftPar = s.substr(0,i+1);
        IF(isPalindrome(leftPar)){
            temp.push_back(leftPar);
            helper(s.substr(i+1), result, temp);
            temp.pop_back();
        }
}
```

```
vector<vector<string>> partition(string s){
    vector<vector<string>> result;
    vector<string> temp;
    helper(s, result, temp);
    return result;
}
};
```

result=[], temp=[], "aab"

i=0: r=[], t=["a"], "ab"

i=0: r=[], t=["a","a"], "b"

i=0: r=["a","a","b"], t=["a","a","b"], ""

i=1: r=[], t=["a","b"], "b" (x)

i=2: r=[], t=["aa"], "b"

i=0: r=["aa","b"], t=["aa","b"], ""

i=1: r=[], t=["a"], "" (∵ "ab" is not Palindrome ∴ don't add to t)

i=1: r=[], t=["aa"], "b" (x)

i=2: r=[], t=[], "" (∵ "aab" not a palindrome)

$\therefore$ r = [["a","a","b"], ["aa","b"]]