Class 71

29 April 2023 09:07

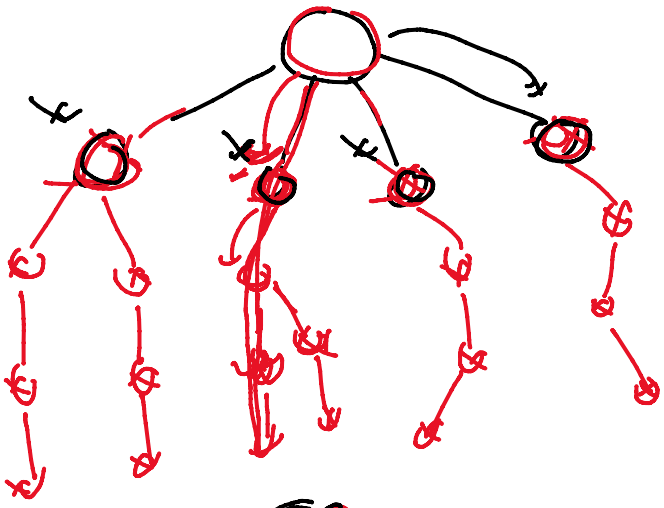
Class 71

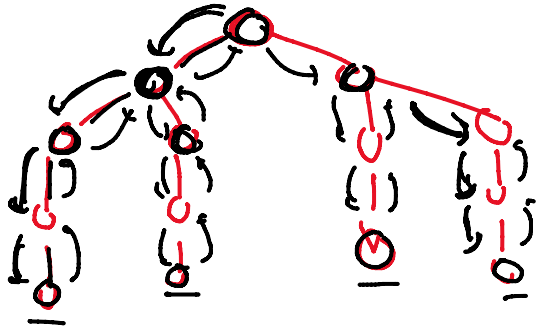
Backtracking → Brute force Approach type

Similar to **Recursion**

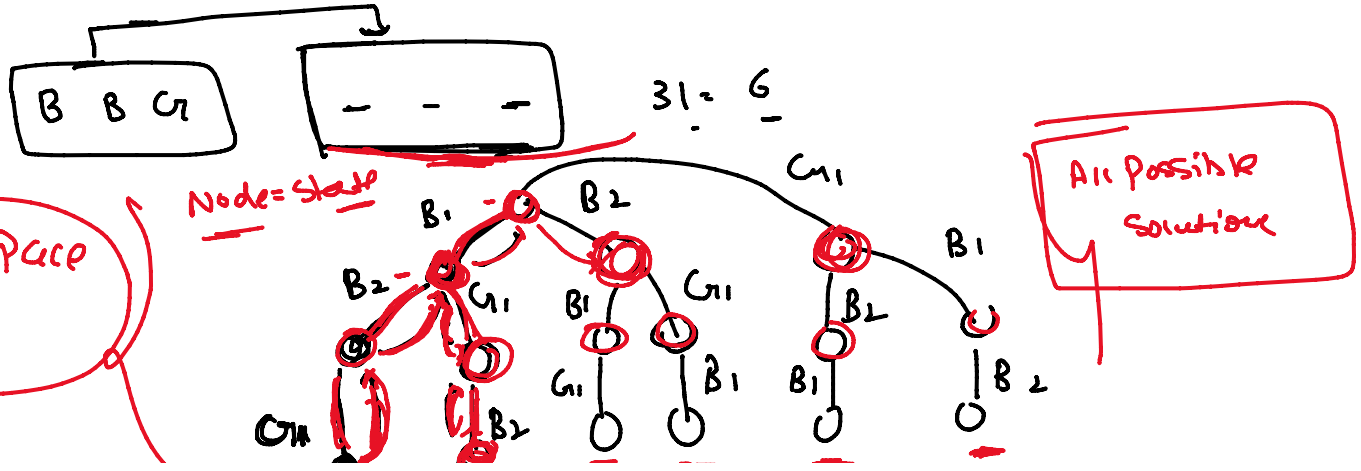
Greedy → Problem → **Multiple Solutions**

↳ we select 1 most **local optimal** solution





Path — 1 complete Path = 1 complete solution B1 B2 _ _



Recursion → [Single Algorithm multiple times]

fact

GCD

gcd (a, b)

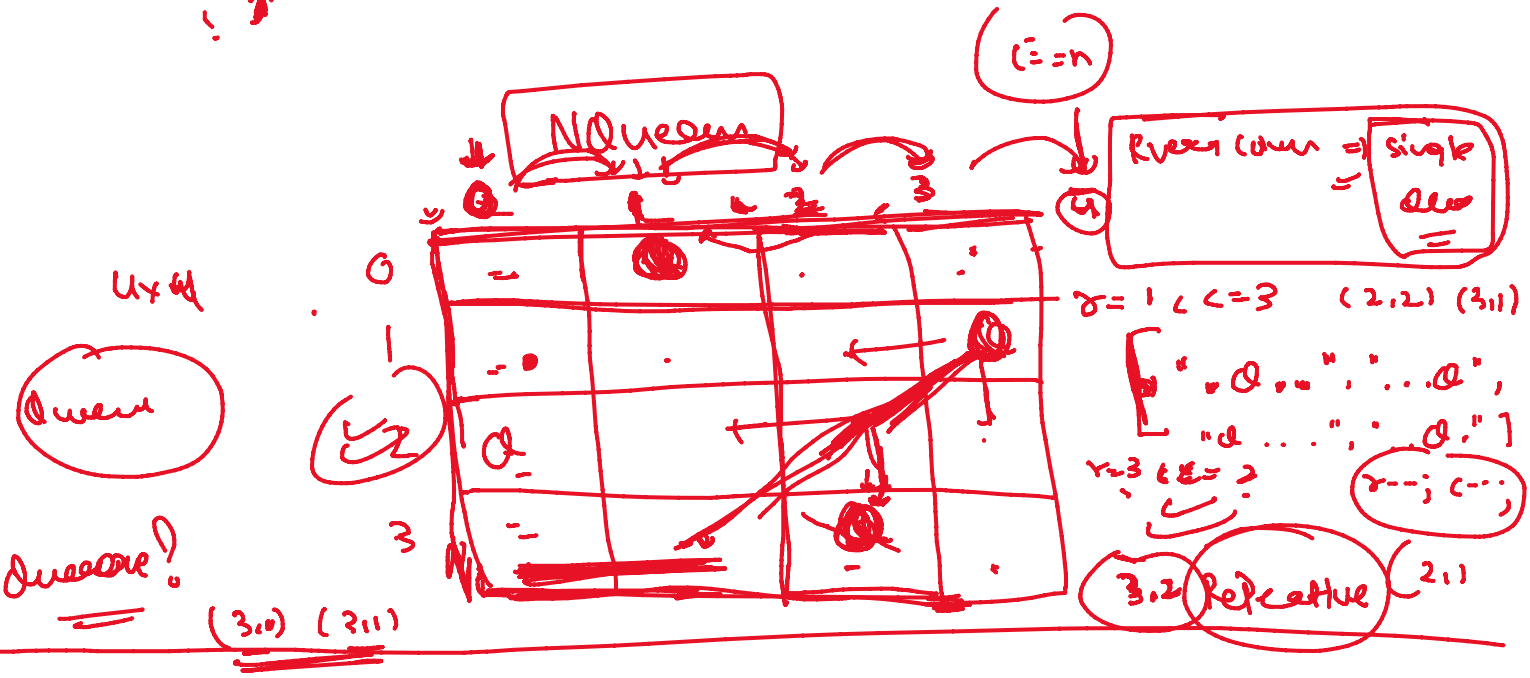
gcd (b, a%b)

gcd (a%b, b%(a%b))

f(n)

f(n-1)

f(n-2)



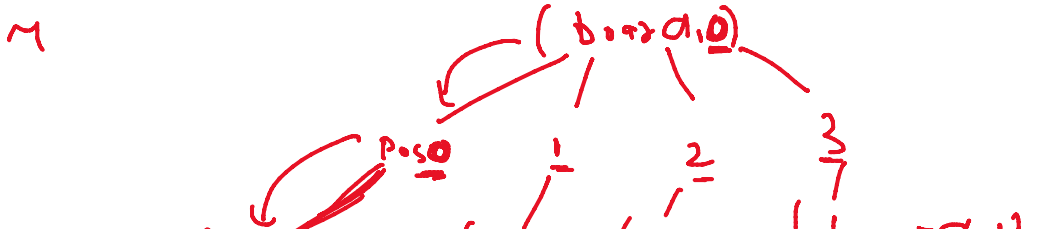

```
vector<vector<string>> SolveNQueen(int n)
{   vector<vector<char>> v(n, vector<char>(n));
    For (i=0; i<n; i++)
    {  For (j=0; j<n; j++)
           v[i][j]='.';
    }
```

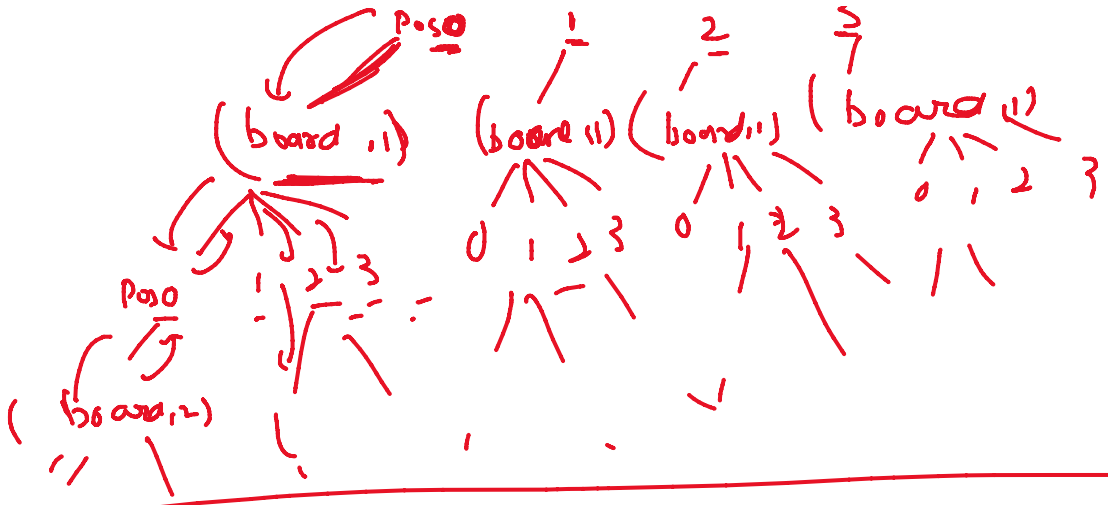
```
nQueen(0, n);

vector<vector<string>> ans;
void nQueen(int i, vector<vector<char>> &board)
{
    int n = board.size();

    if (i == n)
    {
        vector<string> temp;
        for(int k=0; k<n; k++)
        {
            string tempans = "";
            for(int l=0; l<n; l++)
            {
                tempans.push_back(v[k][l]);
            }
            temp.push_back(tempans);
        }
        ans.push_back(temp); return;
    }

    for(int j=0; j<n; j++)
    {
        if(check(board, j, i, n))
        {
            board[j][i] = 'Q';
            nQueen(i+1, board);
            board[j][i] = '.';
        }
    }
}
```

0 - n-1

Back tracking

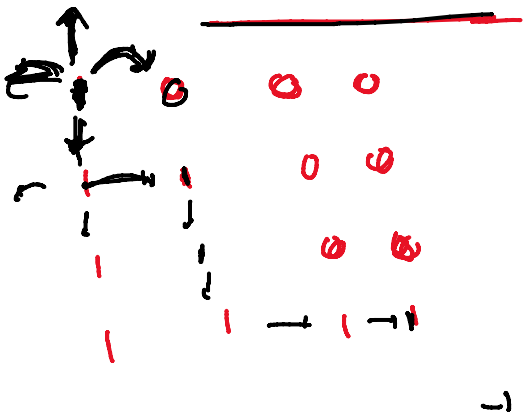
```
bool check (vector < vector < char >> & board, int i, int j, int n)
{   // Doesn't exist same row
    For ( k=0; k < j; k++ )
    { if (board[i][k] == '0' )
            return false
    }
    // upper diagonals and lower diagonals
    int r= i; int c = j;
    while ( r>=0 and c>=0)
    { if ( board [r][c] == '0' ) return false;
        r-- ;
        c-- ; ]
    }

    r=i; & c = j;
  while ( r<n and c >=0)
  {     if (board [r] [c]== '0') return false
        r+1;
  }     c--;

  return true;
}
```



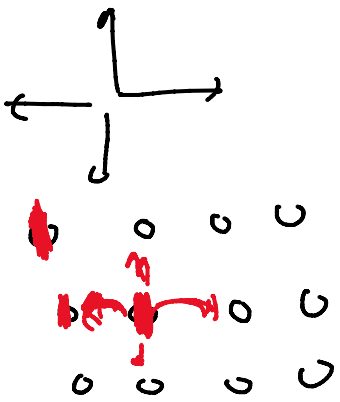


Rat in a Maze

Possible



All Possible Paths



```
int main()
{   int mat[N][N] =
    { { 1, 0, 0, 0},
      { 1, 0, 0, 0},
      { 1, 1, 0, 0},
      { 1, 1, 1, 1} };

    int sol[N][N] = { {0,0,0,0},
                      {0,0,0,0},
                      {0,0,0,0},
                      {0,0,0,0} };

if ( solve( mat, 0, 0, sol) == true)
```

```
if ( solve ( mat, 0,0, sol) == true)
{    // path exists

}
else {
// doesn't exist
}
```

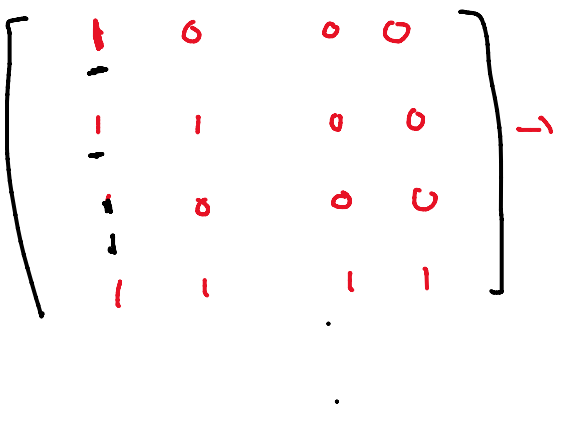
```
bool solve ( int mat[N][N], int x, int y, int sol[N][N])
{
    if ( x == N-1 && y == N-1 && mat[N-1][N-1] == 1)
    {   sol[x][y] = 1;
        return true;
    }
```

```
if ( safe(m, x, y) == true)
{   if ( sol[x][y] == 1)
        return false;
    sol[x][y] = 1;
```

```
if ( solve ( mat, x+1, y, sol) == true)
    return true;
if ( solve ( mat, x-1, y, sol) == true
    return true;
if ( solve ( mat, x, y+1, s) == true)
```

false

```
if (solve (mat, x, y+1, s) == true)
    return true;

if (solve (mat, x, y-1, s) == true)
    return true

sol[x][y] = 0;

return false;
}
return false;
```




```
bool safe (int m[N][N], int x, int y)
{
    if (x >= 0 and x < N and y >= 0 and y < N and m[x][y] == -1)
        return true;

    return false;
}
```

1 0 0 0
1 1 0 1