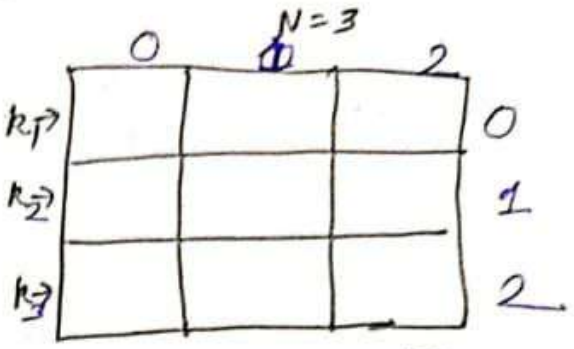
~~Backtracking~~ **Mixed Drill**

A func calling itself → Rec / Backtracking ← Implementing brute force using recursion

Framework : L C C M D

[Brute Force] → Explore everything valid

Q: Given NXN board. You have to place each knight in each row such that no two knights attack each other.



Appl:-

```
for(k1=0; k1<n; k1++)
    for(k2=0; k2<n; k2++)
        for(k3=0; k3<n; k3++)
```

levels of loops.

```
if[ abs(k1-k2)!=2 &&
    abs(k1-k3)!=1 && abs(k2-k3)!=2 ]
```

valid configuration.

Inc n → so inc. loops too.

variably nest loops → can be done using backtracking.

It is nthng but a variable size nesting of for loops as required.

- 5 things to decide Whenever you are writing Backtracking Problem

Level → Places where you need to decide → per row where would be the knight.

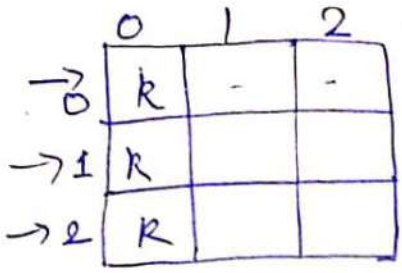
choice → choices are the place you are deciding → which column for this row

check → No If that is even valid → No 2 knights attack

Move → make the choice ~~Place knight~~, recurse & revert → Place knight

Decide (base case) → • correctness check → print all configuration
• print / collect / count
• return

choice = [0, 1, 2]



~~void~~ sec (row)
~~for~~ (col: choices)
if (check (row, col)) {

For the prev. loop, one level loop, the Two level loops are running, ~~for the prev loop, one level loop~~ once that is over when it comes back, the one level loop goes one step forward and goes to the second level loop again.

↓

~~What~~ Backtracking concept

Pseudocode :-

```
sec (row) → level
Decide [ if (row == N) {
           print (board)
           return;
         }
  for (col : choices)
    if (check (row, col)) {
Move → [   board [row] [col] = 'k'
           sec (row + 1);
           board [row] [col] = ' ';
    }
```

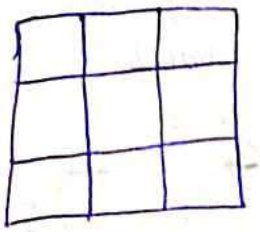
```
int n;
char board[15][15];
void reset(){
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            board[i][j]=' ';
        }
    }
}
void printer(){
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            cout<<board[i][j];
        }
        cout<<endl;
    }
    cout<<endl;
}
```

```
bool inside(int row,int col){
    if(row<n&&col<n&&row>=0&&col>=0)return 1;
    return 0;
}
int dx[]={-1,-2,-2,-1};
int dy[]={-2,-1,1,2};

int can_place(int row,int col){
    for(int k=0;k<4;k++){
        if(inside(row+dx[k],col+dy[k]) && board[row+dx[k]][col+dy[k]]=='K'){
            return 0;
        }
    }
    return 1;
}
```

```
// L - Level
void rec(int level){
    cout<<level<<endl;
    printer();
    // D - Decide
    if(level==n){
        return;
    }
    // C - Choices
    for(int col=0;col<n;col++){
        // C - Check
        if(can_place(level,col)){
            // M - Move
            // placing
            board[level][col]='K';
            // recurse.
            rec(level+1);
            // revert
            board[level][col]='.';

        }
    }
}
```

```
void solve(){
    cin>>n;
    rec(0);
}
```

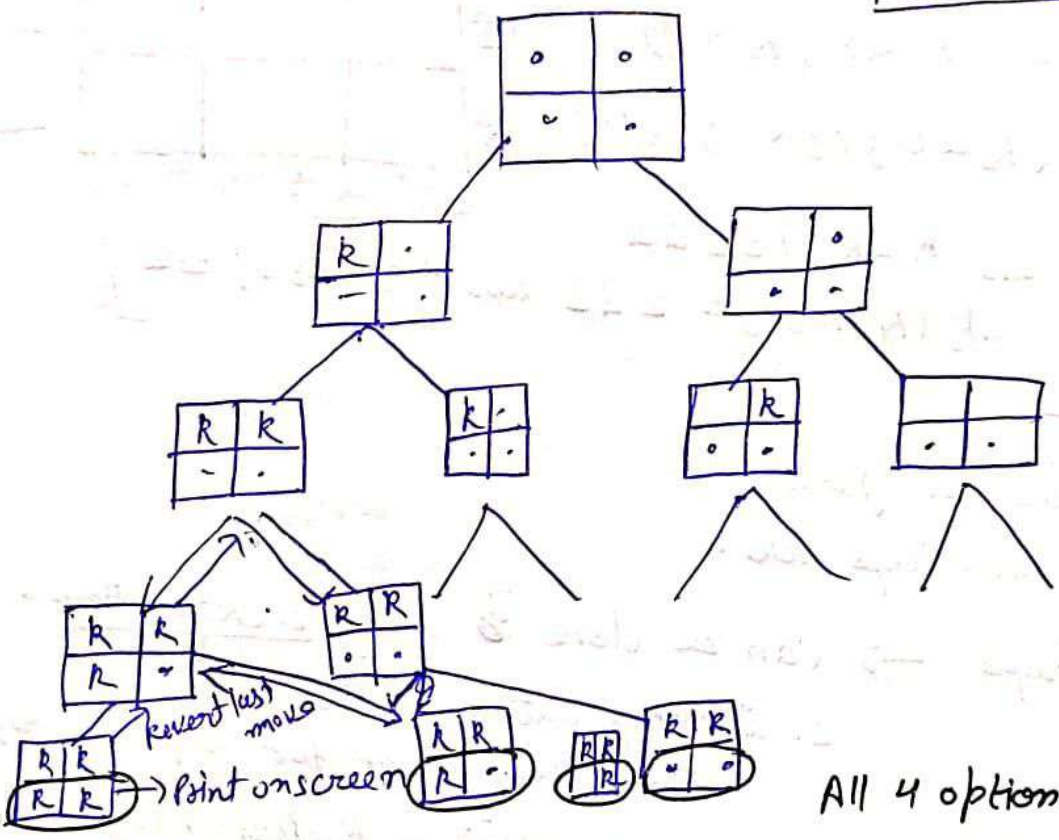
Q. Any cells can have knight or not. Generate all pssbl configuras where no two knights are attacking each other.



level → cell
choice → Place a knight or not (2^{N^2}) → cells
check → check safe
Moves → Place, recurse, revert
Desire → ✓✓

→ Per square two choices

Recursion Tree



All 4 options generated

```cpp
#include<bits/stdc++.h>
using namespace std;

int n;
// DS to store configuration.
char board[15][15];
void reset(){
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            board[i][j]='.';
        }
    }
}
void printer(){
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            cout<<board[i][j];
        }
        cout<<endl;
    }
    cout<<endl;
}

bool inside(int row,int col){
    if(row<n&&col<n&&row>=0&&col>=0)return 1;
    return 0;
}
int dx[]={-1,-2,-2,-1};
int dy[]={-2,-1,1,2};

int can_place(int row,int col){
    for(int k=0;k<4;k++){
        if(inside(row+dx[k],col+dy[k]) && board[row+dx[k]][col+dy[k]]=='K'){
            return 0;
        }
    }
    return 1;
}
```

```
// L - Level
void rec(int row,int col){
    // D - Decide
    if(row==n){
        printer();
        return;
    }
    // C - Choices
    // Place
    {
        // C - Check
        if(can_place(row,col)){
            // M - Move
            // placing
            board[row][col]='K';
            // recurse.
            if(col==n-1)rec(row+1,0);
            else rec(row,col+1);
            // revert
            board[row][col]='.';
        }
    }
    // Don't Place
    {
        // M - Move
        // recurse.
        if(col==n-1)rec(row+1,0);
        else rec(row,col+1);
    }
}


void solve(){
    cin>>n;
    reset();
    rec(0,0);
}
```

Q. Arr → [3 2 5]

Print all permutations of array.

① All elements are distinct ② There can be repeats

① Data structure

② LCMD

Level → Sal index

choice → which original

~~How~~ check → already taken

Move → Place, recc, revert

Decide → Print

Brute Force :- (By loops)

[_ _ _]
0 1 2

0 → for (int 0 = ...)

1 → for (ind = 1 ...)

2 → for (ind 2 ...)

[ind i ≠ id j]

```cpp
#include<bits/stdc++.h>
using namespace std;

int n;
// DS to store configuration.
int arr[15];
int perm[15];

map<int,int> freq;

// L - Level
void rec(int idx){
    // decide
    if(idx==n){
        for(int i=0;i<n;i++)cout<<perm[i]<<" ";
        cout<<endl;
        return;
    }
    // C
    auto temp = freq;
    for(auto v:temp){
        // Check
        if(v.second>0){
            // Move
            freq[v.first]--;
            perm[idx]=v.first;

            rec(idx+1);

            perm[idx]=-1;
            freq[v.first]++;
        }
    }
}
void solve(){
    cin>>n;
    for(int i=0;i<n;i++){
        cin>>arr[i];
        freq[arr[i]]++;
    }

    rec(0);
}
```