August 24, 2025

# BACKTRACKING FRAMEWORK & USAGE

* Difference between Recursion, Brute Force and Backtracking.

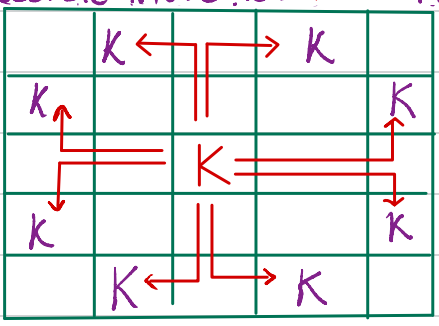
Recursion is a coding paradigm.

Brute Force refers to exploring all valid possibilities for a solution.

Backtracking is implementing Brute Force using Recursion.

Ques. Given a NXN Board, Place a Knight in each row. How many ways are there to place Knights with above condition given no two Knights attack each other.

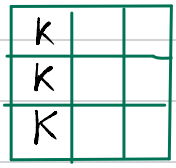
Knights Move in 'L' manner. The board shows possible move for 'K' to Place.




Example:

1. N = 3

```
for (K1=0; K1<N; K1++)
    for(K2=0; K2<N; K2++)
        for (K3=0; K3<N; K3++)
        { if(abs(K1-K2)!=2 &&
              abs (K1-K3)!=1 &&
              abs(K2-K3)!=2)
          ⇒ Valid Configuration
          else ⇒ Not Valid
        }
```




2. N = 4

What would change in previous solution so that it works for N=4.

- add 1 more loop, K4.
- add 1 more check for K4 Knight.

Increasing N is increasing the number of loop and adding check.

We now have to work on a solution where we can variably nest loops.

i.e, if n=5, it should have 5 loops etc.

This is exactly what we achieve using Backtracking.

Backtracking codes are nothing but variably sized nested loops.

Solution using the LCCMD frame work:

The for loops in our solution can be seen as levels.

1. levels → places where you need to decide.
   In this problem, levels → row
2. Choices → choices on the place you are deciding
   In this case, Choices → which column in current level (row)
3. Check → if the above choice made is valid
   In this case, Check → no 2 Knights attack

4. **Move** → Make the choice, Recurse and Revert.
   Move → Place the Knight
5. **Decide** → if its correct then print/collect/collect and return.
   Decide → Print.

Five things (level, choice, check, Move, Decide) that we have to know whenever we are working on a Backtracking code.

→ pseudocode

```
rec(row)        // row = level
    if (row == n){
        print(board); return;
    }
    for(col : choices)
        if(check(row, col))
        {
            board[row][col] = 'K'
            rec(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};

void 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;
}
```

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

Update : can place multiple Knights in a row.

How many ways to place the Knights such that no two Knights attack each other?

A valid configuration :

| | | |
|---|---|---|
| K | | K |
| | K | |
| K | | K |

L → each cell

Each row can no longer be a level because in every row, here we have to make multiple choice instead of one

C → Place a Knight or not.

C → Check Safe.

M → place, recurse(), revert

D → ✓✓

AlgoZenith

```
// 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[level][col]='.';
                  row

        }
    }
    // Don't Place
    {
        // M - Move
        // recurse.
        if(col==n-1)rec(row+1,0);
        else rec(row,col+1);
    }
}
```

```
// L - Level
void rec(int idx){
    // decide
    if(idx==n){
        for(int i=0;i<n;i++)cout<<perm[i]<<" ";
        cout<<endl;
        return;
    }
    // C
    for(int i=0;i<n;i++){
        // C
        if(taken[i]){
            // M
            taken[i]=1;
            perm[idx]=arr[i];

            rec(idx+1);

            perm[idx]=-1;
            taken[i]=0;
        }
    }
}
```

Using set :

```
set<int> available;


// L - Level
void rec(int idx){
    // decide
    if(idx==n){
        for(int i=0;i<n;i++)cout<<perm[i]<<" ";
        cout<<endl;
        return;
    }
    // C
    set<int> temp = available;
    for(auto v:temp){
        available.erase(v);
        perm[idx] = v;
        rec(idx+1);
        perm[idx] = -1;
        available.insert(v);
    }
}
```

Ques 2. Given an array, print all permutations of the array.

① All elements are distinct.

② There can be repeats.

Solution for ① :

Arr → [3, 2, 5]

level → solution index

Choice → which original element

Check → already taken or not

Move → place, rec(), revert

Decide → print

Solution for ② :

Above discussed solutions will fail for ②

```
map<int,int> freq; //Map Method to
                   stem duplicates in
                   choices.
// 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){
        if(v.second>0){
            freq[v.first]--;
            perm[idx]=v.first;
            rec(idx+1);
            perm[idx]=-1;
            freq[v.first]++;
        }
    }
}
```

## Ques 3. Generate all N sized array whose sum == K.

```
int perm[15];
int sum = 0;

void rec(int level){
    // pruning
    if(sum>k)return;

    if(level==n){
        if(sum==k){
            for(int i=0;i<n;i++)cout<<perm[i]<<" ";cout<<endl;
        }
        return ;
    }
    for(int i=1;i<=k;i++){
        sum +=i;
        perm[level]=i;
        rec(level+1);
        perm[level]=-1;
        sum-=i;
    }
}
```



## Homework

1. Print all subsets with duplicates
2. " " " without duplicates