# Recursion

Recursion is a fundamental concept in computer science and mathematics where a function calls itself in order to solve a problem. The main idea is to break down a problem into smaller, more manageable sub-problems that are similar in nature to the original problem. A recursive function typically has two main parts:

1. **Base Case**: A condition that stops the recursion, preventing an infinite loop. This is the simplest instance of the problem which can be solved directly.
2. **Recursive Case**: The part of the function where it calls itself with a modified argument, gradually moving towards the base case.

### Characteristics of Recursion

- **Self-Similarity**: The problem is broken down into smaller instances of the same problem.
- **Base Case(s)**: Essential to ensure that the recursion terminates.
- **Recursive Step(s)**: Defines how the function calls itself with modified parameters.

### When to Use Recursion
Recursion is particularly useful when a problem can naturally be divided into similar sub-problems, such as:

- Traversing hierarchical data structures like trees and graphs.
- Solving problems defined by recurrence relations.
- Implementing algorithms like quicksort, mergesort, and divide-and-conquer strategies.

### Advantages and Disadvantages

**Advantages**:
- Simplifies code for problems that can be expressed naturally with recursive structure.
- Often leads to more elegant and easier-to-read solutions for complex problems.

**Disadvantages**:
- Can lead to excessive memory use and stack overflow if the recursion depth is too large.
- Generally less efficient in terms of time and space compared to iterative solutions, due to function call overhead.

Understanding recursion is key to mastering algorithms and data structures in computer science, as it provides a powerful tool for solving a wide range of problems.

i) Fibonacci Number (LC 509)

The **Fibonacci numbers**, commonly denoted `F(n)` form a sequence, called the **Fibonacci sequence**, such that each number is the sum of the two preceding ones, starting from `0` and `1`. That is,

```
F(0) = 0, F(1) = 1
F(n) = F(n - 1) + F(n - 2), for n > 1.
```

Given `n`, calculate `F(n)`.

**Example 1:**

```
Input: n = 2
Output: 1
Explanation: F(2) = F(1) + F(0) = 1 + 0 = 1.
```

```
class Solution {
    public int fib(int n) {

        if (n <= 1) {
            return n;
        }

        int fi = fib(n-1);
        int fj = fib(n-2);
        int res = fi + fj;
        return res;
    }
}
```

ii) Palindrome Partitioning (LC 131)

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

**Example 1:**

```
Input: s = "aab"
Output: [["a","a","b"],["aa","b"]]
```

1) Combination (LC 77)

Given two integers `n` and `k`, return *all possible combinations of* `k` *numbers chosen from the range* `[1, n]`.

You may return the answer in **any order**.

**Example 1:**

```
Input: n = 4, k = 2
Output: [[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]
Explanation: There are 4 choose 2 = 6 total combinations.
Note that combinations are unordered, i.e., [1,2] and [2,1] are considered to be the same combination.
```

```
class Solution {

    public void f(int idx, int[] nums, List<List<Integer>> res, ArrayList<Integer> sub, int k) {

        if (sub.size() == k) {
            res.add(new ArrayList<>(sub));
            return;
        }

        if (idx >= nums.length) {
            return;
        }

        sub.add(nums[idx]);
        f(idx + 1, nums, res, sub, k);
        sub.remove(sub.size() - 1);
        f(idx + 1, nums, res, sub, k);
    }

    public List<List<Integer>> combine(int n, int k) {

        int[] nums = new int[n];

        for (int i = 1; i < n + 1; i++) {
            nums[i-1] = i;
        }

        List<List<Integer>> res = new ArrayList<>();
        ArrayList<Integer> sub = new ArrayList<>();

        f(0, nums, res, sub, k);

        return res;

    }
}
```

2) Combination Sum (LC 39)

Given an array of **distinct** integers `candidates` and a target integer `target`, return *a list of all* ***unique combinations*** *of* `candidates` *where the chosen numbers sum to* `target`. You may return the combinations in **any order**.

The **same** number may be chosen from `candidates` an **unlimited number of times**. Two combinations are unique if the frequency of at least one of the chosen numbers is different.

The test cases are generated such that the number of unique combinations that sum up to `target` is less than `150` combinations for the given input.

**Example 1:**

```
Input: candidates = [2,3,6,7], target = 7
Output: [[2,2,3],[7]]
Explanation:
2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used multiple times.
7 is a candidate, and 7 = 7.
These are the only two combinations.
```

```
class Solution {

    public void f(int idx, List<List<Integer>> res, int[] can, ArrayList<Integer> sub, int tar) {

        if (tar == 0) {
            res.add(new ArrayList<>(sub));
            return;
        }

        if (tar < 0 || idx == can.length) {
            return;
        }

        sub.add(can[idx]);
        f(idx, res, can, sub, tar - can[idx]);
        sub.remove(sub.size() - 1);
        f(idx + 1, res, can, sub, tar);
    }

    public List<List<Integer>> combinationSum(int[] candidates, int target) {

        List<List<Integer>> res = new ArrayList<>();
        ArrayList<Integer> sub = new ArrayList<>();

        f(0, res, candidates, sub, target);

        return res;
    }
}
```

3) Combination SumII (LC 40)

Given a collection of candidate numbers ( `candidates` ) and a target number ( `target` ), find all unique combinations in `candidates` where the candidate numbers sum to `target` .

Each number in `candidates` may only be used **once** in the combination.

**Note:** The solution set must not contain duplicate combinations.

**Example 1:**

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

```
class Solution {

    public void f(int ind, int[] can, int tar, ArrayList<Integer> haf, List<List<Integer>> res){

        if (tar == 0){
            res.add(new ArrayList<>(haf));
            return;
        }

        for (int i = ind; i < can.length; i++) {

            if (i > ind && can[i] == can[i - 1]) {
                continue;
            }

             if (tar < can[i]) {
                break;
            }

            haf.add(can[i]);
            f(i + 1, can, tar -can[i], haf, res);
            haf.remove(haf.size() - 1);
        }
    }

    public List<List<Integer>> combinationSum2(int[] candidates, int target) {

        List<List<Integer>> res = new ArrayList<>();

        ArrayList<Integer> haf = new ArrayList<>();

        for (int i = 0; i < candidates.length; i++) {

        for (int j = 0; j < candidates.length; j++){

            if (candidates[i] < candidates[j]){

                int temp = candidates[i];
                candidates[i] = candidates[j];
                candidates[j] = temp;
            }
        }
    }
```

```
        f(0, candidates, target, haf, res);

        return res;

    }
}
```

4) Combination SumIII (LC 216)

Find all valid combinations of `k` numbers that sum up to `n` such that the following conditions are true:

- Only numbers `1` through `9` are used.
- Each number is used **at most once**.

Return *a list of all possible valid combinations*. The list must not contain the same combination twice, and the combinations may be returned in any order.

**Example 1:**

```
Input: k = 3, n = 7
Output: [[1,2,4]]
Explanation:
1 + 2 + 4 = 7
There are no other valid combinations.
```

```
class Solution {

    public void f(int ind, int k, int tar, ArrayList<Integer> haf, List<List<Integer>> res, int[] nums) {

        if (tar == 0 && haf.size() == k) {
            res.add(new ArrayList<>(haf));
            return;
        }

        if (ind == nums.length) {
            return;
        }

        haf.add(nums[ind]);

        f(ind + 1, k, tar - nums[ind], haf, res, nums);

        haf.remove(haf.size() - 1);

        f(ind + 1, k, tar, haf, res, nums);
    }

    public List<List<Integer>> combinationSum3(int k, int n) {

        int nums[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};

        ArrayList<Integer> haf = new ArrayList<>();

        List<List<Integer>> res = new ArrayList<>();

        f(0, k, n, haf, res, nums);

        return res;
    }
}
```

5) Combination SumIV (LC 377)

Given an array of **distinct** integers `nums` and a target integer `target`, return *the number of possible combinations that add up to* `target`.

The test cases are generated so that the answer can fit in a **32-bit** integer.

**Example 1:**

```
Input: nums = [1,2,3], target = 4
Output: 7
Explanation:
The possible combination ways are:
(1, 1, 1, 1)
(1, 1, 2)
(1, 2, 1)
(1, 3)
(2, 1, 1)
(2, 2)
(3, 1)
Note that different sequences are counted as different combinations.
```

```
class Solution {

    public int Solve(int[] nums, int target, int[] dp) {

        if (target == 0) {
            return 1;
        }

        if (target < 0) {
            return 0;
        }

        if (dp[target] != -1) {
            return dp[target];
        }

        int res = 0;

        for (int i = 0; i < nums.length; i++) {
            res += Solve(nums, target - nums[i], dp);
        }

        dp[target] = res;
        return dp[target];
    }

    public int combinationSum4(int[] nums, int target) {

        int[] dp = new int[target + 1];

        Arrays.fill(dp, -1);

        return Solve(nums, target, dp);
    }
}
```

6) Subset (LC 78)

Given an integer array `nums` of **unique** elements, return *all possible subsets (the power set)*.

The solution set **must not** contain duplicate subsets. Return the solution in **any order**.

**Example 1:**

```
Input: nums = [1,2,3]
Output: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]
```

```
class Solution {

    public void f(int ind, List<List<Integer>> res, ArrayList<Integer> haf, int[] nums, int n) {
        if (haf.size() == n) {
            return;
        }

        if (ind == n) {
            return;
        }

        haf.add(nums[ind]);
        res.add(new ArrayList<>(haf));
        f(ind + 1, res, haf, nums, n);
        haf.remove(haf.size() - 1);
        f(ind + 1, res, haf, nums, n);
    }

    public List<List<Integer>> subsets(int[] nums) {

        List<List<Integer>> res = new ArrayList<>();
        ArrayList<Integer> haf = new ArrayList<>();

        res.add(new ArrayList<>(haf));
        f(0, res, haf, nums, nums.length);

        return res;
    }
}
```

7) SubsetII (LC 90)

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 1:**

```
Input: nums = [1,2,2]
Output: [[],[1],[1,2],[1,2,2],[2],[2,2]]
```

```
class Solution {

    public void f(int ind, List<List<Integer>> res, ArrayList<Integer> haf, int[] nums, int n) {

        if (haf.size() == n) {
            return;
        }

        for (int k = ind; k < n; k++) {
            if (k > ind && nums[k] == nums[k - 1]){
                continue;
            }

            haf.add(nums[k]);
            res.add(new ArrayList<>(haf));
            f(k + 1, res, haf, nums, n);
            haf.remove(haf.size() - 1);
        }
    }

    public List<List<Integer>> subsetsWithDup(int[] nums) {
        List<List<Integer>> res = new ArrayList<>();
        ArrayList<Integer> haf = new ArrayList<>();
        int n = nums.length;

        res.add(new ArrayList<>());
        Arrays.sort(nums);
        f(0, res, haf, nums, n);

        return res;
    }
}
```

8) Permutation (LC 46)

Given an array `nums` of distinct integers, return *all the possible permutations*. You can return the answer in **any order**.

**Example 1:**

```
Input: nums = [1,2,3]
Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

```
class Solution {

    public void swap(int i, int j, int[] nums) {
        int tem = nums[i];
        nums[i] = nums[j];
        nums[j] = tem;
    }

    public void f(int idx, List<List<Integer>> res, int[] nums) {

        if (idx == nums.length) {
            res.add(new ArrayList<>(nums));
            return;
        }

        for (int i = idx; i < nums.length; i++) {
            swap(i, idx, nums);
            f(idx + 1, res, nums);
            swap(i, idx, nums);
        }
    }

    public List<List<Integer>> permute(int[] nums) {

        List<List<Integer>> res = new ArrayList<>();

        f(0, res, nums);

        return res;
    }
}
```

9) PermutationII (LC 47)

Given a collection of numbers, `nums`, that might contain duplicates, return *all possible unique permutations* ***in any order***.

**Example 1:**

```
Input: nums = [1,1,2]
Output:
[[1,1,2],
 [1,2,1],
 [2,1,1]]
```

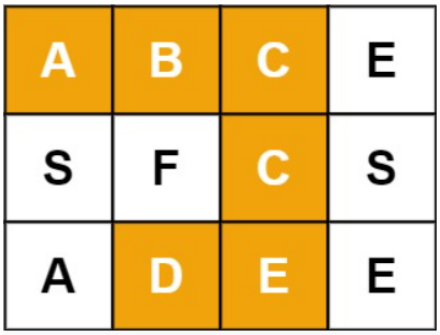
```
class Solution {

    public void f(int ind, List<List<Integer>> res, ArrayList<Integer> haf, boolean[] mark, int n, int[] nums) {

        if (haf.size() == n) {
            res.add(new ArrayList<>(haf));
            return;
        }

        for (int k = 0; k < n; k++) {
            if (k > 0 && nums[k] == nums[k - 1] && mark[k-1] == false) {
                continue;
            }

            if (mark[k] == false) {
                haf.add(nums[k]);
                mark[k] = true;
                f(k, res, haf, mark, n, nums);
                mark[k] = false;
                haf.remove(haf.size() - 1);
            }


        }
    }

    public List<List<Integer>> permuteUnique(int[] nums) {

        List<List<Integer>> res = new ArrayList<>();
        ArrayList<Integer> haf = new ArrayList<>();
        int n = nums.length;
        boolean[] mark = new boolean[n];

        f(0, res, haf, mark, n, nums);

        return res;
    }
}
```

10) Word Search (LC 79)

Given an `m x n` grid of characters `board` and a string `word`, return `true` *if* `word` *exists in the grid.*

The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

**Example 1:**




```
Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"
Output: true
```

```
class Solution {

    public boolean f(int i, int j, char[][] board, String word, int idx, boolean[][] vis, int[] di, int[] dj, int m, int n) {

        if (idx == word.length()) {
            return true;
        }

        for (int k = 0; k < 4; k++) {
            int ni = i + di[k];
            int nj = j + dj[k];

            if (ni >= 0 && ni < m && nj >= 0 && nj < n && vis[ni][nj] == false && board[ni][nj] ==
word.charAt(idx)) {
                vis[i][j] = true;
                boolean res = f(ni, nj, board, word, idx + 1, vis, di, dj, m, n);
                vis[i][j] = false;
                if (res == true) {
                    return true;
                }
            }
        }

        return false;
    }

    public boolean exist(char[][] board, String word) {

        int[] di = {0, 1, 0, -1};
        int[] dj = {1, 0, -1, 0};

        int m = board.length;
        int n = board[0].length;

        boolean[][] vis = new boolean[m][n];

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (word.charAt(0) == board[i][j]) {
                    vis[i][j] = true;
```

```
                boolean res = f(i, j, board, word, 1, vis, di, dj, m, n);
                vis[i][j] = false;
                if(res == true) {
                    return true;
                }
            }
        }
    }

    return false;
  }
}
```

11 ) Word BreakII (LC 140)

Given a string `s` and a dictionary of strings `wordDict`, add spaces in `s` to construct a sentence where each word is a valid dictionary word. Return all such possible sentences in **any order**.

**Note** that the same word in the dictionary may be reused multiple times in the segmentation.

**Example 1:**

```
Input: s = "catsanddog", wordDict = ["cat","cats","and","sand","dog"]
Output: ["cats and dog","cat sand dog"]
```

```
class Solution {
    public void f(int ind, List<String> res, Set<String> word, String haf, String s){
        if(ind == s.length()){
            res.add(haf.trim());
            return;
        }

        for(int i = ind + 1; i <= s.length(); i++){

            String substring = s.substring(ind, i);

            if (word.contains(substring)){
                f(i, res, word, haf + substring + " ", s);
            }

        }
    }
    public List<String> wordBreak(String s, List<String> wordDict) {
        Set<String> word = new HashSet<>(wordDict);

        List<String> res = new ArrayList<>();

        f(0, res, word, "", s);

        return res;

    }
}
```

12) Rat In a Maze (GFG)

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 **'U'(up)**, **'D'(down)**, **'L' (left)**, **'R' (right)**. Value 0 at a cell in the matrix represents that it is blocked and rat cannot move to it while value 1 at a cell in the matrix represents that rat can be travel through it.

**Note**: In a path, no cell can be visited more than one time. If the source cell is 0, the rat cannot move to any other cell.

**Example 1:**

**Input**:
N = 4
m[][] = {{1, 0, 0, 0},
{1, 1, 0, 1},
{1, 1, 0, 0},
{0, 1, 1, 1}}
**Output:**
DDRDRR DRDDRR

```
// m is the given matrix and n is the order of matrix
class Solution {

   public static void f(int i, int j, int[] di, int[] dj, ArrayList<String> res, String cur, int n, int[][] m, boolean[][]
visited)
   {

      if (i == n - 1 && j == n - 1){
         res.add(new String(cur));
         return;
      }

      String dir = "RDLU";

      for (int k = 0; k < dir.length(); k++){
         int ni = i + di[k];
         int nj = j + dj[k];

         if (ni >= 0 && ni < n && nj >= 0 && nj < n && m[ni][nj] == 1 && visited[ni][nj] == false){
            visited[i][j] = true;
            f(ni, nj, di, dj, res, cur + dir.charAt(k), n, m, visited);
            visited[i][j] = false;
         }
      }
   }

   public static ArrayList<String> findPath(int[][] m, int n) {
      // Your code here

      int[] di = {0, 1, 0, -1};
      int[] dj = {1, 0, -1, 0};
      ArrayList<String> res = new ArrayList<>();
      if (m[0][0] == 0 || m[n-1][n-1] == 0){
         return res;
      }
      boolean visited[][] = new boolean[n][n];
      f(0, 0, di, dj, res, "", n, m, visited);
      return res;
   }
}
```

13) Valid Sudoku (LC 36)

Determine if a `9 x 9` Sudoku board is valid. Only the filled cells need to be validated **according to the following rules**:

1. Each row must contain the digits `1-9` without repetition.
2. Each column must contain the digits `1-9` without repetition.
3. Each of the nine `3 x 3` sub-boxes of the grid must contain the digits `1-9` without repetition.

**Note:**

- A Sudoku board (partially filled) could be valid but is not necessarily solvable.
- Only the filled cells need to be validated according to the mentioned rules.

```
class Solution {

    public boolean isSafe(char[][] board, int k,int row, int col) {

        for (int x = 0; x < 9; x++) {

            if ( (board[x][col] == k) || (board[row][x] == k) || ( board[3 * (row / 3) + (x / 3)][3 * (col / 3) + (x % 3)]
== k) ) {
                return false;
            }
        }

        return true;
    }

    public boolean place(char[][] board) {

        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                if (board[i][j] != '.') {
                    continue;
                }
                for (char k = '1'; k <= '9'; k++) {
                    if (isSafe(board, k, i, j)) {
                        board[i][j] = k;

                        boolean res = place(board);
                        if (res) {
                            return true;
                        }

                        board[i][j] = '.';
                    }
                }
                return false;
            }
        }
        return true;
    }

    public boolean isValidSudoku(char[][] board) {

        boolean res = place(board);

        return res;
    }
}
```

14) Sudoku Solver (LC 37)

Write a program to solve a Sudoku puzzle by filling the empty cells.

A sudoku solution must satisfy **all of the following rules**:

1. Each of the digits `1-9` must occur exactly once in each row.
2. Each of the digits `1-9` must occur exactly once in each column.
3. Each of the digits `1-9` must occur exactly once in each of the 9 `3x3` sub-boxes of the grid.

The `'.'` character indicates empty cells.

```
class Solution {

    public boolean safe(char k, int row, int col, char[][] board) {

        for (int x = 0; x < 9; x++) {
            if ( (board[x][col] == k) || (board[row][x] == k) || (board[3 * (row/3) + (x/3)][3 * (col / 3) + (x%3)] == k)
) {
                return false;
            }
        }
        return true;
    }

    public boolean place(char[][] board) {

        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                if (board[i][j] != '.') {
                    continue;
                }

                for (char k = '1'; k <= '9'; k++) {
                    if (safe(k, i, j, board)) {
                        board[i][j] = k;
                        boolean res = place(board);
                        if (res == true) {
                            return true;
                        }
                        board[i][j] = '.';
                    }
                }
                return false;
            }
        }
        return true;
    }

    public void solveSudoku(char[][] board) {
        place(board);
    }
}
```

15 N Queen (LC 51)

The **n-queens** puzzle is the problem of placing `n` queens on an `n x n` chessboard such that no two queens attack each other.

Given an integer `n`, return *all distinct solutions to the **n-queens puzzle***. You may return the answer in **any order**.

Each solution contains a distinct board configuration of the n-queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space, respectively.

**Approach I**

```
class Solution {

    public boolean check(int row, int col, ArrayList<String> board, int n) {
        int i = row;
        int j = col;

        while (i >= 0 && j >= 0) {
            if (board.get(i).charAt(j) == 'Q') {
                return false;
            }
            i--;
            j--;
        }

        i = row;
        j = col;

        while (j >= 0) {
            if (board.get(i).charAt(j) == 'Q') {
                return false;
            }
            j--;
        }

        i = row;
        j = col;

        while (i < n && j >= 0) {
            if (board.get(i).charAt(j) == 'Q') {
                return false;
            }
            i++;
            j--;
        }

        return true;
    }

    public void place(int col, ArrayList<String> board, List<List<String>> res, int[] di, int[] dj, int n) {

        // base case
        if (col == n) {
            res.add(new ArrayList<>(board));
            return;
        }

        // recus

        for (int row = 0; row < n; row++) {
            if (check(row, col, board, n)) {
                char[] rowfull = board.get(row).toCharArray();
                rowfull[col] = 'Q';
                board.set(row, new String(rowfull));
```

```
                place(col+1, board, res, di, dj, n);

                rowfull[col] = '.';
                board.set(row, new String(rowfull));
            }
        }
    }

    public List<List<String>> solveNQueens(int n) {

        List<List<String>> res = new ArrayList<>();
        ArrayList<String> board = new ArrayList<>();

        String s = ".".repeat(n);

        for (int i = 0; i < n; i++) {
            board.add(s);
        }

        int[] di = {0, 1, 0, -1};
        int[] dj = {1, 0, -1, 0};

        place(0, board, res, di, dj, n);

        return res;

    }
}
```

**Approach II**

```
import java.util.ArrayList;

class Solution {

    public void f(int col, ArrayList<String> board, List<List<String>> res, int n, int[] up, int[] left, int[] down){

        if (col == n){
            res.add(new ArrayList<>(board));
            return;
        }

        for (int row = 0; row < n; row++){

            char[] curRow = board.get(row).toCharArray();

            if (up[n - 1 + col - row] == 0 && down[col + row] == 0 && left[row] == 0){

                curRow[col] = 'Q';
                board.set(row, new String(curRow));

                up[n - 1 + col - row] = 1;
                left[row] = 1;
                down[row + col] = 1;

                f(col + 1, board, res, n, up, left, down);

                curRow[col] = '.';
                board.set(row, new String(curRow));
```

```
                up[n - 1 + col - row] = 0;
                left[row] = 0;
                down[row + col] = 0;

            }
        }

    }

    public List<List<String>> solveNQueens(int n) {

        List<List<String>> res = new ArrayList<>();
        ArrayList<String> board = new ArrayList<>();

        String s = ".".repeat(n);

        for (int i = 0; i < n; i++){
            board.add(s);
        }

        int[] up = new int[2 * n - 1];
        int[] left = new int[n];
        int[] down = new int[2 * n - 1];

        f(0, board, res, n, up, left, down);

        return res;
    }
}
```

16 N Queen II (LC 52)

The **n-queens** puzzle is the problem of placing `n` queens on an `n x n` chessboard such that no two queens attack each other.

Given an integer `n`, return *the number of distinct solutions to the* ***n-queens puzzle***.

```
class Solution {

    // public boolean isSafe() {}

    public void place(int col, int n, int[] leftUp, int[] left, int[] leftDown, int[] res) {


        if (col == n) {
            res[0] = res[0] + 1;
            return;
        }

        for (int row = 0; row < n; row++) {
            if (leftUp[n - 1 + col - row] == 0 && left[row] == 0 && leftDown[row + col] == 0) {

                leftUp[n - 1 + col - row] = 1;
                left[row] = 1;
                leftDown[row + col] = 1;

                place(col + 1, n , leftUp, left, leftDown, res);

                leftUp[n - 1 + col - row] = 0;
                left[row] = 0;
                leftDown[row + col] = 0;
            }
        }

    }

    public int totalNQueens(int n) {

        int[] leftUp = new int[2 * n - 1];
        int[] left = new int[n];
        int[] leftDown = new int[2 * n - 1];

        int[] res = new int[1];

        place(0, n, leftUp, left, leftDown, res);

        return res[0];
    }
}
```