DFS   Depth-First Search

- Start at the root node and explore as far as possible along each branch before backtracking.
- Graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a boolean visited array. A graph can have more than one DFS traversal.
- https://www.cs.usfca.edu/~galles/visualization/DFS.html
- https://github.com/rayliu7717/CCC_CLASS/blob/main/CCC_GRAPH_DFS.java

DFS   Template

```
●   Result = []
void DFS ( path, list)
    if(match exist condition)
            result.add(path)
●       For (item : list)
            select this item
            DFS ( path, list) // back track
            cancel the selection
```

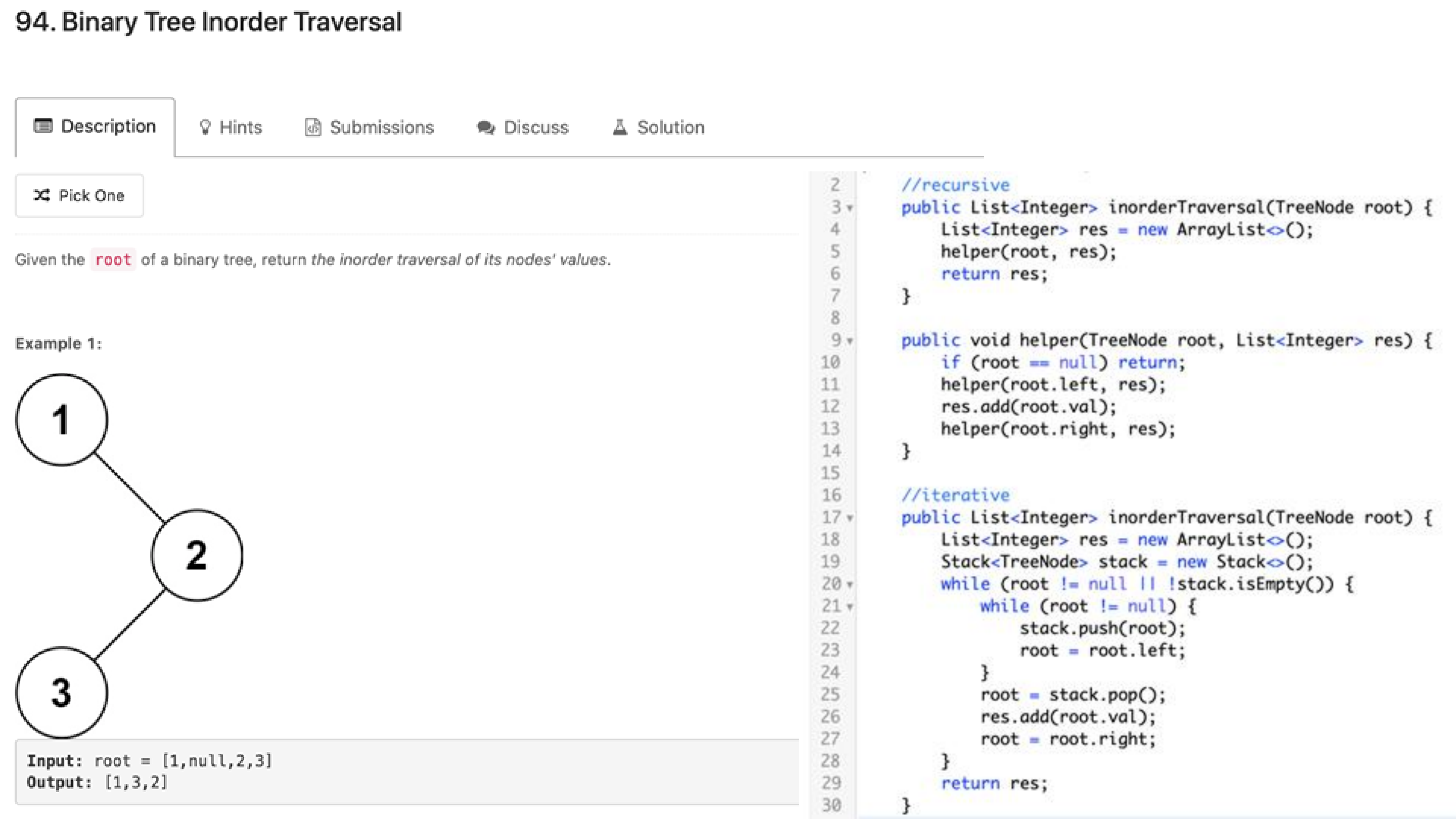
DFS is a Brute Force to enumerate all combinations.
Enumerate recursively,
Cannot use "for loop" to implement since we don't know how many loop level yet.

DFS Classic Problems

- Leetcode 78 Subset

  https://leetcode.com/problems/subsets/
- Leetcode 46 Permutations

  https://leetcode.com/problems/permutations/submissions/
- Leetcode 77 Combinations
- Leetcode 37 Sudoku Solver
- Leetcode 51 N-Queens



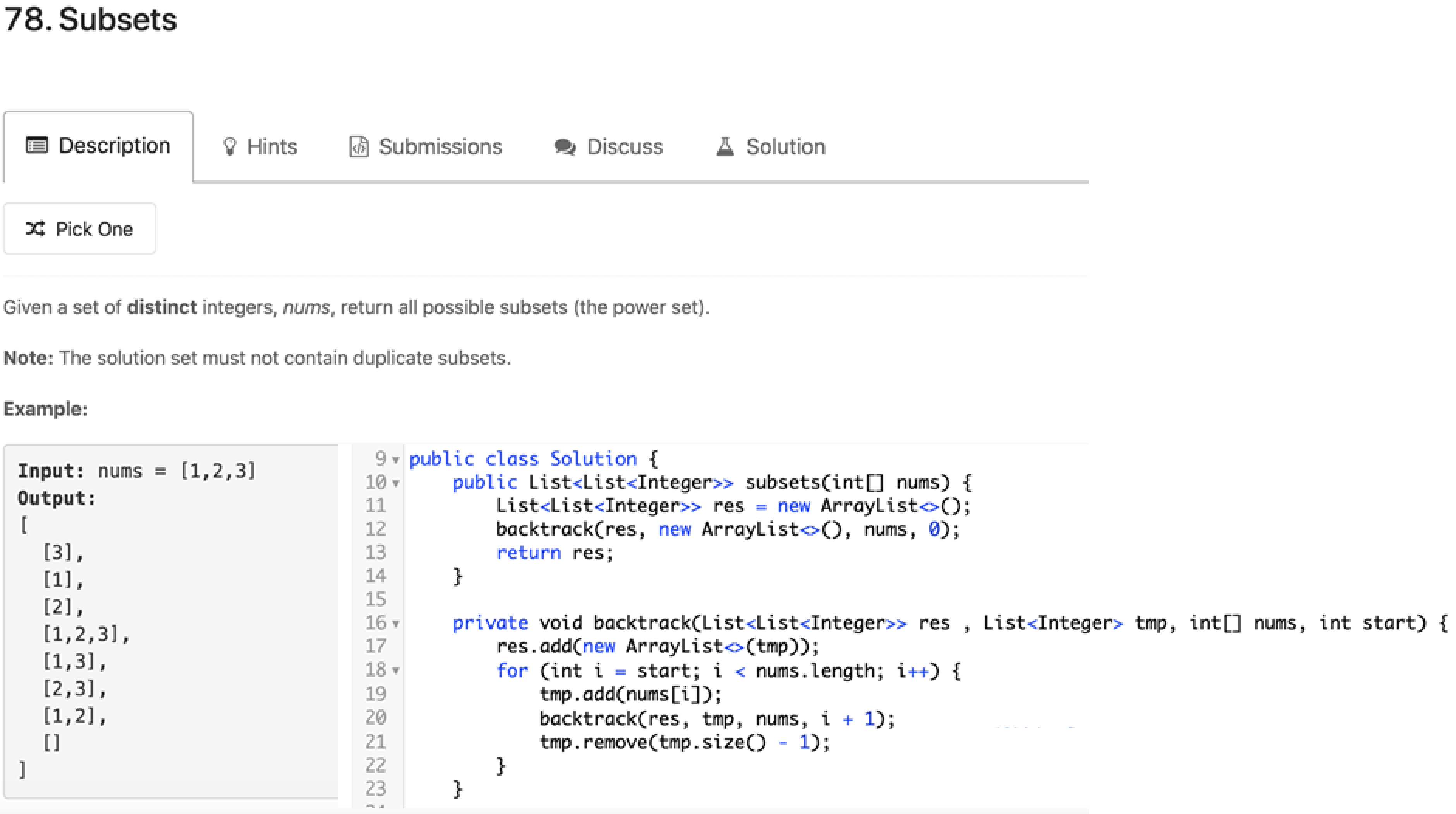
94. Binary Tree Inorder Traversal

Given the `root` of a binary tree, return *the inorder traversal of its nodes' values*.

Example 1:

```
Input: root = [1,null,2,3]
Output: [1,3,2]
```

```java
    //recursive
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        helper(root, res);
        return res;
    }

    public void helper(TreeNode root, List<Integer> res) {
        if (root == null) return;
        helper(root.left, res);
        res.add(root.val);
        helper(root.right, res);
    }

    //iterative
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        Stack<TreeNode> stack = new Stack<>();
        while (root != null || !stack.isEmpty()) {
            while (root != null) {
                stack.push(root);
                root = root.left;
            }
            root = stack.pop();
            res.add(root.val);
            root = root.right;
        }
        return res;
    }
```



78. Subsets

Given a set of **distinct** integers, *nums*, return all possible subsets (the power set).

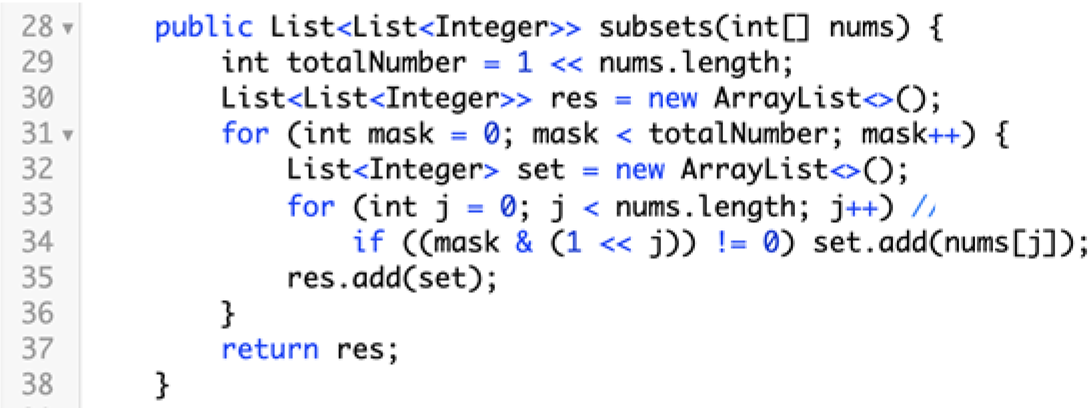
Note: The solution set must not contain duplicate subsets.

Example:

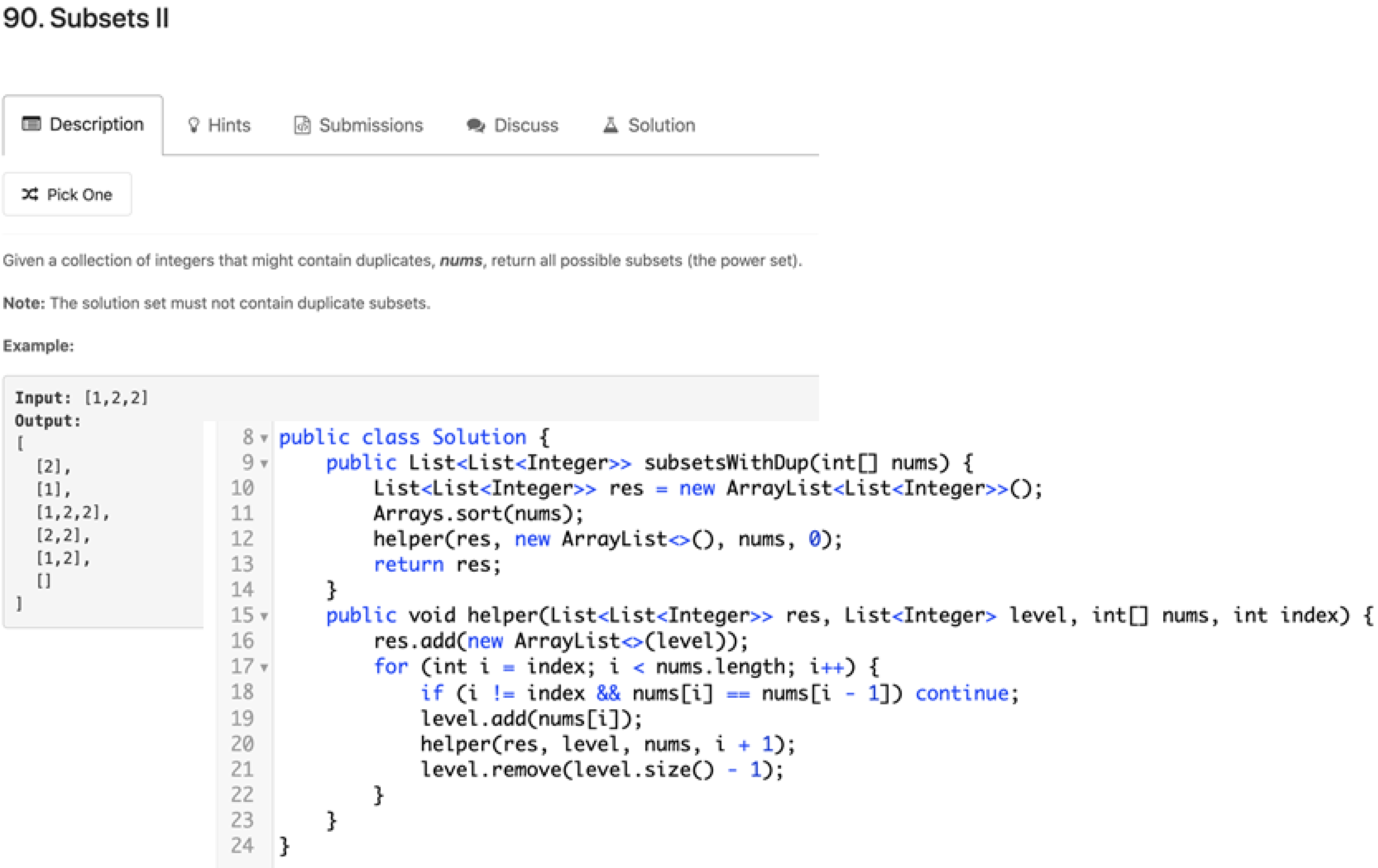
```
Input: nums = [1,2,3]
Output:
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

```java
public class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> res = new ArrayList<>();
        backtrack(res, new ArrayList<>(), nums, 0);
        return res;
    }

    private void backtrack(List<List<Integer>> res , List<Integer> tmp, int[] nums, int start) {
        res.add(new ArrayList<>(tmp));
        for (int i = start; i < nums.length; i++) {
            tmp.add(nums[i]);
            backtrack(res, tmp, nums, i + 1);
            tmp.remove(tmp.size() - 1);
        }
    }
```

	mask $2^3 = 8$							
1	0	0	0	0	1	1	1	1
2	0	0	1	1	0	0	1	1
3	0	1	0	1	0	1	0	1
	[ ]	[1]	[2]	[2 3]	[1]	[1 3]	[1 2 3]	[1 2 3]



```
    public List<List<Integer>> subsets(int[] nums) {
        int totalNumber = 1 << nums.length;
        List<List<Integer>> res = new ArrayList<>();
        for (int mask = 0; mask < totalNumber; mask++) {
            List<Integer> set = new ArrayList<>();
            for (int j = 0; j < nums.length; j++) //
                if ((mask & (1 << j)) != 0) set.add(nums[j]);
            res.add(set);
        }
        return res;
    }
```



90. Subsets II

Given a collection of integers that might contain duplicates, ***nums***, return all possible subsets (the power set).

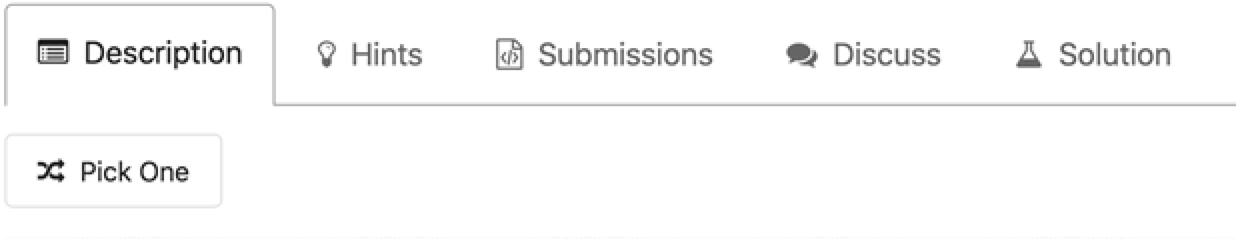
Note: The solution set must not contain duplicate subsets.

Example:

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

```java
public class Solution {
    public List<List<Integer>> subsetsWithDup(int[] nums) {
        List<List<Integer>> res = new ArrayList<List<Integer>>();
        Arrays.sort(nums);
        helper(res, new ArrayList<>(), nums, 0);
        return res;
    }
    public void helper(List<List<Integer>> res, List<Integer> level, int[] nums, int index) {
        res.add(new ArrayList<>(level));
        for (int i = index; i < nums.length; i++) {
            if (i != index && nums[i] == nums[i - 1]) continue;
            level.add(nums[i]);
            helper(res, level, nums, i + 1);
            level.remove(level.size() - 1);
        }
    }
}
```

46. Permutations

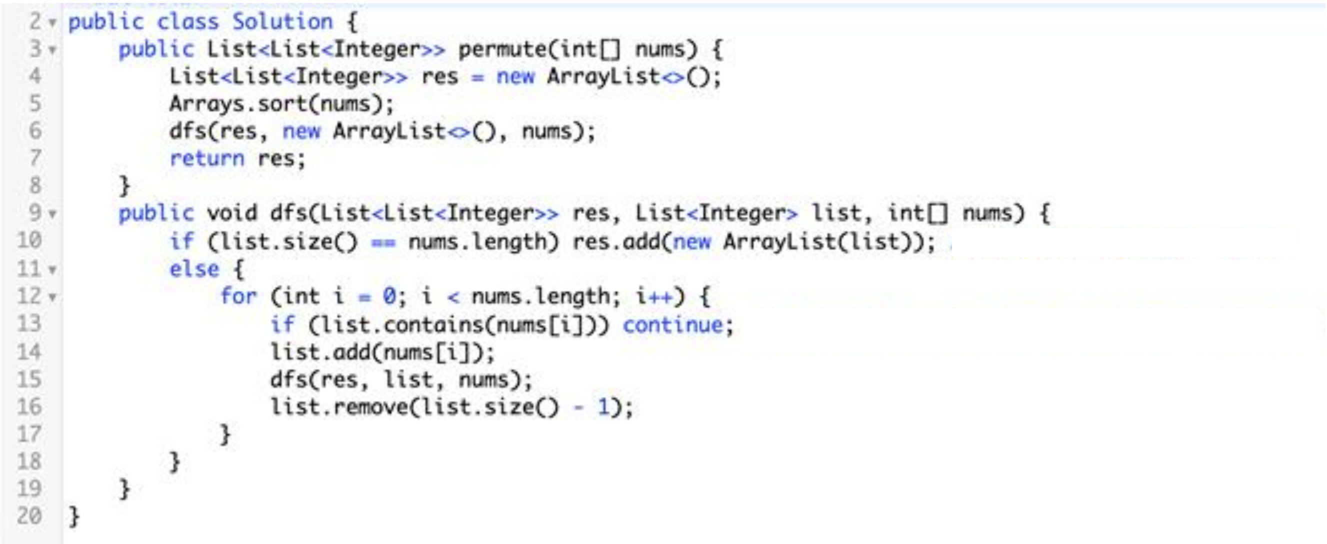


T(n)= O(*N*!).

Given a collection of **distinct** integers, return all possible permutations.

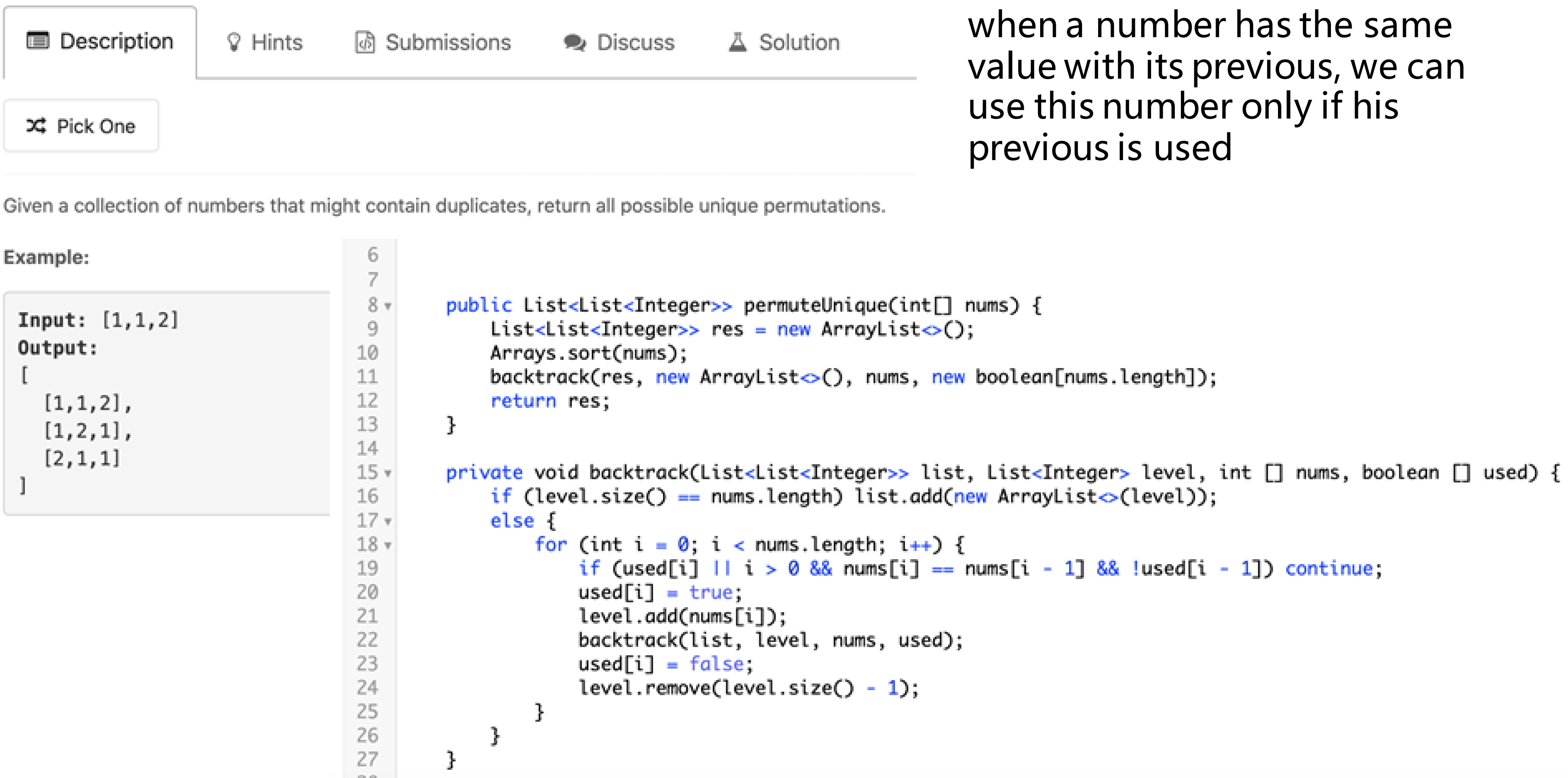
Example:

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



```java
public class Solution {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> res = new ArrayList<>();
        Arrays.sort(nums);
        dfs(res, new ArrayList<>(), nums);
        return res;
    }
    public void dfs(List<List<Integer>> res, List<Integer> list, int[] nums) {
        if (list.size() == nums.length) res.add(new ArrayList(list));
        else {
            for (int i = 0; i < nums.length; i++) {
                if (list.contains(nums[i])) continue;
                list.add(nums[i]);
                dfs(res, list, nums);
                list.remove(list.size() - 1);
            }
        }
    }
}
```

47. Permutations II



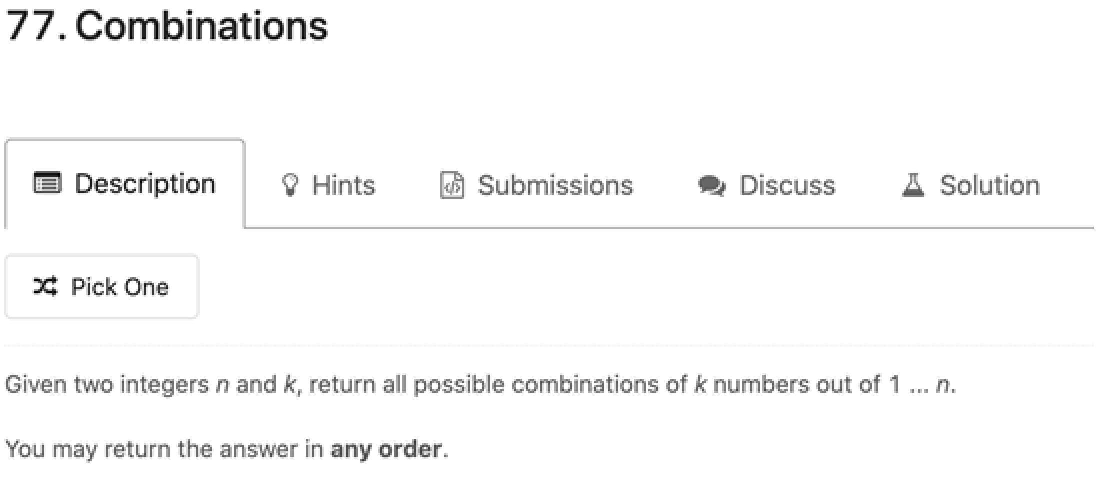
Given a collection of numbers that might contain duplicates, return all possible unique permutations.

Example:

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

```java


    public List<List<Integer>> permuteUnique(int[] nums) {
        List<List<Integer>> res = new ArrayList<>();
        Arrays.sort(nums);
        backtrack(res, new ArrayList<>(), nums, new boolean[nums.length]);
        return res;
    }

    private void backtrack(List<List<Integer>> list, List<Integer> level, int [] nums, boolean [] used) {
        if (level.size() == nums.length) list.add(new ArrayList<>(level));
        else {
            for (int i = 0; i < nums.length; i++) {
                if (used[i] || i > 0 && nums[i] == nums[i - 1] && !used[i - 1]) continue;
                used[i] = true;
                level.add(nums[i]);
                backtrack(list, level, nums, used);
                used[i] = false;
                level.remove(level.size() - 1);
            }
        }
    }
```

when a number has the same value with its previous, we can use this number only if his previous is used

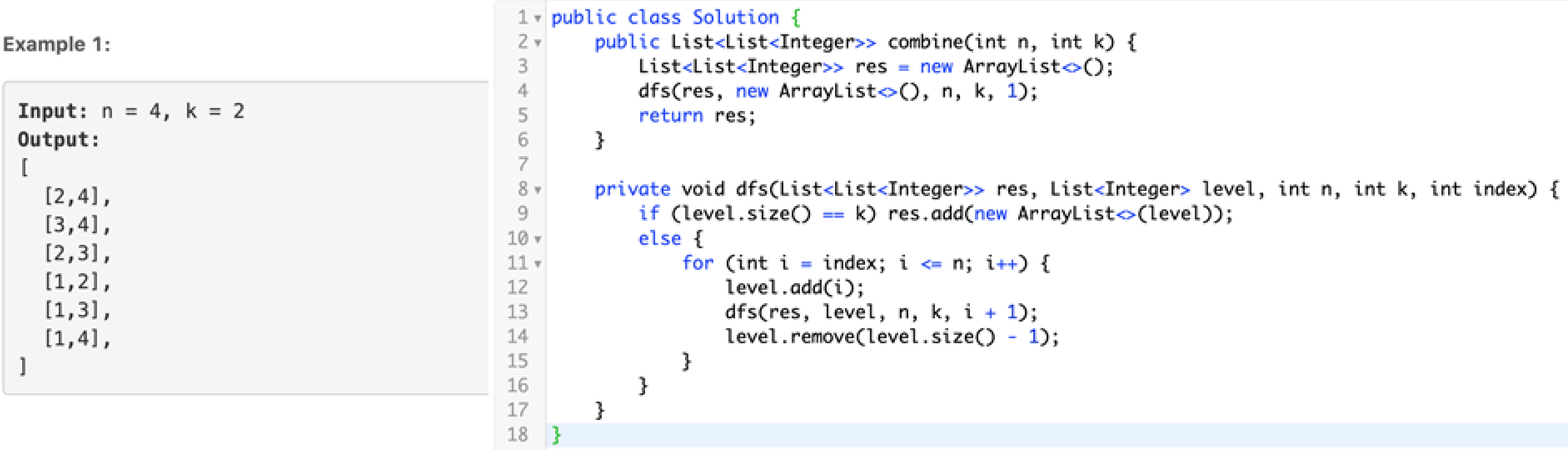


77. Combinations

Given two integers *n* and *k*, return all possible combinations of *k* numbers out of 1 ... *n*.

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

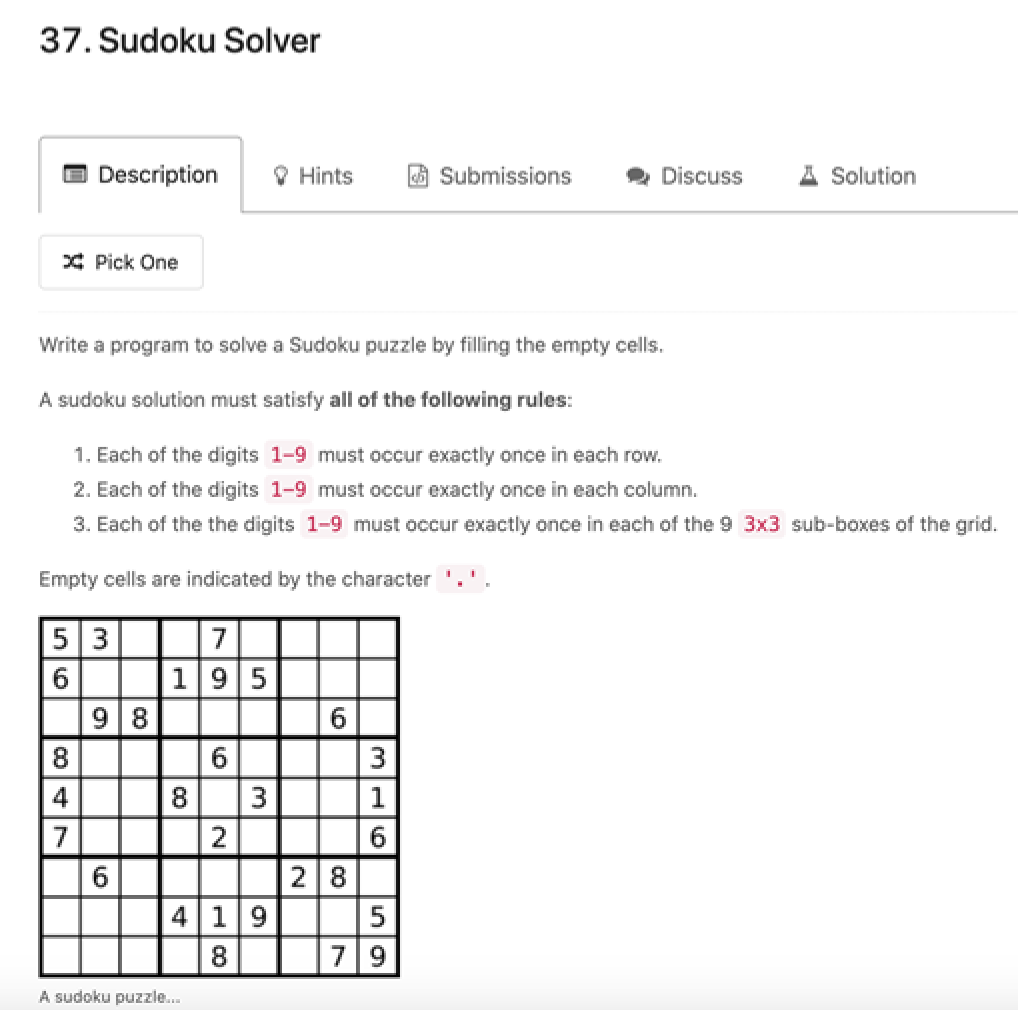
T(n)= $C_N^k = \frac{N!}{(N-k)!k!}$



Example 1:

```
Input: n = 4, k = 2
Output:
[
  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
```

```java
public class Solution {
    public List<List<Integer>> combine(int n, int k) {
        List<List<Integer>> res = new ArrayList<>();
        dfs(res, new ArrayList<>(), n, k, 1);
        return res;
    }

    private void dfs(List<List<Integer>> res, List<Integer> level, int n, int k, int index) {
        if (level.size() == k) res.add(new ArrayList<>(level));
        else {
            for (int i = index; i <= n; i++) {
                level.add(i);
                dfs(res, level, n, k, i + 1);
                level.remove(level.size() - 1);
            }
        }
    }
}
```



37. Sudoku Solver

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

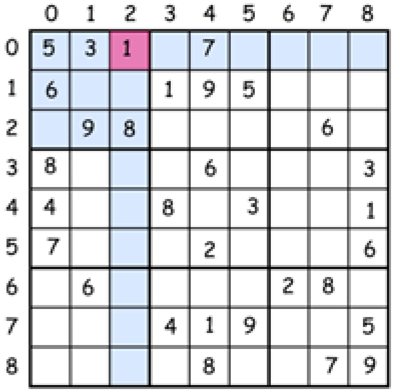
A sudoku solution must satisfy **all of the following rules**:

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

Empty cells are indicated by the character `'.'`.

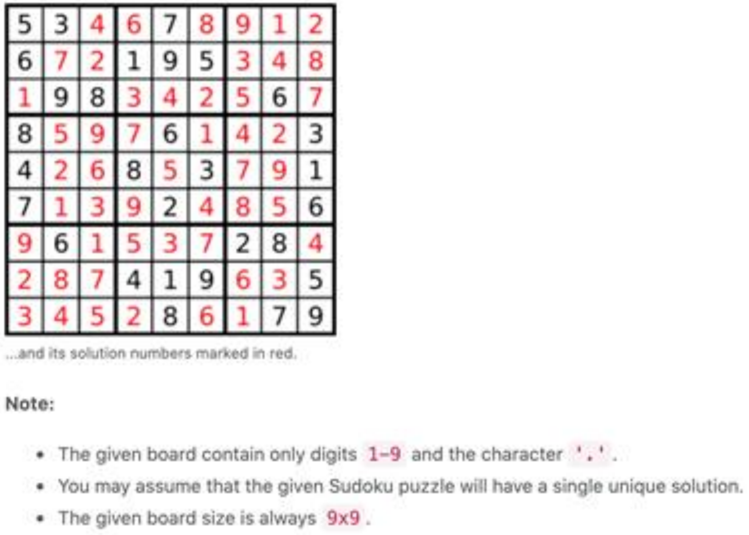
5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

A sudoku puzzle...



	0	1	2	3	4	5	6	7	8
0	5	3	1		7				
1	6			1	9	5			
2		9	8					6	
3	8				6				3
4	4			8		3			1
5	7				2				6
6		6					2	8	
7				4	1	9			5
8					8			7	9

Constraints propagation : no more 1s in rows[0], columns[2] and boxes[0]

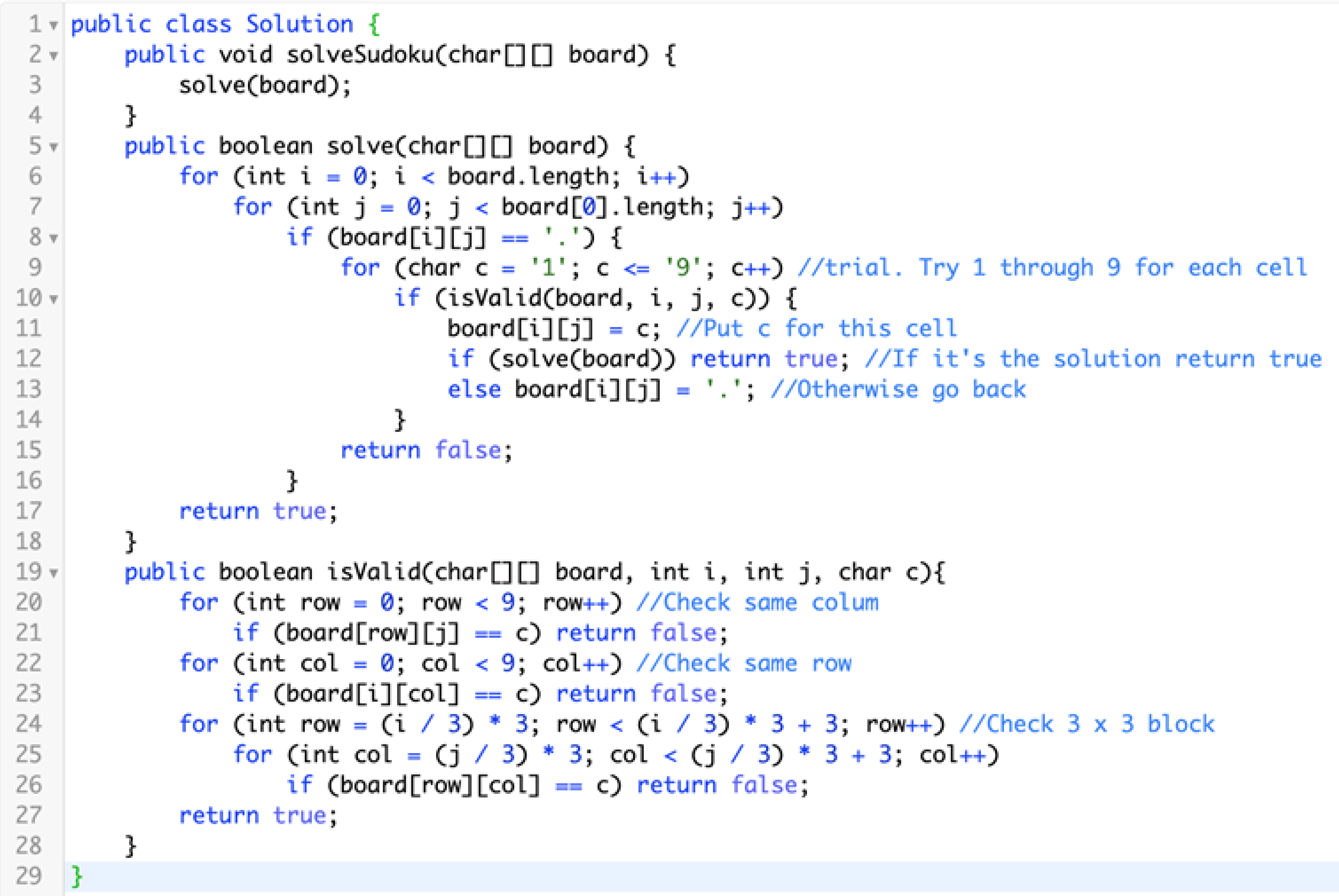


5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

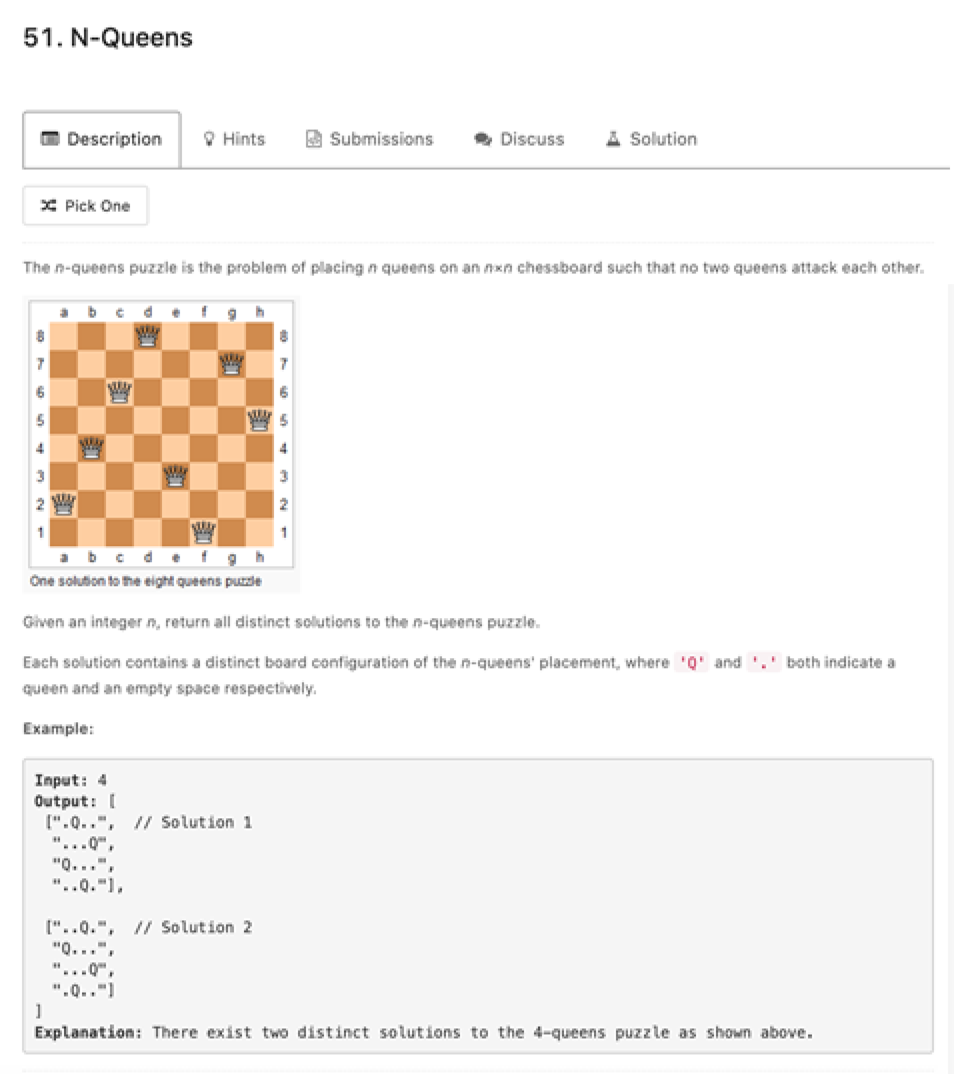
...and its solution numbers marked in red.

Note:

- The given board contain only digits `1-9` and the character `'.'`.
- You may assume that the given Sudoku puzzle will have a single unique solution.
- The given board size is always `9x9`.



```java
public class Solution {
    public void solveSudoku(char[][] board) {
        solve(board);
    }
    public boolean solve(char[][] board) {
        for (int i = 0; i < board.length; i++)
            for (int j = 0; j < board[0].length; j++)
                if (board[i][j] == '.') {
                    for (char c = '1'; c <= '9'; c++) //trial. Try 1 through 9 for each cell
                        if (isValid(board, i, j, c)) {
                            board[i][j] = c; //Put c for this cell
                            if (solve(board)) return true; //If it's the solution return true
                            else board[i][j] = '.'; //Otherwise go back
                        }
                    return false;
                }
        return true;
    }
    public boolean isValid(char[][] board, int i, int j, char c){
        for (int row = 0; row < 9; row++) //Check same colum
            if (board[row][j] == c) return false;
        for (int col = 0; col < 9; col++) //Check same row
            if (board[i][col] == c) return false;
        for (int row = (i / 3) * 3; row < (i / 3) * 3 + 3; row++) //Check 3 x 3 block
            for (int col = (j / 3) * 3; col < (j / 3) * 3 + 3; col++)
                if (board[row][col] == c) return false;
        return true;
    }
}
```



51. N-Queens

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

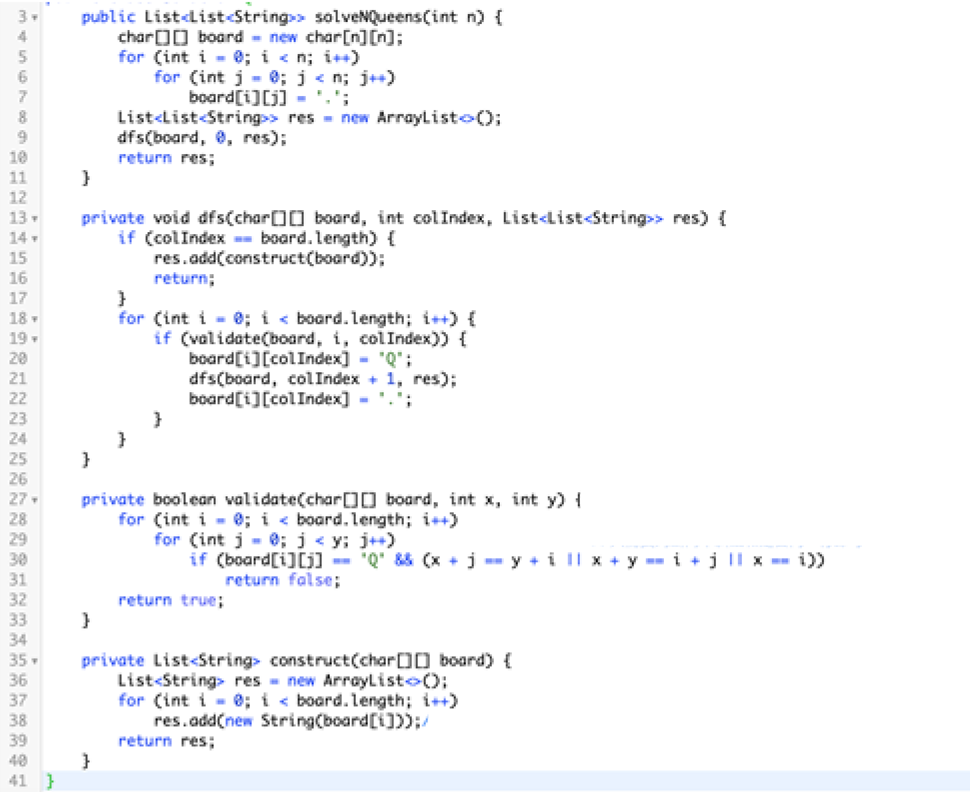
One solution to the eight queens puzzle

Given an integer *n*, return all distinct solutions to the *n*-queens puzzle.

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

Example:

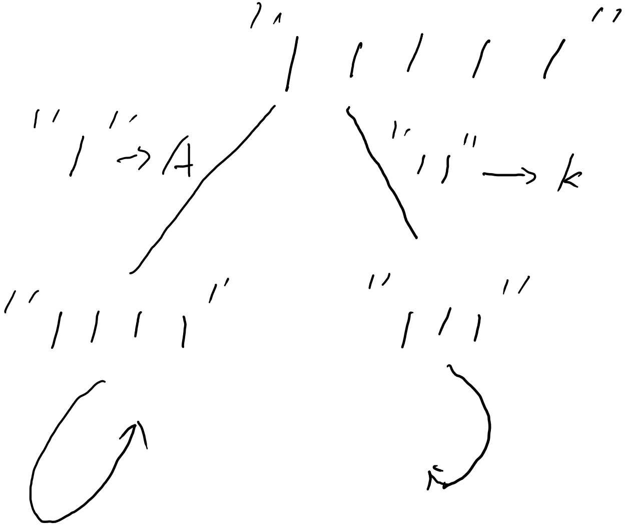
```
Input: 4
Output: [
 [".Q..",  // Solution 1
  "...Q",
  "Q...",
  "..Q."],

 ["..Q.",  // Solution 2
  "Q...",
  "...Q",
  ".Q.."]
]
Explanation: There exist two distinct solutions to the 4-queens puzzle as shown above.
```



```java
    public List<List<String>> solveNQueens(int n) {
        char[][] board = new char[n][n];
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                board[i][j] = '.';
        List<List<String>> res = new ArrayList<>();
        dfs(board, 0, res);
        return res;
    }

    private void dfs(char[][] board, int colIndex, List<List<String>> res) {
        if (colIndex == board.length) {
            res.add(construct(board));
            return;
        }
        for (int i = 0; i < board.length; i++) {
            if (validate(board, i, colIndex)) {
                board[i][colIndex] = 'Q';
                dfs(board, colIndex + 1, res);
                board[i][colIndex] = '.';
            }
        }
    }

    private boolean validate(char[][] board, int x, int y) {
        for (int i = 0; i < board.length; i++)
            for (int j = 0; j < y; j++)
                if (board[i][j] == 'Q' && (x + j == y + i || x + y == i + j || x == i))
                    return false;
        return true;
    }

    private List<String> construct(char[][] board) {
        List<String> res = new ArrayList<>();
        for (int i = 0; i < board.length; i++)
            res.add(new String(board[i]));
        return res;
    }
}
```

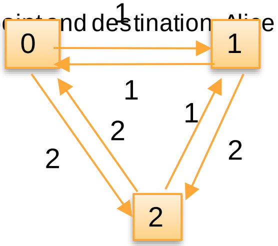
Leetcode 91 decode ways

https://leetcode.com/problems/decode-ways/description/



Travel Plan (DFS)

- There are n cities, and the adjacency matrix arr represents the distance between any two cities. arr[i][j] represents the distance from cityi to city j .Alice made a travel plan on the weekend. She started from city 0, then she traveled other cities 1 ~ n-1, and finally returned to city 0. Alice wants to know the minimum distance she needs to walk to complete the travel plan. Return this minimum distance. Except for city 0, every city can only pass once, and city 0 can only be the starting point and destination. Alice can't pass city 0 during travel.



- Input:
- [[0,1,2],[1,0,2],[2,1,0]]
- Output:
- 4
- Explanation:
- There are two possible plans.
- The first, city 0-> city 1-> city 2-> city 0, cost = 5.
- The second, city 0-> city 2-> city 1-> city 0, cost = 4.

```
public class Solution {
    /**
     * @param arr: the distance between any two cities
     * @return: the minimum distance Alice needs to walk to 
complete the travel plan
     */
    void dfs(int [][] arr, int nowpos, int n, boolean[] vis, int 
sum, int cnt, int[] ans )
    {
        // exit
        if(cnt == n -1){
            ans[0] = Math.min(ans[0], sum+ arr[nowpos][0]);
            return;
        }
        for(int i = 1;i<n; ++i){
            if(!vis[i]){
                vis[i] = true;
                dfs(arr,i, n, vis,sum+ arr[nowpos][i], cnt + 1, 
ans);
                vis[i] = false;  //backtrak
            }
        }
    }
    public int travelPlan(int[][] arr) {
        // Write your code here.
        int n = arr.length;
        boolean [] vis = new boolean[n];
        int[] ans = new int[1];
        ans[0] = Integer.MAX_VALUE;
        dfs(arr, 0, n, vis, 0, 0,ans);
        return ans[0];
    }
}
```

CCC '04 S3 – Spreadsheet (DFS)

1. Each Cell is a Graph Node (id is r,c)
2. Letter with number is graph edge
3. Traverse each Node to check circle
4. All the nodes on the circle will mark with "*"
5. If no circle, calculate the sum number.

```
int dfs(int r, int c, grid[][], vis[][]) {
  if (isNumeric(grid[r][c])) return grid[r][c].toInt();
  if (vis[r][c] || grid[r][c] == "*") return -1;
  vis[r][c] = true;
  String [] depend = grid[r][c].split("+");
   int sum = 0;
   for (int i = 0; i < depend.length; i++) {
     int ret = dfs(depend[i].charAt(0)-'A', depend[i].charAt(1)-'0' - 1);
     if (ret==-1) { grid[r][c] = "*"; return -1;}
     else sum+=ret;
   }
   grid[r][c] = sum.toString();
   return sum;
}

void spreadsheet(grid, rows, cols)
{
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
          vis = new boolean[rows][cols];
           dfs(i, j, grid, vis );
        }
    }
}
```