递归 & DFS & BFS & 分治

Climbing Stairs

mutual exclusive, compleete exhaustive

Invert Binary Tree

如果 当前节点为 null, 则终止
建立一个 temp, 交换左右节点, 左右节点各自继续作为新的根节点进行子节点的左右互换

Validate Binary Search Tree

https://www.youtube.com/watch?v=MILxfAbIhrE

递归检查当前元素是否在上下界之中
中序遍历，比较上一个元素跟当前元素

```
class Solution {
    public boolean isValidBST(TreeNode root, Integer lower, Integer upper) {
        if (root == null) return true;
        int val = root.val;
        if (lower != null && val <= lower) return false;
        if (upper != null && val >= upper) return false;
        return isValidBST(root.left, lower, val) && isValidBST(root.right, val, upper);
    }
    public boolean isValidBST(TreeNode root){
        return isValidBST(root, null, null);
    }
}
```

Generate Parentheses

左括号没有达到限制就添加左括号
左括号数量大于右括号就添加右括号
左右两边括号数量都达到限制代表已经得到最终结果

```
class Solution {
    public void generate(int left, int right, int n, String s, List<String> res) {
        if (left == n && right == n) {
            res.add(s);
            return;
        }
        if (left < n)
            generate(left + 1, right, n, s + "(", res);
        if (left > right)
            generate(left, right + 1, n, s + ")", res);
    }

    public List<String> generateParenthesis(int n) {
        List<String> res = new ArrayList<>();
        generate(0, 0, n, "", res);
        return res;
    }
}
```

Number Of Islands

```
public class Solution {
    private int n;
    private int m;

    public int numIslands(char[][] grid) {
        int count = 0;
        n = grid.length;
        if (n == 0) return 0;
        m = grid[0].length;
        //  traversal each cordinations
        for (int i = 0; i < n; i++)
            for (int j = 0; j < m; j++)
                if (grid[i][j] == '1') {
                    //  DFS:
                    //  find all island thoese near this island
                    //  mark them as waters
                    DFSMarking(grid, i, j);
                    ++count;
                }
        return count;
    }

    private void DFSMarking(char[][] grid, int i, int j) {
        if (i < 0 || j < 0 || i >= n || j >= m || grid[i][j] != '1') return;
        grid[i][j] = '0';
        DFSMarking(grid, i + 1, j);
        DFSMarking(grid, i - 1, j);
        DFSMarking(grid, i, j + 1);
        DFSMarking(grid, i, j - 1);
    }
}
```

Binary Tree Inorder Traversal

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

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

Binary Tree Preorder Traversal

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

    public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        preorder(root, res);
        return res;
    }
}
```

N-ary Tree Level Order Traversal

逐层处理当前节点，利用辅助队列方便操作先来后到的子层级

```
class Solution {
    public List<List<Integer>> levelOrder(Node root) {
        List<List<Integer>> res = new LinkedList<List<Integer>>();
        if (root == null) return res;
        Queue<Node> q = new LinkedList<>();
        q.offer(root);
        while (!q.isEmpty()) {
            List<Integer> currLevel = new LinkedList<>();
            int len = q.size();
            for (int i = 0; i < len; ++i) {
                Node curr = q.poll();
                currLevel.add(curr.val);
                for (Node n : curr.children)
                    q.offer(n);
            }
            res.add(currLevel);
        }
        return res;
    }
}
```

Find Largest Value in Each Tree Row

N-ary Tree Postorder Traversal

```
class Solution {
    public void postorder(Node root, List<Integer> res) {
        if (root == null) return;
        for (Node i : root.children)
            postorder(i, res);
        res.add(root.val);
    }
    public List<Integer> postorder(Node root) {
        List<Integer> res = new ArrayList<>();
        postorder(root, res);
        return res;
    }
}
```

N-ary Tree Preorder Traversal

```
class Solution {
    public void preorder(Node root, List<Integer> res) {
        if (root == null) return;
        res.add(root.val);
        for (Node i : root.children)
            preorder(i, res);
    }
    public List<Integer> preorder(Node root) {
        List<Integer> res = new ArrayList<>();
        preorder(root, res);
        return res;
    }
}
```

Minimum Genetic Mutation

```
class Solution {
    public int minMutation(String start, String end, String[] bank) {
        if(start.equals(end)) return 0;

        //  enhance check banks performance
        Set<String> bankSet = new HashSet<String>();
        for(String b: bank) bankSet.add(b);

        //  group possible chars
        char[] charSet = new char[]{'A', 'C', 'G', 'T'};

        int level = 0;
        Set<String> visited = new HashSet<String>();
        Queue<String> queue = new LinkedList<String>();

        //  start from "start"
        queue.offer(start);
        visited.add(start);

        while(!queue.isEmpty()) {
            int size = queue.size();

            while(size-- > 0) {
                String curr = queue.poll();
                if(curr.equals(end)) return level;

                //  generate each possible genetics
                char[] currArray = curr.toCharArray();
                for(int i = 0; i < currArray.length; ++i) {
                    char old = currArray[i];
                    for(char c: charSet) {
                        currArray[i] = c;
                        //  generate a new genetic
                        String next = new String(currArray);
                        //  check new genetic legality
                        if(!visited.contains(next) && bankSet.contains(next)) {
                            visited.add(next);
                            queue.offer(next);
                        }
                    }
                    currArray[i] = old;
                }
            }
            ++ level;
        }
        return -1;
    }
}
```

Maximum Depth of Binary Tree

节点为空就返回 0, 否则
return Math.max(left, right) + 1

Validate Binary Search Tree

验证当前节点值是否在 上限下限区间内
不断更新上限下限

```
public boolean isValidBST(TreeNode root, Integer lower, Integer upper) {
    if (root == null) return true;
    int val = root.val;
    if (lower != null && val <= lower) return false;
    if (upper != null && val >= upper) return false;
    return isValidBST(root.left, lower, val) && isValidBST(root.right, val, upper);
}

public boolean isValidBST(TreeNode root) {
    return isValidBST(root, null, null);
}
```

```
public boolean isValidBST(TreeNode root) {
    Stack<TreeNode> stack = new Stack<>();
    double inorder = - Double.MAX_VALUE;
    while (!stack.isEmpty() || root != null) {
        while (root != null) {
            stack.push(root);
            root = root.left;
        }
        root = stack.pop();
        if (root.val <= inorder) return false;
        inorder = root.val;
        root = root.right;
    }
    return true;
}
```

Minimum Depth of Binary Tree

比较 左右节点 谁小
要考虑如果树 没有或只有一个节点
if left == 0 || right == 0
return left + right + 1

Pow(x, n)

x^n 等于 (x^1/2n)*(x^1/2n) -> half * half
不断分一半 就是 logN
分一半叫分解问题
half * half 叫合并问题
处理 n 为负数
n 如果为负数 就让 x 为 1 / x;
n 为 -n;

终止条件为 n 二分后为 0
就返回 1

牛顿迭代法

```
class Solution {
    private double fastPow(double x, long n) {
        if (n == 0) return 1.0;
        double half = fastPow(x, n / 2);
        return n % 2 == 0 ? half * half : half * half * x;
    }
    public double myPow(double x, int n) {
        long N = n;
        if (N < 0) { //  negative n
            x = 1 / x;
            N = -N;
        }
        return fastPow(x, N);
    }
};
```

```
int mysqrt(int x){
        double tmpx = x;
        double k = 1.0;
        double k0 = 0.0;
        while(abs(k0-k) >= 1){
                k0 = k;
                k = (k + tmpx/k)/2;
        }
        return (int)k;
}
```

Lowest Common Ancestor

inorder 遍历整个树, 记录到达 p, q 两个点的路径
比较两个路径, 找到两个路径的不同点, 不同点之前的节点就是共同祖先
【需要额外空间】

https://www.youtube.com/watch?v=13m9ZCB8gjw

如果当前节点已找到或者为 null, 返回给上一层
查看左边的节点
查看右边的节点
如果左右两边都找到了节点，返回，说明当前节点就是它们的公共祖先
否则返回左或右节点(谁不为null返回谁)

Construct Binary Tree from Preorder and Inorder Traversal

preorder 找根节点, inorder 找到根节点位置, 根节点左边为 left, 根节点右边为 right

反复此步骤

https://www.youtube.com/watch?v=LrmitUxgXbc

使用 HashMap 去缓存 Inorder 的 index

Word Ladder II

- 在 Word Ladder I 小小改动即可

Word Ladder

双向 **BFS** 解法 性能是 **BFS** 的 **1/2**

核心是BFS
每次转化当前单词到字典中其中一个单词，再把字典中找到的单词删除 以免出现环形转换
- 先将字典转为 HashSet 方便查询与删除
- 使用 Queue, 将开始单词入队，开始循环检查 Queue 是否为空，队列未空则出队，对每次出队的单词进行每个字母 从 a-z 的字符替换，将新单词进行比较
- 如果新单词已经找到，则返回 level + 1
- 如果新单词在词典内，则将新单词放入 Queue 并将 HashSet 中的词典单词移除，避免反复处理
- 检查队列是否为空的代码最底部进行 ++ level
- 返回 0

```
class Solution {
    public int ladderLength(String beginWorld, String endWord, List<String> wordList) {
        Set<String> stringSet = new HashSet<String>();
        for (String s : wordList) stringSet.add(s);
        if (!stringSet.contains(endWord)) return 0;

        Queue<String> queue = new LinkedList<String>();
        queue.offer(beginWorld);
        int level = 1;

        while (!queue.isEmpty()) {
            int size = queue.size();
            while (size -- > 0) {
                String currentWorld = queue.poll();
                char[] worldChars = currentWorld.toCharArray();
                for (int j = 0; j < worldChars.length; ++j) {
                    char originalChar = worldChars[j];
                    for (char c = 'a'; c < 'z'; ++c) {
                        if (worldChars[j] == c) continue;
                        worldChars[j] = c;
                        String newWorld = String.valueOf(worldChars);
                        if (newWorld.equals(endWord)) return level + 1;
                        if (stringSet.contains(newWorld)) {
                            queue.offer(newWorld);
                            stringSet.remove(newWorld);
                        }
                    }
                    worldChars[j] = originalChar;
                }
            }
            ++ level;
        }
        return 0;
    }
}
```

Combinations

基于 DFS 从 1… n

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

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

Subset

求排列、组合的题 考虑用 DFS 来做
此题先对 input sort 一下, 然后记得每一层 add subset 时，对 subset deepcopy 一下

https://www.youtube.com/watch?v=33M1yV_FeB0

对当前层可 选 可 不选, 然后进入下一层

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

    public void dfs(int[] nums, List<List<Integer>> res, List<Integer> subset, int index) {
        if (index == nums.length) {
            res.add(new ArrayList<Integer>(subset));
            return;
        }
        dfs(nums, res, subset, index + 1);
        subset.add(nums[index]);
        dfs(nums, res, subset, index + 1);
        subset.remove(subset.size() - 1);
    }
}
```

https://leetcode.com/problems/permutations/discuss/18239/A-general-approach-to-backtracking-questions-in-Java-(Subsets-Permutations-Combination-Sum-Palindrome-Partioning)

Majority Element

```
public class Solution {
    public int majorityElement(int[] nums) {
        Arrays.sort(nums);
       int len = nums.length;
       return nums[len/2];
    }
}
```

```
class Solution {
    private int countInRange(int[] nums, int num, int lo, int hi) {
        int count = 0;
        for (int i = lo; i <= hi; ++i) {
            if (nums[i] == num) ++ count;
        }
        return count;
    }

    private int majorityElement (int[] nums, int lo, int hi) {
        if (lo == hi) return nums[lo];
        int mid = (hi - lo) / 2 + lo;
        int left = majorityElement(nums, lo, mid);
        int right = majorityElement(nums, mid + 1, hi);
        if (left == right) return left;
        int leftCount = countInRange(nums, left, lo, hi);
        int rightCount = countInRange(nums, right, lo, hi);
        return leftCount > rightCount ? left : right;
    }

    public int majorityElement(int[] nums) {
        return majorityElement(nums, 0, nums.length-1);
    }
}
```

Letter Combinations of a Phone Number

最顶层拿到第一个数字映射的字母
对每一个字母进行下潜(也把当前层的字母带到下一层去)
当下潜结束, 将所生成的字符串添加进 res

```
class Solution {
    public List<String> letterCombinations(String digits) {
        if (digits == null || digits.length() == 0) return new LinkedList<String>();
        Map<Character, String> map = new HashMap<>();
        map.put('2', "abc");
        map.put('3', "def");
        map.put('4', "ghi");
        map.put('5', "jkl");
        map.put('6', "mno");
        map.put('7', "pqrs");
        map.put('8', "tuv");
        map.put('9', "wxyz");
        List<String> res = new LinkedList<>();
        search("", digits, 0, res, map);
        return res;
    }

    public void search(String s, String digits, int index, List<String> res, Map<Character, String> map) {
        //  arrive bottom terminate it
        if (index == digits.length()) {
            res.add(s);
            return;
        }
        String letters = map.get(digits.charAt(index)); //  current level
        for (int i = 0; i < letters.length(); ++ i) {
            search(s + letters.charAt(i), digits, index + 1, res, map); //  drill another level
        }
    }
}
```

Premutations

第一层遍历数组所有元素, 不断 drill down 到 nums.length 层
每一层结果 遇到已在 state 的元素则 skip

```
class Solution {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> res = new ArrayList<>();
        if (nums == null || nums.length == 0) return res;
        dfs(new ArrayList<Integer>(), res, nums);
        return res;
    }

    public void dfs(List<Integer> state, List<List<Integer>> res, int[] nums) {
        if (state.size() == nums.length) {
            res.add(new ArrayList<Integer>(state));
            return;
        }
        for (int i = 0; i < nums.length; ++i) {
            if (state.contains(nums[i])) continue; //  检查重复元素
            state.add(nums[i]);
            dfs(state, res, nums); //  drill down 到 nums.length 层, nums^2 的味道
            state.remove(state.size() - 1);
        }
    }
}
```

N Queens

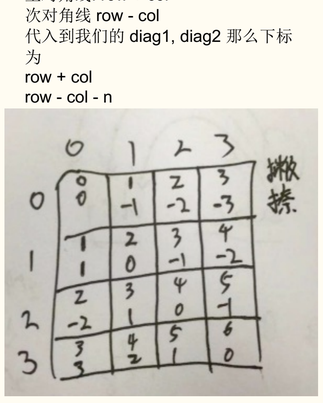
初始化一个 n*n board, 默认填充 “.” // reusable board, prevent duplicate generate board
初始化一个 new boolean[n] 记录 cols 位置占用情况 // instead of using HashSet
各初始化一个 new boolean[2*n] 记录 diag1、diag2 占用情况 // instead of using HashSet

主算法:
遍历 cols, 对**无法放置**(见下面注解)的 column 进行 continue
对每个可放置的 column 都不断下浅到下一层 row 去找可行性放置位置,
当遍历到最后一层 则将当前 board 转换为输出格式 并 添加至 res
每次回到上一层, 对 board、cols、diag1、diag2 还原下浅前状态

无法放置:
当前 col 的位置已经有 Queen || 当前主对角线已经被使用 || 当前次对角线已经被使用：
(cols[col] || diag1[row + n - col] || diag2[row + col])

为什么 diag1,2 是 2n 长度的数组? 因为算对角线时会存在负数 所以数组会爆掉, 那双倍长度的数组, 我们在存取 次对角线 boolean 标记时, index 都 加减 n, 就可以防止数组爆掉

对角线计算方式
主对角线: row + col
次对角线 row - col
代入到我们的 diag1, diag2 那么下标为
row + col
row - col - n




```
private void dfs(char[][] board, int row, int n, List<List<String>> res, boolean[] cols, boolean[] diag1, boolean[] diag2) {
    if (row == n) {
        List<String> toAdd = new ArrayList<String>();
        for (int i = 0; i < n; ++i)
            toAdd.add(String.valueOf(board[i]));
        res.add(toAdd);
        return;
    }

    for (int col = 0; col < n; ++col) {
        //  diag1 maybe is a negative number, so plus n for prevent out-boundary of array:
        //  diag1 index = row + n - col - 1
        //  diag2 index = row + col
        if (cols[col] || diag1[row + n - col - 1] || diag2[row + col])
            continue;
        //  mark current level
        cols[col] = true;
        diag1[row + n - col - 1] = true;
        diag2[row + col] = true;
        board[row][col] = 'Q';

        //  drill down
        dfs(board,  row: row + 1, n, res, cols, diag1, diag2);

        //  reverse current level
        board[row][col] = '.';
        cols[col] = false;
        diag1[row + n - col - 1] = false;
        diag2[row + col] = false;
    }
}

public List<List<String>> solveNQueens(int n) {
    List<List<String>> res = new ArrayList<List<String>>();
    //  reusable board
    char[][] curr = new char[n][n];
    for (int i = 0; i < n; i ++) {
        for (int j = 0; j < n; j ++) {
            curr[i][j] = '.';
        }
    }

    //  instead of using hash
    boolean[] col = new boolean[n];
    boolean[] diag1 = new boolean[2 * n - 1]; //  2*n for negative diag1 index
    boolean[] diag2 = new boolean[2 * n - 1]; //  2*n for negative diag1 index
    //  start dfs
    dfs(curr,  row: 0, n, res, col, diag1, diag2);
    return res;
}
```