Table of Contents

Introduction 1.1

Search & Backtracking 搜索与回溯 1.2

- Tree 与 BackTracking 的比较 1.2.1
- Subsets, Combination 与 Permutation 1.2.2
- Subsets & Combinations & Combination Sum 1.2.3
- 枚举法 1.2.4
- N 皇后 + 矩阵 Index Trick 1.2.5
- Sudoku 数独 + 矩阵 Index Trick 1.2.6
- Word Ladder I & II 1.2.7
- Number of ways 类 1.2.8
- DFS flood filling 1.2.9
- Strobogrammatic 数生成 1.2.10
- String 构造式 DFS + Backtracking 1.2.11
- Word Pattern I & II 1.2.12
- (G) Binary Watch 1.2.13
- (FB) Phone Letter Combination 1.2.14
- 常见搜索问题的迭代解法 1.2.15

String，字符串类 1.3

- 多步翻转法 1.3.1
- Substring 结构和遍历 1.3.2
- Palindrome 问题 1.3.3
- Palindrome Continued 1.3.4
- String / LinkedList 大数运算 1.3.5
- 序列化与压缩 1.3.6
- 5/24 String 杂题 1.3.7
- Knuth–Morris–Pratt 字符串匹配 1.3.8
- Lempel–Ziv–Welch 字符串压缩算法 1.3.9

(G) Decode String 1.3.10
(G) UTF-8 Validation 1.3.11
Binary Tree，二叉树 1.4
各种 Binary Tree 定义 1.4.1
LCA 类问题 1.4.2
三序遍历，vertical order 1.4.3
Post order traversal 的应用 1.4.4
Min/Max/Balanced Depth 1.4.5
BST 1.4.6
子树结构 1.4.7
Level Order traversal 1.4.8
Morris 遍历 1.4.9
修改结构 1.4.10
创建 / 序列化 1.4.11
子树组合，BST query 1.4.12
路径与路径和 1.4.13
NestedInteger 类 1.4.14
(FB) 从 Binary Tree Path 看如何递归转迭代 1.4.15
(FB) Binary Tree Path 比较路径大小 1.4.16
比较好玩的 Binary Tree 概率题 1.4.17
Segment & Fenwick Tree，区间树 1.5
Segment Tree 基础操作 1.5.1
Segment Tree 的应用 1.5.2
Fenwick Tree (Binary Indexed Tree) 1.5.3
Range Sum Query 2D - Immutable 1.5.4
Union-Find，并查集 1.6
Union-Find，并查集基础 1.6.1
Union-Find, 并查集应用 1.6.2
Dynamic Programming, 动态规划 1.7
6/20, 入门 House Robber 1.7.1

7/12, Paint Fence / House 1.7.2
6/24, 滚动数组 1.7.3
6/24, 记忆化搜索 1.7.4
6/24, 博弈类 DP 1.7.5
博弈类DP, Flip Game 1.7.6
6/25, 区间类DP 1.7.7
6/27, subarray 划分类，股票 1.7.8
7/2, 字符串类 1.7.9
Bomb Enemies 1.7.10
8/2，背包问题 1.7.11
(G) Max Vacation 1.7.12
(11/4新增) AST 子树结构 DP 1.7.13
LinkedList，链表 1.8
6/9, LinkedList，反转与删除 1.8.1
6/11, LinkedList 杂题 1.8.2
(FB) 链表的递归与倒序打印 1.8.3
LinkedIn 面经，算法题 1.9
6/17, LinkedIn 面经题 1.9.1
6/28, LinkedIn 面经题 1.9.2
7/6, LinkedIn 面经 1.9.3
Shortest Word Distance 类 1.9.4
DFA Parse Integer 1.9.5
Two Pointers，双指针 1.10
3 Sum, 3 Sum Closest / Smaller, 4 Sum 1.10.1
对撞型，灌水类 1.10.2
对撞型，partition类 1.10.3
Wiggle Sort I & II 1.10.4
双指针，窗口类 1.10.5
双指针，窗口类 1.10.6
Heap，排序 matrix 中的 two pointers 1.10.7

Bit & Math，位运算与数学 1.11
Bit Manipulation，对于 '1' 位的操作 1.11.1
Math & Bit Manipulation, Power of X 1.11.2
坐标系 & 数值计算类 1.11.3
Add Digits 1.11.4
用 int 做字符串 signature 1.11.5
Interval 与 扫描线 1.12
Range Addition & LCS 1.12.1
7/5, Interval 类，扫描线 1.12.2
Trie，字典树 1.13
6/9, Trie, 字典树 1.13.1
单调栈，LIS 1.14
4/13 LIS 1.14.1
栈， 单调栈 1.14.2
Largest Divisible Subset 1.14.3
Binary Search 类 1.15
Matrix Binary Search 1.15.1
Array Binary Search 1.15.2
Find Peak Element I & II 1.15.3
**Median of Two Sorted Arrays 1.15.4
Graph & Topological Sort，图 & 拓扑排序 1.16
有向 / 无向 图的基本性质和操作 1.16.1
拓扑排序, DFS 做法 1.16.2
拓扑排序, BFS 做法 1.16.3
Course Schedule I & II 1.16.4
Alien Dictionary 1.16.5
Undirected Graph, BFS 1.16.6
Undirected Graph, DFS 1.16.7
矩阵，BFS 最短距离探索 1.16.8
欧拉回路，Hierholzer算法 1.16.9

AI, 迷宫生成 1.16.10
AI, 迷宫寻路算法 1.16.11
(G) Deep Copy 无向图成有向图 1.16.12
括号与数学表达式的计算 1.17
Iterator 类 1.18
Majority Element，Moore's Voting 1.19
Matrix Inplace Operations 1.20
常见数据结构设计 1.21
(G) Design / OOD 类算法题 1.22
随机算法 & 数据结构 1.23
(FB) I/O Buffer 1.24
(FB) Simplify Path, H-Index I & II 1.25
(FB) Excel Sheet, Remove Duplicates 1.26
Integer 的构造，操作，序列化 1.27
Frequency 类问题 1.28
Missing Number 类，元素交换，数组环形跳转 1.29
8/10, Google Tag 1.30
(FB) Rearrange String k Distance Apart 1.31
Abstract Algebra 1.32
Chap1 -- Why Abstract Algebra ? 1.32.1
Chap2 -- Operations 1.32.2
Chap3 -- The Definition of Groups 1.32.3

My Algorithm Notes

感谢转专业几年来学校老师同学和刷题小伙伴的各种帮助~

8月底开始认真面试。

已拿到 **Facebook, Google** 和 **Amazon - Audible offer**，不面了。

里面代码和总结都是自己一点一点写的，也包括很多第一次做时候的粗糙 / 错误版本，就当记录下自己的思路和犯的错误吧~

大多数代码的第一原则是："自己觉得易懂 + 当场能写清楚和讲明白"，所以很多代码其实不像 lc 答案那样特别简洁。

Search & Backtracking 搜索与回溯

普遍要求返回“所有解”，大多为暴力穷举，本质上是划出搜索结构树做 **DFS.**

要注意这类题和 **DP** 的区别与联系，有些题比如 **Word Break II** 符合 **DP** 性质，可以在搜索类基础上实施记忆化搜索，但是这样的题不多。

DFS + Backtracking 研究

- **Tree 上比较常用的是先 ADD ，然后分别在 leaf node return 之前还有两边 dfs 结束之后 REMOVE**
- **其实就是 pre-order traversal.**
- **其他的 DFS 是 element 已经加好了之后才开始的，而 Tree 是带着当前在考虑的 element 开始的。**

来看两道题，一个是LintCode的 Permutations，一个是 LeetCode 的 Binary Tree Paths:

二者之间处理 backtracking 的方式和递归结构有细微的差别。

> Permutations dfs 传的参数中不包括当前元素； Binary Tree Paths dfs 参数中包含当前考虑的元素；

- **5/25复习注：其实两个都带了当前要考虑的元素，只不过一个是用 index 形式（array），一个是 TreeNode.**

> Permutations 在 dfs 结束后回溯； Tree Paths 在 leaf node 回溯，在最后一个 dfs 后回溯；
>
> 另一个重要区别是，Permutations 是给定 array 的，可以使用 O(1) 的 random index access; 而 tree 不行，每次只能带着当前节点 root 去做 dfs.

DFS + Backtracking 都有三个步骤：

- Add element
- DFS
- Remove element

其中随着 dfs 传递参数的不同，三个步骤的位置会有变化。

在 Tree 上做 dfs + backtracking 比较适合用 dfs 带着当前考虑的 node 为参数，先 Add 然后在 leaf node 上做 Remove 的方式；

在中间结果是 **String** 的情况下，如果想保存一个 **object** 的 **reference** 可以用 **StringBuilder,** 同时也可以利用 **immutable** 的性质直接传新的 **String copy (**空间占用多一些**)**，这样可以免去回溯的步骤。

```
class Solution {
    /**
     * @param nums: A list of integers.
     * @return: A list of permutations.
     */
    public ArrayList<ArrayList<Integer>> permute(ArrayList<Integ
er> nums) {
        // write your code here
        ArrayList<ArrayList<Integer>> result = new ArrayList<Arr
ayList<Integer>>();
        if(nums == null || nums.size() == 0) return result;

        helper(result, new ArrayList<Integer>(), nums);
        return result;
    }

    private void helper(ArrayList<ArrayList<Integer>> result,
                        ArrayList<Integer> list,
                        ArrayList<Integer> nums){

        if(list.size() == nums.size()){
            result.add(new ArrayList<Integer>(list));
            return;
        }

        for(int i = 0; i < nums.size(); i++){
            if(list.contains(nums.get(i))) continue;
            list.add(nums.get(i));
            helper(result, list, nums);
            // 抹掉末尾
            list.remove(list.size() - 1);
        }
    }
}
```

```
public class Solution {
    public List<String> binaryTreePaths(TreeNode root) {
        List<String> list = new ArrayList<String>();
        if(root == null) return list;
        dfs(list, root, new StringBuilder());
        return list;
    }

    private void dfs(List<String> list, TreeNode node, StringBui
lder sb){
        if(node == null) return;

        // 存盘，记录当前位置
        int length = sb.length();
        ------------ADD-------------
        sb.append(node.val);
        ------------Check--------------
        if(node.left == null && node.right == null){
            list.add(sb.toString());
            // 归位（抹掉最后一个数字）
            sb.setLength(length);
            return;
        }

        -------------DFS---------------
        sb.append("->");
        dfs(list, node.left, sb);
        dfs(list, node.right, sb);
        -----------Backtrack-------------
        // 归位（抹掉最后一个箭头和前面的数字）
        sb.setLength(length);
    }
}
```

一个先加后 dfs 的例子，看起来稍微蛋疼了点；

```
public class Solution {
    public List<String> binaryTreePaths(TreeNode root) {
        List<String> list = new ArrayList<String>();
        if(root == null) return list;
        StringBuilder sb = new StringBuilder();
        sb.append(root.val);
        if(root.left == null && root.right == null){
            list.add(sb.toString());
            return list;
        }
        dfs(list, root, sb);
        return list;
    }

    private void dfs(List<String> list, TreeNode node, StringBui
lder sb){
        if(node == null) return;

        if(node.left == null && node.right == null){
            list.add(sb.toString());
            return;
        }

        sb.append("->");
        int length = sb.length();
        if(node.left != null){
            sb.append(node.left.val);
            dfs(list, node.left, sb);
            sb.setLength(length);
        }
        if(node.right != null){
            sb.append(node.right.val);
            dfs(list, node.right, sb);
            sb.setLength(length);
        }

    }
}
```

另一个比较有意思的解法，利用 String immutable 默认生成新 copy 的办法，当你 dfs 之间不是 by reference 指向同一个 object / collection ，也就不需要 backtracking 了。

```
public class Solution {
    public List<String> binaryTreePaths(TreeNode root) {
        List<String> res = new ArrayList<String>();
        if (root == null){
            return res;
        }
        return getPaths (root, "", res);
    }

    public List<String> getPaths(TreeNode root, String str, List
<String> res){
        if (root.left == null && root.right == null){
            if (str.equals("")){
                str += root.val;
            } else {
                str += "->"+root.val;
            }
            res.add (str);
            return res;
        }

        if (str.equals("")){
            str += root.val;
        } else {
            str += "->"+root.val;
        }

        if (root.left != null)  getPaths (root.left, str, res);
        if (root.right != null)  getPaths (root.right, str, res)
;

        return res;
    }
}
```

Subsets, Combination 与 Permutation

> **Combination** 类问题最重要的是去重， **dfs()** 函数里带一个 **index** 参数可以很好的解决这个问题。
>
> 顺序问题中有“单序列”和“全序列”顺序，分别对应一个序列中元素的顺序和整个序列中子序列顺序。可以通过子序列翻转或者全局翻转操作，利用两次翻转相互抵消的特点解决序列顺序问题。

用数学语言描述就是：**'** 代表 **inverse**

S = ABCD

S' = D'C'B'A'

(A'B'C'D')' = DCBA

LintCode - Combinations

Combination 类问题是典型的搜索问题，除了 DFS + backtracking 之外，combination 里最重要的就是“去重”，怎么让自己的搜索树不回头地往前走。

在这个问题里，k = depth of tree，n = branching factor. 当然因为解个数的唯一性，不是每个节点的 fan out 都一样。

在这个具体问题里，因为解是单调连续增加的序列 1,2..n，去重方法上可以稍微取巧一些：dfs里增加一个“前一个元素”的参数，每一层递归只考虑比上一个元素大的值。

```
public class Solution {
    /**
     * @param n: Given the range of numbers
     * @param k: Given the numbers of combinations
     * @return: All the combinations of k numbers out of 1..n
     */
    public List<List<Integer>> combine(int n, int k) {
        // write your code here
        List<List<Integer>> rst = new ArrayList<List<Integer>>()
;
        dfs(rst, new ArrayList<Integer>(), n, k, 0);
        return rst;
    }

    private void dfs(List<List<Integer>> rst, List<Integer> list
, int n, int k, int prev){
        if(list.size() == k) {
            rst.add(new ArrayList<Integer>(list));
            //list.remove(list.size() - 1);
        }

        for(int i = prev + 1; i <= n; i++){
            if(list.contains(i)) continue;
            list.add(i);
            dfs(rst, list, n, k, i);
            list.remove(list.size() - 1);
        }
    }
}
```

LintCode - Combination Sum

相似的题有一样的思路，不同的题有不同的坑。

为了去重的考虑，还是要 dfs 参数里带 index. 这里有一个细微的差别，因为同一个数可以用多次，新一层 dfs 迭代的时候要从上一层一样的 index 开始。然而还是要注意不要去看 index 之前的元素。

同时因为同一个元素可以用多次，必须要有一个有效的 dfs 终止条件，不然搜索树会永远一直加下去。。考虑到给定条件是“所有元素都为正整数”，我们就可以在当前 sum > target 的时候终止搜索。如果可以重复元素又有负数的话，这题就没法做了。

OJ 要求每一个结果都是 sorted list，稍微有点二逼，注意不能直接 sort 函数里的 list，因为会打乱其他 dfs 中的结果（same memory address pass by value），要去新建一个 list copy 再调用 Collections.sort()

```
public class Solution {
    /**
     * @param candidates: A list of integers
     * @param target:An integer
     * @return: A list of lists of integers
     */
    public List<List<Integer>> combinationSum(int[] candidates,
int target) {
        // write your code here
        List<List<Integer>> rst = new ArrayList<List<Integer>>()
;
        if(candidates == null || candidates.length == 0) return
rst;

        dfs(rst, new ArrayList<Integer>(), candidates, target, 0
, 0);

        return rst;
    }

    private void dfs(List<List<Integer>> rst, List<Integer> list
, int[] candidates,
                     int target, int curSum, int index){
        if(curSum > target) return;
        if(curSum == target){
            List<Integer> newList = new ArrayList<Integer>(list)
;
            Collections.sort(newList);
            rst.add(newList);
            return;
```

```
        }

        for(int i = index; i < candidates.length; i++){
            list.add(candidates[i]);
            curSum += candidates[i];

            dfs(rst, list, candidates, target, curSum, i);

            curSum -= candidates[i];
            list.remove(list.size() - 1);
        }
    }
}
```

LintCode - Subsets

```
class Solution {
    /**
     * @param S: A set of numbers.
     * @return: A list of lists. All valid subsets.
     */
    public ArrayList<ArrayList<Integer>> subsets(int[] nums) {
        // write your code here
        ArrayList<ArrayList<Integer>> rst = new ArrayList<ArrayL
ist<Integer>>();
        if(nums == null || nums.length == 0) return rst;

        Arrays.sort(nums);
        dfs(rst, new ArrayList<Integer>(), nums, 0);

        return rst;
    }

    private void dfs(ArrayList<ArrayList<Integer>> rst,
                     List<Integer> list, int[] nums, int index){
        rst.add(new ArrayList<Integer>(list));

        for(int i = index; i < nums.length; i++){
            list.add(nums[i]);
            dfs(rst, list, nums, i + 1);
            list.remove(list.size() - 1);
        }
    }
}
```

LintCode - Permutations

都是简单套路和小变形。这次每轮dfs都考虑所有元素（所以不用传 index 参数了），传个 boolean array 只挑没用过的数字就可以了。

```java
class Solution {
    /**
     * @param nums: A list of integers.
     * @return: A list of permutations.
     */
    public ArrayList<ArrayList<Integer>> permute(ArrayList<Integ
er> nums) {
        // write your code here
        ArrayList<ArrayList<Integer>> rst = new ArrayList<ArrayL
ist<Integer>>();
        if(nums == null || nums.size() == 0) return rst;

        boolean[] used = new boolean[nums.size()];
        dfs(rst, new ArrayList<Integer>(), nums, used);

        return rst;
    }

    private void dfs(ArrayList<ArrayList<Integer>> rst, List<Int
eger> list,
                     ArrayList<Integer> nums, boolean[] used){
        if(list.size() == nums.size()){
            rst.add(new ArrayList<Integer>(list));
            return;
        }

        for(int i = 0; i < nums.size(); i++){
            if(used[i]) continue;
            list.add(nums.get(i));
            used[i] = true;

            dfs(rst, list, nums, used);

            used[i] = false;
            list.remove(list.size() - 1);
        }
    }
}
```

LintCode - Permutations II

基本没啥区别。只加了新的一行，确保下在一个 dfs 返回，回溯结束之后，下一个数别选一样的就行了。

```
class Solution {
    /**
     * @param nums: A list of integers.
     * @return: A list of unique permutations.
     */
    public ArrayList<ArrayList<Integer>> permuteUnique(ArrayList
<Integer> nums){
        ArrayList<ArrayList<Integer>> rst = new ArrayList<ArrayL
ist<Integer>>();
        if(nums == null || nums.size() == 0) return rst;

        boolean[] used = new boolean[nums.size()];
        Collections.sort(nums);
        dfs(rst, new ArrayList<Integer>(), nums, used);

        return rst;
    }

        private void dfs(ArrayList<ArrayList<Integer>> rst, List
<Integer> list,
                     ArrayList<Integer> nums, boolean[] used){
        if(list.size() == nums.size()){
            rst.add(new ArrayList<Integer>(list));
            return;
        }

        for(int i = 0; i < nums.size(); i++){
            if(used[i]) continue;
            list.add(nums.get(i));
            used[i] = true;

            dfs(rst, list, nums, used);

            used[i] = false;
```

```
            list.remove(list.size() - 1);
            while(i < nums.size() - 1 && nums.get(i + 1) == nums
.get(i)){
                i ++;
            }
        }
    }
}
```

Subsets & Combinations & Combination Sum

Subsets

Backtracking 不变的经典啊。

唯一注意的地方是，start == nums.length 的时候其实也是要添加结果的，不然会错。写回溯的时候，base case 要想清楚。

```
public class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> rst = new ArrayList<>();

        dfs(rst, new ArrayList<>(), nums, 0);

        return rst;
    }

    private void dfs(List<List<Integer>> rst, List<Integer> list
, int[] nums, int start){
        //if(start > nums.length) return;

        rst.add(new ArrayList<Integer>(list));

        for(int i = start; i < nums.length; i++){
            list.add(nums[i]);

            dfs(rst, list, nums, i + 1);

            list.remove(list.size() - 1);
        }
    }
}
```

Subsets II

有重复元素也不难，先 sort，然后每一步上注意同样的元素别重复加一遍就好了。

```
public class Solution {
    public List<List<Integer>> subsetsWithDup(int[] nums) {
        List<List<Integer>> rst = new ArrayList<>();

        Arrays.sort(nums);

        dfs(rst, new ArrayList<>(), nums, 0);

        return rst;
    }

    private void dfs(List<List<Integer>> rst, List<Integer> list
, int[] nums, int start){
        rst.add(new ArrayList<Integer>(list));

        for(int i = start; i < nums.length; i++){
            list.add(nums[i]);

            dfs(rst, list, nums, i + 1);

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

            while(i + 1 < nums.length && nums[i] == nums[i + 1])
 i++;
        }
    }
}
```

Combinations

这类题的搜索树都是极度向左倾斜的结构，节点数为 2^n;

```java
public class Solution {
    public List<List<Integer>> combine(int n, int k) {
        List<List<Integer>> rst = new ArrayList<>();

        dfs(rst, new ArrayList<Integer>(), n, k, 1);

        return rst;
    }

    private void dfs(List<List<Integer>> rst, List<Integer> list
, int n, int k, int cur){
        if(k == 0){
            rst.add(new ArrayList<>(list));
            return;
        }

        for(int i = cur; i <= n; i++){
            list.add(i);

            dfs(rst, list, n, k - 1, i + 1);

            list.remove(list.size() - 1);
        }
    }
}
```

Combination Sum

为什么觉得我好像在这 gitbook 里写过这题。。

- 元素可以重复选用，传进去的 index 可以以 i 为准。
- 但是还是不能走回头路，不然会有重复答案。
- 所有元素为正整数非常重要，要么这题可能有无数个解。

```
public class Solution {
    public List<List<Integer>> combinationSum(int[] candidates,
int target) {
        List<List<Integer>> rst = new ArrayList<>();

        dfs(rst, new ArrayList<>(), candidates, target, 0, 0);

        return rst;
    }

    private void dfs(List<List<Integer>> rst, List<Integer> list
, int[] nums, int target, int sum, int start){
        if(sum > target) return;
        if(sum == target){
            rst.add(new ArrayList<Integer>(list));
            return;
        }

        for(int i = start; i < nums.length; i++){
            list.add(nums[i]);

            dfs(rst, list, nums, target, sum + nums[i], i);

            list.remove(list.size() - 1);
        }
    }
}
```

Combination Sum II

- 没重复元素，想避免重复解，用 **index** 单调向前；
- 有重复元素，想避免重复解，**index** 之外每轮 **dfs** 末尾直接把指针移到下一个新元素上，一个元素在一轮 **for loop** 里只加一次。

```
public class Solution {
    public List<List<Integer>> combinationSum2(int[] candidates,
int target) {
        List<List<Integer>> rst = new ArrayList<>();

        Arrays.sort(candidates);
        dfs(rst, new ArrayList<>(), candidates, target, 0, 0);

        return rst;
    }

    private void dfs(List<List<Integer>> rst, List<Integer> list
, int[] nums, int target, int sum, int start){
        if(sum > target) return;
        if(sum == target){
            rst.add(new ArrayList<Integer>(list));
            return;
        }

        for(int i = start; i < nums.length; i++){
            list.add(nums[i]);

            dfs(rst, list, nums, target, sum + nums[i], i + 1);

            list.remove(list.size() - 1);
            while(i + 1 < nums.length && nums[i] == nums[i + 1])
 i++;
        }
    }
}
```

Combination Sum III

Trivial problem.

```
public class Solution {
    public List<List<Integer>> combinationSum3(int k, int n) {
        List<List<Integer>> rst = new ArrayList<>();

        dfs(rst, new ArrayList<>(), n, k, 0, 1);

        return rst;
    }

    private void dfs(List<List<Integer>> rst, List<Integer> list
, int target, int k, int sum, int start){
        if(sum > target || k < 0) return;
        if(sum == target && k == 0){
            rst.add(new ArrayList<Integer>(list));
            return;
        }

        for(int i = start; i <= 9; i++){
            list.add(i);

            dfs(rst, list, target, k - 1, sum + i, i + 1);

            list.remove(list.size() - 1);
        }
    }
}
```

Combination Sum IV

题目换成这样了之后依然是一个搜索问题，但是有了新的有趣特性。

主要在于这题和之前的 combination sum 还不太一样，它把 [1,3] 和 [3,1] 这种重复搜索算成两个解。于是强行制造了 overlap subproblems.

于是这题用搜索解会 TLE，加个 hashmap 做记忆化搜索，立刻就过了。

24ms ，时间复杂度较高，不如下面那个迭代的写法速度快。

O(n^d)，d代表得到 >= target 的搜索深度。

```
public class Solution {
    public int combinationSum4(int[] nums, int target) {
        return dfs(nums, 0, target, new HashMap<Integer, Integer
>());
    }

    private int dfs(int[] nums, int curSum, int target, HashMap<
Integer, Integer> map){
        if(curSum > target) return 0;
        if(curSum == target) return 1;

        if(map.containsKey(curSum)) return map.get(curSum);

        int count = 0;
        for(int i = 0; i < nums.length; i++){
            count += dfs(nums, curSum + nums[i], target, map);
        }

        map.put(curSum, count);
        return count;
    }
}
```

论坛里还有比较有趣的迭代写法，只需要 3ms，原理和 climbing stairs 差不多，建一个大小等于 target + 1 的 array 代表多少种不同的方式跳上来，依次遍历即可。

时间复杂度 O(n * target) + O(n log n)

```
public class Solution {
    public int combinationSum4(int[] nums, int target) {
        Arrays.sort(nums);
        int[] res = new int[target + 1];
        for (int i = 1; i < res.length; i++) {
        for (int num : nums) {
            if (num > i)
            break;
        else if (num == i)
            res[i] += 1;
        else
            res[i] += res[i-num];
        }
    }
        return res[target];
    }
}
```

原理和 climbing stairs 差不多，建一个大小等于 target + 1 的 array 代表多少种不同的方式跳上来，依次遍历即可。

时间复杂度 O(n * target)

注意这里内外循环的顺序不能错，要先按 **sum** 从小到大的顺序看，然后才是遍历每个元素。因为所谓 **bottom-up**，是相对于 **sum** 而言的，不是相对于已有元素的 **index** 而言的。

可以看到，在只取 **min/max** 的时候我们不同的循环顺序都能保证正确答案；但是求解的数量时，要以 **target value** 为外循环。

```java
public class Solution {
    public int combinationSum4(int[] nums, int target) {
        // dp[sum] = number of ways to get sum
        int[] dp = new int[target + 1];

        // initialize, one way to get 0 sum with 0 coins
        dp[0] = 1;

        for(int j = 1; j <= target; j++){
            for(int i = 0; i < nums.length; i++){
                if(j - nums[i] >= 0) dp[j] += dp[j - nums[i]];
            }
        }

        return dp[target];
    }
}
```

枚举法与它的马甲们

DFS + Backtracking 的搜索量很大。如果每次 **DFS** 之前都要进行条件检查的话，优化条件函数可以显著提升程序速度。如 **N-Queens** 和 **Palindrome Partitioning** 还有 **Sudoku Solver**

Generate Parentheses

Too trivial. 这题和二叉树其实挺像的，因为在每一个位置都只有两种可能 "(" 和 ")".

```
public class Solution {
    public List<String> generateParenthesis(int n) {
        List<String> list = new ArrayList<String>();

        dfs(list, new StringBuilder(), n, n);

        return list;
    }

    private void dfs(List<String> list, StringBuilder sb, int le
ft, int right){
        if(left == 0 && right == 0){
            list.add(sb.toString());
            return;
        }

        int length = sb.length();

        if(left > 0){
            sb.append('(');
            dfs(list, sb, left - 1, right);
            sb.setLength(length);
        }

        if(right > left){
            sb.append(')');
            dfs(list, sb, left, right - 1);
            sb.setLength(length);
        }

    }
}
```

Restore IP Addresses

依旧是 DFS + Backtracking，每一步有三种可能：选一位数，两位数，还有三位数。

一个需要注意的边界条件是当选取多位数时，第一位不能是0，因为 '020' 或 '05' 都不是有效的 IP 地址。

当 DFS 的可能性比较多，而且只在 index 上有区别的时候，用 for 循环就好了。

```
public class Solution {
    public List<String> restoreIpAddresses(String s) {
        List<String> list = new ArrayList<String>();
        if(s.length() > 12) return list;

        dfs(list, new StringBuilder(), s, 0, 0);

        return list;
    }

    private void dfs(List<String> list, StringBuilder sb,
                     String s, int ipIndex, int strIndex){
        if(ipIndex > 4) return;
        if(ipIndex == 4 && strIndex == s.length()){
            sb.setLength(sb.length() - 1);
            list.add(sb.toString());
            return;
        }

        int length = sb.length();
        // Add one digit number
        for(int i = 0; i < 3; i++){
            if(strIndex + i < s.length()){
                String substr = s.substring(strIndex, strIndex +
1 + i);
                if(isValid(substr)){
                    sb.append(substr);
                    sb.append('.');

                    dfs(list, sb, s, ipIndex + 1, strIndex + 1 +
 i);

                    sb.setLength(length);
```

```
                }
            }
        }
    }

    private boolean isValid(String str){
        if(str.charAt(0) == '0'){
            return str.equals("0");
        }
        int num = Integer.parseInt(str);

        if(num > 0 && num <= 255) return true;
        return false;
    }
}
```

Palindrome Partitioning

一个比较简单暴力的搜索办法就是直接 DFS + Backtracking，每一步上需要考虑的字问题数量和当前字符串剩余长度相等。 这个解法是过不了 LeetCode OJ 的，会超时。

```
 public class Solution {
    public List<List<String>> partition(String s) {
        List<List<String>> rst = new ArrayList<List<String>>();

        dfs(rst, new ArrayList<String>(), s, 0);

        return rst;
    }

    private void dfs(List<List<String>> rst, List<String> list,
String s, int index){
        if(index == s.length()){
            rst.add(new ArrayList<String>(list));
            return;
        }

        for(int i = index; i < s.length(); i++){
            String substr = s.substring(index, i + 1);
            if(!isPalindrome(substr)) continue;

            list.add(substr);
            dfs(rst, list, s, i + 1);
            list.remove(list.size() - 1);
        }
    }

    private boolean isPalindrome(String str){
        int left = 0;
        int right = str.length() - 1;
        while(left > right){
            if(str.charAt(left) != str.charAt(right)) return fal
se;
            left ++;
            right --;
        }

        return true;
    }
}
```

于是发现每次都要跑一个 isPalindrome() 太耗时间了，同一个字符串会被检查很多次，不如用 DP 把任意两个 index 之间的字符串缓存下，加速 isPalindrome() 的验证。

然并卵，还是超时。

主要原因在于，从每一个字符作为起点向两边扩张的赋值方式，还是不够快。

```
public class Solution {
    public List<List<String>> partition(String s) {
        List<List<String>> rst = new ArrayList<List<String>>();

        boolean[][] dp = new boolean[s.length()][s.length()];
        for(int i = 0; i < s.length(); i++){
            dp[i][i] = true;
            int left = i - 1;
            int right = i + 1;
            while(left >= 0 && right < s.length()){
                if(s.charAt(left) == s.charAt(right))
                    dp[left][right] = true;

                left --;
                right ++;
            }
        }

        dfs(rst, new ArrayList<String>(), s, 0, dp);

        return rst;
    }

    private void dfs(List<List<String>> rst, List<String> list,
String s, int index, boolean[][] dp){
        if(index == s.length()){
            rst.add(new ArrayList<String>(list));
            return;
        }
```

```
        for(int i = index; i < s.length(); i++){
            if(!dp[index][i]) continue;

            String substr = s.substring(index, i + 1);
            list.add(substr);
            dfs(rst, list, s, i + 1, dp);
            list.remove(list.size() - 1);
        }
    }

}
```

于是就有了这段代码，思路参考了 LeetCode 论坛，里面的小伙伴们好机智啊。

dp[i][j] 代表从index i ~ j 的字符串是不是 palindrome. dp[i][j] = dp[j][i].

其实也可以只填 dp[i][j]，也就是矩阵的 lower half 三角形，不过到时候调用的时候要注意放 index 的先后顺序，不然结果会错。求稳的话，就一次都设一样的吧。

(7/6 号) 回头复习过程中发现，这就是做 search 类题，遇到了记忆化搜索啊。

在 palindrome 问题中，子串之间有天然的 optimal substructure，很适合利用 dp 做预处理解决。

建立一个 String 的所有 substring 是否为 palindrome 的矩阵，时间复杂度为等差数列，约为 $$0.5 n^2$$ (矩阵沿对角线划分的一半区域)

```
public class Solution {
    public List<List<String>> partition(String s) {
        List<List<String>> rst = new ArrayList<List<String>>();

        boolean[][] dp = new boolean[s.length()][s.length()];
        for(int i = 0; i < s.length(); i++){
            for(int j = 0; j <= i; j++){
                if(s.charAt(i) == s.charAt(j) && (i - j <= 2 ||
dp[i - 1][j + 1]))
                    dp[i][j] = dp[j][i] = true;
            }
        }

        dfs(rst, new ArrayList<String>(), s, 0, dp);

        return rst;
    }

    private void dfs(List<List<String>> rst, List<String> list,
String s, int index, boolean[][] dp){
        if(index == s.length()){
            rst.add(new ArrayList<String>(list));
            return;
        }

        for(int i = index; i < s.length(); i++){
            if(!dp[i][index]) continue;

            String substr = s.substring(index, i + 1);
            list.add(substr);
            dfs(rst, list, s, i + 1, dp);
            list.remove(list.size() - 1);
        }
    }

}
```

Letter Combinations of a Phone Number

有个小地方要注意，终止条件是

```
sb.length() == digits.length()
```

而不是

```
pos >= digits.length()
```

不然在 test case 为 “22” 的时候在返回了所有正确答案之后会多出来一组 "a","b","c"

```
public class Solution {
    public List<String> letterCombinations(String digits) {
        List<String> list = new ArrayList<String>();
        if(digits == null || digits.length() == 0) return list;
        dfs(list, new StringBuilder(), digits, 0);

        return list;
    }

    private void dfs(List<String> list, StringBuilder sb, String
 digits, int pos){
        if(sb.length() == digits.length()){
            list.add(sb.toString());
            return;
        }

        for(int i = pos; i < digits.length(); i++){
            String str = getAlphabets(digits.charAt(i));
            for(int j = 0; j < str.length(); j++){
                int length = sb.length();
                sb.append(str.charAt(j));
                dfs(list, sb, digits, i + 1);
                sb.setLength(length);
            }
        }
    }
```

```
    private String getAlphabets(Character digit){
        if(digit == '2') return "abc";
        if(digit == '3') return "def";
        if(digit == '4') return "ghi";
        if(digit == '5') return "jkl";
        if(digit == '6') return "mno";
        if(digit == '7') return "pqrs";
        if(digit == '8') return "tuv";
        if(digit == '9') return "wxyz";

        return "";
    }
}
```

N 皇后及其 Index Trick

N-Queens I

非常经典的 N 皇后问题。我自己写了个能用的版本，但是没有 LeetCode 论坛上这个人的解法简洁明了。思路倒是一样的。

我一开始尝试优化，进行剪枝减少每次需要搜索的格子，比如正解中的后与后之间一定是一个“马步”，因此只考虑在一个后被放置之后其他的“马步”位置。 然而这样会漏掉一些解，因为一个正解中不是所有后之间都是“马步”联系起来的，而更像在 graph 中多个 strongly connected components 的状态，如图

One solution to the eight queens puzzle

因此不能单凭“马步”去保证下一个正确的皇后位置，还是要老老实实遍历搜索。

在每步遍历搜索中， 都要 call 一次 isValid() 函数来判断当前格子是否有效，在 dfs + backtracking 的结构不变的情况下，isValid()函数的性能就会成为程序的瓶颈。

一个比较简单直接的办法是，维护一个 list 代表当前的皇后位置（每一个位置都是 valid）在考虑增加新一个王后时，和 list 里已有的每一个皇后比较是否 valid. 问题在于这个 list 的大小是和我们的棋盘大小 n 成正比的，假设一个 list 从0个皇后一直增加到n个皇后，在这个过程中要进行 $$1 + 2 + 3 ... + n = O(n^2)$$ 次 isValid 操作。

所以一个更取巧的方式是，利用一个皇后会占用这个格子的所有行，列以及对角线的特点，维护所有“行，列，45度对角线，135度对角线” 的 boolean array，这样在每次考虑一个新位置的时候，只需要 $$O(1)$$ 的时间进行一次操作就可以了。

boolean[] 完全可以，Java 的 BitSet 也不错，不过我实际跑的结果来看，BitSet 在用时上会慢一些，原因在这个帖子：

Boolean-vs-bitset-which-is-more-efficient

boolean[] 一个 **boolean** 占用大约 **1 byte** 空间（**JVM dependent**），**BitSet** 平均占用 **1 bit. boolean[]** 的优点是更 **CPU efficient,** 只有在数组非常大（**1000w**以上）二者的速度持平。**BitSet** 内部实现是建立 **long[]** ，用 **64-bit** 的块去分配 **BitSet** 的字节。

- **45**度对角线（左上到右下）的 **row + column** 值是恒等的（一加一减），并且每条对角线的 **row + column** 值唯一, 在 **$$[0,1,2 ... 2n - 2]$$** 区间
- **135**度对角线（左下到右上）的 **row - column** 值是恒等的（同加同减），并且每条对角线的 **row - column** 值唯一，在 **$$[-（n-1） ... -2,-1,0, 1, 2 ... n - 1]$$** 区间
- 边长为 **$$n$$** 的棋盘，有 **$$2n - 1$$** 条对角线，并且中间值 **$$(0,0)$$** 的 **$$index$$** 为 **$$n - 1$$**

```
public class Solution {
    public List<List<String>> solveNQueens(int n) {
        boolean[]
            //ocp0 = new boolean[n],
            ocp90 = new boolean[n],
            // 总共 2n - 1 种可能性，(0,0) 的 index 是 n - 1
            ocp45 = new boolean[2 * n - 1],
            ocp135 = new boolean[2 * n - 1];
        List<List<String>> ans = new ArrayList<List<String>>();
        char[][] map = new char[n][n];
        for (char[] tmp : map) Arrays.fill(tmp, '.'); //init

        solve(0, n, map, ans, ocp45, ocp90, ocp135);
        return ans;
```

```
    }

    private void solve(int depth, int n, char[][] map, List<List
<String>> ans,
    boolean[] ocp45, boolean[] ocp90, boolean[] ocp135) {
        if (depth == n) {
            addSolution(ans, map);
            return;
        }

        for (int j = 0; j < n; j++)
        // 每次都进行新的一行，所以行不用检查
        // 90度代表棋盘的列
        // 45度对角线（从左上到右下）同一条线上 row + col 是相等的，因此
可用作 index
        // 135度对角线（从左下到右上）可从 (0,0) 即 index (n - 1) 为起点

        // 因为同一条对角线 row - col 值恒定，可用作 offset 表示对角线
的 index.
            if (!ocp90[j] && !ocp45[depth + j] && !ocp135[j - de
pth + n - 1]) {
                ocp90[j] = true;
                ocp45[depth + j] = true;
                ocp135[j - depth + n - 1] = true;
                map[depth][j] = 'Q';

                solve(depth + 1, n, map, ans, ocp45, ocp90, ocp1
35);

                ocp90[j] = false;
                ocp45[depth + j] = false;
                ocp135[j - depth + n - 1] = false;
                map[depth][j] = '.';
            }
    }

    private void addSolution(List<List<String>> ans, char[][] ma
p) {
        List<String> cur = new ArrayList<String>();
        for (char[] i : map) cur.add(String.valueOf(i));
```

```
            ans.add(cur);
        }
    }
```

N-Queens II

很可惜，这题解与解之间不存在 optimal substructure 可以有效利用记忆化搜索。老老实实的 DFS + backtracking 还是必要的。有一些比较妖孽的解法比如 Kunth's 的 Dancing Links，刷题面试的时候，这种大招还是没必要去开了。

既然不需要输出最终解，也就省去了传递 rst 和把棋盘转成 String 的过程。一个问题是，如何在这种dfs + backtracking中维护一个 primitive type 变量，比如有效解的总数？

Java 中默认 primitive type 是 pass by value，而且 Integer type 虽然作为一个 object 存在但是是 immutable, 不能用来作为全局参数多个函数共同更新。

直接建全局变量太蠢了，也不好看。

只求解的数量时，可以让 **dfs** 函数直接返回 **int**，每次迭代的时候把其他迭代的返回结果加起来就好了。

```
public class Solution {
    public int totalNQueens(int n) {
        return dfs(0, n, new boolean[n], new boolean[2*n - 1], n
ew boolean[2*n - 1]);
    }

    private int dfs(int row, int n, boolean[] cols, boolean[] di
ag45, boolean[] diag135){
        if(row == n){
            return 1;
        }

        int solutionCount = 0;
        for(int col = 0; col < n; col++){
            if(!cols[col] && !diag45[row + col] && !diag135[row
- col + n - 1]){
                cols[col] = true;
                diag45[row + col] = true;
                diag135[row - col + n - 1] = true;

                solutionCount += dfs(row + 1, n, cols, diag45, d
iag135);

                cols[col] = false;
                diag45[row + col] = false;
                diag135[row - col + n - 1] = false;
            }
        }
        return solutionCount;
    }
}
```

数独与矩阵 index trick

Valid Sudoku

这题本身不是很难，也有很多写法都可以 AC. 但是不同解法的简洁程度不同，也能体现出对二维矩阵结构理解上的差异。

Sudoku 是一个结构比较特殊的二维矩阵，因为作为一个正方形矩阵，大多数问题只需要考虑“行”“列”与“对角线”(N-Queens)，而 Sudoku 需要还考虑“子矩阵”，也就是说需要另外的 index trick 去简洁高效的处理“遍历子矩阵”的问题。

下面的代码来自 LeetCode 论坛，简洁准确，值得学习~

x = k / 3 + (i / 3) * 3;

y = k % 3 + (i % 3) * 3;

一个二维矩阵可以变成一维向量，通过 i / size 表示“行”，i % size 表示“列”。

把 Sudoku 以“子矩阵”为单位去想的话，是一个新的 3x3 二维矩阵

(i / 3) * 3 可以作为一个“子矩阵”的 offset，代表在 3x3 子矩阵中的“行”，

(i % 3) * 3 代表子矩阵的“列”。

由于最外圈循环为 i，给定 i 之后我们要实现每个 block 的遍历，因此和 i 相关的 index 用于定位 block.

而后再去加上矩阵本身的“行” k / 3 与 “列” k % 3 实现对于每一个原矩阵坐标 (i, k)，得到对应的子矩阵坐标 (x,y) 映射。

```java
public class Solution {
    public boolean isValidSudoku(char[][] board) {
        for(int i = 0; i < 9; i++){
            boolean[] row = new boolean[9];
            boolean[] col = new boolean[9];

            for(int j = 0; j < 9; j++){
                if(board[i][j] != '.'){
                    int num = board[i][j] - '1';
                    if(row[num]) return false;

                    row[num] = true;
                }

                if(board[j][i] != '.'){
                    int num = board[j][i] - '1';
                    if(col[num]) return false;

                    col[num] = true;
                }
            }

            boolean[] cell = new boolean[9];
            for(int k = 0; k < 9; k++){
               int x = k / 3 + (i / 3) * 3;
               int y = k % 3 + (i % 3) * 3;

               if(board[x][y] != '.'){
                   int num = board[x][y] - '1';
                   if(cell[num]) return false;

                   cell[num] = true;
               }
            }
        }

        return true;
    }
}
```

Sudoku Solver

这个解借鉴的 LeetCode 论坛答案，思路和 N-Queens 很像，不同之处是存的所有 row, col 和 block.

这个答案据作者 自己描述 在老版本LeetCode Java 上没太大区别，然而 C++ 比较明显。

这题因为检查 “isValid” 的次数非常多，要直接开一个 2D array 了，matrix[index][num] 代表“第 index 行/列/子矩阵” 的“数字num”

因此在 block 的 index 上面可以更取巧一些，给定一个原始矩阵的位置，直接用 block = col / 3 + (row / 3) * 3 就可以代表一个具体 block.

rows[i][num] = cols[j][num] = blocks[k][num] = true;

另外一个很重要的一点是，不是每条 dfs 的搜索都会返回最终的解，因为很多尝试最后会进入死胡同无法解出来，因此每一层 dfs 不能只是单纯的 void 不返回任何值，而是要返回 true / false 代表这条路线走下去是否能得到正确答案。

```
public class Solution {
   public void solveSudoku(char[][] board) {
       boolean[][] rows = new boolean[9][9];
       boolean[][] cols = new boolean[9][9];
       boolean[][] blocks = new boolean[9][9];

       for(int i = 0; i < 9; i++){
           for(int j = 0; j < 9; j++){
               if(board[i][j] != '.'){
                   int num = board[i][j] - '1';
                   int k = j / 3 + (i / 3) * 3;
                   rows[i][num] = cols[j][num] = blocks[k][num]
= true;
               }
           }
       }
```

```
        dfs(0, board, rows, cols, blocks);
    }

    private boolean dfs(int index, char[][] board, boolean[][] r
ows,
                        boolean[][] cols, boolean[][] blocks){
        if(index == 81) return true;
        int row = index / 9;
        int col = index % 9;
        int block = col / 3 + (row / 3) * 3;

        if(board[row][col] != '.') {
            return dfs(index + 1, board, rows, cols, blocks);
        } else {
            for(char chr = '1'; chr <= '9'; chr ++){
                int num = chr - '1';
                if(!rows[row][num] && !cols[col][num] && !blocks
[block][num]){
                    board[row][col] = chr;
                    rows[row][num] = true;
                    cols[col][num] = true;
                    blocks[block][num] = true;

                    if(dfs(index + 1, board, rows, cols, blocks)
) return true;

                    board[row][col] = '.';
                    rows[row][num] = false;
                    cols[col][num] = false;
                    blocks[block][num] = false;
                }
             }
        }

        return false;
    }
}
```

5/18 搜索类，Word Ladder I & II

Word Ladder I

著名 gay 题，时间复杂度爆炸的典范，也是目前我知道的 leetcode 题里唯一一道非常适合使用 bi-directional bfs 的问题。

比较独特的是这题不是 DFS + Backtracking，主要原因在于我们要确保“路线最短”，这个更适合用 BFS 解决，两层字典首次相交的地方一定是最短路线。

```
public class Solution {
    public int ladderLength(String beginWord, String endWord, Se
t<String> wordList) {
        if(beginWord.length() != endWord.length()) return 0;
        if(wordList == null || wordList.size() == 0) return 0;
        if(beginWord.equals(endWord)) return 2;

        HashSet<String> beginSet = new HashSet<String>();
        HashSet<String> endSet = new HashSet<String>();

        beginSet.add(beginWord);
        endSet.add(endWord);

        wordList.remove(beginWord);
        wordList.remove(endWord);

        return bfs(2, beginSet, endSet, wordList);
    }

    private int bfs(int steps, Set<String> beginSet, Set<String>
 endSet, Set<String> wordSet){

        HashSet<String> nextLevel = new HashSet<String>();

        for(String beginWord : beginSet){
            char[] charArray = beginWord.toCharArray();
```

```
            for(int pos = 0; pos < charArray.length; pos++){
                for(char chr = 'a'; chr <= 'z'; chr++){
                    charArray[pos] = chr;
                    String str = String.valueOf(charArray);
                    if(endSet.contains(str)) return steps;

                    if(wordSet.contains(str)){
                        nextLevel.add(str);
                        wordSet.remove(str);
                    }
                }
                charArray = beginWord.toCharArray();
            }
        }
        if(nextLevel.size() == 0) return 0;

        if(endSet.size() < nextLevel.size()){
            return bfs(steps + 1, endSet, nextLevel, wordSet);
        } else {
            return bfs(steps + 1, nextLevel, endSet, wordSet);
        }
    }
}
```

除了递归之外，还可以用迭代。反正迭代的 BFS 好写。

```
public class Solution {
    public int ladderLength(String beginWord, String endWord, Se
t<String> wordList) {
        if(beginWord.length() != endWord.length()) return 0;
        if(wordList == null || wordList.size() == 0) return 0;
        if(beginWord.equals(endWord)) return 2;

        HashSet<String> beginSet = new HashSet<String>();
        HashSet<String> endSet = new HashSet<String>();

        beginSet.add(beginWord);
        endSet.add(endWord);
```

```
        wordList.remove(beginWord);
        wordList.remove(endWord);

        int steps = 2;

        while(!beginSet.isEmpty()){
            HashSet<String> nextLevel = new HashSet<String>();
            for(String word : beginSet){
                char[] charArray = word.toCharArray();
                for(int pos = 0; pos < charArray.length; pos++){
                    for(char chr = 'a'; chr <= 'z'; chr++){
                        charArray[pos] = chr;
                        String str = String.valueOf(charArray);
                        if(endSet.contains(str)) return steps;

                        if(wordList.contains(str)){
                            nextLevel.add(str);
                            wordList.remove(str);
                        }
                    }
                    charArray = word.toCharArray();
                }
            }

            steps ++;
            if(nextLevel.size() < endSet.size()){
                beginSet = nextLevel;
            } else {
                beginSet = endSet;
                endSet = nextLevel;
            }
        }

        return 0;
    }
}
```

Word Ladder II

LeetCode 著名 gay 题的加强版，烦人程度更上一层楼。。。当要返回所有 path 的时候，细节处理就更多了。其实这题和 Longest Increasing Subsequence 的 follow-up ，返回所有 LIS 有相通的地方，都是维护一个 Directed Graph 关系并且从某一个终点进行 dfs 返回所有 valid path.

这题的最终解法是一个 Bi-directional BFS 与 DFS 的结合，因为 bfs 过程中起点终点会调换，为了保证 path 的正确性要 keep track of direction.

有向图找最短距离用 BFS

有向图返回所有路径用 DFS

HashMap 中 value 是 List 可以直接 map.get(key).add(value);

```
public class Solution {

    boolean isConnected = false;

    public List<List<String>> findLadders(String beginWord, Stri
ng endWord, Set<String> wordList) {

        Set<String> top = new HashSet<String>();
        top.add(beginWord);

        Set<String> bot = new HashSet<String>();
        bot.add(endWord);

        Map<String, List<String>> map = new HashMap<String, List
<String>>();
        // BFS 在这里只是起到一个更新 hashmap 的作用，hashmap 存着所有
有效的 adjacency list
        bfs(top, bot, wordList, false, map);

        List<List<String>> rst = new ArrayList<List<String>>();
```

```
        if(!isConnected) return rst;

        List<String> list = new ArrayList<String>();
        list.add(beginWord);

        // DFS 负责返回并写入所有 paths
        dfs(rst, list, beginWord, endWord, map);

        return rst;
    }

    // @param swap: whether we are pointing at the right directi
on, start->end
    //              since we use bi-directional BFS it's necessa
ry to keep track of
    //              directions in hashmap to construct paths
    public void bfs(Set<String> top, Set<String> bot, Set<String
> dict, boolean swap, Map<String, List<String>> map){

        HashSet<String> nextLevel = new HashSet<String>();

        // 避免搜索重复元素，免得搜索路径上出现环
        dict.removeAll(top);
        dict.removeAll(bot);

        for(String src : top){
            for(int pos = 0; pos < src.length(); pos++){
                char[] array = src.toCharArray();
                for(char chr = 'a'; chr <= 'z'; chr++){
                    if(array[pos] == chr) continue;
                    array[pos] = chr;
                    String next = String.valueOf(array);

                    // 决定 src 与 next 两个单词的 directed edge 方向

                    String key = (swap) ? next: src;
                    String value = (swap) ? src: next;

                    if(!map.containsKey(key)) map.put(key, new A
```

```
rrayList<String>());

                    // 前面已经把先后顺序确定了，map更新就老老实实 key
- value 就好
                    if(bot.contains(next)){
                        map.get(key).add(value);
                        isConnected = true;
                    }
                    if(dict.contains(next)){
                        map.get(key).add(value);
                        nextLevel.add(next);
                    }
                }
            }

        }

        if(isConnected || nextLevel.size() == 0) return;

        // 这里直接扔 true / false 是不对的，应该用原来传进来的变量 swa
p 来决定
        if(nextLevel.size() <= bot.size()){
            bfs(nextLevel, bot, dict, swap, map);
        } else {
            bfs(bot, nextLevel, dict, !swap, map);
        }

    }

    public void dfs(List<List<String>> rst, List<String> list, S
tring start, String end, Map<String, List<String>> map){
        if(start.equals(end)){
            rst.add(new ArrayList<String>(list));
            return;
        }

        // 没有这行会导致 null pointer exception，毕竟不是每个 word
都在 hashmap 里
        // word 自己是叶节点，死胡同
        if(!map.containsKey(start)) return;
```

```
        for(String next : map.get(start)){
            list.add(next);
            dfs(rst, list, next, end, map);
            list.remove(list.size() - 1);
        }
    }
}
```

Number of ways 问题类

一般来讲这类问题 dp 比较多，因为很多时候只要返回数量的话，并不一定需要穷举所有情况，有很多利用 subproblem 重复还有对称性的 trick 可以用。

Android Unlock Patterns

这题作为 google 近期热点题，挺有意思的，有趣在里面的细节和考点很多。

- 这个更多算是误会，实际上的连线没有那么复杂，走一个马步形状是不算 go through 的，因此最简单的办法，反正键盘数量有限，真正需要考虑的线路总共就 8 条，开个数组写出来就好了。
    - 如果 passThrough 是 0 ，说明 i，j 的连线不过其他点；
    - 其他情况说明 passThrough 的值就是路过需要检查的点。
- **这里的 dfs 结构更接近于 Tree 的形式，传进去的参数是当前在考虑的元素(但是已经确认是 valid input)，还没有做任何 visited 的标记和计数。因此有别于大多数其他 backtracking 的问题在循环里 backtrack，这个是在 dfs 函数自身的开始 / 结束处做 backtracking.**
- 另外一点是，我们有一个长度区间，而不仅仅是像大多数搜索问题那样，找到叶节点 / 找到解直接返回。这题当我们找到一个合理解的时候，需要继续往下探索，同时把以当前节点为起点，下面所有在合理深度范围的路径个数融合返回。
- 处理这个问题的一个简单，稍显暴力的做法，也是下面代码的做法，就是每次搜索时候固定深度，搜到目标深度就不搜了，返回 1 就行。优点是这样代码很好写；缺点是效率不高，因为同一个路径其实探了好几次。尤其在 “返回所有路径” 的搜索模式中，这种做法显然是不占优势的。

```
public class Solution {
    public int numberOfPatterns(int m, int n) {
```

```
        int[][] passThrough = new int[10][10];
        boolean[] visited = new boolean[10];
        // Rows
        passThrough[1][3] = passThrough[3][1] = 2;
        passThrough[4][6] = passThrough[6][4] = 5;
        passThrough[7][9] = passThrough[9][7] = 8;
        // Cols
        passThrough[1][7] = passThrough[7][1] = 4;
        passThrough[2][8] = passThrough[8][2] = 5;
        passThrough[3][9] = passThrough[9][3] = 6;
        // Diagnals
        passThrough[1][9] = passThrough[9][1] = 5;
        passThrough[3][7] = passThrough[7][3] = 5;

        int count = 0;

        for(int i = m; i <= n; i++){
            count += 4*dfs(visited, passThrough, 1, i - 1) +
                     4*dfs(visited, passThrough, 2, i - 1) +
                     dfs(visited, passThrough, 5, i - 1);
        }

        return count;
    }

    // take currently considering element
    private int dfs(boolean[] visited, int[][] passThrough, int
curNum, int lenRemain){
        if(lenRemain < 0) return 0;
        if(lenRemain == 0) return 1;

        int count = 0;
        visited[curNum] = true;
        for(int i = 1; i <= 9; i++){
            if(!visited[i] && (passThrough[curNum][i] == 0 || vi
sited[passThrough[curNum][i]])){
                count += dfs(visited, passThrough, i, lenRemain
- 1);
            }
        }
```

```
        visited[curNum] = false;

        return count;
    }
}
```

另一种沿途计数的代码实现是这样，具体细节处理我还是得学习一个。。这个写法就很适合输出所有路径了，改一下执行条件即可。

- 为什么初始传进去的 curLen = 1?
- 因为能作为参数传进去的 num ，都是valid move，当前的有效长度其实已经是 len + 1 了。
- 我们做的是一个 Tree 类 DFS，带着一个一定合理但又没加进去的元素进 DFS，长度 +1， 只在函数最开始和末尾做 backtracking.

```
public class Solution {
    public int numberOfPatterns(int m, int n) {
        int[][] passThrough = new int[10][10];
        boolean[] visited = new boolean[10];
        // Rows
        passThrough[1][3] = passThrough[3][1] = 2;
        passThrough[4][6] = passThrough[6][4] = 5;
        passThrough[7][9] = passThrough[9][7] = 8;
        // Cols
        passThrough[1][7] = passThrough[7][1] = 4;
        passThrough[2][8] = passThrough[8][2] = 5;
        passThrough[3][9] = passThrough[9][3] = 6;
        // Diagnals
        passThrough[1][9] = passThrough[9][1] = 5;
        passThrough[3][7] = passThrough[7][3] = 5;

        return 4*dfs(visited, passThrough, 1, 1 , m, n) +
               4*dfs(visited, passThrough, 2, 1, m, n) +
               dfs(visited, passThrough, 5, 1,  m, n);
    }

    //
    private int dfs(boolean[] visited, int[][] passThrough, int
curNum, int curLen, int m, int n){
```

```
        int cumuCount = 0;
        if(curLen >= m) cumuCount++;
        if(curLen >= n) return cumuCount;

        visited[curNum] = true;
        curLen ++;
        for(int i = 1; i <= 9; i++){
            if(!visited[i] && (passThrough[curNum][i] == 0 || vi
sited[passThrough[curNum][i]])){
                cumuCount += dfs(visited, passThrough, i, curLen
, m, n);
            }
        }
        visited[curNum] = false;

        return cumuCount;
    }
}
```

Combination Sum IV

题目换成这样了之后依然是一个搜索问题，但是有了新的有趣特性。

主要在于这题和之前的 combination sum 还不太一样，它把 [1,3] 和 [3,1] 这种重复搜索算成两个解。于是强行制造了 overlap subproblems.

于是这题用搜索解会 TLE，加个 hashmap 做记忆化搜索，立刻就过了。

24ms ，时间复杂度较高，不如下面那个迭代的写法速度快。

O(n^d)，d代表得到 >= target 的搜索深度。

```
public class Solution {
    public int combinationSum4(int[] nums, int target) {
        return dfs(nums, 0, target, new HashMap<Integer, Integer
>());
    }

    private int dfs(int[] nums, int curSum, int target, HashMap<
Integer, Integer> map){
        if(curSum > target) return 0;
        if(curSum == target) return 1;

        if(map.containsKey(curSum)) return map.get(curSum);

        int count = 0;
        for(int i = 0; i < nums.length; i++){
            count += dfs(nums, curSum + nums[i], target, map);
        }

        map.put(curSum, count);
        return count;
    }
}
```

考虑到可能的有效解路径一定是 [0,target] 之间，我们可以开一个定长的 array 做 hash，在 LC 上速度就快很多，1ms.

```
public class Solution {
    public int combinationSum4(int[] nums, int target) {
         int[] count = new int[target + 1];
         Arrays.fill(count, -1);
         return combinationSumHelper(nums, target, count);
    }

    private int combinationSumHelper(int[] candidates, int targe
t, int[] count){
        if(target == 0) return 1;
        if(target < 0) return 0;
        if(count[target] != -1) return count[target];

        int sum = 0;
        for(int i = 0; i < candidates.length; i++){
            sum += combinationSumHelper(candidates, target - can
didates[i],count);
        }

        count[target] = sum;
        return count[target];
     }
}
```

另外一种继承了背包类问题思想(Backpack III，单个元素无限取) 的 bottom-up 循环解法，非常的简洁：

时间复杂度 O(n * target)

注意这里内外循环的顺序不能错，要先按 **sum** 从小到大的顺序看，然后才是遍历每个元素。因为所谓 **bottom-up**，是相对于 **sum** 而言的，不是相对于已有元素的 **index** 而言的。

```
public class Solution {
    public int combinationSum4(int[] nums, int target) {
        // dp[sum] = number of ways to get sum
        int[] dp = new int[target + 1];

        // initialize, one way to get 0 sum with 0 coins
        dp[0] = 1;

        for(int j = 1; j <= target; j++){
            for(int i = 0; i < nums.length; i++){
                if(j - nums[i] >= 0) dp[j] += dp[j - nums[i]];
            }
        }

        return dp[target];
    }
}
```

Decode Ways

第一遍写的，改了好多次，如果是面试第一次遇到的话，遇到这类题切记“细心”。

特殊 case : 11,101,110, 501..

- 如果遇到有歧义的情况，原理和 **climbing stairs** 类似，当前位置的合理解个数要考虑到前面两个子问题的合理解个数，即 **dp[i]** 要看 **dp[i - 1]** 和 **dp[i - 2];**
- 前一位不是 **0** ，并且能和当前 **digit** 组成合理字母，就 **dp[i] = dp[i - 1] + dp[i - 2];**
- 前一位组不了，就不产生新路线，**dp[i] = dp[i - 1];**
- 重点在于 **"0"** 的处理。
    - 开头直接遇到 **0**，返回 **0**；

- 任何位置连续遇到两个 **0** ，无解，返回 **0**；
- 前一位数不能和当前 **0** 组成字母，无解，返回 **0**；
- 前一位数能和当前的 **0** 组成字母，**dp[i] = dp[i - 2];**

```
    public int numDecodings(String s) {
        if(s == null || s.length() == 0) return 0;
        if(s.charAt(0) == '0') return 0; // Can't start with 0

        int[] dp = new int[s.length() + 1];
        dp[0] = 1;
        dp[1] = 1;

        for(int i = 1; i < s.length(); i++){
            int cur = s.charAt(i) - '0';
            int prev = s.charAt(i - 1) - '0';
            int num = prev * 10 + cur;

            if(cur == 0){
                if(prev == 0) return 0; // Error - 00
                else if(num <= 26) dp[i + 1] = dp[i - 1]; // we
can't discard current 0
                else return 0; // 40, 50, etc.
            } else {
                if(prev != 0 && num <= 26) dp[i + 1] = dp[i] + d
p[i - 1]; // "101" = 1 way
                else dp[i + 1] = dp[i];
            }
        }

        return dp[s.length()];
    }
```

LC 论坛里关于这题还有好多种其他解法，比较简洁的有如下两个：

从左向右：

" "
1
12
120
1203
12031
120314
20
1
2
12
3
4
14

```
public int numDecodings(String s) {
        if(s == null || s.length() == 0) return 0;
        if(s.charAt(0) == '0') return 0;

        int n = s.length();
        int[] dp = new int[n + 1];

        dp[0] = 1;
        dp[1] = 1;

        for(int i = 2; i <= n; i++) {
            int oneDigit = Integer.valueOf(s.substring(i-1, i));
            int twoDigits = Integer.valueOf(s.substring(i-2, i))
;
            if(oneDigit >= 1 && oneDigit <= 9) {
               dp[i] += dp[i - 1];
            }
            if(twoDigits >= 10 && twoDigits <= 26) {
                dp[i] += dp[i - 2];
            }
        }
        return dp[n];
    }
```

从右向左：

```
    public int numDecodings(String s) {
        int n = s.length();
        if (n == 0) return 0;

        int[] memo = new int[n+1];
        memo[n]  = 1;
        memo[n-1] = s.charAt(n-1) != '0' ? 1 : 0;

        for (int i = n - 2; i >= 0; i--)
            if (s.charAt(i) == '0') continue;
            else memo[i] = (Integer.parseInt(s.substring(i,i+2))
<=26)
                            ? memo[i+1]+memo[i+2]
                            : memo[i+1];

        return memo[0];
    }
```

5/19 搜索类，DFS flood filling

Word Search

这类在 2D 矩阵上 DFS 的题都挺送分的，写的时候重在简洁。 和这题非常相似的还有下面的 Number of Islands.

边界处理在 **DFS** 一开始做，免得后面的多向 **DFS** 里再重写

如果要 **backtrack**，就先把自己格子设成其他字符再 **DFS**，免得死循环

```
public class Solution {
    public boolean exist(char[][] board, String word) {
        for(int i = 0; i < board.length; i++){
            for(int j = 0; j < board[0].length; j++){
                if(dfs(board, word, 0, i, j)) return true;
            }
        }
        return false;
    }

    private boolean dfs(char[][] board, String word, int index,
int row, int col) {
        if(index == word.length()) return true;
        if(row < 0 || row >= board.length) return false;
        if(col < 0 || col >= board[0].length) return false;
        if(board[row][col] != word.charAt(index)) return false;


        board[row][col] = '#';

        boolean rst = (
            dfs(board, word, index + 1, row - 1, col) ||
            dfs(board, word, index + 1, row + 1, col) ||
            dfs(board, word, index + 1, row, col + 1) ||
            dfs(board, word, index + 1, row, col - 1)
                      );

        board[row][col] = word.charAt(index);

        return rst;
    }
}
```

Number of Islands

和上一道 Word Search 就完全是一个套路，只是细节不同，这题 DFS 的主要动作是 flood-filling，把相连接的所有格子都填上，算作一个岛。既然不是 search 就不需要 backtracking 那一套了，填就行。

```
public class Solution {
    public int numIslands(char[][] grid) {
        int count = 0;
        for(int i = 0; i < grid.length; i++){
            for(int j = 0; j < grid[0].length; j++){
                if(grid[i][j] == '1'){
                    dfs(grid, i , j);
                    count ++;
                }
            }
        }

        return count;
    }

    private void dfs(char[][] grid, int x, int y){
        if(x < 0 || x >= grid.length) return;
        if(y < 0 || y >= grid[0].length) return;

        if(grid[x][y] == '0') return;

        grid[x][y] = '0';

        dfs(grid, x + 1, y);
        dfs(grid, x - 1, y);
        dfs(grid, x, y + 1);
        dfs(grid, x, y - 1);
    }
}
```

Surrounded Regions

第一次写的时候用了个 HashSet 记录哪些点访问过，显得麻烦，还浪费了额外空间。

- 这种在矩阵上做 **flood filling** 的问题，可以靠自定义字符做标记，取代用额外空间的记录方式。

这段代码的逻辑就是从四个边开始碰到 'O' 就往里扫，把扫到的都标上 'S' 代表有效湖；最后过一遍的时候除了 'S' 的都标成 'X' 就好了。

然而在大矩阵上 stackoverflow... 看来无脑 dfs 的做法还不够经济啊。。。

```
public class Solution {
    public void solve(char[][] board) {
        if(board == null || board.length == 0) return;
        int rows = board.length;
        int cols = board[0].length;

        for(int i = 0; i < cols; i++){
            if(board[0][i] == 'O'){
                dfs(board, 0, i);
            }
            if(board[rows - 1][i] == 'O'){
                dfs(board, rows - 1, i);
            }
        }

        for(int i = 1; i < rows - 1; i++){
            if(board[i][0] == 'O'){
                dfs(board, i, 0);
            }
            if(board[i][cols - 1] == 'O'){
                dfs(board, i, cols - 1);
            }
        }

        for(int i = 0; i < rows; i++){
            for(int j = 0; j < cols; j++){
                if(board[i][j] == 'S'){
                    board[i][j] = 'O';
                } else {
                    board[i][j] = 'X';
                }
            }
        }

    }
```

```
    private void dfs(char[][] board, int row, int col){
        if(row < 0 || row >= board.length) return;
        if(col < 0 || col >= board[0].length) return;
        if(board[row][col] != 'O') return;

        board[row][col] = 'S';

        dfs(board, row + 1, col);
        dfs(board, row - 1, col);
        dfs(board, row, col + 1);
        dfs(board, row, col - 1);
    }
}
```

写了个 BFS 的在一个中型矩阵上 TLE了，我有点方。。

```
public class Solution {
    public void solve(char[][] board) {
        if(board == null || board.length == 0) return;
        int rows = board.length;
        int cols = board[0].length;
        Queue<Integer> queue = new LinkedList<Integer>();

        for(int i = 0; i < cols; i++){
            if(board[0][i] == 'O'){
                queue.offer(i);
                bfs(board, 0, i, queue);
            }
            if(board[rows - 1][i] == 'O'){
                queue.offer((rows - 1) * cols + i);
                bfs(board, rows - 1, i, queue);
            }
        }

        for(int i = 1; i < rows - 1; i++){
            if(board[i][0] == 'O'){
                queue.offer(i * cols);
                bfs(board, i, 0, queue);
```

```
            }
            if(board[i][cols - 1] == 'O'){
                queue.offer(i * cols + cols - 1);
                bfs(board, i, cols - 1, queue);
            }
        }

        for(int i = 0; i < rows; i++){
            for(int j = 0; j < cols; j++){
                if(board[i][j] == 'S'){
                    board[i][j] = 'O';
                } else {
                    board[i][j] = 'X';
                }
            }
        }

    }

    private void bfs(char[][] board, int row, int col, Queue<Int
eger> queue){
        int rows = board.length;
        int cols = board[0].length;

        if(!isValid(row, rows, col, cols)) return;

        while(!queue.isEmpty()){
            Integer index = queue.poll();
            int x = index / cols;
            int y = index % cols;
            if(board[x][y] == 'O') board[x][y] = 'S';

            if(isValid(x + 1, rows, y, cols) && board[x + 1][y]
== 'O') queue.offer((x + 1) * cols + y);
            if(isValid(x - 1, rows, y, cols) && board[x - 1][y]
== 'O') queue.offer((x - 1) * cols + y);
            if(isValid(x, rows, y + 1, cols) && board[x][y + 1]
== 'O') queue.offer(x * cols + y + 1);
            if(isValid(x, rows, y - 1, cols) && board[x][y - 1]
== 'O') queue.offer(x * cols + y - 1);
```

```
        }

        return;
    }

    private boolean isValid(int row, int rows, int col, int cols)
{
        if(row < 0 || row >= rows) return false;
        if(col < 0 || col >= cols) return false;
        return true;
    }
}
```

同样的 DFS 逻辑，做了如下改动之后就不会 stackoverflow 了：

- DFS 时不进入最外围一圈的位置

简单来讲是在尽可能限制 DFS call stack 的层数，控制 DFS 的深度。从正确性上讲，上面的写法也是正确的，但是在极端情况下如果某一个从边沿出发的 DFS 连通了另一个边沿出发的 DFS，会导致一次的搜索路径非常长，于是 stackoverflow. 既然边沿的格子无论如何都要检查一遍，就把外圈封住，减少每个起点的 search tree depth.

```
public class Solution {
    public void solve(char[][] board) {
        if(board == null || board.length == 0) return;
        int rows = board.length;
        int cols = board[0].length;

        for(int i = 0; i < cols; i++){
            if(board[0][i] == 'O'){
                dfs(board, 0, i);
            }
            if(board[rows - 1][i] == 'O'){
                dfs(board, rows - 1, i);
            }
        }

        for(int i = 1; i < rows - 1; i++){
```

```
            if(board[i][0] == 'O'){
                dfs(board, i, 0);
            }
            if(board[i][cols - 1] == 'O'){
                dfs(board, i, cols - 1);
            }
        }

        for(int i = 0; i < rows; i++){
            for(int j = 0; j < cols; j++){
                if(board[i][j] == 'S'){
                    board[i][j] = 'O';
                } else {
                    board[i][j] = 'X';
                }
            }
        }

    }

    private void dfs(char[][] board, int row, int col){
        if(row < 0 || row >= board.length) return;
        if(col < 0 || col >= board[0].length) return;

        if(board[row][col] != 'O') return;

        board[row][col] = 'S';

        // DFS 时不进入最外圈
        if(row + 2 < board.length && board[row + 1][col] == 'O')
 dfs(board, row + 1, col);
        if(row - 2 >= 0 && board[row - 1][col] == 'O') dfs(board
, row - 1, col);
        if(col + 2 < board[0].length && board[row][col + 1] == '
O') dfs(board, row, col + 1);
        if(col - 2 >= 0 && board[row][col - 1] == 'O') dfs(board
, row, col - 1);
    }
}
```

这题当然也可以用 Union-Find 写，先把所有最外圈的 boundry 连上，然后把里面的相邻 'O' 做 union，最后扫矩阵的时候，如果对应的 root 不是 boundry root 就留下，不然都改成 'X'.

不过只是在这个问题上，不是很简洁。

Strobogrammatic 数生成

Strobogrammatic Number II

为什么一个这么简单的 DFS 能超过 89% ..

注意：**index == 0** 并且 **i == 0** 的时候要跳过，免得在起始位置填上 **0 .**

```java
public class Solution {
    public List<String> findStrobogrammatic(int n) {
        List<String> list = new ArrayList<>();
        char[] num1 = {'0','1','8','6','9'};
        char[] num2 = {'0','1','8','9','6'};
        char[] number = new char[n];

        dfs(list, number, num1, num2, 0);

        return list;
    }

    private void dfs(List<String> list, char[] number, char[] nu
m1, char[] num2, int index){
        int left = index;
        int right = number.length - index - 1;

        if(left > right){
            list.add(new String(number));
            return;
        }
        // We can fill in 0,1,8 only
        if(left == right){
            for(int i = 0; i < 3; i++){
                number[left] = num1[i];
                dfs(list, number, num1, num2, index + 1);
            }
        } else {
            for(int i = 0; i < num1.length; i++){
                if(index == 0 && i == 0) continue;
                number[left] = num1[i];
                number[right] = num2[i];
                dfs(list, number, num1, num2, index + 1);
            }
        }
    }
}
```

Strobogrammatic Number III

Google 面经里的 follow-up 是，给定一个上限 n ，输出所有上限范围内的数。

办法土了点，遍历所有 lowLen ~ highLen 区间的长度，生成所有可能的结果，考虑到区间可能是大数，我们就改一下，自己写一个 String compare 函数好了。

后来发现有点多余，可以直接用内置的 str1.compareTo(str2).

超过 81.92% ~

```
public class Solution {
    int count = 0;
    public int strobogrammaticInRange(String low, String high) {
        int lowLen = low.length();
        int highLen = high.length();

        char[] num1 = {'0','1','8','6','9'};
        char[] num2 = {'0','1','8','9','6'};

        for(int i = lowLen; i <= highLen; i++){
            char[] number = new char[i];
            dfs(number, num1, num2, 0, low, high);
        }

        return count;
    }

    private void dfs(char[] number, char[] num1, char[] num2, int
 index, String low, String high){
        int left = index;
        int right = number.length - index - 1;

        if(left > right){
            String num = new String(number);
            if(compare(low, num) <= 0 && compare(num, high) <= 0
) count++;
            return;
        } else if(left == right){
            for(int i = 0; i < 3; i++){
```

```
                    number[left] = num1[i];
                    dfs(number, num1, num2, index + 1, low, high);
                }
            } else {
                for(int i = 0; i < 5; i++){
                    if(index == 0 && i == 0) continue;
                    number[left] = num1[i];
                    number[right] = num2[i];
                    dfs(number, num1, num2, index + 1, low, high);
                }
            }
        }

        // -1 : str1 is bigger
        // 1 : str 2 is bigger
        // 0 : equal
        private int compare(String str1, String str2){
            if(str1.length() > str2.length()) return 1;
            else if(str1.length() < str2.length()) return -1;
            else {
                for(int i = 0; i < str1.length(); i++){
                    int digit1 = str1.charAt(i) - '0';
                    int digit2 = str2.charAt(i) - '0';

                    if(digit1 != digit2) return (digit1 > digit2) ? 1
 : -1;
                }
            }
            // Equal
            return 0;
        }

    }
```

String 构造式 DFS + Backtracking

然而 “字符串构造式 **DFS + backtracking**”，还有最简单直接又好分析时间复杂度的做法，就是从 **String** 中每个字符的角度出发，去考虑 “拿 / 不拿”。

StringBuilder 的构造式 **DFS + Backtracking**，对 “加 / 不加” 的先后顺序是敏感的，下面两题中，同样的代码，都是

- 先走“不加”的分支
- 再走“加”的分支

原因在于，这个模板是 “带着当前考虑元素” 的 **DFS**，在当前函数尾部回溯。当两个 **dfs** 同处一层，而同层 **dfs** 只在最尾部才回溯的时候，如果先跑添加的分支，会在后面执行不添加分支时候出错。

否则就一次 **dfs** 后面跟着一次 **sb.setLength()**，也可以。

Generalized Abbreviation

这题完全可以从 word 出发，以 length 和 pos 的角度遍历去 DFS 解决，比较符合大多数搜索问题的状态空间结构。

然而 “字符串构造式 **DFS + backtracking**”，还有最简单直接又好分析时间复杂度的做法，就是从 **String** 中每个字符的角度出发，去考虑 “拿 / 不拿”。

超过 93.61%

```
public class Solution {
    public List<String> generateAbbreviations(String word) {
         List<String> list = new ArrayList<>();
         dfs(list, new StringBuilder(), word.toCharArray(), 0, 0
);
         return list;
    }

    private void dfs(List<String> list, StringBuilder sb, char[]
 word, int index, int curNum){
        int len = sb.length();
        if(index == word.length){
            if(curNum != 0) sb.append(curNum);
            list.add(sb.toString());
        } else {
            // Don't add, merge to current number
            dfs(list, sb, word, index + 1, curNum + 1);

            // Add current char
            if(curNum != 0) sb.append(curNum);
            dfs(list, sb.append(word[index]), word, index + 1, 0
);
        }
        sb.setLength(len);
    }
}
```

Remove Invalid Parentheses

这题的结构就和 **Different ways to add parenthese** 不一样，括号之间的相对位置是非常重要的，而且有连续性，不能像那题一样直接在操作符上建一个 **expression tree** 出来，**divide & conquer.**

首先继承上一题 Stack 的经验，我们很容易可以知道需要的最少 edit 次数，以及 invalid 字符位置：跑一遍算法，看 Stack 里剩几个元素，分别在哪就行了。

> 只有一个非法括号的情况：
>
> - **(( )) ( ) '(' (( )) ( )**
> - 可以发现对于 **'('** ，只能向右删下一个 **'('**，相邻的 **'('** 是重复解
> - **( ) (( )) ')' ( ) (( ))**
> - 同理，**')'** 只能往左边找**')'**，相邻 **')'** 为重复解。
>
> 有两个非法括号的情况：
>
> - **( ) (( )) ')' ( ) '(' (( )) ( )**
> - **( ) (( )) ( ) (( )) ( )**
> - **( ) (( )) ')' ( ) '(' (( )) ( )**
> - 可以发现 **')'** 有两个可能的位置，**'('** 有两个可能的位置，于是是 **4** 种组合。

于是这题就变成了一个搜索问题~

> - 问题**1**：既然在反复在字符串上进行修改，如何还能保证 **List<>** 里面的非法字符位置还是正确的？
> - 问题**2**：动态修改的 **string** ，动态变化的 **index**，要处理很多细节保证他们的 **match.**

觉得顺着这个思路越想细节越多。。于是参考了下论坛的 **DFS** 思路，在所有 **DFS** 解法中，我最喜欢这个，非常简洁易懂，而且不依赖什么 **trick**，便于和面试官交流。

- 先扫一遍原 **string** ，记录下需要移除的左括号数量和右括号数量，保证最后的解最优；

- 于是开始 **DFS**，对于每一个新的字符，我们都有两个选择：'留' 或者 '不留'，就像二叉树的分叉一样，留下了 **dfs + backtracking** 的空间。
- 于是当我们针对各种情况进行 **dfs** 的时候，我们一定可以考虑到所有可能的有效 **path**，接下来需要定义什么是 "有效 **path**"
- **base case** 是左右括号和开括号数量都为零，并且 **index** 读取完了所有字符，则把结果添加进去；
- 如果在任何时刻，左右括号和开括号的数量为负，我们都可以直接剪枝返回。

这种解法的主要优点是好理解，空间上因为只用了一个 **StringBuilder** 非常经济，相比 **BFS** 速度和空间上都优异很多。如果说进一步优化的空间在哪，那就是对于 "重复状态" 的缓存与记忆化搜索还有提高的空间。

于是很 **naive** 的尝试了下加入 **Set<>** 来记录已经访问过的 **StringBuilder** 状态试图剪枝，然而很容易就出了 "没有任何返回结果" 的 **bug** ，其原因是，这个 **dfs + backtracking** 的代码本来就是在一个神似 **binary tree** 的结构上做一个 **dfs** 穷举而已，并不是 **top-down recursion** 那种子问题覆盖的结构，所以不适合用记忆化搜索解决。非要用的话至少 **dfs** 的返回类型就不能是 **void** ，至少也得是个 **List<>** 或者 **Set<>** 之类的 "子问题结果" 嘛。

```
public class Solution {
    public List<String> removeInvalidParentheses(String s) {
        Set<String> set = new HashSet<>();

        int leftCount = 0;
        int rightCount = 0;
        int openCount = 0;

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

```
            if(s.charAt(i) == '(') leftCount++;
            if(s.charAt(i) == ')'){
                if(leftCount > 0) leftCount--;
                else rightCount ++;
            }
        }

        dfs(set, s, 0, leftCount, rightCount, openCount, new Str
ingBuilder());

        return new ArrayList<String>(set);
    }

    private void dfs(Set<String> set, String str, int index, int
 leftCount,
                     int rightCount, int openCount, StringBuilde
r sb){
        if(index == str.length() && leftCount == 0 && rightCount
 == 0 && openCount == 0){
            set.add(sb.toString());
            return;
        }

        if(index == str.length() || leftCount < 0 || rightCount
< 0 || openCount < 0) return;

        char chr = str.charAt(index);
        int len = sb.length();

        if(chr == '('){
            // Remove current '('
            dfs(set, str, index + 1, leftCount - 1, rightCount,
openCount, sb);
            // Keep current '('
            dfs(set, str, index + 1, leftCount, rightCount, open
Count + 1, sb.append(chr));
        } else if(chr == ')'){
            // Remove current ')'
            dfs(set, str, index + 1, leftCount, rightCount - 1,
openCount, sb);
```

```
            // Keep current ')'
            dfs(set, str, index + 1, leftCount, rightCount, open
Count - 1, sb.append(chr));
        } else {
            // Just keep the character
            dfs(set, str, index + 1, leftCount, rightCount, open
Count, sb.append(chr));
        }

        // Back-tracking
        sb.setLength(len);
    }
}
```

Expression Add Operators

num = "105", target = 5;

返回 ["10-5", "1x0+5", "1+0x5", "1x05", "1-0x5"] 是错的

- 不应该直接把 "05" 当成数字；
- 乘法符号有优先级问题。

在 Different ways to add parentheses 里，这么干没有任何问题，因为默认是加括号的，左右子树可以完全分离，单独求值。

而这题里，用 divide & conquer 无可避免地遇到符号优先级问题。于是看了一圈 LC 论坛，大家的做法其实都是 DFS + backtracking ..

这题非要用 divide & conquer 也不是不行，但是极其麻烦，如图所示：

对于每一个节点，我们需要考虑其左子树的 String, value，还有抹去最后一步运算的 String 与 value .. 对于右子树，我们还需要其抹去第一步运算的 String 与 value 才能正确做 join.

所以说，珍爱生命，有乘法就远离 divide & conquer 吧。

一个思路不错的帖子，这种做法就不用考虑 **join**了，只要做乘法的时候看它的前一个元素就行；

流程：

- **dfs + backtracking** 思路
- **dfs** 函数中存 **"**左面表达式的当前值**"** 和 **"**上一步操作的结果**"** **(**用于处理优先级**)**

- 参数都用 **Long**，防止溢出，毕竟 **String** 可能是个大数
- 如果 **start** 位置是 **0** 而且当前循环的 **index** 还是 **0**，直接 **return**，因为一轮循环只能取一次 **0**；
- **start == 0** 的时候，直接考虑所有可能的起始数字扔进去，同时更新 **cumuVal** 和 **prevVal;**
- 除此之外，依次遍历所有可能的 **curVal parse**，按三种不同的可能操作做 **dfs + backtracking;**
    - 加法：直接算，**prevVal** 参数传 **curVal**，代表上一步是加法；
    - 减法：直接算，**prevVal** 参数传 **-curVal**，代表上一步是减法；
    - 乘法：要先减去 **prevVal** 抹去上一步的计算，然后加上 **prevVal** ***curVal*** 代表当前值；同时传的 ***preVal*** 参数也等于 ***prevVal*** **curVal.**
    - 乘法操作这种处理方式，在遇到连续乘法的时候可以看到是叠加的；但是前一步操作如果不是乘法，则可以优先计算乘法操作。
- **10-5-2x6** ，遇到乘法之前是 **cumuVal = 3, preVal = -2;** 新一轮 **dfs** 时 **curVal = 6,** 先退回前一步操作 **- prevVal**，得到 **5**，其实就是之前 **(10 - 5)** 的结果，然后加上当前结果 **(-2 x 5)**，**prevVal** 设成 **-10** 传递过去即可。

```
public class Solution {
    public List<String> addOperators(String num, int target) {
        List<String> rst = new ArrayList<>();
        dfs(rst, new StringBuilder(), num, target, 0, 0, 0);
        return rst;
    }
```

```
    private void dfs(List<String> rst, StringBuilder sb, String
num, int target, int startPos, long cumuVal, long prevVal){
        if(startPos == num.length() && cumuVal == target){
            rst.add(sb.toString());
            return;
        }

        for(int i = startPos; i < num.length(); i++){
            // We can pick one zero to start with; but not multi
ple
            if(num.charAt(startPos) == '0' && i != startPos) ret
urn;
            int len = sb.length();
            String curStr = num.substring(startPos, i + 1);
            long curVal = Long.parseLong(curStr);

            if(startPos == 0){
                dfs(rst, sb.append(curVal), num, target, i + 1,
curVal, curVal);
                sb.setLength(len);
            } else {
                dfs(rst, sb.append("+" + curStr), num, target, i
 + 1, cumuVal + curVal, curVal);
                sb.setLength(len);
                dfs(rst, sb.append("-" + curStr), num, target, i
 + 1, cumuVal - curVal, -curVal);
                sb.setLength(len);
                dfs(rst, sb.append("*" + curStr), num, target, i
 + 1, cumuVal - prevVal + prevVal * curVal, prevVal * curVal);
                sb.setLength(len);
            }
        }
    }
}
```

Word Pattern I & II

Word Pattern

热手题，对于 bijection mapping 就是两个 hashmap 互相查，和 Isomorphic Strings 一样。

```
public class Solution {
    public boolean wordPattern(String pattern, String str) {
        HashMap<String, String> pat2word = new HashMap<>();
        HashMap<String, String> word2pat = new HashMap<>();

        String[] strs = str.split(" ");
        if(strs.length != pattern.length()) return false;

        for(int i = 0; i < strs.length; i++){
            String pat = "" + pattern.charAt(i);
            String word = strs[i];

            if(!pat2word.containsKey(pat) && !word2pat.containsK
ey(word)){
                pat2word.put(pat, word);
                word2pat.put(word, pat);
            } else {
                if(!pat2word.containsKey(pat)) return false;
                if(!word2pat.containsKey(word)) return false;

                if(!pat2word.get(pat).equals(word)) return false
;
                if(!word2pat.get(word).equals(pat)) return false
;
            }
        }

        return true;
    }
}
```

Word Pattern II

犯错**1**：没考虑到当前的 **pat / word** 都在 **map** 里而且合法的情况，这时候需要继续向下探。

犯错2: pattern.length() == 0 代表着搜索的结束，但是不是 return true 的充分条件。所以要在 pattern 为 0 的时候正确结束免得越界，同时也要检查 str.length() 是否也等于 0.

```
public class Solution {
    public boolean wordPatternMatch(String pattern, String str)
{
        return dfs(pattern, str, new HashMap<String, String>(),
new HashMap<String, String>());
    }

    private boolean dfs(String pattern, String str,
                        HashMap<String, String> pat2word,
                        HashMap<String, String> word2pat){

        if(pattern.length() == 0) return (str.length() == 0);

        String pat = "" + pattern.charAt(0);

            for(int j = 0; j < str.length(); j++){
                String word = str.substring(0, j + 1);
                if(!pat2word.containsKey(pat) && !word2pat.conta
insKey(word)){
                    pat2word.put(pat, word);
                    word2pat.put(word, pat);

                    if(dfs(pattern.substring(1), str.substring(j
 + 1), pat2word, word2pat)) return true;

                    pat2word.remove(pat);
                    word2pat.remove(word);
                } else if(pat2word.containsKey(pat) && word2pat.
containsKey(word)){
                    if(pat2word.get(pat).equals(word) && word2pa
t.get(word).equals(pat)){
                        if(dfs(pattern.substring(1), str.substri
ng(j + 1), pat2word, word2pat)) return true;
                    }
                }
                /*
```

```
                else {
                    if(!pat2word.containsKey(pat))  continue;
                    if(!word2pat.containsKey(word)) continue;

                    if(!pat2word.get(pat).equals(word)) continue
;
                    if(!word2pat.get(word).equals(pat)) continue
;
                }
                */
            }

        return false;
    }
}
```

这种写法在速度上可以进一步改进，比如当看到 char 已经在 map 时，我们就直接把其对应的 word 取出来，不出问题的话就可以继续，否则可以立刻返回 false 进行剪枝和early termination.

然而下面的代码用时上和原来的没什么不同。。可能是 test case 数量太小吧。面试被问到时作为一个 follow up 优化写上去还是不错的。

```
public class Solution {
    public boolean wordPatternMatch(String pattern, String str)
{
        return dfs(pattern, str, new HashMap<String, String>(),
new HashMap<String, String>());
    }

    private boolean dfs(String pattern, String str,
                        HashMap<String, String> pat2word,
                        HashMap<String, String> word2pat){

        if(pattern.length() == 0) return (str.length() == 0);

        String pat = "" + pattern.charAt(0);

        if(pat2word.containsKey(pat)){
```

```
            String word = pat2word.get(pat);
            if(word.length() > str.length()) return false;

            else if(word.equals(str.substring(0, word.length()))
){
                if(dfs(pattern.substring(1), str.substring(word.
length()), pat2word, word2pat)) return true;
            } else {
                return false;
            }
        }

            for(int j = 0; j < str.length(); j++){
                String word = str.substring(0, j + 1);
                if(!pat2word.containsKey(pat) && !word2pat.conta
insKey(word)){
                    pat2word.put(pat, word);
                    word2pat.put(word, pat);

                    if(dfs(pattern.substring(1), str.substring(j
 + 1), pat2word, word2pat)) return true;

                    pat2word.remove(pat);
                    word2pat.remove(word);
                } else if(pat2word.containsKey(pat) && word2pat.
containsKey(word)){
                    if(pat2word.get(pat).equals(word) && word2pa
t.get(word).equals(pat)){
                        if(dfs(pattern.substring(1), str.substri
ng(j + 1), pat2word, word2pat)) return true;
                    }
                }
                /*
                else {
                    if(!pat2word.containsKey(pat))  continue;
                    if(!word2pat.containsKey(word)) continue;

                    if(!pat2word.get(pat).equals(word)) continue
;
                    if(!word2pat.get(word).equals(pat)) continue
```

```
;
                }
                */
            }

        return false;
    }
}
```

(G) Binary Watch

http://www.1point3acres.com/bbs/thread-141438-1-1.html

题目是binary watch：上边是小时，下边是分钟，最左边最significant，最右边为1。给你数字n，return所有可能的时间，以string为表达形式.

E.G. 给你1,那return:{0:00,1:00,2:00,4:00,8:00,0:01.....}

这题挺简单的，就是一个简单位运算和 mask + dfs 枚举。

- **0x12345678** 的 **hex-decimal** 表示方式中，一个数代表 **4 bit**，**0-9 a-f** 总共代表 **16** 个数；
- 位数不到 **8** 个的说明前面默认都是 **0**；
- **0x0000003f = 0000 0000 ...0011 1111**，最后一个位段是 **1111 = f**，前面那个 **0011 = 3;**

- 为了避免重复，进一步剪枝，这题要像 **combination** 一样传一个 **index** ，单调往后扫。

```
    public static List<String> binaryWatch(int n) {
        List<String> list = new ArrayList<>();
        dfs(list, n, 0, 0);
        return list;
    }

    public static void dfs(List<String> rst, int n, int num, int
 index){
        if(n == 0){
            int minutes = num & (0x0000003f);
            int hours = num >> 6;

            if(hours < 24 && minutes < 60) rst.add("" + hours +
":" + minutes);

            return;
        }

        for(int i = index; i < 10; i++){
            if((num & (1 << i)) == 0){
                dfs(rst, n - 1, num + (1 << i), index + 1);
            }
        }
    }

    public static void main(String[] args){
        List<String> list = binaryWatch(1);

        for(String str : list){
            System.out.println(str);
        }
        System.out.println(list.size() + " Answers.");
    }
```

(FB) Phone Letter Combination

Phone Letter Combination

把这题单独拿出来，是因为这是个非常典型的递归 / 迭代都很直观的搜索类问题，一个 **DFS**，一个 **BFS.**

BFS 就靠 **Queue**，以 **queue** 首长度 **== i** 来判断层数，反复做 **join.** 另外维护一个 **String[]** 用作字典查询。

```
    public List<String> letterCombinations(String digits) {
        LinkedList<String> queue = new LinkedList<String>();
        if(digits == null || digits.length() == 0) return queue;

        queue.add("");
        String[] letters = new String[]{"", "", "abc", "def", "g
hi", "jkl", "mno", "pqrs", "tuv", "wxyz"};

        for(int i = 0; i < digits.length(); i++){
            int cur = digits.charAt(i) - '0';
            while(queue.peek().length() == i){
                String str = queue.remove();
                char[] candidates = letters[cur].toCharArray();
                for(char chr : candidates){
                    queue.add(str + chr);
                }
            }
        }

        return queue;
    }
```

DFS 写法就是做了无数次的 **backtracking** 写法。本质上，都是殊途同归的状态空间搜索罢了。

```
    public List<String> letterCombinations(String digits) {
        List<String> list = new ArrayList<String>();
        if(digits == null || digits.length() == 0) return list;
        String[] letters = new String[]{"", "", "abc", "def", "g
hi", "jkl", "mno", "pqrs", "tuv", "wxyz"};

        dfs(list, new StringBuilder(), digits, letters, 0);

        return list;
    }

    private void dfs(List<String> list, StringBuilder sb, String
 digits, String[] letters, int pos){
        if(sb.length() == digits.length()){
            list.add(sb.toString());
            return;
        }

        for(int i = pos; i < digits.length(); i++){
            String str = letters[digits.charAt(i) - '0'];
            for(int j = 0; j < str.length(); j++){
                int length = sb.length();
                sb.append(str.charAt(j));
                dfs(list, sb, digits, letters, i + 1);
                sb.setLength(length);
            }
        }
    }
```

常见搜索问题的迭代解法

从搜索树结构上讲，这类问题所有解的构造方式呈现 **BFS** 的分层结构，可以手动建立分层处理的机制处理。与之相对的，最常见的递归解法是从 **DFS** 的角度去探索状态空间。

搜索问题的迭代解法主要分这么两种，取决于最终解的构造方式，也即 **F(n)** 和 **F(n - 1)** 之间的关系。

- **append** 类
  - 比如 **subsets** 中，每次迭代出现的新解，都是由新元素 **append** 到已知解的后面构造出来的；**(**不过所有过程中的已知解都留下来了**)**
  - 在 **Phone letter combination** 里也是一个道理，只不过这次我们不存过渡结果，只保留最后一层。
- **insert** 类
  - 比如 **permutation** ，这里解的构造方式靠 **"insert"**，所以 **BFS** 分层的搜索结构中，会反复出现靠插入构造出的新结果。

Subsets

这题三种解法的分析：

http://bangbingsyb.blogspot.com/2014/11/leetcode-subsets-i-ii.html

迭代：

方法2：添加数字构建subset

起始subset集为：[]
添加S0后为：[], [S0]
添加S1后为：[], [S0], [S1], [S0, S1]
添加S2后为：[], [S0], [S1], [S0, S1], [S2], [S0, S2], [S1, S2], [S0, S1, S2]
红色subset为每次新增的。显然规律为添加Si后，新增的subset为克隆现有的所有subset，并在它们后面都加上Si。

```
public class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> rst = new ArrayList<>();
        rst.add(new ArrayList<Integer>());

        for(int i = 0; i < nums.length; i++){
            int n = rst.size();
            for(int j = 0; j < n; j++){
                List<Integer> list = new ArrayList<Integer>(rst.
get(j));
                list.add(nums[i]);
                rst.add(list);
            }
        }

        return rst;
    }
}
```

位运算：

由于S[0: n-1]组成的每一个subset，可以看成是对是否包含S[i]的取舍。S[i]只有两种状态，包含在特定subset内，或不包含。所以subset的数量总共有2^n个。所以可以用0~2^n-1的二进制来表示一个subset。二进制中每个0/1表示该位置的S[i]是否包括在当前subset中。

```
public class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        int totalAnswer = 1 << nums.length;
        List<List<Integer>> rst = new ArrayList<>(totalAnswer);

        for(int encoding = 0; encoding < totalAnswer; encoding++
){
            List<Integer> curList = new ArrayList<>();
            for(int j = 0; j < nums.length; j++){
                if((encoding & (1 << j)) != 0) curList.add(nums[
j]);
            }
            rst.add(curList);
        }

        return rst;
    }
}
```

Letter Combinations of a Phone Number

```
    public List<String> letterCombinations(String digits) {
        List<String> rst = new ArrayList<String>();
        if(digits == null || digits.length() == 0) return rst;
        String[] letters = new String[]{"", "", "abc", "def", "g
hi", "jkl",
                                        "mno", "pqrs", "tuv", "w
xyz"};
        rst.add("");
        for(int i = 0; i < digits.length(); i++){
            int digit = digits.charAt(i) - '0';
            String next = letters[digit];
            int size = rst.size();
            List<String> nextLvl = new ArrayList<>();
            for(int j = 0; j < next.length(); j++){
                char chr = next.charAt(j);
                for(int k = 0; k < size; k++){
                    String newStr = rst.get(k) + chr;
                    nextLvl.add(newStr);
                }
            }
            rst = nextLvl;
        }
        return rst;
    }
```

Permutations

讨论帖子：

https://discuss.leetcode.com/topic/6377/my-ac-simple-iterative-java-python-solution

http://bangbingsyb.blogspot.com/2014/11/leetcode-permutations-i-ii.html

想迭代做这题，要先从理解为什么对于 cardinality = n 的集合，其总共的 permutation = n! 开始，permutation 的构造过程是：

- 第 **n** 层 **permutation** 的每个解中，有 **n** 个元素；

- 从**n = 1** 开始，放入第 **1** 个元素；
- 下一层要新加入的元素是 **n + 1**，在第 **n + 1** 层；
- 我们要从 **n** 层的所有结果中，插入所有可以插入的位置，对于每一个 **size = n** 的解，可插入位置为 **n + 1** 个；
- 于是每一层新得到的解的总数 **F(n + 1) = F(n) * (n + 1),** 可知 **F(n) = n!**

- 注意点**1**：一开始要加个 **{}** 进去，要么无法起始；
- 注意点**2**：**List.add(index, ele)** 记熟，不是 **insert**
- 注意点**3:** 插入范围要比 **prevList.size()** 大一个，所以具体有效的插入 **index** 是【**0, prevList.size()**】 的闭区间，

```
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> rst = new ArrayList<List<Integer>>()
;
        int n = nums.length;
        rst.add(new ArrayList<>());

        for(int i = 0; i < n; i++){
            int size = rst.size();
            List<List<Integer>> nextLvl = new ArrayList<List<Int
eger>>();
            for(int j = 0; j < size; j++){
                List<Integer> prevList = rst.get(j);
                for(int k = 0; k <= prevList.size(); k++){
                    List<Integer> curList = new ArrayList<Intege
r>(prevList);
                    curList.add(k, nums[i]);
                    nextLvl.add(curList);
                }
            }
            rst = nextLvl;
        }

        return rst;
    }
```

String，字符串类

5/22 String，多步翻转法

Reverse Words in a String II

两步翻转法，参考 这里 的 topological sort. 其实这个 follow up 比问题一简单，因为已经告诉你了没有 trailing zeros 而且中间空格只有一个。

while 循环里套 while 的时候，记得把外层的判断条件也放到内层，否则容易越界。

```
public class Solution {
    public void reverseWords(char[] s) {
        int wordStart = 0;
        int index = 0;

        while(index < s.length){
            while(index < s.length && s[index] != ' ') index++;

            reverse(s, wordStart, index - 1);

            index ++;
            wordStart = index;
        }

        reverse(s, 0, s.length - 1);
    }

    private void reverse(char[] s, int start, int end){
        while(start < end){
            char temp = s[start];
            s[start] = s[end];
            s[end] = temp;

            start ++;
            end --;
        }
    }
}
```

Reverse Words in a String I

稍微麻烦点，要处理各种 trailing space 和中间

```
public class Solution {
    public String reverseWords(String s) {
        if(s.length() == 0) return "";
        int wordStart = 0;
        int index = 0;
```

```
        char[] array = s.toCharArray();

        while(index < s.length()){
            while(index < s.length() && array[index] == ' ') ind
ex ++;
            wordStart = index;
            while(index < s.length() && array[index] != ' ') ind
ex ++;
            reverse(array, wordStart, index - 1);
        }

        StringBuilder sb = new StringBuilder();
        index = s.length() - 1;

        while(index >= 0){
            while(index >= 0 && array[index] == ' ') index --;
            while(index >= 0 && array[index] != ' '){
                sb.append(array[index--]);
            }
            sb.append(' ');
        }

        return sb.toString().trim();
    }

    private void reverse(char[] s, int start, int end){
        while(start < end){
            char temp = s[start];
            s[start] = s[end];
            s[end] = temp;

            start ++;
            end --;
        }
    }
}
```

Rotate Array

多步翻转法的另一个应用。

```
public class Solution {
    public void rotate(int[] nums, int k) {
        k = k % nums.length;

        reverse(nums, 0, nums.length - k - 1);
        reverse(nums, nums.length - k, nums.length - 1);
        reverse(nums, 0 , nums.length - 1);
    }

    private void reverse(int[] nums, int start, int end){
        while(start < end){
            int temp = nums[start];
            nums[start] = nums[end];
            nums[end] = temp;

            start ++;
            end --;
        }
    }
}
```

A1 a2 ..an b1 b2 ...bn -> a1b1a2b2...anbn

AaBb - ABab

a1 a2 (a3 a4 b1 b2) b3 b4 - a1 a2 b2 b1 || a4 a3 b3 b4

因为 a1 a2 长度等于 b1 b2

5/22 String，substring 问题

Longest Substring Without Repeating Characters

- **Substring** 类问题和 **CLRS** 的 **Rod-Cutting** 有非常紧密的联系。

Figure 15.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 15.1. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        q = max(q, p[i] + r[j - i])
    r[j] = q
return r[n]
```

- 给定一个长度为 **n** 的 **String**，它所有的 **substring** 数量是 **n(n + 1) / 2** ，是一个 **quadratic** 的数量级。所以有些问题如果暴力遍历的话复杂度是没法让人满意的，某些问题如果用二维 **DP** 也至少需要 **O(n^2)** 的时间与空间开销。

- 给定一个长度为 **n** 的 **String**，切成若干个 **pieces** 总共有 **$$2^{n-1}$$** 种切法，即对于所有 **$$n-1$$** 个分界点上，选择**“切/不切”.**

- 此类问题最常用的优化，就是利用子串性质， **abuse** 子串的结构。

- 同时维护一个类似 **sliding window** 的结构去向尾部移动，如果是 **KMP pattern matching**，不回滚的 **window / pattern** 就可以达到 **linear time.**

在这个问题里，substring 里面一定没有重复字符，因此可以开一个 boolean array 作为 hash 记录 window 里面已经存在的字符。

同时如果在看到一个新字符之后出现重复的话，以这个字符为结尾的最长 substring 一定在 sliding window 里面同一个字符之后。

- **“必须以当前字符结尾”** 是字符串问题中很常见的 **optimal substructure,** 因此这个问题也类似于 **DP** 问题。

- 这题我后面在双指针的地方重写了，比这个简单。

```
public class Solution {
    public int lengthOfLongestSubstring(String s) {
        if(s.length() <= 1) return s.length();
        int start = 0;
        int end = 0;
        int max = 1;
        boolean[] hash = new boolean[256];
        while(end < s.length()){
            int key = (int) s.charAt(end);
            if(!hash[key]){
                hash[key] = true;
                end ++;
                max = Math.max(max, end - start);
            } else {
                while(start < end && s.charAt(start) != s.charAt
(end)){
                    hash[(int)s.charAt(start)] = false;
                    start ++;
                }
                start ++;
                end ++;
            }
        }

        return max;
    }
}
```

- 双指针重写版:

```java
public class Solution {
    public int lengthOfLongestSubstring(String s) {
        if(s.length() <= 1) return s.length();
        int max = 1;
        boolean[] hash = new boolean[256];
        int j = 0;
        for(int i = 0; i < s.length(); i++){
            while(j < s.length()){
                if(!hash[s.charAt(j)]){
                    hash[s.charAt(j++)] = true;
                } else {
                    break;
                }
            }
            max = Math.max(max, j - i);
            hash[s.charAt(i)] = false;
        }

        return max;
    }
}
```

5/22 String, Palindrome 问题

> **Palindrome 的定义是，给定 S, S=S'**
>
> **KMP 算法的核心是 failure function ，在 currect position 不匹配的情况下，在之前的 substring中寻找最长的 suffix 后缀，使得它和 substring 中的 prefix 前缀相同。**

Longest Palindromic Substring

考虑到总共可能的 palindrome 数量，我就先写了一个比较挫的写法，middle-out 由每个字符作为起点，往两边扫。

Palindrome 长度可能是奇数，也可能是偶数。所以指定 seed 往两边扩展的时候要注意先把 start / end 指针推到“不是 seed” 的位置上。

去 LeetCode 论坛看了一圈，发现他们用的写法也基本就是这么挫的。。

这个问题是有 $$O(n)$$ 解法的，见 Manacher's Algorithm ， 不过对于面试的时间来讲要求有点太高了，就和要求你手写 KMP 一样。

```
public class Solution {
    public String longestPalindrome(String s) {
        if(s.length() <= 1) return s;
        if(s.length() == 2){
            if(s.charAt(0) == s.charAt(1)) return s;
            else return s.substring(1);
        }

        int maxStart = 0;
        int maxEnd = 0;
        int maxLength = 1;

        for(int i = 0; i < s.length(); i++){
            int start = i - 1;
            while(start >= 0 && s.charAt(start) == s.charAt(i))
start --;
            int end = i + 1;
            while(end < s.length() && s.charAt(end) == s.charAt(
i)) end ++;

            while(start >= 0 && end < s.length()){
                if(s.charAt(start) == s.charAt(end)){
                    start --;
                    end ++;
                } else {
                    break;
                }
            }
            if(end - start - 1 > maxLength){
                maxLength = end - start - 1;
                maxStart = start + 1;
                maxEnd = end - 1;
            }
        }

        return s.substring(maxStart, maxEnd + 1);
    }
}
```

Valid Palindrome

Trivial problem. 稍微烦一点的地方在于我们要忽略字符，并且不区分大小写（原始 String 里有字符也有大小写）

ASC II 表里，一个字母的小写值与大写值的差正好是 32 （小写字母值更大）

Character.isLetterOrDigit(char chr)

str.toLowerCase();

```
public class Solution {
    public boolean isPalindrome(String s) {
        if(s.length() <= 1) return true;
        s = s.toLowerCase();

        int start = 0;
        int end = s.length() - 1;

        while(start < end){
            while(start < end && !Character.isLetterOrDigit(s.ch
arAt(start)))
                start ++;
            while(start < end && !Character.isLetterOrDigit(s.ch
arAt(end)))
                end --;

            if(s.charAt(start) == s.charAt(end)){
                start ++;
                end --;
            } else {
                return false;
            }
        }

        return true;
    }
}
```

Palindrome Permutation

Trivial problem. 扫一遍所有字符，如果出现过奇数次数的字符超过一个，就不能构造出 Palindrome 了。考察的就是一个简单的对 Palindrome 结构的理解。

```
public class Solution {
    public boolean canPermutePalindrome(String s) {
        if(s.length() <= 1) return true;

        int[] hash = new int[256];
        for(int i = 0; i < s.length(); i++){
            int index = (int) s.charAt(i);
            hash[index] ++;
        }

        boolean oneOdd = false;
        for(int i = 0; i < 256; i++){
            if(hash[i] % 2 == 0){
                continue;
            } else {
                if(oneOdd) return false;
                oneOdd = true;
            }
        }

        return true;
    }
}
```

Palindrome Permutation II

下面的代码是一个比较粗暴的第一版本，因为要返回所有正确解，属于一个搜索问题。根据 palindrome 结构做 DFS + backtracking.

看了下论坛，发现别人的提交也都是这个思路。。

```
public class Solution {
    public List<String> generatePalindromes(String s) {
        int[] hash = new int[256];
        for(int i = 0; i < s.length(); i++){
            int index = (int) s.charAt(i);
            hash[index] ++;
        }
```

```
        char center = '.';
        boolean oneOdd = false;
        for(int i = 0; i < 256; i++){
            if(hash[i] % 2 == 0){
                continue;
            } else {
                if(oneOdd) return new ArrayList<String>();

                oneOdd = true;
                center = (char) i;
                hash[i] --;
            }
        }

        char[] array = new char[s.length()];
        List<String> list = new ArrayList<String>();

        if(oneOdd) {
            array[s.length() / 2] = center;
            dfs(list, array, hash, s.length() / 2 - 1, s.length(
) / 2 + 1);
        } else {
            dfs(list, array, hash, s.length() / 2 - 1, s.length(
) / 2);
        }

        return list;
    }

    private void dfs(List<String> list, char[] array, int[] hash
, int left, int right){
        if(left < 0 || right >= array.length){
            list.add(new String(array));
            return;
        }

        for(int i = 0; i < 256; i++){
            if(hash[i] <= 0) continue;

            array[left] = (char) i;
```

```
                array[right] = (char) i;
                hash[i] -= 2;

                dfs(list, array, hash, left - 1, right + 1);

                array[left] = '.';
                array[right] = '.';
                hash[i] += 2;
            }
        }
    }
```

[重点] Shortest Palindrome

这题在 LeetCode 上的标注难度为 "Hard".

我一开始的写法比较的简单粗暴，直接用写了一个 isPalindrome 函数去判断一个 substring 是不是 palindrome，然后从给定的 string 左面开始一直往右扫，去找到从最左边字符开始，最长的 palindrome，然后把剩下的右边部分复制一份，翻转，接到原来的 string 上.

但是超时了，时间复杂度太高。挂在了一个“aaaaaaaa...” 非常长但是结构非常简单的 test case 上。

这个思路是没有问题的，但是逐个 substring 去调用 O(n) 时间的函数还是时间复杂度太高了，没能有效利用 palindrome 的结构性质。

```
public class Solution {
    public String shortestPalindrome(String s) {
        if(s.length() <= 1) return s;
        // Find the lonest palindrome starting from left most of
 string
        int maxLength = 1;
        for(int i = 1; i < s.length(); i++){
            if(isPalindrome(s, 0, i)){
                maxLength = Math.max(maxLength, i + 1);
            }
        }
        StringBuilder sb = new StringBuilder(s.substring(maxLeng
th));
        String toAdd = sb.reverse().toString();

        return toAdd + s;

    }

    private boolean isPalindrome(String s, int left, int right){
        while(left < right){
            if(s.charAt(left) != s.charAt(right)) return false;
            left ++;
            right --;
        }

        return true;
    }
}
```

根据这个思路的另一种做法是，依然利用 longest palindrome 一定是从第一个字符开始的特点，从字符串末尾不断向起点逼近，寻找最终的位置。

如果出现 i j 位置的字符串不同的情况，则重置 i = 0 , j = end - 1 继续看。

不过本质上讲，这种做法和我刚才写的第一种没有任何区别，只不过是一个从里向外，一个从外向里。。。于是这个写法在 **large test case** 上依然 **TLE**，还没有利

用好 substring 结构的精髓。

这个解法的代码是参考的 这个帖子 ，2015年9月写的，那个时候还没有大数据的 test case.

```
public class Solution {
    public String shortestPalindrome(String s) {
        if(s.length() <= 1) return s;

        int i = 0;
        int end = s.length() - 1;
        int j = end;

        StringBuilder sb = new StringBuilder();

        while(i < j){
            if(s.charAt(i) == s.charAt(j)){
                i ++;
                j --;
            } else {
                i = 0;
                end --;
                j = end;
            }
        }

        return new StringBuilder(s.substring(end + 1)).reverse()
.toString() + s;

    }

}
```

于是在论坛 这个帖子 里出现了字符串大杀器，KMP. 关于这个问题，Yu's Garden 的帖子讲的非常赞，推荐一读。

KMP 算法的核心是 next function ，在 currect position 不匹配的情况下，在之前的substring中寻找最长的 suffix 后缀，使得它和 substring 中的 prefix 前缀相同。

如果我们的字符串可以分拆成两段 $$S = QP$$， 我们想要求的是最长的 palindrome $$Q$$. 设 $$S'$$ 为 String $$S$$ 的反序字符串，给定

$$SS'$$ = $$QPP'Q'$$ ，由于 $$Q$$ 是 palindrome，可知 $$Q = Q'$$，二者分别是是 $$SS'$$ 的前缀与后缀，因此可以直接通过计算 $$SS'$$ 的 failure function 求出 $$Q$$ 的最大长度。

这个问题能成功 AC 的关键就是。。一言不合就上 KMP !

第二天重写发现的问题：

Pattern 顺序不能错，是 SS' 而不是 S'S

```
public class Solution {
    public String shortestPalindrome(String s) {
        if(s.length() <= 1) return s;
        int[] kmp = getKmpTable(s + "#" + new StringBuilder(s).r
everse().toString());
        int length = kmp[kmp.length-1];

        return new StringBuilder(s.substring(length)).reverse().
toString() + s;

    }

    private int[] getKmpTable(String pattern){
        int M = pattern.length();
        int[] next = new int[M];

        int k = 0;
        for (int q = 1; q < M; q++) {
            while(k > 0 && pattern.charAt(k) != pattern.charAt(q
)) {
                k = next[k-1];
            }
            if (pattern.charAt(k) == pattern.charAt(q)) {
                k++;
            }
            next[q] = k;
        }
        return next;
    }

}
```

5/24 String, Palindrome Continued

- 给定一个 **String**，考虑所有 **substring / interval** 的过程和 **CLRS** 中的 **rod-cutting** 是非常接近的，每一个可能的 **substring** 其实都只是下图中的某一段。大多数问题中 **substring** 的结构只会比 **rod-cutting** 更简单，比如只切一下，或者 **rod** 一定从最左端开始，**etc.**

Figure 15.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 15.1. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        q = max(q, p[i] + r[j - i])
    r[j] = q
return r[n]
```

- 在做 **DP** 之前，仔细思考状态转移方程与递推关系式（**optimal substructure**），尤其是求 **min / max** 的 **DP**，要认真考虑下到底最优解之间是不是“相邻”的。

- 判断一个 **DP** 结构的结果正确与否，要看从 **base case** 开始，每一次 **update** 的结构是否是正确解

Palindrome Partitioning II

LeetCode Hard 题。这题其实得到个能用的正确解不是难事，依照 Palindrome Patitioning I 的思路根据所有 substring 建个 dp 矩阵，然后递归也可以算，不过太慢了。在这种只要求返回最终解 int 的题里，一般都有比较妖孽的优化或者 dp，比较暴力的 divide & conquer 是不够的。

这个问题其实是在问，给定你一个 string ，虽少可以拆成几个 palindrome substring (减一).

试了几种超时/错误的解法之后，发现这题其实就是 rod cutting 的变种。

```
public class Solution {
    public int minCut(String s) {
        int n = s.length();
        if(n <= 1) return 0;

        boolean[][] dp = new boolean[n][n];

        for(int i = 0; i < s.length(); i++){
            for(int j = 0; j <= i; j++){
                if(s.charAt(i) == s.charAt(j) &&( i - j <= 2 ||
dp[i - 1][j + 1])){
                    dp[i][j] = dp[j][i] = true;
                }
            }
        }

        return getMinPieces(s, dp, 0, n - 1) - 1;
```

```
    }

    private int getMinPieces(String s, boolean[][] isPalindrome,
int start, int end){
        if(start == end) return 1;
        if(isPalindrome[start][end]) return 1;

        int min = s.length();
        for(int i = start; i < end; i++){
        // 回头再看，这段其实就是未优化版的 dp 逻辑。
        // 把 left 替换成 dp[]，right 替换成 isPalindrome 就是 dp 了。

        // 再仔细思考下的话，每一对 index 的区间 [i , j]
        // 其实在递归中重复出现了许多次，并且又满足 optimal substructur
e.
            int left = getMinPieces(s, isPalindrome, start, i);
            int right = getMinPieces(s, isPalindrome, i + 1, end
);
            min = Math.min(min, left + right);
        }

        return min;
    }
}
```

另一个尝试是直接 int[][] dp, 过了 20/28 个 test case，不过结果错误，挂在了 test case "ababbbabbaba" 上，应该返回3，这个代码返回1.

正确的minCut 是 aba|b|bbabb|aba

我的代码在检查"ababbb" 这个substring 的时候会出错，在看 a|(babbb) 的时候，我没考虑到其实 j 可以向右跳两步构造出最短的 palindrome. 目前的 DP 代码只假设了相邻一步的，导致结果不正确。

这个错误说明了：

在 **Palindrome** 中，依赖相邻字符的**optimal substructure** 只在 **“**某个子串是不是**palindrome”** 上有效。

在**“**最少子串数量**”**上，只检查相邻字符的 **optimal substructure**是有问题的，因为最优的 **cut** 可能在任意的其他位置，而不是相邻。

```
public class Solution {
    public int minCut(String s) {
        int n = s.length();
        if(n <= 1) return 0;

        int[][] dp = new int[n][n];

        for(int i = 0; i < s.length(); i++){
            for(int j = i; j >= 0; j--){
                if(s.charAt(i) == s.charAt(j)){
                    if(i - j < 2){
                        dp[i][j] = dp[j][i] = 1;
                    } else {
                        int middle = dp[i - 1][j + 1];
                        int left = dp[i - 1][j] + 1;
                        int right = dp[i][j + 1] + 1;

                        dp[i][j] = dp[j][i] = Math.min(middle, M
ath.min(left, right));
                    }
                } else {
                    if(i - j < 2){
                        dp[i][j] = dp[j][i] = 2;
                    } else {
                        int middle = dp[i - 1][j + 1] + 2;
                        int left = dp[i - 1][j] + 1;
                        int right = dp[i][j + 1] + 1;

                        dp[i][j] = dp[j][i] = Math.min(middle, M
ath.min(left, right));
                    }
                }
            }
        }

        return dp[0][s.length() - 1] - 1;
    }
}
```

借鉴了论坛上的解法之后，能 AC 的代码如下：

这个代码其实是做了两个 DP. 一个是利用 palindrome substring 结构里合理的相邻 optimal substructure，构造出所有 substring 的 isPalindrome 矩阵；

另一个是关于 substring palindrome 数量的，虽然上一段代码已经说明了，我们不能只根据数量去推导 optimal substructure，因为给定 S, S + 'x' 的最优 cut 不一定相邻，因此破坏了 “相邻 optimal substructure” 的条件。

然而，如果已知 **S** 是 **palindrome** 的话，**S + 'x'** 一定不是 **palindrome**，这就是一个有效的**“**相邻 **optimal substructure”.**

在下面这段代码里，我们只有在已知一个 substring 是 palindrome 的情况下，才去利用这层递推关系式。

每一个 i 位置只会被更新一次，并且是正解，因为在每次 i 的循环中，我们检查了所有可能的 substring，并且在发现 palindrome 的情况下更新了当前 i 的最小值。

- 在每一个位置 **i** 上，左边都是之前的 **dp** 计算好的，右边都是在循环中自己检查的，每个位置的最优解是两段拼接的结果。

```
public class Solution {
    public int minCut(String s) {
        if(s == null || s.length() <= 1) return 0;
        int len = s.length();

        boolean[][] isPalindrome = new boolean[len][len];
        int[] dp = new int[len];

        for(int i = 0; i < len; i++){
            dp[i] = i;
            for(int j = 0; j <= i; j++){
                if(s.charAt(i) == s.charAt(j) && (i - j < 2 || i
sPalindrome[j + 1][i - 1])){
                    isPalindrome[i][j] = isPalindrome[j][i] = tr
ue;
                    if(j == 0){
                        dp[i] = 0;
                    } else {
                        dp[i] = Math.min(dp[i], dp[j - 1] + 1);
                    }
                }
            }
        }

        return dp[len - 1];
    }
}
```

5/25 String 杂题，String 大数运算

Add Binary

- 当问题非常简单的时候，解题重点就从优化时间复杂度变成了优化代码简洁性。

我就不回头看自己那么挫的第一个 AC 答案了。。贴个论坛的(加上我自己的改动)里面有几个很巧妙的优化使代码变得简洁易懂。

这题代码简洁的重点在于处理“终止条件”，因为两个 string 很可能长度不一，也有可能两个 string 加完了之后还有进位没处理。

这段代码给出的答案是：

- 输入长短不一就都放在 **while loop** 里，在读取字符时把短的做 **padding.**
- **While loop** 里三个 **'OR'** 的条件保证了三种不同情况下都会继续读取，而其他两个自动 **pad 0.**

```java
public class Solution {
    public String addBinary(String a, String b) {
        StringBuilder sum = new StringBuilder();
        int i = a.length() - 1;
        int j = b.length() - 1;
        int carry = 0;
        while (i >= 0 || j >= 0 || carry == 1) {
            int digitA = i < 0 ? 0 : a.charAt(i--) - '0';
            int digitB = j < 0 ? 0 : b.charAt(j--) - '0';
            sum.append((digitA + digitB + carry) % 2);
            carry = (digitA + digitB + carry) / 2;
        }
        return sum.reverse().toString();
    }
}
```

Add Two Numbers

上题完全一样的思路应用在链表上就是这样了。

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(0);
        ListNode head = dummy;

        int carry = 0;
        while(l1 != null || l2 != null || carry > 0){
            int num1 = (l1 == null) ? 0 : l1.val;
            int num2 = (l2 == null) ? 0 : l2.val;

            ListNode node = new ListNode((num1 + num2 + carry) %
10);
            carry = (num1 + num2 + carry) / 10;

            if(l1 != null) l1 = l1.next;
            if(l2 != null) l2 = l2.next;

            head.next = node;
            head = head.next;
        }

        return dummy.next;
    }
}
```

Multiply Strings

又进阶了， 这次是乘法，比加法复杂。下面的代码是我写的略为粗糙的第一版。

这题的时间复杂度不会低于 $$O(mn)$$ ，因为毕竟每个 digit 都要和另外的 string 相乘一次。 所以把运算拆成两部分，一部分是 longer string 与单数位乘法，另一个是一个数位结束之后，两数相加。每一个数位上的结果乘出来之后要对应的做 0 padding，代表左位移。

提交时候遇到的一个 bug 是 "9133" x "0" 的情况，要记得在 multiply() 函数中如果输入是 0 就直接返回 "0"，这是乘法和加法运算的一个不同之处，否则会返回“0000”.

```
public class Solution {
    public String multiply(String num1, String num2) {
        String longStr = (num1.length() > num2.length()) ? num1
: num2;
        String shortStr = (num1.length() > num2.length()) ? num2
 : num1;

        String rst = "";

        int index = 0;
        while(shortStr.length() - 1 - index >= 0){
            int num = shortStr.charAt(shortStr.length() - 1 - in
dex) - '0';
            String cur = multiplyOne(longStr, num);
            for(int i = index; i > 0; i--){
                cur += "0";
            }

            rst = addTwo(rst, cur);
            index ++;
        }

        return rst;
    }

    private String multiplyOne(String src, int num){
        if(num == 0) return "0";

        StringBuilder sb = new StringBuilder();
        int carry = 0;
```

```
        for(int i = src.length() - 1; i >= 0; i--){
            int digit = src.charAt(i) - '0';
            sb.append((digit * num + carry) % 10);
            carry = (digit * num + carry) / 10;
        }
        if(carry > 0) sb.append(carry);

        return sb.reverse().toString();
    }

    private String addTwo(String num1, String num2){
        StringBuilder sb = new StringBuilder();
        int i = num1.length() - 1;
        int j = num2.length() - 1;
        int carry = 0;
        while(i >= 0 || j >= 0 || carry > 0){
            int digit1 = (i >= 0) ? num1.charAt(i--) - '0': 0;
            int digit2 = (j >= 0) ? num2.charAt(j--) - '0': 0;
            sb.append((digit1 + digit2 + carry) % 10);
            carry = (digit1 + digit2 + carry) / 10;
        }

        return sb.reverse().toString();
    }
}
```

前面的做法对于每一个 digit 都做了一个乘法和加法操作，并没有特别好的利用到乘法的特点，中间操作过多而且生成了好多不必要的 String.

下面的解法是九章算法的，简洁高效。

- 两个位数为 **m** 和 **n** 的数字相乘，乘积不会超过 **m + n** 位。
- 乘法操作从右往左计算时，每次完成相加就确定了当前 **digit.**
- 同一个 **digit** 多次修改，用 **int[ ].**

```
public class Solution {
    public String multiply(String num1, String num2) {
        if(num1 == null || num2 == null) return null;

        int maxLength = num1.length() + num2.length();
        int[] nums = new int[maxLength];
        int i, j, product, carry;

        for(i = num1.length() - 1; i >= 0; i--){
            // 中间部分相当于多位数乘一位数，起始 carry 为 0
            carry = 0;
            for(j = num2.length() - 1; j >= 0; j--){
                int a = num1.charAt(i) - '0';
                int b = num2.charAt(j) - '0';

                product = nums[i + j + 1] + a * b + carry;
                nums[i + j + 1] = product % 10;
                carry = product / 10;
            }
            // 循环结束，最左面为当前最高位数，如果 carry 还有就设过去
            nums[i + j + 1] = carry;
        }
        StringBuilder sb = new StringBuilder();
        int index = 0;
        while(index < maxLength - 1 && nums[index] == 0) index +
+;
        while(index < maxLength) sb.append(nums[index++]);

        return sb.toString();
    }
}
```

5/26 String， 序列化与压缩

- **Bencode** 格式“长度 **+** 分隔符 **+** 字符串”， 和 **OS** 里的 **metadata header** 是一个思路，在最开始告诉你 **data** 地址 **/** 长度 **/ offset**

- **Escape** 符的定义是：**"An escape character is a character which invokes an alternative interpretation on subsequent characters in a character sequence."**

Encode and Decode Strings

这题挺好玩的，既实用又有意思，做了一番调查之后有两种方法都可以写。

- 这题考察的就是在 **serialization** 中如何解决各种歧义，还有 **corner case** 是否考虑的足够周全。

- 当发现题目本身重点考察 **corner case** 的时候，一定记得先列出需要解决的问题，把各种 **corner case** 先列出来再动手。

输入字符在 256 个 ASC II 码的集合中，所用的 encode 字符也是同一个集合。所以不能引入新的特殊符号去解决问题。

- 用什么字符做分隔符，表示 string 的间隔？
- 如果原字符中有分隔符，如何区分？
- 如何正确的 encode 空字符串，还有多个空字符串？

举个例子，我如果用 '#' 代表字符分隔符，'#' 代表原字符串中的 '#' 的话，几个必须要考虑的 test case 是：

- ["\"]
- ["#"]
- ["\#"]
- [""]
- ["", ""]

先说一下我几次不成功的尝试。。下面的代码不能 AC，只能过 305 / 316 个 test case.

基本 encode 思路是让每个 encoded string 不以 '#' 结尾，落单的 '#' 代表空字符串。

然后 decode 时依次读取，看到字符先加进去，如果发现后面的是 '#' 而前一个是 '\' 的话，抹掉一位。

不过依然解决的不完美，因为下面这个 test case 里

Input: ["" ""]

Output: [""]

Expected: ["",""]

中间的过渡 string 是 "#"。

这个问题在 StackOverflow 上有个帖子讲的不错，思路和我原来的代码也很接近。

```
public class Codec {

    // Encodes a list of strings to a single string.
    public String encode(List<String> strs) {
        StringBuilder sb = new StringBuilder();
        for(String str : strs){
            for(int i = 0; i < str.length(); i++){
                if(str.charAt(i) == '#'){
                    sb.append('\\');
                }
                sb.append(str.charAt(i));
            }
            sb.append('#');
        }
        if(sb.length() > 1) sb.setLength(sb.length() - 1);
```

```
        return sb.toString();
    }

    // Decodes a single string to a list of strings.
    public List<String> decode(String s) {
        List<String> list = new ArrayList<String>();
        StringBuilder sb = new StringBuilder();

        for(int i = 0; i < s.length(); i++){
            if(s.charAt(i) != '#'){
                sb.append(s.charAt(i));
            } else {
                // Found '\#' , remove '\' and add '#'
                if(i - 1 >= 0 && s.charAt(i - 1) == '\\'){
                    sb.setLength(sb.length() - 1);
                    sb.append('#');
                } else {
                    list.add(sb.toString());
                    sb.setLength(0);
                }
            }
        }
        if(sb.length() > 0){
            list.add(sb.toString());
        }

        return list;
    }
}

// Your Codec object will be instantiated and called as such:
// Codec codec = new Codec();
// codec.decode(codec.encode(strs));
```

BitTorrent 有一个P2P跨平台的序列化规范，简单易懂，叫 Bencode, 简单讲就是“长度 + 分隔符 + 字符串”的 encoding.

于是有了这段参考 LeetCode 论坛之后的代码。 其实空间时间上不算特别经济，但是胜在简洁。原 string 中间如果有 delimiter 也不要紧，因为可以根据 length 直接跳过，再寻找 delimiter 的时候一定是有效字符。

- 思想上讲和 **OS** 的 **file system** 是非常像的，在实际 **data** 最前面的 **header** 里会存 **metadata**，告诉你下一段数据的内存地址或者**offset**。

```
public class Codec {

    // Encodes a list of strings to a single string.
    public String encode(List<String> strs) {
        StringBuilder sb = new StringBuilder();
        for(String str : strs){
            sb.append(str.length());
            sb.append(':');
            sb.append(str);
        }
        return sb.toString();
    }

    // Decodes a single string to a list of strings.
    public List<String> decode(String s) {
        List<String> list = new ArrayList<String>();

        int i = 0;
        while(i < s.length()){
            int delimiter = s.indexOf(':', i);
            int size = Integer.valueOf(s.substring(i, delimiter)
);
            list.add(s.substring(delimiter + 1, delimiter + 1 +
size));
            i = delimiter + 1 + size;
        }

        return list;
    }
}
```

继续参考了一下论坛的各种解法之后，不利用length信息这题也可以做，只是需要对“escape符”有更好的理解才行。

- **Escape** 符的正确定义是：看到了就跳过，但是 **escape** 符后面的一定是有效字符。
- 于是一个落单的 **'#'** 代表单词 **delimiter ;**
- 原字符串的 **'#'** 变成了 **'/ + #'**；这样保证可以加进去但是又不会作为 **'#'** 单独被判断；
- 原字符串的 **'/'** 就变成了 **'//'**，实际意义是 **"escape 符 + 有效字符"**。

```
public class Codec {

    // Encodes a list of strings to a single string.
    public String encode(List<String> strs) {
        StringBuilder sb = new StringBuilder();
        for(String str : strs){
            for(int i = 0; i < str.length(); i++){
                char chr = str.charAt(i);
                if(chr == '/'){
                    sb.append("//");
                } else if(chr == '#'){
                    sb.append("/#");
                } else {
                    sb.append(chr);
                }
            }
            sb.append("#");
        }

        return sb.toString();
    }
```

```
    // Decodes a single string to a list of strings.
    public List<String> decode(String s) {
        List<String> res = new ArrayList<>();
        StringBuilder str = new StringBuilder();

        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '/') {
                str.append(s.charAt(++i));
            } else if (s.charAt(i) == '#') {
                res.add(str.toString());
                str.setLength(0);
            } else {
                str.append(s.charAt(i));
            }
        }

        return res;
    }
}

// Your Codec object will be instantiated and called as such:
// Codec codec = new Codec();
// codec.decode(codec.encode(strs));
```

Unique Word Abbreviation

这题本身非常简单，但是写 sample test case 的人有点智障。。。得看着挂了的 test case 一点一点猜才能知道这题到底是想干啥。

这题的意思是，我们要看的是一个 key 到底是否有效； 给你一个 word，如果字典里面完全没有一样的 abbreviation -> true; 如果有一样的 abbreviation 但是全和你的 word 是一样的单词，也返回 true；其他的都是 false;

```
public class ValidWordAbbr {
    HashMap<String, String> map;
    public ValidWordAbbr(String[] dictionary) {
        map = new HashMap<String, String>();
        for(String str:dictionary){
            String key = getKey(str);
            // If there is more than one string belong to the sa
me key
            // then the key will be invalid, we set the value to
 ""
            if(map.containsKey(key)){
                if(!map.get(key).equals(str)){
                    map.put(key, "");
                }
            } else {
                map.put(key, str);
            }
        }
    }

    public boolean isUnique(String word) {
        return !map.containsKey(getKey(word))||map.get(getKey(wo
rd)).equals(word);
    }

    String getKey(String str){
        if(str.length() <= 2) return str;
        return str.charAt(0)+Integer.toString(str.length()-2)+st
r.charAt(str.length()-1);
    }
}
```

Generalized Abbreviation

典型的搜索问题，对于 "word" ，该词的搜索图和状态空间如图所示：

第一遍 AC 的代码~

- 错误**1**：不能盲目用 **indexOf(len)** 去找下一个起点，因为 **len** 作为一个数字是可以重复的。
- 错误**2**：用 **lastIndexOf** 也不行，因为数字不都是一位数，有肯能会出现重复数字的两位数。
- 因此， 最稳的做法是，**pos** 一定新 **string** 的起点，然后加上数字 **string** 的长度，再加一。

```
public class Solution {
    public List<String> generateAbbreviations(String word) {
        List<String> list = new ArrayList<String>();
        list.add(word);
        dfs(list, word, 0);
        return list;
    }

    private void dfs(List<String> list, String word, int startIn
dex){
        if(startIndex >= word.length()) return;

        // Iterate through all lengths
        for(int len = 1; len <= word.length() - startIndex; len+
+){
            for(int pos = startIndex; pos <= word.length() - len
; pos++){
                String nextWord = word.substring(0, pos) + len +
 word.substring(pos + len);
                String number = "" + len;
                list.add(nextWord);
                dfs(list, nextWord, pos + number.length() + 1);
            }
        }
    }

}
```

参考了论坛之后，觉得这个做法也特别巧妙，图像化的话，和我刚才画的搜索图不一样，而是从原始 **String** 出发，对于每个位置进行检查；每个位置都有 **"**加字母**"** 或者 **"**不加，并入数字**"** 两种选择，和 **remove parentheses** 与 **add operators** 的思路非常的像，时间复杂度也一目了然，**O(2^n).**

这种在 **String** 上，相对于每个字符考虑多种可能的 **DFS + backtracking**，简单强大，非常值得总结一个**~**

```
public class Solution {
    public List<String> generateAbbreviations(String word) {
         List<String> list = new ArrayList<>();
         dfs(list, new StringBuilder(), word.toCharArray(), 0, 0
);
         return list;
    }

    private void dfs(List<String> list, StringBuilder sb, char[]
 word, int index, int curNum){
        int len = sb.length();
        if(index == word.length){
            if(curNum != 0) sb.append(curNum);
            list.add(sb.toString());
        } else {
            // Don't add, merge to current number
            dfs(list, sb, word, index + 1, curNum + 1);

            // Add current char
            if(curNum != 0) sb.append(curNum);
            dfs(list, sb.append(word[index]), word, index + 1, 0
);
        }
        sb.setLength(len);
    }
}
```

(Google) 字符串嵌套压缩 / 解压缩

变种1：

http://www.1point3acres.com/bbs/thread-189365-1-1.html

上来大概问了5分钟background然后直接上题，一个string decompression的题。。不知道是不是原题反正没见过。。题目如下.

2[abc]3[a]c => abcabcabcaaac; 2[ab3[d]]2[cc] => abdddabdddcc 输入 输出 一开始用了一个栈，写着写着嵌套的逻辑卡住了，最后用俩stack解决。。然后follow-up问的是不要输出string而是输出解压后的K-th character，主要也还是嵌套情况就从内

到外把疙瘩解开以后再算。。然后我问俩问题就结束了。整体感觉妹子面试官人很nice 反应很快而且不是特别picky的那种。

变种2：

“3a2[mtv]ac”, decompress to: aaamtvmtvac，括号可以嵌套。 这个我觉得不是很难，大概花了15分钟理清了思路并写好了代码，大概就是找匹配括号递归解，面试官也找不到bug表示认同。

但吊诡的地方来了，面试官说把这种字符串compress回去...这显然有多种情况，于是我问是不是要求压缩后最短，面试官说肯定越短越好。 比如对于aaaa, 肯定4a比2[aa]好。

5/24 String 杂题

今天脑洞开的有点大，写点水题压压惊。。

Valid Parentheses

我发现当输入情况就那么几个的时候，用 switch case 是个省事的好办法。。可以直接把 hashmap 都省了。

```java
public class Solution {
    public boolean isValid(String s) {
        if(s.length() == 0) return true;
        Stack<Character> stack = new Stack<Character>();

        for(int i = 0; i < s.length(); i++){
            char chr = s.charAt(i);
            if(stack.isEmpty()){
                stack.push(chr);
            } else {
                if(chr=='(' || chr=='[' || chr == '{'){
                    stack.push(chr);
                } else {
                    if(getPair(chr) == stack.peek()){
                        stack.pop();
                    } else {
                        return false;
                    }
                }
            }
        }

        return stack.isEmpty();
    }

    private char getPair(char chr){
        switch(chr){
            case ')':
                return '(';
            case ']':
                return '[';
            case '}':
                return '{';
            default:
                return '.';
        }
    }
}
```

Longest Common Prefix

自己匆忙写的，有点挫。

```
public class Solution {
    public String longestCommonPrefix(String[] strs) {
        if(strs == null || strs.length == 0) return "";

        StringBuilder sb = new StringBuilder();
        int index = 0;
        int maxLength = 0;
        for(int i = 0; i < strs.length; i++){
            maxLength = Math.max(maxLength, strs[i].length());
        }

        while(index < maxLength){
            if(index == strs[0].length()) return sb.toString();
            char chr = strs[0].charAt(index);
            for(int i = 0; i < strs.length - 1; i++){
                String cur = strs[i];
                String next = strs[i+1];

                if(index == cur.length() || index == next.length
()){
                    return sb.toString();
                }
                if(cur.charAt(index) != next.charAt(index)){
                    return sb.toString();
                }
            }
            sb.append(chr);
            index ++;
        }


        return sb.toString();
    }
}
```

论坛上好一点的写法：

```
public class Solution {
    public String longestCommonPrefix(String[] strs) {
        if (strs.length < 1 || strs == null) {
            return "";
        }
        if (strs.length == 1) {
            return strs[0];
        }
        //find the shortest String
        int shortest = 0;
        int len = strs[0].length();
        for (int i = 1; i < strs.length; i++) {
            int curLen = strs[i].length();
            if (curLen < len) {
                len = curLen;
                shortest = i;
            }
        }
        //find the longest common prefix
        String sub = strs[shortest];
        for (int i = 0; i < strs.length; i++) {
            while (strs[i].indexOf(sub) != 0) {
                sub = sub.substring(0, sub.length()-1);
            }
        }
        return sub;
    }
}
```

- 当问题非常简单的时候，解题重点就从优化时间复杂度变成了优化代码简洁性。

Group Anagrams

比较 Trivial 的问题，没啥特别好说的。。

这题主要的乐趣在于怎么把多个是 anagram 的 string 映射到同一个 key 上。简单的办法是排序，或者开个 int[] 统计字母数量，然后生成一个类似于 1a2b4f 这种的 string signature.

除此之外比较 fancy 的写法是利用 prime number 做 string hashing，容错率不太好而且我觉得用 prime number 总是要去证明一下算法的正确性，不太适用于面试。

```
public class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
        List<List<String>> rst = new ArrayList<List<String>>();
        if(strs == null || strs.length == 0) return rst;

        HashMap<String, List<String>> map = new HashMap<String,
List<String>>();

        for(String str : strs){
            char[] array = str.toCharArray();
            Arrays.sort(array);
            String sortedStr = new String(array);

            if(!map.containsKey(sortedStr)){
                List<String> list = new ArrayList<String>();
                list.add(str);
                map.put(sortedStr, list);
            } else {
                map.get(sortedStr).add(str);
            }
        }

        for(String str : map.keySet()){
            // OJ wants each list to be sorted
            Collections.sort(map.get(str));
            rst.add(map.get(str));
        }

        return rst;
    }
}
```

Length of Last Word

简直惊了。。这题有难度吗。。

```
public class Solution {
    public int lengthOfLastWord(String s) {
        int length = 0;
        for(int i = s.length() - 1; i >= 0; i--){
            if(s.charAt(i) == ' ') continue;
            while(i >= 0 && s.charAt(i) != ' '){
                length ++;
                i --;
            }
            break;
        }

        return length;
    }
}
```

Knuth–Morris–Pratt 字符串匹配

当初刚转CS研究生，上九章算法班的时候，第一讲就是 strStr() ，黄老师特别强调面试时候不要因为知道某个 fancy 算法就去写。当时觉得面试时候写个 KMP 挺高大上的，有炫技嫌疑而且容易犯错误。

上了两年计算机课之后，我现在觉得 KMP 这东西挺简单的，很好实现又很好理解，为什么不写。感谢马里奥，在考试题里能出现“ linear time 实现支持 wildcard 的 KMP ”算法之后，写个原版的 KMP 简直是太良心了。。。

两年的时间，真快啊。

这题的 KMP 解法已经在 LeetCode 论坛上到处都是而且被提交烂了。不过我自己还是更喜欢 CLRS 上的伪代码。

```
public class Solution {
    public int strStr(String haystack, String needle) {
        if(haystack.length() < needle.length()) return -1;
        if(needle.length() == 0) return 0;

        int[] next = getNext(needle);
        int q = 0; // number of chars matched in pattern
        for(int i = 0; i < haystack.length(); i++){
            while(q > 0 && needle.charAt(q) != haystack.charAt(i
)){
                q = next[q - 1];
            }
            if(needle.charAt(q) == haystack.charAt(i)){
                q ++;
            }
            if(q == needle.length()){
                return i - needle.length() + 1;
            }
        }
        return -1;
    }
```

```
    private int[] getNext(String pattern){
        int M = pattern.length();
        int[] next = new int[M];
        int k = 0; // number of chars matched in pattern
        for(int i = 1; i < M; i++){
            while(k > 0 && pattern.charAt(k) != pattern.charAt(i
))){
                k = next[k - 1];
            }
            if(pattern.charAt(k) == pattern.charAt(i)){
                k ++;
            }
            next[i] = k;
        }

        return next;
    }
}
```

next[] 里的 k = 正确 match 的长度

next[] 中，每个位置的数字是由 k 赋值的，代表“如果下一个字符串挂了，在我这个位置截止的字符串正确 match 的长度是多少”

- 于是这个 getNext() 函数就很好解释了。 next[] 的大小等于 pattern 长度，k 初始值为 0.
- next[0] = 0 因为 substring 长度如果只为 1 的话，前面没东西和它匹配。
- 于是开始一个 while 循环，迭代寻找如果当前字符串挂了，我们目前的最长 suffix 到底多长，有可能会跳很多步。这个写法有点类似于 disjoint set 里面 weighted union-find 的 path compression 实现，就是一个 while 循环迭代赋值 index 一直到正确的 / base case 为止。 k > 0 这个条件很重要，不然如果在第一个字符串挂了之后，会去找 next[-1] 就越界了。
- 每次我们在 index k 上挂的时候，是去找 next[k - 1] 的 k 值是什么。原因是 length 与 index 间有 1 的 offset ，我们去看 index = k 的位置其实是在考虑要不要把 length 设成 k + 1.

- 此后如果当前字符串匹配，就把 k + 1，赋值到当前 next[i] 上。赋值之后就不会再改了。

match 函数的逻辑基本和 getNext 完全一样，k 代表目前的 text 上 match pattern 的字符串长度。

- 当 q = pattern.length() 的时候，从 i 开始往回挪动 q 步，因为挪动前 i 处在 pattern 最后一个字符，要再往回挪动一个位置。
- i - needle.length() + 1;

(G) 面经题 http://www.1point3acres.com/bbs/thread-199776-1-1.html

给两个字符串，找到第二个在第一个中第一次出现的位置（自己写**string.indexOf**这个函数吧），**followup1**，找一个字符串中**period**的字符段，**followup2**，找到**period**次数最少的，例如**abababab**，**ab**出现了**4**次，**abab**出现了**2**次，返回**2**

Lempel–Ziv–Welch 字符串压缩算法

MIT 的 Information and Entropy 公开课讲的信息论与压缩，讲的非常 nice ！唯一美中不足的地方是教授一步一步复现 LZW 算法的时候一开始搞错了。。卡住之后才发现出了 bug 修正过来，囧

http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-050j-information-and-entropy-spring-2008/videos-homework-and-readings/unit-2-lecture-1/

LZW 算法的一个 pdf 课件。

http://web.mit.edu/6.02/www/s2012/handouts/3.pdf

简单看了两天笔记之后，总结了下，教材上常见的字符串压缩可以分为 Huffman encoding 和 LZ 系列 - LZ77, LZ78, LZW 等等。

- ## Huffman Encoding:
  - 建立在字符pattern的“出现概率”上，把出现概率大的pattern用更短的encoding 方式表达出来，这样就可以节省原始字符串的期望长度。
  - Huffman Encoding 的结构是树状的，一种常见的 variable-length code (又称前缀树) 大概长这样：

  - 其中 0 - B; 10 - A; 110 - C，以此类推。
  - Theorem 3.1 Huffman coding produces a code tree whose expected length is optimal, when restricted to symbol-by-symbol coding with symbols drawn in IID fashion from a known symbol probability

distribution.

- 在 **symbol** 的 **pdf** 已知的情况下，**Huffman Encoding** 的期望长度是最优的。问题就出在这个 **“symbol pdf** 已知**”** 上，因为实际应用中很多时候得到 **symbol** 的 **pdf** 不但昂贵，也不擅于处理不同 **message** 之间，**symbol** 出现概率可能出现的大幅度变化。而且这种方法做了一个类似 **naive bayes** 的假设，就是 **symbol** 之间是 **iid** 关系，**independent and identically distributed.**

因此我们需要一种 **adaptive** 并且 **variable-length** 的算法，因此有了 **Lempel-Ziv-Welch** 算法。

- **LZW** 算法最有意思的一点是，在 **stream of input/output** 进行的过程中，双方不需要传递用于**encode/decode** 的 **String table** ，**decoder** 可以**"**猜**"**出来应该对应的是什么。 而且如果原始 **table** 已经满了的话，可以直接重新初始化，继续 **streaming.**

S	msg. byte	lookup	result	transmit	string table
–	a	–	–	–	–
a	b	ab	not found	index of a	table[256] = ab
b	c	bc	not found	index of b	table[257] = bc
c	a	ca	not found	index of c	table[258] = ca
a	b	ab	found	–	–
ab	c	abc	not found	256	table[259] = abc
c	a	ca	found	–	–
ca	b	cab	not found	258	table[260] = cab
b	c	bc	found	–	–
bc	a	bca	not found	257	table[261] = bca
a	b	ab	found	–	–
ab	c	abc	found	–	–
abc	a	abca	not found	259	table[262] = abca
a	b	ab	found	–	–
ab	c	abc	found	–	–
abc	a	abca	found	–	–
abca	b	abcab	not found	262	table[263] = abcab
b	c	bc	found	–	–
bc	a	bca	found	–	–
bca	b	bcab	not found	261	table[264] = bcab
b	c	bc	found	–	–
bc	a	bca	found	–	–
bca	b	bcab	found	–	–
bcab	c	bcabc	not found	264	table[265] = bcabc
c	a	ca	found	–	–
ca	b	cab	found	–	–
cab	c	cabc	not found	260	table[266] = cabc
c	a	ca	found	–	–
ca	b	cab	found	–	–
cab	c	cabc	found	–	–
cabc	a	cabca	not found	266	table[267] = cabca
a	b	ab	found	–	–
ab	c	abc	found	–	–
abc	a	abca	found	–	–
abca	b	abcab	found	–	–
abcab	c	abcabc	not found	263	table[268] = abcabc
c	– end –	–	–	index of c	–

Figure 3-6: LZW encoding of string "abcabcabcabcabcabcabcabcabcabcabc"

received	string table	decoding
a	–	a
b	table[256] = ab	b
c	table[257] = bc	c
256	table[258] = ca	ab
258	table[259] = abc	ca
257	table[260] = cab	bc
259	table[261] = bca	abc
262	table[262] = abca	abca
261	table[263] = abcab	bca
264	table[264] = bacb	bcab
260	table[265] = bcabc	cab
266	table[266] = cabc	cabc
263	table[267] = cabca	abcab
c	table[268] = abcabc	c

Figure 3-7: LZW decoding of the sequence $a, b, c, 256, 258, 257, 259, 262, 261, 264, 260, 266, 263, c$

一段比较简单粗暴的代码，跑了几个 test case 结果无误，压缩效率尚可。LZW 因为 prefix 搜索非常多，可以用 Trie 进行优化。

里面一些细节因为时间有限先不赘述了，复习去。

- 实际应用中 **key** 的 **size** 最好小点，至少最好不要比一个 **char** 大(同时保证字典不会 **overflow)**，下面的代码用的就是 **8-bit byte** 做字典 **HashMap** 与 **TrieNode** 的 **key**，有效范围在 **0 ~ 255** 之间。

```
public class Solution {
    private static class TrieNode{
        char chr;
        byte dictIndex;
        TrieNode[] children = new TrieNode[26];
        public TrieNode(){}
        public TrieNode(char chr, byte dictIndex){
            this.chr = chr;
            this.dictIndex = dictIndex;
        }
    }

    // returns code of transmit as integer

    private static byte trieTraverse(String str, int[] index,
                                     HashMap<Byte, String> map,
                                     TrieNode trieRoot, byte[] cu
rCode){
        TrieNode curNode = trieRoot;
        StringBuilder sb = new StringBuilder();
        byte dictIndex = (byte) (str.charAt(index[0]) - 'a');

        while(index[0] < str.length()){
            char curChr = str.charAt(index[0]++);
            int intChr = curChr - 'a';
            sb.append(curChr);
            dictIndex = curNode.dictIndex;

            if(curNode.children[intChr] == null){
                TrieNode newNode = new TrieNode(curChr, curCode[0
]);
                curNode.children[intChr] = newNode;
                map.put(curCode[0]++, sb.toString());
                index[0]--;
```

```
                break;
            } else {
                curNode = curNode.children[intChr];
                dictIndex = curNode.dictIndex;
            }

        }

        return dictIndex;
    }


    private static String encode(String str, HashMap<Byte, Strin
g> map,
                                 TrieNode trieRoot, byte[] curCo
de){
        StringBuilder sb = new StringBuilder();
        int[] index = new int[1];

        while(index[0] < str.length()){
            byte code = trieTraverse(str, index, map, trieRoot,
curCode);
            sb.append(code).append(' ');
        }

        return sb.toString();
    }

    private static String decode(String str, HashMap<Byte, Strin
g> map){
        StringBuilder sb = new StringBuilder();
        String[] strs = str.split(" ");
        for(String token : strs){
            byte index = (byte) Integer.parseInt(token);
            sb.append(map.get(index));
        }
        return sb.toString();
    }

    public static void main(String[] args) {
```

```
        String[] testcases = new String[]{"abcabcabcabcabcabcabc
abcabcabcabcabc",
                "aaaaabbbbbaaaaabbbb",
                "ittyttybitbinbitbitcoin",
                "abracadabrabrabra",
                "aaabbbmtvmtvmtvaaabbb"};
        String input2 = testcases[0];

        HashMap<Byte, String> map = new HashMap<>();
        TrieNode trieRoot = new TrieNode();

        for(int i = 0; i < 26; i++){
            char chr = (char)('a' + i);
            TrieNode node = new TrieNode(chr, (byte)i);
            trieRoot.children[i] = node;
            map.put((byte) i, String.valueOf(chr));
        }

        byte[] curCode = new byte[1];
        curCode[0] = 26;

        String str = encode(input2, map, trieRoot, curCode);
        String output = decode(str, map);

        System.out.println("Input : " + input2);
        System.out.println("Encode: " + str);
        System.out.println("Output: " + output);
        for(int i = 0; i < curCode[0]; i++){
            System.out.println(i + " : " + map.get((byte) i) + "
 ");
        }
    }
}
```

(G) Decode String

Decode String

匆忙写的，有点丑。。之后要优化下。

步骤上不麻烦，遇到嵌套的括号，就把原始 String 变成更小的子问题，递归处理就好了。于是这题操作上总共就三件事：

- 一边扫一边记录当前数字，times 初始化 = 0；
- 找到当前括号的匹配括号；
- 括号之间就是一个新的子问题，递归处理之后把结果用循环添加就好了。

```java
public String decodeString(String s) {
    if(s == null || s.length() == 0) return "";
    StringBuilder sb = new StringBuilder();

    for(int i = 0; i < s.length(); i++){
        char chr = s.charAt(i);
        if(Character.isDigit(chr)){
            int times = 0;
            while(i < s.length() && s.charAt(i) != '['){
                times = times * 10 + (s.charAt(i) - '0');
                i ++;
            }
            int matchIndex = findMatchIndex(s, i);
            String repeat = decodeString(s.substring(i + 1,
matchIndex));

            for(int time = 0; time < times; time++){
                sb.append(repeat);
            }
            i = matchIndex;
        } else {
            sb.append(chr);
        }
    }
    return sb.toString();
}

private int findMatchIndex(String s, int index){
    int count = 0;
    for(; index < s.length(); index++){
        if(s.charAt(index) == '[') count ++;
        else if(s.charAt(index) == ']') count --;

        if(count == 0) break;
    }

    return index;
}
```

(G) UTF-8 Validation

UTF-8 Validation

Binary Tree

6/1 Tree , 各种 Binary Tree 定义

参考 **wikipedia** 各类二叉树的定义

每个节点 **children = 0 / 2**，为 **full binary tree**

简单讲就是没有奇葩的单节点“拐弯”。

按 **level order** 从左到右依次（尽量）填满，为 **complete binary tree.**

LCA 类问题

- **LCA** 类问题是二叉树的高频问题之一；
- 有只给 **root** 的；
- 还有不给 **root** 给 **parent pointer** 的。
- 想面 **FB**，最好把各种二叉树问题的 **recursion / iteration** 还有 **root / parent pointer** 的写法都练熟才行，只 **AC** 是不够的。

Lowest Common Ancestor of a Binary Search Tree

> - 递归 ✓
> - 迭代 ✓
> - 复杂度分析 ✓

LCA 问题就是判断给定两个 node {p,q} 与 root 之间的相对位置。在 BST 里这种相对关系看 node.val 就可以。

时间复杂度 **O(n)**，如果 **Tree** 的形状是一条线往左或右的 **full binary tree** 的话。

```
public class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode
 p, TreeNode q) {
        // Got to check if p and q is null as well;
        if(root == null) return root;

        if(root.val > p.val && root.val > q.val)
            return lowestCommonAncestor(root.left, p, q);
        if(root.val < p.val && root.val < q.val)
            return lowestCommonAncestor(root.right, p, q);

        return root;
    }
}
```

因为是尾递归，显而易见的改法是用 **while**循环省去递归占用的系统栈空间；

```
public class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode
 p, TreeNode q) {
        // Got to check if p and q is null as well;
        while(root != null){
            if(root.val > p.val && root.val > q.val) root = root
.left;
            else if(root.val < p.val && root.val < q.val) root =
 root.right;
            else return root;
        }
        return root;
    }
}
```

Lowest Common Ancestor of a Binary Tree

> - 递归 ✓
> - 迭代 ✓
> - **parent指针 todo**
> - 复杂度分析 ✓

直接写的暴力解法，知道肯定过不了。。判断子树中 contains 的时间复杂度太高了，而且重复调用很多，完全没优化。

这种解法的时间复杂度是 **O(n log n)**，因为对于一个 **node** 来讲，它被 **check** 的次数等于它和 **root** 的距离 **(**也就是 **height)**。

极少会有 **O(n^2)** 的 **binary tree** 算法，因为那意味着每个节点相对于整个树重新计算，而不再是自己从属的路径或者高度。

```
public class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode
p, TreeNode q) {
        if(root == null) return root;

        if(root.val == p.val) return p;
        if(root.val == q.val) return q;

        if(containsNode(root.left, p) && containsNode(root.left,
q)){
            return lowestCommonAncestor(root.left, p, q);
        }
        if(containsNode(root.right, p) && containsNode(root.righ
t, q)){
            return lowestCommonAncestor(root.right, p, q);
        }

        return root;
    }

    private boolean containsNode(TreeNode root, TreeNode node){
        if(root == null) return false;
        if(root.val == node.val) return true;

        return (containsNode(root.left, node) || containsNode(ro
ot.right, node));

    }
}
```

加 containsNode() 函数比较多此一举，其实可以直接用这个函数在 binary tree 上的递归性质去调用自身解决。

函数返回的是 "对于给定 Node 为 root 的 tree 中是否包含 p 或者 q，只要包含一个，就不返回 null 了，而我们相对于当前节点为 root 的结果，就看两边递归的 return value 决定."

时间复杂度 **O(n)**，相对于每个 **node** 来讲，只会被函数调用和计算一次。

另一种时间复杂度的分析方式是，这题的递归结构不过是个 post-order traversal，遍历的复杂度当然是 O(n).

想到这，迭代的写法也比较明显了，写个 post-order traversal 就行。

```
public class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode
 p, TreeNode q) {
        if(root == null || root == q || root == p) return root;

        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);

        if(left != null && right != null) return root;
        else return (left == null) ? right : left;
    }
}
```

一种可能的迭代的写法是借鉴了我们教授 Martin Farach-Colton 的那篇 The LCA Problem Revisited 的 idea，就是先对 Tree 做 pre-processing 然后用 array 保存过渡结果方便快速 query.

其实就是 inorder pre-process 了一遍之后，把 binary tree 当 BST 用。因为 in-order 的 index 就像 BST 里的大小一样，可以直接确定几个节点在树中的相对位置。同时因为我们都是从 root 开始一层一层往下搜索的，也能保证每次循环的 root 都一定 valid.

这个代码的优点是如果做多次 query 的话有一个 pre-processing 的缓存可以很快返回结果；缺点就是多了个 pre-processing 的过程，query 次数少的时候不占便宜。

这个代码可以 AC，基本思路就是先跑一遍 inorder traversal 记录下每个 node 在整个树里面对应的位置；利用 hashmap 对每个 node 实现均摊复杂度 O(1) 的查找，然后根据对应的节点 index 判断 p，q 相对于 root 在 tree 里的位置关系。

中间跑了一次迭代版的 binary tree inorder traversal.

这两个代码都是一次写完直接 AC 的。。。Tree的问题只要思路不搞错可真容易过啊。。。

这个写法只能算是个有意思的迭代思路，空间时间耗费都挺大的，代码量也长，如果追求速度与效率的话，就不要用这个写法骗自己。。

不过如果在同一个 **Tree** 上要跑好多次 **LCA**，这个做法还是比较可取的~

```
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode
 p, TreeNode q) {
        if(root == null) return root;
        if(root.val == p.val) return p;
        if(root.val == q.val) return q;

        HashMap<TreeNode, Integer> map = new HashMap<TreeNode, I
nteger>();
        Stack<TreeNode> stack = new Stack<TreeNode>();

        TreeNode cur = root;
        int index = 0;
        while(cur != null || !stack.isEmpty()){
            while(cur != null){
                stack.push(cur);
                cur = cur.left;
            }
            TreeNode node = stack.pop();
            map.put(node, index++);
            cur = node.right;
        }

        return getLCA(root, p, q, map);
    }

    private TreeNode getLCA(TreeNode root, TreeNode p, TreeNode
q, HashMap<TreeNode, Integer> map){
        if(root == null) return root;
        if(root.val == p.val) return p;
```

```
        if(root.val == q.val) return q;

        while(root != null){
            if(map.get(q) < map.get(root) && map.get(p) < map.ge
t(root)){
                root = root.left;
            } else if(map.get(q) > map.get(root) && map.get(p) >
 map.get(root)){
                root = root.right;
            } else {
                break;
            }
        }

        return root;
    }
```

给一个 二叉树 ， 求最深节点的最小公共父节点。

http://www.1point3acres.com/bbs/thread-199739-1-1.html

```
       1
      / \
     2   3
        / \
       4   5

    return 3

       1
      / \
     2   3
    /   / \
   6   4   5

   return 1
```

先用 recursive ， 很快写出来了， 要求用 iterative.

这题的关键点只有一个：意识到只需要求最底层 level 最左面和最右面节点的 LCA 就可以了。

因此这个问题的递归与迭代就变成了两个子问题：

- 如何递归求 **level order** 和 **LCA**
- 如何迭代求 **level order** 和 **LCA**

白板上写了一遍，都不难，和这章前面以及 level order 的内容完全一样，就不进一步贴代码了。

5/29 Tree 三序遍历，Vertical Order

Binary Tree Preorder Traversal

- preorder 直接用 stack;
- inorder 用 stack + cur;
- postorder 用 stack + cur + prev;

> - 递归 ✓
> - 迭代 ✓
> - **parent** ✓
> - 复杂度 ✓

迭代的写法过于 trivial 就不细说了，记得因为 stack 先入后出的特点，push 的时候先 push 右边的就行。

```
public class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> list = new ArrayList<>();
        Stack<TreeNode> stack = new Stack<>();
        if(root != null) stack.push(root);

        while(!stack.isEmpty()){
            TreeNode node = stack.pop();
            list.add(node.val);
            // Remember to reverse order.. right -> left
            if(node.right != null) stack.push(node.right);
            if(node.left != null) stack.push(node.left);
        }

        return list;
    }
}
```

头一次写带 parent 指针的写法还挺不习惯的。。改了好一会儿。我觉得这个博客写的思路不错，不过要注意这哥们的代码有 bug，root 为 1，左面一条线连到 2 , 3 然后 1 的右孩子为 4 的情况，这个代码只能返回 1,2,3，看不到 4.

https://haixiaoyang.wordpress.com/2012/04/06/pre-order-traverse-of-a-binary-tree-with-parent-pointer/

在写 **parent** 指针遍历的时候，应该多想想 **stack** 写法每次 **pop** 的回溯位置，是很重要的参考。

测试了几个 **case** 没有问题的代码：

- 以 **cur != null** 为 **while** 循环条件；
- 直接输出 **cur.val**
- **cur** 左面有东西就 **cur = cur.left;**
- 左面没有但右面有，**cur = cur.right;**

- 其他情况就属于要往上回溯了，注意这个回溯长度可以是任意的，我们要注意把 **cur** 放到正确的位置上。
- 我们要找的位置相对于从左往上的路线而言，是 **cur** 往上看第一个右节点不为 **null** 的地方；相对于从右往上的路线而言就是无脑往上走直到走向成从左往上为止。
- 在任何时刻追溯完毕发现 **cur.parent** 为空，说明走完了。
- 否则 **cur** 就是 **cur.parent.right.**

追溯流程主要就是图中这两步。

```
    public static void printPreorder(TreeNode root) {
        TreeNode cur = root;
        while(cur != null){
            System.out.println(cur.val);

            if(cur.left != null){
                cur = cur.left;
            } else if(cur.right != null){
                cur = cur.right;
            } else {
                while(cur.parent != null && (cur.parent.right ==
null || cur == cur.parent.right)){
                    cur = cur.parent;
                }
                if(cur.parent == null) break;
                cur = cur.parent.right;
            }
        }
    }
```

Binary Tree Inorder Traversal

- preorder 直接用 stack;
- inorder 用 stack + cur;
- postorder 用 stack + cur + prev;

> - 递归 ✓
> - 迭代 ✓
> - **parent**
> - 复杂度 ✓

stack 写法要记住的细节是，两个 while 循环里都是 cur != null

```
public class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> list = new ArrayList<Integer>();
        Stack<TreeNode> stack = new Stack<TreeNode>();

        TreeNode cur = root;
        while(cur != null || !stack.isEmpty()){
            while(cur != null){
                stack.push(cur);
                cur = cur.left;
            }
            TreeNode node = stack.pop();
            list.add(node.val);
            cur = node.right;
        }

        return list;
    }
}
```

In-order 的 parent 指针写法稍微 tricky 一些，考虑到 in-order 的特性和面经问到的内容，我写了两个函数，主要是 getSuccessor()，然后把 root 放到二叉树最左面为起点依次调用。

getSuccessor() 流程，其实有点像 morris - traversal:

- 如果 **cur** 右子树不为空，返回右子树里面最左面的点**(**也即 **cur.right** 一路向左最远的点**)**
- 否则一路沿着**(**右**child - parent)** 的路线往上走，然后返回 **parent** 就行了。

```
public static void printInorder(TreeNode root) {
     TreeNode cur = root;
     while(cur.left != null){
         cur = cur.left;
     }
     // Now current is at left most pos
     while(cur != null){
         System.out.println(cur.val);
         cur = getSuccessor(cur);
     }
 }

 private static TreeNode getSuccessor(TreeNode cur){
     // Find leftmost node in right subtree
     if(cur.right != null){
         cur = cur.right;
         while(cur.left != null) cur = cur.left;
     } else {
     //
         while(cur.parent != null && cur == cur.parent.right
){
             cur = cur.parent;
         }
         if(cur.parent == null) return null;

         cur = cur.parent;
     }
     return cur;
 }
```

Inorder Successor in BST

这题没 parent pointer,给的也只是 root，但是还算挺有意思的，和上一题有联系。BST 的好处是在任意 node 上都可以通过与目标值比大小来确定方向， 因此就有比较简洁的迭代写法。

BST 里面，任意位置，任意楼层，都可以通过 value 的比较确定相对位置，这是 BST 一个最好用的性质。

因此在 **BST** 里面，确定起来就很简单了，从 **root** 往下走，每次往左拐的时候，存一下，记录着最近一个看到的比 **p.val** 大的 **node** 就行了。

```
public class Solution {
    public TreeNode inorderSuccessor(TreeNode root, TreeNode p)
{
        TreeNode rst = null;

        while(root != null){
            if(root.val > p.val){
                rst = root;
                root = root.left;
            } else {
                root = root.right;
            }
        }

        return rst;
    }
}
```

顺着这个思路讲，**BST** 里面找 **in-order predecessor** 也特简单，这次往左走的时候不记，往右走的时候记，就行了。

```
public class Solution {
    public TreeNode inorderSuccessor(TreeNode root, TreeNode p)
{
        TreeNode rst = null;

        while(root != null){
            if(root.val > p.val){
                root = root.left;
            } else {
                rst = root;
                root = root.right;
            }
        }

        return rst;
    }
}
```

Binary Tree Postorder Traversal

这题是二叉树遍历里面迭代写起来最麻烦的一个，迭代写法在 LeetCode 的标注难度是 Hard.

- **6/5** 复习时候写错了，第一个地方是 **post order** 因为每一步用 **stack** 顶元素作为新的 **cur**，在 **while** 循环中只需要以 **!stack.isEmpty()** 为条件去判断。第二点是， **post-order** 是【左，右，中】，压栈的时候还是先看左再看右的，顺序不能错。

- **post order** 遍历中最重要的是 **prev** 与 **cur** 的相对关系，相当于存了上一步的动作，用作下一步的方向。

- 用一个 **stack** 存着所有的 **candidate node**，栈顶为当前 **candidate**，并且以栈是否为空做唯一判断标准（还有没有要看的 **candidate**）

```
public class Solution {
  public List<Integer> postorderTraversal(TreeNode root) {
      List<Integer> list = new ArrayList<Integer>();
      Stack<TreeNode> stack = new Stack<TreeNode>();

      TreeNode cur = root;
      TreeNode prev = null;
      if(root != null) stack.push(root);

      while(!stack.isEmpty()){
          cur = stack.peek();

          // cur 在 prev 下面，要尝试探索所有 cur 下面的节点
          // 如果 cur 是叶节点的话，会在最后把 prev 赋值成 cur，再
下一轮的时候被 pop 掉。
          if(prev == null || prev.left == cur || prev.right
== cur){
              if(cur.left != null){
                  stack.push(cur.left);
              } else if(cur.right != null){
                  stack.push(cur.right);
              }
          } else if(cur.left == prev){
          // 刚把左边处理完回来，看看右边还有没有节点
              if(cur.right != null) stack.push(cur.right);
          } else {
          // 左右子树都处理完了，处理当前节点。
              list.add(cur.val);
              stack.pop();
          }
          prev = cur;
      }

      return list;
  }
}
```

这题还有一种比较耍流氓的写法，因为太过 trivial 就不当做这题的重点去提了。。。。

这种写法的主要缺点是，当 **tree** 非常大我们只需要输出正确结果时，**reverse list** 的写法必须依赖于存储所有的

- 对于给定序列 **S**，定义 **S'** 为反序序列
- 定义 **R** 为 **root node** 序列，有 **R = R'**
- 定义 **C** 为子节点序列，正确顺序为“左-右”
- 那么 **post order** 的序列顺序为 **CR**
- 如果在做**pre order**时生成 **RC'** 序列，那么反序之后可以得到 **(RC')' = CR' = CR = post order** 序列。

```
public class Solution {
  public List<Integer> postorderTraversal(TreeNode root) {
      List<Integer> list = new ArrayList<Integer>();
      Stack<TreeNode> stack = new Stack<TreeNode>();

      if(root != null) stack.push(root);

      while(!stack.isEmpty()){
          TreeNode node = stack.pop();
          list.add(node.val);
          if(node.left != null) stack.push(node.left);
          if(node.right != null) stack.push(node.right);
      }
      Collections.reverse(list);
      return list;
  }
}
```

Binary Tree Vertical Order Traversal

- 以 **root** 为原点，每次左走一步为 **offset -1** ，右走一步为 **offset +1**，把所有 **offset** 相等的 **node** 放到一个 **list** 里。

这题我一个不成功的尝试是用 dfs 解决。代码上讲 dfs 可以得到所有的 List 并且里面包含所有正确元素，但是无论是 preorder , inorder 还是 postorder，我们不能保证每一个 list 里的元素顺序是正确的，因为可能某一个子树沿着另一边走的很远，导致 dfs 时先把这个点加了进去。

因此为了保证正确顺序，还是得用 bfs，level order traversal.

Map 的选择上可以用 HashMap 或者 TreeMap 都可以，TreeMap 里因为 key 默认是排序的遍历起来省事一些，不过 insert 的时间复杂度更高，导致速度变慢，不如记录 index 里面 key 的范围然后遍历时间上经济。

```
  public class Solution {
      private class Tuple{
        TreeNode node;
        int index;
        public Tuple(TreeNode node, int index){
            this.node = node;
            this.index = index;
        }
      }

      public List<List<Integer>> verticalOrder(TreeNode root) {
        List<List<Integer>> rst = new ArrayList<List<Integer>>()
;
        Map<Integer, List<Integer>> map = new HashMap<Integer, L
ist<Integer>>();

        Queue<Tuple> queue = new LinkedList<Tuple>();
        if(root != null) queue.offer(new Tuple(root, 0));
        int min = 0;
        int max = 0;
        while(!queue.isEmpty()){
            int size = queue.size();
            for(int i = 0; i < size; i++){
                Tuple tuple = queue.poll();
                min = Math.min(min, tuple.index);
                max = Math.max(max, tuple.index);
                if(!map.containsKey(tuple.index)){
                    List<Integer> list = new ArrayList<Integer>(
);
```

```
                    list.add(tuple.node.val);
                    map.put(tuple.index, list);
                } else {
                    map.get(tuple.index).add(tuple.node.val);
                }

                if(tuple.node.left != null) queue.offer(new Tupl
e(tuple.node.left, tuple.index - 1));
                if(tuple.node.right != null) queue.offer(new Tup
le(tuple.node.right, tuple.index + 1));
            }
        }

        for(int i = min; i <= max; i++){
              if(map.containsKey(i))  rst.add(map.get(i));
        }

        return rst;
    }
}
```

Post order traversal 的应用

Binary tree 的 post-order 遍历很实用，其遍历顺序的特点决定了其 bottom-up 的返回顺序，每次子树处理完了在当前节点上汇总结果，可以解决很多 和 subtree, tree path 相关的问题，在多叉树的情况下，也很容易扩展到各类的 search 问题，比如 Android Unlock Patterns.

Lexicographical path

```
        5
       / \
      3   2
     / \   \
    2   4   4
             \
              1
```

FB 面经，自底向上的 path 有【1,4,2,5】，【4,3,5】，【2,3,5】，要求按自底向上的 lexicographical order 返回排序的 path，比如在这里是 【1,4,2,5】, 【2,3,5】, 【4,3,5】

首先从这个树的结构我们可以发现。。不把最后的 leaf node 看完之前我们是不能知道所有 list 大小信息的，比如这里最后突然出现了一个 1 ，而其他位置都没有比它小的，这条 path 就突然变的最小了。

- 自底向上的**return**顺序 **-- Post order**
- 子树计算完在当前节点汇总的计算 **-- Merge Sort**

想到这层就已经很显然了，代码过于 trivial ，时间紧张就不写了。

LCA of deepest leaf node

这题在 LCA 类问题里有，递归和迭代的都可以解，不过都是 two-pass 的。 Post-order 可以 one-pass.

白板写了一下，发现也挺 trivial ..

```
static TreeNode curLCA = null;
static int maxDepth = 0;

private static int postOrder(TreeNode root, int depth){
    if(root == null) return 0;
    if(root.left == null && root.right == null){
        if(depth > maxDepth){
            curLCA = root;
            maxDepth = depth;
        }
        return depth;
    }

    int left = postOrder(root.left, depth + 1);
    int right = postOrder(root.right, depth + 1);

    if(left == right && left >= maxDepth){
        maxDepth = Math.max(left, maxDepth);
```

```
            curLCA = root;
        }

        return Math.max(left, right);
    }

    public static void main(String[] args) {
        /*
        TreeNode nodeA = new TreeNode('A');
        TreeNode nodeB = new TreeNode('B');
        TreeNode nodeC = new TreeNode('C');
        TreeNode nodeE = new TreeNode('E');
        TreeNode nodeF = new TreeNode('F');
        TreeNode nodeH = new TreeNode('H');
        TreeNode nodeG = new TreeNode('G');
        TreeNode nodeI = new TreeNode('I');
        TreeNode nodeZ = new TreeNode('Z');

        nodeA.left = nodeB;
        nodeA.right = nodeC;
        nodeB.left = nodeE;
        nodeB.right = nodeF;
        nodeF.left = nodeG;
        nodeF.right = nodeI;
        nodeC.right = nodeH;
        nodeH.right = nodeZ;
        */

        TreeNode nodeA = new TreeNode('A');
        TreeNode nodeB = new TreeNode('B');
        TreeNode nodeC = new TreeNode('C');
        TreeNode nodeD = new TreeNode('D');
        TreeNode nodeE = new TreeNode('E');

        nodeA.left = nodeB;
        nodeA.right = nodeC;
        nodeC.left = nodeD;
        nodeC.right = nodeE;

        postOrder(nodeA, 0);
```

```
        System.out.println(curLCA.chr);
    }
```

Find largest subtree

见本章后面"子树结构"里面的原题。

5/30 Tree, Min/Max/Balanced Depth

- 一个树 **depth** 的定义是，最深处 **node** 的 **depth**，所以要取 **max.**

Maximum Depth of Binary Tree

```
public class Solution {
    public int maxDepth(TreeNode root) {
        if(root == null) return 0;

        return Math.max(maxDepth(root.left), maxDepth(root.right
)) + 1;
    }
}
```

Minimum Depth of Binary Tree

一个意思，不过注意拐弯处如果缺了一个 child 并不代表是 valid path，因为这个节点可能不是 leaf node.

```
public class Solution {
    public int minDepth(TreeNode root) {
        if(root == null) return 0;

        if(root.left == null) return minDepth(root.right) + 1;
        if(root.right == null) return minDepth(root.left) + 1;

        return Math.min(minDepth(root.left), minDepth(root.right
)) + 1;
    }
}
```

Balanced Binary Tree

已经不平衡的地方就直接 return -1 ，避免去进一步做不必要的递归。

```
public class Solution {
    public boolean isBalanced(TreeNode root) {
        return (getDepth(root) != -1);
    }

    private int getDepth(TreeNode root){
        if(root == null) return 0;

        int left = getDepth(root.left);
        int right = getDepth(root.right);

        if(left == -1 || right == -1) return -1;
        if(Math.abs(left - right) > 1) return -1;

        return Math.max(left, right) + 1;
    }
}
```

5/31 Tree, BST

- 递归时，要考虑状态可能的不连续性。
- 取左右子树的 **min/max** 之后加上当前节点是最常见的递归结构，保证了 **path** 自顶向下的连续。
- **BST** 做 **in order traversal** 得到的结果是 **sorted.**
- **BST** 中 **root** 的左子树所有点小于 **root;** 右子树所有点大于 **root.**
- 一个正确的 **BST** 中每一个点都有其合理的取值闭区间，为【左子树 **max ,** 右子树 **min**】，最左右两端的点为一端开放区间。
- 维护大小为 **k** 的 **MaxHeap:**

```
PriorityQueue<Integer> queue = new PriorityQueue<>(k, Collection
s.reverseOrder());
```

Tree 类问题中，递归转迭代的常用手法，就是利用尾递归。

Closest Binary Search Tree Value

所谓“最近的点”，可能是 **parent** ，可能是 **child**，可能在左边，也可能在右边。

- 所以一要存好 **prev;**
- 二要两边都探，不能沿着一边硬走。

```
public class Solution {
    public int closestValue(TreeNode root, double target) {
        return find(root, target, null);
    }

    public int find(TreeNode root, double target, Integer prev){
        if(root == null) return -1; // Error
        if(prev == null) prev = root.val;

        prev = (Math.abs(root.val - target) < Math.abs(prev - ta
rget)) ? root.val: prev;

        if(root.left != null && target < root.val){
            return find(root.left, target, prev);
        }
        if(root.right != null && target > root.val){
            return find(root.right, target, prev);
        }

        return prev;
    }
}
```

这题论坛里还有更简洁直接的迭代写法，其实就是尾递归变迭代；

```
public int closestValue(TreeNode root, double target) {
    int ret = root.val;
    while(root != null){
        if(Math.abs(target - root.val) < Math.abs(target - ret))
{
            ret = root.val;
        }
        root = root.val > target? root.left: root.right;
    }
    return ret;
}
```

Validate Binary Search Tree

一开始写挂了两次，因为忘了另外的条件，就是右子树里面的最小值一定要大于 root，并且左子树里面的最大值一定要小于 root 才行。

- 多研究下 **subtree** 的结构和性质，目前对这个理解还不够。
- 递归时，要考虑状态可能的不连续性，这题和 **Binary Tree Maximum Path Sum** 很像。

写的比较粗糙的第一版，用 Tuple 存非连续子树信息；

```
public class Solution {
    private class Tuple{
        boolean isValid;
        TreeNode min;
        TreeNode max;
        public Tuple(boolean isValid, TreeNode min, TreeNode max)
{
            this.isValid = isValid;
            this.min = min;
            this.max = max;
        }
    }

    public boolean isValidBST(TreeNode root) {
```

```
        return helper(root).isValid;
    }

    private Tuple helper(TreeNode root){
        if(root == null) return new Tuple(true, null, null);

        Tuple left = helper(root.left);
        Tuple right = helper(root.right);

        if(!left.isValid || !right.isValid) return new Tuple(fal
se, null, null);

        if(left.max != null && root.val <= left.max.val) return
new Tuple(false, null, null);
        if(right.min != null && root.val >= right.min.val) return
 new Tuple(false, null, null);

        TreeNode min = null;
        TreeNode max = null;
        if(left.min == null){
            min = root;
        } else {
            min = (root.val < left.min.val) ? root : left.min;
        }

        if(right.max == null){
            max = root;
        } else {
            max = (root.val > right.max.val) ? root : right.max;
        }

        return new Tuple(true, min, max);
    }
}
```

顺着这个思路，更简洁清晰的写法是这样。

- 这个写法是在 **BST** 中定义“区间”，即对于一个正在考虑的 **root**，检查值是否处于合理区间内，也即 【左子树**max** ，右子树**min**】之间。

- 利用 **Integer** 是 **object** 的性质，用 **null reference** 代表开区间，避免 **node** 值为 **Integer.MIN/MAX** 的情况。

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

    public boolean isValidBST(TreeNode root,Integer min, Integer
 max) {
        if(root == null) return true;

        if(min != null && root.val <= min) return false;
        if(max != null && root.val >= max) return false;
        return isValidBST(root.left, min, root.val) && isValidBS
T(root.right, root.val, max);
    }
}
```

同样的，这题用 inorder traversal 也可以做，也不需要设全局变量。

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

Kth Smallest Element in a BST

这题考察的就是 BST 的性质: in order = sorted.

另一种应对多次查询的好办法是 **augmented binary tree;** 第一次用 **O(n)** 的时间统计记录一下每个 **node** 左子树节点个数，后面的每次查询就可以 **O(height)** 的时间搜索了。

Follow up:

What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the kthSmallest routine?

维护一个大小为 k 的 max heap，一直有 insert 的时候好办；有 delete 而且删掉的 node 又在 heap 里就只好找一下 in order successor 了。

```
PriorityQueue<Integer> queue = new PriorityQueue<>(k, Collection
s.reverseOrder());
```

```
public class Solution {
    public int kthSmallest(TreeNode root, int k) {
        Stack<TreeNode> stack = new Stack<TreeNode>();
        TreeNode cur = root;

        while(cur != null || !stack.isEmpty()){
            while(cur != null){
                stack.push(cur);
                cur = cur.left;
            }

            TreeNode node = stack.pop();
            if(--k == 0) return node.val;
            cur = node.right;
        }

        return root.val;
    }
}
```

Inorder Successor in BST

简单写法是 naive 的 in order traversal. 只要是 binary tree 都可以用。

```
public class Solution {
    public TreeNode inorderSuccessor(TreeNode root, TreeNode p)
{
        Stack<TreeNode> stack = new Stack<TreeNode>();
        TreeNode cur = root;
        boolean reachedP = false;

        while(cur != null || !stack.isEmpty()){
            while(cur != null){
                stack.push(cur);
                cur = cur.left;
            }
            TreeNode node = stack.pop();
            if(reachedP) return node;
            if(node == p) reachedP = true;

            cur = node.right;
        }
        return null;
    }
}
```

- 不过这题还可以进一步利用 **BST** 的性质，不依赖 **stack**，只依靠值去模拟 **inorder** 的过程。

```
public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {
    TreeNode res = null;
    while(root!=null) {
        if(root.val > p.val) {
            res = root;
            root = root.left;
        }
        else root = root.right;
    }
    return res;
}
```

更像 in order 的写法是这样：

```
public class Solution {
    public TreeNode inorderSuccessor(TreeNode root, TreeNode p)
{
        TreeNode rst = null;

        while(root != null){
            while(root != null && root.val > p.val){
                rst = root;
                root = root.left;
            }
            if(root != null) root = root.right;
        }

        return rst;
    }
}
```

Convert Sorted Array to Binary Search Tree

利用 BST inorder = sorted 的性质，一道很有意思的递归题。

在 BST 里多用区间的思想思考问题。

```
public class Solution {
    public TreeNode sortedArrayToBST(int[] nums) {
        return helper(nums, 0, nums.length - 1);
    }

    private TreeNode helper(int[] nums, int start, int end){
        if(start > end) return null;
        if(start == end) return new TreeNode(nums[start]);

        int mid = start + (end - start) / 2;

        TreeNode root = new TreeNode(nums[mid]);
        root.left = helper(nums, start, mid - 1);
        root.right = helper(nums, mid + 1, end);

        return root;
    }
}
```

Verify Preorder Sequence in Binary Search Tree

- 只给定一个 **preorder** 顺序是无法生成唯一 **binary tree** 的，也可以生成多个 **valid BST.**

所以这题的题意需要澄清下，正确意思是，给定一个 preorder sequence，只要这个 sequence 可以生成一个 valid BST，都返回 true.

这样我们的判断条件变成了这样，给定 array 如 5(123)(678)，第一个元素一定为 root，后面的比 root 小的连续序列为左子树，比 root 大的连续序列为右子树。

左右子树可以为空，但是不能违反 BST 子树性质。所以如果 > root 的连续数组后面又出现了 < root 的元素，一定是 false.

这题的写法上有点像 quicksort，要注意 index.

- 为了省心起见，这类 **subarray** 的 **divide & conquer** 最好把 **length <= 2** 的直接返回了，不然会多出许多 **corner case** 要考虑。
- O(nlogn) time and O(1) space

```
public class Solution {
    public boolean verifyPreorder(int[] preorder) {
        return helper(preorder, 0, preorder.length - 1);
    }

    private boolean helper(int[] preorder, int start, int end){
        if(end - start <= 1) return true;

        int breakPoint = start + 1;
        int root = preorder[start];

        // breakPoint should stop at index of first element > ro
ot
        // if no left subtree, breakPoint stops at start;
        for(int i = start + 1; i <= end; i++){
            if(preorder[i] < root) breakPoint ++;
            else break;
        }

        for(int i = breakPoint; i <= end; i++){
            if(preorder[i] < root) return false;
        }

        return helper(preorder, start + 1, breakPoint - 1)
               && helper(preorder, breakPoint, end);
    }
}
```

Recover Binary Search Tree

- **Java** 里一切都是 **pass by value**，当传入的是 **object reference** 的时候，实际传入的就是 **object** 的内存地址。

- 因此，带泛型的各种数据结构，**Stack, List** 等，即使里面放的都是 **TreeNode** 并且进行了各种 **add/remove/pop** 操作，对这些 **object** 的修改也会反映到原来的 **Tree** 上。

Hard 题，因为得 O(1) space. 这意味着做个 in order 存在 array 里面再扫的做法行不通，连 stack 也不能用了。。

先假设下我们有 inorder array，如何找那对 swapped pair ?

在中间 1234567 -> 1264537

在两端 1234567 -> 7234561

右端 1234567 -> 1237564

左端 1234567 -> 5234167

- 从左往右找递增序列里面第一个变小的，**prev** 为 **swapped;**
- 从右往左找递减序列里面第一个变大的，**prev** 为 **swapped.**
- 找错误元素可以简化成一次遍历，第一次找到违章元素，**prev** 是 **swapped;** 然后继续扫，第二次看到违章元素，**cur** 是 **swapped.**

所以顺着这个思路的第一种暴力写法是建一个 arraylist 存 inorder traversal，然后扫两遍去找到底应该 swap 哪两个。然而 TLE 了，test case 故意坑人，教育我们不要一言不合就遍历。。。

```
public class Solution {
    public void recoverTree(TreeNode root) {
        List<TreeNode> list = new ArrayList<TreeNode>();
        Stack<TreeNode> stack = new Stack<TreeNode>();
        TreeNode cur = root;

        while(cur != null || !stack.isEmpty()){
            while(cur != null){
                stack.push(cur);
                cur = cur.left;
            }
```

```
            TreeNode node = stack.pop();
            list.add(node);
            cur = node.right;
        }

        if(list.size() == 2){
            swap(list.get(0), list.get(1));
            return;
        }

        TreeNode p = null;
        TreeNode q = null;

        for(int i = 1; i < list.size(); i++){
            if(list.get(i).val < list.get(i - 1).val){
                p = list.get(i - 1);
                break;
            }
        }

        for(int i = list.size() - 2; i >= 0; i--){
            if(list.get(i + 1).val > list.get(i).val){
                q = list.get(i + 1);
                break;
            }
        }

        swap(p, q);

        return;
    }

    private void swap(TreeNode p, TreeNode q){
        if(p == null || q == null) return;
        int temp = p.val;
        p.val = q.val;
        q.val = temp;
    }
}
```

于是改良版的写法是，先搞一个从左往右的 inorder，然后找第一个违章元素 on the fly.

然后做一个从右往左的 inorder，找第一个违章元素 on the fly.

这样不需要遍历整个 list，可以利用 early termination.

于是。。。卧槽，居然 AC 了？

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public void recoverTree(TreeNode root) {

        Stack<TreeNode> stack = new Stack<TreeNode>();
        TreeNode cur = root;
        TreeNode p = null;

        while(cur != null || !stack.isEmpty()){
            while(cur != null){
                stack.push(cur);
                cur = cur.left;
            }
            TreeNode node = stack.pop();
            if(p == null){
                p = node;
            } else {
                if(node.val < p.val){
                    break;
                } else {
                    p = node;
                }
            }
            cur = node.right;
```

```
        }

        stack.clear();
        cur = root;
        TreeNode q = null;

        while(cur != null || !stack.isEmpty()){
            while(cur != null){
                stack.push(cur);
                cur = cur.right;
            }
            TreeNode node = stack.pop();
            if(q == null){
                q = node;
            } else {
                if(node.val > q.val){
                    break;
                } else {
                    q = node;
                }
            }
            cur = node.left;
        }

        swap(p, q);

        return;
    }

    private void swap(TreeNode p, TreeNode q){
        if(p == null || q == null) return;
        int temp = p.val;
        p.val = q.val;
        q.val = temp;
    }
}
```

遍历和迭代更简洁的写法在论坛这个帖子里写的非常好。

找错误元素可以简化成一次遍历，第一次找到违章元素，prev 是 swapped; 然后继续扫，第二次看到违章元素，cur 是 swapped.

递归版：

```
public void recoverTree(TreeNode root) {
    //use inorder traversal to detect incorrect node

    inOrder(root);

    int temp = first.val;
    first.val = second.val;
    second.val = temp;
}

TreeNode prev = null;
TreeNode first = null;
TreeNode second = null;

public void inOrder(TreeNode root){
    if(root == null) return;
    //search left tree
    inOrder(root.left);

    //in inorder traversal of BST, prev should always have small
er value than current value
    if(prev != null && prev.val >= root.val){
        //incorrect smaller node is always found as prev node
        if(first == null) first = prev;
      //incorrect larger node is always found as curr(root) node
        second = root;
    }

    //update prev node
    prev = root;

    //search right tree
    inOrder(root.right);
}
```

迭代版：

```
public class Solution {
```

```
    public void recoverTree(TreeNode root) {

        Stack<TreeNode> stack = new Stack<TreeNode>();
        TreeNode cur = root;
        TreeNode prev = null;
        TreeNode p = null;
        TreeNode q = null;

        while(cur != null || !stack.isEmpty()){
            while(cur != null){
                stack.push(cur);
                cur = cur.left;
            }
            TreeNode node = stack.pop();

            if(prev != null && node.val <= prev.val){
                if(p == null) p = prev;
                q = node;
            }

            /* 上面这里如果写成这样，会在 [0,1] 的 case 上挂掉
               原因是只有两个 node 的情况下 q 还没来得及被赋值
               就结束了，于是 q == null 会导致无法 swap.
               拿掉那个 else 可以保证不管怎样 q 指向 cur node.
                if(p == null){
                    p = prev;
                } else {
                    q = node;
                }
            */
            prev = node;
            cur = node.right;
        }

        swap(p, q);
    }

    private void swap(TreeNode p, TreeNode q){
        if(p == null || q == null) return;
        int temp = p.val;
```

```
        p.val = q.val;
        q.val = temp;
    }
}
```

6/1 Tree，子树结构

- 需要检查子树结构的题都需要一个 **helper** 函数，带着两个 **root**，递归解决。 如果默认没给，就自己写一个。
- 子树类问题如果出现不连续，或者需要多个子树信息的时候，自定义 **SubtreeTuple** 是最合适的选择。
  - **subtree size (int);**
  - 在 **size / count** 这种非负情况下，还可以把符号当 **flag** 用;
  - **subtree min/max (int);**
- 这种递归结构中先处理完 **left / right** 再来汇总结果的，其实就是 **post-order traversal.** 这点在 **search** 类 **dfs** 中也很常见，比如安卓解锁，数解锁方式数量的做法。

Tree 类问题另一个递归转迭代的思路就是，观察下递归是 **pre-order, in-order** 还是 **post-order**，然后对应的靠 **stack** 保存状态，模拟整个过程即可。

Same Tree

Trivial problem.

```
public class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {
        if(p == q) return true;
        if(p == null || q == null) return false;
        if(p.val != q.val) return false;

        return isSameTree(p.left, q.left) && isSameTree(p.right,
 q.right);
    }
}
```

迭代写法，思路很简单，就是按照同一个顺序做 **DFS (pre-order)**，每步上检查下元素值和 **stack** 大小就行，如果两个树一样，那么在迭代过程中所有的状态也都应该是一样的。

这个写法的子树访问路线以及顺序，和递归的写法是完全一样的。也就是说，其实这个迭代写法的思路，是完全建立在递归写法的代码上：

- 递归中先对两个 **root** 判断 **(**中**)**
- 然后递归处理两棵子树：
- 先左，后右；
- 这就是 **pre-order** 嘛。

```
    public boolean isSameTree(TreeNode p, TreeNode q) {
        Stack<TreeNode> stack1 = new Stack<>();
        Stack<TreeNode> stack2 = new Stack<>();

        if(p != null) stack1.push(p);
        if(q != null) stack2.push(q);

        while(!stack1.isEmpty() && !stack2.isEmpty()){
            TreeNode node1 = stack1.pop();
            TreeNode node2 = stack2.pop();
            // Check cur
            if(node1.val != node2.val) return false;
            // Add Next
            if(node1.right != null) stack1.push(node1.right);
            if(node2.right != null) stack2.push(node2.right);

            if(stack1.size() != stack2.size()) return false;

            if(node1.left != null) stack1.push(node1.left);
            if(node2.left != null) stack2.push(node2.left);

            if(stack1.size() != stack2.size()) return false;
        }


        return stack1.size() == stack2.size();
    }
```

Symmetric Tree

这题和上一题非常像，都是给你两个 root，去判断他们的结构，考虑到要做 symmetric tree 所以每次递归的时候参数是 p.left 和 q.right ，而不是每次都同一方向。

```
public class Solution {
    public boolean isSymmetric(TreeNode root) {
        if(root == null) return true;

        return isSymmetric(root.left, root.right);
    }

    private boolean isSymmetric(TreeNode p, TreeNode q){
        if(p == q) return true;
        if(p == null || q == null) return false;
        if(p.val != q.val) return false;

        return isSymmetric(p.left, q.right) && isSymmetric(p.rig
ht, q.left);
    }
}
```

- **Bonus :** 用迭代。
- 这题用迭代的写法和思路，思路像 **google onsite** 面经里出现过的，交替输出 **kth level** 的节点。 **Queue(deque)** 的写法很好写，空间优化上就需要用 **push** 顺序相反的两个 **stack** 了。

具体思路参考了下论坛，我当初的写法是用两个 queue 存两个 level，对于每个 level 用类似 two pointer 的方式去检查元素，不过因为 queue 的大小一直变动，而且 queue 的 implementation 直接 access by index 也不太方便，写起来很麻烦。

比较好的思路是根据题意，直接把每个 level 拆成两个 queue，像 segment tree 似的，一个 queue 对应左子树，一个 queue 对应右子树，插入的时候，如果 q1 放左节点，q2 就放右节点，vice versa. 如果结果正确的话，两个 queue 里面的元素完全一样。

```
public class Solution {
    public boolean isSymmetric(TreeNode root) {
        if(root == null) return true;

        Queue<TreeNode> q1 = new LinkedList<TreeNode>();
        Queue<TreeNode> q2 = new LinkedList<TreeNode>();

        evalAndAdd(root, root, q1, q2);

        while(!q1.isEmpty()){
            int size = q1.size();
            for(int i = 0; i < size; i++){
                TreeNode n1 = q1.poll();
                TreeNode n2 = q2.poll();

                if(!evalAndAdd(n1.left, n2.right, q1, q2)) return
 false;
                if(!evalAndAdd(n1.right, n2.left, q1, q2)) return
 false;
            }
        }
        return true;
    }

    private boolean evalAndAdd(TreeNode n1, TreeNode n2, Queue<T
reeNode> q1, Queue<TreeNode> q2){
        if(n1 == null && n2 == null) return true;
        if(n1 == null || n2 == null) return false;

        if(n1.val == n2.val){
            q1.offer(n1);
            q2.offer(n2);
            return true;
        }
        return false;
    }
}
```

同样一题，用 **stack** 省空间的做法，其实和两个 **queue** 的思路基本完全一样。。

两种做法都是把树 **traverse** 了一遍，只不过顺序不同。有的时候我觉得，各种 **tree** 的 **traversal**，其实就像花式 **for loop** 一个 **array** 一样。

这题的双 **stack** 代码和 **Same Tree** 基本一样，只是 **push** 顺序不同而已。其代码的相似与不同可以追溯到各自的递归解法中，**Tree** 类问题递归结构是迭代写法的指引。

```
    public boolean isSymmetric(TreeNode root) {
        if(root == null) return true;

        Stack<TreeNode> stack1 = new Stack<>();
        Stack<TreeNode> stack2 = new Stack<>();

        stack1.push(root);
        stack2.push(root);

        while(!stack1.isEmpty() && !stack2.isEmpty()){
            TreeNode node1 = stack1.pop();
            TreeNode node2 = stack2.pop();

            if(node1.val != node2.val) return false;

            if(node1.left != null)  stack1.push(node1.left);
            if(node2.right != null) stack2.push(node2.right);

            if(stack1.size() != stack2.size()) return false;

            if(node1.right != null) stack1.push(node1.right);
            if(node2.left != null)  stack2.push(node2.left);

            if(stack1.size() != stack2.size()) return false;
        }

        return stack1.size() == stack2.size();
    }
```

Largest BST Subtree

这题和 Binary Tree Maximum Path Sum 的联系非常密切，要一起研究。

这题因为执着于用一个 helper 函数同时做 “验证BST” 和 “数子树大小”的工作，做了很多次失败提交。

- 需要传递的信息太多就自定义 **SubtreeTuple**

同时这题的定义也稍微有点模糊，正确定义是：如果整棵树都是 BST，那么返回 tree size; 反之返回左右子树的最大 size ，而不考虑 root. 这个 “不考虑root” 稍微有点歧义，因为如果右子树不是 BST，左子树是 BST，并且 root.val 大于左子树的情况下，按理讲算上 root 也是一个 BST 的，只是这题我们不考虑而已。

假如我们只用一个返回 int 的函数来层层递归，需要处理这些问题：

- 正解的的子树很可能和 root 以及上层的 node 不是连续的;
- 如果某个子树不是 BST (size = 0)，也意味着上层的所有 node 都不能包含这个子树;
- 给定 root，要验证这个 root 是否在合理的左右子树极值区间内；

这些问题都不是一个 int 就能完美解决的。

时间复杂度 O(n log n)，一次检查 BST/getSize 为O(n)，最多重复调用 O(log n) 次

```
public class Solution {
    public int largestBSTSubtree(TreeNode root) {
        if(root == null) return 0;
        if(isBST(root, null, null)) return getSize(root);

        return Math.max(largestBSTSubtree(root.left), largestBST
Subtree(root.right));
    }

    private boolean isBST(TreeNode root, Integer min, Integer ma
x){
        if(root == null) return true;
        if(min != null && root.val <= min) return false;
        if(max != null && root.val >= max) return false;

        return isBST(root.left, min, root.val) && isBST(root.rig
ht, root.val, max) ;
    }

    private int getSize(TreeNode root){
        if(root == null) return 0;

        return getSize(root.left) + getSize(root.right) + 1;
    }
}
```

自定义 **SubtreeTuple** 的写法：

- 自底向上连续传递的只有 **size**，代表这个 **tree** 下面最大的 **BST subtree size.**
- **size** 的绝对值代表以这个 **node** 为 **tree root** 的最大 **BST subtree** 大小；
- **size** 的符号代表到底是不是 **BST.**

- 当我们有一个一定非负的变量时(在这里是 **size**)，符号就成了 **boolean** 一样的可利用信息。

时间复杂度 **O(n)**，每个 **node** 只访问一次，没有重复递归调用。

```
public class Solution {
    private class SubtreeTuple{
        // size of tree, negative value to reprensent invalid BST

        int size;
        // subtree min / max value
        int min;
        int max;
        public SubtreeTuple(int size, int min, int max){
            this.size = size;
            this.min = min;
            this.max = max;
        }
    }

    public int largestBSTSubtree(TreeNode root) {
        return Math.abs(helper(root).size);
    }

    private SubtreeTuple helper(TreeNode root){
        if(root == null) return new SubtreeTuple(0, Integer.MAX_
VALUE, Integer.MIN_VALUE);

        SubtreeTuple left = helper(root.left);
        SubtreeTuple right = helper(root.right);

        if(left.size < 0 || root.val <= left.max || right.size <
0 || root.val >= right.min){
            return new SubtreeTuple(Math.max(Math.abs(left.size)
, Math.abs(right.size)) * -1,
                                    Math.min(left.min, root.val)
, Math.max(right.max, root.val));
        } else {
            // current left + right + root is a valid BST
```

```
            return new SubtreeTuple(left.size + right.size + 1,
                                    Math.min(root.val, left.min)
,
                                    Math.max(root.val, right.max
));
        }
    }
}
```

Count Univalue Subtrees

对于 **subtree** 特征以及 **flag** 的处理上，和上一道题可以用完全一样的套路。

唯一的不同在于，我们要返回的是总数，而不是 **max** ，所以要左右加一起才行。

```
public class Solution {
    private class Tuple{
        // ABS : max count
        // Sign: +/- , is/not univalue subtree
        //
        int count;
        Integer val;
        public Tuple(int count, Integer val){
            this.count = count;
            this.val = val;
        }
    }
    public int countUnivalSubtrees(TreeNode root) {
        return Math.abs(helper(root).count);
    }

    private Tuple helper(TreeNode root){
        if(root == null) return new Tuple(0, null);

        Tuple left = helper(root.left);
        Tuple right = helper(root.right);

        if(left.count < 0 || right.count < 0 ||
          (left.val != null && !left.val.equals(root.val)) ||
          (right.val != null && !right.val.equals(root.val))){
            return new Tuple((Math.abs(left.count) + Math.abs(ri
ght.count)) * -1, 0);
        } else {
            return new Tuple(left.count + right.count + 1, root.
val);
        }
    }
}
```

Count Complete Tree Nodes

- 如果一个 **Tree** 是 **complete tree**，那所有的 **subtree** 也都是 **complete tree.**

直接扫肯定 TLE .. 要利用好 complete tree 的定义和性质。

参考了一下论坛之后，写了这个解居然也 TLE，都 O(log n * log n) 了

```
public class Solution {
    public int countNodes(TreeNode root) {
        if(root == null) return 0;
        if(root.left == null && root.right == null) return 1;

        int leftPath = 0;
        int rightPath = 0;
        TreeNode cur = root;
        while(cur != null){
            leftPath ++;
            cur = cur.left;
        }
        cur = root;
        while(cur != null){
            rightPath ++;
            cur = cur.right;
        }

        if(leftPath == rightPath) return (2 << (leftPath - 1)) -
1;

        return countNodes(root.left) + countNodes(root.right) + 1
;
    }
}
```

能 AC 的代码是论坛上的这个:

- **如果一个 Tree 是 complete tree，那所有的 subtree 也都是 complete tree.**

1 << l 其实包含了每一层上加一（root） 的步骤。

按照这个代码的执行方式，每一次的左右子树必定有一个是 prefect tree，于是可以根据 depth 决定下一步处理哪棵。

```
public class Solution {
    public int countNodes(TreeNode root) {
        if (root == null) {
            return 0;
        }
        int l = leftHeight(root.left);
        int r = leftHeight(root.right);
        if (l == r) { // left side is full
            return countNodes(root.right) + (1<<l);
        }
        return countNodes(root.left) + (1<<r);
    }

    private int leftHeight(TreeNode node) {
        int h = 0;
        while (node != null) {
            h++;
            node = node.left;
        }
        return h;
    }
}
```

(G) 面经

http://www.1point3acres.com/bbs/forum.php?mod=viewthread&tid=197372&highlight=google

Given a binary tree, find if there are two subtrees that are the same. (i.e. the tree structures are the same; the values on all corresponding nodes are the same). You should find the largest subtree and don't use brute force.

这篇 **paper** 值得一看，关于 **tree / subtree isomorphism**

贴个 **hx** 的思路，仔细讨论和思考之后觉得靠谱。

假设identical的子树share一个group id. NULL的group id是0.其它子树的group id按照group在后序遍历中第一次出现的时间顺序决定。 这样我们可以一边后序遍历子树，一边建两个hashmap:

(root value, group id of left, group id of right) -> group id of root group id -> subtrees of this group

这样我们做一遍后序遍历同时维护层数最高、subtree数量>=2的group id就行了。

这个其实跟用整棵子树的serialization作为key做法差不多，只是对key作了压缩。

如果只是要 size 的话，一个 hashmap 就够。

按着这个思路试着写了一下，附带两个 **test case**，返回的是最大 **subtree** 的 **size.**

第一个 **comment** 掉的 **test case** 和图里的一样，返回 **4 .**

第二个是左右完全一样的 **binary tree**，中间缺一个 **leaf node**，返回 **6.**

```
public class Solution {
    private static class TreeNode{
        int val;
        TreeNode left, right;
        public  TreeNode(int val){
            this.val = val;
        }
    }

    private static class TreeTuple{
        String key;
        int id;
        int size;
        public TreeTuple(String key, int id, int size){
            this.key = key;
            this.id = id;
            this.size = size;
        }
    }

    public static int largestSubtree(TreeNode root){
        HashMap<String, Integer> keyMap = new HashMap<>();
        HashMap<Integer, List<TreeTuple>> groupMap = new HashMap
<>();
        int[] id = new int[1];
        id[0] = 1;

        postOrder(root, keyMap, groupMap, id);

        Iterator<Integer> iter = groupMap.keySet().iterator();

        int maxSize = 0;

        while(iter.hasNext()){
            int groupId = iter.next();
            List<TreeTuple> list = groupMap.get(groupId);
            if(list.size() > 1) maxSize = Math.max(maxSize, list
```

```
.get(0).size);

        }

        return maxSize;
    }

    // keyMap : Key - String - (val, leftId, rightId)
    //          Val - Integer - groupId
    // groupMap : Key - Integer - groupId
    //            Val - Integer - number of occurrances
    private static TreeTuple postOrder(TreeNode root, HashMap<St
ring, Integer> keyMap,
                                       HashMap<Integer, List<Tre
eTuple>> groupMap, int[] id){

        if(root == null) return new TreeTuple("0,0,0", 0, 0);

        TreeTuple left = postOrder(root.left, keyMap, groupMap,
id);
        TreeTuple right = postOrder(root.right, keyMap, groupMap
, id);

        int curId;
        TreeTuple curTuple;

        String key = "" + root.val + "," + left.id + "," + right
.id;
        if(!keyMap.containsKey(key)){
            curId = id[0]++;
            keyMap.put(key, curId);
            groupMap.put(curId, new ArrayList<>());
            curTuple = new TreeTuple(key, curId, left.size + rig
ht.size + 1);
            groupMap.get(curId).add(curTuple);
        } else {
            curId = keyMap.get(key);
            curTuple = new TreeTuple(key, curId, left.size + rig
ht.size + 1);
            groupMap.get(curId).add(curTuple);
```

```
        }

        return curTuple;
    }


    public static void main(String[] args) {
        /*
        TreeNode root = new TreeNode(5);
        TreeNode root2 = new TreeNode(9);
        TreeNode root3 = new TreeNode(2);
        TreeNode root4 = new TreeNode(7);
        TreeNode root5 = new TreeNode(3);
        TreeNode root6 = new TreeNode(12);
        TreeNode root7 = new TreeNode(10);
        TreeNode root8 = new TreeNode(9);
        TreeNode root9 = new TreeNode(2);
        TreeNode root10 = new TreeNode(7);
        TreeNode root11 = new TreeNode(3);

        root.left = root2;
        root.right = root6;
        root2.left = root3;
        root2.right = root4;
        root4.left = root5;
        root6.left = root7;
        root6.right = root8;
        root8.left = root9;
        root8.right = root10;
        root10.left = root11;
        */

        TreeNode root = new TreeNode(0);
        TreeNode rootl2 = new TreeNode(1);
        TreeNode rootr3 = new TreeNode(1);
        TreeNode rootl4 = new TreeNode(2);
        TreeNode rootr5 = new TreeNode(2);
        TreeNode rootl6 = new TreeNode(3);
        TreeNode rootr7 = new TreeNode(3);
        TreeNode rootl8 = new TreeNode(4);
```

```
        TreeNode rootr9 = new TreeNode(4);
        TreeNode rootl10 = new TreeNode(5);
        TreeNode rootr11 = new TreeNode(5);
        TreeNode rootl12 = new TreeNode(6);
        TreeNode rootr13 = new TreeNode(6);

        root.left = rootl2;
        root.right = rootr3;
        rootl2.left = rootl4;
        rootl2.right = rootl6;
        rootl4.left = rootl8;
        rootl4.right = rootl10;
        rootl6.right = rootl12;

        rootr3.left = rootr5;
        rootr3.right = rootr7;
        rootr5.left = rootr9;
        rootr5.right = rootr11;
        rootr7.right = rootr13;

        System.out.println(largestSubtree(root));
    }
}
```

6/2, Tree, Level Order traversal

Binary Tree Level Order Traversal

送分题，就当热手了。

迭代的写法和 CLRS 的伪代码一模一样。

```
public class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> rst = new ArrayList<List<Integer>>()
;
        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        if(root != null) queue.offer(root);
        while(!queue.isEmpty()){
            int size = queue.size();
            List<Integer> list = new ArrayList<>();
            for(int i = 0; i < size; i++){
                TreeNode node = queue.poll();
                list.add(node.val);
                if(node.left != null) queue.offer(node.left);
                if(node.right != null) queue.offer(node.right);
            }
            rst.add(list);
        }

        return rst;
    }
}
```

这题的递归写法挺有意思的。

- 递归写法本质上是对 **binary tree** 做了一个 **pre-order dfs.**
- 更靠近 **root** 的层一定会比下一层先建出来

- 对于同一个 **level**，左面的节点一定比右面节点先 **add.**

```
public class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> rst = new ArrayList<List<Integer>>()
;
        dfs(rst, root, 0);
        return rst;
    }

    private void dfs(List<List<Integer>> rst, TreeNode root, int
 level){
        if(root == null) return;

        if(level >= rst.size()){
            rst.add(new ArrayList<Integer>());
        }

        rst.get(level).add(root.val);

        dfs(rst, root.left, level + 1);
        dfs(rst, root.right, level + 1);
    }
}
```

Binary Tree Level Order Traversal II

写法一样，最后 reverse 一下就好了。

```
public class Solution {
    public List<List<Integer>> levelOrderBottom(TreeNode root) {
       List<List<Integer>> rst = new ArrayList<List<Integer>>();
        dfs(rst, root, 0);
        Collections.reverse(rst);
        return rst;
    }

    private void dfs(List<List<Integer>> rst, TreeNode root, int
 level){
        if(root == null) return;

        if(level >= rst.size()){
            rst.add(new ArrayList<Integer>());
        }

        rst.get(level).add(root.val);

        dfs(rst, root.left, level + 1);
        dfs(rst, root.right, level + 1);
    }
}
```

Binary Tree Zigzag Level Order Traversal

挺 trivial 的问题。

```
public class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> rst = new ArrayList<List<Integer>>()
;
        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        boolean reverse = false;
        if(root != null) queue.offer(root);
        while(!queue.isEmpty()){
            int size = queue.size();
            List<Integer> list = new ArrayList<Integer>();
            for(int i = 0; i < size; i++){
                TreeNode node = queue.poll();
                list.add(node.val);
                if(node.left != null) queue.offer(node.left);
                if(node.right != null) queue.offer(node.right);
            }
            if(reverse) Collections.reverse(list);

            rst.add(list);
            reverse = !reverse;
        }

        return rst;
    }
}
```

Populating Next Right Pointers in Each Node

Level order 的写法非常的 trivial，不过这题规定不许手动建任何额外空间（不许存level或者用迭代写，虽然递归也用空间）

考虑到这题给定 perfect tree，不需要考虑中间有拐弯或者空位的问题，递归解决的思路就是用一个 helper 函数传入当前的两个节点（为上一个节点的 left/right child）

- 连接左右
- 迭代依次连接左节点右侧的 path 与 右节点左侧的 path
- 递归下一层

```
public class Solution {

    public void connect(TreeLinkNode root) {
        if(root == null) return;
        if(root.left == null && root.right == null) return;

        helper(root.left, root.right);
    }

    private void helper(TreeLinkNode p, TreeLinkNode q){
        if(p == null && q == null) return;
        if(p == null || q == null) return;

        p.next = q;

        TreeLinkNode l = p;
        TreeLinkNode r = q;
        while(q.right != null && q.left != null){
            p = p.right;
            q = q.left;

            p.next = q;
        }

        helper(l.left, l.right);
        helper(r.left, r.right);
    }
}
```

眞正的 O(1) 空间是用固定数量的指针进行迭代循环。

- 这种写法是眞正的 **level order** 添加，利用了上一层完成之后的 **next** 指针手动做了 **level order** 遍历。

```
public class Solution {
    public void connect(TreeLinkNode root) {
        if(root == null) return;

        TreeLinkNode cur = root;
        TreeLinkNode leftMost = null;
        while(cur.left != null){
            leftMost = cur.left;
            while(cur != null){
                cur.left.next = cur.right;
                if(cur.next == null){
                    cur.right.next = null;
                } else {
                    cur.right.next = cur.next.left;
                }
                cur = cur.next;
            }
            cur = leftMost;
        }
    }
}
```

- 在参考了下一题 **follow up** 的写法之后，这类问题的通解：

```
public class Solution {
    public void connect(TreeLinkNode root) {
        while(root != null){
            TreeLinkNode dummy = new TreeLinkNode(0);
            TreeLinkNode cur = dummy;
            for( ; root != null; root = root.next){
                if(root.left != null){
                    cur.next = root.left;
                    cur = cur.next;
                }
                if(root.right != null){
                    cur.next = root.right;
                    cur = cur.next;
                }
            }
            root = dummy.next;
        }
    }
}
```

Populating Next Right Pointers in Each Node II

搞了半天一直处于一个追着各种 test case 改 Bug 的阶段，而且要改的细节越来越多。。。写题如果发现自己陷入了这个阶段，还是果断换思路把。

这题的迭代思路和上一题一样，充分利用好上一层已经是连好的next指针来做 level order. 然而有几个很烦的地方需要处理:

- 中间有空缺节点的话，怎么让上层找到“下一个”正确的节点
- 如果某一层最左边有空缺的话，如何在进入下一层的时候找到正确的起点

这个解法的思路非常简洁优美。因为对题本质的理解更进一步了：

- 我们要做的其实是 **level order** 建一个 **LinkedList** 出来。

于是这题几个最烦人的细节处理，都可以靠 dummy node 解决。

- 这题也是继 **word ladder** 之后另一个不同的 **for** 循环写法，字符和链表都很适合用 **for loop** 实现遍历。

```
public class Solution {
    public void connect(TreeLinkNode root) {
        while(root != null){
            TreeLinkNode dummy = new TreeLinkNode(0);
            TreeLinkNode cur = dummy;
            for( ; root != null; root = root.next){
                if(root.left != null){
                    cur.next = root.left;
                    cur = cur.next;
                }
                if(root.right != null){
                    cur.next = root.right;
                    cur = cur.next;
                }
            }
            root = dummy.next;
        }
    }
}
```

Binary Tree Right Side View

这题本质上还是在做 level order 遍历，因为最终结果上 node 是按层数依次添加的。

于是回想下递归版的 level order traversal，我们做 dfs pre-order

- 【中，左，右】的顺序可以保证上面一层的节点永远会比下一层的先建立；
- 同时对于每一层，我们一定会先发现最左面的节点，而后才是右面依次发现。

把这个性质根据题意改一下，我们就有了

- 【中，右，左】首先保证了 **level order**，因为对于所有子树，中节点要比左右子树先执行；

- 同时因为子树顺序是【右，左】，同一 **level** 的节点一定是从右到左被发现的。

因此只要注意判断条件，只把第一次发现的元素添加进去，然后做改变顺序版 preorder traversal 就好了。

- 同理也很容易解 **Left Side View.**

```
public class Solution {
    public List<Integer> rightSideView(TreeNode root) {
        List<Integer> list = new ArrayList<Integer>();

        helper(root, list, 0);

        return list;
    }

    private void helper(TreeNode root, List<Integer> list, int l
evel){
        if(root == null) return;

        if(level == list.size()) list.add(root.val);
        helper(root.right, list, level + 1);
        helper(root.left, list, level + 1);
    }
}
```

6/3, Tree, Morris 遍历

这个帖子 的描述和代码非常好，还有这个。

我一开始以为 Morris 遍历会改变原树的结构所以不能用，后来发现并不会。。。于是这种技巧还是很值得掌握的。

1968年，Knuth提出说能否将该问题的空间复杂度压缩到O(1)，同时原树的结构不能改变。大约十年后，1979年，Morris在《Traversing Binary Trees Simply and Cheaply》这篇论文中用一种Threaded Binary Tree的方法解决了该问题。

每次访问root左子树之前，先找到左子树里面最右面的点，并把其 right 指针连到 root 上；左子树遍历完这个点之后，再把这个多出来的指针拆掉。

Morris in-order 流程，利用 threaded binary tree.

```
public class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> list = new ArrayList<Integer>();
        TreeNode cur = root;
        while(cur != null){
            if(cur.left == null){
                list.add(cur.val);
                cur = cur.right;
            } else {
                TreeNode prev = cur.left;
                while(prev.right != null && prev.right != cur){
                    prev = prev.right;
                }
                if(prev.right == null){
                    prev.right = cur;
                    // Uncomment for pre-order
                    // list.add(cur.val);
                    cur = cur.left;
                } else {
                    prev.right = null;
                    // Uncomment for in-order
                    // list.add(cur.val);
                    cur = cur.right;
                }
            }
        }
        return list;
    }
}
```

于是这道要求用 **O(n)** 时间 **O(1)** 空间的题就可以真正按照题目要求解决了。

Recover Binary Search Tree

```
public class Solution {
```

```
    public void recoverTree(TreeNode root) {
        TreeNode cur = root;
        TreeNode prevNode = null;
        TreeNode p = null;
        TreeNode q = null;

        while(cur != null){
            if(cur.left == null){
                if(prevNode != null && prevNode.val >= cur.val){
                    if(p == null) p = prevNode;
                    q = cur;
                }
                // Set prev node for scanning
                prevNode = cur;
                cur = cur.right;
            } else {
                TreeNode prev = cur.left;
                while(prev.right != null && prev.right != cur){
                    prev = prev.right;
                }
                if(prev.right == null){
                    prev.right = cur;
                    cur = cur.left;
                } else {
                    prev.right = null;

                    if(prevNode != null && prevNode.val >= cur.v
al){
                        if(p == null) p = prevNode;
                        q = cur;
                    }
                    // Set prev node for scanning
                    prevNode = cur;
                    cur = cur.right;
                }
            }
        }

        swap(p, q);
    }
```

```
    private void swap(TreeNode p, TreeNode q){
        if(p == null || q == null) return;
        int temp = p.val;
        p.val = q.val;
        q.val = temp;
    }
}
```

Morris 的 post-order 遍历还要建一个 dummy node 以及反序输出。。感觉不是非常现实。。。有空复习的时候我再研究研究这种 trick 吧。

6/3, Tree，修改结构

Flatten Binary Tree to Linked List

很符合递归思想的写法，美中不足是每次都要把左子树遍历一遍好去找尾端节点，如果整个树左子树体积非常大而右子树很小的话，时间复杂度会很高。最差的情况是，整个树是一个只有左边的链表，时间复杂度可以达到 O(n^2)，而且用递归还要花费栈空间。

```
public class Solution {
    public void flatten(TreeNode root) {
        if(root == null) return;

        flatten(root.left);
        flatten(root.right);

        TreeNode left = root.left;
        TreeNode right = root.right;

        root.left = null;
        root.right = left;
        while(root.right != null){
            root = root.right;
        }
        root.right = right;
    }
}
```

于是这题有特别赞的O(n)时间O(1)空间写法，还是利用 Morris 遍历。

Morris 遍历的特点是寻找左子树中能沿着右边走最长的节点，并且利用这个节点做文章；这题是把 right 指针直接指向 root 的右节点了，相当于每次缩进去【root.left -> 左子树最长向右路径】这段到右子树上，如此反复。因此省去了最坏情况下重复遍历寻找链表尾的过程。

```
public class Solution {
    public void flatten(TreeNode root) {
        TreeNode cur = root;
        while(cur != null){
            if(cur.left == null){
                cur = cur.right;
            } else {
                TreeNode prev = cur.left;
                while(prev.right != null){
                    prev = prev.right;
                }
                prev.right = cur.right;
                cur.right = cur.left;
                cur.left = null;
            }
        }
    }
}
```

Reverse LinkedList

链表经典水题，把它放在这是因为它和下一题真的很像，只是维度上更单一而已。

```
public class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode prev = null;
        while(head != null){
            ListNode temp = head.next;
            head.next = prev;
            prev = head;
            head = temp;
        }
        return prev;
    }
}
```

Binary Tree Upside Down

这题与其说是 Tree 类题目，不如说更像 LinkedList... 因为有很多的存 temp 和改动 reference ptr 的过程，尤其是迭代版，活脱一个反转列表。

- 迭代版

```
public TreeNode upsideDownBinaryTree(TreeNode root) {
    TreeNode curr = root;
    TreeNode temp = null;
    TreeNode prev = null;

    while(curr != null) {
        TreeNode next = curr.left;
        curr.left = temp;
        temp = curr.right;
        curr.right = prev;

        prev = curr;
        curr = next;
    }
    return prev;
}
```

- 递归版

```
public TreeNode upsideDownBinaryTree(TreeNode root) {
    if(root == null || root.left == null) {
        return root;
    }

    TreeNode newRoot = upsideDownBinaryTree(root.left);
    root.left.left = root.right;   // node 2 left children
    root.left.right = root;         // node 2 right children
    root.left = null;
    root.right = null;
    return newRoot;
}
```

(FB) BST to doubly linked-list

LC 论坛的讨论帖 http://articles.leetcode.com/convert-binary-search-tree-bst-to/

递归

思路 ： **http://www.geeksforgeeks.org/convert-given-binary-tree-doubly-linked-list-set-3/**

- 完全就是 **in - order** 的递归结构，左-中-右
- 核心在于 “中” 这步上，如何正确做好 “拼接” 工作
- 我们需要存一个全局变量 **prev** 用于保存 **"**左子树的最后一个节点**"**，在每步上，和 **root** 做双向拼接； **prev** 初始化为 **null;**
- 额外用于遍历 **LinkedList** 还需要存下 **head ;** 在 **prev** 为 **null** 的时候 **root** 就代表着最左面的节点，设一下就好，之后就不用管了。

时间复杂度 **O(n).**

```
private static class TreeNode{
    int val;
    TreeNode left,right;
    public TreeNode(int val){
        this.val = val;
    }
}

static TreeNode prev;
static TreeNode head;

// In-order
public static void convert(TreeNode root){
    if(root == null) return;

    convert(root.left);
```

```
        if(prev == null){
            head = root;
        } else {
            root.left = prev;
            prev.right = root;
        }
        prev = root;

        convert(root.right);

    }

    public static void main(String[] args){

        TreeNode node1 = new TreeNode(1);
        TreeNode node2 = new TreeNode(2);
        TreeNode node3 = new TreeNode(3);
        TreeNode node4 = new TreeNode(4);
        TreeNode node5 = new TreeNode(5);
        TreeNode node6 = new TreeNode(6);
        TreeNode node7 = new TreeNode(7);
        TreeNode node8 = new TreeNode(8);
        TreeNode node9 = new TreeNode(9);
        TreeNode node10 = new TreeNode(10);

        node2.left = node1;
        node2.right = node3;
        node4.left = node2;
        node4.right = node5;
        node6.left = node4;
        node6.right = node9;
        node9.left = node8;
        node8.left = node7;
        node9.right = node10;

        convert(node6);

        while(head != null){
            System.out.print(" " + head.val);
            head = head.right;
```

```
        }
    }
```

迭代(Stack)

- **In-order** 跑一遍，每次 **pop** 出来的时候，我们就有 **root** 了；
- 然后拼接的逻辑处理和递归的方法完全一样，这次连全局变量都不用，简单直接~

时间O(n)，空间 O(log n)

```
    // In-order
    public static TreeNode convert(TreeNode root){
        if(root == null) return null;

        TreeNode prev = null;
        TreeNode head = null;

        Stack<TreeNode> stack = new Stack<>();
        TreeNode cur = root;
        while(!stack.isEmpty() || cur != null){
            while(cur != null){
                stack.push(cur);
                cur = cur.left;
            }
            TreeNode node = stack.pop();

            if(prev == null){
                head = node;
            } else {
                prev.right = node;
                node.left = prev;
            }

            prev = node;
            cur = node.right;
```

```
        }
        return head;
    }

    public static void main(String[] args){

        TreeNode node1 = new TreeNode(1);
        TreeNode node2 = new TreeNode(2);
        TreeNode node3 = new TreeNode(3);
        TreeNode node4 = new TreeNode(4);
        TreeNode node5 = new TreeNode(5);
        TreeNode node6 = new TreeNode(6);
        TreeNode node7 = new TreeNode(7);
        TreeNode node8 = new TreeNode(8);
        TreeNode node9 = new TreeNode(9);
        TreeNode node10 = new TreeNode(10);

        node2.left = node1;
        node2.right = node3;
        node4.left = node2;
        node4.right = node5;
        node6.left = node4;
        node6.right = node9;
        node9.left = node8;
        node8.left = node7;
        node9.right = node10;

        TreeNode head = convert(node6);

        while(head != null){
            System.out.print(" " + head.val);
            head = head.right;
        }
    }
```

6/7, Tree, 创建 / 序列化

- 要学会用 **continuous subarray** 的眼光去看三序遍历数组，子树结构 **=** 子序列结构
- 对于任意指定子树，在 **inorder / preorder / postorder** 中都是长度一样的连续 **subarray**，只是位置不同。

Construct Binary Tree from Preorder and Inorder Traversal

这题挺常见的，而且在树类问题里我也很喜欢，很考察对于树的理解。

```
            [1]
          /     \
       [5]       [4]
      /   \        \
    [2]   [3]       [6]
                   /
                 [7]
```

- In-order: 【(2,5,3) 1 (4,7,6)】
- Pre-order: 【1 (5,2,3) (4,6,7)】
- Post-order: 【(2,3,5) (7,6,4) 1】

In-order 和 pre-order 单独都只能提供一部分树的信息，只依靠一个无法建立出完全一样的树，因为有歧义。

> - **In-order :** 对于指定位置 **index** 的 **root**
> - 对于每个 **tree / subtree, array** 结构
> - 【左子树】【**root**】【右子树】
> - **Pre-order:**
> - 对于每个 **tree / subtree, array** 结构
> - 【**root**】【左子树】【右子树】
> - **Post-order:**
> - 对于每个 **tree / subtree, array** 结构
> - 【左子树】【右子树】【**root**】

递归建树的过程是

- 建当前 root;
- 建 左/右 子树；
- 建 右/左 子树；

因此根据 inorder 和 preorder 的性质，我们用 preorder 的顺序决定“先建哪个为 root”，用 inorder 的相对位置决定 “左右子树是谁”。

因此这个问题是关于 preorder 的遍历，对于每个元素要在 inorder 中寻找相对位置。

对于任意指定子树，在 **inorder / preorder / postorder** 中都是长度一样的连续 **subarray**，只是位置不同。

```
public class Solution {
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        return build(preorder, inorder, 0, 0, inorder.length - 1
);
    }

    private TreeNode build(int[] preorder, int[] inorder, int pr
eorderIndex,
                           int inorderStart, int inorderEnd){
        if(inorderStart > inorderEnd || preorderIndex >= preorde
r.length) return null;

        int rootVal = preorder[preorderIndex];
        int pos = inorderStart;
        for(int i = inorderStart; i <= inorderEnd; i++){
            if(inorder[i] == rootVal){
                pos = i;
                break;
            }
        }

        int leftSubTreeLength = pos - inorderStart;

        TreeNode root = new TreeNode(rootVal);
        root.left = build(preorder, inorder, preorderIndex + 1,
inorderStart, pos - 1);
        root.right = build(preorder, inorder, preorderIndex + le
ftSubTreeLength + 1, pos + 1, inorderEnd);

        return root;
    }
}
```

Construct Binary Tree from Inorder and Postorder Traversal

掌握了这个性质之后，把 preorder 换成 postorder 也是一样的配方，一样的味道。

```
public class Solution {
    public TreeNode buildTree(int[] inorder, int[] postorder) {
        return build(inorder, postorder, postorder.length - 1, 0
, inorder.length - 1);
    }

    private TreeNode build(int[] inorder, int[] postorder, int i
ndex, int inorderStart, int inorderEnd){
        if(inorderStart > inorderEnd || index < 0) return null;

        int rootVal = postorder[index];
        int pos = inorderStart;
        for(int i = inorderStart; i <= inorderEnd; i++){
            if(inorder[i] == rootVal){
                pos = i;
                break;
            }
        }

        int rightSubtreeLength = inorderEnd - pos;

        TreeNode root = new TreeNode(rootVal);
        root.left = build(inorder, postorder, index - rightSubtr
eeLength - 1, inorderStart, pos - 1);
        root.right = build(inorder, postorder, index - 1, pos + 1
, inorderEnd);

        return root;
    }
}
```

Serialize and Deserialize Binary Tree

这也是个很有意思的问题，核心问题类似于 encode / decode strings，在于 “如何确定唯一的 binary tree ?” 对于 List of String 来讲，只需要正确做好单词的分隔还有分隔符的消除歧义；而 Tree 上可以有 level 的分隔，left / right subtree 的分隔。

相对来讲Tree的优势是，这个 TreeNode 里面只存数字，所以有很多额外的字母和符号可以利用，不用担心所用字符的歧义问题（不然就得定义 escape 符了）

http://www.geeksforgeeks.org/serialize-deserialize-binary-tree/

- **If given Tree is Binary Search Tree?**

  If the given Binary Tree is Binary Search Tree, we can store it by either storing preorder or postorder traversal. In case of Binary Search Trees, only preorder or postorder traversal is sufficient to store structure information.

- **If given Binary Tree is Complete Tree?**

  A Binary Tree is complete if all levels are completely filled except possibly the last level and all nodes of last level are as left as possible (Binary Heaps are complete Binary Tree). For a complete Binary Tree, level order traversal is sufficient to store the tree. We know that the first node is root, next two nodes are nodes of next level, next four nodes are nodes of 2nd level and so on.

- **If given Binary Tree is Full Tree?**

  A full Binary is a Binary Tree where every node has either 0 or 2 children. It is easy to serialize such trees as every internal node has 2 children. We can simply store preorder traversal and store a bit with every node to indicate whether the node is an internal node or a leaf node.

- **How to store a general Binary Tree?**

  A simple solution is to store both Inorder and Preorder traversals. This solution requires requires space twice the size of Binary Tree. We can save space by storing Preorder traversal and a marker for NULL pointers.

解决这题的关键就是，自己补位，构建一个 **full binary tree** 出来。

自己第一遍 AC 的代码。思想就是做 preorder ，不过把左右子树分别用括号括起来。空节点会生成空括号。这样对于一棵子树的 substring，第一个 matching 括号就代表左子树，后面的就是右子树。

这种写法的好处一是和这章之前的思想有联系和继承，第二是不需要用逗号做分隔（但是每个叶节点会生成一对空括号）

递归的时候注意抹去括号的 index 细节，并且在一开始检查下是不是 "()".

另外一个经验是，处理字符串的时候尽量直接用内置函数，比如 **str.indexOf('x')** 这种，知道题的重点不是字符串就别每次都手写了。

```
public class Codec {

    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
        if(root == null) return "";

        String left = serialize(root.left);
        String right = serialize(root.right);

        return Integer.toString(root.val) + "(" + left + ")" + "
(" + right + ")";
    }


    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        if(data == null || data.length() == 0) return null;
        if(data.equals("()")) return null;

        int indexFirstLeft = data.indexOf('(');
        int rootVal = Integer.parseInt(data.substring(0, indexFi
rstLeft));

        int leftCount = 1;
        int indexMatchingRight = indexFirstLeft + 1;

        while(indexMatchingRight < data.length() && leftCount !=
0){
            if(data.charAt(indexMatchingRight) == '(') leftCount
 ++;
            if(data.charAt(indexMatchingRight) == ')') leftCount
```

```
 --;

            indexMatchingRight ++;
        }

        String leftSubtree = data.substring(indexFirstLeft + 1,
indexMatchingRight - 1);
        String rightSubtree = data.substring(indexMatchingRight
+ 1, data.length() - 1);

        TreeNode root = new TreeNode(rootVal);
        root.left = deserialize(leftSubtree);
        root.right = deserialize(rightSubtree);

        return root;
    }
}
```

论坛上比较简洁正规的 **pre-order DFS** 递归写法：

```
    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
        StringBuilder sb = new StringBuilder();
        dfsSerialize(root, sb);
        return sb.toString().trim();
    }

    private void dfsSerialize(TreeNode root, StringBuilder sb){
        if(root == null){
            sb.append("#").append(" ");
            return;
        }

        sb.append(root.val).append(" ");
        dfsSerialize(root.left, sb);
        dfsSerialize(root.right, sb);
    }
```

```
    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        String[] strs = data.split(" ");
        int[] index = new int[1];

        return dfsDeserialize(strs, index);
    }

    private TreeNode dfsDeserialize(String[] strs, int[] index){
        if(strs[index[0]].equals("#")){
            index[0]++;
            return null;
        }

        TreeNode root = new TreeNode(Integer.parseInt(strs[index[
0]]));
        index[0]++;
        root.left = dfsDeserialize(strs, index);
        root.right = dfsDeserialize(strs, index);

        return root;
    }
```

另一种写法是 BFS，源代码是 LeetCode 论坛上的：

这个写法里会在 queue 里放入 null.

核心思想是，用 **#** 补位，构造一个 **"full binary tree"**，每个节点有 **0 / 2** 个子节点。因此在我们构造树的过程中，我们永远可以一次看两个元素，并且把子节点加入队列，保证算法的正确性。

```
public class Codec {
    public String serialize(TreeNode root) {
        if (root == null) return "";
        Queue<TreeNode> q = new LinkedList<TreeNode>();
        StringBuilder res = new StringBuilder();
        q.add(root);
        while (!q.isEmpty()) {
```

```
            TreeNode node = q.poll();
            if (node == null) {
                res.append("#,");
                continue;
            }
            res.append(node.val + ",");
            q.add(node.left);
            q.add(node.right);
        }
        return res.toString();
    }

    public TreeNode deserialize(String data) {
        if (data == "") return null;
        Queue<TreeNode> q = new LinkedList<TreeNode>();
        String[] values = data.split(",");
        TreeNode root = new TreeNode(Integer.parseInt(values[0])
);
        q.add(root);
        for (int i = 1; i < values.length; i++) {
            TreeNode parent = q.poll();
            if (!values[i].equals("#")) {
                TreeNode left = new TreeNode(Integer.parseInt(va
lues[i]));
                parent.left = left;
                q.add(left);
            }
            if (!values[++i].equals("#")) {
                TreeNode right = new TreeNode(Integer.parseInt(v
alues[i]));
                parent.right = right;
                q.add(right);
            }
        }
        return root;
    }
}
```

在 **Binary Tree** 的各种遍历中，**BFS** 都是比较耗费空间的一种，所以一个显然的优化与 **follow up** 就是，能不能用 **DFS** ，迭代做。

去论坛看了一圈，发现对于 **full binary tree**，**pre-order** 和 **in-order** 的 **DFS** 都行，挺有意思的~

这种做法其实就是一种对 **pre-order DFS** 做法的 **Stack** 模拟。也是 **Tree** 类问题递归转迭代的常用手段。

- 关键点**1**： 要存一个 **boolean** 记录下当前要填的点是不是左节点；
- 关键点**2**：这个 **boolean** 的变化要看当前 **boolean** 值以及新填上的点是不是叶节点；
- 关键点**3**：对于填充右节点的情况，链接完了就直接 **pop()**，有别于填充左子树时候用的 **peek().** 因为 **preorder** 右子树结束之后 **stack frame** 就出栈了。

```
// Encodes a tree to a single string.
public String serialize(TreeNode root) {
    Stack<TreeNode> stack = new Stack<>();
    StringBuilder sb = new StringBuilder();
    stack.push(root);

    while(!stack.isEmpty()){
        TreeNode node = stack.pop();
        if(node == null){
            sb.append("#").append(" ");
            continue;
        }

        sb.append(node.val).append(" ");
        stack.push(node.right);
        stack.push(node.left);
    }
    return sb.toString().trim();
```

```
    }

    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        Stack<TreeNode> stack = new Stack<>();
        String[] strs = data.split(" ");
        if(strs[0].equals("#")) return null;
        TreeNode root = new TreeNode(Integer.parseInt(strs[0]));
        stack.push(root);
        int index = 1;
        boolean doLeft = true;
        while(index < strs.length){
            String str = strs[index++];
            TreeNode node = (str.equals("#"))
                            ? null
                            : new TreeNode(Integer.parseInt(str)
);
            if(doLeft){
                stack.peek().left = node;
                if(node == null) doLeft = false;
            } else {
                stack.pop().right = node;
                if(node != null) doLeft = true;
            }

            if(node != null) stack.push(node);
        }
        return root;
    }
```

Verify Preorder Serialization of a Binary Tree

先贴个用自己理解写的做法：

Stack 里面存 node ，记录来路，对于每个 node，存一个 boolean 表示 “左子树处理完没”。这样当我们每次处理完一个 node 之后，都以栈顶为准，如果栈顶 boolean = false，就设成 true，代表左子树看完了，开始看右子树；否则就一路 pop 到第一个左子树没处理完的节点为止。

- **corner case1: "#"** 代表空树，合法；
- **corner case2: stack** 只能空一次，中间如果再出现 **stack** 为空然后往里加 **node** 的情况，说明是多个 **root**，返回 **false;**

```
public class Solution {
    private class Node{
        boolean isLDone;
        public Node(){
            isLDone = false;
        }
    }

    public boolean isValidSerialization(String preorder) {
        if(preorder.equals("#")) return true;

        Stack<Node> stack = new Stack<>();
        String[] strs = preorder.split(",");

        for(int i = 0; i < strs.length; i++){
            // Multiple roots
            if(i != 0 && stack.isEmpty()) return false;

            String str = strs[i];
            if(str.equals("#")){
                // Invalid leaf
                if(stack.isEmpty()) return false;
                Node top = stack.peek();
                if(!top.isLDone){
                    top.isLDone = true;
                } else {
                    while(!stack.isEmpty()){
                        stack.pop();
                        if(!stack.isEmpty() && !stack.peek().isL
Done){
                            stack.peek().isLDone = true;
                            break;
                        }
                    }
```

```
                }
            } else {
                stack.push(new Node());
            }
        }

        return stack.isEmpty();
    }
}
```

另一种妖孽的写法是 **dietpepsi** 的 **graph degree** 思路，在这个帖子

这个思路可以验 **pre-order** 与 **post-order** 的 **serialization.**

- **all non-null node provides 2 outdegree and 1 indegree (2 children and 1 parent), except root**
- **all null node provides 0 outdegree and 1 indegree (0 child and 1 parent).**

```
public boolean isValidSerialization(String preorder) {
    String[] nodes = preorder.split(",");
    int diff = 1;
    for (String node: nodes) {
        if (--diff < 0) return false;
        if (!node.equals("#")) diff += 2;
    }
    return diff == 0;
}
```

最后一个思路是，借鉴这章之前 **Serialization** 的 **pre-order** 重建法，保存 **doLeft** 的 **boolean** 变量模拟重建树的过程。当然，这里 **stack** 存的其实是一个 **stack frame**，而不再是具体 **node**，左右子树也不重要了。

```
    private class TreeNode{
```

```
        int val;
        public TreeNode(int val){
            this.val = val;
        }
    }

    public boolean isValidSerialization(String preorder) {
        if(preorder.equals("#")) return true;

        Stack<TreeNode> stack = new Stack<>();
        String[] strs = preorder.split(",");
        TreeNode root = (!strs[0].equals("#"))
                        ? new TreeNode(0)
                        : null;
        stack.push(root);
        boolean doLeft = true;

        for(int i = 1; i < strs.length; i++){
            // Multiple roots
            if(i != 0 && stack.isEmpty()) return false;
            if(stack.peek() == null) return false;

            String str = strs[i];
            TreeNode node = (str.equals("#"))
                            ? null
                            : new TreeNode(0);

            if(doLeft){
                if(node == null) doLeft = false;
            } else {
                if(stack.isEmpty()) return false;
                stack.pop();
                if(node != null) doLeft = true;
            }
            if(node != null) stack.push(node);
        }

        return stack.isEmpty();
    }
```

6/8, Tree，子树组合，BST query

Java 带泛型的 Collections 确实是可以放 null 的， 比如 Queue, List...

Unique Binary Search Trees

这题考察的是 Binary Tree 和 BST 的结构，比较容易想到的思路是画几个 base case 出来然后开始往上加新的 node，不过容易陷进去，开始去抠 detail 去开始想如何添加新点而不违反 BST 性质。

上面那个思路只对了一半，因为 F(n) 是和之前的解 F(n - a) 有联系的，但是思考方向错了。

正确的思路是，给定 n 个 node 之后，直接看 inorder 数组【1, 2, 3, 4 .. n-1, n】

那么选任意位置的元素为 root，都可以建出来一个 valid BST，左子树为 index 左边的 subarray，右子树为 index 右边的 subarray.

于是这就变成了一个利用递归结构的“组合”问题了，解的数量左右相乘。inorder 是 BST 的灵魂啊。

下面是第一次写 AC 的代码，有点粗糙。

```
public class Solution {
    public int numTrees(int n) {
        if(n <= 2) return n;

        int[] dp = new int[n + 1];
        dp[0] = 1;
        dp[1] = 1;
        dp[2] = 2;

        for(int i = 3; i <= n; i++){
            for(int j = 0; j <= i - 1; j++){
                int leftCount = j;
                int rightCount = i - 1 - j;

                dp[i] += dp[leftCount] * dp[rightCount];
            }
        }


        return dp[n];
    }
}
```

Unique Binary Search Trees II

顺着上题的思路，这题也很好做。让我比较惊讶的是改好了拼写之后居然一次提交直接 AC....

```
public class Solution {
    public List<TreeNode> generateTrees(int n) {
        return build(1, n);
    }

    private List<TreeNode> build(int left, int right){
        List<TreeNode> list = new ArrayList<TreeNode>();

        if(left > right){
            return list;
```

```
        }
        if(left == right){
            list.add(new TreeNode(left));
            return list;
        }

        for(int i = left + 1; i <= right - 1; i++){
            List<TreeNode> leftTree = build(left, i - 1);
            List<TreeNode> rightTree = build(i + 1, right);

            for(int leftPtr = 0; leftPtr < leftTree.size(); left
Ptr++){
                for(int rightPtr = 0; rightPtr < rightTree.size(
); rightPtr++){
                    TreeNode root = new TreeNode(i);
                    root.left = leftTree.get(leftPtr);
                    root.right = rightTree.get(rightPtr);

                    list.add(root);
                }
            }

        }

        List<TreeNode> rightTree = build(left + 1, right);
        for(int i = 0; i < rightTree.size(); i++){
            TreeNode root = new TreeNode(left);
            root.right = rightTree.get(i);
            list.add(root);
        }

        List<TreeNode> leftTree = build(left, right - 1);
        for(int i = 0; i < leftTree.size(); i++){
            TreeNode root = new TreeNode(right);
            root.left = leftTree.get(i);
            list.add(root);
        }


        return list;
```

```
    }
}
```

参考了下论坛，同一个思路，比较简洁的写法是

Java 带泛型的 Collections 确实是可以放 null 的， 比如 Queue, List...

在这题里最两边的情况下，list 里直接加个 null 作为 node 用就可以让代码变得非常简洁。

```
public class Solution {
    public List<TreeNode> generateTrees(int n) {
        return n > 0 ? build(1, n) : new ArrayList<TreeNode>();
    }

    private List<TreeNode> build(int left, int right){
        List<TreeNode> list = new ArrayList<TreeNode>();

        for(int i = left; i <= right; i++){
            List<TreeNode> leftTree = build(left, i - 1);
            List<TreeNode> rightTree = build(i + 1, right);

            for(int leftPtr = 0; leftPtr < leftTree.size(); left
Ptr++){
                for(int rightPtr = 0; rightPtr < rightTree.size(
); rightPtr++){
                    TreeNode root = new TreeNode(i);
                    root.left = leftTree.get(leftPtr);
                    root.right = rightTree.get(rightPtr);

                    list.add(root);
                }
            }

        }

        if(list.size() == 0) list.add(null);

        return list;
    }
}
```

Closest Binary Search Tree Value

比较简单的问题，唯一需要考虑的是状态的不连续性；最接近的点可能是 BST 一直往下走的 node，也可能是前面某个 node.

用区间的思想去理解的话，每个 node 都有自己的 valid 区间，区间与区间是重合的。对于给定 target，我们先要找到 target 是什么，然后决定 target 到底靠近重合区间的左边还是右边。

```
public class Solution {
    public int closestValue(TreeNode root, double target) {
        return find(root, target, null);
    }

    public int find(TreeNode root, double target, Integer prev){
        if(root == null) return -1; // Error
        if(prev == null) prev = root.val;

        prev = (Math.abs(root.val - target) < Math.abs(prev - ta
rget)) ? root.val: prev;

        if(root.left != null && target < root.val){
            return find(root.left, target, prev);
        }
        if(root.right != null && target > root.val){
            return find(root.right, target, prev);
        }

        return prev;
    }
}
```

下面是论坛上的写法，思想一致，更取巧了一些。

```
public int closestValue(TreeNode root, double target) {
    TreeNode node = (root.val>target)?root.left:root.right;
    if (node == null) {
        return root.val;
    }
    int value = closestValue(node, target);
    return Math.abs(root.val-target) > Math.abs(value-target)?va
lue:root.val;
}
```

Closest Binary Search Tree Value II

这道题是在二叉树上做 range query.

二叉树上做 range query 普遍要依靠 getPrev() 和 getNext() 函数，或者利用 Parent 指针做 traversal.

但是 BST 不一样，如上题所说， inorder 是 BST 的灵魂。因此这道题可以分为三部分；

```
- 做出 inorder traversal list
- 找 target 对应的 index
- 从 index 两边扫，寻找 k 个元素
```

只在主函数的起始 index 上出了一个小 bug，基本算是一次 AC.

时间复杂度 O(n) + O(log n) + O(k) = O(n)

```
public class Solution {
    public List<Integer> closestKValues(TreeNode root, double ta
rget, int k) {

        List<Integer> inorder = inorder(root);
        int index = binarySearch(inorder, target);
        List<Integer> list = new ArrayList<Integer>();

        int start = index - 1;
        int end = index;
        while(start >= 0 && end < inorder.size()){
            int num = (Math.abs(inorder.get(start) - target) < M
ath.abs(inorder.get(end) - target))
                        ? inorder.get(start--)
                        : inorder.get(end++);
            list.add(num);
            if(list.size() == k) return list;
        }
        while(start >= 0){
            list.add(inorder.get(start--));
            if(list.size() == k) return list;
        }
```

```
        while(end < inorder.size()){
            list.add(inorder.get(end++));
            if(list.size() == k) return list;
        }

        return list;
    }

    private List<Integer> inorder(TreeNode root){
        List<Integer> list = new ArrayList<Integer>();
        TreeNode cur = root;
        while(cur != null){
            if(cur.left == null){
                list.add(cur.val);
                cur = cur.right;
            } else {
                TreeNode prev = cur.left;
                while(prev.right != null && prev.right != cur){
                    prev = prev.right;
                }
                if(prev.right == null){
                    prev.right = cur;
                    cur = cur.left;
                } else {
                    prev.right = null;
                    list.add(cur.val);
                    cur = cur.right;
                }
            }
        }
        return list;
    }

    private int binarySearch(List<Integer> list, double target){
        int start = 0;
        int end = list.size() - 1;
        while(start + 1 < end){
            int mid = start + (end - start) / 2;
            if(list.get(mid) == target){
                return mid;
```

```
            } else if(target < list.get(mid)){
                end = mid;
            } else {
                start = mid;
            }
        }

        if(target < list.get(start)) return start;
        if(target > list.get(end)) return end;

        return (Math.abs(list.get(start) - target) < Math.abs(li
st.get(end) - target)) ? start: end;
    }
}
```

LeetCode 论坛上还有一些其他的写法，也很有意思。这个写法是 O(n log n) + O(k log n)，用自定义的 min heap.

```
public List<Integer> closestKValues(TreeNode root, final double
t, int k) {
    List<Integer>ret = new ArrayList<Integer>();
    PriorityQueue<TreeNode> queue = new PriorityQueue<TreeNode>(
new Comparator<TreeNode>(){
       public int compare(TreeNode n1, TreeNode n2) {
           return Math.abs(n1.val - t) < Math.abs(n2.val - t) ? -
1 : 1;
       }
    });
    findClosest(root, queue);
    while(k-- > 0) ret.add(queue.poll().val);
    return ret;
}

public void findClosest(TreeNode root, PriorityQueue<TreeNode> q
ueue) {
    if(root == null) return;
    findClosest(root.left, queue);
    queue.add(root);
    findClosest(root.right, queue);
}
```

Java 内建的 LinkedList 库是双向链表，implements Deque.

这个解法我很喜欢，类似于 **sliding window maximum** 一样，维护一个大小为 **k** 的 **sliding window**，在树上做 **inorder traversal**，每当 **window size == k** 但是新 **node** 值比 **head** 的值更接近于 **target** 的时候，就给替换掉。当后来发现新来的 **node** 不比 **head** 更接近于 **target** 的时候，就可以返回结果了，而且因为 **LinkedList** 继承了 **List**，连类型转换都不需要。

时间复杂度 **O(n)** ，还带 **early termination**，非常简洁高效的解法，比我写的那个速度快，而且简洁明了。

```
public List<Integer> closestKValues(TreeNode root, double target
, int k) {
    List<Integer> res = new LinkedList<Integer>();
    helper(root, target, k, res);
    return res;
}
private void helper(TreeNode root, double target, int k, List<In
teger> res) {
    if (root == null) {
        return;
    }
    helper(root.left,target,k,res);
    if (res.size()< k) {
        res.add(root.val);
    } else {
        if (Math.abs(res.get(0)-target) > Math.abs(root.val-targ
et)) {
            res.remove(0);
            res.add(root.val);
        } else {
            return;
        }
    }
    helper(root.right,target,k,res);
}
```

Diameter of a Binary Tree

路径与路径和

Binary Tree Paths

Given a binary tree, return all root-to-leaf paths.

For example, given the following binary tree:

1 / \ 2 3 \ 5 All root-to-leaf paths are:

["1->2->5", "1->3"]

- 用 **StringBuilder** 传递可以省去递归中新建的 **string copies.**
- **StringBuilder** 的回溯也很简单，直接 **setLength(int len)** 就行。

在 leaf node 上以 sb.setLength() 收尾是为了 backtracking，因为后面的两个 dfs 都没有做，不走到底是不会回溯的，要由到了 leaf node 那一层递归进行处理。这点和常见的 subsets 和 permutations 不太一样，那两题中收尾直接 add 然后 return 就可以了，而回溯在 dfs 之后做。

在 dfs 之后直接 backtracking 会遇到问题，例如路径上某个分叉上只有一边有节点，向 null 方向探索之后会删掉来路。

```
public class Solution {
    public void help(List<String> list, TreeNode node, StringBui
lder sb) {
        if (node == null) return;

        // 存盘，保存当前路径
        int len=sb.length();
        sb.append(node.val);

        if (node.left == null && node.right == null) {
            list.add(sb.toString());
            // 抹掉末尾数字
            sb.setLength(len);
            return;
        }

        sb.append("->");
        help(list, node.left, sb);
        help(list, node.right, sb);
        //  抹掉箭头和前一个数字
        sb.setLength(len);
    }

    public List<String> binaryTreePaths(TreeNode root) {
        List<String> res = new ArrayList<String>();
        help(res, root, new StringBuilder());
        return res;
    }
}
```

113. Path Sum II

和前一题研究的 Tree DFS + Backtracking 完全一个套路。只不过回溯的时候，记得把 curSum 也给回溯了就行。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
```

```
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<List<Integer>> pathSum(TreeNode root, int sum) {
        List<List<Integer>> rst = new ArrayList<List<Integer>>()
;
        if(root == null) return rst;
        dfs(rst, root, new ArrayList<Integer>(), 0, sum);
        return rst;
    }

    private void dfs(List<List<Integer>> rst, TreeNode root, Lis
t<Integer> list, int curSum, int targetSum){
        if(root == null) return;

        list.add(root.val);
        curSum += root.val;

        if(root.left == null && root.right == null){
            if(curSum == targetSum){
                rst.add(new ArrayList<Integer>(list));
                list.remove(list.size() - 1);
                return;
            }
        }

        dfs(rst, root.left, list, curSum, targetSum);
        dfs(rst, root.right, list, curSum, targetSum);
        curSum -= root.val;
        list.remove(list.size() - 1);
    }
}
```

- 5/27 号重做了一遍，这次的代码

```
public class Solution {
    public List<List<Integer>> pathSum(TreeNode root, int sum) {
        List<List<Integer>> rst = new ArrayList<List<Integer>>()
;

        dfs(rst, new ArrayList<Integer>(), root, sum);

        return rst;
    }

    private void dfs(List<List<Integer>> rst, List<Integer> list
, TreeNode root, int sum){
        if(root == null) return;

        if(root.left == null && root.right == null){
            if(root.val == sum){
                list.add(root.val);
                rst.add(new ArrayList<Integer>(list));
                list.remove(list.size() - 1);
                return;
            } else {
                return;
            }
        }

        list.add(root.val);
        dfs(rst, list, root.left, sum - root.val);
        dfs(rst, list, root.right, sum - root.val);
        list.remove(list.size() - 1);

    }
}
```

112. Path Sum I

简化版，这次不用存所有的 Paths 了，套用 Tree DFS 模板可以，不过在只求 boolean / int 的情况下，也有更简单直接的写法。

```
public class Solution {
    public boolean hasPathSum(TreeNode root, int sum) {
        return dfs(root, 0, sum);
    }

    private boolean dfs(TreeNode root, int curSum, int targetSum)
{
        if(root == null) return false;

        curSum += root.val;
        if(root.left == null && root.right == null){
            if(curSum == targetSum) return true;
        }

        boolean left = dfs(root.left, curSum, targetSum);
        boolean right = dfs(root.right, curSum, targetSum);
        curSum -= root.val;

        return (left || right);
    }
}
```

或者更直接点，

```
public class Solution {
    public boolean hasPathSum(TreeNode root, int sum) {
        if(root == null) return false;
        if(root.left == null && root.right == null && root.val =
= sum) return true;
        return (hasPathSum(root.left, sum - root.val) || hasPath
Sum(root.right, sum - root.val));
    }
}
```

Binary Tree Maximum Path Sum

这题有点 Tricky，不好做。假定有如下的Tree:

```
        5
       / \
      1   -1
     / \  / \
    0 -2  3  2
```

最大和路径有这么几种可能：

- 从 root 出发，路上看到负数，不采用；
- 从 root 出发，路上看到负数，负数后面存在总和超过负数节点的路径；
- 最大和在某个从 leaf node 往上走的一条路径上，不过 root.
- 左路径最大，采用左路径；
- 右路径最大，采用右路径；
- 单独节点最大，可能是 左/右/根 其中之一。

换句话说，一个重要的问题是，我们只能从 root 开始，也没有 parent 指针，但是最优的路径可能却和 root 是不连续的，这就切断了 Binary Tree divide & conquer / Tree DFS 里面大多数做法中非常依赖的性质，即层层递归之前 左/右 子树和根节点的联系。

然而套路还是要用的，要么这题就没法做了。。好在问题没有要求返回具体 path，只要一个 max sum, 想连接全局最优就要用一个全局变量 int max. 从 leaf node 开始 bottom-up 进行处理的时候还不需要考虑“切断”的问题，因此还可以用套路，注意随时更新全局 max 就好。从 bottom-up 的角度看，这是一个从底部不停 merge 最优的子路径在根节点会和的过程。

每一层的“三角形”路径都要和全局最优 update 一下，然而不是有效的 path. 最终 return 的结果只是“必须包括当前节点” 的最大 path，由此完成连续的递归结构（recurrence substructure）

另外一个小技巧是，在求 max sum 的情况下，每一个节点可以有 “选”（root.val）或者 “不选”（0） 两种选择，在单个

```
public class Solution {
    // 全局变量，用于连接不连续的状态
    int max = Integer.MIN_VALUE;
    public int maxPathSum(TreeNode root) {
        dfsBottomUp(root);
        return max;
    }

    private int dfsBottomUp(TreeNode root){
        if(root == null) return 0;

        // 检查两边的做大路径和，或者直接抛弃（取值为0）
        // 因此当一个小三角形一边为负数的时候
        // 最后返回的结果看起来是三个点的和，其实只是一条边
        int left = Math.max(0, dfsBottomUp(root.left));
        int right = Math.max(0, dfsBottomUp(root.right));

        // 检查通过当前 “root” 的三角形路线（拐弯）
        // 不需要单独再取 Left / Right 中的最大值
        // 因为在 Bottom-Up 的递归中左右子树的最大路径已经被更新过了
        // 即其中某个分支为负时，最大子树和 = 最大路径和
        max = Math.max(max, left + right + root.val);

        // 传递到上一层的路线必须连续且不能拐弯，保持连续的递归状态
        return Math.max(left, right) + root.val;
    }
}
```

如果不喜欢这种用全局变量的方式，也可以用如下九章的解法，其实骨子里面的思路一样，把全局变量用一个 tuple 包起来了。

比较值得参考的是这种 dfs 之间传 tuple 的方式很好的解决了信息传递的问题，其中的变量可以是符合递归连续结构的，也可以是全局的，看起来比较适合 generalize 到其他 Tree 的问题。

```
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: An integer.
```

```
     */
    private class ResultType {
        // singlePath: 从root往下走到任意点的最大路径，至少包含一个点
        // maxPath: 从树中任意到任意点的最大路径，这条路径至少包含一个点
        int singlePath, maxPath;
        ResultType(int singlePath, int maxPath) {
            this.singlePath = singlePath;
            this.maxPath = maxPath;
        }
    }

    private ResultType helper(TreeNode root) {
        if (root == null) {
            return new ResultType(Integer.MIN_VALUE, Integer.MIN
_VALUE);
        }
        // Divide
        ResultType left = helper(root.left);
        ResultType right = helper(root.right);

        // Conquer
        int singlePath =
            Math.max(0, Math.max(left.singlePath, right.singlePa
th)) + root.val;

        int maxPath = Math.max(left.maxPath, right.maxPath);
        maxPath = Math.max(maxPath,
                           Math.max(left.singlePath, 0) +
                           Math.max(right.singlePath, 0) + root.
val);

        return new ResultType(singlePath, maxPath);
    }

    public int maxPathSum(TreeNode root) {
        ResultType result = helper(root);
        return result.maxPath;
    }

}
```

Sum Root to Leaf Numbers

都是套路啊，套路。Top-Bottom 的递归回溯。

```
public class Solution {
    public int sumNumbers(TreeNode root) {
        return dfs(root, 0);
    }

    // 把当前考虑的节点作为参数的 dfs 结构
    private int dfs(TreeNode root, int num){
        // 只在叶节点上做计算，这里说明不是有效 path
        if(root == null) return 0;

        -------------ADD----------------
        num += root.val;

        ------------Leaf Node-----------
        if(root.left == null && root.right == null){
            return num;
        }

        ------------DFS------------------
        int left = dfs(root.left, num * 10);
        int right = dfs(root.right, num * 10);

        --------Backtracking------------
        num -= root.val;

        return left + right;
    }
}
```

如果 dfs 里面共享的变量是 String 或者 primitive type ，那么不存在多层递归共同 reference 的问题，backtracking 的步骤也就可以省去了。

从这个角度看，其实很多看起来非常简洁的 Binary Tree 问题都是因为 dfs 函数里面没有 by reference 的东西，属于 dfs + backtracking 的低配简化版。

```
public class Solution {
    public int sumNumbers(TreeNode root) {
        return dfs(root, 0);
    }

    private int dfs(TreeNode root, int num){
        if(root == null) return 0;

        if(root.left == null && root.right == null){
            return num + root.val;
        }

        int left = dfs(root.left, (num + root.val) * 10);
        int right = dfs(root.right, (num + root.val) * 10);

        return left + right;
    }
}
```

Binary Tree Longest Consecutive Sequence

全局最优解不一定和 root 与 dfs 递归连续，因而用全局变量 int max 解决。 其他部分无压力套模板。

不爱用全局变量也好办，建个长度为 **1** 的 **int[]** 当参数传就行了。

```
    public int longestConsecutive(TreeNode root) {
        int[] max = new int[1];
        dfs(root, null, 0, max);
        return max[0];
    }

    private void dfs(TreeNode root, TreeNode parent, int length,
int[] max){
        // Not valid path to leaf node
        if(root == null) return;

        // ADD
        if(parent == null || root.val - parent.val == 1){
            length ++;
        } else {
            length = 1;
        }

        max[0] = Math.max(max[0], length);

        dfs(root.left, root, length, max);
        dfs(root.right, root, length, max);

    }
```

NestedInteger 类

所有的 **NestedInteger** 问题，都是多叉树的问题。

这树，长这样：

(G) Flatten List

递归的很简单，对于每棵 tree root 都是一个【左 - 右】的顺序 dfs 处理其 subtrees.

```
    public List<Integer> flatten(List<NestedInteger> nestedList)
{
        // Write your code here
        List<Integer> rst = new ArrayList<>();
        for(NestedInteger node : nestedList){
            dfs(rst, node);
        }
        return rst;
    }

    private void dfs(List<Integer> rst, NestedInteger node){
        if(node.isInteger()){
            rst.add(node.getInteger());
        } else {
            for(NestedInteger next : node.getList()){
                dfs(rst, next);
            }
        }
    }
```

迭代先试了下 **BFS** ，发现顺序有问题。靠 **Stack** 遇到 **List** 就反向遍历 **push** 倒是能跑通。

这个迭代写法其实就是自己用 **Stack** 复现了一遍递归的过程。

```
    public List<Integer> flatten(List<NestedInteger> nestedList)
{
        // Write your code here
        List<Integer> rst = new ArrayList<>();
        Stack<NestedInteger> stack = new Stack<>();

        for(int i = nestedList.size() - 1; i >= 0; i--){
            stack.push(nestedList.get(i));
        }

        while(!stack.isEmpty()){
            NestedInteger node = stack.pop();
            if(node.isInteger()){
                rst.add(node.getInteger());
            } else {
                List<NestedInteger> list = node.getList();
                for(int i = list.size() - 1; i >= 0; i--){
                    stack.push(list.get(i));
                }
            }
        }
        return rst;
    }
```

Nested List Weight Sum

就是最简单的 DFS，非常 trivial 的问题。。

```
public class Solution {
    public int depthSum(List<NestedInteger> nestedList) {
        int sum = 0;
        for(NestedInteger num : nestedList){
            sum += dfs(num, 1);
        }
        return sum;
    }

    private int dfs(NestedInteger nestedInt, int depth){
        if(nestedInt.isInteger()){
            return depth * nestedInt.getInteger();
        } else {
            int sum = 0;
            List<NestedInteger> list = nestedInt.getList();
            for(NestedInteger num : list){
                sum += dfs(num, depth + 1);
            }
            return sum;
        }
    }
}
```

Nested List Weight Sum II

从上往下递归可以传参数，自底向上递归传 tuple.

比较诡异的是 [[-1], [[-1]]] 这样的 test case ，第一个 [-1] 的 weight 居然是 2，导致最终结果是 -3 .. 让我觉得这题的 test case 定义有点不清楚。。

于是下面这个 dfs 的代码会出错，因为没正确处理当前 list 也是嵌套的正确 weight. 这段代码的思路是对于每一个位置，其 weight = 其子树的最深距离，但是和原题的定义不一样。

这题的正确理解是，每当看到 Integer，代表这是一个leaf node； 每当看到一个 List，代表这是一个 subtree.

```
public class Solution {
    private class Tuple{
        int sum;
        int depth;
        public Tuple(int val, int dep){
            sum = val;
            depth = dep;
        }
    }
    public int depthSumInverse(List<NestedInteger> nestedList) {
        return dfs(nestedList).sum;
    }

    private Tuple dfs(List<NestedInteger> list){
        int sum = 0;
        int maxDepth = 1;
        for(NestedInteger node : list){
            if(!node.isInteger()){
                Tuple tuple = dfs(node.getList());
                sum += tuple.sum;
                maxDepth = Math.max(maxDepth, tuple.depth + 1);
            }
        }
        for(NestedInteger node : list){
            if(node.isInteger()) sum += maxDepth * node.getInteg
er();
        }

        return new Tuple(sum, maxDepth);
    }
}
```

于是乎这题的正确打开姿势其实是，自顶向下 level order 的看，只要下面还有一层，就把当前的所有结果都再加上一遍，起到相乘的效果；这样随着探索的不断深入，就可以正确地得到每层的正确 weight 了，因为每个 node 的 weight = 这个 node 到树最深节点的距离，一个天然的 BFS 问题。

```
public class Solution {
    public int depthSumInverse(List<NestedInteger> nestedList) {
        int total = 0, prevAll = 0;
        while(!nestedList.isEmpty()){
            List<NestedInteger> nextLvl = new ArrayList<NestedIn
teger>();
            for(NestedInteger next : nestedList){
                if(next.isInteger()){
                    prevAll += next.getInteger();
                } else {
                    nextLvl.addAll(next.getList());
                }
            }

            total += prevAll;
            nestedList = nextLvl;
        }

        return total;
    }
}
```

这题当然也有 DFS 解法，要先把图过一遍，侦查好 maxDepth; 然后递归解决的时候，每一层的权重是 maxDepth - curDepth;

居然能一次 AC ，好神奇。。

```
public class Solution {
    public int depthSumInverse(List<NestedInteger> nestedList) {
        int maxDepth = 0;
        for(NestedInteger next : nestedList){
            maxDepth = Math.max(getMaxDepth(next), maxDepth);
        }

        int sum = 0;
        for(NestedInteger next : nestedList){
            sum += dfs(next, maxDepth, 0);
        }
```

```
        return sum;
    }

    private int dfs(NestedInteger node, int maxDepth, int curDep
th){
        if(node.isInteger()){
            return node.getInteger() * (maxDepth - curDepth);
        } else {
            int sum = 0;
            for(NestedInteger next : node.getList()){
                sum += dfs(next, maxDepth, curDepth + 1);
            }
            return sum;
        }
    }

    private int getMaxDepth(NestedInteger num){
        if(num.isInteger()) return 1;

        int max = 0;
        List<NestedInteger> nestedList = num.getList();
        for(NestedInteger next : nestedList){
            max = Math.max(getMaxDepth(next), max);
        }

        return max + 1;
    }
}
```

Flatten Nested List Iterator

这题和 BST iterator 很像，因为实际上都是利用 stack + cur 指针做一个 inorder 遍历。

不同之处是，Binary Tree 是双叉的，有个 node 就可以满足输出当前元素 + 寻找下一元素的需要，我们这个情况要复杂一些。

NestedInteger 是一种树状结构，其中每一个是 List 的元素代表一个三角形，下面有自己的子树。

了解了这个结构之后，为了遍历寻找下一个元素，就需要依靠 List 作为一个 Collection interface 里自带的 iterator 了，在这里 iterator 就充当了 BST 里面 cur 指针的地位，用于在树上定位，和寻找下一个元素； 同时 Stack<> 所存储的，就是各个 iterator.

实现过程中要注意的是，test case 会根据 boolean hasNext 决定是否继续输出，而这种结构不同于 binary tree，可能会有 [[ ]] 这种情况，此时 stack 里有东西，cur.hasNext() 也返回 true，无法正确得知下面是否真的有元素存在。

所以要在 hasNext 里面执行程序逻辑。

同时对于一个 iterator，如果已经没有新元素了也不必要 push 到 stack 中。

速度超过 **80.50%** ，下面的代码能 **AC**

但是我觉得不算很严谨，如果测试用例调用很多次 **hasNext()**，会对现有的 **iterator** 造成影响，不应该是正确的。

```java
public class NestedIterator implements Iterator<Integer> {
    Stack<Iterator<NestedInteger>> stack;
    Iterator<NestedInteger> cur;
    Integer next;

    public NestedIterator(List<NestedInteger> nestedList) {
        stack = new Stack<Iterator<NestedInteger>>();
        cur = nestedList.iterator();
        next = null;
    }

    @Override
    public Integer next() {
        return next;
    }

    @Override
    public boolean hasNext() {
        while(cur.hasNext() || !stack.isEmpty()){
            while(cur.hasNext()){
                NestedInteger elem = cur.next();
                if(elem.isInteger()) {
                    next = elem.getInteger();
                    return true;
                } else {
                    if(cur.hasNext()) stack.push(cur);
                    cur = elem.getList().iterator();
                }
            }
            if(!stack.isEmpty()) cur = stack.pop();
        }

        return false;
    }
}
```

我比较认同的写法是这种，用内部函数来准备 **next()** 和 **hasNext() API** 的返回。

```java
public class NestedIterator implements Iterator<Integer> {

    Stack<Iterator<NestedInteger>> stack;
    Iterator<NestedInteger> cur;
    Integer num;

    public NestedIterator(List<NestedInteger> nestedList) {
        stack = new Stack<>();
        cur = nestedList.iterator();
        num = internalNext();
    }

    private Integer internalNext(){
        if(cur.hasNext()){
            NestedInteger elem = cur.next();
            if(elem.isInteger()){
                return elem.getInteger();
            } else {
                if(cur.hasNext()) stack.push(cur);
                cur = elem.getList().iterator();
                return internalNext();
            }
        } else {
            if(stack.isEmpty()) return null;
            cur = stack.pop();
            return internalNext();
        }
    }

    @Override
    public Integer next() {
        Integer tmp = num;
        num = internalNext();
        return tmp;
    }

    @Override
    public boolean hasNext() {
        return (num != null);
    }
```

```
}
```

中间的部分改成迭代也很简单，这样就行；

```
    private Integer internalNext(){
        while(!stack.isEmpty() || cur.hasNext()){
            if(cur.hasNext()){
                NestedInteger elem = cur.next();
                if(elem.isInteger()){
                    return elem.getInteger();
                } else {
                    if(cur.hasNext()) stack.push(cur);
                    cur = elem.getList().iterator();
                    continue;
                }
            } else {
                if(stack.isEmpty()) return null;
                cur = stack.pop();
                continue;
            }
        }
        return null;
    }
```

(FB高频) 如何递归转迭代

例题：给定 Binary Tree，print 所有 root - leaf 的 path.

```
          A
         / \
       B    D
      /    / \
     C    E    F
```

- Input : Root A
- Output :
  - A,B,C
  - A,D,E
  - A,D,F

递归：

比较简单直接，就是在树上 DFS + backtrack，带着当前节点 root 和当前路径 list 递归处理即可。这题的结构和 LC 那道 Binary Paths 一样。

```
    public static void printPaths_recursion(TreeNode root){
        dfs(root, new ArrayList<>());
    }

    private static void dfs(TreeNode root, List<TreeNode> list){
        if(root == null) return;

        list.add(root);

        if(root.left == null && root.right == null){
            for(TreeNode node : list) System.out.print(node.val
+ ",");
            System.out.println();
            list.remove(list.size() - 1);
            return;
        }

        dfs(root.left, list);
        dfs(root.right, list);

        list.remove(list.size() - 1);
    }
```

迭代解法 1：

- **Queue**，空间占用和时间都高一些。
- 基本思路就是用一个 **Queue** 存所有当前的 **path**，队列中的 **path** 数量与进出关系都和正常的 **level order traversal** 完全一样，每次队列末尾的节点，都是 **current node.**
- 中间 **poll** 的时候看到叶子节点，就把 **list** 输出出来就好。
- 这种解法的问题一是空间占用是 **O(N)** 的，相比 **Stack** 高；二是每次生成新 **node** 的时候其实是在做一次 **collection** 的 **copy** 操作，时间复杂度上也不经济。

```
private static void printPath_bfs(TreeNode root){
    Queue<List<TreeNode>> queue = new LinkedList<>();

    if(root != null){
        List<TreeNode> rootList = new ArrayList<>();
        rootList.add(root);
        queue.offer(rootList);
    }

    while(!queue.isEmpty()){
        List<TreeNode> curList = queue.poll();
        TreeNode curNode = curList.get(curList.size() - 1);

        if(curNode.left == null && curNode.right == null){
            for(TreeNode node : curList) System.out.print(no
de.val + ",");
            System.out.println();
        }

        if(curNode.left != null){
            List<TreeNode> list = new ArrayList<>(curList);
            list.add(curNode.left);
            queue.offer(list);
        }
        if(curNode.right != null){
            List<TreeNode> list = new ArrayList<>(curList);
            list.add(curNode.right);
            queue.offer(list);
        }
    }
}
```

迭代解法 2:

Stack + HashSet

迭代解法 3: Stack + 自定义 StackFrame

这种做法最有意思，我也最喜欢，觉得只要有合适的 field 属性，很容易 generalize 到各种其他递归解法中。

首先对于二叉树，我们可以这样定义 StackFrame 的三个状态：

- **0 :** 刚入栈，未访问子树；
- **1 :** 正在访问左子树，返回代表左子树访问完毕；
- **2 :** 正在访问右子树，返回代表右子树访问完毕；

这样代码如下，每次 **stack** 里存的，都是当前路径 **(**配合一个 **List** 做高效 **print path** 操作**)**，而每次栈顶的 **StackFrame** 都记录了当前 **node** 的访问状态和下一步的动向。

这种 **0/1/2** 的状态表示方式看着有点像 **Graph** 里面做 **DFS / BFS** 的标注，其实不完全一样，只是在这题我们的树一定是二叉而已。

对于多叉树的情况，改动也不会很难；这次 **index** 代表着 **“**下一个要访问的 **child index”**，当 **index == children.size()** 的时候，我们就可以知道当前 **node** 的所有子节点都访问完了。

```
public static void printPaths(TreeNode root){
        Stack<StackFrame> stack = new Stack<>();
        List<TreeNode> list = new ArrayList<>();
        if(root != null) stack.push(new StackFrame(0, root));

        while(!stack.isEmpty()){
            StackFrame curFrame = stack.peek();
            TreeNode curNode = curFrame.node;

            if(curNode == null) {
                stack.pop();
            } else if(curFrame.status == 0){
                list.add(curNode);
                curFrame.status = 1;
                stack.push(new StackFrame(0, curNode.left));
            } else if(curFrame.status == 1){
                curFrame.status = 2;
                stack.push(new StackFrame(0, curNode.right));
            } else {
                if(curNode.left == null && curNode.right == null
) {
                    for (TreeNode node : list) System.out.print(
"" + node.val + ",");
                    System.out.println();
                }
                stack.pop();
                list.remove(list.size() - 1);
            }
        }
    }
```

(FB) Binary Tree Path 比较路径大小

http://www.1point3acres.com/bbs/thread-128584-1-1.html

第二道题是 给个Tree 不一定是平衡的， 要求 把所有路径排序后 按字符串那样的比较大小方法 找出最小的路径 时间要求线性的。 比如
5 / \ 10 3 1 7 8

路径有 5 10 1 ； 5 10 7 ； 5 3 8

排序后 1 5 10 ； 5 7 10 ； 3 5 8

所以按字符串类型排序 为 1 5 10 < 3 5 8 < 5 7 10；

比较好玩的 Binary Tree 概率题

给一个node有parent指针的 complete tree，每个node的值为 0 / 1；每个parent的值为两个子node的 “AND” 结果；现在把一个leaf翻牌子（0变1或者1变0）

1，写代码 (太简单了不贴了。。)

2, 时间复杂度的期望值是多少？(一颗赛艇)

leaf node 区分左右的话，可能有4种组合，每种组合分别有 flip{left, right} 两种选择，于是：

- 0,0 - 左，变成1； 右，变成1
- 0,1 - 左，不变； 右，变成0
- 1,0 - 左，变成0；右，不变；
- 1,1 - 左，不变，右，不变；

总共会造成 parent node 的值和之前有变化的可能是 4 种，所以在每一层 parent - child 的关系中任意 flip 一个 child，有 1/2 的几率造成 parent 的值也被 flip.

假设二叉树中总共有 N 的节点，其深度/自底向上路径长是 h = O(logN)，那我们自底向上的更新路径实际次数就会在 【1, h】 的闭区间内，最少更新次数为 1 (计算了 leaf，但是发现对 parent 无任何影响)，最大计算次数为 h ，代表一路更新到了 root.

那我们总共会计算【1,2,3 ... h】层的概率就是 【1 , 1/2 , (1/2)^2 , (1/2)^3 ... (1/2)^h】，其对应的计算次数期望值就是 1 + 1 *1/2* + *2* (1/2)^2 + 3 *(1/2)^3* + *.... h* (1/2)^h = 1 + Sigma (h / 2^h)，在 n 趋近于无限的情况下，upper bound 是 1 + 2 = 3 ~ 所以总共的计算次数期望值确实是常数。

Segment & Fenwick Tree

这组问题的分类上基本延续了九章高级算法班的大纲，加了些听课时候的笔记，和自己之后查资料的思考。

Segment Tree 基础操作

- **Segment Tree** 是一个 **Full Binary Tree**，每个节点子节点数量为 **0** 或 **2.**
- 给定含 **n** 个元素的数组区间，对应的 **Segmeng Tree** 节点数量最多为 **2n - 1.**
- **Build O(n)**
- **Update O(log n)**
- **Query O(log n)**
- 偶数长度区间会拆出两个偶数**OR**奇数长度区间；奇数长度区间会拆出一个偶数**+**一个奇数长度区间， 最终以长度为 **1** 的区间为叶节点。

Segment Tree Build

比较基本，就是很普通的建二叉树。

```
public class Solution {
    /**
     *@param start, end: Denote an segment / interval
     *@return: The root of Segment Tree
     */
    public SegmentTreeNode build(int start, int end) {
        // write your code here
        if(start > end) return null;
        SegmentTreeNode root = new SegmentTreeNode(start, end);
        if(start == end) return root;
        int mid = start + (end - start) / 2;
        root.left = build(start, mid);
        root.right = build(mid + 1, end);

        return root;
    }
}
```

Segment Tree Build II

这个例子更具有实际意义一点，因为这个建出来的 Segment Tree 已经可以用了。

- 对于给定无序数组，**Segment Tree** 可以利用递归在 **O(n)** 时间建立。
- **Segment Tree** 总共节点个数为 **2n - 1**，建立每个节点操作时间为 **O(1).**
- 另一个角度看的话，总共有 **n** 个叶节点，那么对应的 **perfect binary tree** 总节点个数就是 **2n - 1.**
- 结构和 **quick sort / merge sort** 非常类似，每一层包含的所有区间覆盖整个数组。

```
public class Solution {
    /**
     *@param A: a list of integer
     *@return: The root of Segment Tree
     */
    public SegmentTreeNode build(int[] A) {
        // write your code here
        return buildHelper(A, 0, A.length - 1);
    }

    private SegmentTreeNode buildHelper(int[] A, int start, int
end){
        if(start > end) return null;
        if(start == end) return new SegmentTreeNode(start, end,
A[start]);

        int mid = start + (end - start) / 2;

        SegmentTreeNode left = buildHelper(A, start, mid);
        SegmentTreeNode right = buildHelper(A, mid + 1, end);

        SegmentTreeNode root = new SegmentTreeNode(start, end, M
ath.max(left.max, right.max));
        root.left = left;
        root.right = right;

        return root;
    }
}
```

Segment Tree Modify

第一次写的时候翻了个错误，只考虑了改的值变大，更新新的 max 的情况，而没考虑更新的值可能是原来某个区间节点的 max，变小之要更新整个到 Root 路径的 max.

- 每次 **update** 都是一次 **top-down** 的递归，实际更新是由最小的区间单位 **bottom-up** 一直到 **root** 的路径更新。

```
public class Solution {
    /**
     *@param root, index, value: The root of segment tree and
     *@ change the node's value with [index, index] to the new g
iven value
     *@return: void
     */
    public void modify(SegmentTreeNode root, int index, int valu
e) {
        // write your code here
        if(root == null) return;
        if(index < root.start || index > root.end) return;

        // Segment Tree 不会出现单独分叉的节点，所以到叶节点可以直接返回。

        if(index == root.start && index == root.end){
            root.max = value;
            return;
        }

        modify(root.left, index, value);
        modify(root.right, index, value);

        root.max = Math.max(root.left.max, root.right.max);
    }
}
```

Segment Tree Query

对于区间覆盖问题可以多画点图，考虑下所有可能的情况。

对于每一层的查询，最多只会分裂成两个节点；

- 如果目标区间完全不在 root 的区间里，直接返回；
- 否则设有效查询区间为
    - max(root.start, query.start)
    - min(root.end, query.end)

- 处理下正确叶节点的位置，递归处理。

```
public class Solution {
    /**
     *@param root, start, end: The root of segment tree and
     *                         an segment / interval
     *@return: The maximum number in the interval [start, end]
     */
    public int query(SegmentTreeNode root, int start, int end) {
        // write your code here
        if(end < root.start) return Integer.MIN_VALUE;
        if(start > root.end) return Integer.MIN_VALUE;

        start = Math.max(start, root.start);
        end = Math.min(end, root.end);

        if(start == root.start && end == root.end) return root.m
ax;

        int left = query(root.left, start, end);
        int right = query(root.right, start, end);

        return Math.max(left, right);
    }
}
```

Segment Tree Query II

这题和上一题没有任何区别。。你是想告诉我 TreeNode 里除了 max/min 还可以存 count 是吗。。。

```
public class Solution {
    /**
     *@param root, start, end: The root of segment tree and
     *                         an segment / interval
     *@return: The count number in the interval [start, end]
     */
    public int query(SegmentTreeNode root, int start, int end) {
        // write your code here
        if(root == null) return 0;
        if(end < root.start) return 0;
        if(start > root.end) return 0;

        start = Math.max(start, root.start);
        end = Math.min(end, root.end);

        if(root.start == start && root.end == end) return root.c
ount;

        int left = query(root.left, start, end);
        int right = query(root.right, start, end);

        return left + right;

    }
}
```

Segment Tree 的应用

- 最适合用 **Segment tree** 的情形最好同时满足以下三点：
  - 区间查找 **min/max**
  - 频繁 **update**
  - 频繁 **query**

Range Sum Query - Mutable

非常不错的一道题，Segment Tree的常用操作都考到了。

这题更快的做法是用 Binary Indexed Tree，有空我研究下。

```
public class NumArray {
    private class SegmentTreeNode{
        int start;
        int end;
        int sum;
        SegmentTreeNode left, right;

        public SegmentTreeNode(){}
        public SegmentTreeNode(int start, int end, int sum){
            this.start = start;
            this.end = end;
            this.sum = sum;
            this.left = null;
            this.right = null;
        }
    }

    SegmentTreeNode root;

    public NumArray(int[] nums) {
```

```
        root = buildTree(nums, 0, nums.length - 1);
    }

    private SegmentTreeNode buildTree(int[] nums, int start, int
 end){
        if(nums == null || nums.length == 0) return null;
        if(start == end) return new SegmentTreeNode(start, end,
nums[start]);

        int mid = start + (end - start) / 2;

        SegmentTreeNode left = buildTree(nums, start, mid);
        SegmentTreeNode right = buildTree(nums, mid + 1, end);

        SegmentTreeNode root = new SegmentTreeNode(start, end, l
eft.sum + right.sum);
        root.left = left;
        root.right = right;

        return root;
    }

    void update(int i, int val) {
        update(root, i, val);
    }

    private void update(SegmentTreeNode root, int i, int val){
        if(root == null) return;
        if(i < root.start) return;
        if(i > root.end) return;

        if(root.start == i && root.end == i) {
            root.sum = val;
            return;
        }

        update(root.left, i, val);
        update(root.right, i, val);

        root.sum = root.left.sum + root.right.sum;
```

```
    }

    public int sumRange(int i, int j) {
        return sumRange(root, i, j);
    }

    private int sumRange(SegmentTreeNode root, int i, int j){
        if(root == null) return 0;
        if(i > root.end) return 0;
        if(j < root.start) return 0;

        i = Math.max(i, root.start);
        j = Math.min(j, root.end);

        if(root.start == i && root.end == j) return root.sum;

        int left = sumRange(root.left, i, j);
        int right = sumRange(root.right, i, j);

        return left + right;
    }
}


// Your NumArray object will be instantiated and called as such:
// NumArray numArray = new NumArray(nums);
// numArray.sumRange(0, 1);
// numArray.update(1, 10);
// numArray.sumRange(1, 2);
```

Range Sum Query - Immutable

这题从类型上讲，看着和上一题非常像。然而其实这题因为需要的操作比较简单，其实就是一个 prefix sum 数组的 dp ...

教育了我们 segment tree 虽屌，也不要一言不合就随便用。。

- 最适合用 **Segment tree** 的情形最好同时满足以下三点：

  - 区间查找
  - 频繁 **update**
  - 频繁 **query**

- 在只有区间没有 **update** 的情况下，其实是一个一维/二维的 **DP** 问题，并不能体现出 **segment tree** 的优势。
- 前缀和数组记得在最前面加上 **sum = 0** 的 **padding.**

```
public class NumArray {
    int[] prefixSum;

    public NumArray(int[] nums) {
        if(nums == null || nums.length == 0) return;

        prefixSum = new int[nums.length + 1];
        prefixSum[0] = 0;
        prefixSum[1] = nums[0];
        for(int i = 1; i < nums.length; i++){
            prefixSum[i + 1] = prefixSum[i] + nums[i];
        }
    }

    public int sumRange(int i, int j) {
        return prefixSum[j + 1] - prefixSum[i];
    }
}


// Your NumArray object will be instantiated and called as such:
// NumArray numArray = new NumArray(nums);
// numArray.sumRange(0, 1);
// numArray.sumRange(1, 2);
```

Range Sum Query 2D - Mutable

这道题当然可以把矩阵降维之后用 segment tree 解，把一个 region 拆分成若干个 interval of rows 然后把结果加起来，但是很慢。

这题既体现了 segment tree的应用，又暴露了 segment tree的问题。

- **width = m, height = n, 现有 1D segment tree 的复杂度**
  - **build O(mn)**
  - **update O(log(mn))**
  - **query O(n * log (mn))**
- 因为这题更适合用 **binary index tree** 解，另一个教程贴在这里，还有这里，加上这个陈老师推荐的中文帖子

Fenwick tree can also be used to update and query subarrays in multidimensional arrays with complexity $O(2^d \log^d n)$, where d is number of dimensions and n is the number of elements along each dimension.

```
public class NumMatrix {

    private class SegmentTreeNode{
        int start;
        int end;
        int sum;
        SegmentTreeNode left;
        SegmentTreeNode right;
        public SegmentTreeNode(){}
        public SegmentTreeNode(int start, int end, int sum){
            this.start = start;
            this.end = end;
            this.sum = sum;
            this.left = null;
            this.right = null;
        }
    }
```

```
    int width;
    int height;
    SegmentTreeNode root;

    private int getIndex(int x, int y){
        return x * width + y;
    }

    public NumMatrix(int[][] matrix) {
        if(matrix == null || matrix.length == 0) return;

        height = matrix.length;
        width = matrix[0].length;

        root = buildTree(matrix, 0, width * height - 1);
    }

    private SegmentTreeNode buildTree(int[][] matrix, int start,
int end){
        if(start == end) return new SegmentTreeNode(start, end,
matrix[start / width][start % width]);

        int mid = start + (end - start) / 2;
        SegmentTreeNode left = buildTree(matrix, start, mid);
        SegmentTreeNode right = buildTree(matrix, mid + 1, end);
        SegmentTreeNode root = new SegmentTreeNode(start, end, l
eft.sum + right.sum);
        root.left = left;
        root.right = right;

        return root;
    }

    public void update(int row, int col, int val) {
        int index = getIndex(row, col);
        update(root, index, val);
    }

    private void update(SegmentTreeNode root, int index, int val)
{
```

```
        if(root == null) return ;
        if(index < root.start) return;
        if(index > root.end) return;

        if(root.start == index && root.end == index){
            root.sum = val;
            return;
        }

        update(root.left, index, val);
        update(root.right, index, val);

        root.sum = root.left.sum + root.right.sum;
    }

    public int sumRegion(int row1, int col1, int row2, int col2)
{
        int sum = 0;
        if(col1 == 0 && col2 == width - 1){
            sum = querySum(root, getIndex(row1, col1), getIndex(
row2, col2));
        } else {
            for(; row1 <= row2; row1++){
                sum += querySum(root, getIndex(row1, col1), getI
ndex(row1, col2));
            }
        }


        return sum;
    }

    private int querySum(SegmentTreeNode root, int start, int en
d){
        if(root == null) return 0;
        if(start > root.end) return 0;
        if(end < root.start) return 0;

        start = Math.max(start, root.start);
        end = Math.min(end, root.end);
```

```
        if(start == root.start && end == root.end) return root.s
um;

        int left = querySum(root.left, start, end);
        int right = querySum(root.right, start, end);

        return left + right;
    }
}


// Your NumMatrix object will be instantiated and called as such:

// NumMatrix numMatrix = new NumMatrix(matrix);
// numMatrix.sumRegion(0, 1, 2, 3);
// numMatrix.update(1, 1, 10);
// numMatrix.sumRegion(1, 2, 3, 4);
```

Fenwick Tree (Binary Indexed Tree)

陈老师说花三天时间研究明白这个没啥意义。不过我看了小半天就看完了。。也没啥难的嘛。。。

给定 n 个元素 array，时间复杂度为

- **build O(n log n)**
- **update O(log n)**
- **query O(log n)**

Fenwick Tree 主要用于求各种维度的区间 **sum**，主要缺点在于建树时间长于 **segment tree** ，需要 **O(n log n)** 时间，还有和面试官解释的时候比较麻烦。。主要优点是好写，而且非常容易扩展到多维情况。

Youtube 视频讲解

binary index tree ，另一个教程贴在这里，还有这里，加上这个陈老师推荐的中文帖子

The two's complement of an N-bit number is defined as the complement with respect to 2^N; in other words, it is the result of subtracting the number from 2^N

Range Sum Query - Mutable

和之前那道题一样，把 segment tree 改成 fenwick tree 的写法如下：

图文并茂的视频讲解~

Idea:

- Binary representation

$$13 = 2^3 + 2^2 + 2^0$$

SUM(13) = RANGE(1, 8) + RANGE(9, 12) + RANGE(13, 13)

= 29 + 10 + 4

SUM(13) = RANGE(1, 8) + RANGE(9, 12) + RANGE(13, 13)

> - **fenwick tree** 可以用 **array** 存，纯靠 **bit manipulation** 操作。
> - **tree** 有一个 **dummy root**，所以对于单维度大小为 **n** 的输入，实际数组会在每一个维度 **+1** 的 **padding.**
> - 因此在每次更新原数组 **index** 位置的数时，在树上实际的 **update** 位置是 **index + 1.**
> - 树的 **update** 过程很像机器学习里面的 **forward/backward propagation**，每次更新之后先计算 **diff**，再把 **diff** 传导过去。因此 **fenwick tree** 有时候需要一个数组去保存所有值，用于计算 **diff.**
> - 对于给定 **index**，树的 **update** 是一个 **index** 逐渐增加的过程，相对的，树的求和是个不断寻找 **parent**，**index** 逐渐减小的过程。

在图上的树状结构中，任意一个点的 **parent** 等于其 **binary representation** 的最右一个 **1** 的 **bit** 上再 **+1;** 比如 **3 = 0011** ，**parent = 8 = 0100.**

从 **child** 到 **parent** 这条线，代表着更新。上图的结构，也是相对于更新操作的 **tree structure.**

而 **query** 是另一种结构和搜索过程，是一步一步把 **index** 的 **binary representation** 拆出来，变成 **13 = 001011** ，对应 **BIT[001000] + BIT[000010] + BIT[000001]** 的过程

```
public class NumArray {
    int[] fenwickTree;
    int length;
    int[] arr;
    public NumArray(int[] nums) {
        length = nums.length;
        arr = new int[length];
        fenwickTree = new int[length + 1];

        for(int i = 0; i < length; i++){
            update(i, nums[i]);
        }
    }

    void update(int i, int val) {
        int diff = val - arr[i];
        arr[i] = val;
        for(int index = i + 1; index <= length; index += (index
& -index)){
            fenwickTree[index] += diff;
        }
    }

    private int getSum(int i){
        int sum = 0;
        while(i > 0){
            sum += fenwickTree[i];
            i -= (i & -i);
        }
        return sum;
    }

    public int sumRange(int i, int j) {
        return getSum(j + 1) - getSum(i);
    }
}
```

Range Sum Query 2D - Mutable

Fenwick Tree 在增加维度上的优势在这题中体现的非常好。

这题如果 **int[][] nums** 直接指向 **matrix** 会在 **update** 操作之后得到错误结果，目前我还没仔细想为什么。。。稳妥起见还是老老实实新建一个吧。

增加维度的逻辑非常简单，只需要把对应的 tree array 增加维度就可以了，这时候新的getSum(i,j)所代表的是，从 (0,0) 开始到 (i,j) 的矩形范围内的 sum，相对于一维 fenwick tree 中的 (0, index) 的和。

- **fenwick tree** 本质上是树状的 **prefix sum** 数组，维度非常灵活，每一个位置上的**getSum()**都代表当前坐标到 **origin** 原点的 **cumulative sum.**
- 因此对于矩阵中任意矩形，都可以看做四个以原点为起点的矩形的相互覆盖，可以以同样的时间复杂度求解矩阵中任意位置任意形状的矩形和。

```
public class NumMatrix {
    int[][] fenwickTree;
    int[][] nums;
    int rows;
    int cols;

    public NumMatrix(int[][] matrix) {
        if(matrix == null || matrix.length == 0) return;

        rows = matrix.length;
        cols = matrix[0].length;
        fenwickTree = new int[rows + 1][cols + 1];
        nums = new int[rows][cols];

        for(int i = 0; i < rows; i++){
            for(int j = 0; j < cols; j++){
                update(i, j, matrix[i][j]);
            }
        }
    }
```

```
    public void update(int row, int col, int val) {
        int diff = val - nums[row][col];
        nums[row][col] = val;
        for(int i = row + 1; i <= rows; i += (i & -i)){
            for(int j = col + 1; j <= cols; j += (j & -j)){
                fenwickTree[i][j] += diff;
            }
        }
    }

    private int getSum(int row, int col){
        int sum = 0;
        for(int i = row; i > 0; i -= (i & -i)){
            for(int j = col; j > 0; j -= (j & -j)){
                sum += fenwickTree[i][j];
            }
        }
        return sum;
    }

    public int sumRegion(int row1, int col1, int row2, int col2)
{
        return getSum(row2 + 1, col2 + 1) + getSum(row1, col1) -
               getSum(row1, col2 + 1) - getSum(row2 + 1, col1);
    }
}
```

Range Sum Query 2D - Immutable

Range Sum Query 2D - Immutable

前缀和数组在 2D 上的应用，注意处理好 padding 和 index offset 就行了。

```
public class NumMatrix {
    int[][] dp;
    public NumMatrix(int[][] matrix) {
        if(matrix == null || matrix.length == 0) return;

        int rows = matrix.length;
        int cols = matrix[0].length;

        dp = new int[rows + 1][cols + 1];

        for(int i = 0; i < rows; i++){
            for(int j = 0; j < cols; j++){
                dp[i + 1][j + 1] = dp[i][j + 1] + dp[i + 1][j] -
 dp[i][j] + matrix[i][j];
            }
        }
    }

    public int sumRegion(int row1, int col1, int row2, int col2)
{
        return dp[row2 + 1][col2 + 1] - dp[row2 + 1][col1] - dp[
row1][col2 + 1] + dp[row1][col1];
    }
}
```

Union-Find，并查集

这组问题的分类上基本延续了九章高级算法班的大纲，加了些听课时候的笔记，和自己之后查资料的思考。

刚转专业跟公开课的时候，第一讲就是 **Robert Sedgewick** 大爷的并查集，当时还很天真的觉得这是个入门知识。。。 后来题做多了觉得这类问题掌握技巧之后确实也不难，大爷诚不我欺。

Union-Find，并查集基础

并查集是一个用于解决 disjoint sets 类问题的常用数据结构，用于处理集合中

- 元素的归属 **find**
- 集合的融合 **union**
- **Online algorithm, stream of input**
- 计算 **number of connected components**
- 不支持 **delete**

CSDN 有一个非常好的总结帖子

我做并查集问题的时候，最喜欢的方式是直接无脑撸一个 **weighted union find class** 出来，然后在具体问题上直接调用接口。。

以下是 **Princeton** 的 **Algorithm** 算法课上的样例代码，这老头可真喜欢用 **array** 啊。。。他的并查集和**KMP**算法构建方式都稍微有点麻烦。

```
public class WeightedQuickUnionPathCompressionUF {
    private int[] parent;  // parent[i] = parent of i
    private int[] size;    // size[i] = number of sites in tree
rooted at i
                           // Note: not necessarily correct if i
 is not a root node
    private int count;     // number of components

    public WeightedQuickUnionPathCompressionUF(int N) {
```

```
        count = N;
        parent = new int[N];
        size = new int[N];
        for (int i = 0; i < N; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    /**
     * @return the number of components
     */
    public int count() {
        return count;
    }


    /**
     * @return the component identifier for the component contai
ning site
     */
    public int find(int p) {
        int root = p;
        while (root != parent[root])
            root = parent[root];
        while (p != root) {
            int newp = parent[p];
            parent[p] = root;
            p = newp;
        }
        return root;
    }

   /**
     * @return true if the two sites p and qare in the same comp
onent;
     *         false otherwise
     */
    public boolean connected(int p, int q) {
        return find(p) == find(q);
```

```
    }

    /**
     * Merges the component containing site p with the
     * the component containing site q.
     *
     * @param  p the integer representing one site
     * @param  q the integer representing the other site
     */
    public void union(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        if (rootP == rootQ) return;

        // make smaller root point to larger one
        if (size[rootP] < size[rootQ]) {
            parent[rootP] = rootQ;
            size[rootQ] += size[rootP];
        }
        else {
            parent[rootQ] = rootP;
            size[rootP] += size[rootQ];
        }
        count--;
    }
```

实际操作中，用 **HashMap** 做 **parent** 和 **size** 比起数组有着不可比拟的优越性。。比如下面这题

Longest Consecutive Sequence

其实我们建的是对于每一个元素的 **1-1 mapping**，或者说是一个 元素之间的 **graph**，表示 **join** 关系。

中间有一个 [2147483646,-2147483647,0,2,2147483644,-2147483645,2147483645] 的 test case 始终出 bug，把 find 函数的返回 type 从 int 改到 Integer 就好了。

看来以后不能总是假设 int 和 Integer 是完全相等的，尤其是这种在 hashmap 里以 Integer 为 Key 的情况，要尽可能的保持类型正确。

```
public class Solution {
    private class WeightedUnionFind{
        private HashMap<Integer, Integer> parent;
        private HashMap<Integer, Integer> size;
        private int maxSize;

        public WeightedUnionFind(){}
        public WeightedUnionFind(int[] nums){
            int N = nums.length;
            parent = new HashMap<Integer, Integer>();
            size = new HashMap<Integer, Integer>();
            maxSize = 1;

            for(int i = 0; i < N; i++){
                parent.put(nums[i], nums[i]);
                size.put(nums[i], 1);
            }
        }

        private int getMaxSize(){
            return this.maxSize;
        }

        // With path compression
        public Integer find(Integer num){
            if(!parent.containsKey(num)) return null;

            Integer root = num;
            while(root != parent.get(root)){
                root = parent.get(root);
            }
            while(num != root){
                Integer next = parent.get(num);
                parent.put(num, root);
                num = next;
            }
```

```
            return root;
        }

        public void union(int p, int q){
            Integer pRoot = find(p);
            Integer qRoot = find(q);

            if(pRoot == null || qRoot == null) return;
            if(pRoot == qRoot) return;

            int pSize = size.get(pRoot);
            int qSize = size.get(qRoot);



            if(pSize > qSize){
                parent.put(qRoot, pRoot);
                size.put(pRoot, pSize + qSize);
                maxSize = Math.max(maxSize, pSize + qSize);
            } else {
                parent.put(pRoot, qRoot);
                size.put(qRoot, pSize + qSize);
                maxSize = Math.max(maxSize, pSize + qSize);
            }
        }


    }
    public int longestConsecutive(int[] nums) {
        if(nums == null || nums.length == 0) return 0;

        WeightedUnionFind uf = new WeightedUnionFind(nums);

        for(int num : nums){
            if(num != Integer.MIN_VALUE) uf.union(num, num - 1);
            if(num != Integer.MAX_VALUE) uf.union(num, num + 1);
        }

        return uf.maxSize;
    }
```

```
}
```

- 仅仅对于这题而言，还有其他的做法，比如每次看到新元素都往两边扫。不过不是这章内容的主题。
- 另一种简单易懂的做法是用 **HashMap** 动态维护“区间”，思路简洁易懂，面试遇到这题的话推荐还是用 **HashMap**，别用并查集。

```
public int longestConsecutive(int[] nums) {
    int maxLen = 0;
    if(nums == null) return maxLen;
    Set<Integer> unvisited = getSet(nums);
    for (int n : nums){
        if(unvisited.isEmpty())break;
        if(!unvisited.remove(n))continue;
        int len = 1;
        int leftOffset = -1;
        int rightOffset = 1;
        while(unvisited.remove(n+leftOffset--))len++;
        while(unvisited.remove(n+rightOffset++))len++;
        maxLen = Math.max(maxLen,len);
    }
    return maxLen;

}

private Set<Integer> getSet( int [] array){
    Set<Integer> set = new HashSet<>();
    for(int n : array) set.add(n);
    return set;
}
```

Number of Islands II

- 常犯错误：二维转一维 **index** 的时候总乘错，搞混。正确的是 **x * cols + y**，以后自己想的时候还是用 **rows / cols** 吧

- 在这题里降维成一维 **index** 是可以的，不过要注意边界处理，否则某一行的最后一个元素会连通到下一行的第一个元素上去。

```
public class Solution {
    private class WeightedUnionFind{
        HashMap<Integer, Integer> parent;
        HashMap<Integer, Integer> size;
        int count;

        public WeightedUnionFind(){
            parent = new HashMap<Integer, Integer>();
            size = new HashMap<Integer, Integer>();
            count = 0;
        }

        public Integer find(Integer index){
            if(!parent.containsKey(index)) return null;

            Integer root = index;
            while(root != parent.get(root)){
                root = parent.get(root);
            }
            while(index != root){
                Integer next = parent.get(index);
                parent.put(index, root);
                index = next;
            }
            return root;
        }

        public void union(Integer a, Integer b){
            Integer aRoot = find(a);
            Integer bRoot = find(b);
```

```
            if(aRoot == null || bRoot == null) return;
            if(aRoot.equals(bRoot)) return;

            int aSize = size.get(aRoot);
            int bSize = size.get(bRoot);

            if(aSize > bSize){
                parent.put(bRoot, aRoot);
                size.put(aRoot, aSize + bSize);
            } else {
                parent.put(aRoot, bRoot);
                size.put(bRoot, aSize + bSize);
            }
            count --;
        }

        public void add(Integer index){
            if(!parent.containsKey(index)){
                parent.put(index, index);
                size.put(index, 1);
                count ++;
            }
        }

        public int getCount(){
            return this.count;
        }

    }



    public List<Integer> numIslands2(int m, int n, int[][] posit
ions) {
        List<Integer> list = new ArrayList<Integer>();
        if(positions == null || positions.length == 0) return li
st;
        WeightedUnionFind uf = new WeightedUnionFind();

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

```
            int x = positions[i][0];
            int y = positions[i][1];
            int index = x * n + y;

            uf.add(index);

            if(x + 1 <= m - 1)   uf.union(index, (x + 1) * n + y
);
            if(x > 0)            uf.union(index, (x - 1) * n + y
);
            if(y + 1 <= n - 1)   uf.union(index, x * n + y + 1);
            if(y > 0)            uf.union(index, x * n + y - 1);

            list.add(uf.getCount());
        }

        return list;
    }

}
```

Number of Connected Components in an Undirected Graph

这题如果是把所有的 nodes 给你，其实很好做，每个点做 dfs 就好了，用 hashset 避免重复访问，毕竟是 undirected graph.

然而这题比较 gay 的地方在于。。。数据是以一个个 edge 的方式给你的，强行让你以一个 union-find 的方式一个一个节点添加，那么显然读取所有 edges 建图再去 dfs 就是很不现实的做法，而且也失去了 online algorithm 的优势。

- **对于 Integer type，要用 a.equals(b)，不要用 ==**

代码轻松愉快~

```
public class Solution {
    private class WeightedUnionFind{
```

```
        HashMap<Integer, Integer> parent;
        HashMap<Integer, Integer> size;
        int count;

        public WeightedUnionFind(){}
        public WeightedUnionFind(int n){
            parent = new HashMap<Integer, Integer>();
            size = new HashMap<Integer, Integer>();
            count = n;
            for(int i = 0; i < n; i++){
                parent.put(i, i);
                size.put(i, 1);
            }
        }

        public Integer find(Integer node){
            if(!parent.containsKey(node)) return null;

            Integer root = node;
            while(root != parent.get(root)){
                root = parent.get(root);
            }
            while(node != root){
                Integer next = parent.get(node);
                parent.put(node, root);
                node = next;
            }

            return root;
        }

        public void union(Integer p, Integer q){
            Integer pRoot = find(p);
            Integer qRoot = find(q);
            if(pRoot == null || qRoot == null) return;
            if(pRoot.equals(qRoot)) return;

            int pSize = size.get(pRoot);
            int qSize = size.get(qRoot);
```

```
            if(pSize > qSize){
                parent.put(qRoot, pRoot);
                size.put(pRoot, pSize + qSize);
            } else {
                parent.put(pRoot, qRoot);
                size.put(qRoot, pSize + qSize);
            }
            count --;
        }

        public int getCount(){
            return this.count;
        }

    }
    public int countComponents(int n, int[][] edges) {
        if(edges == null || edges.length == 0) return n;
        WeightedUnionFind uf = new WeightedUnionFind(n);

        for(int i = 0; i < edges.length; i++){
            int node1 = edges[i][0];
            int node2 = edges[i][1];

            uf.union(node1, node2);
        }

        return uf.getCount();
    }
}
```

6/7, Union-Find, 并查集应用

Graph Valid Tree

- 开始做题之前，要注意先把定义和可能的各种 **test case** 正确输出弄清楚。在 **LeetCode** 上就只能反复提交试错，但是面试的时候，做题之前这些都是要通过交流去问清楚的，要么后面会浪费很多时间去涂涂改改，甚至发现一开始就答错了。。

给定一个 Set of Nodes ，Tree 要满足两个条件：

- 无环
- 单 root 节点

把这两点搞清楚之后，这题就很简单了。先一个一个 edge 去加，如果发现有环的话直接返回 false；否则跑到最后，看看最终的 number of connected components 是不是 1.

```
public class Solution {
    private class WeightedUnionFind{
        HashMap<Integer, Integer> parent;
        HashMap<Integer, Integer> size;
```

```
        boolean hasCycle;
        int count;

        public WeightedUnionFind(int n){
            parent = new HashMap<Integer, Integer>();
            size = new HashMap<Integer, Integer>();
            hasCycle = false;
            count = n;

            for(int i = 0; i < n; i++){
                parent.put(i, i);
                size.put(i, 1);
            }
        }

        public Integer find(Integer node){
            if(!parent.containsKey(node)) return null;
            Integer root = node;
            while(root != parent.get(root)){
                root = parent.get(root);
            }
            while(node != root){
                Integer next = parent.get(node);
                parent.put(node, root);
                node = next;
            }
            return root;
        }

        public void union(Integer nodeA, Integer nodeB){
            Integer rootA = find(nodeA);
            Integer rootB = find(nodeB);
            if(rootA == null || rootB == null) return;
            if(rootA.equals(rootB)) {
                hasCycle = true;
                return;
            }

            int sizeA = size.get(rootA);
            int sizeB = size.get(rootB);
```

```
            if(sizeA > sizeB){
                parent.put(rootB, rootA);
                size.put(rootA, sizeA + sizeB);
            } else {
                parent.put(rootA, rootB);
                size.put(rootB, sizeA + sizeB);
            }
            count --;
        }

        public boolean hasCycle(){
            return this.hasCycle;
        }

        public boolean isTree(){
            return (!this.hasCycle) && (count == 1);
        }

    }

    public boolean validTree(int n, int[][] edges) {
        if(edges == null || edges.length == 0){
            if(n > 1) return false;
            else return true;
        }

        WeightedUnionFind uf = new WeightedUnionFind(n);

        for(int i = 0; i < edges.length; i++){
            int nodeA = edges[i][0];
            int nodeB = edges[i][1];
            uf.union(nodeA, nodeB);
            if(uf.hasCycle()) return false;
        }

        return uf.isTree();
    }
}
```

Surrounded Regions

第一次写的时候用了个 HashSet 记录哪些点访问过，显得麻烦，还浪费了额外空间。

- 这种在矩阵上做 **flood filling** 的问题，可以靠自定义字符做标记，取代用额外空间的记录方式。

这段代码的逻辑就是从四个边开始碰到 'O' 就往里扫，把扫到的都标上 'S' 代表有效湖；最后过一遍的时候除了 'S' 的都标成 'X' 就好了。

然而在大矩阵上 stackoverflow... 看来无脑 dfs 的做法还不够经济啊。。。

```
public class Solution {
    public void solve(char[][] board) {
        if(board == null || board.length == 0) return;
        int rows = board.length;
        int cols = board[0].length;

        for(int i = 0; i < cols; i++){
            if(board[0][i] == 'O'){
                dfs(board, 0, i);
            }
            if(board[rows - 1][i] == 'O'){
                dfs(board, rows - 1, i);
            }
        }

        for(int i = 1; i < rows - 1; i++){
            if(board[i][0] == 'O'){
                dfs(board, i, 0);
            }
            if(board[i][cols - 1] == 'O'){
                dfs(board, i, cols - 1);
            }
        }
```

```
        for(int i = 0; i < rows; i++){
            for(int j = 0; j < cols; j++){
                if(board[i][j] == 'S'){
                    board[i][j] = 'O';
                } else {
                    board[i][j] = 'X';
                }
            }
        }

    }

    private void dfs(char[][] board, int row, int col){
        if(row < 0 || row >= board.length) return;
        if(col < 0 || col >= board[0].length) return;
        if(board[row][col] != 'O') return;

        board[row][col] = 'S';

        dfs(board, row + 1, col);
        dfs(board, row - 1, col);
        dfs(board, row, col + 1);
        dfs(board, row, col - 1);
    }
}
```

写了个 BFS 的在一个中型矩阵上 TLE了，我有点方。。

```
public class Solution {
    public void solve(char[][] board) {
        if(board == null || board.length == 0) return;
        int rows = board.length;
        int cols = board[0].length;
        Queue<Integer> queue = new LinkedList<Integer>();

        for(int i = 0; i < cols; i++){
            if(board[0][i] == 'O'){
                queue.offer(i);
```

```
                bfs(board, 0, i, queue);
            }
            if(board[rows - 1][i] == 'O'){
                queue.offer((rows - 1) * cols + i);
                bfs(board, rows - 1, i, queue);
            }
        }

        for(int i = 1; i < rows - 1; i++){
            if(board[i][0] == 'O'){
                queue.offer(i * cols);
                bfs(board, i, 0, queue);
            }
            if(board[i][cols - 1] == 'O'){
                queue.offer(i * cols + cols - 1);
                bfs(board, i, cols - 1, queue);
            }
        }

        for(int i = 0; i < rows; i++){
            for(int j = 0; j < cols; j++){
                if(board[i][j] == 'S'){
                    board[i][j] = 'O';
                } else {
                    board[i][j] = 'X';
                }
            }
        }

    }

    private void bfs(char[][] board, int row, int col, Queue<Int
eger> queue){
        int rows = board.length;
        int cols = board[0].length;

        if(!isValid(row, rows, col, cols)) return;

        while(!queue.isEmpty()){
            Integer index = queue.poll();
```

```
            int x = index / cols;
            int y = index % cols;
            if(board[x][y] == 'O') board[x][y] = 'S';

            if(isValid(x + 1, rows, y, cols) && board[x + 1][y]
== 'O') queue.offer((x + 1) * cols + y);
            if(isValid(x - 1, rows, y, cols) && board[x - 1][y]
== 'O') queue.offer((x - 1) * cols + y);
            if(isValid(x, rows, y + 1, cols) && board[x][y + 1]
== 'O') queue.offer(x * cols + y + 1);
            if(isValid(x, rows, y - 1, cols) && board[x][y - 1]
== 'O') queue.offer(x * cols + y - 1);
        }

        return;
    }

    private boolean isValid(int row, int rows, int col, int cols)
{
        if(row < 0 || row >= rows) return false;
        if(col < 0 || col >= cols) return false;
        return true;
    }
}
```

同样的 DFS 逻辑，做了如下改动之后就不会 stackoverflow 了：

- DFS 时不进入最外围一圈的位置

简单来讲是在尽可能限制 DFS call stack 的层数，控制 DFS 的深度。从正确性上讲，上面的写法也是正确的，但是在极端情况下如果某一个从边沿出发的 DFS 连通了另一个边沿出发的 DFS，会导致一次的搜索路径非常长，于是 stackoverflow. 既然边沿的格子无论如何都要检查一遍，就把外圈封住，减少每个起点的 search tree depth.

```
public class Solution {
    public void solve(char[][] board) {
        if(board == null || board.length == 0) return;
        int rows = board.length;
```

```
        int cols = board[0].length;

        for(int i = 0; i < cols; i++){
            if(board[0][i] == 'O'){
                dfs(board, 0, i);
            }
            if(board[rows - 1][i] == 'O'){
                dfs(board, rows - 1, i);
            }
        }

        for(int i = 1; i < rows - 1; i++){
            if(board[i][0] == 'O'){
                dfs(board, i, 0);
            }
            if(board[i][cols - 1] == 'O'){
                dfs(board, i, cols - 1);
            }
        }

        for(int i = 0; i < rows; i++){
            for(int j = 0; j < cols; j++){
                if(board[i][j] == 'S'){
                    board[i][j] = 'O';
                } else {
                    board[i][j] = 'X';
                }
            }
        }

    }

    private void dfs(char[][] board, int row, int col){
        if(row < 0 || row >= board.length) return;
        if(col < 0 || col >= board[0].length) return;

        if(board[row][col] != 'O') return;

        board[row][col] = 'S';
```

```
        // DFS 时不进入最外圈
        if(row + 2 < board.length && board[row + 1][col] == 'O') dfs(board, row + 1, col);
        if(row - 2 >= 0 && board[row - 1][col] == 'O') dfs(board, row - 1, col);
        if(col + 2 < board[0].length && board[row][col + 1] == 'O') dfs(board, row, col + 1);
        if(col - 2 >= 0 && board[row][col - 1] == 'O') dfs(board, row, col - 1);
    }
}
```

这题当然也可以用 Union-Find 写，先把所有最外圈的 boundry 连上，然后把里面的相邻 'O' 做 union，最后扫矩阵的时候，如果对应的 root 不是 boundry root 就留下，不然都改成 'X'.

不过只是在这个问题上，不是很简洁。

动态规划

这组问题的分类上基本延续了九章高级算法班的大纲，加了些听课时候的笔记，和自己之后查资料的思考。

动态规划问题的种类更杂，一般来讲也更 **tricky** 一些。因为这类问题建立在对问题的状态空间和子问题结构有清晰的认识，并且能正确定义和缓存子问题结果上。即使最后的代码看起来可能只是几个循环，但是从复杂程度上大多数要高于暴力搜索和二叉树。

见到题基本的判断，用 **DP** 做的题大多返回 **boolean / int** ，求 **Max /Min** ，而且不能打乱原输入顺序。

动态规划有两个重要定义，一个叫 **"optimal substructure"**，另一个叫 **"overlap subproblem".**

各种排序 / Tree 类问题中，都会用到 divide & conquer 的思想，去把问题分成若干个 **"disjoint"** subproblems，然后递归解决。

"Disjoint" subproblem 在 Tree 类问题上体现的最为明显，左子树是左子树的问题，右子树是右子树的问题。因此 Tree 类问题上，更多的是解决“**disjoint subproblem** 的整合” 还有 “非连续 **subproblem** 的处理”。

而动态规划的中心思想是，面对 search tree 里都是 **"overlap subproblem"**，如何根据问题结构制定缓存策略，避免重叠问题重复计算。

根据**CLRS**，动态规划分为两种：

- **top-down with memoization (**递归记忆化搜索**)**
    - 等价于带缓存的，搜索树上的 **DFS**
    - 比较贴近于新问题正常的思考习惯

- **bottom-up (自底向上循环迭代)**
  - 以 **"reverse topological order"** 处理
  - 每个子问题下面依赖的所有子问题都算完了才开始计算当前
  - 一般依赖于子问题之间天然的 **"size"** 联系

两种解法 big-O 时间复杂度一致，bottom-up 的 constant factor 小一些。

Figure 15.3 The recursion tree showing recursive calls resulting from a call CUT-ROD(p, n) for $n = 4$. Each node label gives the size n of the corresponding subproblem, so that an edge from a parent with label s to a child with label t corresponds to cutting off an initial piece of size $s - t$ and leaving a remaining subproblem of size t. A path from the root to a leaf corresponds to one of the 2^{n-1} ways of cutting up a rod of length n. In general, this recursion tree has 2^n nodes and 2^{n-1} leaves.

Rod-Cutting 的"subproblem graph"（子问题图），可以看到如果以 "subtree" 为单位定义 "subproblem" 的话，这个子问题图的结构有非常多的 "identical subtree" ，也即多个 "overlap subproblem". 这样结构的子问题图有 2^N 的节点和 2^(N-1) 个叶节点，不做任何记忆化搜索的话必然是指数级时间复杂度。

因此，动态规划的时间复杂度分析和子问题图 G = (V,E) 的结构息息相关。V = Vertex，代表每一个子问题； E = Edge，代表子问题之间结构。一般来讲，解决一个子问题的时间复杂度和其对应 V 的 out-degree 成正比；子问题的数量等于 V 的数量，既然每个子问题只被计算一次，在整个子问题图上做动态规划的复杂度，和 O(E + V) 成正比。

动态规划的另一个要素是 **"optimal substructure":**

- **A problem is said to have optimal substructure if an optimal solution can be constructed efficiently from optimal solutions of its subproblems.**
- **optimal substructure** 指，一个问题的最优解，可以由其子问题的最优解构造出来。

What is optimal/non-optimal substructure? A problem is said to have optimal substructure if an optimal solution can be constructed efficiently from optimal solutions of its subproblems. This property is used to determine the usefulness of dynamic programming and greedy algorithms for a problem What is local and global optimum? Local optimum of an optimization problem is a solution that is optimal (either maximal or minimal) within a neighboring set of candidate solutions. Global optimum - is the optimal solution among all possible solutions, not just those in a particular neighborhood of values. How to prove that Greedy algorithm yields global optimum? Usually, global optimum can be proven by induction. **Typically, a greedy algorithm is used to solve a problem with optimal substructure if it can be proved by induction that this is optimal at each step.** Otherwise, providing the problem exhibits overlapping subproblems as well, dynamic programming is used. To prove that an optimization problem can be solved using a greedy algorithm, we need to prove that the problem has the following:

- Optimal substructure property: an optimal global solution contains the optimal solutions of all its subproblems.
- Greedy choice property: a global optimal solution can be obtained by greedily selecting a locally optimal choise.
- Matroids can be used as well in some case used to mechanically prove that a particular problem can be solved with a greedy approach.

You will find yourself following a common pattern in discovering optimal substructure:

1. You show that a solution to the problem consists of making a choice, such as choosing an initial cut in a rod or choosing an index at which to split the matrix chain. Making this choice leaves one or more subproblems to be solved.
2. You suppose that for a given problem, you are given the choice that leads to an optimal solution. You do not concern yourself yet with how to determine this choice. You just assume that it has been given to you.
3. Given this choice, you determine which subproblems ensue and how to best characterize the resulting space of subproblems.
4. You show that the solutions to the subproblems used within an optimal solution to the problem must themselves be optimal by using a “cut-and-paste” technique. You do so by supposing that each of the subproblem solutions is not optimal and then deriving a contradiction. In particular, by “cutting out” the nonoptimal solution to each subproblem and “pasting in” the optimal one, you show that you can get a better solution to the original problem, thus contradicting your supposition that you already had an optimal solution. If an optimal solution gives rise to more than one subproblem, they are typically so similar that you can modify the cut-and-paste argument for one to apply to the others with little effort.

6/20, 动态规划，定义 & 性质

House Robber

题本身不难，Easy 难度。

先从 Top-Down 的角度来想，如果我们定义 maxProfit(n) 为长度为 n 的 array 中所能得到的最大利益的话，不难看出在计算 maxProfit(n) 的时候，它的值只和前两个 subproblem 相关，即 maxProfit(n - 1) 和 maxProfit(n - 2).

由此我们发现了 **DP** 的第一个性质 ： **overlap subproblems.**

同时如果 maxProfit(n - 1) 和 maxProfit(n - 2) 都是其对应子问题的最优解的话，我们可以只利用这两个子问题的 solution，加上对当前元素的判断，构造出 maxProfit(n) 的最优解。

因此我们满足了 **DP** 的正确性性质**: optimal substructure.**

此题的另外考点在于空间的优化。因为每一个 **size = n** 的问题只和前面的 **2** 个子问题相关，我们显然不需要存储所有的状态，而可以用滚动数组优化空间占用。 滚动数组的简单用法就是**“**膜**”**（**=**。**=**），即对 **dp[]** 的 **index** 取 **mod**，

除数等于需要保存的状态数。

在数组 **index % mod** 的做法其实相当于做了一个 **circular buffer,** 使得固定长度的数组收尾相接，依序覆盖。

题外话，**cicular buffer** 很适合用于实现 **fixed-size queue**，一个 **java** 实现看一看这个帖子

```
public class Solution {
    public int rob(int[] nums) {
        if(nums == null || nums.length == 0) return 0;
        if(nums.length == 1) return nums[0];
        if(nums.length == 2) return Math.max(nums[0], nums[1]);

        int[] dp = new int[2];
        dp[0] = nums[0];
        dp[1] = Math.max(nums[0], nums[1]);

        for(int i = 2; i < nums.length; i++){
            dp[i % 2] = Math.max(dp[(i - 2) % 2] + nums[i], dp[(
i - 1) % 2]);
        }

        return dp[(nums.length - 1) % 2];

    }
}
```

House Robber II

这题考察环形数组中，**subproblem** 的拆分与覆盖。

数组变成环形之后，就需要考虑首尾相接的问题了~ 理论上说，对于长度为 n 的环形数组，我们现在有了 n 种不同的切分方式，去处理长度为 n 的线性数组。

不过我们不需要考虑所有的可能切分，因为只有相邻元素之间会出现问题。我们的 **subproblem** 不能再以 **size = n** 开始 **top-down**了，因为无法保证正确性； 但是 **size = n - 1** 的所有 **subproblems**，一定是正确的。我们只需要考虑，如何拆分 **size = n - 1** 的 **subproblems**，并且如何用他们构造出全局最优解。

只需要把环形数组拆分成两个头尾不同的 **size = n - 1** 的 **subproblems** 就可以了：

【1, 7, 5, 9, 2】

下面的 **subproblem** 覆盖了所有不抢最后一座房子的 **subproblems**；

【(1, 7, 5, 9) 2】

如下的 **subproblem** 覆盖了所有不抢第一座房子的 **subproblems**；

【1,(7, 5, 9, 2)】

如果最后的最优解第一座和最后一座房子都不抢，也一定会被包含在左右两个 **subproblem** 的范围内，因为其 **size = n - 2;**

【1, (7, 5, 9), 2】

由于我们不能同时抢第第一和最后一座房子，上面两个 **overlap subproblem** 一定覆盖了所有子问题的最优解，并且符合全局最优解的 **optimal substructure**，保证了算法的正确性。

易错点： 后半段数组的 **index offset**，合理的处置是用完全一样的 **for loop**，只在实际取元素的时候做 **nums[i + 1]**，不然计算后半段以 **i = 3** 开始时，覆盖的 **dp[]** 位置是错的。

```
public class Solution {
    public int rob(int[] nums) {
        if(nums == null || nums.length == 0) return 0;
        if(nums.length == 1) return nums[0];
        if(nums.length == 2) return Math.max(nums[0], nums[1]);

        int leftMax = helper(nums, 0);
        int rightMax = helper(nums, 1);

        return Math.max(leftMax, rightMax);
    }

    private int helper(int[] nums, int start){
        int max = 0;
        int[] dp = new int[2];
        dp[0] = nums[start];
        dp[1] = Math.max(nums[start], nums[start + 1]);

        for(int i = 2; i < nums.length - 1; i++){
            dp[i % 2] = Math.max(dp[(i - 1) % 2], dp[(i - 2) % 2
] + nums[i + start]);
        }

        return dp[(nums.length - 2) % 2];
    }
}
```

House Robber III

这题其实很像 **POJ 2342 Anniversary party** ，区别在于这里的树只有两叉，结构简单很多。

回顾下这章一开始对动态规划的总结，可以很容易看出来，这题的左右子树 **subproblem** 是 **disjoint.**

因此这道题只有 **optimal substructure**，而没有 **overlap subproblems.**

先贴一段错误的代码，思路就是用 level order 遍历整棵树，存下 curSum ，然后用 House Robber 一样的思路去做 DP. 错误的原因在于，不应该直接舍弃掉一层，因为层与层之间并不是 fully connected，比如[2,1,3,null,4]，最优解是相邻两层上不相连的两个 node.

```
public class Solution {
    public int rob(TreeNode root) {
        if(root == null) return 0;

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

        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        if(root != null) queue.offer(root);
        while(!queue.isEmpty()){
            int size = queue.size();
            int curSum = 0;
            for(int i = 0; i < size; i++){
                TreeNode node = queue.poll();
                curSum += node.val;
                if(node.left != null) queue.offer(node.left);
                if(node.right != null) queue.offer(node.right);
            }
            lvlSum.add(curSum);
        }

        if(lvlSum.size() == 1) return lvlSum.get(0);
        if(lvlSum.size() == 2) return Math.max(lvlSum.get(0), lv
lSum.get(1));

        int[] dp = new int[2];
        dp[0] = lvlSum.get(0);
        dp[1] = Math.max(lvlSum.get(0), lvlSum.get(1));

        for(int i = 2; i < lvlSum.size(); i++){
            dp[i % 2] = Math.max(dp[(i - 1) % 2], dp[(i - 2) % 2
] + lvlSum.get(i));
        }


        return dp[(lvlSum.size() - 1) % 2];

    }
}
```

这道题只有 **DP** 的一个性质： **optimal substructure**，即用 **subproblem (subtree)** 的最优解可以构造出全局最优解。**optimal substructure**性质在 **Tree** 类问题中非常常见，因此遇到一个问题的时候，要注意按照其性质属性仔细辨别正确解法。

在 **Tree** 上做递归的时候返回顺序已然是 **bottom-up**，相对于每一个节点，并没有重复计算，也没有 **overlap subproblems**，因此这只是一个正常的递归搜索问题，而不需要依赖 **DP** 进行优化。

dfs 时间复杂度已经是 **O(n) , n = # of nodes**

```
public class Solution {
    public int rob(TreeNode root) {
        if(root == null) return 0;
        if(root.left == null && root.right == null) return root.
val;


        int leftLvl = 0, rightLvl = 0;
        int subleftLvl = 0, subrightLvl = 0;

        if(root.left != null){
            leftLvl = rob(root.left);
            subleftLvl = rob(root.left.left) + rob(root.left.rig
ht);
        }
        if(root.right != null){
            rightLvl = rob(root.right);
            subrightLvl = rob(root.right.left) + rob(root.right.
right);
        }

        return Math.max(subleftLvl + subrightLvl + root.val, lef
tLvl + rightLvl);
    }
}
```

漆栅栏，漆墙，再漆房

Paint Fence

惭愧啊，想了半天自己也没想到好解法，觉得自己还没把这类 array DP 的问题做熟。

其实这题和 House robber 还有 Fibonacci Number 非常像， 都是 T(n) 取决于 T(n - 1) 和 T(n - 2)，在 House Robber 里，我们在取值上有一个连续性的制约：不能偷位置连续的房子；在这题里，我们刷当前位置的时候，也有颜色上的制约，我们需要的是做些思考，把这个制约找出来。

在刷第 **n** 块栅栏颜色的时候，要么要和 **n - 1** 的颜色不一样，要么和 **n - 2** 的颜色不一样。

满足 “颜色不一样” 的选择，显然是 k - 1 种，于是

- **T[n] = (k - 1) * (T[n - 1] + T[n - 2]);**

```
public class Solution {
    public int numWays(int n, int k) {
        if(n == 0) return 0;
        if(n == 1) return k;
        if(n == 2) return k * k;

        int[] dp = new int[3];
        dp[0] = 0;
        dp[1] = k;
        dp[2] = k * k;


        for(int i = 3; i <= n; i++){
            dp[i % 3] = (k - 1) * (dp[(i - 1) % 3] + dp[(i - 2)
% 3]);
        }

        return dp[n % 3];
    }
}
```

Paint House

吸取了上一题的教训之后，做这题时候思路就明朗多了。

我们当前的涂漆选择和最小价值都取决于前一块，首先颜色不能一样，其次两种颜色的选择中，我们要选总 cost 最小的。

一种最简单直接的思路就是，开三个数组，分别代表着以当前位置结尾，刷 "RGB" 所能得到的最小 cost，往复循环即可，因为只需要保存两个状态，所以时间复杂度也是 O(1).

要注意可能会有在当前位置上两种颜色的涂漆 cost 一样的情况，不能随意选择，因为可能下一个位置上就会出现差异。

类似 best time to buy and sell stock with cooldown，dp[] 的数量等于操作与状态的数量，代表着“由此操作结尾”。

```
public class Solution {
    public int minCost(int[][] costs) {
        if(costs == null || costs.length == 0) return 0;

        int n = costs.length;

        int[] minRed = new int[2];
        int[] minBlue = new int[2];
        int[] minGreen = new int[2];

        minRed[0] = 0; minBlue[0] = 0; minGreen[0] = 0;

        for(int i = 1; i <= n; i++){
            minRed[i % 2] = costs[i - 1][0] + Math.min(minBlue[(
i - 1) % 2], minGreen[(i - 1) % 2]);
            minBlue[i % 2] = costs[i - 1][1] + Math.min(minRed[(
i - 1) % 2], minGreen[(i - 1) % 2]);
            minGreen[i % 2] = costs[i - 1][2] + Math.min(minBlue
[(i - 1) % 2], minRed[(i - 1) % 2]);
        }

        return Math.min(minRed[n % 2], Math.min(minBlue[n % 2],
minGreen[n % 2]));
    }
}
```

论坛中另一种巧妙但是比较偷的解法是，直接在 cost[][] 上做更新，想法不错，可能的问题是，我们破坏了原有的 cost[][] 信息。

```
public class Solution {
public int minCost(int[][] costs) {
    if(costs==null||costs.length==0){
        return 0;
    }
    for(int i=1; i<costs.length; i++){
        costs[i][0] += Math.min(costs[i-1][1],costs[i-1][2]);
        costs[i][1] += Math.min(costs[i-1][0],costs[i-1][2]);
        costs[i][2] += Math.min(costs[i-1][1],costs[i-1][0]);
    }
    int n = costs.length-1;
    return Math.min(Math.min(costs[n][0], costs[n][1]), costs[n][
2]);
}
```

(FB) Paint House II

O(n * k * k) 的算法非常直观，顺着上一题的思路就行了。

要注意这题多了一个条件：The cost of painting each house with a certain color is different，换句话说，不用再担心当前的 cost 一样，需要记录每一步不同颜色的 cost 确保算法正确性的问题了。

但是这不意味着，贪心法就是正确的，因为 localMin 不意味着 gloalMin，在当前位置选择总 cost 最低的选择，有可能会遇到下一步所有其他颜色 cost 都特别高，但是却已经不能再选同样颜色的境地。

居然一次 AC .. 虽然说代码比较粗糙。

- dp[i][j] = 第 i 个房子刷 j 号漆的 min cost
- dp[i][j] 的值是 当前位置刷 j 号漆的花费 cost[i][j] (忽略 off by one) 加上前一个栅栏dp[i - 1][] 里，所有不用 j 号漆的最小值。
- 暴力解法的话，对于每一个 j 都需要扫前一层的 j - 1 个值去找最小，来更新每个位置的 minCost Except itself，如图所示；

-

- 然而我们可以用类似 product of array except itself 的思路，去做对于每一个位置 i, left[] 和 right[] 的最小值，这样我们就可以用 O(k) 的时间和 O(k) 空间完成新一轮的 min cost except itself 数组更新。

时间复杂度 O(nk) ，空间复杂度 O(nk)，用时 6ms，超过 58.27%.

```
    public int minCostII(int[][] costs) {
        if(costs == null || costs.length == 0) return 0;

        int n = costs.length;
        int k = costs[0].length;
        int[][] minCost = new int[n][k];

        int[] minExceptSelf = new int[k];

        for(int i = 0; i < n; i++){
            int[] leftMin = new int[k + 1];
            int[] rightMin = new int[k + 1];

            leftMin[0] = Integer.MAX_VALUE;
```

```
            rightMin[k] = Integer.MAX_VALUE;

            for(int j = 0; j < k; j++){
                minCost[i][j] = minExceptSelf[j] + costs[i][j];
            }

            for(int j = 1; j <= k; j++){
                leftMin[j] = Math.min(leftMin[j - 1], minCost[i]
[j - 1]);
            }
            for(int j = k - 1; j >= 0; j--){
                rightMin[j] = Math.min(rightMin[j + 1], minCost[
i][j]);
            }
            for(int j = 0; j < k; j++){
                minExceptSelf[j] = Math.min(leftMin[j], rightMin
[j + 1]);
            }
        }

        int min = Integer.MAX_VALUE;
        for(int i = 0; i < k; i++){
            min = Math.min(min, minCost[n - 1][i]);
        }

        return min;
    }
```

except self array 的构造和 **index ~** 我悲剧的发现横着写多了换成竖着的有点反应不过来。。面试还是老老实实写横着的吧。

参考论坛之后，这题还有更巧妙的 O(1) 空间做法，时间复杂度依然是 O(nk) 但是常数系数更低。

- 保存之前的 1st 小cost和其颜色；
- 保存之前的 2nd 小cost;
- 扫新位置的时候，如果发现试图染色的编号 j == 1st 小的 index, 就直接拿 2nd 小 cost; 其他照旧。
- 扫新一轮循环的时候，依次更新几个变量
- 我们只需要保存两层循环中，6个变量的值就可以了。

可以看到，这个算法的正确性依赖于 **“每个位置没有重复 cost”**，我们才得以只保存两个值和一个指针就能保证正确性。

如果 **cost** 可以重复的话，还得靠前面那个解法。

```
public class Solution {
    public int minCostII(int[][] costs) {
        if(costs == null || costs.length == 0) return 0;

        int n = costs.length;
        int k = costs[0].length;

        int prevMin = 0;
        int prevMinPtr = -1;
        int prevSecMin = 0;

        for(int i = 0; i < n; i++){
            int curMin = Integer.MAX_VALUE;
            int curMinPtr = -1;
            int curSecMin = Integer.MAX_VALUE;

            for(int j = 0; j < k; j++){
                int val = (j == prevMinPtr ? prevSecMin : prevMi
n) + costs[i][j];

                if(val < curMin){
                    curSecMin = curMin;
                    curMin = val;
                    curMinPtr = j;
                } else if (val < curSecMin){
                    curSecMin = val;
                }
            }

            prevMin = curMin;
            prevMinPtr = curMinPtr;
            prevSecMin = curSecMin;
        }

        return prevMin;
    }
}
```

6/24, 动态规划，矩阵路径，滚动数组

潜水归来。刷起吧。

待刷类别: 记忆化搜索DP，博弈类DP，区间类DP，背包类DP，划分类DP

Unique Paths II

挺经典的 2D-array 上做 DP 问题。

```
public class Solution {
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {
        if (obstacleGrid == null || obstacleGrid.length == 0) re
turn 0;

        int m = obstacleGrid.length, n = obstacleGrid[0].length;
        int[] dp = new int[n + 1];
        dp[1] = 1;

        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (obstacleGrid[i - 1][j - 1] == 1) dp[j] = 0;
                else dp[j] = dp[j] + dp[j - 1];
            }
        }
        return dp[n];
    }
}
```

Minimum Path Sum

思路一样，都属于 2D-array 上的路径问题。考虑到没有负数，也不存在往回走的情况，因此这个问题的 optimal substructure 比较简单，只依赖于来路的两个位置。

```
public class Solution {
    public int minPathSum(int[][] grid) {
        int rows = grid.length;
        int cols = grid[0].length;

        for(int i = 1; i < cols; i++){
            grid[0][i] += grid[0][i - 1];
        }
        for(int i = 1; i < rows; i++){
            grid[i][0] += grid[i - 1][0];
        }

        for(int i = 1; i < rows; i ++){
            for(int j = 1; j < cols; j++){
                grid[i][j] = Math.min(grid[i - 1][j], grid[i][j
- 1]) + grid[i][j];
            }
        }

        return grid[rows - 1][cols - 1];
    }
}
```

Maximal Square

一年前不熟悉动态规划去硬想的时候，困扰了我有段时间的题。

首先不难判断出，这是一个在给定 2D-array 上直接做 DP 的问题，有天然的 overlap subproblems，因而重点在于 states 之间的关联和构造，寻找 optimal substructure.

正方形在 2D-array 上的定位相对简单，只需要有一个顶点，加上边长就可以，其中对于所有正方形，顶点位置固定。既然矩阵 DP 大多从左上向右下扫，用正方形右下角定位比较简便。

对于每一个位置 (i,j):

- 其边长向左扩展受限于 (i, j-1);

- 其边长向上扩展受限于 **(i-1, j);**
- 其边长向对角线扩展受限于**(i-1, j-1);**

因此，以**(i,j)** 为右下角的最大正方形边长，取决于以上三个点。**Optimal substructure** 确定。

剩下的优化就是滚动数组了，我们只需要保存前面一行以及当前行的左面，因此 int[2][cols].

循环初始化的细节要注意下，此外还要注意取**mod**的时候 **%** 是在最后，而不是 **(i % 2) - 1**

```
public class Solution {
    public int maximalSquare(char[][] matrix) {
        if(matrix == null || matrix.length == 0) return 0;
        int rows = matrix.length;
        int cols = matrix[0].length;
        int max = 0;

        int[][] dp = new int[2][cols];

        for(int i = 0; i < cols; i++){
            dp[0][i] = matrix[0][i] - '0';
            max = Math.max(dp[0][i], max);
        }

        for(int i = 1; i < rows; i++){
            dp[i % 2][0] = matrix[i][0] - '0';
            for(int j = 1; j < cols; j++){
                if(matrix[i][j] == '0'){
                    dp[i % 2][j] = 0;
                } else {
                    dp[i % 2][j] = Math.min(dp[i % 2][j - 1],
                                   Math.min(dp[(i - 1) % 2][j],
                                            dp[(i - 1) % 2][j - 1
])) + 1;
                }
                max = Math.max(dp[i % 2][j], max);
            }
        }

        return max * max;
    }
}
```

Maximal Square Follow Up

01矩阵里面寻找一个对角线为 1， 其他全为 0 的矩阵

这题 LintCode 上还没有，所以就自己开 IDE 写，自己测 test case 了。思路和上一题类似。

由于连续对角线构造的依然是正方形，我们可以以一样的状态定义和转移方程来定义 (i,j) = 以 (i, j) 为起点的最大 1 对角线正方形。

对于(i , j)，我们可以通过(i - 1, j - 1) 来判断在对角线方向可以延伸多长；然而同时也要保证在 (i, j) 左面的宽为1的矩形全部是 0，上面宽为1的矩形也全部是0，其长度至少要等于 (i - 1, j - 1) 上的对角线长度。如果条件都满足，则可以继续 extend.

因此从 optimal substructure 来讲，和 Maximal Square 非常相似，都是从【左】【上】【对角线】来寻找能 valid 延伸的最长距离，其中最短的那个作为当前位置的 bottleneck.

我们需要构造额外的两个 DP 矩阵，用于记录对于每个位置 (i, j) 能向上和向左延伸的最长连续 0 长度。 O(mn) 时间， O(mn) 空间。

leftLen 和 upLen 也可以用滚动数组优化，不过貌似只能优化其中一个。

```
public class Main {

    private static int maximalDiagnoal(int[][] matrix){
        if(matrix == null || matrix.length == 0) return 0;
        int rows = matrix.length;
        int cols = matrix[0].length;

        int[][] leftLen = new int[rows][cols];
        int[][] upLen = new int[rows][cols];

        int max = 0;

        for(int i = 0; i < cols; i++){
            if(matrix[0][i] == 0) upLen[0][i] = 1;
            max = Math.max(max, matrix[0][i]);
        }
        for(int i = 0; i < rows; i++){
            if(matrix[i][0] == 0) leftLen[i][0] = 1;
            max = Math.max(max, matrix[i][0]);
        }
```

```
        for(int i = 1; i < rows; i++){
            for(int j = 1; j < cols; j++){
                if(matrix[i][j] == 0){
                    leftLen[i][j] = leftLen[i][j-1] + 1;
                    upLen[i][j] = upLen[i-1][j] + 1;
                }
                max = Math.max(max, matrix[i][j]);
                // By default 0
            }
        }

        int[][] dp = new int[2][cols];

        for(int i = 0; i < cols; i++){
            dp[0][i] = matrix[0][i];
        }

        for(int i = 1; i < rows; i++){
            dp[i % 2][0] = matrix[i][0];
            for(int j = 1; j < cols; j++){
                if(matrix[i][j] == 1){
                    int diag = dp[(i-1) % 2][j-1];
                    int left = leftLen[i][j-1];
                    int up = upLen[i-1][j];

                    int min = Math.min(diag, Math.min(left, up))
;

                    dp[i % 2][j] = min + 1;
                    if(min + 1 > max){
                        max = min + 1;
                        System.out.println("Row : " + i + "  Col
 : " + j);
                        System.out.println("Current Max : " + (m
in + 1));
                    }
                }
            }
        }
```

```
        return max;
    }

    public static void main(String[] args){

        int[][] matrix1 =  {{0,1,1,0,0,1},
                            {1,0,0,1,0,1},
                            {0,1,1,0,0,1},
                            {1,1,0,1,0,1},
                            {1,0,0,0,1,0}};

        int[][] matrix2 =  {{1,0,0,0,0,1},
                            {0,1,0,0,0,1},
                            {0,0,1,0,0,1},
                            {0,0,0,1,0,1},
                            {0,0,0,0,1,0}};

        int[][] matrix3 =  {{1,0,0,0,0,1},
                            {0,1,0,0,0,1},
                            {0,0,1,0,0,1},
                            {0,0,0,1,0,1},
                            {0,0,1,0,1,0}};
        testRun(matrix1);
        testRun(matrix2);
        testRun(matrix3);

    }

    private static void testRun(int[][] matrix){
        for(int i = 0; i < matrix.length; i++){
            for(int j = 0; j < matrix[0].length; j++){
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();
        }

        System.out.println("Max square with 1's only at diagnal
: " + maximalDiagnoal(matrix));
    }
```

```
}
```

Maximal Rectangle

这题显然就比 Maximal Square 复杂了，因为只有一个边长无法确定矩形位置，更无法确定面积。因为多了一个变量，所以我们的 DP 要多加一个维度，dp[i][j][k] = 以(i, j) 为右下角，底边长度为 k 的最大矩形高度。时间复杂度为 O(n^3)

我不太建议也不太觉得面试会考这种三维的 DP... 这题其实更适合用 maximal histogram 的思路去做，等我之后再回来搞定这题。

6/24, 动态规划，记忆化搜索

Longest Increasing Continuous Subsequence

从左到右扫一遍，再从右往左扫一遍就可以了。。。这题只是后面记忆化搜索的引子。

```
public class Solution {
    /**
     * @param A an array of Integer
     * @return  an integer
     */
    public int longestIncreasingContinuousSubsequence(int[] A) {
        // Write your code here
        if(A == null || A.length == 0) return 0;
        int max = 1;
        int cur = 1;
        for(int i = 1; i < A.length; i++){
            if(A[i] > A[i - 1]){
                cur ++;
                max = Math.max(max, cur);
            } else {
                cur = 1;
            }
        }

        cur = 1;
        for(int i = A.length - 2; i >= 0; i--){
            if(A[i] > A[i + 1]){
                cur ++;
                max = Math.max(max, cur);
            } else {
                cur = 1;
            }
        }

        return max;
    }
}
```

Longest Increasing Path in a Matrix

去年的 **Google** 高频题，著名的 **POJ 1088 -** 滑雪。当时在论坛里看到这道题的时候，觉得完全没思路，也不知道自己什么时候能独立当场把这道题做出来。

如今一次**AC**的感觉，真爽啊。

这题是 **2D-array** 上的搜索问题，参考 **number of islands** 和 **word search.**

- **Optimal Substructure**
  - 全局最优解一定由从起点 **maxPath(i , j)** 和其相邻的有效起点 **maxPath(i', j')** 的 **subproblem** 构造而成
  - 由于最优解 **path** 对单调性有要求，因此无需修改原矩阵，也无需担心下一个位置回头造成死循环
- **Overlap Subproblems**
  - 对于每一个起点 **maxPath(i , j)** ，都需要 **top-bottom** 的递归解决其下一个有效位置 **maxPath(i', j')** 的子问题，并且高度覆盖。

```
public class Solution {
    public int longestIncreasingPath(int[][] matrix) {
        if(matrix == null || matrix.length == 0) return 0;
        int rows = matrix.length;
        int cols = matrix[0].length;
        int[] dp = new int[rows * cols];

        int max = 0;
        for(int i = 0; i < rows; i++){
            for(int j = 0; j < cols; j++){
                max = Math.max(max, memoizedSearch(i, j, cols, r
ows, matrix, dp));
            }
        }

        return max;
    }

    private int memoizedSearch(int x, int y, int cols, int rows,
```

```
                                int[][] matrix, int[] dp){
        if(x < 0 || x >= rows) return 0;
        if(y < 0 || y >= cols) return 0;

        int index = x * cols + y;
        if(dp[index] != 0) return dp[index];

        int left = 0, right = 0;
        int top = 0, bottom = 0;

        if(y - 1 >= 0 && matrix[x][y - 1] > matrix[x][y]) left =
 memoizedSearch(x, y - 1, cols, rows, matrix, dp);
        if(y + 1 < cols && matrix[x][y + 1] > matrix[x][y]) righ
t = memoizedSearch(x, y + 1, cols, rows, matrix, dp);
        if(x - 1 >= 0 && matrix[x - 1][y] > matrix[x][y]) top =
memoizedSearch(x - 1, y, cols, rows, matrix, dp);
        if(x + 1 < rows && matrix[x + 1][y] > matrix[x][y]) bott
om =memoizedSearch(x + 1, y, cols, rows, matrix, dp);

        int maxLR = Math.max(left, right) + 1;
        int maxTB = Math.max(top, bottom) + 1;
        int max = Math.max(maxLR, maxTB);

        dp[index] = max;

        return max;
    }
}
```

6/24, 动态规划，博弈类 DP

Coins in a Line

这题当然有取巧的 % 3 做法，不过不是很适合 generalize 到其他问题上。

Optimal Substructure

- 全局解 **win(n)** 取决于 **win(n - 1)** 和 **win(n - 2)** 的解，可以由 **subproblem** 的解构造出全局解。

Overlap Subproblem

- 就像 **Fibonacci Number** 结构一样。

```
public class Solution {
    /**
     * @param n: an integer
     * @return: a boolean which equals to true if the first play
er will win
     */
    public boolean firstWillWin(int n) {
        // write your code here
        if(n == 0) return false;
        if(n <= 2) return true;

        boolean[] dp = new boolean[n + 1];
        dp[0] = false;
        dp[1] = true;
        dp[2] = true;

        for(int i = 3; i <= n; i++){
            dp[i] = (!dp[i - 1]) || (!dp[i - 2]);
        }

        return dp[n];
    }
}
```

Coins in a Line II

当存在“价值”之后，这题就是比较典型的博弈问题了，在 AI 里最常用的是 MiniMax 算法，如图所示，对于当前玩家来讲，会在当前的选择中选择 Max Profit; 而下一层对手的回合对手会选择留下 Min Profit 的走法。

按照 MiniMax 算法的思路，游戏策略如下：

当前还有 **n** 个硬币时 **F(n)**，每一步都可以选择拿 **1** 或 **2** 个硬币；

- 自己拿 **1** 个，对手子问题**F(n - 1)**：
  - 对手拿 **1** 个 **-** 自己下一步为**F(n - 2)**；
  - 对手拿 **2** 个 **-** 自己下一步为**F(n - 3);**
- 自己拿 **2** 个，对手子问题**F(n - 2)**：
  - 对手拿 **1** 个 **-** 自己下一步为**F(n - 3)**；
  - 对手拿 **2** 个 **-** 自己下一步为**F(n - 4);**

其中因为隔了一步对手的回合，自己的下一个 **subproblem** 值一定为两种可能中最小的一个。而对于本回合，自己要取拿 **1** 或 **2** 个硬币中受益最大的选择。

一次处理一个回合的 **MiniMax** 过程。

```
public class Solution {
    /**
     * @param values: an array of integers
```

```
     * @return: a boolean which equals to true if the first play
er will win
     */
    public boolean firstWillWin(int[] values) {
        // write your code here
        int n = values.length;
        if(n <= 2) return true;

        int[] dp = new int[n + 1];
        Arrays.fill(dp, -1);
        dp[0] = 0;
        dp[1] = values[n - 1];
        dp[2] = values[n - 1] + values[n - 2];

        int sum = 0;
        for(int num : values){
            sum += num;
        }

        return 2 * memoizedSearch(n, values, dp) > sum;
    }

    private int memoizedSearch(int coins, int[] values, int[] dp)
{
        if(coins < 0) return 0;
        if(dp[coins] != -1) return dp[coins];
        int n = values.length;

        int oneMax = values[n - coins] + Math.min(memoizedSearch
(coins - 2, values, dp),
                                                  memoizedSearch
(coins - 3, values, dp));

        int twoMax = values[n - coins] + values[n - coins + 1]
                     + Math.min(memoizedSearch(coins - 3, values
, dp),
                                memoizedSearch(coins - 4, values
, dp));

        dp[coins] = Math.max(oneMax, twoMax);
```

```
        return dp[coins];
    }
}
```

Coins in a Line III

深入理解了上一道题之后，这题就变得非常好做了。相比上一道题，这个问题在决策分支上做了变化，一次只能拿一个硬币，但是有两个位置选择。换句话说，每一步依然会有两个分支，但是结构略有不同。

这种结构上的不同影响最大的是 dp[][]，因为我们不能再固定一端，用一个维度来表示剩下硬币构成的数组，因为从两端取硬币的时候，总共的区间数量 (start, end) 是 O(n^2)，需要用两个维度表示。

- **dp[start][end] 当前玩家从index = (start, end) 区间内的最大 value**

除此之外题目的结构和记忆化搜索并没有太大区别，依然是 MiniMax + Memoization.

ps: 假设没有面值为 0 的硬币，相比上一题跳过了 Arrays.fill -1 的初始化。

```
public class Solution {
    /**
     * @param values: an array of integers
     * @return: a boolean which equals to true if the first play
er will win
     */
    public boolean firstWillWin(int[] values) {
        // write your code here
        int n = values.length;
        if(n <= 1) return true;

        // dp[i][j] = maximum value current player can get
        // between start = i, end = j
        int[][] dp = new int[n][n];

        dp[0][0] = values[0];
```

```
        dp[n - 1][n - 1] = values[n - 1];

        int sum = 0;
        for(int num : values){
            sum += num;
        }

        return 2 * memoizedSearch(0, n - 1, values, dp) > sum;
    }

    private int memoizedSearch(int start, int end, int[] values,
int[][] dp){
        if(start > end) return 0;
        if(dp[start][end] != 0) return dp[start][end];

        int takeLeft = values[start]
                    + Math.min(memoizedSearch(start + 2, end, va
lues, dp),
                               memoizedSearch(start + 1, end - 1
, values, dp));

        int takeRight = values[end]
                    + Math.min(memoizedSearch(start + 1, end - 1
, values, dp),
                               memoizedSearch(start, end - 2, va
lues, dp));

        dp[start][end] = Math.max(takeLeft, takeRight);
        return dp[start][end];
    }
}
```

Guess Number Higher or Lower II

这题按类型分的话可以划为博弈类 DP，类似的还有 Lintcode 上的 Coins in a Line 1,2,3

我们先定义 dp[j]，代表着如果我们在区间 [i , j] 内进行查找，所需要的最少 cost 来保证找到结果。(当然，因为给定数字是 [1, n]，这里有一个 index off by one 的问题)。不难发现对于最开始的函数输入 n ，我们的最终结果就是 dp[0][n - 1] ，也即数字区间 [1 , n] 保证得到结果所需要的最小 cost.

如果以 top-down recursion 的方式分析这个问题，可以发现对于区间 [i, j] ，我们的猜测 i <= k <= j 我们可能出现以下三种结果：

```
    1. k 就是答案，此时子问题的额外 cost = 0 ，当前位置总 cost  = k + 0
;
    2. k 过大，此时我们的有效区间缩小为 [i , k - 1] 当前操作总 cost  = k
 + dp[k - 1];
    3. k 过小，此时我们的有效区间缩小为 [k + 1 , j] 当前操作总 cost  = k
 + dp[k + 1][j];
```

由于我们需要“保证得到结果”，也就是说对于指定 k 的选择，我们需要准备最坏情况 cost 是以下三种结果生成的 subproblem 中cost 最大的那个； 然而同时对于一个指定区间 [i , j] ，我们可以选择任意 i <= k <= j ，对于这个 k 的主观选择可以由我们自行决定，我们要选的是 k s.t. 其子问题的 cost + 当前操作 cost 最小的一个，至此，每次决策就构成了一次 MiniMax 的博弈。

同时因为我们有很多的 overlapping subproblems ，而且问题本身具有 optimal substructure，提高算法效率最简单直观的方式，就是用 int[][] dp 做缓存，来进行自顶向下的记忆化搜索 (top-down memoized search).

```
public class Solution {
    public int getMoneyAmount(int n) {
        // dp[i][j] min cost to guarantee to win from interval [
i , j]
        return getMinCost(0, n - 1, new int[n][n]);
    }

    private int getMinCost(int start, int end, int[][] dp){
        if(start >= end) return 0;

        if(dp[start][end] != 0) return dp[start][end];

        int minCost = Integer.MAX_VALUE;
        for(int i = start; i < end; i++){
            minCost = Math.min(minCost,  (i + 1) + Math.max(getM
inCost(start, i - 1, dp),
                                                                getM
inCost(i + 1, end, dp)));
        }
        dp[start][end] = minCost;

        return dp[start][end];
    }
}
```

Flip Game，Nim Game

Flip Game

- new String(charArray)
- String.copyValueOf(charArray)

这两种做法在 dfs + backtracking 中都可以正确就地 copy charArray 的值。

```
public List<String> generatePossibleNextMoves(String s) {
    List<String> list = new ArrayList<>();
    char[] charArr = s.toCharArray();

    for(int i = 0; i < charArr.length - 1; i++){
        if(charArr[i] == '+' && charArr[i + 1] == '+'){
            charArr[i] = '-';
            charArr[i+1] = '-';

            list.add(new String(charArr));

            charArr[i] = '+';
            charArr[i+1] = '+';
        }
    }

    return list;
}
```

Flip Game II

博弈类 DP ~ 天然的记忆化搜索，而且状态因为用 String 可以完全表示，也很好处理。

在 recursion 过程中，因为只能把 "++" 变成 "--"，我们每一步的状态空间和选择会越来越小，而且没有回头路或者环。于是每一步之后，新的状态都是一个更小的子问题。

Optimal Substructure: 如果当前状态下，我的任何 move 可以得到一个对手无法获胜的局面，则我当前局面必胜。所以计算过程就是遍历所有可能 move，top-down 记忆化搜索就好了，循环中间可以直接 early termination.

超过 95.83%，速度不错~

```
public class Solution {
    public boolean canWin(String s) {
        // Key : current state of game
        // Value : If the current player can win.
        HashMap<String, Boolean> map = new HashMap<>();

        return memoizedSearch(s, map);
    }

    private boolean memoizedSearch(String s, HashMap<String, Boo
lean> map){
        if(map.containsKey(s)) return map.get(s);

        boolean canWin = false;
        char[] charArr = s.toCharArray();

        for(int i = 0; i < s.length() - 1; i++){
            if(charArr[i] == '+' && charArr[i + 1] == '+'){
                charArr[i] = '-';
                charArr[i + 1] = '-';

                boolean rst = memoizedSearch(new String(charArr)
, map);
                if(!rst){
                    canWin = true;
                    break;
                }

                charArr[i] = '+';
                charArr[i + 1] = '+';
            }
        }

        map.put(s, canWin);

        return canWin;
    }
}
```

6/25, 动态规划，区间类DP

- 求一段区间的解 **min/max/count**
- 相比划分类 **DP** ，区间类 **DP** 为连续相连的 **subproblem** ，中间不留空，更有 **divide & conquer** 的味道。
- 转移方程通过区间更新
- 从大到小的更新

Matrix-chain multiplication (算法导论)

- 给定矩阵向量 **[A1, A2, A3 .. An]**
- 矩阵乘法有结合律，所以任意的 **parenthesization** 结果一样
- **Dimension (a x b)** 乘 **(b x c)** 得到 **(a x c)** ，总计算量为 **a x b x c**
- 可能的括号加法为 **Catalan** 数，**O(2^n)**，因而搜索不合适。

$$P(n) = \begin{cases} 1 & \text{if } n = 1 , \\ \displaystyle\sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 . \end{cases} \tag{15.6}$$

Problem 12-4 asked you to show that the solution to a similar recurrence is the sequence of ***Catalan numbers***, which grows as $\Omega(4^n/n^{3/2})$. A simpler exercise (see Exercise 15.2-3) is to show that the solution to the recurrence (15.6) is $\Omega(2^n)$. The number of solutions is thus exponential in n, and the brute-force method of exhaustive search makes for a poor strategy when determining how to optimally parenthesize a matrix chain.

- 让 **dp[i][j]** 代表**(i , j)** 区间内最优的括号顺序运算次数
  - 符合 **optimal substructure**，反证法
- **A = rows * cols**，假如从 **k** 分开左右 **i <= k < j** ，如下 **k = 5** 时：
- **[A1, A2, A3, A4, A5 || A6, A7,A8,A9]**
- 左子问题为 **A1.rows x A5.cols**
- 右子问题为 **A6.rows x A9.cols**
- 其中 **A5.cols = A6.rows**
- 其总花费为 **dp[1,5] + dp[6,9] + A1.rows * A5.cols * A9.cols**

至此，对于任意 **size (i , j)** 的向量区间，我们都可以遍历所有合理 **k** 的切分点，实现记忆化的 **divide & conquer**，当前区间的最优解一定由其最优子区间拼接而成。

```
MATRIX-CHAIN-ORDER(p)
n = p.length − 1
let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n              // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        m[i, j] = ∞
        for k = i to j − 1
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

子问题图如下。其实就是一个 **n x n** 的矩阵对角线，代表所有的子区间。

Figure 15.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

Palindrome Partitioning II

上一题的求所有区间最优解进行拼接的思路和 optimal substructure 结构和这题非常像，再贴一遍，感受一下。

不过 Matrix Chain Multiplication 要比这个复杂，时间复杂度为 O(n^3). 毕竟每个切点上会生成两个 subproblems.

```
public class Solution {
    public int minCut(String s) {
        if(s == null || s.length() <= 1) return 0;
        int len = s.length();

        boolean[][] isPalindrome = new boolean[len][len];
        int[] dp = new int[len];

        for(int i = 0; i < len; i++){
            dp[i] = i;
            for(int j = 0; j <= i; j++){
                if(s.charAt(i) == s.charAt(j) && (i - j < 2 || i
sPalindrome[j + 1][i - 1])){
                    isPalindrome[i][j] = isPalindrome[j][i] = tr
ue;
                    if(j == 0){
                        dp[i] = 0;
                    } else {
                        dp[i] = Math.min(dp[i], dp[j - 1] + 1);
                    }
                }
            }
        }

        return dp[len - 1];
    }
}
```

Stone Game

著名的区间类 DP 入门题 -- 石子归并

以数组【3,4,5,6】为例，进行归并的 subproblem graph 如下，path 上的数字代表每一步的 cost.

- 我们一定可以得到一个 **height balanced complete tree**，因为每步都只归并两堆石子，只是每步的 **branching factor** 不同；
- 所有 **subproblem** 的叶节点 **cost** 一致，为所有石子的总和。

这种画法的结构是对的，但是并不合理，因为没有体现出**“overlap subproblems”**，每个子问题看起来都像独立问题一样。

这样就明显多了，而且和前面的 **“区间划分DP”** 联系紧密。

自己用记忆化搜索写的第一版，比较粗糙~

- 每次归并的 **cost =** 归并两个区间的最优 **cost +** 两个区间的区间和
- 因此区间最优用 **dp[][]** 记忆化搜索，区间和 **sum[][]** 可以 **O(n ^ 2)** 时间预处理。
- **O(n^2) preprocess + O(n^2) number of intervals * O(n) number of candidate cuts = O(n^2) + O(n^3)**
- 可以看到，记忆化搜索中，**dp[][]** 每一个位置只会被遍历一次而且不会再生成新的 **subproblems**，其时间复杂度和 **bottom-up** 的迭代循环是一样的。

```
public class Solution {
    /**
     * @param A an integer array
     * @return an integer
     */
    public int stoneGame(int[] A) {
        // Write your code here
        if(A == null || A.length == 0) return 0;
        int n = A.length;

        // Minimum cost to merge interval dp[i][j]
        int[][] dp = new int[n][n];
        int[][] sum = new int[n][n];

        // Pre-process interval sum
        for(int i = 0; i < n; i++){
            for(int j = i; j >= 0; j--){
                if(j == i) sum[i][j] = A[i];
                else sum[i][j] = sum[j][i] = A[j] + sum[j + 1][i
];
            }
        }
```

```
        return memoizedSearch(0, n - 1, A, dp, sum);
    }

    private int memoizedSearch(int start, int end, int[] A, int[
][] dp, int[][] sum){
        if(start > end) return 0;
        if(start == end) return 0;
        if(start + 1 == end) return A[start] + A[end];

        if(dp[start][end] != 0) return dp[start][end];

        int min = Integer.MAX_VALUE;
        for(int i = start; i < end; i++){
            int cost = memoizedSearch(start, i, A, dp, sum) + me
moizedSearch(i + 1, end, A, dp, sum) + sum[start][i] + sum[i + 1
][end];
            min = Math.min(min, cost);
        }

        dp[start][end] = min;

        return min;
    }
}
```

对于 **interval sum** ，根据搜索结构可以做一个显而易见的优化，因为每次 **split** 的 **start, pivot, end** 我们都知道，而且合并（**start, end**） 区间的两堆石子，最终的区间和一定为 **(start, end)** 的区间和，用一维的 **prefix sum** 数组就可以了。

用 **prefix sum** 数组要记得初始化时候的 **int[n + 1] zero padding**，还有取值时候对应的 **sum[end + 1] - sum[start + 1 - 1] offset.**

```
public class Solution {
    /**
     * @param A an integer array
     * @return an integer
```

```
     */
    public int stoneGame(int[] A) {
        // Write your code here
        if(A == null || A.length == 0) return 0;
        int n = A.length;

        // Minimum cost to merge interval dp[i][j]
        int[][] dp = new int[n][n];
        int[] sum = new int[n + 1];

        // Pre-process interval sum
        for(int i = 0; i < n; i++){
            sum[i + 1] = sum[i] + A[i];
        }

        return memoizedSearch(0, n - 1, A, dp, sum);
    }

    private int memoizedSearch(int start, int end, int[] A, int[
][] dp, int[] sum){
        if(start > end) return 0;
        if(start == end) return 0;
        if(start + 1 == end) return A[start] + A[end];

        if(dp[start][end] != 0) return dp[start][end];

        int min = Integer.MAX_VALUE;
        for(int i = start; i < end; i++){
            int cost = memoizedSearch(start, i, A, dp, sum) + me
moizedSearch(i + 1, end, A, dp, sum) + sum[end + 1] - sum[start]
;
            min = Math.min(min, cost);
        }

        dp[start][end] = min;

        return min;
    }
}
```

Burst Balloons

这题和石子归并很像，更像 Matrix Chain Multiplication. 都是区间类 DP，而且原数组会随着操作逐渐减小，动态变化。

然而就算是动态变化的数组，变化的也并不是状态，而只是子状态的范围，记忆化搜索中的 (start, end).

所以这题的难点在于，如何在动态变化的数组中，依然正确定义并计算 **subproblem.**

问题一：边界气球

- 考虑到计算方式为相邻气球乘积，可以两边放上 1 来做 padding，不会影响最后结果的正确性。

问题二：子问题返回后，如何处理相邻气球？

- 在stone game中，最后融合两个区间要靠区间和；
- 在busrt balloon中，两个区间返回时已经都被爆掉了，融合区间靠的是两个区间最外面相邻的气球。（因此 padding 才很重要）
- 正如 Matrix Chain Multiplication 中，左右区间相乘结束返回时，最后融合那步的 cost = A(start).rows * A(k).cols * A(end).cols

```
public class Solution {
    public int maxCoins(int[] nums) {
        if(nums == null || nums.length == 0) return 0;
        int n = nums.length;
        int[] arr = new int[n + 2];
        arr[0] = 1;
        arr[n + 1] = 1;
        for(int i = 0; i < n; i++){
            arr[i + 1] = nums[i];
        }

        int[][] dp = new int[n + 2][n + 2];


        return memoizedSearch(1, n, arr, dp);
    }

    private int memoizedSearch(int start, int end, int[] arr, int
[][] dp){
        if(dp[start][end] != 0) return dp[start][end];

        int max = 0;
        for(int i = start; i <= end; i++){
            int cur = arr[start - 1] * arr[i] * arr[end + 1];
            int left = memoizedSearch(start, i - 1, arr, dp);
            int right = memoizedSearch(i + 1, end, arr, dp);

            max = Math.max(max, cur + left + right);
        }

        dp[start][end] = max;

        return max;
    }
}
```

Scramble String

弄了半天写了个错误的版本，只考虑了 cut 位置对齐的情况，可以过 157 / 281 个 test cases, 然而像 "abc" 和 "bca" 这种起始位置就不对齐的就会出错。

a | bc

bc | a

所以很显然的，O(n^3) 泡汤了~

http://www.blogjava.net/sandy/archive/2013/05/22/399605.html

下面的是基于九章答案的记忆化搜素解法，改了我好久。。。

改写过程中一直在犯的错误是，在 subcall 中 s1,s2 已经是 substring 的情况下，依然用上一层传过来的参数作为参考去切分新的 substring. 这是错误的，只需要在参数中得到的 s1, s2 上切割就好了，因为传进来的并不是最原始的 string.

每一层 search 中，参数里面的 start / end / n 代表着相对于原始 string 的位置，用于查询和记录 DP; 而这一层的 s1, s2 又是新的子问题，除了涉及传参和DP之外的地方，都以 s1, s2 为准。

s.substring(i,j) 中，最后截取的 substring 长度就是 j - i.

```
public class Solution {
    public boolean isScramble(String s1, String s2) {
        if(!isAnagram(s1, s2)) return false;

        int n = s1.length();

        // dp[i][j][k] : s1 starting from index i, s2 string fro
m index j
        //                pick k chars, are we getting scrambled
strings ?

        // 0 : not searched, 1 : true, -1 : false;
        int[][][] dp = new int[n][n][n + 1];
```

```
        return isScrambleMemo(s1, s2, 0, 0, n, dp);
    }

    private boolean isScrambleMemo(String s1, String s2, int one
Start, int twoStart, int n, int[][][] dp){
        if(dp[oneStart][twoStart][n] != 0) return (dp[oneStart][
twoStart][n] == 1) ? true : false;

        if(s1.equals(s2)){
            dp[oneStart][twoStart][n] = 1;
            return true;
        }
        if(!isAnagram(s1, s2)){
            dp[oneStart][twoStart][n] = -1;
            return false;
        }


        // i = number of characters we take
        for(int i = 1; i < s1.length() ; i++){
            String s1Left = s1.substring(0, i);
            String s1Right = s1.substring(i, s1.length());

            String leftSideS2Left = s2.substring(0, i);
            String leftSideS2Right = s2.substring(i, s2.length()
);

            String rightSideS2Left = s2.substring(0, s2.length()
 - i);
            String rightSideS2Right = s2.substring(s2.length() -
 i, s2.length());

            if(isScrambleMemo(s1Left, leftSideS2Left, oneStart,
twoStart, i, dp) &&
                isScrambleMemo(s1Right, leftSideS2Right, oneStar
t + i, twoStart + i, n - i, dp)) {

                dp[oneStart][twoStart][n] = 1;
                return true;
            }
```

```
            if(isScrambleMemo(s1Left, rightSideS2Right, oneStart
, twoStart + n - i, i, dp) &&
               isScrambleMemo(s1Right, rightSideS2Left, oneStart
 + i, twoStart, n - i, dp)) {

                dp[oneStart][twoStart][n] = 1;
                return true;
            }
        }

        dp[oneStart][twoStart][n] = -1;
        return false;
    }

    // Assuming only lower case letters
    private boolean isAnagram(String s1, String s2){
        if(s1.length() != s2.length()) return false;
        int[] hash = new int[26];
        for(int i = 0; i < s1.length(); i++){
            int index = s1.charAt(i) - 'a';
            hash[index] ++;
        }
        for(int i = 0; i < s2.length(); i++){
            int index = s2.charAt(i) - 'a';
            hash[index] --;
            if(hash[index] < 0) return false;
        }
        return true;
    }
}
```

6/27, 动态规划，subarray 划分类

所谓 “划分类” **DP**，是指给定原数组之后，将其划分为 **k** 个子数组，使其 **sum / product** 最大 **/** 最小的 **DP** 类问题。

而这类问题的 **optimal substructure** 是，对于给定区间的 **globalMax**，其最优解一定由所属区间内的 **localMax** 区间组成，可能不连续。 以“必须以当前结尾” 来保证 **local** 子问题之间的连续性；用 **global** 来记录最优解之间可能的不连续性。

划分类**DP** 与 区间类**DP** 主要区别在于，我们只取其中 **k** 段，而中间部分可以留空；而区间类 **DP** 子问题之间是连贯而不留空的。区间类 **DP** 是记忆化的 **divide & conquer,** 划分类 **DP** 则是不连续 **subarray** 的组合，不同于区间类 **DP** 每一个区间都要考虑与计算，划分类**DP** 很多元素和子区间是可以忽略的，有贪心法的思想在里面，因此也依赖于正确的 **local / global** 结构来保证算法的正确性。

Kadane's Algorithm 相比 **prefix sum** 的特点是，必须要以 **nums[0]** 做初始化，从 **index = 1** 开始搜，优点是在处理 **prefix product** 的时候更容易处理好“**-5**” 和 “**0**” 的情况。

这类靠 **prefix sum/product** 的 **subarray** 问题，要注意好对于每一个子问题的 **local / global** 最优解结构，其中 **local** 解都是“以当前结尾 **&&** 连续”，而 **global** 未必。

Prefix sum 数组的 **local / global** 通用模板，求 **min / max** 皆可，使用时需要注意初始条件以及顺序；

```
        int[] leftMax = new int[n];
        int prefixSum = 0;
        // 代表从起始位置开始，值最小的连续和数组
        int localMin = 0;
        int globalMax = Integer.MIN_VALUE;
        for(int i = 0; i < n; i++){
            prefixSum += nums[i];
            globalMax = Math.max(globalMax, prefixSum - localMin
);
            localMin = Math.min(localMin, prefixSum);

            leftMax[i] = globalMax;
        }
```

Minimum Subarray

下面这两道 Max / Min Subarray 完全是一个套路： Kadane's Algorithm.

因为代码和思路看起来过于简单，比较容易忽略掉这题思路，还有正确性的分析。

比如 CodeForce 上这个帖子的另一种写法：

- 对于每一个位置 **i**，我们维护
  - 当前 **prefix sum**；
  - **subarray** 最小 **prefix sum**
  - **subarray** 最大 **prefix sum**
- 其中每一个 subarray，都可以用 prefix sum 之间做减法获得。
- 即，这种迭代算法完整考虑了整个 array 中所有的 candidate subarray，因此保证了算法正确性。

```
public class Solution {
    /**
     * @param nums: a list of integers
     * @return: A integer indicate the sum of minimum subarray
     */
    public int minSubArray(ArrayList<Integer> nums) {
        // write your code
        int max = 0, min = Integer.MAX_VALUE;
        int prefixSum = 0;
        for(int i = 0; i < nums.size(); i++){
            prefixSum += nums.get(i);
            min = Math.min(min, prefixSum - max);
            max = Math.max(max, prefixSum);
        }

        return min;
    }
}
```

Kadane's Algorithm 其实是在维护一个 **sliding window**，不过每个迭代的时候，在考虑以当前元素为新起点之余，又砍去了之前的部分。

input : [-1,5,6,-1,-2,4]

- iter0: max = -1, prev = sum[-1];
- iter1: max = 5, prev = sum[5];
- iter2: max = 11, prev = sum[5,6];
- iter3: max = 11, prev = sum[5,6,-1];
- iter4: max = 11, prev = sum[5,6,-1,-2];
- iter5: max = 12, prev = sum[5,6,-1,-2,4]

可以看到，在 **kadane's algorithm** 中，我们保存的这个 **prevMin** 始终是连续的，起点不一定是哪，但是终点一定是 **current**，是一个 **local** 最优解。我们每次迭代，用 **local** 最优解去更新 **global** 最优解。

```
public class Solution {
    /**
     * @param nums: a list of integers
     * @return: A integer indicate the sum of minimum subarray
     */
    public int minSubArray(ArrayList<Integer> nums) {
        // write your code
        int prevMin = 0;
        int cur = nums.get(0);
        int minRst = nums.get(0);
        for(int i = 1; i < nums.size(); i++){
            cur = Math.min(nums.get(i), prevMin + nums.get(i));
            minRst = Math.min(minRst, cur);
            prevMin = cur;
        }

        return minRst;
    }
}
```

Maximum Subarray

用区间和的思路：

```
public class Solution {
    public int maxSubArray(int[] nums) {
        if(nums == null || nums.length == 0) return 0;

        int max = Integer.MIN_VALUE, min = 0;
        int prefixSum = 0;
        for(int i = 0; i < nums.length; i++){
            prefixSum += nums[i];
            max = Math.max(max, prefixSum - min);
            min = Math.min(min, prefixSum);
        }

        return max;
    }
}
```

用 Kadane's Algorithm:

```
public class Solution {
    public int maxSubArray(int[] nums) {
        if(nums == null || nums.length == 0) return 0;

        int cur = nums[0];
        int max = nums[0];
        for(int i = 1; i < nums.length; i++){
            cur = Math.max(nums[i], cur + nums[i]);
            max = Math.max(max, cur);
        }

        return max;
    }
}
```

Maximum Product Subarray

L 家面经题。

相比较 prefix sum 的问题，操作改成乘法之后有几个不同的地方需要处理：

- 负号。遇到负号时，原本的 **min / max** 会直接反转。最优解可能是有一系列正数相乘而来，但也可能是一系列正数中带着偶数个数的负数。
    - 保存以当前元素截止的 **min / max subarray** （保证连续性），遇到负数时调换相乘。
- 加法操作遇到 **0** 是无所谓的，乘法操作却会导致直接切断 **prefix product.**
    - 每个位置的 **max / min subarray** 同时考虑当前元素，正确处理 **0** 之余保证连续性。

其实这个做法与其说像 **prefix product** ，不如说更像 **Kadane's Algorithm.** 可以看到这个算法可以有效免疫各种切断的情况，只维护到当前位置结尾的 **max / min subarray** 结果就可以了。

在这里面，**min/max** 是连续的，**local** 以 **i** 结尾的； **rst** 是 **global** 非连续的。

> - 前缀积在这题上没用，所有的都是 **localMin / localMax**，迭代更新。
> - 不能用 **val = 1** 来做初始化，要用 **nums[0]**；
> - **nums[i]** 为负时，要多开个临时变量，不然会覆盖掉下一行需要的值

```
public class Solution {
    public int maxProduct(int[] nums) {
        int max = nums[0], min = nums[0];
        int rst = nums[0];

        for(int i = 1; i < nums.length; i++){
            if(nums[i] > 0){
                max = Math.max(nums[i], max * nums[i]);
                min = Math.min(nums[i], min * nums[i]);
            } else {
                int oldMax = max;
                max = Math.max(nums[i], min * nums[i]);
                min = Math.min(nums[i], oldMax * nums[i]);
            }
            rst = Math.max(rst, max);
        }

        return rst;
    }
}
```

Maximum Subarray II

Subarray I 只是开胃菜的话，这道题就开始比较认真考察如何利用 prefix sum 做 DP 解决 subarray sum 类问题了。

这次光用 **local** 变量更新然后返回 **global** 最优还不够，我们需要保存对于每一个子问题的 **global** 最优（可能并不以当前位置结尾），用于后面的左右两边全局查询。

我们需要利用 prefix sum 数组，定义两个 dp[]

- **left[i]** 代表从最左边到 **i** 位置所能取得的最大 **subarray sum;**
- **right[i]** 代表从最右边到 **i** 位置所能取得的最大 **subarray sum;**

- 两个数组都是 **global** 解。
- 每次迭代的变量 **minSum** 和 **prefixSum** 是相对于每个位置的 **local** 解。

这样对于每一个位置 i ，我们都可以用 left[] 和 right[] 的子数组把最优解拼接出来。

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: An integer denotes the sum of max two non-overla
pping subarrays
     */
    public int maxTwoSubArrays(ArrayList<Integer> nums) {
        // write your code
        if(nums == null || nums.size() == 0) return 0;

        int n = nums.size();

        // Maximum subarray value from left/right with n elements

        int[] left = new int[n];
        int[] right = new int[n];

        int prefixSum = 0;
        int minSum = 0;
        int max = Integer.MIN_VALUE;
        for(int i = 0; i < n; i++){
            prefixSum += nums.get(i);
            max = Math.max(max, prefixSum - minSum);
            minSum = Math.min(minSum, prefixSum);
            left[i] = max;
        }

        prefixSum = 0;
        minSum = 0;
        max = Integer.MIN_VALUE;
        for(int i = n - 1; i >= 0; i--){
            prefixSum += nums.get(i);
            max = Math.max(max, prefixSum - minSum);
```

```
            minSum = Math.min(minSum, prefixSum);
            right[i] = max;
        }

        int rst = Integer.MIN_VALUE;

        for(int i = 0; i < n - 1; i++){
            rst = Math.max(rst, left[i] + right[i + 1]);
        }

        return rst;
    }
}
```

同样的思路，用 Kadane's Algorithm 就要注意初始化问题，不然无法正确处理 array 两端的负数。

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: An integer denotes the sum of max two non-overla
pping subarrays
     */
    public int maxTwoSubArrays(ArrayList<Integer> nums) {
        // write your code
        if(nums == null || nums.size() == 0) return 0;

        int n = nums.size();

        // Maximum subarray value from left/right with n elements

        int[] left = new int[n];
        int[] right = new int[n];

        int prevMax = nums.get(0);
        int max = nums.get(0);
        left[0] = nums.get(0);
        for(int i = 1; i < n; i++){
            prevMax = Math.max(nums.get(i), nums.get(i) + prevMa
```

```
x);
            max = Math.max(max, prevMax);
            left[i] = max;
        }

        prevMax = nums.get(n - 1);
        max = nums.get(n - 1);
        right[n - 1] = nums.get(n - 1);

        for(int i = n - 2; i >= 0; i--){
            prevMax = Math.max(nums.get(i), nums.get(i) + prevMa
x);
            max = Math.max(max, prevMax);
            right[i] = max;
        }

        int rst = Integer.MIN_VALUE;

        for(int i = 0; i < n - 1; i++){
            rst = Math.max(rst, left[i] + right[i + 1]);
        }

        return rst;
    }
}
```

Maximum Subarray III

Maximum SubArray III

- 状态 State
  - localMax[i][j] 表示前i个元素，取j个子数组，第i个元素必须取的最大值
  - globalMax[i][j]表示前i个元素，取j个子数组，第i个元素随便取不取的最大值
- 方程 Function
  - localMax[i][j] = Max(localMax[i-1][j], globalMax[i-1][j-1]) + nums[i-1];
  - globalMax[i][j] = Max(globalMax[i-1][j], localMax[i][j]);
- 初始化 Intialization
  - localMax[k][k-1] = Integer.MIN_VALUE;
- 答案 Answer
  - globalbest[n][k]

optimal substructure 确定之后，最重要的是要思考循环的实现顺序：

是从 切分数 **k** 开始循环，还是从数组位置 **i** 开始循环？

首先思考这个性质：对于切割数为 **j** 的数组，如果其 **size = i < j**，是无法得出正确结果的。换句话说，每次循环真正的正确起点，一定得至少是从 **i = j** 开始， **i** 受限于 **j.**

那么顺着这个思路，如果以 **i** 作为外循环，内循环 **j** 一定是在 **[0, i]** 区间内，并且 **j <= k.** 因此内循环需要两个条件：

- **j <= k;**
- **j <= i;**

对于任意指定 **j** ，需要做一个类似于我们之前计算 **subarray I & II** 时所需要的初始化，确保即使当前位置为负，作为以当前位置结尾的 **localMax** 依然选中此元素。因此先做一个 **localMax[j - 1][j] = Integer.MIN_VALUE;**

另一个需要着重注意的是，当 **i = j** ，即元素数 **=** 切分数时，即使当前元素为负，我们的 **globalMax** 也别无选择，必须以 **localMax[i][j]** 为准，采用当前元素。

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @param k: An integer denote to find k non-overlapping sub
arrays
     * @return: An integer denote the sum of max k non-overlappi
ng subarrays
     */
    public int maxSubArray(int[] nums, int k) {
        // write your code here
        if(nums == null || nums.length == 0) return 0;

        int n = nums.length;


        int[][] localMax = new int[n + 1][k + 1];
        int[][] globalMax = new int[n + 1][k + 1];

        for(int i = 1; i <= n; i++){
            for(int j = 1; j <= k && j <= i; j++){
                localMax[j - 1][j] = Integer.MIN_VALUE;

                localMax[i][j] = Math.max(localMax[i - 1][j], gl
obalMax[i - 1][j - 1])
                                + nums[i - 1];
                if(i == j) globalMax[i][j] = localMax[i][j];
                else globalMax[i][j] = Math.max(localMax[i][j],
globalMax[i - 1][j]);
            }
        }

        return globalMax[n][k];
    }
}
```

这题的另一种循环写法，参照的九章答案。可以看到当循环内计算内容一致时，外循环的顺序是完全可以依据条件调换的，就像 **Sparse Matrix Multiplication** 一样。

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @param k: An integer denote to find k non-overlapping sub
arrays
     * @return: An integer denote the sum of max k non-overlappi
ng subarrays
     */
    public int maxSubArray(int[] nums, int k) {
        // write your code here
        if(nums == null || nums.length == 0) return 0;

        int n = nums.length;


        int[][] localMax = new int[n + 1][k + 1];
        int[][] globalMax = new int[n + 1][k + 1];

        for(int j = 1; j <= k; j++){
            localMax[j - 1][j] = Integer.MIN_VALUE;
            for(int i = j; i <= n; i++){
                localMax[i][j] = Math.max(localMax[i - 1][j], gl
obalMax[i - 1][j - 1])
                                + nums[i - 1];
                if(i == j) globalMax[i][j] = localMax[i][j];
                else globalMax[i][j] = Math.max(localMax[i][j],
globalMax[i - 1][j]);
            }
        }

        return globalMax[n][k];
    }
}
```

Maximum Subarray Difference

这题和之前求 **maximum non-overlap subarrays** 的思路基本一致，只不过要求的数组变成了四个：**leftMax, leftMin, rightMax** 和 **rightMin.**

于是这题变成了一个非常适合考察计算各种 **local / global prefix sum** 数组模板熟练度的问题。

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: An integer indicate the value of maximum differe
nce between two
     *          Subarrays
     */
    public int maxDiffSubArrays(int[] nums) {
        // write your code here
                // write your code
        int n = nums.length;

        int[] leftMax = new int[n];
        int[] leftMin = new int[n];
        int[] rightMax = new int[n];
        int[] rightMin = new int[n];

        int prefixSum = 0;
        int localMin = 0;
        int globalMax = Integer.MIN_VALUE;
        for(int i = 0; i < n; i++){
            prefixSum += nums[i];
            globalMax = Math.max(globalMax, prefixSum - localMin
);
            localMin = Math.min(localMin, prefixSum);

            leftMax[i] = globalMax;
        }

        prefixSum = 0;
        int localMax = 0;
        int globalMin = Integer.MAX_VALUE;
```

```
        for(int i = 0; i < n; i++){
            prefixSum += nums[i];
            globalMin = Math.min(globalMin, prefixSum - localMax
);
            localMax = Math.max(localMax, prefixSum);
            leftMin[i] = globalMin;
        }

        prefixSum = 0;
        localMin = 0;
        globalMax = Integer.MIN_VALUE;
        for(int i = n - 1; i >= 0; i--){
            prefixSum += nums[i];
            globalMax = Math.max(globalMax, prefixSum - localMin
);
            localMin = Math.min(localMin, prefixSum);
            rightMax[i] = globalMax;
        }

        prefixSum = 0;
        localMax = 0;
        globalMin = Integer.MAX_VALUE;
        for(int i = n - 1; i >= 0; i--){
            prefixSum += nums[i];
            globalMin = Math.min(globalMin, prefixSum - localMax
);
            localMax = Math.max(localMax, prefixSum);
            rightMin[i] = globalMin;
        }

        int rst = Integer.MIN_VALUE;

        for(int i = 0; i < n - 1; i++){
            int ans = Math.max(Math.abs(leftMax[i] - rightMin[i
+ 1]),
                               Math.abs(leftMin[i] - rightMax[i
 + 1]));
            rst = Math.max(rst, ans);
        }
```

```
        return rst;
    }
}
```

Best Time to Buy and Sell Stock

很简单的问题，就当练手了。

```
public class Solution {
    public int maxProfit(int[] prices) {
        if(prices == null || prices.length == 0) return 0;
        int min = prices[0];
        int maxProfit = 0;
        for(int i = 1; i < prices.length; i++){
            maxProfit = Math.max(maxProfit, prices[i] - min);
            min = Math.min(min, prices[i]);
        }
        return maxProfit;
    }
}
```

Best Time to Buy and Sell Stock II

这个也基本是送分题，因为可以参与多次交易(unlimited)，只需要把每次的increase 加在一起就可以了。

```
public class Solution {
    public int maxProfit(int[] prices) {
        if(prices == null || prices.length == 0) return 0;
        int profit = 0;
        int min = prices[0];
        for(int i = 1; i < prices.length; i++){
            if(prices[i] > min){
                profit += prices[i] - min;
            }
            min = prices[i];
        }

        return profit;
    }
}
```

Best Time to Buy and Sell Stock III

在透彻理解了 Maximum Subarray II 之后，这道题完全就是个套了马甲的简化版。

k 次交易 = k 个 non-overlapping subarray

以这个角度去想，无非就是从两个方向扫描，利用 localMin / localMax 与当前元素的差值，去构造从左边/右边扫的 dp 数组。

- **left[i] :** 从最左面到 **i** 所能获得的最大利益（单次交易）
- **right[i] :** 从 **i** 到最右面所能获得的最大利益（单次交易）

然后拼起来就好了。要注意的细节：

- 每个位置并不是非交易不可，需要用 **0** 来代表不做交易的情况；
- 不是非要做 “两次交易” 不可，因此 **left[end]** 作为一次交易的代表，也要考虑在内。

```
public class Solution {
    public int maxProfit(int[] prices) {
        if(prices == null || prices.length <= 1) return 0;
        int n = prices.length;

        int[] left = new int[n];
        int[] right = new int[n];

        int localMin = prices[0];
        int globalMax = Integer.MIN_VALUE;
        for(int i = 1; i < n; i++){
            globalMax = Math.max(globalMax, Math.max(0, prices[i
] - localMin));
            localMin = Math.min(localMin, prices[i]);
            left[i] = globalMax;
        }

        int localMax = prices[n - 1];
        globalMax = Integer.MIN_VALUE;
        for(int i = n - 1; i >= 0; i--){
            globalMax = Math.max(globalMax, Math.max(0, localMax
 - prices[i]));
            localMax = Math.max(localMax, prices[i]);
            right[i] = globalMax;
        }

        int rst = 0;

        for(int i = 0; i < n - 1; i++){
            rst = Math.max(rst, left[i] + right[i + 1]);
        }
        // might be completely on left side;
        rst = Math.max(rst, left[n - 1]);

        return rst;
    }
}
```

Best Time to Buy and Sell Stock IV

延续了 subarray III 的优良传统，用 localMax 指代“必须在当前位置结束”来保证 local 子问题之间状态的连续性，用 globalMax 来记录状态之间可能的不连续性。

股票和 **subarray** 问题之间稍微有不同： **subarray** 里面初始化为 **0** ，有时候元素为 **-5** 但是不得取，因此有些位置需要 **Integer.MIN_VALUE** 的初始化； 然而股票我可以选择不卖，初始的 **0** 就正好。

同样道理，股票中也不需要处理 **subarray** 问题中 **i = j** 时候必须强行拿当前元素，觉得亏我大不了不卖嘛，交易少于 **k** 也没事。

记得内循环条件是 **j * 2 <= i + 1**，而不是 **j <= i / 2**，否则报错，因为 **index off-by-one** 处理错了。

```
public class Solution {
    public int maxProfit(int k, int[] prices) {
        if(prices == null || prices.length == 0) return 0;

        int n = prices.length;

        if(k >= n / 2){
            int profit = 0;
            for(int i = 1; i < n; i++){
                int diff = prices[i] - prices[i - 1];
                if(diff > 0) profit += diff;
            }
            return profit;
        }

        int[][] localMax = new int[n][k + 1];
        int[][] globalMax = new int[n][k + 1];

        for(int i = 1; i < n; i++){
            int diff = prices[i] - prices[i - 1];
            for(int j = 1; j <= k && j * 2 <= i + 1; j++){
                localMax[i][j] = Math.max(localMax[i - 1][j], gl
obalMax[i - 1][j - 1]) + diff;
                globalMax[i][j] = Math.max(localMax[i][j], globa
lMax[i - 1][j]);
            }
        }


        return globalMax[n - 1][k];
    }
}
```

Best Time to Buy and Sell Stock with cooldown

这个写法继承了原来股票问题的思路和 local 结构： 必须在第 i 天卖。 现在多了一种新操作"You do nothing"，就需要定义一个新数组。

考虑到股票 diff 的可拼接性质，sell[i] 可以直接由 sell[i - 1] + diff 构造而成；同时也可以由 do_nothing[i - 2] + diff 构造成，代表着上一次 sell 操作发生在 i - 3 事，i - 2 do_nothing, i - 1 买入， i 卖出的情形。

由这题我们可以发现，实际需要的 **dp[]** 数量是取决于操作数量的，要以“用当前操作结尾”来定义 **localMax.** 只不过在之前的问题中，因为买入卖出可以拼接，我们略去了 **buy** 的数组。

这题扩展开来的话，也可以扩展到 **cooldown k** 天，所依赖的状态，初始化，循环的起始位置会有变化而已。

```
public class Solution {
    public int maxProfit(int[] prices) {
        if(prices == null || prices.length <= 1) return 0;

        int n = prices.length;

        // max money in hand after each action
        int[] sell = new int[n];
        int[] donothing = new int[n];

        sell[1] = prices[1] - prices[0];

        for(int i = 2; i < n; i++){
            donothing[i] = Math.max(donothing[i - 1], sell[i - 1
]);
            sell[i] = prices[i] - prices[i - 1] + Math.max(sell[
i - 1], donothing[i - 2]);
        }

        return Math.max(sell[n - 1], donothing[n - 1]);
    }
}
```

居然有人用状态机去写这类题， 好巧妙啊。

https://leetcode.com/discuss/72030/share-my-dp-solution-by-state-machine-thinking

在整个交易过程中，我们只可能处于如上图所示的三种状态中。

这个解法的要点是，每个 **state** 的 **in-degree** 代表能到达当前 **state** 的所有可能操作。

- **s0** 可能由上一次的 **s0**，或上一次的 **s2** 卖掉而来；
- **s1** 可能由上一次的 **s1**，或者上一次的 **s0** 买入而来；
- **s2** 只可能由 **s1** 的卖出而来。

```java
public class Solution {
    public int maxProfit(int[] prices) {
        int n = prices.length;
        if (n <= 1) return 0;
        int[] s0 = new int[n];
        int[] s1 = new int[n];
        int[] s2 = new int[n];

        s0[0] = 0;
        s1[0] = -prices[0];
        s2[0] = Integer.MIN_VALUE;

        for (int i = 1; i < n; i++) {
            s0[i] = Math.max(s0[i - 1], s2[i - 1]);
            s1[i] = Math.max(s1[i - 1], s0[i - 1] - prices[i]);
            s2[i] = s1[i - 1] + prices[i];
        }

        return Math.max(s0[n - 1], s2[n - 1]);
    }
}
```

7/2, 动态规划，字符串类

字符串 **DP** 类的很多典型问题，用这一张图都能说明白：

两个 String 放一起，从某个位置开始看看是否 match; 要是不 match 了，就得错开一位看；要是 match 了，就都往后挪一位看。在 Edit Distance 里面，稍微特殊一点，因为我们还有一个 replace 操作，可以直接修正当前 mismatch，让两个字符串都挪动一格位置。

于是对于每一个 String，就有了一个对于其位置敏感的维度 dp[i]，代表从最左面开始的 i 个字符构成的字符串，也是子问题的结构定义。因为我们有两个 String 并且需要枚举位置之间可能的各种交错穿插，我们就需要两个维度 dp[i][j]，代表第一个字符串中的 i 个字符和第二个字符串中的 j 个字符匹配情况。

Longest Common Subsequence (CLRS)

研究一个新问题时，最好还是从算法导论开始。

给定长度为 m 的字符串 {1,2,3...m} ，其 subsequences 的数量总数为 2^m，即对每个字符选择 “取 / 不取”。

Theorem 15.1 (Optimal substructure of an LCS)
Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

于是有了下面的可视化版本~

这个版本的做法非常适合存储和输出 **optimal path**，也可以应用到 **Longest Increasing Subsequence** 中，用于 **follow-up** 情况下输出 **sequence.**

比如输入数组为 **X = [x1, x2 .. xn]**，其排序后的数组为 **X' = [x'1, x'2 .. x'n]**

- **LIS** 即 **X** 与 **X'** 的 **LCS**，判断条件完全一样，元素相等向右下走。如果每一步上都存行进方向，那么最后从右下角往左上出发，每一步指向左上角的箭头都是 **LIS** 的元素。

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

滚动数组优化版的代码.

最大值肯定在 **dp[][]** 的最右下角。

```
public class Solution {
    /**
     * @param A, B: Two strings.
     * @return: The length of longest common subsequence of A an
d B.
     */
    public int longestCommonSubsequence(String A, String B) {
        // write your code here
        if(A == null || B == null) return 0;
        int n = A.length();
        int m = B.length();
        if(n == 0 || m == 0) return 0;

        int[][] dp = new int[2][m + 1];

        for(int i = 1; i <= n; i++){
            for(int j = 1; j <= m; j++){
                if(A.charAt(i - 1) == B.charAt(j - 1)){
                    dp[i % 2][j] = dp[(i - 1) % 2][j - 1] + 1;
                } else {
                    dp[i % 2][j] = Math.max(dp[(i - 1) % 2][j],
dp[i % 2][j - 1]);
                }

            }
        }

        return dp[n % 2][m];
    }
}
```

One Edit Distance

一开始未经思考尝试以 LCS 长度判断，但是这个思路是错的，因为 S 和 T 的 LCS 长度可以满足条件，但是 **mismatch** 的字符却不一定是在同一个位置。况且，LCS 的时间复杂度是 O(mn), 这题应该很明显有 O(n) 的解法。

正解借鉴了论坛的代码，其实非常简洁。

核心思想是，既然有且只有 **1** 个位置 **mismatch**，我们可以直接在找到 **mismatch** 位置上判断：

- 让 **n, m** 为两个 **String** 的长度
- 如果 **m == n** ，**mismatch** 之后的子字符串一定完全相等；
- 否则长的那段在 **i** 之后的 **substring** 一定与短的以 **i** 开始的 **substring** 全等；
- 如果在循环末尾还没有发现 **mismatch**，两个字符串末尾必然会有 **1** 的长度差，否则 **S == T.**

在字符串 **DP** 中，原始字符串上的 **substring**，等价于原始区间内的 **subproblem.**

```
public class Solution {
    public boolean isOneEditDistance(String s, String t) {
        int n = s.length();
        int m = t.length();

        if(Math.abs(n - m) > 1) return false;

        for(int i = 0; i < Math.min(m,n); i++){
            if(s.charAt(i) != t.charAt(i)){
                if(n == m) return s.substring(i + 1).equals(t.su
bstring(i + 1));
                if(n > m) return s.substring(i + 1).equals(t.sub
string(i));
                if(n < m) return s.substring(i).equals(t.substri
ng(i + 1));
            }
        }

        return Math.abs(n - m) == 1;
    }
}
```

Edit Distance

结合上一道题的错误来看，可以很容易看到只依靠 LCS 长度解决这道题的乏力；因为 LCS 长度和 edit distance 对字符串结构的利用是不一样的，mismatch 的字符可以出现错位，而 edit distance 不支持一次操作进行修正。算法导论上也写的非常清楚，鉴定两条 DNA 序列的相似度，substring 是一种思路（KMP），LCS是一种思路，而 edit distance 是另一种思路。

关于这个问题最好的 slides, by Stanford

然而相同的是，这道题思考并寻找 **optimal substructure** 的思路是非常接近的，都是直接从最终答案 **String Z** 的结构 **top-down** 往回看。因为对于每个 **string**，我们对其任意操作都会构造出来一个只与它自己相差一个字符的新 **string.**

- **S[1,2,3..n]** 为 **String S;**
- **T[1,2,3..m]** 为 **String T;**
- **Z[1,2...k]** 为最少修改之后的 **String Z;**
    - 注意这里 **Z** 可以有多个正解，因为可以有多个最优的正确操作。
    - 类似于 **LCS**，**Z** 也可以是多条路径。

于是；

- 若 **S[n] == T[m]** ，则**Z[1,2.. k - 1]** 为 **S[n - 1]** 和 **T[m - 1]** 的解；
- 若 **S[n] != T[m]:**
    - 最优解为 **S[1,2 ..n]** 与 **T[1,2 .. m - 1]** 构造而成，
        - 删掉 **S[n]**

    - **OR** 增加 **T[m];**
  - 最优解为 **S[1,2 .. n - 1]** 与 **T[1,2 .. m]** 构造而成
    - 删掉 **T[m]**
    - **OR** 增加 **T[n];**
  - 最优解为 **S[1,2 .. n - 1]** 和 **T[1,2 .. m - 1]** 构造而成
    - 直接把 **S[n]** 和 **T[m] replace** 成一样的。

来自 **Stanford** 课件，**bottom - up** 的 **String** 构造法。我们定义 **D(i, j - 1)** 为 **insertion** 是因为 **i** 是外循环位置，**j** 是内循环位置；因此当 **i** 固定，相对于 **j** 取了前一个位置 **+** 新字符的时候是 **j** 的 **insertion**； 而 **D(i - 1, j)** 代表 **j** 在新循环中并没有增加长度，反而从 **i** 里删除了。

考虑到所有操作都在双循环内完成，调换循环位置并不会影响算法正确性，但是会赋予每次操作相反的意义，即调换等价互补操作 **"S + 1"** 和 **"T - 1".**

其实和算法导论的图是一个意思，方向不同而已。

Dan Jurafsky

The Edit Distance Table

N	9	8	9	10	11	12	11	10	9	**8**
O	8	7	8	9	10	11	10	9	8	9
I	7	6	7	8	9	10	9	8	9	10
T	6	5	6	7	8	9	8	9	10	11
N	5	4	5	6	7	8	9	10	11	10
E	4	3	4	5	6	7	8	9	10	9
T	3	4	5	6	7	8	7	8	9	8
N	2	3	4	5	6	7	8	7	8	7
I	1	2	3	4	5	6	7	6	7	8
#	**0**	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

如果这题每种操作附带了 **cost** ，要求最小的 **cost** 怎么办？

不难看出，**add** 和 **delete** 作为互补操作，其 **cost** 是一样的；即使不一样，我们也可以在两个 **String** 的构造过程中，总是选择更小的那个。

Dan Jurafsky

Weighted Min Edit Distance

- Initialization:

```
D(0,0) = 0
D(i,0) = D(i-1,0) + del[x(i)];     1 < i ≤ N
D(0,j) = D(0,j-1) + ins[y(j)];     1 < j ≤ M
```

- Recurrence Relation:

$$D(i,j) = \min \begin{cases} D(i-1,j) & + \text{del}[x(i)] \\ D(i,j-1) & + \text{ins}[y(j)] \\ D(i-1,j-1) & + \text{sub}[x(i),y(j)] \end{cases}$$

- Termination:

D(N,M) is distance

```
public class Solution {
    public int minDistance(String word1, String word2) {
        int n = word1.length();
        int m = word2.length();

        int[][] dp = new int[n + 1][m + 1];

        for(int i = 1; i <= n; i++){
            dp[i][0] = i;
        }
        for(int i = 1; i <= m; i++){
            dp[0][i] = i;
        }

        for(int i = 1; i <= n; i++){
            for(int j = 1; j <= m; j++){
                if(word1.charAt(i - 1) == word2.charAt(j - 1)){
                // 注意这步考虑 dp[i - 1][j - 1]同时再考虑
                // min(dp[i - 1][j], dp[i][j - 1]) + 1 也合乎逻辑
，一样可以 AC
                    dp[i][j] = dp[i - 1][j - 1];
                } else {
                    dp[i][j] = Math.min(dp[i - 1][j - 1],
                               Math.min(dp[i - 1][j],
                                        dp[i][j - 1])) + 1;
                }
            }
        }
        return dp[n][m];
    }
}
```

(Google, Snapchat面经)

Minimum insertions to form a palindrome

和 **Edit distance** 非常像，考虑到 **add/delete** 在构造 **string** 上的等价性质，问题的 **optimal substructure** 即为

- **dp[i][j] = substring(i,j)** 范围内，构造 **palindrome** 的最小编辑次数
  - 如果 **s[i] == s[j]**
    - **dp[i][j] = dp[i + 1][j - 1] (**不需要操作**)**
    - 同时考虑 **i , j** 相邻的情况
  - 如果 **s[i] != s[j]**，那么我们可以经 **add/delete** 操作构造出当前的 **s(i, j)**
    - **s(i + 1, j) + ADD**
    - **s(i, j - 1) + ADD**

注意这题不支持 **replace**，如果支持的话，**dp[i][j]** 还要看一个新状态，**dp[i + 1][j - 1]**

```
public class Main {
    private static int minEditDistance(String str){
        if(str == null || str.length() <= 1) return 0;

        int n = str.length();

        // dp[i, j] = min edit distance for substring (i , j);
        int[][] dp = new int[n][n];
        for(int i = 1; i < n; i++){
            for(int j = i; j >= 0; j--){
                if(j == i) {
                    dp[j][i] = dp[i][j] = 0;
                } else {
                    if(str.charAt(i) == str.charAt(j)){
                        dp[j][i] = dp[i][j] = (j + 1 == i) ? 0
                                : dp[j + 1][i - 1];
                    } else {
                        dp[j][i] = dp[i][j] = (j + 1 == i) ? 1
                                : Math.min(dp[j + 1][i], dp[j][i
 - 1]) + 1;
                    }
                }
            }
        }

        return dp[0][n - 1];
    }

    public static void main(String[] args){
        System.out.println(minEditDistance("geekforgeeks"));
    }
}
```

WildCard Matching

Hard 难度题，字符串 DP.

首先我们可以观察到最终结果只需要返回 true / false ，而不是很在乎具体到哪个位置的时候 wildcard 匹配了什么字符，所以是一个用 DP 的信号。

顺着这个思路想，这题具有这章其他 DP 类问题都非常相似的结构和 optimal substructure ，即都是 1 / 2 个 string 从小往大“生长”，每一步需要做 match 的判断构造出更长 string 的正解；以这题的结构，不难想出 dp 的数组结构就是 dp[i][j] 代表 s 的前 i 个字符串能否和 p 的前 j 个字符串成功匹配，以 dp[n][m] 为最终解。

剩下的问题就是，如何正确识别所有情况。

s[i] , p[j] 为 s , p 的第 i / j 个字符，略去 off-by-one 的 index 问题

- 当 **p[j] =** 常规字母时；
  - 如果 **s[i] == p[j]**，当前位置 **match;**
  - 同时如果之前的字符串 **dp[i - 1][j - 1]** 也 **match** ，则 **dp[i][j] match;**

- 当 **p[j]** 为 **'?'** 时；
  - 当前位置一定 **match**，只看 **dp[i - 1][j - 1]** 是否也 **match;**
- 当 **p[j]** 为 **'*'** 时；
  - 只 **match** 当前一个字符，看 **dp[i - 1][j - 1];**
    - **LeetCode** 测试了下，其实这行拿掉也是正确的
  - 不 **match** 任何字符，看 **dp[i][j - 1] (**忽略 **p[j]** 的存在**)**
  - **match** 多个字符，看 **dp[i - 1][j]**
    - 注意这里由于循环结构的关系，其实对于每一个 **j** ，我们会去考虑 **[0 , i-1]** 所有的可能性，所以可以用一个状态指代所有前面的 **match.**

最后要注意一下 **dp[0][0], dp[0][i]** 的初始化。

```java
public boolean isMatch(String s, String p) {
    if(s == null || p == null) return false;
    int n = s.length();
    int m = p.length();

    boolean[][] dp = new boolean[n + 1][m + 1];

    dp[0][0] = true;

    for(int i = 1; i <= m; i++){
        if(p.charAt(i - 1) == '*') dp[0][i] = true;
        else break;
    }

    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= m; j++){
            if(p.charAt(j - 1) == '?' ||
               s.charAt(i - 1) == p.charAt(j - 1)){
                dp[i][j] = dp[i - 1][j - 1];
            } else if(p.charAt(j - 1) == '*'){
                dp[i][j] = dp[i - 1][j] || dp[i][j - 1];
            }
        }
    }

    return dp[n][m];
}
```

Regular Expression Mathcing

也是 Hard 题，长得还和 Wildcard Matching 特别像。

最大的不同是，在这里 '*' 号是和其前一个字符有联系的，和其前一个字符一起，代表着“多个或0个星号前面的字符”。 这里我们需要假设 p 不会以星号开始，也不会有连续多个星号出现，不然现有题意描述是无法解决这些问题的。

- 遇到常规字母和 **'.'** 的处理和上一题没有任何区别；
- 遇到 **p[j]** 为星号时：
  - 如果 **p[j - 1]** 是 **'.',** 那么这个星号
    - **dp[i - 1][j]** 以当前星号匹配一个或多个多个字符；
    - **dp[i][j - 1]** 只让 **p[j - 1]** 匹配，当前星号不匹配字符；
    - **dp[i][j - 2]** 同下。

- 否则，星号位置能否匹配取决于 **dp[i][j - 2]**，即让**(p[j - 1] + 当前星号)**都不匹配任何字符。

以这两道题看，**dp[i - 1][j]** 都代表着 **“以 p 当前 * 字符，匹配 s 的[1,n]长度字符”**

这题**dp[0][i]**初始化和 **Wildcard** 不太一样，因为会有 **c*a*b** 这种情况，多个星号跳着出现，不要立刻 **break** 掉，而要扫到底，**dp[0][i]** 要看 **dp[0][i - 2];**

```java
    public boolean isMatch(String s, String p) {
        if(s == null || p == null) return false;
        int n = s.length();
        int m = p.length();

        boolean[][] dp = new boolean[n + 1][m + 1];

        dp[0][0] = true;

        for(int i = 2; i <= m; i++){
            if(p.charAt(i - 1) == '*') dp[0][i] = dp[0][i - 2];
        }


        for(int i = 1; i <= n; i++){
            for(int j = 1; j <= m; j++){
                char chars = s.charAt(i - 1);
                char charp = p.charAt(j - 1);

                if(charp == '.' || charp == chars){
                    dp[i][j] = dp[i - 1][j - 1];
                } else if (charp == '*'){
                    char charPrev = p.charAt(j - 2);
                    if(charPrev == '.' || charPrev == chars){
                        dp[i][j] = dp[i - 1][j] || dp[i][j - 2]
|| dp[i][j - 1];
                    } else {
                        dp[i][j] = dp[i][j - 2];
                    }
                }
            }
        }

        return dp[n][m];
    }
```

Word Break

这是一道很像 rod-cutting 还有 palindrome paritioning 的题，利用的结构也一样。

要保证算法的正确性，我们要迭代考虑题目中所有的 substring 组合，不然可能会漏掉正解。考虑到题目可能的 substring 数量，这题的时间复杂度至少为 O(n^2)，因此我们能做的就是利用好 substring 相互覆盖的地方避免重复计算。

```
public class Solution {
    public boolean wordBreak(String s, Set<String> wordDict) {
        int n = s.length();
        boolean[][] canBreak = new boolean[n][n];
        for(int i = 0; i < n; i++){
            for(int j = i; j >= 0; j--){
                String str = s.substring(j, i + 1);
                if(wordDict.contains(str)) canBreak[i][j] = canB
reak[j][i] = true;
                if(j > 0 && canBreak[j][i] && canBreak[0][j - 1]
) canBreak[0][i] = canBreak[i][0] = true;
            }
        }

        return canBreak[0][n - 1];
    }
}
```

在如上代码 AC 之后，思考之后可以发现两个很明显的优化：

optimal substructure 是，如果 [0, j] 可以 break 并且 [j + i , i] 在字典里，那么 [0 , j] 就可以 break.

- 我们不需要存储所有区间是否可以 **break**，只取从 **index = 0** 开始到 **i** 为止是否可以就行了；空间优化成了 **O(n);**
- 因此如果回头扫到了当前位置可以 **break** 的时候，我们就可以做 **early termination.**
- 同时对于每个考虑的中间位置 **j** ，如果在 **j** 上不能 **break**，那么取 **substring** 和检查字典的必要都没有。

```
    public boolean wordBreak(String s, Set<String> wordDict) {
        int n = s.length();
        boolean[] canBreak = new boolean[n + 1];
        canBreak[0] = true;

        for(int j = 1; j <= n; j++){
            for(int i = j; i >= 0; i--){
                if(!canBreak[i]) continue;
                String str = s.substring(i, j);
                if(wordDict.contains(str)) canBreak[j] = true;
            }
        }

        return canBreak[n];
    }
```

Word Break II

这题有简单暴力的 **dfs + backtracking**，然而为了优化计算时间，也要利用备忘录法和记忆化搜索。

这题做出来容易，AC 有点难，因为用时卡的厉害，有坑爹的 test case :

【s ="aaaaaaaaaaaaaaaaa...aaaa" , dict = "a", "aa", "aaa", "aaaa"】

硬搜的时间复杂度为 O(2^n)，搜索树结构见算法导论的 rod-cutting.

最容易想到的几种优化方式：

- 预处理，建 **boolean[][]** 表示所有可能区间内是不是 **word**，方便**dfs** 时判断要不要进入下一层循环；
- 记录字典里最长单词长度作为 **step size**，**dfs** 每次循环寻找位置时限制最远的搜索距离；（用于优化 **s** 相对于 **word** 非常长的情况）

然而单独靠以上两种优化都还不够。。

https://leetcode.com/discuss/80266/java-dfs-memoization-dfs-and-pruning-solutions-with-analysis

事实证明在这题里，直接用 **HashSet + substring** 检查字典，要比预处理计算 **boolean[][]** 快。 **(8ms vs. 22ms)**

O(len(wordDict) ^ len(s / minWordLenInDict))

Because there're len(wordDict) possibilities for each cut，同时空间占用可能会比较大。

```
public class Solution {
    public List<String> wordBreak(String s, Set<String> wordDict)
 {
        int n = s.length();
        boolean[][] isWord = new boolean[n][n];
        int maxLength = 0;

        for(String str : wordDict){
            maxLength = Math.max(maxLength, str.length());
        }

        return dfs(s, wordDict, new HashMap<Integer, List<String
>>() , 0, maxLength);
    }

    private List<String> dfs(String s, Set<String> wordDict, Has
hMap<Integer, List<String>> map, int start, int maxLength){
        if(map.containsKey(start)) return map.get(start);

        List<String> rst = new ArrayList<String>();
        if(start >= s.length()) rst.add("");
        for(int i = start; i - start <= maxLength && i < s.lengt
h(); i++){
            String str = s.substring(start, i + 1);

            if(wordDict.contains(str)){
                List<String> afterWards = dfs(s, wordDict, map,
i + 1, maxLength);
```

```
                for(String next : afterWards){
                    if(i + 1 < s.length()) rst.add(str + " " + n
ext);
                    else rst.add(str);
                }
            }
        }

        map.put(start, rst);
        return rst;
    }
}
```

Interleaving String

首先从题目结构来看，和这章的前几道字符串 **DP** 非常相似，都是一个字符串 **"**构造问题**"**，即用小的 **substring** 通过生长和拼接，构造出更大的目标 **string.** 一般这类问题有天然的 **bottom-up** 思路，当然，**top-bottom** 的递归思路也是完全可行的。

首先贴一个自己写的暴力解法，三维 dp，dp[i][j][k] 代表着"s1 的前 i 个字符和 s2 的前 j 个字符，能否构造成 s3 的前 k 个字符"。

下面代码时间复杂度 O(s1.len * s2.len * s3.len)，可以 AC，但是显然还有优化的空间。

```
public class Solution {
    public boolean isInterleave(String s1, String s2, String s3)
{
        if(s1 == null || s2 == null || s3 == null) return false;
        if(s3.length() != s1.length() + s2.length()) return false
;
        if(s1.length() == 0) return s2.equals(s3);
        if(s2.length() == 0) return s1.equals(s3);

        int s1Len = s1.length();
        int s2Len = s2.length();
        int s3Len = s3.length();
```

```
        // dp[i][j][k] if the first i chars from s1 and first j
chars from s2
        // can interleave to produce first k chars of s3

        boolean[][][] dp = new boolean[s1Len + 1][s2Len + 1][s3L
en + 1];

        dp[0][0][0] = true;

        for(int k = 1; k <= s3Len; k++){
            char char3 = s3.charAt(k - 1);

            for(int i = 1; i <= k && i <= s1Len; i++){
                char char1 = s1.charAt(i - 1);
                for(int j = 1; j <= k && j <= s2Len; j++){
                    char char2 = s2.charAt(j - 1);

                    if(char1 == char3 && k - i >= 0 && k - i <=
s2Len && dp[i - 1][k - i][k - 1]) {
                        dp[i][k - i][k] = true;
                    }
                    if(char2 == char3 && k - j >= 0 && k - j <=
s1Len && dp[k - j][j - 1][k - 1]) {
                        dp[k - j][j][k] = true;
                    }

                }
            }
        }

        return dp[s1Len][s2Len][s3Len];
    }
}
```

简单观察之后发现正确答案的限制条件：**k = i + j**，换句话说，**k** 可以用 **i + j** 表示出来，因此不需要以 **k** 作为单独维度遍历。

同时我们也要处理好 **i = 0** 和 **j = 0** 的初始化条件。

时间空间复杂度 O(s1.len * s2.len)

```
public class Solution {
    public boolean isInterleave(String s1, String s2, String s3)
{
        if(s1 == null || s2 == null || s3 == null) return false;
        if(s3.length() != s1.length() + s2.length()) return false
;
        if(s1.length() == 0) return s2.equals(s3);
        if(s2.length() == 0) return s1.equals(s3);

        int s1Len = s1.length();
        int s2Len = s2.length();
        // dp[i][j][k] if the first i chars from s1 and first j
chars from s2
        // can interleave to produce first i + j chars of s3

        boolean[][] dp = new boolean[s1Len + 1][s2Len + 1];

        dp[0][0] = true;
        for(int i = 0; i < s1Len; i++){
            if(s1.charAt(i) == s3.charAt(i) && dp[i][0]) dp[i + 1
][0] = true;
        }
        for(int i = 0; i < s2Len; i++){
            if(s2.charAt(i) == s3.charAt(i) && dp[0][i]) dp[0][i
 + 1] = true;
        }

        for(int i = 1; i <= s1Len; i++){
            for(int j = 1; j <= s2Len; j++){
                char char1 = s1.charAt(i - 1);
                char char2 = s2.charAt(j - 1);
                char char3 = s3.charAt(i + j - 1);

                if(char1 == char3 && dp[i - 1][j])  dp[i][j] = t
rue;
                if(char2 == char3 && dp[i][j - 1])  dp[i][j] = t
rue;
            }
```

```
        }

        return dp[s1Len][s2Len];
    }
}
```

Bomb Enemies

Bomb Enemy

非常明显的记忆化搜索，核心问题只有一个：

- 给定一行/列，如何最高效地计算出每个位置在当前 行/列 上能炸到的最大值。

跑了个简单 Case，结论是，雷可以穿人。和炸弹人差不多。。。

于是下图这个简单暴力的方法就出来了，因为写的比较匆忙没有进行任何简洁性的优化，所以看着就不能忍，需要改进。

思路如下：

- 建立 rowMax[][] 和 colMax[][] 矩阵，记录对于每一个位置 (i, j) ，其在 row / col 上能炸到的最大敌人数
- 遍历每一个 row / col
- 先建一个数组，从左向右扫，保存一个 maxCount 变量，遇到 'W' 就归零，遇到 'E' 就增加，遇到 '0' 不变，一路上把每个位置赋值成 maxCount;
- 然后再从右往左，在每个位置上取新的 maxCount (因为对于每个区间，这次一定会先看到最大的 count 数)，遇到 '0' 就赋值，遇到 'E' 只取 maxCount，赋值0 (不能在敌人头上扔炸弹)，遇到 'W' maxCount 归零，赋值也是0.

因此对于每一行，复杂度都是 O(cols)，对于每一列，复杂度都是 O(rows)，整个预处理的过程用时为 O(rows * cols)，最后的扫描也是 P(rows * cols).

```
public class Solution {
    public int maxKilledEnemies(char[][] grid) {
        if(grid == null || grid.length == 0) return 0;

        int rows = grid.length;
        int cols = grid[0].length;

        int[][] rowMax = new int[rows][cols];
```

```
        int[][] colMax = new int[rows][cols];

        for(int i = 0; i < rows; i++){
            int[] count = new int[cols];
            int enemyCount = 0;
            for(int j = 0; j < cols; j++){
                if(grid[i][j] == 'W') enemyCount = 0;
                if(grid[i][j] == 'E') enemyCount ++;

                count[j] = enemyCount;
            }
            int maxCount = 0;
            for(int j = cols - 1; j >= 0; j--){
                if(grid[i][j] == '0') {
                    maxCount = Math.max(maxCount, count[j]);
                    rowMax[i][j] = maxCount;
                }
                if(grid[i][j] == 'W') {
                    maxCount = 0;
                    rowMax[i][j] = 0;
                }
                if(grid[i][j] == 'E') {
                    maxCount = Math.max(maxCount, count[j]);
                    rowMax[i][j] = 0;
                }
            }
        }

        for(int i = 0; i < cols; i++){
            int[] count = new int[rows];
            int enemyCount = 0;
            for(int j = 0; j < rows; j++){
                if(grid[j][i] == 'W') enemyCount = 0;
                if(grid[j][i] == 'E') enemyCount ++;

                count[j] = enemyCount;
            }
            int maxCount = 0;
            for(int j = rows - 1; j >= 0; j--){
                if(grid[j][i] == '0') {
```

```
                    maxCount = Math.max(maxCount, count[j]);
                    colMax[j][i] = maxCount;
                }
                if(grid[j][i] == 'W') {
                    maxCount = 0;
                    colMax[j][i] = 0;
                }
                if(grid[j][i] == 'E') {
                    maxCount = Math.max(maxCount, count[j]);
                    colMax[j][i] = 0;
                }
            }
        }

        int max = 0;
        for(int i = 0; i < rows; i++){
            for(int j = 0; j < cols; j++){
                max = Math.max(max, rowMax[i][j] + colMax[i][j])
;
            }
        }

        return max;
    }
}
```

上面我第一次 **AC** 的代码能过，但是代码量略大，而且空间使用也不算经济。下面是参考 **LC** 论坛上的解法：

- 核心思想是，**row** 和 **col** 的缓存只会在**(**上**/**左**)**就是 **"W"** 时需要更新
- 因此初始化或者**(**上**/**左**)**是墙的时候，可以计算一个临时结果，在遇到另一个 **"W"** 时停止；
- 这个缓存在遇到新一个**(**上**/**左**)**是 **"W"** 的格子之前，都是有效的，无需重复计算。

- 考虑到循环是 **row , col** 的顺序，我们的 **rowCache** 一个变量就够了，但是 **colCache** 得存个数组才行。

```
    public int maxKilledEnemies(char[][] grid) {
        if(grid == null || grid.length == 0) return 0;
        int rows = grid.length;
        int cols = grid[0].length;

        int max = 0;
        int rowCache = 0;
        int[] colCache = new int[cols];

        for(int i = 0; i < rows; i++){
            for(int j = 0; j < cols; j++){
                if(j == 0 || grid[i][j - 1] == 'W'){
                    rowCache = 0;
                    for(int k = j; k < cols && grid[i][k] != 'W'
; k++){
                        rowCache += grid[i][k] == 'E' ? 1 : 0;
                    }
                }

                if(i == 0 || grid[i - 1][j] == 'W'){
                    colCache[j] = 0;
                    for(int k = i; k < rows && grid[k][j] != 'W'
; k++){
                        colCache[j] += grid[k][j] == 'E' ? 1 : 0
;
                    }
                }

                if(grid[i][j] == '0') max = Math.max(max, rowCac
he + colCache[j]);
            }
        }

        return max;
    }
```

8/2，背包问题

挺经典的 dp 问题，就是不知道为啥 leetcode 上没有。

特点一：至少要用目标值作为一个 **DP** 维度

特点二：对具体元素敏感(比如 **01** 背包)，但对同一 **totalSize / totalValue** 来讲，有时候如何构造而来的具体路线不重要。这是由元素特性和背包 **size** 的约束条件决定的，也是有别于其他种类 **DP** 的一个重要区别。

比如 **Combination Sum IV**，就是典型的 **“**对最终结果敏感，对元素个数不敏感**”** 的问题，每次可以取任意元素，任意个，因此只用 **dp[sum]** 一个维度做 **dp** 就够了。

在只取 **min/max** 的时候我们不同的循环顺序都能保证正确答案；但是求解的数量时，要以 **target value** 为外循环。

Backpack

先从暴力搜索开始，结构图如下：

对于每个元素都会衍生出 (取/不取) 两种选择，所有可能的策略则是从最左面到最右面的一条 path，也即 binary tree 从 root 到 leaf node 的路径个数。

显然，路径个数 = 叶节点个数 = 2^n.

接下来的观察是关于题目性质的，可以发现我们最关注的是 “sum”，而不太关心这个 sum 是怎么来的。比如 sum = 10 的时候，我们可以取【2,3,5】 或者 【3,7】，虽然是两种不同的解，但是最终效果完全一样，也就是完全一样的“状态”。因此，如果我们用 sum 来定义状态，就会发现很多的 overlap subproblems，可以进一步采用记忆化搜索进行优化。

另一个需要注意的地方是，每个元素只能取一次，只有 “取” 或者 “不取” 的选择，因此当前 index 上的状态只能取决于之前的状态，而不能重复考虑当前元素。

dp[i][sum] = 前 i 个元素里我们能不能凑出来 sum.

- **dp[i][sum]** 要么取决于 **dp[i - 1][sum] (**不取当前元素**)**
- 要么取决于 **dp[i - 1][sum - nums[i]] (**取当前元素**)**

- 其中每一行 **i** 都只考虑前一行 **i - 1** 的值。

```
public class Solution {
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A: Given n items with size A[i]
     * @return: The maximum size
     */
    public int backPack(int m, int[] A) {
        // write your code here
        // dp[i][j] for the first i elements, can we make sum of
 j
        boolean[][] dp = new boolean[A.length + 1][m + 1];

        for(int i = 0; i <= A.length; i++) dp[i][0] = true;

        for(int i = 1; i <= A.length; i++){
            for(int j = 1; j <= m; j++){
                if(j - A[i - 1] >= 0)
                    dp[i][j] = (dp[i - 1][j] || dp[i - 1][j - A[
i - 1]]);
                else dp[i][j] = dp[i - 1][j];
            }
        }

        for(int i = m; i >= 0; i--)
            if(dp[A.length][i]) return i;

        return -1;
    }
}
```

考虑到我们只需要存最多两行的结果，滚动数组优化就显而易见了。

其实在背包九讲里面也有写过只存一行， 新一行的结果每次从右往左扫的，更经济些。简易程度上我还是更喜欢用取 mod 的滚动数组写法。

```
public class Solution {
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A: Given n items with size A[i]
     * @return: The maximum size
     */
    public int backPack(int m, int[] A) {
        // write your code here
        // dp[i][j] for the first i elements, can we make sum of
 j
        boolean[][] dp = new boolean[2][m + 1];

        for(int i = 0; i <= A.length; i++) dp[i % 2][0] = true;

        for(int i = 1; i <= A.length; i++){
            for(int j = 1; j <= m; j++){
                if(j - A[i - 1] >= 0)
                     dp[i % 2][j] = (dp[(i - 1) % 2][j] || dp[(i
 - 1) % 2][j - A[i - 1]]);
                else dp[i % 2][j] = dp[(i - 1) % 2][j];
            }
        }

        for(int i = m; i >= 0; i--)
            if(dp[A.length % 2][i]) return i;

        return -1;
    }
}
```

Backpack II

这次每个元素除了 size 之外也具有 value，就变成了更典型的 01 背包问题。

问题的结构依然具有 01 背包的性质，选一条 path 使得最终总价值最大，path 总数为 2^n. 同时对于同一个总的空间占用 totalSize 或者 totalValue，我们并不在乎它是怎么构造出来的，只要不重复选元素就行。

因此我们 dp 结构一致，而采用 int[][] 来记录

dp[i][j] 包里有 **j** 的空间，可以取前 **i** 个元素时，所能获得的最大收益。

同时因为每次新迭代循环中，**i** 是不会走回头路的，所以状态与状态之间可以在不违反题意的情况下衔接起来。

```
public class Solution {
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A & V: Given n items with size A[i] and value V[i]
     * @return: The maximum value
     */
    public int backPackII(int m, int[] A, int V[]) {
        // write your code here
        // dp[i][j] within first i elements with space j, maximu
m profit gain
        int[][] dp = new int[A.length + 1][m + 1];

        for(int i = 1; i <= A.length; i++){
            for(int j = 1; j <= m; j++){
                if(j - A[i - 1] >= 0){
                    dp[i][j] = Math.max(dp[i - 1][j] , dp[i - 1]
[j - A[i - 1]] + V[i - 1]);
                } else {
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }

        return dp[A.length][m];
    }
}
```

Perfect Squares

仔细观察一下这题:

- 我们要凑出来一个和正好是 **n** 的选择组合；

- 能选的元素是固定数量的 **perfect square (有的会超)**
- 一个元素可以选多次；

这就是背包啊！

```
public class Solution {
    public int numSquares(int n) {
        // All perfect squares less than n
        int[] dp = new int[n + 1];
        Arrays.fill(dp, n + 1);
        dp[0] = 0;

        for(int i = 1; i <= n; i++){
            for(int j = 1; j * j <= n; j++){
                if(i - j * j >= 0) dp[i] = Math.min(dp[i], dp[i
- j * j] + 1);
            }
        }

        return dp[n];
    }
}
```

Coin Change

背包类问题的马甲题。

- 每个元素可以取任意次；
- 只问最小的硬币数，就类似于 **climbing stairs**，只靠 **sum** 一个维度就可以了，但是循环依然是两层。

比较直观的写法是这样(内外循环的顺序无所谓，但是最好从 amount 那个维度开始)

```
public class Solution {
    public int coinChange(int[] coins, int amount) {
        // dp[n] = min number of coins to make amount n;
        int[] dp = new int[amount + 1];
        Arrays.fill(dp, amount + 1);
        dp[0] = 0;

        for(int i = 1; i <= amount; i++){
            for(int j = 0; j < coins.length; j++){
                if(i - coins[j] >= 0) dp[i] = Math.min(dp[i], dp
[i - coins[j]] + 1);
            }
        }

        return dp[amount] == amount + 1 ? -1 : dp[amount];
    }
}
```

比较优化和取巧的写法是这样，以面值

```
public class Solution {
    public int coinChange(int[] coins, int amount) {
        // dp[n] = min number of coins to make amount n;
        int[] dp = new int[amount + 1];
        Arrays.fill(dp, amount + 1);
        dp[0] = 0;

        for(int j = 0; j < coins.length; j++){
            for(int i = coins[j]; i <= amount; i++){
                dp[i] = Math.min(dp[i], dp[i - coins[j]] + 1);
            }
        }

        return dp[amount] == amount + 1 ? -1 : dp[amount];
    }
}
```

Backpack III

和背包 II 看起来结构一样，但是这次多了一个新条件：同一个元素可以取多次。

然而这个条件会带来一个重要的改变：元素的具体位置和数量都不重要了。我们可以直接扔掉这个维度，就像 **Coin Change** 和 **Combination Sum IV** 一样。

因此可以看到，只有对元素的选择(位置) / (数量)有限制性条件的 **DP**，才会需要另一个维度。

- 对位置有要求的，用 **i** 代表“前 **i** 个元素”；
- 对数量有要求的，用 **j** 代表“选 **k** 个元素”；
- 两个全有要求的，就两个维度都加上去。
- 两个维度都加上去，我们就有了 **k sum** 问题。

```
public class Solution {
    /**
     * @param A an integer array
     * @param V an integer array
     * @param m an integer
     * @return an array
     */
    public int backPackIII(int[] A, int[] V, int m) {
        // Write your code here
        // Maximum value we can get
        int[] dp = new int[m + 1];

        for(int i = 0; i < A.length; i++){
            for(int j = 1; j <= m; j++){
                if(j - A[i] >= 0)
                    dp[j] = Math.max(dp[j], dp[j - A[i]] + V[i])
;
            }
        }

        return dp[m];
    }
}
```

Combination Sum IV

原理和 climbing stairs 差不多，建一个大小等于 target + 1 的 array 代表多少种不同的方式跳上来，依次遍历即可。

时间复杂度 O(n * target)

注意这里内外循环的顺序不能错，要先按 **sum** 从小到大的顺序看，然后才是遍历每个元素。因为所谓 **bottom-up**，是相对于 **sum** 而言的，不是相对于已有元素的 **index** 而言的。

可以看到，在只取 **min/max** 的时候我们不同的循环顺序都能保证正确答案；但是求解的数量时，要以 **target value** 为外循环。

```
public class Solution {
    public int combinationSum4(int[] nums, int target) {
        // dp[sum] = number of ways to get sum
        int[] dp = new int[target + 1];

        // initialize, one way to get 0 sum with 0 coins
        dp[0] = 1;

        for(int j = 1; j <= target; j++){
            for(int i = 0; i < nums.length; i++){
                if(j - nums[i] >= 0) dp[j] += dp[j - nums[i]];
            }
        }

        return dp[target];
    }
}
```

K sum

背包类问题的扩展版本~

我们关注的维度有三个：

- 前 **i** 个元素，因为一个元素只能选一次；
- 选了 **j** 个元素，因为我们要求选 **k** 个数；
- 总和 **sum**，用于每次试图添加新元素时用来查询。

时间：**O(len * k * target)**，三重循环

空间：**O(k * target)**

```
public class Solution {
    /**
     * @param A: an integer array.
     * @param k: a positive integer (k <= length(A))
     * @param target: a integer
     * @return an integer
     */
    public int kSum(int A[], int k, int target) {
        // int[i][j][k] : number of ways to get sum of k by pick
ing
        //                j numbers from first i numbers
        int[][][] dp = new int[2][k + 1][target + 1];

        dp[0][0][0] = 1;
        dp[1][0][0] = 1;

        for(int i = 1; i <= A.length; i++){
            for(int j = 1; j <= k; j++){
                for(int v = 1; v <= target; v++){
                    // i has an offset of 1, becasue when i = 1
                    // we are actually referring to index 0
                    if(v >= A[i - 1]){
                        dp[i % 2][j][v] = dp[(i - 1) % 2][j][v]
+
                                          dp[(i - 1) % 2][j - 1]
[v - A[i - 1]];
                    } else {
                        dp[i % 2][j][v] = dp[(i - 1) % 2][j][v];
                    }
                }
            }
        }
        return dp[A.length % 2][k][target];
    }
}
```

(G) Max Vacation

http://www.1point3acres.com/bbs/thread-158696-1-1.html

First round: 给定一堆城市，boolean array表示两个城市是否可以在一个周末直飞，然后给你每周在每个城市的holiday days，求一年最多能歇几天.并求出城市方案。

dp, dp[i][j] 表示在 第i week停留在j 城市能获得的最大放假天数，dp[i][j] = max(dp[i][j], dp[i - 1][k]) if k j直达 + holidays[i][j];

(11/4新增) AST 子树结构 DP

一个朋友看面经看到的，说是 **VMWare** 某次 **on campus interview** 的问题，我觉得挺有意思，就补充到这个分类里了~

给一个字符串，字符串是一个表达式.

比如： **T & F ^ F | T & T ^ T**。

T , F 分别表示 **true OR false**， 操作符有 **and, or, xor**。 给你任意一个表达式，你可以任意加括号组合，问一个有多少组合的方法使这个表达式能得到 **True**。

比如 **T ^ F & T** 可以写成 （**T ^ F) & T** 或者 **T ^ (F & T)** 所以一共是两种。

这个问题比较 simple 而 naive 的切入方式，就是先从建立 abstract syntax tree (AST) 开始，像我们做 reverse polish notation 求值一样，根据 operator 和 operand 来建立树形结构。

在这里我们不同的加括号方式，导致了不同的运算顺序，最终导致了不同的 **AST** 子树结构。如果我们对这个 **AST** 去做 **in-order traversal**，会发现所有解的 **in-order** 结果都和不加括号的原始输入一样。

画出来的话，第一个就是

```
      &
     / \
    ^   T
   / \
  T   F
```

第二个是

```
      ^
     / \
    T   &
       / \
      T   F
```

那么这个问题在结构上最简单粗暴的定义，就是 -- 枚举出所有可能的 AST 树，并且计算出到底有多少个树可以 evalute true. 暴力解就有了。

下一步是，寻找可能的 **overlapping subproblem**，和 **recurrence relation**

常见的做法就是 "加/减一个元素"，比如说刚才的输入是 T ^ F & T ，加一个 F & ，变成 F & T ^ F & T.

这个时候立刻可以看出，一种划分方式是 F & (T ^ F & T)，括号里的部分就是刚才计算的问题。因此可以发现，这个问题虽然每一个 AST 是 disjoint 的树形结构，但是树与树之间，是存在 overlapping subproblem 的；体现在 AST 上，就是 identical subtree; 体现在输入 String 上，就是 identical subarray.

因此，这一定是一个可以靠动态规划优化的问题。

考虑到我们的操作符有三种，"AND"，"OR" 和 "XOR"，这题和其他的问题还稍微有点不一样：AND 操作符在两边找 True，OR 操作符在两边找到 True 就行，XOR 要找一边 True 一边 False. 所以这题的子问题划分好了之后，我们需要同时记录这个 subproblem 里，到底有多少种 evaluate true/false 的解。(对于固定子树 size N，这其实是一个常数)

对于给定的 **String input**，这题的 **top-down** 解决步骤是：

> - 依次枚举每一个操作符的位置作为 **root**，同时以该操作符为分界线，把输入分成左右两个子问题；
> - 根据左右子问题分别的 **true/false** 数量，还有当前操作符的类型，计算出当前 **subtree** 结构的 **true/false** 数量；
> - 一个 **subproblem** 的 **true/false** 数量，是其所有以操作符为 **root** 的划分结果的和；
> - 所有子问题都可以 **top-down with memoization**，用记忆化搜索，字符串做 **key**，**true/false** 解的数量做 **value.**
> - **bottom-up** 的迭代方式也可以，考虑到子串的连续性，可以直接用 **int[i][j]** 的方式来定义，从第 **i** 到第 **j** 个 **index** 上的子问题里，有多少个 **true/false** 就行了。

一个记忆化搜索的例子，同一种做法还有优化空间，比如用区间 **int start, int end** 来定义一个子问题。这种写法比较省事，不过毕竟还是生成了很多个 **substring** 的 **copy**，还有 **string** 做 **key** 在 **hashmap** 中查找的开销。

> - 每个子问题对应一个 **tuple**，其 **T/F** 各自一个 **count;**
> - 对于左右两个子问题，总共可能的组合数为 **leftTotal * rightTotal;**
> - 当操作符为 **'&'** 时，当前位置的 **trueCnt = leftTrue * rightTrue;**
> - 当操作符为 **'|'** 时，当前位置的 **falseCnt = leftFalse * rightFalse;**
> - 当操作符为 **'^'** 时，当前位置的 **trueCnt = leftTrue * rightFalse + leftFalse * rightTrue;**
> - 在每个操作符上，**trueCnt + falseCnt = totalCnt;**

```
public class Solution {
    private static class Tuple{
        int trueCnt;
        int falseCnt;
```

```
        public Tuple(int trueCnt, int falseCnt){
            this.trueCnt = trueCnt;
            this.falseCnt = falseCnt;
        }
    }

    private static Tuple evaluate(String str, HashMap<String, Tu
ple> map){
        // Assuming input string is valid;
        // Thus all boolean is at EVEN index; all operator at OD
D index.

        // base case
        if(str.length() == 1){
            return (str.equals("T")) ? new Tuple(1,0)
                                     : new Tuple(0,1);
        }

        if(map.containsKey(str)){
            System.out.println("Cache hit for substring : " + st
r);
            return map.get(str);
        }

        Tuple tuple = new Tuple(0,0);

        for(int i = 1; i < str.length(); i += 2){
            Tuple left = evaluate(str.substring(0, i), map);
            Tuple right = evaluate(str.substring(i + 1), map);

            char operator = str.charAt(i);
            int curTotal = (left.falseCnt + left.trueCnt) *
                           (right.falseCnt + right.trueCnt);

            int curTrue = 0;
            int curFalse = 0;

            if(operator == '&'){
                curTrue = left.trueCnt * right.trueCnt;
                curFalse = curTotal - curTrue;
```

```
            } else if(operator == '|'){
                curFalse = left.falseCnt * right.falseCnt;
                curTrue = curTotal - curFalse;
            } else if(operator == '^'){
                curTrue = left.trueCnt * right.falseCnt +
                              left.falseCnt * right.trueCnt;
                curFalse = curTotal - curTrue;
            }

            tuple.trueCnt += curTrue;
            tuple.falseCnt += curFalse;
        }

        map.put(str, tuple);
        return tuple;
    }

    public static void main(String[] args) {
        String str1 = "T&F^T";
        String str2 = "T&F^T|F";
        String str3 = "T&F&T|T^F";
        String str4 = "T&F^T^T&F&T|T^F";

        HashMap<String, Tuple> map = new HashMap<>();
        Tuple tuple = evaluate(str3, map);

        System.out.println(tuple.trueCnt);
    }
}
```

Different Ways to Add Parentheses

先说一个自己的做法，写的是搜索，对于每一个 input string ，我们扫描并建立两个 list; 一个是 operands，一个是 operators. 每次 dfs 的时候挑一对数字和一个操作符，计算，修改 list 做 dfs，返回之后再把 list 改回来。

过了 15/25 个 test case 之后卡在了 "2*3-4*5" 上面，原因是多了一个额外的 -14，因为两个 subproblem 加括号的方式是一样的，顺序不同而已。

所以暴力搜索的问题在于，如果前面和后面加括号的方式是一样的，需要额外手段来判断“重复”。

而DFS 中想 cache 一个 List<>，远远不如 String 方便。

如果实在追求暴力到底，就干脆对每一个状态都搞序列化，记录当前的 operands & operators ，遇到重复的状态就直接返回了。

因此为了避免 dfs + backtracking 可能遇到的 overlap subproblem 返回多个结果的问题，可以先直接 divide & conquer，因为这个搜索结构是树状的，天生的 disjoint.

根据这个思想，我们可以以“操作符”为分隔，借鉴编译器和 reverse polish notation 中的 "expression tree" 来进行计算，结构如下：

这样左右子树都进行的完美的分隔，而且应为 input 为 string ，也非常容易对子问题进行记忆化搜索。

Divide & Conquer, 8ms，超过 41.50%.

```
public class Solution {
    public List<Integer> diffWaysToCompute(String input) {

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

        for(int i = 0; i < input.length(); i++){
            if(!Character.isDigit(input.charAt(i))){
                char operator = input.charAt(i);

                List<Integer> left = diffWaysToCompute(input.sub
string(0, i));
                List<Integer> right = diffWaysToCompute(input.su
bstring(i + 1));

                for(int num1 : left){
                    for(int num2 : right){
                        if(operator == '+') rst.add(num1 + num2)
;
                        if(operator == '-') rst.add(num1 - num2)
;
                        if(operator == '*') rst.add(num1 * num2)
;
                    }
                }
            }
        }

        if(rst.size() == 0) rst.add(Integer.parseInt(input));

        return rst;
    }
}
```

带记忆化搜索，**3ms**，超过 **91.61%**

```
public class Solution {
    public List<Integer> diffWaysToCompute(String input) {
        return helper(input, new HashMap<String, List<Integer>>(
```

```
));
    }

    private List<Integer> helper(String str, HashMap<String, Lis
t<Integer>> map){
        if(map.containsKey(str)) return map.get(str);

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

        for(int i = 0; i < str.length(); i++){
            char chr = str.charAt(i);
            if(!Character.isDigit(chr)){
                List<Integer> leftList = helper(str.substring(0,
 i), map);
                List<Integer> rightList = helper(str.substring(i
 + 1), map);

                for(int leftNum : leftList){
                    for(int rightNum : rightList){
                        if(chr == '+') list.add(leftNum + rightN
um);
                        if(chr == '-') list.add(leftNum - rightN
um);
                        if(chr == '*') list.add(leftNum * rightN
um);
                    }
                }
            }
        }

        if(list.size() == 0) list.add(Integer.parseInt(str));

        map.put(str, list);
        return list;
    }
}
```

LinkedList，链表

6/9, LinkedList，链表基础，反转与删除

写了这么久奇形怪状的数据结构，终于开始灌水了。。哈哈哈哈

LinkedList 常用操作：

- 翻转
- 找 kth element
- partition
- 排序
- 合并

常用技巧：

- dummy node
- 快慢指针
- 多个 ptr

Reverse Linked List

这种 swap 中有个很有意思的性质，如果下一行的等号左面等于上一行的等号右边，并且以第一行的等号左边结尾，那最后实际做的事情是 swap 了第一行等号右，和最后一行等号左。

```
public class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode prev = null;
        while(head != null){
            ListNode temp = head.next;
            head.next = prev;
            prev = head;
            head = temp;
        }
        return prev;
    }
}
```

Reverse Linked List II

这题还挺 **tricky** 的，写了好几次。

```
public class Solution {
    public ListNode reverseBetween(ListNode head, int m, int n)
{
        if(m == n) return head;

        ListNode dummy = new ListNode(0);
        dummy.next = head;

        ListNode prev = dummy;
        ListNode cur = head;

        for(int i = 0; i < m - 1; i++){
            prev = prev.next;
            cur = cur.next;
        }

        for(int i = 0; i < n - m; i++){
            ListNode next = cur.next;
            cur.next = next.next;
            next.next = prev.next;
            prev.next = next;
        }


        return dummy.next;
    }
}
```

Remove Nth Node From End of List

涉及链表删除操作的时候，稳妥起见都用 **dummy node**，省去很多麻烦。因为不一定什么时候 **head** 就被删了。

```
public class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;

        ListNode slow = dummy;
        ListNode fast = dummy;
        for(int i = 0; i < n; i++){
            fast = fast.next;
        }

        while(fast.next != null){
            slow = slow.next;
            fast = fast.next;
        }

        slow.next = slow.next.next;
        return dummy.next;
    }
}
```

Remove Linked List Elements

这题完全是一个道理，只不过注意下没准要删除的元素是连续的，那种情况下不要动 head.

```
public class Solution {
    public ListNode removeElements(ListNode head, int val) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        head = dummy;
        while(head.next != null){
            if(head.next.val == val){
                head.next = head.next.next;
            } else {
                head = head.next;
            }
        }

        return dummy.next;
    }
}
```

Delete Node in a Linked List

这题稍微有点怪，因为没有 prev 指针，其实只能把后边的值都给 shift 过来。。。

既然是删除，还是用 dummy node.(不然会有 bug，[0]-->[1] 删 [0] 会返回 [1]-->[1] 而不是 [1])

```
public class Solution {
    public void deleteNode(ListNode node) {
        ListNode dummy = new ListNode(0);
        dummy.next = node;
        while(node.next != null){
            dummy.next.val = node.next.val;
            dummy = dummy.next;
            node = node.next;
        }
        dummy.next = null;
    }
}
```

6/11, LinkedList 杂题

Intersection of Two Linked Lists

最早听到这题还是夏天实习吃饭的时候，现在也不知道那哥们都在忙啥。

```java
public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode
 headB) {
        int lenA = 0;
        int lenB = 0;
        ListNode ptrA = headA;
        ListNode ptrB = headB;
        while(ptrA != null){
            ptrA = ptrA.next;
            lenA ++;
        }
        while(ptrB != null){
            ptrB = ptrB.next;
            lenB ++;
        }

        ptrA = headA;
        ptrB = headB;

        if(lenA > lenB){
            for(int i = 0; i < lenA - lenB; i++){
                ptrA = ptrA.next;
            }
        } else {
            for(int i = 0; i < lenB - lenA; i++){
                ptrB = ptrB.next;
            }
        }

        while(ptrA != ptrB){
            ptrA = ptrA.next;
            ptrB = ptrB.next;
        }

        return ptrA;
    }
}
```

Palindrome Linked List

O(n) 时间，O(1) 空间。好久不写链表了，感觉写的有点粗糙。。

用都指向 **head** 的快慢指针可以判断链表长度奇偶，最后 **fast == null** 的时候为偶，**slow** 指向后半单第一个节点；**fast.next == null** 的时候链表长度为奇数，**slow** 指向中间节点。

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public boolean isPalindrome(ListNode head) {
        if(head == null || head.next == null) return true;

        ListNode slow = head;
        ListNode fast = head;
        while(fast != null && fast.next != null){
            slow = slow.next;
            fast = fast.next.next;
        }
        if(fast == null){
            ListNode headB = reverse(slow);
            return compare(head, headB);
        } else {
            ListNode headB = slow.next;
            slow = null;
            headB = reverse(headB);
            return compare(head, headB);
        }

    }
```

```
    private boolean compare(ListNode headA, ListNode headB){
        while(headA != null && headB != null){
            if(headA.val != headB.val) return false;

            headA = headA.next;
            headB = headB.next;
        }
        return true;
    }

    private ListNode reverse(ListNode head){
        ListNode prev = null;
        while(head != null){
            ListNode next = head.next;
            head.next = prev;
            prev = head;
            head = next;
        }
        return prev;
    }
}
```

Remove Duplicates from Sorted List

能这样 30秒 AC 的良心水题，已经不多了=。=

```
public class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        while(head != null && head.next != null){
            if(head.val == head.next.val){
                head.next = head.next.next;
            } else {
                head = head.next;
            }
        }

        return dummy.next;
    }
}
```

Merge Two Sorted Lists

40 秒能 AC 的良心水题，也不多啊！

```java
public class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(0);
        ListNode head = dummy;

        while(l1 != null && l2 != null){
            if(l1.val < l2.val){
                head.next = l1;
                l1 = l1.next;
            } else {
                head.next = l2;
                l2 = l2.next;
            }
            head = head.next;
        }
        while(l1 != null){
            head.next = l1;
            l1 = l1.next;
            head = head.next;
        }
        while(l2 != null){
            head.next = l2;
            l2 = l2.next;
            head = head.next;
        }

        return dummy.next;
    }
}
```

Merge k Sorted Lists

在 LeetCode 早期年代，这样的题居然可以算成 Hard 难度。。

```
public class Solution {

    private class NodeComparator implements Comparator<ListNode>
{
        public int compare(ListNode one, ListNode two){
            return one.val - two.val;
        }
    }
    public ListNode mergeKLists(ListNode[] lists) {
        if(lists == null || lists.length == 0) return null;

        ListNode dummy = new ListNode(0);
        ListNode head = dummy;

        PriorityQueue<ListNode> heap = new PriorityQueue<ListNod
e>(new NodeComparator());
        for(ListNode node : lists){
            if(node != null) heap.offer(node);
        }

        while(!heap.isEmpty()){
            ListNode node = heap.poll();
            head.next = node;
            head = head.next;

            if(node.next != null) heap.offer(node.next);
        }

        return dummy.next;

    }
}
```

Odd Even Linked List

拼接链表，认准多个 **dummy node.**

```java
public class Solution {
    public ListNode oddEvenList(ListNode head) {
        ListNode oddDummy = new ListNode(0);
        ListNode evenDummy = new ListNode(0);
        ListNode odd = oddDummy;
        ListNode even = evenDummy;

        boolean isOdd = true;
        while(head != null){
            if(!isOdd){
                even.next = head;
                even = even.next;
            } else {
                odd.next = head;
                odd = odd.next;
            }
            head = head.next;
            isOdd = !isOdd;
        }
        odd.next = evenDummy.next;
        even.next = null;

        return oddDummy.next;
    }
}
```

Swap Nodes in Pairs

一开始想复杂了，还挑着拆成两个 list .. 其实就是注意下存中间两个 node，还有这两个 node 的 prev 与 next，循环解决就行。

```java
public class Solution {
    public ListNode swapPairs(ListNode head) {
        if(head == null || head.next == null) return head;
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode prev = dummy;
        while(head != null && head.next != null){
            ListNode cur1 = head;
            ListNode cur2 = head.next;
            ListNode next = head.next.next;

            prev.next = cur2;
            cur2.next = cur1;
            cur1.next = next;

            head = next;
            prev = cur1;
        }

        return dummy.next;
    }
}
```

(FB) 链表的递归与倒序打印

(FB) 在不修改链表结构和数据的情况下，倒序输出链表

他妈的，写完代码之后发现我特意建一个页面总结这题的目的是什么。。

- 递归
- 迭代

```
    private static void reversePrintList_recursive(ListNode head)
{
        if(head.next == null){
            System.out.print(head.val + ",");
            return;
        }
        reversePrintList_recursive(head.next);
        System.out.print(head.val + ",");
    }

    private static void reversePrintList_iterative(ListNode head)
{
        int size = 0;
        ListNode cur = head;
        while(cur != null){
            size++;
            cur = cur.next;
        }

        for(; size > 0; size --){
            ListNode ptr = head;
            for(int i = 0; i < size - 1; i++) ptr = ptr.next;
            System.out.print(ptr.val + ",");
```

```
        }
    }

    public static void main(String[] args) {
        // 5 -> 4 -> 3 -> 2 -> 1 -> 0 -> NULL

        ListNode node5 = new ListNode(5);
        ListNode node4 = new ListNode(4);
        ListNode node3 = new ListNode(3);
        ListNode node2 = new ListNode(2);
        ListNode node1 = new ListNode(1);
        ListNode node0 = new ListNode(0);

        node5.next = node4;
        node4.next = node3;
        node3.next = node2;
        node2.next = node1;
        node1.next = node0;

        reversePrintList_recursive(node5);
        reversePrintList_iterative(node5);
    }
```

LinkedIn 面经

集中在 **LeetCode LinkedIn tag** 的题目，也有围绕着题目思想的各种扩展分类内容。

6/17, LinkedIn 面经题

Sparse Matrix Multiplication

我现在知道这题为什么 AC 率这么高了。。因为直接写个 triple nested loop 也能过。。

然后就是以 1021ms 的傲人速度打败了 3% 的用户 =。= 绝大多数 submission 在 60~70ms 之间。

所以看来这题的主要思路还是“空间换时间”。

```
public class Solution {
    public int[][] multiply(int[][] A, int[][] B) {
        int rowsA = A.length;
        int colsA = A[0].length;
        int rowsB = B.length;
        int colsB = B[0].length;

        int[][] rst = new int[rowsA][colsB];

        for(int i = 0; i < rowsA; i++){
            for(int j = 0; j < colsB; j++){
                for(int k = 0; k < colsA; k++){
                    if(A[i][k] == 0 || B[k][j] == 0) continue;
                    rst[i][j] += A[i][k] * B[k][j];
                }

            }
        }

        return rst;
    }
}
```

经过一番简单优化之后，时间已然降到了 324ms，用 List of Lists 存 A/B 矩阵中不为 0 的 row/col

这题用 HashMap，Key = Integer, Value = List，600ms;

改用 List of Lists，324ms;

最后用 List[] 里面放 List，248ms.

```
public class Solution {
    private class Tuple{
        int index;
        int val;
        public Tuple(int index, int val){
            this.index = index;
            this.val = val;
        }
    }

    public int[][] multiply(int[][] A, int[][] B) {
        int rowsA = A.length;
        int colsA = A[0].length;
        int rowsB = B.length;
        int colsB = B[0].length;

        int[][] rst = new int[rowsA][colsB];

        List[] rowList = new List[rowsA];
        List[] colList = new List[colsB];

        for(int i = 0; i < rowsA; i++){
            List<Tuple> newList = new ArrayList<Tuple>();
            for(int j = 0; j < colsA; j++){
                if(A[i][j] != 0) newList.add(new Tuple(j, A[i][j
]));
            }
            rowList[i] = newList;
        }

        for(int i = 0; i < colsB; i++){
            List<Tuple> newList = new ArrayList<Tuple>();
```

```
            for(int j = 0; j < rowsB; j++){
                if(B[j][i] != 0) newList.add(new Tuple(j, B[j][i
]));
            }
            colList[i] = newList;
        }

        for(int i = 0; i < rowsA; i++){
            for(int j = 0; j < colsB; j++){
                rst[i][j] = multiply(rowList[i], colList[j]);
            }
        }

        return rst;
    }

    private int multiply(List<Tuple> A, List<Tuple> B){
        if(A == null || B == null) return 0;
        if(A.size() == 0 || B.size() == 0) return 0;

        int sum = 0;
        int ptrA = 0;
        int ptrB = 0;
        while(ptrA < A.size() && ptrB < B.size()){
            if(A.get(ptrA).index == B.get(ptrB).index){
                sum += A.get(ptrA++).val * B.get(ptrB++).val;
            } else if(A.get(ptrA).index < B.get(ptrB).index){
                ptrA ++;
            } else {
                ptrB ++;
            }
        }
        return sum;
    }
}
```

优化方法参考论坛这个帖子，里面提到了一个 CMU Lecture ，明天有空看看。

- **70ms** 的解其实很简单；考虑到外面都是 **i,j,k** 的三重循环，所有操作都在最里面执行，可以直接把**index** 的顺序交换，这样可以利用其中某个位置为 **0** 的特点，直接跳过最内圈的循环。

- 交换 **j , k**

```
public class Solution {
    public int[][] multiply(int[][] A, int[][] B) {
        int rowsA = A.length;
        int colsA = A[0].length;
        int rowsB = B.length;
        int colsB = B[0].length;

        int[][] rst = new int[rowsA][colsB];

        for(int i = 0; i < rowsA; i++){
            for(int k = 0; k < colsA; k++){
                if(A[i][k] == 0) continue;

                for(int j = 0; j < colsB; j++){
                    if(B[k][j] == 0) continue;
                    rst[i][j] += A[i][k] * B[k][j];
                }

            }
        }

        return rst;
    }
}
```

- 交换 **i , k**

```java
public class Solution {
    public int[][] multiply(int[][] A, int[][] B) {
        int rowsA = A.length;
        int colsA = A[0].length;
        int rowsB = B.length;
        int colsB = B[0].length;

        int[][] rst = new int[rowsA][colsB];


        for(int k = 0; k < colsA; k++){
            for(int j = 0; j < colsB; j++){
                if(B[k][j] == 0) continue;

                for(int i = 0; i < rowsA; i++){
                    if(A[i][k] == 0) continue;
                    rst[i][j] += A[i][k] * B[k][j];
                }

            }
        }

        return rst;
    }
}
```

Isomorphic Strings

这题考察的是，如何实现一个“双向 one-to-one onto mapping (bijection)”，原 domain 是 String S 的字符集，目标 domain 是 String T 的字符集。不能出现 one-to-many 或者 many-to-one.

写一会儿很快就可以发现，一个 hashmap 是不够的，至少不够快。因为一个 hashmap 只能做一个方向的 mapping，不能高效反方向查找有没有出现 one-to-many / many-to-one 的情况。

```
public class Solution {
    public boolean isIsomorphic(String s, String t) {
        if(s.length() != t.length()) return false;

        HashMap<Character, Character> mapS = new HashMap<Charact
er, Character>();
        HashMap<Character, Character> mapT = new HashMap<Charact
er, Character>();

        for(int i = 0; i < s.length(); i++){
            char charS = s.charAt(i);
            char charT = t.charAt(i);

            if(mapT.containsKey(charT) && mapT.get(charT) != cha
rS) return false;
            if(mapS.containsKey(charS) && mapS.get(charS) != cha
rT) return false;

            mapS.put(charS, charT);
            mapT.put(charT, charS);
        }

        return true;
    }
}
```

既然输入都是字符串，方便起见，可以用 int[256] 代替 hashmap 加速。

```
public class Solution {
    public boolean isIsomorphic(String s, String t) {
        if(s.length() != t.length()) return false;

        int[] mapS = new int[256];
        int[] mapT = new int[256];

        for(int i = 0; i < s.length(); i++){
            char charS = s.charAt(i);
            char charT = t.charAt(i);

            if(mapT[charT] != 0 && mapT[charT] != (int) charS) r
eturn false;
            if(mapS[charS] != 0 && mapS[charS] != (int) charT) r
eturn false;

            mapS[charS] = (int)charT;
            mapT[charT] = (int)charS;
        }

        return true;
    }
}
```

6/28, LinkedIn 面经题

Two Sum III - Data structure design

这是个典型的 data structure design trade off 问题，因为 add() 和 find() 函数总有一个会很慢，问题在于让那个慢，就要取决于具体的 workload. 真问到这种问题一定要先问清楚。

需要注意的 corner case 是，[add(0), find(0)]，这类情况，还有 [add(2), add(2), find(4)] 这类。

Quick-Find:

```
public class TwoSum {

    HashSet<Integer> num = new HashSet<>();
    HashSet<Integer> sum = new HashSet<>();

    // Add the number to an internal data structure.
    public void add(int number) {
        if(num.contains(number)) {
            sum.add(number * 2);
        } else {
            Iterator<Integer> iter = num.iterator();
            while(iter.hasNext()){
                int num = iter.next();
                sum.add(num + number);
            }
            num.add(number);
        }
    }

    // Find if there exists any pair of numbers which sum is equ
al to the value.
    public boolean find(int value) {
        return sum.contains(value);
    }
}
```

事实证明，拿到 **iterator** 手动 **iter.next** 要比直接用 **enhanced for loop** 快。

Quick-Add:

```
public class TwoSum {

    HashMap<Integer, Integer> map = new HashMap<Integer, Integer
>();

    // Add the number to an internal data structure.
    public void add(int number) {
        if(map.containsKey(number)){
            map.put(number, 2);
        } else {
            map.put(number, 1);
        }
    }

    // Find if there exists any pair of numbers which sum is equ
al to the value.
    public boolean find(int value) {
        Iterator<Integer> iter = map.keySet().iterator();

        while(iter.hasNext()){
            int key = iter.next();
            if(map.containsKey(value - key)){
                if(key != value - key || map.get(key) == 2) retu
rn true;
            }
        }
        return false;
    }
}
```

Text Justification

这题不怎么考算法思想和数据结构，就是看你怎么处理各种 test case 以及字符串边边角角的细节问题。。

假如 L = 10 的话，我们需要重点考虑的是这么几种情况；

- **x x x _ _ x x _ x x**

- **x x x x x _ _ _ _ _**
- **x x x _ x _ x _ x _**
- 每一行最左面肯定是填满的，没有空余；
- 每一个单词后面都要空格分开，结尾除外；
- 每一行最右面可能正好填满，也可能有空格；
- 如果正好到了最后一个单词，或者连续两个大单词无法 fit 到同一行，则只有一个单词，后面全是空格；

于是我们的设计思路围绕着这几种情况考虑：

- 维护目前空间占用，如果下一个单词放不下了就开始在 [start, end] 区间内构造单词
- 让每个单词自带一个空格占用
- 为了应对一个单词正好占满整行的情况，起始空间占用为 -1，此后依次增加 word[i].length + 1;

进入填词阶段：

- 每个单词后面的空格数为标准 space + extra，其中 space 默认为 1， extra 默认为 0；
- 如果不是一行只有一个单词的情况，则根据 gap 数计算 space 与 extra 值；
  - space = (剩余空间 / gap 数) + 1;
  - extra = (剩余空间 % gap 数)；
- 启动 StringBuilder，先把第一个单词填上；
- 而后进入循环按照 space 与 extra 数填充；extra 最坏情况下也只会在一个单词后面多加上一个；
- 如果是一个单词一行的情况，则计算剩余空白，都填上空格。
- 设立新起点，新一轮扫描。

```
public class Solution {
    public List<String> fullJustify(String[] words, int maxWidth)
```

```
 {
        List<String> list = new ArrayList<String>();

        int N = words.length;
        int right = 0;
        for(int left = 0; left < N; left = right){
            // Each word comes with one space;
            // Except the first word, so start with -1.
            int len = -1;
            for(right = left; right < N && len + words[right].le
ngth() + 1 <= maxWidth; right ++){
                len += words[right].length() + 1;
            }

            // Those are in case there's only one word picked, o
r in last line
            int space = 1;
            int extra = 0;
            if(right != left + 1 && right != N){
                // right - left - 1 is number of gaps
                space = (maxWidth - len) / (right - left - 1) + 1
;
                extra = (maxWidth - len) % (right - left - 1);
            }
            StringBuilder sb = new StringBuilder(words[left]);
            for(int i = left + 1; i < right; i++){
                for(int j = 0; j < space; j++) sb.append(' ');
                if(extra-- > 0) sb.append(' ');
                sb.append(words[i]);
            }

            int rightSpace = maxWidth - sb.length();
            while(rightSpace-- > 0) sb.append(' ');
            list.add(sb.toString());
        }

        return list;
    }
}
```

7/6, LinkedIn 面经

Find Leaves of Binary Tree

这题本身不难，但是有一个常见错误：

在 **dfs** 中直接 **root = null** 是无法删除 **root** 的.

因为对于 **object**， **java** 是 **pass by value,** 传递过去的是 **object reference.**

在这个帖子中举例：

Figure 5: (Java) Pass-by-value example

```
public void foo(Dog d) {
    d = new Dog("Fifi"); // creating the "Fifi" dog
}



Dog aDog = new Dog("Max"); // creating the "Max" dog
// at this point, aDog points to the "Max" dog
foo(aDog);
// aDog still points to the "Max" dog
```

换句话讲，**dfs** 中作为参数的 **TreeNode root** 是一个新建的 **reference**，默认指向的是原参数 **root** 的同一个位置。因此我们可以对它指向的 **object** 进行各种操作.

但是当我们更改这个 **reference** 指向时，我们只是改了函数自己的那个 **copy** 所指向的 **object**，而对原始参数指向的东西没有任何改变。

于是这题如果试图在 **dfs** 中用 **root = null** 删除节点，实际上我们不会删除任何节点，只会更改在自己这层 **call** 中， **root** 指向的节点而已。因此我们之前可以用其他 **method call** 在 **dfs** 中对各种 **list, set, map** 和 **collection** 里面的值进行修改，但是改 **method call** 里面的 **reference** 并不会对原 **object** 造成任何影响。

```
public class Solution {
    public List<List<Integer>> findLeaves(TreeNode root) {
        List<List<Integer>> rst = new ArrayList<List<Integer>>()
;

        while(root != null) {
            List<Integer> list = new ArrayList<Integer>();
            if(dfs(list, root)) root = null;
            rst.add(list);
        }

        return rst;
    }

    private boolean dfs(List<Integer> list, TreeNode root){
        if(root == null) return false;
        if(root.left == null && root.right == null){
            list.add(root.val);
            return true;
        }

        if(dfs(list, root.left)) root.left = null;
        if(dfs(list, root.right)) root.right = null;

        return false;
    }
}
```

Find the Celebrity

对于两个人 A 与 B, 有如下可能：

knows(A, B)

- **A --> B (A knows B)**
    - **A** 一定不是明星

- **A <-- B (B knows A)**
    - **B** 一定不是明星
- **A ... B (** 互相不认识 **)**
    - **B** 一定不是明星

因为给定条件“所有人都认识明星，而明星不认识任何人”，对于任意一对 A 和 B，我们仅仅做一次 knows 比较就可以排除掉一个人，我们就可以存一个 ptr 依次扫描整个数组。

当最后只剩下一个可能性的时候，我们依然有必要检查一下，因为我们不确定这个 cur 是不是认识数组里的其他人，或者是不是其他人都认识 cur. 因此再一次循环进行判断 + 记录认识 cur 的人数就可以了。

```
public class Solution extends Relation {
    public int findCelebrity(int n) {
        int cur = 0;
        for(int i = 1; i < n; i++){
            if(knows(cur, i)) cur = i;
        }

        int knowCount = 0;
        for(int i = 0; i < n; i++){
            if(i == cur) continue;

            if(knows(cur, i)) return -1;
            else if(knows(i, cur)) knowCount ++;
        }

        return knowCount == n - 1 ? cur: -1;
    }
}
```

Product of Array Except Self

老题了，前缀积数组的应用。

```java
public class Solution {
    public int[] productExceptSelf(int[] nums) {
        int N = nums.length;

        int[] leftCumProd = new int[N + 1];
        int[] rightCumProd = new int[N + 1];

        leftCumProd[0] = 1;
        rightCumProd[N] = 1;

        for(int i = 0; i < N; i++){
            leftCumProd[i + 1] = leftCumProd[i] * nums[i];
        }
        for(int i = N - 1; i >= 0; i--){
            rightCumProd[i] = rightCumProd[i + 1] * nums[i];
        }

        int[] rst = new int[N];
        for(int i = 0; i < N; i++){
            rst[i] = leftCumProd[i] * rightCumProd[i + 1];
        }

        return rst;
    }
}
```

Factor Combinations

Word Search II 之后留下了后遗症，看到这种问题也想用记忆化搜索加快速度。

不过这题的 subproblem 不太一样。 因为当最开始的 n 真的等于质数时，我们返回 empty list; 但是如果这是个 dfs 嵌套的 method call，我们却要把这个质数加到 list 里。 换句话说，同样 size / interval 的 subproblem， 返回的结果却要依情况而定，这是违反了记忆化搜索和动态规划性质的。

简单说就是，这题的 **subproblem** 之间，不具有 **optimal substructure.**

dfs(12) 并不等于 2 + dfs(6) 和 3 + dfs(4) 与 4 + dfs(3) ..

于是我们仔细观察 sample output 中，n = 32 的正解，重新排列顺序之后得到：

n = 32

- **[16,2]**
- **[8, 4]**
- **[8, 2, 2]**
- **[4, 4, 2]**
- **[4, 2, 2, 2]**
- **[2, 2, 2, 2, 2]**

我们可以发现把解按照递减顺序排列之后，这是一个类似 combination 的问题，因为：

- 每一个解是若干个元素的组合，解与解之间大小不等
- 同一组解之间，元素的选取有单调性 **(去重)**

具体实现上，为了保证不盲目把初始 n 加到结果中，第一层 dfs 里传进去的 prev 是 n - 1.

这题的 **dfs** 里 **base case** 还不太一样，我们做的是试图直接把 **n** 加进去作为一个解，同时要注意不能直接 **return** ，否则后面的解就都看不到了。

真正的 **return** ，是在对于 **n** 来讲从大到小连 **i = 2** 都尝试过作为 **factor** 之后自然结束搜索。

```
public class Solution {
    public List<List<Integer>> getFactors(int n) {
        List<List<Integer>> rst = new ArrayList<List<Integer>>()
;

        dfs(rst, new ArrayList<Integer>(), n, n - 1);

        return rst;
    }

    private void dfs(List<List<Integer>> rst,
                     List<Integer> list, int n, int prev){
        if(n <= prev){
            list.add(n);
            rst.add(new ArrayList<Integer>(list));
            list.remove(list.size() - 1);
            //return ;
        }

        for(int i = n / 2; i >= 2; i--){
            if(n % i == 0 && i <= prev){
                list.add(i);
                dfs(rst, list, n / i, i);
                list.remove(list.size() - 1);
            }
        }
    }
}
```

Repeated DNA Sequences

我想了半天怎么用 KMP 做这题复杂度比较低，看到 tag 里有 hashtable 就直接用 hashmap 套用了一下，然后不但 AC 了还在运行速度上超过了 76% 的人。。。这题的用意到底是什么。。。

唯一需要注意的是对于出现超过 2 次的 str 不要重复添加，因此 hashmap 里留一个 count ，根据 count 行事。

```java
public class Solution {
    public List<String> findRepeatedDnaSequences(String s) {
        HashMap<String, Integer> map = new HashMap<>();
        List<String> list = new ArrayList<>();
        int N = s.length();
        for(int i = 0; i <= N - 10; i++){
            String str = s.substring(i, i + 10);
            if(map.containsKey(str)){
                int count = map.get(str);
                if(count == 1) list.add(str);
                map.put(str, count + 1);
            } else {
                map.put(str, 1);
            }
        }

        return list;
    }
}
```

去地里看了一圈，这题更有意思的解法是利用 **DNA** 只有四种字母的性质去做 **encode / decode.**

Submission Details

31 / 31 test cases passed.

Runtime: **30 ms**

Accepted Solutions Runtime Distribution

因为字母只有四种可能，我们可以用 2 bits 表示任意字母；同时因为 sequence 长度为 10，所以我们只需要 20 bits 就可以 Uniquely represent 一个 sequence，即自己实现无 collision 的 hashing. 简便起见，一个 32-bit int 就够了。

encode 时，对于每一个给定的 substring，遍历每个字母，然后根据字母决定其数字，给最后两位赋值之后 << 2.

decode 时，每次把当前数字除以 4 看余数，根据余数决定字母，而后 >> 2.

这样对于每一个 string / int ，其 encode / decode 都是唯一的。

空间占用为 2^20 = 1048576 个 int = 4.194304 MB ，代表可能的 hash 值数量。

```
public class Solution {
    public List<String> findRepeatedDnaSequences(String s) {
        List<String> list = new ArrayList<>();
        int N = s.length();
        int[] hash = new int[1048576];
```

```
        for(int i = 0; i <= N - 10; i++){
            String str = s.substring(i, i + 10);
            int index = encode(str);
            hash[index] ++;
            if(hash[index] == 2) list.add(str);
        }

        return list;
    }

    private int encode(String s){
        int num = 0;
        for(int i = 0; i < s.length(); i++){
            char c = s.charAt(i);
            num = num << 2;
            if(c == 'A') num += 0;
            if(c == 'C') num += 1;
            if(c == 'G') num += 2;
            if(c == 'T') num += 3;
        }
        return num;
    }

    private String decode(int num){
        StringBuilder sb = new StringBuilder();
        for(int i = 0; i < 10; i++){
            if(num % 4 == 0) sb.append('A');
            if(num % 4 == 1) sb.append('C');
            if(num % 4 == 2) sb.append('G');
            if(num % 4 == 3) sb.append('T');

            num = num >> 2;
        }
        return sb.reverse().toString();
    }
}
```

Evaluate Reverse Polish Notation

轻松愉快，一次 AC.

```
public class Solution {
    public int evalRPN(String[] tokens) {
        Stack<Integer> stack = new Stack<>();
        for(String str : tokens){
            if(!isOperator(str)){
                stack.push(Integer.parseInt(str));
            } else {
                int num2 = stack.pop();
                int num1 = stack.pop();

                if(str.equals("+")) stack.push(num1 + num2);
                if(str.equals("-")) stack.push(num1 - num2);
                if(str.equals("*")) stack.push(num1 * num2);
                if(str.equals("/")) stack.push(num1 / num2);
            }
        }

        return stack.peek();
    }

    private boolean isOperator(String str){
        if(str.equals("+") || str.equals("-")) return true;
        if(str.equals("/") || str.equals("*")) return true;
        return false;
    }
}
```

Shortest Word Distance 类

Shortest Word Distance

数组内单词可能有重复，所以需要在 **word1 & word2** 的多次出现位置中，寻找最近的。

用一些数组的小 **trick**，比如 **index = -1** 作为 **“还没找到”** 的初始化条件，而两个指针所保存的，都是 **word1 / word2** 所最近出现的位置，**one-pass** 即可。

```
public class Solution {
    public int shortestDistance(String[] words, String word1, St
ring word2) {
        int minDis = Integer.MAX_VALUE;
        int oneIndex = -1, twoIndex = -1;

        for(int i = 0; i < words.length; i++){
            if(words[i].equals(word1)) oneIndex = i;
            if(words[i].equals(word2)) twoIndex = i;

            if(oneIndex != -1 && twoIndex != -1)
                minDis = Math.min(minDis, Math.abs(oneIndex - tw
oIndex));
        }
        return minDis;
    }
}
```

Shortest Word Distance II

这样的问题设计方式实在提醒自己问清楚问题要求，比如 **wordList** 大小，是否有重复，函数调用次数是否频繁等等。这题在原先基础上增加了 **data structure API** 调用的考量。

思路很简单，调用多次之后每次再去重新扫很不经济，需要数据结构去存储每个 **word** 出现的位置，每次 **API** 调用时直接在两个 **list** 中找最小距离。

比较两个 **list** 的方法类似窗口型双指针的基本思想：虽然双指针所能组成的 **pair** 数量是 **n^2**，但是对于指定位置的 **ptr**，我们可以证明后面的所有 **pair** 组合都是没有意义的，因此可以把指针单向移动。

```java
public class WordDistance {
    HashMap<String, List<Integer>> map;
    public WordDistance(String[] words) {
        map = new HashMap<String, List<Integer>>();

        for(int i = 0; i < words.length; i++){
            String str = words[i];
            if(map.containsKey(str)){
                map.get(str).add(i);
            } else {
                List<Integer> list = new ArrayList<Integer>();
                list.add(i);
                map.put(str, list);
            }
        }
    }

    public int shortest(String word1, String word2) {
        List<Integer> oneList = map.get(word1);
        List<Integer> twoList = map.get(word2);

        int ptr1 = 0, ptr2 = 0;
        int minDis = Integer.MAX_VALUE;

        while(ptr1 < oneList.size() && ptr2 < twoList.size()){
            minDis = Math.min(minDis, Math.abs(oneList.get(ptr1)
 - twoList.get(ptr2)));
            if(oneList.get(ptr1) < twoList.get(ptr2)){
                ptr1 ++;
            } else {
                ptr2 ++;
            }
        }

        return minDis;
    }
}
```

Shortest Word Distance III

改成 word1 有可能等于 word2 之后，这题的重点就变成了各种 corner case 怎么处理，比如

【"a","c","a","a"】, word1 = "a", word2 = "a"

【"a", "a"】, word1 = "a", word2 = "a"

换句话讲，就是看你能否考虑到各种情况正确地 assign index.

```
public class Solution {
    public int shortestWordDistance(String[] words, String word1
, String word2) {
        int ptr1 = -1, ptr2 = -1;
        boolean isSame = word1.equals(word2);
        int minDis = Integer.MAX_VALUE;

        for(int i = 0; i < words.length; i++){
            if(words[i].equals(word1)){
                if(isSame){
                    ptr1 = ptr2;
                    ptr2 = i;
                } else {
                    ptr1 = i;
                }
            } else if(words[i].equals(word2)){
                ptr2 = i;
            }

            if(ptr1 != -1 && ptr2 != -1) minDis = Math.min(minDi
s, Math.abs(ptr1 - ptr2));
        }

        return minDis;
    }
}
```

DFA Parse Integer

Valid Number

这题看一眼就会发现要考虑的各种奇葩 case 实在是太多了。。。对于这类需要 parse 的同时考虑 N 多种不同状态的字符串处理问题，就老老实实用 DFA (Deterministic Finite Automata) 吧，要不会被折磨疯的。

记得先把输入字符 **trim** 一下，免得前后空格不好处理。

状态列表，每一个独立状态用一个 **char** 表示；

- **s :** 还未看到 **+ / -** 号的起始状态
- **S :** 已看到符号的起始状态
- **N :** 读到数字了，还没遇到 **'.'** 或 **'e'**
- **d :** 首次碰到 **'.'**，后面还没有遇到数字
- **D :** 已碰到 **'.'** 并且后面有数字
- **e :** 首次碰到 **'e'**，后面还没有数字
- **E :** 已碰到 **'e'** 并且后面有数字
- **F :** 违法状态，一定为 **false;**

DFA 结构如图~ 所有没画出来的边都是 **'F'**，以双圈结束(小写字母)的也都是 **'F'.**

```
public class Solution {
    public boolean isNumber(String s) {
        s = s.trim();
        int N = s.length();
```

```
        char curState = 's';
        for(int i = 0; i < N; i++){
            curState = transition(curState, s.charAt(i));
            if(curState == 'F') return false;
        }

        return (curState == 'd' || curState == 's' || curState =
= 'e') ? false: true;
    }

    private char transition(char curState, char chr){
        switch(curState){
            case 's':
                if(chr == '-' || chr == '+') return 'S';
                else if(Character.isDigit(chr)) return 'N';
                else if(chr == '.') return 'd';
            case 'S':
                if(Character.isDigit(chr)) return 'N';
                else if(chr == '.') return 'd';
                else return 'F';
            case 'N':
                if(Character.isDigit(chr)) return 'N';
                else if(chr == '.') return 'D';
                else if(chr == 'e') return 'e';
                else return 'F';
            case 'd':
                if(Character.isDigit(chr)) return 'D';
                else return 'F';
            case 'D':
                if(Character.isDigit(chr)) return 'D';
                else if(chr == 'e') return 'e';
                else return 'F';
            case 'e':
                if(chr == '-' || chr == '+') return 'e';
                else if(Character.isDigit(chr)) return 'E';
                else return 'F';
            case 'E':
                if(Character.isDigit(chr)) return 'E';
                else return 'F';
```

```
            case 'F':
                return 'F';
            default:
                return 'F';
        }
    }
}
```

String to Integer (atoi)

一个意思，就是 DFA 结构稍微变了点。

检查 overflow 简便方法：

- **next = cur * 10 + newVal;**
- **if(next / 10 != cur), overflow.**

```
public class Solution {
    public int myAtoi(String str) {
        str = str.trim();

        int num = 0;
        int sign = 1;
        char curState = 's';

        for(int i = 0; i < str.length(); i++){
            curState = transition(curState, str.charAt(i));
            if(curState == 'N') sign = -1;
            if(curState == 'F') return num * sign;

            if(curState == 'S'){
                int val = (int) str.charAt(i) - '0';
                int next = num * 10 + val;
                if(next / 10 != num) return (sign == 1) ? Intege
r.MAX_VALUE : Integer.MIN_VALUE;
                num = next;
            }
```

```
        }
        if(curState == 's' || curState == 'P' || curState == 'N'
) return 0;

        return num * sign;
    }

    private char transition(char curState, char cur){
        switch(curState){
            case 's':
                if(cur == '+') return 'P';
                else if(cur == '-') return 'N';
                else if(Character.isDigit(cur)) return 'S';
                else return 'F';
            case 'S':
                if(Character.isDigit(cur)) return 'S';
                else return 'F';
            case 'N':
                if(Character.isDigit(cur)) return 'S';
                else return 'F';
            case 'P':
                if(Character.isDigit(cur)) return 'S';
                else return 'F';
            default:
                return 'F';
        }
    }
}
```

Reverse Integer

刷题刷到现在我才深切体会到当初 bloomberg 面试题是多么的傻逼。。

```
public class Solution {
    public int reverse(int x) {
        int num = 0;

        while(x != 0){
            int val = x % 10;
            int next = num * 10 + val;
            if(next / 10 != num) return 0;
            num = next;
            x /= 10;
        }

        return num;
    }
}
```

Two Pointers

这组问题的分类上基本延续了九章高级算法班的大纲，加了些听课时候的笔记，和自己之后查资料的思考。

我一直觉得有些 **two pointers** 的问题和用递归处理有相通的地方。用递归处理一个问题的时候，最重要的一点在于正确定义 **“**子问题**”** 的结构。而 **two pointers** 的很多问题，其实我们都是在维护两个 **pointer** 之间那个连续区间，作为我们的子问题。而这两个 **pointer** 之所以可以单调移动，是因为其中一个指针所在位置，和后面区间所有元素组成的 **pair** ，都不是有效解。

3 Sum, 3 Sum Closest / Smaller, 4 Sum

3Sum

这题唯一的问题就是去重，三个指针都要注意一次迭代，同样只加一次，要靠 while 推一下。

- **result.add(Arrays.asList(nums[i], nums[j], nums[k]));**
- **result.add(new int[]{1,2,3});**

```
public class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        List<List<Integer>> list = new  ArrayList<List<Integer>
>();
        if(nums == null || nums.length == 0) return list;
        Arrays.sort(nums);

        for(int i = 0; i < nums.length - 2; i++){
            int left = i + 1;
            int right = nums.length - 1;
            while(left < right){
                int sum = nums[i] + nums[left] + nums[right];
                if(sum == 0){
                    list.add(Arrays.asList(nums[i], nums[left +
+], nums[right --]));

                    while(left < right && nums[left] == nums[le
ft - 1]) left ++;
                    while(left < right && nums[right] == nums[r
ight + 1]) right --;
                } else if(sum < 0){
                    left ++;
                } else{
                    right --;
                }
            }

            while(i < nums.length - 2 && nums[i] == nums[i + 1]
) i++;
        }

        return list;
    }
}
```

3Sum Closest

Trivial problem.

```
public class Solution {
    public int threeSumClosest(int[] nums, int target) {
        if(nums == null || nums.length < 3) return -1;

        int curMin = Math.abs(nums[0] + nums[1] + nums[2] - targ
et);
        int curSum = nums[0] + nums[1] + nums[2];

        Arrays.sort(nums);

        for(int i = 0; i < nums.length - 2; i++){
            int left = i + 1;
            int right = nums.length - 1;
            while(left < right){
                int sum = nums[i] + nums[left] + nums[right];
                if(sum == target){
                    return sum;
                } else if(sum < target){
                    left ++;
                } else if(sum > target){
                    right --;
                }

                if(Math.abs(sum - target) < curMin){
                    curMin = Math.abs(sum - target);
                    curSum = sum;
                }
            }
        }

        return curSum;
    }
}
```

3Sum Smaller

更有意思一点的双指针题~ 姿势水平比前几道高，利用 sum < target 时，i 和 left 不动，介于 left 和 right (包括 right) 之间的所有元素 sum 也一定小于 target 的单调性。

```
public class Solution {
    public int threeSumSmaller(int[] nums, int target) {
        if(nums == null || nums.length < 3) return 0;

        int count = 0;
        Arrays.sort(nums);

        for(int i = 0; i < nums.length - 2; i++){
            int left = i + 1;
            int right = nums.length - 1;
            while(left < right){
                int sum = nums[i] + nums[left] + nums[right];
                if(sum >= target){
                    right --;
                } else {
                    count += (right - left);
                    left ++;
                }
            }
        }

        return count;
    }
}
```

4Sum

稍微 gay 了点，但是优化空间变大了，处理的好的话，可以超过 96.09 % ~

- 每一轮的指针都要注意去重；
- 每一轮进入内循环之前，可以直接看一眼 **index** 与

  - **index** 后面的连续几个数
  - **nums[]** 末尾的最后几个数

- 是不是会比 **target** 小或者大，如果 **index** 与后面连续的几个数加起来都没 **target** 大，那么检查这个 **index** 的必要都没有，可以直接跳过；如果 **index** 与最末尾的几个数相加依然小于 **target**，那么说明我们应该直接去看下一个更大的数当 **index.**

```
public class Solution {
    public List<List<Integer>> fourSum(int[] nums, int target) {
        List<List<Integer>> total = new ArrayList<>();
        int n = nums.length;
        if(n < 4) return total;

        Arrays.sort(nums);

        for(int i = 0; i < n - 3;i++)
        {
            if(i > 0 && nums[i] == nums[i - 1]) continue;
            if(nums[i] + nums[i + 1] + nums[i + 2] + nums[i + 3]
 > target) break;
            if(nums[i] + nums[n - 3] + nums[n - 2] + nums[n - 1]
 < target) continue;

            for(int j = i + 1; j < n - 2; j++)
            {
                if(j > i + 1 && nums[j] == nums[j - 1]) continue
;
                if(nums[i] + nums[j] + nums[j + 1] + nums[j + 2]
 > target) break;
                if(nums[i] + nums[j] + nums[n - 2] + nums[n - 1]
 < target) continue;

                int left = j + 1, right = n - 1;
                while(left < right){
                    int sum = nums[left] + nums[right] + nums[i]
 + nums[j];
```

```
                    if(sum < target) left++;
                    else if(sum > target) right--;
                    else{
                        total.add(Arrays.asList(nums[i],nums[j],
nums[left++],nums[right--]));

                        while(left < right && nums[left] == nums
[left - 1]) left ++;
                        while(left < right && nums[right] == num
s[right + 1]) right --;
                    }
                }
            }
        }
        return total;
    }
}
```

6/13, Two Pointers, 对撞型，灌水/two sum

双指针算法普遍分为三种

- 对撞类
  - **two sum** 类，**partition** 类
- 窗口类
- 并行型
- **Princeton** 公开课的课件，讲 **2-way** 和 **3-way partitioning**
- 掌握 **3-way partition** 和 **k-way partition.**

对撞型指针的本质是，不需要扫描多余的状态。在两边指针的判断之后，可以直接跳过其中一个指针与所有另外一面 **index** 组成的 **pair.**

Two Sum II

这道题如果暴力搜索的话，需要考虑的 pair 个数显然是 O(n^2). 但是用对撞型指针，可以有效跳过/剪枝不合理的 pair 组合。

排序之后的数组具有了单调性，两根指针的移动就代表了“增加”和“减少”。这种单调性保证了一条重要性质：

- 如果 **nums[i] + nums[j] > target**，那么 **nums[i ~ j-1] + nums[j] > target.**
- 因此 **i,j** 指向的不仅仅是元素，也代表了利用单调性，**i,j** 之间所有的 **pair.**

```
public class Solution {
    /**
     * @param nums: an array of integer
     * @param target: an integer
     * @return: an integer
     */
    public int twoSum2(int[] nums, int target) {
        // Write your code here
        if(nums == null || nums.length == 0) return 0;

        Arrays.sort(nums);
        int count = 0;

        int left = 0;
        int right = nums.length - 1;
        while(left < right){
            if(nums[left] + nums[right] <= target){
                left ++;
            } else {
                count += right - left;
                right --;
            }
        }

        return count;
    }
}
```

Triangle Count

总的思路对了，但是一开始的做法掉到了坑里，就是用 i = 1 to n 做最外循环，j 和 k 都在 i 后面。

- 这种思路的问题是，想 **num[i] + num[j] > num[k]** 的话，如果当前条件不满足，**j++** 和 **k--** 都可以是对的。就违反了双指针不同程度上利用的单调性，一次只把一个指针往一个方向推，而且不回头。

- 所以顺着这个问题想，如果我们最外围遍历的是 **k**，而 **i, j** 都处于 **k** 的左面，那么每次就只有一个正确方向，挪动一个指针可以保证正确性了，即让 **num[i] + num[j]** 变大，或者变小。

```java
public class Solution {
    /**
     * @param S: A list of integers
     * @return: An integer
     */
    public int triangleCount(int S[]) {
        // write your code here
        Arrays.sort(S);
        int count = 0;

        for(int k = 0; k < S.length; k++){
            int i = 0;
            int j = k - 1;
            while(i < j){
                if(S[i] + S[j] <= S[k]){
                    i ++;
                } else {
                    count += j - i;
                    j --;
                }
            }
        }

        return count;
    }
}
```

Container With Most Water

这种“灌水”类问题和双指针脱不开关系，都要根据木桶原理维护边界；在矩阵中是外围的一圈高度，在数组中就是左右两边的高度。

- 因为最低木板的高度决定水位，高位的木板有着另一种单调性，即高位木板向里移动的所有位置，形成的 **container** 水量都小于原来位置。

```
public class Solution {
    /**
     * @param heights: an array of integers
     * @return: an integer
     */
    public int maxArea(int[] heights) {
        // write your code here
        if(heights == null || heights.length == 0) return 0;
        int max = 0;
        int left = 0;
        int right = heights.length - 1;

        while(left < right){
            int width = right - left;
            int curArea = Math.min(heights[left], heights[right]
) * width;
            max = Math.max(max, curArea);

            if(heights[left] < heights[right]){
                left ++;
            } else {
                right --;
            }
        }

        return max;
    }
}
```

Trapping Rain Water

木桶原理的算法版，最两边的板子形成一个木桶，由两块板子最低的那块决定水位。每次都从两边最低的方向向里扫描。

- 因为 **i,j** 当前高度未必是当前左右两边最高高度，每次更新的时候要注意维护下。

```
public class Solution {
    public int trap(int[] height) {
        if(height == null || height.length <= 2) return 0;
        int trapped = 0;
        int i = 0;
        int j = height.length - 1;

        int leftMax = height[i];
        int rightMax = height[j];

        while(i < j){
            if(leftMax < rightMax){
                if(i + 1 < height.length && height[i + 1] < left
Max){
                    trapped += leftMax - height[i + 1];
                }
                i++;
                leftMax = Math.max(leftMax, height[i]);
            } else {
                if(j - 1 >= 0 && height[j - 1] < rightMax){
                    trapped += rightMax - height[j - 1];
                }
                j--;
                rightMax = Math.max(rightMax, height[j]);
            }
        }

        return trapped;
    }
}
```

6/13, Two Pointers，对撞型，partition类

Kth Largest Element in an Array

- **partition** 类双指针

这类问题我一向喜欢用 "<=" 作为循环和跳过元素的判断条件，然后用 swap. 和九章模板上的写法不太一样。

- 需要规范一下自己写 **array** 问题的变量名用法
- **left, right** 代表移动指针。
- **start, end** 代表区间起始。

```
public class Solution {
    public int findKthLargest(int[] nums, int k) {
        return partition(nums, k, 0, nums.length - 1);
    }

    private int partition(int[] nums, int k, int start, int end)
{

        int pivot = nums[start];
        int left = start;
        int right = end;

        while(left <= right){
            while(left <= right && nums[left] >= pivot) left++;
            while(left <= right && nums[right] <= pivot) right--
;
            if(left < right) swap(nums, left, right);
        }

        swap(nums, start, right);

        if(k == right + 1) return nums[right];
        if(k > right + 1){
            return partition(nums, k, right + 1, end);
        } else {
            return partition(nums, k, start, right - 1);
        }
    }

    private void swap(int[] nums, int a, int b){
        int temp = nums[a];
        nums[a] = nums[b];
        nums[b] = temp;
    }
}
```

- 这题也可以用 **3-way partitioning** 的思路写，复杂度一样

```java
public class Solution {
    public int findKthLargest(int[] nums, int k) {
        return partition(nums, k, 0, nums.length - 1);
    }

    private int partition(int[] nums, int k, int start, int end)
{

        int pivot = nums[start];
        int left = start;
        int right = end;
        int cur = start;

        while(cur <= right){
            if(nums[cur] > pivot){
                swap(nums, cur ++, left ++);
            } else if (nums[cur] < pivot){
                swap(nums, cur, right--);
            } else {
                cur ++;
            }
        }

        right = cur - 1;

        if(k == right + 1) return nums[right];
        if(k > right + 1){
            return partition(nums, k, right + 1, end);
        } else {
            return partition(nums, k, start, right - 1);
        }
    }

    private void swap(int[] nums, int a, int b){
        int temp = nums[a];
        nums[a] = nums[b];
        nums[b] = temp;
    }
}
```

Kth Smallest Element in a Sorted Matrix

Quick Select 在 2D 上的应用。

注意这题复杂度最优的解法是用 **heap** 做类似多路归并的。不过某种原因目前在 **LC** 上这种解法要比用 **heap** 的快一倍。。

```
public class Solution {
    public int kthSmallest(int[][] matrix, int k) {
        if(matrix == null || matrix.length == 0) return -1;

        int rows = matrix.length;
        int cols = matrix[0].length;

        return kthHelper(matrix, 0, rows * cols - 1, k);
    }

    private int kthHelper(int[][] matrix, int start, int end, int
 k){
        int rows = matrix.length;
        int left = start;
        int right = end;
        int pivot = matrix[start / rows][start % rows];

        while(left <= right){
            while(left <= right && matrix[left / rows][left % ro
ws] <= pivot) left++;
            while(left <= right && matrix[right / rows][right %
rows] >= pivot) right--;

            if(left < right) swap(matrix, left, right);
        }
        swap(matrix, start, right);

        if(right + 1 == k){
            return matrix[right / rows][right % rows];
        } else if(right + 1 < k){
            return kthHelper(matrix, right + 1, end, k);
        } else {
```

```
            return kthHelper(matrix, start, right - 1, k);
        }
    }

    private void swap(int[][] matrix, int a, int b){
        int rows = matrix.length;

        int temp = matrix[a / rows][a % rows];
        matrix[a / rows][a % rows] = matrix[b / rows][b % rows];
        matrix[b / rows][b % rows] = temp;
    }
}
```

Nuts & Bolts Problem

一道去年曾经卡了我两天的题目。到最后才发现是他们 test case 写错了。。。

这题和 quick select 的 partition 思路完全一致，但是细节不同。

- 所有的不等式判断条件都没有 **'='** 号。
- 所有元素 **1-1** 对应，不会有多个元素等于 **pivot.**
- **pivot** 元素来自外部，而不是内部已知位置。

如果 partition array 不带等于号的写，数组里有多个元素等于 pivot 就会挂了。

原因在于，在这题中，每次 partition 用的是外部元素，而且 array 里有且只有一个正确 pivot 元素与其对应，所以最终一定会 left == right 停留在 pivot 上。

这题如果像 partition 一样全带 '=' 去判断，其实也是可以正确把数组 partition 的，但是不同于 quick select 里面固定用 num[start] 做 pivot，这种写法里面我们不知道正确的 pivot 位置在哪。

假如按 left <= right，各种 cmp >= 0 <= 0 挪动

【2，1，|7|，8，3，9，|4|，5，6】，pivot = 5

【2，1，4，|8|，|3|，9，7，5，6】

【2，1，4，|3|，|8|，9，7，5，6】

总的来说；

- 全带等号的比较方式，可以 **partition**，要手动找 **pivot** 位置做收尾 **swap**，适用于自己选好 **pivot** 的情况

- 全不带等号的比较方式，如果 **pivot** 元素有唯一对应，可以 **partition** 并且 **left == right** 停留在 **pivot** 上；否则多个等于 **pivot** 时会死循环，因为不带等号无法让指针跨过 等于 **pivot** 的元素。

```
/**
 * public class NBCompare {
 *     public int cmp(String a, String b);
 * }
 * You can use compare.cmp(a, b) to compare nuts "a" and bolts "
b",
 * if "a" is bigger than "b", it will return 1, else if they are
 equal,
 * it will return 0, else if "a" is smaller than "b", it will re
turn -1.
 * When "a" is not a nut or "b" is not a bolt, it will return 2,
 which is not valid.
*/
public class Solution {
    /**
     * @param nuts: an array of integers
     * @param bolts: an array of integers
     * @param compare: a instance of Comparator
     * @return: nothing
     */
    public void sortNutsAndBolts(String[] nuts, String[] bolts,
```

```
NBComparator compare) {
        // write your code here
        helper(nuts, bolts, compare, 0, nuts.length - 1);
    }

    private void helper(String[] nuts, String[] bolts, NBCompara
tor compare,
                        int start, int end){
        if(start >= end) return;
        int pivot = partitionNuts(nuts, bolts[start], start, end
, compare);
        partitionBolts(bolts, nuts[pivot], start, end, compare);

        helper(nuts, bolts, compare, start, pivot - 1);
        helper(nuts, bolts, compare, pivot + 1, end);
    }

    private int partitionNuts(String[] nuts, String bolt, int st
art, int end,
                                NBComparator compare){
        int left = start;
        int right = end;

        while(left < right){
            while(left < right && compare.cmp(nuts[left], bolt)
< 0) left ++;
            while(left < right && compare.cmp(nuts[right], bolt)
 > 0) right --;
            if(left < right) swap(nuts, left, right);
        }

        // 最终 left = right ，都停留在 pivot 上
        return left;
    }

    private int partitionBolts(String[] bolts, String nut, int s
tart, int end,
                                NBComparator compare){
        int left = start;
        int right = end;
```

```
        while(left < right){
            while(left < right && compare.cmp(nut, bolts[left])
> 0) left ++;
            while(left < right && compare.cmp(nut, bolts[right])
 < 0) right --;
            if(left < right) swap(bolts, left, right);
        }

        return right;
    }

    private void swap(String[] arr, int a, int b){
        String temp = arr[a];
        arr[a] = arr[b];
        arr[b] = temp;
    }
};
```

Sort Colors

非常有名的算法，人称“荷兰旗算法” Dutch national flag problem，经典的 "3-way partitioning"算法，用于解决 quick sort 类排序算法中对于重复元素的健壮性问题，在原有 2-way partitioning 的基础上把所有 val == key 的元素集中于数组中间，实现【（小于），（等于），（大于）】的分区。

- ## Princeton 公开课的课件，讲 2-way 和 3-way partitioning

- 一张图说明 **Dijkstra** 的 **3-way partitioning**，左右指针维护 **< key** 和 **> key** 的元素，**[left , cur - 1]** 为 **= key** 的元素，**[cur, right]** 为未知元素。

- 只有在和 **right** 换元素时，**cur** 指针的位置是不动的，因为下一轮还要看一下换过来的元素是不是 **< key** 要放到左边。

O(n) 时间复杂度

```
public class Solution {
    public void sortColors(int[] nums) {
        int left = 0;
        int right = nums.length - 1;
        int i = 0;

        while(i <= right){
            if(nums[i] < 1){
                swap(nums, left++, i++);
            } else if(nums[i] > 1){
                swap(nums, i, right--);
            } else {
                i ++;
            }
        }
    }

    private void swap(int[] nums, int a, int b){
        int temp = nums[a];
        nums[a] = nums[b];
        nums[b] = temp;
    }
}
```

Sort Colors II

3-way partition 的进阶版本，k-way partition.

- 利用已知最大和最小 **key**，维护双指针对撞
- 数组结构是
- 【最小**key**】【**cur + unknown**】【最大**key**】

这种写法的复杂度分析稍微麻烦一些，最大为 O(n * k/2) 每次复杂度上限为O(n)，最多进行 k / 2 次循环，然而要考虑每次迭代会固定去掉头尾两部分数组，导致 n 逐渐变小，实际的复杂度要根据 k 个数字的分部情况来看，k 的分布越偏向于【1，k】 的两侧，复杂度就越低。

```java
class Solution {
    /**
     * @param colors: A list of integer
     * @param k: An integer
     * @return: nothing
     */
    public void sortColors2(int[] colors, int k) {
        // write your code here

        int left = 0;
        int right = colors.length - 1;
        int cur;
        int lowColor = 1;
        int highColor = k;
        while(lowColor < highColor){
            cur = left;
            while(cur <= right){
                if(colors[cur] == lowColor){
                    swap(colors, cur ++, left ++);
                } else if(colors[cur] == highColor){
                    swap(colors, cur, right --);
                } else {
                    cur ++;
                }
            }

            lowColor ++;
            highColor --;
        }
    }

    private void swap(int[] colors, int a, int b){
        int temp = colors[a];
        colors[a] = colors[b];
        colors[b] = temp;
    }

}
```

Sort Letters by Case

顺着 3-way partitioning 的思路，其实这种写法也很适合解决 2-way partitioning 的问题。

```
public class Solution {
    /**
     *@param chars: The letter array you should sort by Case
     *@return: void
     */
    public void sortLetters(char[] chars) {
        //write your code here
        int left = 0;
        int right = chars.length - 1;
        int cur = 0;

        while(cur <= right){
            if(chars[cur] >= 'a'){
                swap(chars, cur ++, left ++);
            } else if(chars[cur] < 'a'){
                swap(chars, cur, right--);
            } else {
                cur ++;
            }
        }

    }

    private void swap(char[] chars, int a, int b){
        char temp = chars[a];
        chars[a] = chars[b];
        chars[b] = temp;
        return;
    }
}
```

6/14, Two Pointers, Wiggle Sort I & II

- **Wiggle sort** 类问题
  - 如何利用单调性
  - 怎么 **partition**
  - 什么顺序穿插
  - 如何 **in-place**

下面这两题 **googe** 都很喜欢。

Wiggle Sort

首先这题一看就是要你 O(n) 解的，而且要 in-place，不然毫无挑战，因为 quick sort 都 O(n log n)

这题在条件上，比 Wiggle Sort II 要宽松，因此代码变的可以很简单。

- **wiggle sort** 依然利用的数组的单调性，只不过这个单调性每走一步会变一次方向；
- 如果两对相邻元素有同样的单调性，就不符合题意要求。但是在这种情况下，交换后一对元素的位置，就一定可以得到正确结果。

- 这题还有一个**google**面经题变种，在这个帖子中，思想非常类似，更利于思考 **wiggle sort** 的本质

```
public class Solution {
    public void wiggleSort(int[] nums) {
        if(nums == null || nums.length < 2) return;

        boolean findBigger = true;
        for(int i = 1; i < nums.length; i++){
            if(findBigger){
                if(nums[i] < nums[i - 1]){
                    swap(nums, i, i - 1);
                }
            } else {
                if(nums[i] > nums[i - 1]){
                    swap(nums, i, i - 1);
                }
            }
            findBigger = !findBigger;
        }
    }

    private void swap(int[] nums, int a, int b){
        int temp = nums[a];
        nums[a] = nums[b];
        nums[b] = temp;
    }
}
```

Wiggle Sort II

在 Wiggle Sort I 的基础上拿掉了等于号条件，整个题都 gay 了好多，因为这次如果相邻两个数是相等的，怎么 swap 都不符合题意。如果单纯的按顺序扫，就不可避免的会遇到到“找正确元素”的步骤，这个步骤一旦多了，复杂度就上去了。

先写个时间空间都各种不正确的写法，感受下思路。从后往前依次找剩余元素中最大的，按正序插入数组。

有的 test case 是【4,5,5,6】，属于"有重复元素" && "重复元素在第二次遍历插入" && "正好和前一轮结尾相邻" 的情况，所以都正序插入会出 bug.

```
public class Solution {
    public void wiggleSort(int[] nums) {
        if(nums == null || nums.length < 2) return;
        Arrays.sort(nums);
        int[] ans = new int[nums.length];
        int index = nums.length - 1;

        for(int i = 1; i < nums.length; index--){
            ans[i] = nums[index];
            i += 2;
        }

        for(int i = 0; i < nums.length; index--){
            ans[i] = nums[index];
            i += 2;
        }

        for(int i = 0; i < nums.length; i++){
            nums[i] = ans[i];
        }
    }
}
```

关于这题，这个帖子写的非常详尽，里面用到了 **virtual indexing** 的思想

A(i) = nums[(1+2*(i)) % (n|1)]

(n|1) 强行变奇数。

```
index:     0  1  2  3   4   5  6  7  8  9
number:   18 17 19 16  15  11 14 10 13 12
```

I rewire it so that the first spot has index 5, the second spot has index 0, etc, so that I might get this instead:

```
index:     5  0  6  1  7  2  8  3  9  4
number:   11 18 14 17 10 19 13 16 12 15
```

And 11 18 14 17 10 19 13 16 12 15 is perfectly wiggly. And the whole partitioning-to-wiggly-arrangement (everything after finding the median) only takes O(n) time and O(1) space.

If the above description is unclear, maybe this explicit listing helps:

Accessing `A(0)` actually accesses `nums[1]`.
Accessing `A(1)` actually accesses `nums[3]`.
Accessing `A(2)` actually accesses `nums[5]`.
Accessing `A(3)` actually accesses `nums[7]`.
Accessing `A(4)` actually accesses `nums[9]`.
Accessing `A(5)` actually accesses `nums[0]`.
Accessing `A(6)` actually accesses `nums[2]`.
Accessing `A(7)` actually accesses `nums[4]`.
Accessing `A(8)` actually accesses `nums[6]`.
Accessing `A(9)` actually accesses `nums[8]`.

上面那个 sort 之后从大到小正序穿插插入的顺序，和这题一样，不同之处是，这个写法更符合题目的本质：不需要排序，只需要 partitioning. 正确 partition 的数组只要按照这个顺序插入，都是正确的。

- 于是问题就分成了三个子问题：
  - 怎么 **partitioning**
  - 什么顺序穿插
  - 如何 **in-place**

```
    public void wiggleSort(int[] nums) {
        int median = findKthLargest(nums, (nums.length + 1) / 2)
;
        int n = nums.length;

        int left = 0, i = 0, right = n - 1;

        while (i <= right) {

            if (nums[newIndex(i,n)] > median) {
                swap(nums, newIndex(left++,n), newIndex(i++,n));
            }
            else if (nums[newIndex(i,n)] < median) {
                swap(nums, newIndex(right--,n), newIndex(i,n));
            }
            else {
                i++;
            }
        }
    }

    private int newIndex(int index, int n) {
        return (1 + 2*index) % (n | 1);
    }
```

6/14, Two Pointers, 双指针，窗口类

Minimum Size Subarray Sum

这种看起来有点 greedy 味道的双指针都不同程度上利用后面状态的增长性质，直接排除一些元素，减少搜索范围。

在这道题里，如果 [i - j] 的 subarray 已经 >= target 了，考虑任何 j 以后的元素都是没有意义的，因为数组都是正数，依然会 >= target，长度还一定比当前的长。

```
public class Solution {
    public int minSubArrayLen(int s, int[] nums) {
        if(s < 0 || nums == null || nums.length == 0) return 0;
        int size = Integer.MAX_VALUE;
        int sum = 0;
        int j = 0;

        for(int i = 0; i < nums.length; i++){
            while(j < nums.length && sum < s){
                sum += nums[j++];
            }
            if(sum >= s) size = Math.min(j - i, size);
            sum -= nums[i];
        }

        if(size == Integer.MAX_VALUE) return 0;

        return size;
    }
}
```

Maximum Size Subarray Sum Equals k

这题的做法其实和上一题没啥关系，也不属于这个分类。。不过看在长得非常像的份上就一起写了吧 = =

这题其实是 prefix sum array + two sum，利用前缀和数组实现快速区间和查询，同时 two sum 的方法快速地位 index.

这种 prefix sum 的下标要格外小心，很容易标错。。target value 差也是，写之前多手动过几个 case 保平安。

```
public class Solution {
    public int maxSubArrayLen(int[] nums, int k) {
        if(nums == null || nums.length == 0) return 0;
        // Key: prefix sum
        // Value: index
        HashMap<Integer, Integer> map = new HashMap<Integer, Int
eger>();
        map.put(0, 0);
        int sum = 0;
        int size = 0;

        for(int i = 0; i < nums.length; i++){
            sum += nums[i];
            if(map.containsKey(sum - k)){
                size = Math.max(size, i - map.get(sum - k) + 1);
            }
            if(!map.containsKey(sum)) map.put(sum, i + 1);
        }

        return size;
    }
}
```

Longest Substring Without Repeating Characters

这题其实我几周前做过了，放在这个分类里，用分类模板再重构一次吧。

代码上确实简洁了很多，而且一次 AC.

```java
public class Solution {
    public int lengthOfLongestSubstring(String s) {
        if(s.length() <= 1) return s.length();
        int max = 1;
        boolean[] hash = new boolean[256];
        int j = 0;
        for(int i = 0; i < s.length(); i++){
            while(j < s.length()){
                if(!hash[s.charAt(j)]){
                    hash[s.charAt(j++)] = true;
                } else {
                    break;
                }
            }
            max = Math.max(max, j - i);
            hash[s.charAt(i)] = false;
        }

        return max;
    }
}
```

Minimum Window Substring

- 我的第一种写法 **validWindow** 函数扫整个 **target string**，时间爆炸，**1200ms +** ，改成只扫 **256** 个 **ASCII** 字符立刻就变成 **35ms** 了。

```
public class Solution {
    public String minWindow(String s, String t) {
        if(t.length() > s.length()) return "";

        int[] window = new int[256];
        int[] count = new int[256];

        for(int i = 0; i < t.length(); i++){
            count[t.charAt(i)] ++;
        }

        int j = 0;
        int size = Integer.MAX_VALUE;
        String rst = "";
        for(int i = 0; i < s.length(); i++){
            while(j < s.length() && !validWindow(t, window, coun
t)){
                window[s.charAt(j++)] ++;
            }
            if(j - i < size && validWindow(t, window, count)){
                rst = s.substring(i, j);
                size = j - i;
            }
            window[s.charAt(i)]--;
        }

        return rst;
    }

    private boolean validWindow(String t, int[] hash, int[] coun
t){
        for(int i = 0; i < 256; i++){
            if(hash[i] < count[i]) return false;
        }

        return true;
    }
}
```

更好的写法是我改写九章的答案，**9ms**，重点在于 **validWindow()** 函数的优化。

- 对 **Target string** 做预处理，返回 **int[] hash** 和需要 **match** 的字符串数量；
- **j** 作为靠前指针，每次都在 **hash** 里把对应字符 **-1**； 如果对应的 **count** 原本为正数，**match** 字符数量 **curCount++;** 否则对应的位置就会变成负数 **count**，不会被记入 **curCount** 的增减中。
- 于是 **int[] hash** 里面的正负，代表了还需要 **match** 的个数，控制了 **curCount** 的增减； **curCount** 就可以作为判断窗口是否 **valid** 的条件。

```java
public class Solution {
    public String minWindow(String s, String t) {
        if(t == null || t.length() == 0 || t.length() > s.length
())
            return "";
        int targetCount = t.length();
        int curCount = 0;
        int[] hash = new int[256];
        preprocess(hash, t);
        int j = 0;
        String rst = "";
        int minSize = Integer.MAX_VALUE;

        for(int i = 0; i < s.length(); i++){
            while(j < s.length() && curCount < targetCount){
                hash[s.charAt(j)] --;
                if(hash[s.charAt(j)] >= 0) curCount ++;
                j ++;
            }
            if(curCount >= targetCount && j - i < minSize){
                minSize = j - i;
                rst = s.substring(i, j);
            }
            hash[s.charAt(i)] ++;
            if(hash[s.charAt(i)] > 0) curCount --;
        }

        return rst;
    }

    private void preprocess(int[] hash, String t){
        for(int i = 0; i < t.length(); i++){
            hash[t.charAt(i)] ++;
        }
    }
}
```

6/15, Two Pointers, 双指针，窗口类

- 这两题有一个 **trick** 和 **Minimum Window Substring** 非常像，就是维护一个 **"curCount"** 代表目前 **(i,j)** 之间 **match** 上的数量，而通过 **hash[]** 的正负充当计数器的作用。

Longest Substring with At Most Two Distinct Characters

```
public class Solution {
    public int lengthOfLongestSubstringTwoDistinct(String s) {
        int maxSize = 0;
        int j = 0;
        int[] hash = new int[256];
        int distinctCount = 0;
        for(int i = 0; i < s.length(); i++){
            while(j < s.length()){
                if(distinctCount == 2 && hash[s.charAt(j)] == 0)
break;

                if(hash[s.charAt(j)] == 0) distinctCount ++;
                hash[s.charAt(j++)]++;
            }
            if(j - i > maxSize){
                maxSize = j - i;
            }
            hash[s.charAt(i)]--;
            if(hash[s.charAt(i)] == 0) distinctCount --;
        }

        return maxSize;
    }
}
```

Longest Substring with At Most K Distinct Characters

和上一题代码完全没有区别，只是把判断条件里面的 count 数字改一下而已。。

```
public class Solution {
    public int lengthOfLongestSubstringKDistinct(String s, int k)
 {
        int curCount = 0;
        int j = 0;
        int maxSize = 0;
        int[] hash = new int[256];
        for(int i = 0; i < s.length(); i++){
            while(j < s.length()){
                if(curCount == k && hash[s.charAt(j)] == 0) break
;
                if(hash[s.charAt(j)] == 0) curCount ++;
                hash[s.charAt(j++)]++;
            }
            if(j - i > maxSize){
                maxSize = j - i;
            }

            hash[s.charAt(i)]--;
            if(hash[s.charAt(i)] == 0) curCount --;
        }

        return maxSize;
    }
}
```

Heap，排序 array / matrix 选 Kth

Find K Pairs with Smallest Sums

这题一开始搞错了，还以为是 two pointer 问题，其实同一个元素是可以复用的，只要两个 index 不一样就行了。因此总共 pair 的数量是 m * n，光是靠单向移动的 two pointer 是不行的，会漏掉正解。得靠 heap.

Given nums1 = [1,1,2], nums2 = [1,2,3], k = 2

Return: [1,1],[1,1]

The first 2 pairs are returned from the sequence: [1,1],[1,1],[1,2],[2,1],[1,2],[2,2],[1,3],[1,3],[2,3]

最无脑的写法，复杂度 O(k * log(mn))，minHeap 里把所有可能的 pair 都存进去了。

```
public class Solution {
    private class MyComparator implements Comparator<int[]>{
        public int compare(int[] a, int[] b){
            return a[0] + a[1] - b[0] - b[1];
        }
    }
    public List<int[]> kSmallestPairs(int[] nums1, int[] nums2,
int k) {
        List<int[]> list = new ArrayList<>();

        PriorityQueue<int[]> minHeap = new PriorityQueue<int[]>(
new MyComparator());

        for(int i = 0; i < nums1.length; i++){
            for(int j = 0; j < nums2.length; j++){
                minHeap.offer(new int[]{nums1[i], nums2[j]});
            }
        }

        while(!minHeap.isEmpty() && k > 0){
            list.add(minHeap.poll());
            k--;
        }

        return list;
    }
}
```

那么显然的优化是，既然我们只要 **k** 个答案，**heap** 里存 **k** 个就行，同时要尽可能的利用数组已经排好序的特性， 让 **index** 单调向前走。

O(k log k) ~ 超过 81%

> - 自定义 **Tuple**，存 **int[] pair** 还有当前 **pair** 相对于 **nums2** 的 **index**
> - 先以 **nums1** 的前 **k** 个数和 **nums2** 的第一个数为起点，初始化 **minHeap**；
> - 每次新 **tuple** 出来的时候，都把 **index2** 往后移一步，**index1** 不变 (还取 **tuple** 里 **pair[0]** 的值)
> - 在这个过程中，一个 **Pair** 的 **ptr1** 和 **ptr2** 是单调递增的，因为 **heap** 里已经存了之前看过的组合了，不能回头，否则会有重复答案。

这是一个借用 **heap** 的 **two pointer** 算法。

```
public class Solution {
    private class Tuple{
        int val;
        int ptr1;
        int ptr2;
        public Tuple(int val, int ptr1, int ptr2){
            this.val = val;
            this.ptr1 = ptr1;
            this.ptr2 = ptr2;
        }
    }

    private class MyComparator implements Comparator<Tuple>{
        public int compare(Tuple a, Tuple b){
            return a.val - b.val;
        }
    }

    public List<int[]> kSmallestPairs(int[] nums1, int[] nums2,
int k) {
        List<int[]> list = new ArrayList<>();
        if(nums1 == null || nums2 == null ||
           nums1.length == 0 || nums2.length == 0)
```

```
            return list;

        PriorityQueue<Tuple> minHeap = new PriorityQueue<Tuple>(
new MyComparator());

        for(int i = 0; i < nums1.length && i < k; i++){
                minHeap.offer(new Tuple(nums1[i] + nums2[0], i, 0
));
        }

        while(!minHeap.isEmpty() && k > 0){
            Tuple tuple = minHeap.poll();
            int ptr1 = tuple.ptr1;
            int ptr2 = tuple.ptr2;
            list.add(new int[]{nums1[ptr1], nums2[ptr2]});

            if(ptr2 + 1 < nums2.length){
                minHeap.offer(new Tuple(nums1[ptr1] + nums2[ptr2
 + 1], ptr1, ptr2 + 1));
            }
            k--;
        }

        return list;
    }
}
```

Kth Smallest Element in a Sorted Matrix

同一道题，扩展到 k 个 array 的情况~

```
public class Solution {
    private class Tuple implements Comparable<Tuple>{
        int x, y, val;

        public Tuple(int x, int y, int val){
            this.val = val;
            this.x = x;
            this.y = y;
        }
        public int compareTo(Tuple tuple){
            return this.val - tuple.val;
        }
    }

    public int kthSmallest(int[][] matrix, int k) {
        PriorityQueue<Tuple> minHeap = new PriorityQueue<Tuple>(
);
        int rows = matrix.length;
        for(int i = 0; i < rows; i++) minHeap.offer(new Tuple(i,
0, matrix[i][0]));
        for(int i = 0; i < k - 1; i++){
            Tuple tuple = minHeap.poll();
            if(tuple.y + 1 == matrix[0].length) continue;
            minHeap.offer(new Tuple(tuple.x, tuple.y + 1, matrix
[tuple.x][tuple.y + 1]));
        }

        return minHeap.peek().val;
    }


}
```

Bit & Math，位运算与数学

Bit Manipulation，数 '1' bits

- 对付 **unsigned int** 要把判断条件改为 **xx != 0** 而不是 **xx > 0**
- **(n & -n)** 是找 **n** 的 **binary** 里最右面的 **1** 所代表的数
- **n - (n & -n)** 效果为减去 **n** 的二进制表示中最右边为 **1** 的 **bit**
- **n + (n & -n)** 就是直接在最低位的 **1** 上做进位加法。

Number of 1 Bits

这题因为是 un-signed int，最高位 = 1 的情况稍微特殊点，把条件改成 != 0 就行了。

```
public class Solution {
    // you need to treat n as an unsigned value
    public int hammingWeight(int n) {
        int count = 0;
        for(int i = 0; i < 32; i++){
            if((n & (1 << i)) != 0) count ++;
        }
        return count;
    }
}
```

另一种做法是这样，会每次移除最末尾的 siginificant digit，直到 n = 0 为止。

```
    public int hammingWeight(int n) {
        int count = 0;
        while(n != 0){
            count ++;
            n = (n & (n - 1));
        }
        return count;
    }
```

Reverse Bits

注意 shift rst 的顺序，先 shift，再赋值。

```
public class Solution {
    // you need treat n as an unsigned value
    public int reverseBits(int n) {
        int rst = 0;
        for(int i = 0; i < 32; i++){
            rst = rst << 1;
            if(((n & (1 << i)) != 0)) rst += 1;
        }
        return rst;
    }
}
```

Counting Bits

此题重点和难点在于：**Do it like a Boss.**

- 12 :.....0000001100
- -12 :.....11111110100
- 12 & -12:.....0000000100

n - (n & -n) 效果为减去 **n** 的二进制表示中最右边为 **1** 的 **bit**

由于结果肯定比 **n** 小，**bit** 数又一定只比 **n** 多一个，就可以迭代求解了~

Fenwick Tree 万岁~

```
public class Solution {
    public int[] countBits(int num) {
        int[] rst = new int[num + 1];

        for(int i = 1; i <= num; i++){
            // previous index = Remove most significant digit
            // then add one as current bit.
            rst[i] = rst[i - (i & -i)] + 1;
        }

        return rst;
    }
}
```

另一种思路是基于这个公式：

- **f[i] = f[i / 2] + i % 2.**

```
public int[] countBits(int num) {
    int[] f = new int[num + 1];
    for (int i=1; i<=num; i++) f[i] = f[i >> 1] + (i & 1);
    return f;
}
```

6/3, Math & Bit Manipulation, Power of X

Power of Two

妖孽的 trick。。

```
public class Solution {
    public boolean isPowerOfTwo(int n) {
        return (n > 0) && (((n - 1) & n) == 0);
    }
}
```

Power of Three

因为 3 不是 2 的整数倍，用二进制的各种 bit manipulation 是没前途的。。

不用循环解的话，要么 log ，要么 mod.

log 的解法是利用了 log 公式，如果不用 log10 的话会因为精度问题出错。（其实这个解法不管怎么说都需要考虑下 numerial analysis 提到过的精度问题。。）

```
 public static boolean isPowerOfThree(int n) {
    if (n <= 0)
        return false;
    double r = Math.log10(n) / Math.log10(3);
    if (r % 1 == 0)
        return true;
    else
        return false;
}
```

妖孽解法是。。1162261467 = 3^19，是整数中最大的 power of 3，所以如果 n 是 power of 3 的话一定可以整除这个数。

这么干和作弊差不多=。= 对喜欢问这种问题的公司致以深深的鄙视。

```
public class Solution {
    public boolean isPowerOfThree(int n) {
        return (n > 0) && (1162261467 % n == 0);
    }
}
```

Power of Four

一个数如果是 power of four，首先是 power of two.

除此之外，只会有一个 '1' 在固定可能的位置上，所以可以直接写个 mask

01010101010101010101010101010101 = 0x55555555

```
public class Solution {
    public boolean isPowerOfFour(int num) {
        return (num > 0) && ((num & (num - 1)) == 0) && ((num &
0x55555555) > 0);
    }
}
```

坐标系 & 数值计算类

--

Valid Perfect Square

一个最简单的思路是从 left = 1, right = num / 2 开始做 binary search，可以在 log(n) 时间内解决，然而在 num = Integer.MAX_VALUE 的情况下， OJ 觉得速度还是太慢了。

在 numerical analysis 里的多项式求近似值，binary search 又称 bisection method，收敛速度为 liner convergence; 而 newton's method 是 quadratic convergence，收敛速度要快很多。

用 newton's method 起始 x_n 值取的好可以收敛的快一点，不过也要注意 corner case，比如 num = 0, 1 ..

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

求根：$f(x) = x^2 - num = 0$

$f'(x) = 2x.$

$$\therefore x_{n+1} = x_n - \frac{x_n^2 - num}{2x_n}$$

$$= x_n - \frac{1}{2}x_n + \frac{1}{2}\frac{num}{x_n}$$

$$= \frac{1}{2}x_n + \frac{1}{2}\frac{num}{x_n}$$

$$= \frac{1}{2}\left(x_n + \frac{num}{x_n}\right)$$

```
public class Solution {
    public boolean isPerfectSquare(int num) {
        if(num == 1 || num == 0) return true;
        final double eps = 0.1;
        double x_n = num / 2;
        double x_prev = 0;
        while(Math.abs(x_n - x_prev) > eps){
            x_prev = x_n;
            x_n = (x_n + num / x_n) / 2;
        }

        return (int)x_n * (int)x_n == num;
    }
}
```

Sqrt(x)

因为是 **double** ，**epsilon** 的精度要高一些。

```
public class Solution {
    public int mySqrt(int x) {
        final double eps = 0.000001;
        double x_n = x;
        double x_prev = 0;
        while(Math.abs(x_n - x_prev) > eps){
            x_prev = x_n;
            x_n = (x_n + x / x_n) / 2;
        }
        return (int) x_n;
    }
}
```

Pow(x, n)

log(N) 的解法就是简单的 divide & conquer. 有一个特别奇葩的 case，N = Integer.MIN_VALUE... 手动设置一下好了。

```
public class Solution {
    public double myPow(double x, int n) {
        if(n == -1) return 1/x;
        if(n == 0) return 1;
        if(n == 1) return x;
        if(n == -2147483648) return (double) 1 / (x * myPow(x, 2
147483647));

        boolean isNeg = false;
        if(n < 0){
            isNeg = true;
            n = -n;
        }

        double rst;
        if(n % 2 == 0){
            double half = myPow(x, n / 2);
            rst = half * half;
        } else {
            double half = myPow(x, n / 2);
            rst = half * half * x;
        }
        return (isNeg) ? 1/rst: rst;
    }
}
```

Line Reflection

其实这题挺像 Two Sum.

自己写的第一版，追着 bug 改了好多次。。一定有更好的写法。。 但是这种写法速度很快，也不是全无可取之处。 7ms(这种) vs 22ms(HashSet)

Submission Details

37 / 37 test cases passed.

Runtime: **7 ms**

Accepted Solutions Runtime Distribution

不建议真这么写，提一下就行了。

（排序麻烦版）基本步骤：

- 把所有点按照 **x** 坐标升序排序，**x** 相同按 **y** 值升序排序
- 从后面到中点，把 **x** 值相同 **y** 值不同的序列翻转，保证对称
- 初始化中点；
- **Two pointer** 开扫：
    - 如果 **P[left]** 和 **P[right]** 的中点不对，**false;**

  - 如果 **P[left]** 和 **P[right]** 的 **X** 值不一致，但是 **Y** 不同，**false;**
  - **left / right** 向里移动，但是跳过完全一样的点；
  - 左右重合于最后一个元素时，再和 **mid check** 一下**;**
- 若以上条件都满足，返回 **true;**

```
public class Solution {
    private class PointComparator implements Comparator<int[]>{
        public int compare(int[] a, int[] b){
            if(a[0] != b[0]) return a[0] - b[0];
            else return a[1] - b[1];
        }
    }
    public boolean isReflected(int[][] points) {
        if(points == null || points.length <= 1) return true;

        Arrays.sort(points, new PointComparator());
        int N = points.length;

        for(int i = N - 1; i >= N / 2; i--){
            int end = i;
            int start = i;
            while(i >= N / 2 && points[i][0] == points[i - 1][0]
){
                i --;
                start --;
            }
            reverse(points, start, end);
        }


        int left = 0;
        int right = points.length - 1;
        double mid = (double)(points[left][0] + points[right][0]
) / 2;
        while(left < right){
```

```
            double curMid = (double)(points[left][0] + points[ri
ght][0]) / 2;
            if(curMid != mid) return false;

            if(points[left][0] != points[right][0] && points[lef
t][1] != points[right][1]) return false;

            while(left < right && points[left][0] == points[left
 + 1][0] && points[left][1] == points[left + 1][1]) left ++;
            while(left < right && points[right][0] == points[rig
ht - 1][0] && points[right][1] == points[right - 1][1]) right --
;

            left ++;
            right --;
        }

        if(left == right && points[left][0] != mid) return false
;

        return true;
    }

    private void reverse(int[][] points, int start, int end){
        while(start < end){
            int[] temp = points[start];
            points[start] = points[end];
            points[end] = temp;
            start ++;
            end --;
        }
    }
}
```

另一种靠 **HashSet** 的写法就简洁有效多了，扫描左右，记录中点再做检测就是。

记录中点时候最好先推算一下，尽量不要存一个 **/2** 之后的 **double** 类，而可以经过计算保证所有的变量都是 **int** 类。

用 **Set** 很好的处理了“多个点视为同一个点”的问题，唯一的问题是，我知道另外一个点的坐标是什么，但是 **hashset** 不能直接搜 **new int[]{newX, newY}**，得靠 **hashcode.**

最简单的做法就是，自己定义 **String** 做 **hashcode. "x | y"** 代表 **point(x, y)** 就可以了。这种做法值得学习一个~

```
public class Solution {
    public boolean isReflected(int[][] points) {
        HashSet<String> set = new HashSet<String>();
        int minX = Integer.MAX_VALUE;
        int maxX = Integer.MIN_VALUE;

        for(int[] point : points){
            minX = Math.min(minX, point[0]);
            maxX = Math.max(maxX, point[0]);
            set.add(point[0] + "|" + point[1]);
        }

        int targetMid = minX + maxX;

        for(int[] point : points){
            int reflectedX = targetMid - point[0];
            int reflectedY = point[1];
            if(!set.contains(reflectedX + "|" + reflectedY)) ret
urn false;
        }

        return true;
    }
}
```

Max Points on a Line

挺高频的一道题，有点 tricky，我自己写的时候改了好多次都没能处理好多个重复点的情况。。因为和上一题不一样，这题多个点要单独算了。好消息是，这类 Math 题见过了就是见过了，也不用太纠结因为参考了别人解法没掌握题型知识的

问题。

注意要点和流程：

- 对于每一个新的点，要单独建 **HashMap** 用于检查斜率。对于每一个点都要新建，是因为检查多点共线要以每个点自己为准，之前点的 **HashMap** 对新的点没有什么卵用。

- 以 **point[i]** 为外循环，内循环里有可能会出现 **point[j]** 与 **point[i]** 一模一样的问题。要存一个变量记录下到底有多少个和 **point[i]** 一样的点，默认为 **1**， 清算完毕后加上去。

- 如果两个点 **x** 值相等，定义斜率为 **Double.MAX_VALUE;**

- 对于 **point[i]** 清算完毕后，要把最大的 **localMax (**和 **i** 同一直线的点 **maxCount)** 加上与 **point[i]** 重复的点的个数，因为这里面每一个点都和 **localMax** 中的点共线。

O(n^2)，速度还不错

Submission Details

27 / 27 test cases passed.

Runtime: **21 ms**

Accepted Solutions Runtime Distribution

```
public class Solution {
    public int maxPoints(Point[] points) {
        if(points == null) return 0;
        if(points.length <= 2) return points.length;

        int max = 0;
        for(int i = 0; i < points.length; i++){
            Point p1 = points[i];
            int samePoint = 1;
            int localMax = 0;
            HashMap<Double, Integer> map = new HashMap<Double, I
nteger>();
```

```
            for(int j = i - 1; j >= 0; j--){
                Point p2 = points[j];
                double slope = 0.0;

                if(p1.y == p2.y && p1.x == p2.x) {
                    samePoint++;
                    continue;
                } else if(p1.x == p2.x) {
                    slope = Double.MAX_VALUE;
                } else {
                    slope = (double)(p1.y - p2.y) / (p1.x - p2.x
);
                }

                if(map.containsKey(slope)){
                    int count = map.get(slope);
                    map.put(slope, count + 1);
                    localMax = Math.max(localMax, count + 1);
                } else {
                    map.put(slope, 1);
                    localMax = Math.max(localMax, 1);
                }
            }

            max = Math.max(max, localMax + samePoint);
        }

        return max;
    }
}
```

Add Digits

循环求解非常容易，重点在总结其数学规律上；

Digital root

用 int 做字符串 signature

Maximum Product of Word Lengths

这题本身不难，就是循环枚举。

关键点是“已知字母为 **a-z**，如何高效检查两个**String**是否有重复字符”

简单粗暴的方法就是 int[26] 的字母 hash，两个字符串之间用 26 次比较操作检查一下。

但是考虑到只有 **26** 种可能的时候，我们可以直接用 **32-bit int** 的 **bit** 来表示同样的信息，而且 **int** 之间可以直接通过 **OR** 和 **AND** 的位运算操作，大大减少每次比较所需要的时间。

注意点**1**：重复字符不要重复设，所以无脑加到 **sig** 上是不行的，要用 **'OR'**

```
public class Solution {
    public int maxProduct(String[] words) {
        if(words == null || words.length == 0) return 0;

        int[] signatures = new int[words.length];

        for(int i = 0; i < words.length; i++){
            String word = words[i];
            int sig = 0;
            for(int j = 0; j < word.length(); j++){
                int offset = word.charAt(j) - 'a';
                sig |= (1 << offset);
            }
            signatures[i] = sig;
        }

        int max = 0;

        for(int i = 0; i < words.length; i++){
            for(int j = i - 1; j >= 0; j--){
                if((signatures[i] & signatures[j]) == 0){
                    max = Math.max(max, words[i].length() * word
s[j].length());
                }
            }
        }

        return max;
    }
}
```

Interval 与 扫描线

这组问题的分类上基本延续了九章高级算法班的大纲，加了些听课时候的笔记，和自己之后查资料的思考。

Range Addition

Range Addition

数组长度为 n ，更新次数为 k 的话，比较 trivial 的做法就是 O(nk) 的暴力解。

然而最优解是 O(n + k)，利用的算法逻辑类似 meetings rooms 的扫描线算法。

仔细思考的时候用的是类似构造法： 先想只有一个 update 操作的时候，然后逐个叠加，寻找新的共同性质。

不难看出我们可以用一个 "carry" 来表示我们目前 interval 的覆盖结果；比如 +4 和 -2 覆盖的地方，净增长一定是 +2，就像扫描线的时候带着当前的 interval / building 一样。同时我们需要定义一个 “起始” 和 “结束” ，代表着当前区间的覆盖结果，对 carry 值进行修正。

因此，只要在起始的位置 +val ，在终止的后一个位置 -val，两次操作就可以保证整个范围的正确更新。

```
public class Solution {
    public int[] getModifiedArray(int length, int[][] updates) {
        if(length <= 0) return new int[0];
        int[] rst = new int[length];
        for(int[] update : updates){
            int start = update[0];
            int end = update[1];
            int val = update[2];

            rst[start] += val;
            if(end + 1 < length) rst[end + 1] -= val;
        }

        int carry = 0;
        for(int i = 0; i < length; i++){
            rst[i] += carry;
            carry = rst[i];
        }

        return rst;
    }
}
```

Longest Consecutive Sequence

把 LCS 这题放这题放这边是因为两题的思路上有异曲同工之妙。

- 一维 **interval** 什么的，只关心 **start** 和 **end** 就好了。

就这意思。

```
    public int longestConsecutive(int[] nums) {
        if(nums == null || nums.length == 0) return 0;

        HashMap<Integer, Integer> map = new HashMap<>();
        int maxLen = 0;
        for(int i = 0; i < nums.length; i++){
            int num = nums[i];
            if(map.containsKey(num)) continue;

            int leftBound = map.containsKey(num - 1) ? map.get(n
um - 1): 0;
            int rightBound = map.containsKey(num + 1) ? map.get(
num + 1): 0;
            int sum = leftBound + rightBound + 1;

            map.put(num, sum);
            maxLen = Math.max(sum, maxLen);

            if(leftBound != 0)  map.put(num - leftBound, sum);
            if(rightBound != 0) map.put(num + rightBound, sum);
        }
        return maxLen;
    }
```

7/5, Interval 类，扫描线

Interval 类问题中最常用的技巧，就是自定义 **IntervalComparator**，把输入按照 **startTime** 升序排序。

对于任意两个区间**A**与**B**，如果

- **A.end > B.start** 并且
- **A.start < B.end**
- 则 **A** 与 **B** 重叠。

按 **start** 排序后，数组有了单调性，上面的判断条件就简化成了 **A.end > B.start** 则一定重叠**.**

排序后的 **Interval** 扫描过程中，为了保证正确性，要格外小心多个 **Interval** 在同一 **x** 时，处理的先后顺序，比如 **skyline problem.**

熟悉下 **TreeMap** 的常用 **API.**

> - **get()**
> - **put()**
> - **containsKey()**
> - **size()**
> - **values() : 返回 Collection<> of V**
> - **lowerKey(K key) ： 返回最大的小于参数的Key，没有则 null**
> - **higherKey(K key)：返回最小的大于参数的Key, 没有则 null**
> - **firstKey() : 返回最小 key**
> - **lastKey() : 返回最大 key**

Meeting Rooms

Trivial Problem.

```
public class Solution {
    private class IntervalComparator implements Comparator<Inter
val>{
        public int compare(Interval a, Interval b){
            return a.start - b.start;
        }
    }

    public boolean canAttendMeetings(Interval[] intervals) {
        if(intervals == null || intervals.length <= 1) return tr
ue;

        Arrays.sort(intervals, new IntervalComparator());

        for(int i = 1; i < intervals.length; i++){
            if(intervals[i].start < intervals[i - 1].end) return
false;

        }

        return true;
    }
}
```

Meeting Rooms II

fb 面经高频题，follow up 是返回那个 overlap 最多的区间。

另一个 **fb onsite** 面经题里，首先给 **n** 个一维 **interval**，返回任意重合最多的点；然后给 **n** 个二维 **rectangle**，返回任意重复最多的点。

扫描线算法。需要注意的是如果有两个 **interval** 首尾相接，要把结束的那个排在 **array** 的前面，先把房间腾出来；否则的话会认为收尾相接的两个 **meeting** 需要占 **2** 个房间，这是错误的。

```
public class Solution {
```

```java
    private class Point{
        int time;
        boolean isStart;
        public Point(int time, boolean isStart){
            this.time = time;
            this.isStart = isStart;
        }
    }

    private class PointComparator implements Comparator<Point>{
        public int compare(Point a, Point b){
            if(a.time == b.time) return a.isStart ? 1 : -1 ;
            else return a.time - b.time;
        }
    }

    public int minMeetingRooms(Interval[] intervals) {
        if(intervals == null || intervals.length == 0) return 0;

        Point[] arr = new Point[intervals.length * 2];

        for(int i = 0; i < intervals.length; i++){
            arr[i * 2] = new Point(intervals[i].start, true);
            arr[i * 2 + 1] = new Point(intervals[i].end, false);
        }

        Arrays.sort(arr, new PointComparator());

        int maxOverLap = 0;
        int curCount = 0;
        for(Point pt : arr){
            if(pt.isStart) curCount ++;
            else curCount --;
            maxOverLap = Math.max(maxOverLap, curCount);
        }

        return maxOverLap;
    }
}
```

(FB) follow-up: 如何返回重叠最多的那个区间？

- 当前 **maxOverlap** 创造新高的时候，存下 **start** 的时间戳，因为这是所有重合区间中 **start** 时间最晚的一个；
- 继续扫描，看到新的 **end** 的时候，存下这个 **end** 的时间戳，因为它是重合区间里 **end** 时间最早的一个；
- 二者之间，就是具体的 **max overlap** 区间。

针对这题还有另外一种写法，用 heap / PriorityQueue，相比之下代码量要小一点，不过不像扫描线算法那样清晰易于理解，而且容易进行修改适应各种 follow-up 情况。

```
public class Solution {
    private class IntervalComparator implements Comparator<Inter
val>{
        public int compare(Interval a, Interval b){
            return a.start - b.start;
        }
    }

    public int minMeetingRooms(Interval[] intervals) {
        if(intervals == null || intervals.length == 0) return 0;

        Arrays.sort(intervals, new IntervalComparator());

        PriorityQueue<Integer> heap = new PriorityQueue<Integer>
();
        heap.offer(intervals[0].end);
        int maxOverLap = 1;
        for(int i = 1; i < intervals.length; i++){
            if(intervals[i].start < heap.peek()){
                maxOverLap ++;
            } else {
                heap.poll();
            }
            heap.offer(intervals[i].end);
        }

        return maxOverLap;
    }
}
```

Merge Intervals

这题需要注意的 **corner case** 有两个：

- 按 **startTime** 排序后， **merge** 的新 **interval** 可能是完全包含于前一个 **interval** 的，如 【**1,4**】，【**2,3**】. 所以当发现 **overlap** 时，新的 **End** 要以两个 **end** 的最大值为准。

- 记得循环尾部做一个收尾，把最后 **merge** 完的结果生成新的 **interval** 也加到 **list** 中。

```
public class Solution {
    private class IntervalComparator implements Comparator<Inter
val>{
        public int compare(Interval a, Interval b){
            return a.start - b.start;
        }
    }

    public List<Interval> merge(List<Interval> intervals) {
        List<Interval> list = new ArrayList<Interval>();
        if(intervals == null || intervals.size() == 0) return li
st;

        Collections.sort(intervals, new IntervalComparator());

        int curStart = intervals.get(0).start;
        int curEnd = intervals.get(0).end;
        for(int i = 1; i < intervals.size(); i++){
            Interval cur = intervals.get(i);
            if(cur.start > curEnd){
                list.add(new Interval(curStart, curEnd));
                curStart = cur.start;
                curEnd = cur.end;
            } else {
                curEnd = Math.max(curEnd, cur.end);
            }
        }

        list.add(new Interval(curStart, curEnd));

        return list;
    }
}
```

Insert Interval

顺着上一题的思路，一个最简单直接的写法就是。。直接把新的 interval 插进去，然后再跑一遍 merge interval 的代码。

考虑到一些极端情况，比如 newInterval 合并了原始 list 中绝大多数 intervals, 在原 list 上进行操作的话需要的比较与删减也会很多，所以从算法复杂度上讲，这种写法至少可以 AC，但不是最优。

时间复杂度 O(n log n) + O(n)

```java
public class Solution {
    private class IntervalComparator implements Comparator<Inter
val>{
        public int compare(Interval a, Interval b){
            return a.start - b.start;
        }
    }

    public List<Interval> insert(List<Interval> intervals, Inter
val newInterval) {
        intervals.add(newInterval);

        Collections.sort(intervals, new IntervalComparator());

        List<Interval> list = new ArrayList<Interval>();

        int curStart = intervals.get(0).start;
        int curEnd = intervals.get(0).end;

        for(int i = 1; i < intervals.size(); i++){
            Interval cur = intervals.get(i);
            if(cur.start > curEnd){
                list.add(new Interval(curStart, curEnd));
                curStart = cur.start;
                curEnd = cur.end;
            } else {
                curEnd = Math.max(curEnd, cur.end);
            }
        }
        list.add(new Interval(curStart, curEnd));

        return list;
    }
}
```

如果要进行进一步优化，要利用原题中 **set of "non-overlapping" intervals** 的条件，所以 **newInterval** 有可能会与任何 **interval** 或 **N** 个 **interval** 的集合，在左右两边 **merge [0,**

N] 个 interval, 从完全 disjoint 到完全合并。

时间复杂度 O(n)，one-pass.

```
    public List<Interval> insert(List<Interval> intervals, Inter
val newInterval) {
        List<Interval> list = new ArrayList<>();

        int ptr = 0;
        int n = intervals.size();

        while(ptr < n && intervals.get(ptr).end < newInterval.st
art) list.add(intervals.get(ptr++));
        while(ptr < n && intervals.get(ptr).start <= newInterval
.end){
            newInterval.start = Math.min(newInterval.start, inte
rvals.get(ptr).start);
            newInterval.end = Math.max(newInterval.end, interval
s.get(ptr).end);
            ptr ++;
        }
        list.add(newInterval);
        while(ptr < n) list.add(intervals.get(ptr++));

        return list;
    }
```

在此基础上还可以进一步优化，不用额外空间，**on-the-fly** 解决战斗。

这种做法可以不花费任何额外空间，但是时间复杂度会更高，因为 **List.remove()** 是一个 **O(n)** 操作，**add(index, val)** 也是 **O(n)** 的。

```
    public List<Interval> insert(List<Interval> intervals, Inter
val newInterval) {
        int ptr = 0;

        while(ptr < intervals.size() && intervals.get(ptr).end <
 newInterval.start) ptr++;
        while(ptr < intervals.size() && intervals.get(ptr).start
 <= newInterval.end){
            newInterval.start = Math.min(newInterval.start, inte
rvals.get(ptr).start);
            newInterval.end = Math.max(newInterval.end, interval
s.get(ptr).end);
            intervals.remove(ptr);
        }
        intervals.add(ptr, newInterval);

        return intervals;
    }
```

Summary Ranges

一个比较简单的区间融合问题。

```
public class Solution {
    public List<String> summaryRanges(int[] nums) {
        List<String> list = new ArrayList<>();
        if(nums == null || nums.length == 0) return list;

        Integer start = null;
        Integer end = null;

        for(int i = 0; i < nums.length; i++){
            if(start == null){
                start = nums[i];
                end = nums[i];
            } else if(nums[i] == end + 1){
                end ++;
            } else {
                if(!start.equals(end))list.add("" + start + "->"
 + end);
                else list.add("" + start);

                start = nums[i];
                end = nums[i];
            }
        }
        if(!start.equals(end))list.add("" + start + "->" + end);
        else list.add("" + start);

        return list;
    }
}
```

Missing Ranges

第一种不太经济的写法是，建一个 boolean[upper - lower + 1] ，扫一遍数组记录下每个数是否出现过，然后根据 boolean[] 的 flag 进行融合。 这种写法首先需要 O(upper - lower) 的 extra space，而且最重要的是，没有利用到原数组 int[] 已经是排序了的性质，因此速度上不给力，而且在 upper = 100000000, lower = -10000000 这种极端情况下，空间占用非常大。

改进后的代码实现过程中，最开始没有考虑全 test case，虽然AC 了但是代码不够 clean. 下面参考论坛里的解法就简明了很多。

如果实现代码的时候发现要写很多 **if else** 处理特殊情况，最好的选择还是暂停一下，多把 **test case** 思考全再动手。

- 【**lower, nums[i] - 1**】 范围内没有数, **ADD;**
- 【**nums[last] + 1**，**upper**】 范围内没有数, **ADD;**
- 动态更新 **lower**，维护当前有效 **range.**

时间复杂度 **O(n).**

```
public class Solution {
    public List<String> findMissingRanges(int[] nums, int lower,
int upper) {
        List<String> list = new ArrayList<String>();
        for(int n : nums){
            int justBelow = n - 1;
            if(lower == justBelow) list.add(lower+"");
            else if(lower < justBelow) list.add(lower + "->" + j
ustBelow);
            lower = n+1;
        }
        if(lower == upper) list.add(lower+"");
        else if(lower < upper) list.add(lower + "->" + upper);
        return list;
    }
}
```

Data Stream as Disjoint Intervals

作为一个 data structure design 题，首先要问好到底哪个 API call 比较频繁，设计上倾向于优化哪个操作。

先写了个略粗糙的版本，维护当前所有 interval 的 list，每次插入操作的时候靠 binary search 找每个 val 的位置，然后分情况考虑。

addNum(int val) 时间复杂度 **O(log n) + O(n)**

getInterval() 时间复杂度 **O(1)**

空间复杂度 **O(n)**

n = 当前有效数字个数

在测试数据集上，这个方法和 **TreeMap** 的做法用时不相上下。

```
public class SummaryRanges {
    List<Interval> list;
    /** Initialize your data structure here. */
    public SummaryRanges() {
        list = new ArrayList<Interval>();
    }

    public void addNum(int val) {
        if(list.size() == 0){
            list.add(new Interval(val, val));
        } else {
            if(list.size() == 1){
                if(val <= list.get(0).end && val >= list.get(0).
start){
                    return;
                }
            }

            if(val < list.get(0).start){
                if(val == list.get(0).start - 1){
                    list.get(0).start = val;
                } else {
                    list.add(0, new Interval(val, val));
                }
                return ;
            } else if (val > list.get(list.size() - 1).end){
```

```
                if(val == list.get(list.size() - 1).end + 1){
                    list.get(list.size() - 1).end = val;
                } else {
                    list.add(new Interval(val, val));
                }
                return ;
            }

            int pos = binarySearch(list, val);

            int prevEnd = list.get(pos).end;
            int nextStart = list.get(pos + 1).start;

            if(val <= prevEnd || val >= nextStart){
              return;
            } else if(val == prevEnd + 1 && val == nextStart - 1
){
                list.get(pos).end = list.get(pos + 1).end;
                list.remove(pos + 1);
            } else if(prevEnd + 1 >= val){
                list.get(pos).end = Math.max(list.get(pos).end,
val);
            } else if(val == nextStart - 1){
                int newStart = list.get(pos + 1).start - 1;
                int newEnd = list.get(pos + 1).end;
                // Will have MLE if don't remove
                list.remove(pos + 1);
                list.add(pos + 1, new Interval(newStart, newEnd)
);
            } else {
                list.add(pos + 1, new Interval(val, val));
            }
        }
    }

    public List<Interval> getIntervals() {
        return list;
    }

    private int binarySearch(List<Interval> list, int val){
```

```
        int left = 0;
        int right = list.size() - 1;
        while(left + 1 < right){
            int mid = left + (right - left) / 2;
            if(list.get(mid).start == val){
                return mid;
            } else if(list.get(mid).start < val){
                left = mid;
            } else {
                right = mid;
            }
        }

        return left;
    }
}
```

第二个版本，自己写了个 LinkedList 维护目前所有数字，每次 call getIntervals() 的时候根据 list 当场生成。

addNum(int val) 时间复杂度 O(n);

getIntervals() 时间复杂度 O(n);

空间：O(n);

n = 当前的有效数字个数。

```
public class SummaryRanges {
    private class Node{
        Integer val;
        Node next;
        public Node(Integer val){
            this.val = val;
            this.next = null;
        }
    }
```

```
    Node head;

    /** Initialize your data structure here. */
    public SummaryRanges() {
        Node dummy = new Node(null);
        head = dummy;
    }

    public void addNum(int val) {
        Node cur = head.next;
        Node prev = head;

        while(cur != null){
            if(cur.val == val) return;
            if(cur.val > val){
                prev.next = new Node(val);
                prev.next.next = cur;
                return;
            }
            prev = cur;
            cur = cur.next;
        }

        prev.next = new Node(val);
    }

    public List<Interval> getIntervals() {
        List<Interval> list = new ArrayList<Interval>();

        Node cur = head.next;
        Integer curStart = null;
        Integer curEnd = null;

        while(cur != null){
            int val = cur.val;
            if(curStart == null && curEnd == null){
                curStart = val;
                curEnd = val;
            } else {
                if(val == curEnd + 1){
```

```
                    curEnd = val;
                } else {
                    list.add(new Interval(curStart, curEnd));
                    curStart = val;
                    curEnd = val;
                }
            }
            cur = cur.next;
        }
        list.add(new Interval(curStart, curEnd));

        return list;
    }

}
```

参考了论坛代码之后，另一种巧妙的做法：

addNum(int val) 时间复杂度：**O(log n)**

getIntervals() 时间复杂度：**o(n log n)**

空间复杂度： **O(n)**

```java
public class SummaryRanges {
    TreeMap<Integer, Interval> tree;

    public SummaryRanges() {
        tree = new TreeMap<>();
    }

    public void addNum(int val) {
        if(tree.containsKey(val)) return;

        Integer l = tree.lowerKey(val);
        Integer h = tree.higherKey(val);

        if(l != null && h != null && tree.get(l).end + 1 == val
&& h == val + 1) {
            // Joining two intervals
            tree.get(l).end = tree.get(h).end;
            tree.remove(h);
        } else if(l != null && tree.get(l).end + 1 >= val) {
            // Completely contained in left interval
            // Try to expand end
            tree.get(l).end = Math.max(tree.get(l).end, val);
        } else if(h != null && h == val + 1) {
            // Right next to right interval
            // Merge
            tree.put(val, new Interval(val, tree.get(h).end));
            tree.remove(h);
        } else {
            // Disjoint new interval
            tree.put(val, new Interval(val, val));
        }
    }

    public List<Interval> getIntervals() {
        return new ArrayList<>(tree.values());
    }
}
```

(FB) 给定的是 stream of intervals，求 cover range.

这么改了之后难了很多。。用 TreeMap 依然可以，但是假设目前 TreeMap 维护的都是 none-overlapping intervals，一个新 interval 加入之后可能会出现若干种情况：

- 完全被某个 **interval** 包含，无影响；
- 完全和其他 **interval disjoint**，插入即可；
- 和左边的 **overlap**，但是不和右边的 **overlap;**
- 和右边的 **overlap**，但是不和左边的 **overlap;**
- 一次跨了多个 **interval**，这里面就有全包括的，包括一半的，在左边搭边的，在右边搭边的。。。

于是我觉得这题用 **TreeMap** 比较和乎人性的做法是，**TreeMap** 只负责维护 **sorted intervals**，需要查询 **coverage** 的时候，直接 **new ArrayList<>(treeMap.values())** 给导出来然后跑一遍 **merge interval ..**

或者，直接维护一个按照 **start time** 排好序的 **disjoint list of intervals**，然后每次新的 **interval** 过来的时候，就 **in-place** 扫一遍做 **merge** 好了~ 这样每次 **insert O(N)**，**getCoverage O(1)**

The Skyline Problem

扫描线算法，因为 building outline 只可能发生在每一段 "start/end" 的边界上，因此以每个 edge 的 (x, height) 排序，而后扫描。

- 当 **x** 值不同时，**x** 小的排在前面；
- **else** 当 **y** 值不同时，**y** 小的排在前面；
    - 调换的话下面的代码会有 **bug**

这题在九章课上讲的时候，使用的是 HashHeap，我们需要一个 maxHeap 来检测对于每一个点，我们建筑的最大高度，但同时在扫描过程中我们需要对每一个具体的 edge 进行删除操作，而 HashHeap 可以做到 O(log n) 的复杂度，Java 默认的 heap 只能做到 O(n).

如何在 heap 中快速删除这个问题和当初准备 Bloomberg 时候的马拉松问题一样，Java 中有现成的解决方案：

TreeMap.

接下来的问题是：

- 当遇到一座楼的右边界时，如何删除其对应的左边界？在 TreeMap 里直接用 Edge class 做 key 是不行的，因为我们无法通过 new 一个参数一模一样的 instance 去做 lookup. 所以在 Key-Value 的设计中，我们应该用 height 做 Key，因为同一座建筑左右墙 height 相等，就可以实现查找。即使有多座 height 一样的左墙，我们也只需要用 count 做 value，把当前 height 的墙数减一即可。

- 对于同一高度的不同建筑，如何确保其在 TreeMap 中不会互相 replace ，从而保证算法的正确性? 在自己 debug 的过程中，即使后面逻辑完全一致，不用负数 height 做 Key 依然会有 bug. ，用负数可以省去 Edge class 里面一个变量，最主要的是，在调用 **Arrays.sort()** 时，可以保证对于同一个 **x** 位置，在增加时永远先处理 **height** 最高的建筑，而删除时永远先删除 **height** 最小的建筑，从而保证了在同一个高度上多个建筑 **Edge** 重叠时，算法的正确性。

- 如何正确处理建筑清光之后，剩余高度为 0 的情况？一开始在 TreeMap 里加一个 (0, 1) 的 entry，代表始终有一个高度为 0 的墙，即地面；

O(Edge 排序) + O(Edge 数 * (插入 / 删除 + 查找))

O(2n log 2n) + O(2n * (log 2n + log 2n)) = O(n log n)

```
public class Solution {
    private class Edge implements Comparable<Edge>{
        int x;
        int y;

        public Edge(int x, int y){
```

```
            this.x = x;
            this.y = y;
        }

        public int compareTo(Edge b){
            // Ascending in x coordinate
            if(this.x != b.x) return this.x - b.x;
            else return this.y - b.y;
        }
    }

    public List<int[]> getSkyline(int[][] buildings) {
        // Key : height of edge
        // Value : count of edges of that height
        TreeMap<Integer, Integer> treeMap = new TreeMap<>();
        List<int[]> list = new ArrayList<>();
        Edge[] edges = new Edge[2 * buildings.length];

        for(int i = 0; i < buildings.length; i++){
            int[] building = buildings[i];
            edges[i * 2] = new Edge(building[0], -building[2]);
            edges[i * 2 + 1] = new Edge(building[1], building[2]
);
        }

        Arrays.sort(edges);
        treeMap.put(0, 1);
        int prevHeight = 0;

        for(Edge edge : edges){
            if(edge.y < 0){
                if(!treeMap.containsKey(-edge.y)){
                    treeMap.put(-edge.y, 1);
                } else {
                    int count = treeMap.get(-edge.y);
                    treeMap.put(-edge.y, count + 1);
                }
            } else {
                int count = treeMap.get(edge.y);
                if(count == 1){
```

```
                    treeMap.remove(edge.y);
                } else {
                    treeMap.put(edge.y, count - 1);
                }
            }

            int curHeight = treeMap.lastKey();
            if(prevHeight != curHeight){
                list.add(new int[]{edge.x, curHeight});
                prevHeight = curHeight;
            }
        }

        return list;
    }
}
```

Trie，字典树

6/9, Trie, 字典树

说到 Trie，必须要贴一下这个 CSDN的帖子 ，讲的太好了。

1.1 什么是Trie树

Trie树，即字典树，又称单词查找树或键树，是一种树形结构，是一种哈希树的变种。典型应用是用于统计和排序大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。它的优点是：最大限度地减少无谓的字符串比较，查询效率比哈希表高。

Trie的核心思想是空间换时间。利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的。

它有3个基本性质：

- 根节点不包含字符，除根节点外每一个节点都只包含一个字符。
- 从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串。
- 每个节点的所有子节点包含的字符都不相同。

1.2 树的构建

举个在网上流传颇广的例子，如下：

题目：给你100000个长度不超过10的单词。对于每一个单词，我们要判断他出没出现过，如果出现了，求第一次出现在第几个位置。

分析：这题当然可以用hash来解决，但是本文重点介绍的是trie树，因为在某些方面它的用途更大。比如说对于某一个单词，我们要询问它的前缀是否出现过。这样hash就不好搞了，而用trie还是很简单。

现在回到例子中，如果我们用最傻的方法，对于每一个单词，我们都要去查找它前面的单词中是否有它。那么这个算法的复杂度就是O(n^2)。显然对于100000的范围难以接受。现在我们换个思路想。假设我要查询的单词是abcd，那么在他前面的单词中，以b，c，d，f之类开头的我显然不必考虑。而只要找以a开头的中是否存在abcd就可以了。同样的，在以a开头中的单词中，我们只要考虑以b作为第二个字母的，一次次缩小范围和提高针对性，这样一个树的模型就渐渐清晰了。

好比假设有b，abc，abd，bcd，abcd，efg，hii 这6个单词，我们构建的树就是如下图这样的：

当时第一次看到这幅图的时候，便立马感到此树之不凡构造了。单单从上幅图便可窥知一二，好比大海搜人，立马就能确定东南西北中的到底哪个方位，如此迅速缩小查找的范围和提高查找的针对性，不失为一创举。

ok，如上图所示，对于每一个节点，从根遍历到他的过程就是一个单词，如果这个节点被标记为红色，就表示这个单词存在，否则不存在。

那么，对于一个单词，我只要顺着他从根走到对应的节点，再看这个节点是否被标记为红色就可以知道它是否出现过了。把这个节点标记为红色，就相当于插入了这个单词。

这样一来我们查询和插入可以一起完成（重点体会这个查询和插入是如何一起完成的，稍后，下文具体解释），所用时间仅仅为单词长度，在这一个样例，便是10。

我们可以看到，trie树每一层的节点数是26^i级别的。所以为了节省空间。我们用动态链表，或者用数组来模拟动态。空间的花费，不会超过单词数×单词长度。

1.3 前缀查询

上文中提到”比如说对于某一个单词，我们要询问它的前缀是否出现过。这样hash就不好搞了，而用trie还是很简单“。下面，咱们来看看这个前缀查询问题：

已知n个由小写字母构成的平均长度为10的单词,判断其中是否存在某个串为另一个串的前缀子串。下面对比3种方法：

- 最容易想到的：即从字符串集中从头往后搜，看每个字符串是否为字符串集中某个字符串的前缀，复杂度为O(n^2)。
- 使用hash：我们用hash存下所有字符串的所有的前缀子串，建立存有子串hash的复杂度为O(n*len)*，而查询的复杂度为*O(n)* O(1)= O(n)。
- 使用trie：因为当查询如字符串abc是否为某个字符串的前缀时，显然以b,c,d....等不是以a开头的字符串就不用查找了。所以建立trie的复杂度为O(n*len)*，而建立+查询在*trie*中是可以同时执行的，建立的过程也就可以成为查询的过程，*hash*就不能实现这个功能。所以总的复杂度为*O(n*len)，实际查询的复杂度也只是O(len)。（说白了，就是Trie树的平均高度h为len，所以Trie树的查询复杂度为O（h）=O（len）。好比一棵二叉平衡树的高度为logN，则其查询，插入的平均时间复杂度亦为O（logN））。

下面解释下上述方法3中所说的为什么hash不能将建立与查询同时执行，而Trie树却可以：

在hash中，例如现在要输入两个串911，911456，如果要同时查询这两个串，且查询串的同时若hash中没有则存入。那么，这个查询与建立的过程就是先查询其中一个串911，没有，然后存入9、91、911；而后查询第二个串911456，没有然后存入9、91、911、9114、91145、911456。因为程序没有记忆功能，所以并不知道911在输入数据中出现过，只是照常以例行事，存入9、91、911、9114、911...。也就是说用hash必须先存入所有子串，然后for循环查询。

而trie树中，存入911后，已经记录911为出现的字符串，在存入911456的过程中就能发现而输出答案；倒过来亦可以，先存入911456，在存入911时，当指针指向最后一个1时，程序会发现这个1已经存在，说明911必定是某个字符串的前缀。

读者反馈@悠悠长风：关于这点，我有不同的看法。hash也是可以实现边建立边查询的啊。当插入911时，需要一个额外的标志位，表示它是一个完整的单词。在处理911456时，也是按照前面的查询9,91,911，当查询911时，是可以找到前面插入

的911，且通过标志位知道911为一个完整单词。那么就可以判断出911为911456的前缀啊。虽然trie树更适合这个问题，但是我认为hash也是可以实现边建立，边查找。

至于，有关Trie树的查找，插入等操作的实现代码，网上遍地开花且千篇一律，诸君尽可参考，想必不用我再做多余费神。

1.4 查询

Trie树是简单但实用的数据结构，通常用于实现字典查询。我们做即时响应用户输入的AJAX搜索框时，就是Trie开始。本质上，Trie是一颗存储多个字符串的树。相邻节点间的边代表一个字符，这样树的每条分支代表一则子串，而树的叶节点则代表完整的字符串。和普通树不同的地方是，相同的字符串前缀共享同一条分支。下面，再举一个例子。给出一组单词，inn, int, at, age, adv, ant, 我们可以得到下面的Trie：

可以看出：

- 每条边对应一个字母。
- 每个节点对应一项前缀。叶节点对应最长前缀，即单词本身。

- 单词inn与单词int有共同的前缀“in”, 因此他们共享左边的一条分支，root->i->in。同理，ate, age, adv, 和ant共享前缀"a"，所以他们共享从根节点到节点"a"的边。
- 查询操纵非常简单。比如要查找int，顺着路径i -> in -> int就找到了。

搭建Trie的基本算法也很简单，无非是逐一把每则单词的每个字母插入Trie。插入前先看前缀是否存在。如果存在，就共享，否则创建对应的节点和边。比如要插入单词add，就有下面几步：

- 考察前缀"a"，发现边a已经存在。于是顺着边a走到节点a。
- 考察剩下的字符串"dd"的前缀"d"，发现从节点a出发，已经有边d存在。于是顺着边d走到节点ad
- 考察最后一个字符"d"，这下从节点ad出发没有边d了，于是创建节点ad的子节点add，并把边ad->add标记为d。

1.5 Trie树的应用

除了本文引言处所述的问题能应用Trie树解决之外，Trie树还能解决下述问题（节选自此文：海量数据处理面试题集锦与Bit-map详解）：

3、有一个1G大小的一个文件，里面每一行是一个词，词的大小不超过16字节，内存限制大小是1M。返回频数最高的100个词。

9、1000万字符串，其中有些是重复的，需要把重复的全部去掉，保留没有重复的字符串。请怎么设计和实现？

10、 一个文本文件，大约有一万行，每行一个词，要求统计出其中最频繁出现的前10个词，请给出思想，给出时间复杂度分析。

13、寻找热门查询： 搜索引擎会通过日志文件把用户每次检索使用的所有检索串都记录下来，每个查询串的长度为1-255字节。假设目前有一千万个记录，这些查询串的重复读比较高，虽然总数是1千万，但是如果去除重复和，不超过3百万个。一个查询串的重复度越高，说明查询它的用户越多，也就越热门。请你统计最热门的10个查询串，要求使用的内存不能超过1G。

(1) 请描述你解决这个问题的思路；

(2) 请给出主要的处理流程，算法，以及算法的复杂度。

Implement Trie (Prefix Tree)

一开始试着用

```
HashSet<Character> || HashSet<TrieNode>
```

存子节点，发现都不太科学，第一种写法没法直接指向下一个 TrieNode，第二种写法每次查询都要新建一个 TrieNode...

既然 HashSet 底层也是直接调用 HashMap，不如用 HashMap 的 key-value pair 来的简单直接。

```
class TrieNode {
    // Initialize your data structure here.
    Character val;
    HashMap<Character, TrieNode> map;
    boolean isWord;
    public TrieNode() {
        val = null;
        map = new HashMap<Character, TrieNode>();
        isWord = false;
    }

    public TrieNode(char chr){
        val = chr;
        map = new HashMap<Character, TrieNode>();
        isWord = false;
    }
}

public class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    // Inserts a word into the trie.
    public void insert(String word) {
```

```
        TrieNode cur = root;
        for(int i = 0; i < word.length(); i++){
            char chr = word.charAt(i);
            if(!cur.map.containsKey(chr)){
                cur.map.put(chr, new TrieNode(chr));
            }
            cur = cur.map.get(chr);

            if(i == word.length() - 1) cur.isWord = true;
        }
    }

    // Returns if the word is in the trie.
    public boolean search(String word) {
        TrieNode cur = root;
        for(int i = 0; i < word.length(); i++){
            char chr = word.charAt(i);
            if(!cur.map.containsKey(chr)) return false;
            cur = cur.map.get(chr);
        }
        return cur.isWord;
    }

    // Returns if there is any word in the trie
    // that starts with the given prefix.
    public boolean startsWith(String prefix) {
        TrieNode cur = root;
        for(int i = 0; i < prefix.length(); i++){
            char chr = prefix.charAt(i);
            if(!cur.map.containsKey(chr)) return false;
            cur = cur.map.get(chr);
        }
        return true;
    }
}
```

Add and Search Word - Data structure design

其实这题如果没有 wildcard，直接用 hashmap 就撸完了。。。因为没有“前缀查询”这个可以明显区分 trie 和 hashmap 查询性能的操作。

题目设计者不会这么善罢甘休的，于是他们引入了 wildcard.

第一版写法，自定义新的 search 函数，自带 parent node，默认为 root. 每次遇到 wildcard 就以所有当前节点的分叉为 parent node 去遍历后面的。

然而 TLE 了。。看了一圈论坛发现写法和我的基本一样，不同之处是我用了 HashMap 省空间，他们用了 Array 省时间，于是我 TLE 了，他们却没有 MLE....

改进了一下，避免在 dfs 时用 substring 花费太多时间空间，而只用一个 string copy + pos index.

然并卵啊，还是超时。下面这段代码的正确性和用时在 lintcode 上是这样：

Accepted

100% test cases passed. Total Runtime: 3369 ms

```
public class WordDictionary {
    private class TrieNode{
        Character chr;
        HashMap<Character, TrieNode> map;
        boolean isWord;

        public TrieNode(){
            map = new HashMap<Character, TrieNode>();
        }
        public TrieNode(char chr){
            this.chr = chr;
            map = new HashMap<Character, TrieNode>();
            isWord = false;
        }
    }

    TrieNode root = new TrieNode();
```

```
    // Adds a word into the data structure.
    public void addWord(String word) {
        TrieNode cur = root;
        for(int i = 0; i < word.length(); i++){
            char chr = word.charAt(i);
            if(!cur.map.containsKey(chr)){
                cur.map.put(chr, new TrieNode(chr));
            }
            cur = cur.map.get(chr);
        }
        cur.isWord = true;
    }

    // Returns if the word is in the data structure. A word could

    // contain the dot character '.' to represent any one letter.

    public boolean search(String word) {
        return search(word, 0, root);
    }

    public boolean search(String word, int index, TrieNode node)
{
        if(index == word.length()) return node.isWord;
        char chr = word.charAt(index);

        if(chr == '.'){
            Set<Character> keySet = node.map.keySet();
            for(Character next : keySet){
                if(search(word, index + 1, node.map.get(next)))
return true;
            }
            return false;
        } else {
            if(!node.map.containsKey(chr)) {
                return false;
            } else {
                return search(word, index + 1, node.map.get(chr)
);
            }
```

```
        }
    }
}
```

同样的代码逻辑，把 map 由 HashMap 换成 array 就过了，说明速度上的差距啊。

- **HashMap trie:**
  - 优点
    - 支持所有 Character
    - 相对更省空间
  - 缺点
    - 查询时间相对长
    - 尤其是有 wildcard 做 DFS 时
- **Array trie:**
  - 优点
    - 查询速度快，尤其是有 wildcard 做 DFS 时
  - 缺点
    - 每个 node 空间占用大
    - 比较依赖指定字符集，比如 'a-z' 这种

LeetCode OJ 事实说明，这题更适合用 array trie 去解，也算是一个根据输入信息选择合适 implementation 的好例子，同样也可以推广到 union-find 的两种 implementation.

```
public class WordDictionary {
    private class TrieNode{
        char chr;
        TrieNode[] map;
        boolean isWord;
```

```
        public TrieNode(){
            map = new TrieNode[26];
        }
        public TrieNode(char chr){
            this.chr = chr;
            map = new TrieNode[26];
        }
    }

    TrieNode root = new TrieNode();

    // Adds a word into the data structure.
    public void addWord(String word) {
        TrieNode cur = root;
        for(int i = 0; i < word.length(); i++){
            char chr = word.charAt(i);
            int index = chr - 'a';
            if(cur.map[index] == null){
                cur.map[index] = new TrieNode(chr);
            }
            cur = cur.map[index];
        }
        cur.isWord = true;
    }

    // Returns if the word is in the data structure. A word could

    // contain the dot character '.' to represent any one letter.

    public boolean search(String word) {
        return search(word, 0, root);
    }

    public boolean search(String word, int index, TrieNode node)
{
        if(index == word.length()) return node.isWord;
        char chr = word.charAt(index);
        int chrIndex = chr - 'a';

        if(chr == '.'){
```

```
            for(int i = 0; i < 26; i++){
                if(node.map[i] != null){
                    if(search(word, index + 1, node.map[i])) ret
urn true;
                }
            }
            return false;
        } else {
            if(node.map[chrIndex] == null) {
                return false;
            } else {
                return search(word, index + 1, node.map[chrIndex
]);
            }
        }
    }
}
```

Word Search II

这题难度是 Hard，一是因为要用到数据结构 trie，另一方面又融合了 Word Search I 里面在矩阵里面搜索的 dfs + backtracking;

- 犯的错误：
  - 矩阵里 DFS 搜索要记得把当前格子 mark 成特殊符号，不然会死循环；
  - 同一个单词可能在矩阵中出现多次，不要重复添加。
  - 因为是先添加新 char 然后直接 dfs 下个位置，并且在递归一开始添加单词，正确的检查位置是在边界检查之前，否则 ["a"] 和 ["a"] 的情况无法正确添加结果，因为 "a" 的四个方向都越界了。

```
public class Solution {
    private class TrieNode{
        char chr;
        boolean isWord;
        TrieNode[] children;

        public TrieNode(){
```

```
            children = new TrieNode[26];
        }

        public TrieNode(char chr){
            this.chr = chr;
            children = new TrieNode[26];
        }
    }

    private void add(String word, TrieNode root){
        TrieNode cur = root;
        for(int i = 0; i < word.length(); i++){
            char chr = word.charAt(i);
            int index = chr - 'a';
            if(cur.children[index] == null){
                cur.children[index] = new TrieNode(chr);
            }
            cur = cur.children[index];
        }
        cur.isWord = true;
    }

    private void searchAndAdd(TrieNode root, char[][] board, int
 x, int y,
                              List<String> list, StringBuilder s
b){
        if(root.isWord) {
            if(!list.contains(sb.toString())) list.add(sb.toStri
ng());
        }

        if(x < 0 || x >= board.length) return;
        if(y < 0 || y >= board[0].length) return;
        if(board[x][y] == '#') return;



        char chr = board[x][y];
        int index = chr - 'a';
        if(root.children[index] == null){
```

```
            return;
        } else {
            int length = sb.length();
            int[] xDirections = {0,0,1,-1};
            int[] yDirections = {-1,1,0,0};

            for(int i = 0; i < 4; i++){
                sb.append(chr);
                board[x][y] = '#';
                searchAndAdd(root.children[index], board, x + xD
irections[i], y + yDirections[i], list, sb);
                board[x][y] = chr;
                sb.setLength(length);
            }
        }
    }


    public List<String> findWords(char[][] board, String[] words)
 {
        List<String> list = new ArrayList<String>();
        if(board == null || board.length == 0 ||
           words == null || words.length == 0)  return list;

        TrieNode root = new TrieNode();

        for(String word : words){
            add(word, root);
        }

        for(int i = 0; i < board.length; i++){
            for(int j = 0; j < board[0].length; j++){
                searchAndAdd(root, board, i, j, list, new String
Builder());
            }
        }

        return list;
    }
}
```

Trie Serialization

九章答案， 我得承认这个写法比我自己之前写的序列化二叉树更巧妙。

这题的解法和我之前自己写的那个 binary tree serialization 其实挺像的，结构都是

root【子树1】【子树2】【子树3】

考虑到 Trie 都有个 dummy root，所以是

【root【子树1】【子树2】【子树3】】

追求这题的 AC 稍微有点太细节了，先放着，我先看看别的题去吧。

LIS，单调栈类

Longest Increasing Subsequence

先从 **O(n^2)** 的 **DP** 做法说起。先从这个开做，是因为 **O(n^2)** 做法的普适性高，而且可以无视维度，易于扩展到各种变形与 **follow-up** 上。

需要返回具体 **LIS sequence** 的时候，加一个新 **array**，记录 **parent index** 一路追踪就可以了~ 和 **Largest Divisible Subset** 一样。

```
public class Solution {
    public int lengthOfLIS(int[] nums) {
        if(nums == null || nums.length == 0) return 0;

        int[] dp = new int[nums.length];
        Arrays.fill(dp, 1);

        int max = 1;

        for(int i = 0; i < nums.length; i++){
            for(int j = i - 1; j >= 0; j--){
                if(nums[i] > nums[j]){
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                    max = Math.max(max, dp[i]);
                }
            }
        }

        return max;
    }
}
```

- 这里要注意“等于号”，不然返回的**index**会错
- **if(dp[start] >= target) return start;**

- **if(dp[end] < target) return end + 1;**
- **return end;**
- 如果 **binary search** 里面指针一直在 **agressively** 往右走，那么最终的返回位置不外乎 **left, right** 和 **right + 1.** 所有小于等于 **num[left]** 的情况都返回 **left.**

这个问题的 follow up 也很有意思，返回一条最长的 subsequence，还有返回所有的 subsequence.

Given an unsorted array of integers, find the length of longest increasing subsequence.

For example, Given [10, 9, 2, 5, 3, 7, 101, 18], The longest increasing subsequence is [2, 3, 7, 101], therefore the length is 4. Note that there may be more than one LIS combination, it is only necessary for you to return the length.

Your algorithm should run in O(n2) complexity.

Follow up: Could you improve it to O(n log n) time complexity?

> A O(n^2) solution is, do Longest Common Subsequence between original array and sorted one, the LCS would be the LIS.
>
> This is also "Online Algorithm" which we keep taking streams of input data one at a time.
>
> For improvement, the key is take advantage of the "Increasing Sequence"
>
> Terminating condition for "Search insert position" is important and needs to be accurate.

```
public class Solution {
    public int lengthOfLIS(int[] nums) {
        if(nums == null || nums.length == 0) return 0;
        int[] dp = new int[nums.length];
        dp[0] = nums[0];
        int ptr = 0;
        for(int i = 1; i < nums.length; i++){
            int pos = binarySearch(dp, 0, ptr, nums[i]);
            if(dp[pos] > nums[i]) dp[pos] = nums[i];
            if(pos > ptr){
                ptr = pos;
                dp[pos] = nums[i];
            }
        }

        return ptr + 1;
    }

    private int binarySearch(int[] dp, int start, int end, int t
arget){
        while(start + 1 < end){
            int mid = start + (end - start) / 2;
            if(target > dp[mid]){
                start = mid;
            } else {
                end = mid;
            }
        }

        // This equality sign is important
        // otherwise it would return wrong insert position
        if(dp[start] >= target) return start;
        if(dp[end] < target) return end + 1;

        return end;
    }
}
```

栈， 单调栈

单调栈例题

有N个人，顺次排列，他们的身高分别为H[i]，每个人向自己后方看，他能看到的人是在他后面离他最近的且比他高的人。请依次输出每个人能看到的人的编号Next[i]，如果他后面不存在比他高的人，则输出-1。

想找 "从当前元素向某一方向的第一个 (大于 / 小于) 自己的元素"，就要靠单调栈来维护单调性，对应的是 (递减 / 递增)。

```
public class Main {

    public static int[] getHeight(int[] heights){
        int n = heights.length;
        int[] arr = new int[n];

        // Stack stores index of people
        Stack<Integer> stack = new Stack<>();
        for(int i = 0; i < n; i++){
            while(!stack.isEmpty() && heights[stack.peek()] <= h
eights[i]){
                arr[stack.pop()] = i;
            }
            stack.push(i);
        }

        while(!stack.isEmpty()){
            arr[stack.pop()] = -1;
        }

        return arr;
    }

    public static void main(String[] args){
        int[] heights1 = {3,2,5,6,1,2};
        int[] heights2 = {1,2,3,4,5,6};
        int[] heights3 = {6,5,4,3,2,1};
        int[] heights4 = {6,1,3,4,2,5};

        int[] arr = getHeight(heights4);
        for(int i = 0; i < arr.length; i++){
            System.out.print(arr[i] + "  ");
        }
    }
}
```

Largest Rectangle in Histogram

顺着上一题的思路，这题比较暴力的解法可以分为三步：

- 从左向右扫，寻找对于每个元素，往右能看到的最远距离；
- 从右向左扫，寻找对于每个元素，往左能看到的远距离；
- 把两个 **arr[]** 的结果综合起来，就是从每个位置出发能构造的最大 **rectangle.**

速度非常慢，只超过了 1.27% ，因为常数项更大。虽然时间复杂度是 O(n)，但是用了 three-pass 才搞定。如果说这种解法有什么优点，就是比较好理解。。是单调栈非常简单直接的应用方式，不加任何 trick.

```
public class Solution {
    public int largestRectangleArea(int[] heights) {
        int[] right = findToRight(heights);
        int[] left = findToLeft(heights);

        int max = 0;
        for(int i = 0; i < heights.length; i++){
            max = Math.max(max, (right[i] - left[i] + 1) * heigh
ts[i]);
        }

        return max;
    }

    private int[] findToRight(int[] heights){
        int[] right = new int[heights.length];
        Stack<Integer> stack = new Stack<>();
        for(int i = 0; i < heights.length; i++){
            while(!stack.isEmpty() && heights[stack.peek()] > he
ights[i]){
                right[stack.pop()] = i - 1;
            }
            stack.push(i);
        }
        while(!stack.isEmpty()){
            int index = stack.pop();
```

```
            right[index] = heights.length - 1;
        }
        return right;
    }

    private int[] findToLeft(int[] heights){
        int[] left = new int[heights.length];
        Stack<Integer> stack = new Stack<>();
        for(int i = heights.length - 1; i >= 0; i--){
            while(!stack.isEmpty() && heights[stack.peek()] > he
ights[i]){
                left[stack.pop()] = i + 1;
            }
            stack.push(i);
        }
        while(!stack.isEmpty()){
            int index = stack.pop();
            left[index] = 0;
        }
        return left;
    }
}
```

Increasing Triplet Subsequence

用 LIS 的角度理解的话这题就是无脑套模板。。

不过题目要求 O(n) 时间 O(1) 空间，就得另外动动脑筋了。

```java
public class Solution {
    public boolean increasingTriplet(int[] nums) {
        if(nums == null || nums.length < 3) return false;
        int n = nums.length;
        int[] dp = new int[n];
        dp[0] = nums[0];
        int index = 0;
        for(int i = 1; i < n; i++){
            int pos = binarySearch(dp, 0, index, nums[i]);
            if(pos > index){
                dp[pos] = nums[i];
                index = pos;
                if(index + 1 >= 3) return true;
            }
            dp[pos] = Math.min(dp[pos], nums[i]);
        }
        return false;
    }

    private int binarySearch(int[] nums, int start, int end, int
 target){
        int left = start;
        int right = end;
        while(left + 1 < right){
            int mid = left + (right - left) / 2;
            if(nums[mid] < target){
                left = mid;
            } else {
                right = mid;
            }
        }

        if(target <= nums[left]) return left;
        if(target > nums[right]) return right + 1;

        return right;
    }
}
```

通过观察这题的具体性质，我们发现在这里我们所谓的 "单调栈" 长度其实是固定的，就是3，等于3了直接返回就行。

因为 3 非常小，我们只需要存两个值，分别代表着单调栈的第一个和第二个位置；当我们碰到第三个位置的情况，就可以返回 true 了。

这个解法的思想也可以推广到任意 k ，k比较小或者为常数的情况，毕竟对于常数 k ，O(k) = O(1).

```
public class Solution {
    public boolean increasingTriplet(int[] nums) {
        if(nums == null || nums.length < 3) return false;
        int n = nums.length;

        int num1 = Integer.MAX_VALUE;
        int num2 = Integer.MAX_VALUE;
        for(int i = 0; i < n; i++){
            if(nums[i] <= num1){
                num1 = nums[i];
            } else if(nums[i] <= num2){
                num2 = nums[i];
            } else {
                return true;
            }
        }

        return false;
    }
}
```

Russian Doll Envelopes

[[2,100],[3,200],[4,300],[5,500],[5,400],[5,250],[6,370],[6,360],[7,380]]

这题试着用 LIS 的写法思路写了一下，然而卡在了这个 test case 上。

- **因为这题，元素之间不存在绝对的大小，不能直接用两个 tuple 去比较，进行 binary search.**

- 两个 **tuple [4,300]** 和 **[5,250]** 之间，按理可以说 **[4, 300]** 更小，但是有可能最优解是由 **[5,250]** 选出来的。

正解的流程：

- 按 **[w, h]** 中的 **w** 升序排序；
- 如果 **w** 相等，则按照 **h** 的降序排序**(**重要！！！**)**
- 后面的就和一维 **LIS** 一样，只考虑 **h** 的维度做 **LIS** 就行了。

难点在于，为什么这么做是正确的？

- 不难看出对于同一个 **w** 的楼层，我们不能按 **h** 升序排列，因为会延长自身，导致在同一个 **w** 上多次套用。
- 因此对于同一 **w** 的情况， 要按照 **h** 降序排列
- 这样当我们看到一个更大的 **h** 时，我们可以确定，这一定是一个新的 **w**，而且根据原来排序的单调性，一定是一个比单调栈内元素都大的新 **w**，可以套上。
- 同时对于单调栈中的任意 **h**，如果 **binary search** 之后的位置 **pos** 位于中间，它一定可以套在 **pos** 之前的所有信封上。

速度超过 87.50%，还不错。

```
public class Solution {
    private class MyComparator implements Comparator<int[]>{
        public int compare(int[] a, int[] b){
            if(a[0] != b[0]) return (a[0] < b[0]) ? -1: 1;
            else return (a[1] < b[1]) ? 1: -1;
        }
    }

    public int maxEnvelopes(int[][] envelopes) {
        if(envelopes == null || envelopes.length == 0) return 0;
```

```
        Arrays.sort(envelopes, new MyComparator());
        int n = envelopes.length;
        int[] dp = new int[n];
        dp[0] = envelopes[0][1];
        int index = 0;

        for(int i = 1; i < n; i++){
            int pos = binarySearch(dp, 0, index, envelopes[i][1]
);
            if(pos > index){
                dp[pos] = envelopes[i][1];
                index = pos;
            }
            dp[pos] = Math.min(dp[pos], envelopes[i][1]);
        }
        return index + 1;
    }

    private int binarySearch(int[] dp, int start, int end, int h
eight){
        int left = start;
        int right = end;
        while(left + 1 < right){
            int mid = left + (right - left) / 2;

            if(height < dp[mid]){
                right = mid;
            } else {
                left = mid;
            }
        }
        if(height <= dp[left]) return left;
        if(height > dp[right]) return right + 1;

        return right;
    }
}
```

Maximal Rectangle

Largest Divisible Subset

这个博客的文章讲的不错~ 重点比我说的好。**https://segmentfault.com/a/1190000005922634**

Largest Divisible Subset

把这题放在 LIS 分类下面，主要是因为长的和 LIS 的 O(n^2) DP 解法很像。

这题正确性的保证：对于排序数组 **nums** 的两个 **index**，**i, j** 并且 **j < i** 的情况下，如果 **nums[i] % nums[j] == 0**，那么包含 **nums[j]** 的 **subset** 里所有元素也一定能整除 **nums[i].** 因为 **nums[j]** 是其 **subset** 中当前最大的元素，而且一定可以整除所有比它小的。

主要不同点：

- 因为最后要输出结果，得存个 **parent** 数组记录每个序列的前一个元素
- 每次往回扫的时候，不能像 **LIS** 那样看到大的就停手，要走到底

这么讲的话，貌似要输出具体 **LIS** 序列的题，也可以这么做。。**O(n^2)** 时间，**O(n)** 空间就可以了。

```
public class Solution {
    public List<Integer> largestDivisibleSubset(int[] nums) {
        List<Integer> rst = new ArrayList<>();
        if(nums == null || nums.length == 0) return rst;

        Arrays.sort(nums);
        int[] dp = new int[nums.length];
```

```
        int[] parent = new int[nums.length];
        Arrays.fill(dp, 1);
        Arrays.fill(parent, -1);

        int maxIndex = -1;
        int maxLen = 1;

        for(int i = 0; i < nums.length; i++){
            for(int j = i - 1; j >= 0; j--){
                if(nums[i] % nums[j] == 0 && dp[i] < dp[j] + 1){
                    dp[i] = dp[j] + 1;
                    parent[i] = j;

                    if(dp[i] > maxLen){
                        maxLen = dp[i];
                        maxIndex = i;
                    }
                }
            }
        }

        if(maxIndex == -1){
            rst.add(nums[0]);
        } else {
            while(maxIndex != -1){
                rst.add(nums[maxIndex]);
                maxIndex = parent[maxIndex];
            }
        }

        return rst;
    }
}
```

Binary Search 类

Matrix Binary Search

Search a 2D Matrix

考点只有一个：2D array 降维成 1D array 的 index trick.

```
public class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        if(matrix == null || matrix.length == 0) return false;

        int rows = matrix.length;
        int cols = matrix[0].length;

        int left = 0;
        int right = rows * cols - 1;
        while(left <= right){
            int mid = left + (right - left) / 2;
            int row = mid / cols;
            int col = mid % cols;

            if(matrix[row][col] == target){
                return true;
            } else if(matrix[row][col] < target){
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }

        return false;
    }
}
```

Search a 2D Matrix II

实现的时候稍微纠结了一下收边的问题，后来一想，如果到了边上还走到越界的位置上，已经说明没有合理解了，跳出循环返回 false 就行。

```
public class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        if(matrix == null || matrix.length == 0) return false;

        int rows = matrix.length;
        int cols = matrix[0].length;

        int x = rows - 1;
        int y = 0;

        while(x >= 0 && y < cols){
            int cur = matrix[x][y];
            if(cur == target){
                return true;
            } else if(cur < target){
                y ++;
            } else {
                x --;
            }
        }

        return false;
    }
}
```

Array Binary Search

Search for a Range

典型的 binary search 题，我还是比较喜欢用 left + 1 < right，不用担心各种奇葩输入导致的越界。

```java
public class Solution {
    public int[] searchRange(int[] nums, int target) {
        int left = 0;
        int right = nums.length - 1;
        int leftEnd = left, rightEnd = right;
        while(left + 1 < right){
            int mid = left + (right - left) / 2;
            if(nums[mid] >= target){
                right = mid;
            } else {
                left = mid;
            }
        }
        if(nums[left] == target) leftEnd = left;
        else if(nums[right] == target) leftEnd = right;
        else leftEnd = -1;

        left = 0;
        right = nums.length - 1;

        while(left + 1 < right){
            int mid = left + (right - left) / 2;
            if(nums[mid] <= target){
                left = mid;
            } else {
                right = mid;
            }
        }
        if(nums[right] == target) rightEnd = right;
        else if(nums[left] == target) rightEnd = left;
        else rightEnd = -1;

        int[] rst = new int[2];
        rst[0] = leftEnd;
        rst[1] = rightEnd;

        return rst;
    }
}
```

First Bad Version

这道不是 LinkedIn 面经题。

是我被 DP 虐多了之后，来无耻地灌个水找找自信。。

```
public class Solution extends VersionControl {
    public int firstBadVersion(int n) {
        int left = 1;
        int right = n;
        while(left + 1 < right){
            int mid = left + (right - left) / 2;
            if(isBadVersion(mid)){
                right = mid;
            } else {
                left = mid;
            }
        }

        if(isBadVersion(left)) return left;
        else if(isBadVersion(right)) return right;

        return -1;
    }
}
```

Search Insert Position

这道不是 LinkedIn 面经题。

是我被 DP 虐多了之后，来无耻地灌个水找找自信。。

```
public class Solution {
    public int searchInsert(int[] nums, int target) {
        int left = 0;
        int right = nums.length - 1;
        while(left + 1 < right){
            int mid = left + (right - left) / 2;
            if(nums[mid] == target){
                return mid;
            } else if(nums[mid] < target){
                left = mid;
            } else {
                right = mid;
            }
        }

        if(target <= nums[left]) return left;
        if(target <= nums[right]) return right;
        if(target > nums[right]) return right + 1;

        return 0;
    }
}
```

Search in Rotated Sorted Array

这题在地里看到过，当时把猴子给弄挂了。。

这题很重要的性质是，没有重复元素。

可以根据 **A[mid]** 与 **A[right]** 的大小关系，先行判断 **mid** 一定位于哪一端；

对于已经确定 **mid** 左/右 的数组，必然有一段区间满足单调性，可以利用来做 **binary search.**

```
public class Solution {
    public int search(int[] nums, int target) {
        int left = 0;
        int right = nums.length - 1;

        while(left + 1 < right){
            int mid = left + (right - left) / 2;

            if(nums[mid] >= nums[right]){
                if(target <= nums[mid] && target >= nums[left]){
                    right = mid;
                } else {
                    left = mid;
                }
            } else {
                if(target >= nums[mid] && target <= nums[right])
{
                    left = mid;
                } else {
                    right = mid;
                }
            }
        }

        if(nums[left] == target) return left;
        if(nums[right] == target) return right;

        return -1;
    }
}
```

Find Minimum in Rotated Sorted Array

在把上一道题做完之后，这题就是一个乞丐版的 search in rotated sorted array. 因为已知没有 duplicates 就更简单了。

```
public class Solution {
    public int findMin(int[] nums) {
        int left = 0;
        int right = nums.length - 1;

        while(left + 1 < right){
            int mid = left + (right - left) / 2;
            if(nums[mid] >= nums[right]){
                left = mid;
            } else {
                right = mid;
            }
        }

        return Math.min(nums[left], nums[right]);
    }
}
```

Find Minimum in Rotated Sorted Array II

加上有重复元素的条件之后，这题立刻就变成 Hard 难度了。原因在于出现了新情况，即我们不知道最小值到底在 mid 的左边还是右边，比如【3,[3],1,3】，中间的 3 和两端值都相等，无法正确地直接砍掉一半。

- 我们依然可以靠 **A[mid]** 与 **A[right]** 的大小关系来二分搜索；
    - **A[mid] > A[right]** 时，**mid** 在左半边，最小值一定在 **mid** 右侧；
    - **A[mid] < A[right]** 时，**mid** 在右半边，最小值一定在 **mid** 左侧；
- **A[mid] == A[right]** 时，无法判断，把 **right** 向左挪一格。

```java
public class Solution {
    public int findMin(int[] nums) {
        int left = 0;
        int right = nums.length - 1;

        while(left + 1 < right){
            int mid = left + (right - left) / 2;
            if(nums[mid] > nums[right]){
                left = mid;
            } else if(nums[mid] < nums[right]){
                right = mid;
            } else {
                right --;
            }
        }

        return Math.min(nums[left], nums[right]);
    }
}
```

Find Peak Element I & II

Find Peak Element

这题有两个重要条件：

- 相邻数字绝不重复
- 数组两端有 -Inf 做 padding

这就意味着在这样的给定条件下，区间 [0, n - 1] 一定存在一个 peak. 我们需要做的，就是利用 binary search 逐渐缩小这个有效区间的大小。

对于 nums[mid]:

- 如果检查其相邻元素，发现 mid 为单调上升，那么 mid 自己可以替代数组一端的 -Inf 作用，自己作为新区间起点的 padding，因为 [mid + 1, right] 区间内一定有 peak;
- 同理，如果 mid 处于一个单调下降的位置，那么 mid 自己可以取代原本的右侧 padding，[left, mid - 1] 区间内一定有 peak;
- 如果两边的元素都比 nums[mid] 大，那么两边都有 peak.
- 如果两边的元素都比 nums[mid] 小，mid 自己就是 peak.

```
public class Solution {
    public int findPeakElement(int[] nums) {
        if(nums == null || nums.length == 0) return -1;
        if(nums.length == 1) return 0;

        int left = 0;
        int right = nums.length - 1;

        while(left + 1 < right){
            int mid = left + (right - left) / 2;
            if(nums[mid - 1] < nums[mid] && nums[mid] > nums[mid
 + 1]){
                // nums[mid] is valid peak
                return mid;
            } else if(nums[mid - 1] < nums[mid] && nums[mid] < n
ums[mid + 1]){
                // nums[mid] is on an increasing trend
                left = mid;
            } else if(nums[mid - 1] > nums[mid] && nums[mid] > n
ums[mid + 1]){
                // nums[mid] is on a decreasing trend
                right = mid;
            } else {
                // nums[mid] is smaller than its neighbor, both
directions have peak
                right = mid;
            }
        }

        return (nums[left] < nums[right]) ? right: left;
    }
}
```

Find Peak Element II

上一题的单调性，是以“点”为单位的，这次我们扩展到 2D array，就要开始以 “行 / 列” 为单位了。

初始条件是类似的：最外围的元素都小于里面，相当于做了一个 padding，同时相邻元素不相等，保证了矩阵里面一定存在 peak. 我们要做的，就是缩小需要查找的矩阵范围。

关键在于 ： 用当前行 / 列最大值的位置，代表整行 / 列。

- 在“灌水”问题里，一个一维数组某个区间内的水位，取决于两端最小值；二维矩阵里，水位也取决于 “木桶” 里面最短的那块板。
- 在“山峰”问题里，一个一维数组里，山峰的位置取决于里面的最大值，或局部最大值；在二维矩阵里，一行的 max 决定了这行所能达到的最高高度，在和相邻元素进行比较时，相邻元素若比 max 大，则 max 所属的行列，就饿一定可以做为新的 padding 边界，把这种单调性传递下去。

```java
public List<Integer> findPeakII(int[][] A) {
    // write your code here
    List<Integer> list = new ArrayList<>();
    int left = 1;
    int right = A[0].length - 2;
    while(left <= right){
        int mid = left + (right - left) / 2;
        int index = getColMaxIndex(A, mid);
        if(A[index][mid] < A[index][mid - 1]){
            right = mid - 1;
        } else if(A[index][mid] < A[index][mid + 1]){
            left = mid + 1;
        } else {
            list.add(index);
            list.add(mid);
            return list;
        }
    }
    return list;
}

private int getColMaxIndex(int[][] A, int col){
    int index = 0;
    int max = A[0][col];
    for(int i = 1; i < A.length - 1; i++){
        if(A[i][col] > max){
            max = A[i][col];
            index = i;
        }
    }
    return index;
}
```

**Median of Two Sorted Arrays

Median of Two Sorted Arrays

对于中位数问题，首先要做的是明白“找中位数” 等价于 find kth largest element，奇数元素找一遍，偶数元素找两遍。

所谓 “第 kth 的元素”，也叫 Order Statistic，在算法导论上有章节对这类问题有很详细的描述。

有向箭头**"A -> B"**代表 **"A > B"**，图中黑点为元素**(quick-select** 分组出来的，不是排序**)**，白点为分组中位数，阴影部分元素，都一定比 **x** 大。

这个算法的核心思想是，每次可以扔掉 **A** 或 **B** 里面，较小的那 **k / 2** 个数，使得 **A** 与 **B** 的剩余搜索范围单调向右，而 **k** 指数缩小。

复杂度 **O(log k)**，**k = (m + n) / 2**

```
public class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums
2) {
        int len = nums1.length + nums2.length;
        if(len % 2 == 1){
            return findKth(nums1, 0, nums2, 0, len / 2 + 1);
        } else {
            return (findKth(nums1, 0, nums2, 0, len / 2) +
                    findKth(nums1, 0, nums2, 0, len / 2 + 1)) /
2.0;
        }
    }

    private int findKth(int[] A, int startA, int[] B, int startB
, int k){
        if(startA == A.length) return B[startB + k - 1];
        if(startB == B.length) return A[startA + k - 1];
        if(k == 1) return Math.min(A[startA], B[startB]);

        int keyA = (startA + k / 2 - 1 < A.length)
                    ? A[startA + k / 2 - 1]
                    : Integer.MAX_VALUE;
        int keyB = (startB + k / 2 - 1 < B.length)
                    ? B[startB + k / 2 - 1]
                    : Integer.MAX_VALUE;
        if(keyA < keyB){
            return findKth(A, startA + k / 2, B, startB, k - k /
2);
        } else {
            return findKth(A, startA, B, startB + k / 2, k - k /
2);
        }
    }
}
```

(F) Median of K sorted arrays

http://stackoverflow.com/questions/6182488/median-of-5-sorted-arrays

其实这个博客里解 median of Two sorted array 的思路更适合解这个 k 的情况，因为这个做法更像图里的 order statistics:

http://fisherlei.blogspot.com/2012/12/leetcode-median-of-two-sorted-arrays.html

这题比较简单的使用 heap 的解法也可以用类似于 Kth Smallest Element in a Sorted Matrix 的 minHeap 做法；自定义一个 Tuple，存有 x, y 和 val 信息并根据 val 值 implement comparable interface，相当于在 m 个排序数组里面找 kth smallest number.

时间复杂度：**m** 个数组，假如每个数组元素平均为 **n**，总共有 **m*n** 个元素，中位数要找 **mn/2** 小的元素，**heap size** 最大为 **m**

因此 **O(mn * log(m))**

Topological Sort，拓扑排序

有向 / 无向 图的基本性质和操作

Graph 是非常重要而又涵盖很广的内容，以至于有单独的“图论”研究方向。LeetCode 上很多问题都可以抽象成“图”，比如搜索类问题，树类问题，迷宫问题，矩阵路径问题，等等。

摘自上学期的 AI 课 CS520 slides，重点在于 DFS 和 BFS 的比较；可以看到；

- BFS 的时间空间占用以 branching factor 为底， 到解的距离 d 为指数增长；空间占用上 Queue 是不会像 DFS 一样只存一条路径的，而是从起点出发越扩越大，因此会有空间不够的风险，空间占用为 O(b^d)。
- DFS 的时间占用以 branching factor 为底，树的深度 m 为指数增长；而空间占用上，却只是 O(bm)，可视化探索过程中只把每个 Node 的所有子节点存在 Stack 上， 探索完了再 pop 出来接着探，因此储存的节点数为 O(bm)。

可以看到无论是 BFS 还是 DFS，树的 branching factor 都是对空间与时间复杂度影响最大的参数；除此之外，BFS 中最重要的是到解的距离，而 DFS 看从当前节点的深度。普遍来讲，DFS 空间上会经济一些，当然也要分情况讨论。

Summary on Uninformed (Tree) Search

Method	Complete?	Optimal?	Time	Space
BFS	Yes, for $b < \infty$	Yes, for uniform edge costs	$O(b^d)$	$O(b^d)$
Uniform cost	Yes, for $b < \infty$ and $\epsilon > fixed\ \delta > 0$	Yes	$O(b^{1+\frac{C^*}{\epsilon}})$	$O(b^{1+\frac{C^*}{\epsilon}})$
DFS	No	No	$O(b^m)$	$O(bm)$
Depth limited DFS	No	No	$O(b^{\ell})$	$O(b\ell)$
Iterative deepening DFS	Yes, for $b < \infty$	Yes, for uniform edge costs	$O(b^d)$	$O(bd)$
Bidirectional	Yes, for BFS with $b < \infty$	Yes, for uniform edge costs	$O(b^{\frac{d}{2}})$	$O(b^{\frac{d}{2}})$

b: branching factor (average)
d: max. solution depth (i.e., depth of x_G)
m: max. depth of any node from x_I
ℓ: depth limit
C^*: optimal solution cost
ϵ: smallest step (edge) cost, must be finite

⇨For graph search on finite state space, DFS is complete and depth limited DFS is complete when $\ell \geq d$

Topological sort，拓扑排序，是 graph 搜索中一种特殊的顺序，本质上还是完全可以靠 BFS / DFS 解决。

有向图 **DFS** 图示：

Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Timestamps within vertices indicate discovery time/finishing times.

有向图 **BFS** 图示：

Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. The value of $u.d$ appears within each vertex u. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances appear below vertices in the queue.

可以看到两种搜索都用了三种颜色表示状态。

- 有向图 **Directed:**
  - **DFS:**
    - **Detect Cycle (Course Schedule)**

> - 暴力解法：DFS + Backtracking，寻找“所有从当前节点的” path，如果试图访问 visited 则有环；缺点是，同一个点会被探索多次，而且要从所有点作为起点保证算法正确性，时间复杂度非常高
> - 最优解法是 CLRS 上用三种状态表示每个节点：
> - "0" 还未访问过;
> - "1" 代表正在访问；
> - "2" 代表已经访问过；
> - DFS 开始时把当前节点设为 "1";
> - 在从任意节点出发的时候，如果我们试图访问一个状态为 "1" 的节点，都说明图上有环。
> - 当前节点的 DFS 结束时，设为 "2";
> - 在找环上，DFS 比起 BFS 最大的优点是"对路径有记忆"，DFS 记得来时的路径和导致环的位置，BFS 却不行。

- ### Topological Sort

> - 类似剥洋葱，可以选择任意点开始向里 DFS ，记录path，后面做的其他 DFS path 都放在之前结果的前面。(CLRS有)
> - 等价做法是，把每次 DFS 探索中的 path 按照由内向外的顺序添加(单序列反序), 后进行的 DFS 结果放在之前运行完的结果后面(全序列反序)，然后 reverse 就好了，可以改良 ArrayList 添加的时间复杂度。
> - 更简单的做法是用 Java 内置的 LinkedList，支持双端操作。

- **Count # of connected components**

- **Determine if A is reachable from B**

BFS：

- **Detect Cycle (Course Schedule)**

> - 首先，在 DAG 上找环，DFS 要比 BFS 强。
> - 其次重点注意，有向图靠 **3** 个状态 **BFS** 是不能正确判断是否有环的，要靠 **indegree**，一个例子是 【**0, 1**】 和 【**1,0**】，环在发现之前已经被标注为 **state 2** 了。
> - 扫一遍所有 edge，记录每个节点的 indegree.
> - 在有向无环图中，一定会存在至少一个 indegree 为 0 的起点，将所有这样的点加入queue。
> - 依次处理queue里的节点，把每次poll出来的节点的 children indegree -1. 减完之后如果 child 的 indegree = 0 了，就也放入队列。
> - 如果图真的没有环，可以顺利访问完所有节点，如果还有剩的，说明图中有环，因为环上节点的 indegree 没法归 0.

- ### Topological Sort

> - 步骤同上，扫的时候按顺序 append 就好了。
> - 类似于“剥洋葱”，从最外面一层出发，逐步向里。

- ### Count # of connected components

- ### Determine if A is reachable from B

> - 这个任务最适合用 BFS 做，从 B 做起点一路扫看看能不能扫到 A 就行了。

- ## Un-directed:

无向图算法的整体思路和有向图基本一致，但是需要重点注意 正确处理 “原路返回” 的情况，免得死循环或者误报。

在无向图的**2**个状态基础上增加一个 “正在访问” 的新状态，加上注意 **prev** 就可以了。

- **DFS:**
  - **Detect Cycle (Graph Valid Tree)**

    > - 用 int[] 表示每个点的状态，其中
    > - 0 代表“未访问”；
    > - 1 代表“访问中”；
    > - 2 代表“已访问”；
    >
    > - **DFS call 里要传入 "prev节点" 参数，避免出现原路返回，或者回到前一个节点误判为有环。(和 directed graph DFS 唯一的不同之处)**
    > - 其他情况下，如果我们试图访问一个状态为 “1” 的节点，都可以说明图中有环。

  - **Topological Sort**

    > - 都无向图了，topological 个毛

  - **Count # of connected components (Graph Valid Tree)**
  - **Determine if A is reachable from B**
- **BFS：**
  - **Detect Cycle (Graph Valid Tree)**

> - 扫描所有 edges 记录图到底长啥样；
> - 用 "0,1,2" states;
> - 随便扔一个点进 queue，标记 "1"，然后 BFS，所有 child = "0" 的都加入队列
>
> - **所有 child 都检查完之后，立刻把当前 node = 2，不然下一层 BFS 会回头去看自己然后误报。**
>
> - 如果遇到 child = "1" 的说明有环

- **Topological Sort**

> - 都无向图了，topological 个毛

- **Count # of connected components (Graph Valid Tree)**

> - 和 Detect Cycle 一样，同时记录下到底做了几次 BFS 才扫遍全图，图上就有几个 connected components

- **Determine if A is reachable from B**

> - BFS 搜到底看看能不能到就可以了，这个和只靠 g(x) 的 uniform-cost search 一致。

拓扑排序 DFS 做法

LintCode - Topological Sort

CLRS 上图论的一小节。给定一个有向图，在拓扑排序中可以有很多个正确解，由若干小段的 list 组成。但是要保证：

> 正确的单序列顺序（具体一个 **list** 之间的元素）
>
> 正确的全序列顺序（所有的 **lists** 之间）

以下图为例，不论先从哪个点开始 DFS，例如 dfs(belt)会得到一个 belt -> jacket 的 list; 但同时因为 pants -> belt，在最终的结果中，包含 pants 的 list 要排在包含 belt 的 list 前面。

TOPOLOGICAL-SORT(G)

1 call DFS(G) to compute finishing times $v.f$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

Figure 22.7(b) shows how the topologically sorted vertices appear in reverse order of their finishing times.

We can perform a topological sort in time $\Theta(V + E)$, since depth-first search takes $\Theta(V + E)$ time and it takes $O(1)$ time to insert each of the $|V|$ vertices onto the front of the linked list.

CLRS 上的算法是，每次 DFS 得到一个符合正确拓扑顺序的 list，保证单序列顺序；每次新的 DFS 之后得到的 list 要排在之前结果的前面，保证全序列顺序。

我的写法是，每次 DFS 得到一个相反顺序的 list； 新的 DFS 返回 list 排在之前结果的后面； 最后把整个 list 翻转，思路类似于 类似于 reverse words in string （正单序列，反全序列），或者 Rotate Array （正单序列，反全序列）里面的多步翻转法。

每次翻转都会改变单序列 & 全序列之间的顺序，对于每一个单序列或者全序列，两次翻转可以互相抵消。

这个思路体现在多步翻转法上就是依次翻转每个单序列，最后全部翻转，因而单序列顺序不变，全序列顺序翻转。

而我这题一开始就是“反单序列”和“反全序列”，因此一次完全翻转之后就可以得到“正单序列”和“正全序列”。

```
/**
 * Definition for Directed graph.
 * class DirectedGraphNode {
 *     int label;
 *     ArrayList<DirectedGraphNode> neighbors;
 *     DirectedGraphNode(int x) { label = x; neighbors = new Arr
ayList<DirectedGraphNode>(); }
 * };
 */
public class Solution {
    /**
     * @param graph: A list of Directed graph node
     * @return: Any topological order for the given graph.
     */
    public ArrayList<DirectedGraphNode> topSort(ArrayList<Direct
edGraphNode> graph) {
        // write your code here
        ArrayList<DirectedGraphNode> list = new ArrayList<Direct
edGraphNode>();
        HashSet<DirectedGraphNode> set = new HashSet<DirectedGra
phNode>();

        for(DirectedGraphNode root : graph){
            if(!set.contains(root)) dfs(list, set, root);
        }
        // 现在的 list 是由若干个顺序倒转的 topological order list 组
成的
        // 这里的处理类似于 reverse words in a string
        // 把每个单词单独翻转之后，再把整个句子翻转
        Collections.reverse(list);

        return list;
    }

    private void dfs(ArrayList<DirectedGraphNode> list,
                     HashSet<DirectedGraphNode> set,
                     DirectedGraphNode root){
```

```
        set.add(root);

        for(DirectedGraphNode node : root.neighbors){
            if(!set.contains(node)) dfs(list, set, node);
        }
        // 到这一步，root 节点的所有 sub path 都已经被访问过了
        // 最后才在 list 中添加元素，得到的是一个反序的 topological order
        list.add(root);
    }
}
```

做了欧拉回路之后，发现一个更好的做法是直接建 **LinkedList**，然后用 **addFirst();**

熟练使用 **API** 很重要啊**~**

```
public ArrayList<DirectedGraphNode> topSort(ArrayList<DirectedGr
aphNode> graph) {
        // write your code here
        LinkedList<DirectedGraphNode> list = new LinkedList<Dire
ctedGraphNode>();
        HashSet<DirectedGraphNode> visited = new HashSet<Directe
dGraphNode>();

        for(DirectedGraphNode root : graph){
            dfs(list, root, visited);
        }


        return new ArrayList<DirectedGraphNode>(list);
    }

    private void dfs(LinkedList<DirectedGraphNode> list, Directe
dGraphNode root, HashSet<DirectedGraphNode> visited){

        if(visited.contains(root)) return;

        for(DirectedGraphNode next : root.neighbors){
            dfs(list, next, visited);
        }

        list.addFirst(root);
        visited.add(root);
    }
```

拓扑排序， BFS 做法

无论是 directed 还是 undirected Graph，其 BFS 的核心都在于 "indegree"，处理顺序也多是从 indegree 最小的开始，从外向内。

我们先用一个 HashMap 统计下所有节点的 indegree; 值越高的，在拓扑排序中位置也就越靠后，因为还有 N = current indegree 的 parent node 们没有处理完。

因此在循环最开始，我们可以把所有 indegree = 0 (也即不在 hashmap 中) 的节点加到 list 中，也作为 BFS 的起点。

在 BFS 过程中，我们依次取出队列里取出节点 node 的 neighbors， next，并且对 next 对应的 indegree -1，代表其 parent node 已经被处理完，有效 indegree 减少。

当-1之后如果 next 的 indegree 已经是 0 ， 则可以加入 list，并且作为之后 BFS 中新的起始节点。

从 Course Schedule 的 BFS 做法学到的经验是，如果图中有环，会有环上的 node 永远无法减到 indegree = 0 的情况，导致一些 node 没有被访问。

这也是当初面试被问到的为什么 JVM 不只靠 reference counting 来做 GC，就是因为有环的时候，indegree 不会减到 0.

Topological Sort

```
public class Solution {
    /**
     * @param graph: A list of Directed graph node
     * @return: Any topological order for the given graph.
     */
```

```
    public ArrayList<DirectedGraphNode> topSort(ArrayList<Direct
edGraphNode> graph) {
        // write your code here
        ArrayList<DirectedGraphNode> list = new ArrayList<Direct
edGraphNode>();

        // Key : node
        // Value : current in-degree count
        HashMap<DirectedGraphNode, Integer> map = new HashMap<>(
);
        for(DirectedGraphNode node : graph){
            for(DirectedGraphNode neighbor : node.neighbors){
                if(!map.containsKey(neighbor)){
                    map.put(neighbor, 1);
                } else {
                    map.put(neighbor, map.get(neighbor) + 1);
                }
            }
        }

        // Now we know each node's in-degree (out is trivial)
        Queue<DirectedGraphNode> queue = new LinkedList<>();

        Iterator<DirectedGraphNode> iter = graph.iterator();
        while(iter.hasNext()){
            DirectedGraphNode node = iter.next();
            // get all nodes without indegree
            if(!map.containsKey(node)){
                queue.offer(node);
                list.add(node);
            }
        }

        while(!queue.isEmpty()){
            DirectedGraphNode node = queue.poll();
            for(DirectedGraphNode next : node.neighbors){
                // node "next"'s parent has been processed
                map.put(next, map.get(next) - 1);
                if(map.get(next) == 0){
                    list.add(next);
```

```
                    queue.offer(next);
                }
            }
        }

        return list;
    }
}
```

Course Schedule 类

- 建 **ArrayList[]** 不能用泛型，**list.get(i)** 默认返回 **Object**，需要 **cast** 一下。

Course Schedule

这题的本质就是，给你一个代表 **graph** 的 **adjacency array**，判断 **graph** 是否有环。其实和 **Graph Valid Tree** 非常像。

DFS 找环性能优异，击败了 **98.42 %** 的提交~

```
public class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisit
es) {
        ArrayList[] graph = new ArrayList[numCourses];
        int[] visited = new int[numCourses];

        for(int i = 0; i < numCourses; i++){
            graph[i] = new ArrayList<Integer>();
        }

        for(int[] num : prerequisites){
            int parent = num[1];
            int child = num[0];
            graph[parent].add(child);
        }

        for(int i = 0; i < numCourses; i++){
            if(visited[i] == 0 && hasCycle(i, visited, graph)) r
eturn false;
        }
```

```
        return true;
    }

    private boolean hasCycle(int cur, int[] visited, ArrayList[]
 graph){
        visited[cur] = 1;

        boolean hasCycle = false;

        for(int i = 0; i < graph[cur].size(); i++){
            int next = (int) graph[cur].get(i);
            if(visited[next] == 1) return true;
            else if(visited[next] == 0){
                hasCycle = hasCycle || hasCycle(next, visited, g
raph);
            }
        }

        visited[cur] = 2;

        return hasCycle;
    }
}
```

BFS 写法，速度超过 82.34%

思路上承接了原来的 **topological sort BFS** 解法，建 **array** 保存所有节点的 **indegree**，同时有数据结构存储每个节点的 **children.**

由于在 **BFS** 时只有 **indegree = 0** 时才会被加入队列，如果 **graph** 中有环，会出现有环的部分永远无法进入 **BFS** 被访问的情况，因此在结尾我们只需要看一下到底有没有点从来没被访问过即可。

这种情况的问题和当初面试时候问为什么 **Java GC** 里不只依靠 **refernce counting** 做的问题是一样的，就是当某个局部出现环形时，无法通过 **BFS** 建 **reference count** 的方式使所有点的当

前 **count = 0**，因为所有点的 **indegree** 都非 **0**，无从开始。

```
public class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisit
es) {
        ArrayList[] graph = new ArrayList[numCourses];
        int[] indegree = new int[numCourses];

        for(int i = 0; i < numCourses; i++){
            graph[i] = new ArrayList<Integer>();
        }

        for(int[] num : prerequisites){
            int parent = num[1];
            int child = num[0];
            graph[parent].add(child);
            indegree[child] ++;
        }

        Queue<Integer> queue = new LinkedList<>();

        for(int i = 0; i < numCourses; i++){
            if(indegree[i] == 0) queue.offer(i);
        }

        int visitedCount = 0;

        while(!queue.isEmpty()){
            int cur = queue.poll();
            visitedCount ++;

            for(int i = 0; i < graph[cur].size(); i++){
                int next = (int) graph[cur].get(i);
                indegree[next] --;
                if(indegree[next] == 0){
                    queue.offer(next);
                }
            }
        }
```

```
        return (visitedCount == numCourses);
    }
}
```

Course Schedule II

超过 80.69%，速度尚可~

思路和上一题完全一样，只不过留了个 index 用于记录拓扑顺序。

```
public class Solution {
    public int[] findOrder(int numCourses, int[][] prerequisites
) {
        int[] rst = new int[numCourses];
        int[] indegree = new int[numCourses];
        ArrayList[] graph = new ArrayList[numCourses];

        for(int i = 0; i < numCourses; i++){
            graph[i] = new ArrayList();
        }

        for(int[] edge : prerequisites){
            int parent = edge[1];
            int child = edge[0];
            graph[parent].add(child);
            indegree[child] ++;
        }

        Queue<Integer> queue = new LinkedList<Integer>();

        for(int i = 0; i < numCourses; i++){
            if(indegree[i] == 0) queue.offer(i);
        }

        int index = 0;
        int visitedCount = 0;
        while(!queue.isEmpty()){
            int cur = queue.poll();
            rst[index ++] = cur;
```

```
            visitedCount ++;

            for(int i = 0; i < graph[cur].size(); i++){
                int next = (int) graph[cur].get(i);
                indegree[next] --;
                if(indegree[next] == 0){
                    queue.offer(next);
                }
            }
        }

        return (visitedCount == numCourses) ? rst : new int[0];
    }
}
```

Alien Dictionary

Alien Dictionary

先说一个自己写这题时犯的错误，就是建 class 的时候没直接建 Node ，而是建了个 Edge，再用 Edge 去建 ArrayList[] 代表 Graph，太麻烦了，只是在故意贴近之前写题的模板而已，因为那几题给的都是 edges.

要自己从头建 Graph ，就应该直接从 Node 写起，记 state, children, indegree 都方便。

题意解析：

> - 同一个单词中的字母先后顺序没有任何意义；
> - 相邻单词之间，按照 Lexicographical order 排列的意义是，双方字符串中第一个不 match 的字符代表这一条 directed edge，即字符的先后顺序；
> - 如果所有字符一致，但是其中一个单词长一些，也无法学习到 edge，但是多出来的那个字符要记得加入字典，暂时作为一个独立 node;
> - 如果发现字典 Graph 中有环，返回空字符串；
> - 如果字典中只有一个单词，返回此单词的任意顺序皆可；
> - 加入新 edge 的时候，记得判重，不要加入重复 child 节点；

超过 69% 的 DFS 写法：

```
public class Solution {
    private class Node{
        int chrInt;
        int state;
        int indegree;
        ArrayList<Integer> children;
        public Node(int chrInt){
            this.chrInt = chrInt;
            state = 0;
            indegree = 0;
            children = new ArrayList<Integer>();
```

```
        }
        public char getChar(){
            return (char) (this.chrInt + 'a');
        }
    }

    public String alienOrder(String[] words) {
        HashMap<Integer, Node> map = new HashMap<>();
        StringBuilder sb = new StringBuilder();

        if(words.length == 1) return words[0];

        for(int i = 0; i < words.length - 1; i++){
            learn(words[i], words[i + 1], map);
        }

        for(int i : map.keySet()){
            if(map.get(i).state == 0 && dfs(i, sb, map)) return
"";
        }

        return sb.reverse().toString();
    }

    private boolean dfs(int cur, StringBuilder sb, HashMap<Integ
er, Node> map) {

        Node node = map.get(cur);

        node.state = 1;
        boolean hasCycle = false;

        for(int i = 0; i < node.children.size(); i++){
            int next = node.children.get(i);
            if(map.get(next).state == 1) hasCycle = true;
            else if (map.get(next).state == 0) {
                hasCycle = hasCycle || dfs(next, sb, map);
            }
        }
```

```
        node.state = 2;
        sb.append(node.getChar());

        return hasCycle;
    }

    private void learn(String word1, String word2, HashMap<Integ
er, Node> map){
        int ptr1 = 0;
        int ptr2 = 0;

        while(ptr1 < word1.length() && ptr2 < word2.length()){
            int parent = word1.charAt(ptr1++) - 'a';
            int child = word2.charAt(ptr2++) - 'a';

            if(!map.containsKey(parent)) map.put(parent, new Nod
e(parent));
            if(!map.containsKey(child)) map.put(child, new Node(
child));

            if(parent != child){
                Node p = map.get(parent);
                Node c = map.get(child);

                if(!p.children.contains(child)){
                    p.children.add(child);
                    c.indegree ++;
                }
                break;
            }
        }
        while(ptr1 < word1.length()){
            int node = word1.charAt(ptr1++) - 'a';
            if(!map.containsKey(node)) map.put(node, new Node(no
de));
        }
        while(ptr2 < word2.length()){
            int node = word2.charAt(ptr2++) - 'a';
            if(!map.containsKey(node)) map.put(node, new Node(no
de));
```

```
        }
    }
}
```

同样的结构，**BFS** 写法：

```
public class Solution {
    private class Node{
        int chrInt;
        int state;
        int indegree;
        ArrayList<Integer> children;
        public Node(int chrInt){
            this.chrInt = chrInt;
            state = 0;
            indegree = 0;
            children = new ArrayList<Integer>();
        }
        public char getChar(){
            return (char) (this.chrInt + 'a');
        }
    }

    public String alienOrder(String[] words) {
        HashMap<Integer, Node> map = new HashMap<>();
        StringBuilder sb = new StringBuilder();

        if(words.length == 1) return words[0];

        for(int i = 0; i < words.length - 1; i++){
            learn(words[i], words[i + 1], map);
        }

        Queue<Node> queue = new LinkedList<>();
        int total = 0;
        for(int i : map.keySet()){
            total++;
            if(map.get(i).indegree == 0) queue.offer(map.get(i))
;
```

```
        }

        int count = 0;
        while(!queue.isEmpty()){
            Node cur = queue.poll();
            sb.append(cur.getChar());
            count ++;

            for(int next : cur.children){
                Node child = map.get(next);
                child.indegree--;
                if(child.indegree == 0) queue.offer(child);
            }
        }

        return (count == total) ? sb.toString() : "" ;
    }


    private void learn(String word1, String word2, HashMap<Integ
er, Node> map){
        int ptr1 = 0;
        int ptr2 = 0;

        while(ptr1 < word1.length() && ptr2 < word2.length()){
            int parent = word1.charAt(ptr1++) - 'a';
            int child = word2.charAt(ptr2++) - 'a';

            if(!map.containsKey(parent)) map.put(parent, new Nod
e(parent));
            if(!map.containsKey(child)) map.put(child, new Node(
child));

            if(parent != child){
                Node p = map.get(parent);
                Node c = map.get(child);

                if(!p.children.contains(child)){
                    p.children.add(child);
                    c.indegree ++;
```

```
                }
                break;
            }
        }
        while(ptr1 < word1.length()){
            int node = word1.charAt(ptr1++) - 'a';
            if(!map.containsKey(node)) map.put(node, new Node(no
de));
        }
        while(ptr2 < word2.length()){
            int node = word2.charAt(ptr2++) - 'a';
            if(!map.containsKey(node)) map.put(node, new Node(no
de));
        }
    }
}
```

不建 **Node class** 而用 **HashMap** 的做法如下：

自己在写这题白板的时候犯了一个小错误，就是一旦发现了 **char1 != char2** ，处理完了要记得 **break**，不要乱学新的 **edge** 出来。

```
    public String alienOrder(String[] words) {
        HashMap<Character, Set<Character>> graphMap = new HashMa
p<>();
        HashMap<Character, Integer> indegreeMap = new HashMap<>(
);

        for(String str : words){
            for(int i = 0; i < str.length(); i++){
                indegreeMap.put(str.charAt(i), 0);
                graphMap.put(str.charAt(i), new HashSet<>());
            }
        }

        for(int i = 0; i < words.length - 1; i++){
            String word1 = words[i];
            String word2 = words[i + 1];
```

```
            for(int index = 0; index < Math.min(word1.length(),
                                                word2.length());
 index++){
                char char1 = word1.charAt(index);
                char char2 = word2.charAt(index);
                if(char1 != char2){
                    Set<Character> neighbours = graphMap.get(cha
r1);
                    if(!neighbours.contains(char2)){
                        int indegree = indegreeMap.get(char2);
                        indegreeMap.put(char2, indegree + 1);
                    }
                    graphMap.get(char1).add(char2);
                    break;
                }
            }
        }

        StringBuilder sb = new StringBuilder();
        Queue<Character> queue = new LinkedList<>();
        for(Character chr : indegreeMap.keySet()){
            if(indegreeMap.get(chr) == 0) queue.offer(chr);
        }
        int count = 0;
        while(!queue.isEmpty()){
            char curChar = queue.poll();
            count ++;
            sb.append(curChar);

            for(Character child : graphMap.get(curChar)){
                int degree = indegreeMap.get(child);
                degree --;
                indegreeMap.put(child, degree);
                if(degree == 0) queue.offer(child);
            }
        }

        return (count == graphMap.keySet().size()) ? sb.toString
(): "";
```

```
    }
```

Undirected Graph, BFS

---

directed graph BFS 里靠 indegree = 0 判断加入队列；

undirected graph BFS 里靠 degree = 0 判断加入队列；

Minimum Height Trees

论坛里看到的，非常赞的思路。
https://discuss.leetcode.com/topic/30572/share-some-thoughts

先说下自己一个 TLE 的初始尝试：根据所有 Edges 建 ArrayList[] 的 Graph，同时存每个点的 Adjacency lists，考虑到是无向图，对于每一个 edge 我们需要在两个 list 中更新才行。然后维护一个 HashMap<> 存着每一个 height 对应的 List<>，然后循环扫描每一个点作为起点进行 BFS，找到最小 height 之后 map.get(height) 就可以了。

然而这种做法的时间复杂度是 O(V * (E + V)) ，每个点都要扫整个 graph ，太高了。

转换一下思路，我们做的事情其实非常近似于 **Find Leaves of Binary Tree** , 对于一个形状和链表一样的图，我们知道最优解一定是中点；对于一个 **inorder traversal array**，我们知道 **height balanced tree** 也一定以中点为 **root;**

- 在这个 **graph** 里，我们要找的，其实也是一层一层剥开最外围 **nodes** 之后，最里面的点。

因此，即使是 **undirected graph**，**degree** 也是非常重要的信息，只不过对于 **undirected graph** 来讲:

- **degree = "in"-degree + "out"-degree** 而已.

AC代码，速度击败了 95.98 % ~

O(E + V)

```
public class Solution {
    public List<Integer> findMinHeightTrees(int n, int[][] edges)
 {
        int[] degrees = new int[n];
        ArrayList[] graph = new ArrayList[n];

        for(int i = 0; i < n; i++){
            graph[i] = new ArrayList<>();
        }

        for(int[] edge : edges){
            graph[edge[0]].add(edge[1]);
            graph[edge[1]].add(edge[0]);
            degrees[edge[0]]++;
            degrees[edge[1]]++;
        }

        Queue<Integer> queue = new LinkedList<Integer>();
        for(int i = 0; i < n; i++){
            if(degrees[i] <= 1) queue.offer(i);
        }

        List<Integer> list = new ArrayList<>();
        while(!queue.isEmpty()){
            int size = queue.size();
            list = new ArrayList<>();
            for(int i = 0; i < size; i++){
                int node = queue.poll();
                list.add(node);
                for(int j = 0; j < graph[node].size(); j++){
```

```
                    int next = (int)graph[node].get(j);
                    degrees[next] --;
                    if(degrees[next] == 1) queue.offer(next);
                }
            }
        }

        return list;
    }
}
```

Graph Valid Tree

这题我之前写 union-find 的时候做过了，确实是更快更高效的写法。不过这题也完全可以用 Graph 上的常规 BFS / DFS 求解.

一个非常精炼的 BFS, DFS, Union-Find 做法总结帖子

这题用 BFS 解需要解决这么几个问题：

- 如何正确 **detect cycle?**
  - 常规 **indegree** 的话，如果有 **cycle** 会有一些点因为 **indegree** 无法进一步缩小的问题永远不被访问，可以记录 **visited count;**
  - 另一种方法是，用 **int[]** 表示每个点的状态，其中
    - **0** 代表“未访问”；
    - **1** 代表“访问中”；
    - **2** 代表“已访问”；

  - 如果在循环的任何时刻，我们试图访问一个状态为 **“1”** 的节点，都可以说明图中有环。

- 如何正确识别图中 **connected components** 的数量**?**
  - 添加任意点，探索所有能到达的点，探索完毕数量 **+1**；
  - 如此往复，直到已探索点的数量 **= # of V in graph** 为止。

BFS 做法，时间复杂度 **O(E + V);**

```java
public class Solution {
    public boolean validTree(int n, int[][] edges) {
        int[] states = new int[n];
        ArrayList[] graph = new ArrayList[n];

        for(int i = 0; i < n; i++){
            graph[i] = new ArrayList();
        }

        for(int[] edge : edges){
            graph[edge[0]].add(edge[1]);
            graph[edge[1]].add(edge[0]);
        }

        Queue<Integer> queue = new LinkedList<>();

        queue.offer(0);
        states[0] = 1;
        int count = 0;

        while(!queue.isEmpty()){
            int node = queue.poll();
            count ++;
            for(int i = 0; i < graph[node].size(); i++){
                int next = (int) graph[node].get(i);

                if(states[next] == 1) return false ;
                else if(states[next] == 0){
                    states[next] = 1;
                    queue.offer(next);
                }
            }
            states[node] = 2;
        }

        return count == n;
    }
}
```

Clone Graph

一次 AC ~ 由于不用考虑环和拓扑顺序，BFS 的方式就很简单，记录下看过哪些点就可以了。

参考了下论坛讨论之后，不太同意大多数解法，因为假设所有点的 label 唯一不是很适合 generalize 这个算法。用 HashMap<> 实现 Node - Node mapping 比较靠谱。

下面的代码简化了一下，参考了九章的写法，先用一个 getNodes 函数返回所有的节点(因为函数就给了一个)，在返回的时候直接 new ArrayList<>(set) 调用 Collection constructor.

这样算法逻辑就分成两步：

- 过一遍所有节点，建新节点放到 HashMap 中；
- 再过一遍所有节点，这次更新每个对应节点的 neighbors.

```
public class Solution {
    public UndirectedGraphNode cloneGraph(UndirectedGraphNode no
de) {
        if(node == null) return node;

        HashMap<UndirectedGraphNode, UndirectedGraphNode> map =
new HashMap<>();

        List<UndirectedGraphNode> nodeList = getNodes(node);

        for(UndirectedGraphNode cur : nodeList){
            map.put(cur, new UndirectedGraphNode(cur.label));
        }

        for(UndirectedGraphNode cur : nodeList){
            UndirectedGraphNode copy = map.get(cur);

            for(UndirectedGraphNode neighbor : cur.neighbors){
                copy.neighbors.add(map.get(neighbor));
            }
        }
```

```
        return map.get(node);
    }

    private List<UndirectedGraphNode> getNodes(UndirectedGraphNo
de node){
        Set<UndirectedGraphNode> visited = new HashSet<>();
        Queue<UndirectedGraphNode> queue = new LinkedList<>();

        queue.offer(node);
        visited.add(node);

        while(!queue.isEmpty()){
            UndirectedGraphNode cur = queue.poll();
            for(UndirectedGraphNode next : cur.neighbors){
                if(!visited.contains(next)){
                    visited.add(next);
                    queue.offer(next);
                }
            }
        }

        return new ArrayList<>(visited);
    }
}
```

Copy List with Random Pointer

链表也是图啊，就是这么简单，轻松，愉快~~

这题还有一个比较妖孽的 follow-up，不让用 HashMap. 做法就是在每个节点后面插入新节点，最后再拆出来就行了，复杂度依然 O(n).

```
public class Solution {
    public RandomListNode copyRandomList(RandomListNode head) {
        HashMap<RandomListNode, RandomListNode> map = new HashMa
p<>();

        RandomListNode cur;

        for(cur = head; cur != null; cur = cur.next){
            map.put(cur, new RandomListNode(cur.label));
        }

        for(cur = head; cur != null; cur = cur.next){
            RandomListNode copy = map.get(cur);

            copy.random = map.get(cur.random);
            copy.next = map.get(cur.next);
        }

        return map.get(head);
    }
}
```

Undirected Graph, DFS

Graph Valid Tree

无向图的 **DFS** 要注意避免“原路返回”的情况，仅仅依靠设 **state = 1** 是不行的，所以 **dfs** 里最好有个参数，代表“前一个节点”，这样在下一步的搜索中可以直接跳过，又避免了误判有环。

```
public class Solution {
    public boolean validTree(int n, int[][] edges) {
        int[] states = new int[n];
        ArrayList[] graph = new ArrayList[n];

        for(int i = 0; i < n; i++){
            graph[i] = new ArrayList();
        }

        for(int[] edge : edges){
            graph[edge[0]].add(edge[1]);
            graph[edge[1]].add(edge[0]);
        }

        if(hasCycle(-1, 0, states, graph)) return false;

        for(int state : states){
            if(state == 0) return false;
        }

        return true;
    }

    private boolean hasCycle(int prev, int cur, int[] states, Ar
rayList[] graph){
        states[cur] = 1;
```

```
        boolean hasCycle = false;

        for(int i = 0; i < graph[cur].size(); i++){
            int next = (int) graph[cur].get(i);
            if(next != prev){
                if(states[next] == 1) return true;
                else if(states[next] == 0){
                    hasCycle = hasCycle || hasCycle(cur, next, s
tates, graph);
                }
            }

        }

        states[cur] = 2;
        return hasCycle;
    }
}
```

矩阵，BFS 最短距离探索

Walls and Gates

这题乍看之下特别像滑雪，于是想试试用 DFS + memoization，于是有了下面的代码，过了 25/62 个 test cases ，错在了一个

- [0 , INF, 0, INF, INF]
- [-1, -1 , 0 , -1 , INF]
- [-1 , 0 , 0 , 0 , INF]
- [-1 , INF , 0 , 0 , -1]

这样的 test case 上。

其原因是，当我们搜索 (0, 4) 这个位置时，dfs 搜索了相邻的 (0, 5)，这时候遇到了比较尴尬的境地:

- 为了避免死循环，我们要把当前元素改一下，免得下一步的 dfs 重新探回来；
- 但是在这个局部上，正解却偏偏要探回来，往左走才能碰到最近的 0.

- 于是我们发现了这题和滑雪最大的不同：滑雪由于取值限制，是不用担心回头路的，但这题会。留回头路会死循环，不留回头路会算错。

于是下面的代码在如上的 test case 上会挂掉；

```
public class Solution {
    public void wallsAndGates(int[][] rooms) {
        for(int i = 0; i < rooms.length; i++){
            for(int j = 0; j < rooms[0].length; j++){
                if(rooms[i][j] == Integer.MAX_VALUE) rooms[i][j]
 = memoizedSearch(rooms, i, j);
            }
        }
    }
```

```
    private int memoizedSearch(int[][] rooms, int x, int y){
        if(x < 0 || x >= rooms.length) return Integer.MAX_VALUE;
        if(y < 0 || y >= rooms[0].length) return Integer.MAX_VAL
UE;

        if(rooms[x][y] == 0) return 0;
        if(rooms[x][y] == -1) return Integer.MAX_VALUE;
        if(rooms[x][y] == Integer.MAX_VALUE){

            rooms[x][y] = -1;

            int N = memoizedSearch(rooms, x - 1, y);
            int S = memoizedSearch(rooms, x + 1, y);
            int W = memoizedSearch(rooms, x, y - 1);
            int E = memoizedSearch(rooms, x, y + 1);

            int min = Math.min(Math.min(N, S), Math.min(W, E));
            if(min == Integer.MAX_VALUE){
                rooms[x][y] = Integer.MAX_VALUE;
                return rooms[x][y];
            } else {
                rooms[x][y] = min + 1;
                return rooms[x][y];
            }

        } else {
            return rooms[x][y];
        }
    }
}
```

“和最短距离”相关的问题，最靠谱的还是BFS 解法。

一个最容易想到的做法是，每个 INF 为起点，扫整个矩阵。每个起点走的最长距离是 O(mn)，总共有 O(mn) 个能的起点，所以复杂度为 O(m^2 * n^2)，太慢了。

改良版本是，我们不从 INF 的角度开始，而是反过来，从 0 开始往回找，沿路只添加看到的 INF; 由于是 BFS，从gate出发，第一次看到某个房间的时候就是最短距离。这样我们有 O(mn) 个可能的起点和位置要搜，然而每个位置只会进入队列一次，所以时间和空间复杂度都是 O(mn).

每个位置只看一次，并不一定要靠记忆化搜索，**DFS / BFS flood filling** 一样可以。事实上，大多数都是靠 **flood filling**，靠记忆化搜索的反而是少数。

另一个小技巧是，处理矩阵不一定非要拍成一维 **index**，队列里直接扔 **int[]** 也完全可以，还省事。

```
public class Solution {
    public void wallsAndGates(int[][] rooms) {
        if(rooms == null || rooms.length == 0) return;

        int rows = rooms.length;
        int cols = rooms[0].length;

        Queue<int[]> queue = new LinkedList<>();

        for(int i = 0; i < rows; i++){
            for(int j = 0; j < cols; j++){
                if(rooms[i][j] == 0) queue.offer(new int[]{i, j}
);
            }
        }

        int distance = 1;

        while(!queue.isEmpty()){
            int size = queue.size();
            for(int i = 0; i < size; i++){
                int[] cur = queue.poll();

                int[] xDir = {0,0,-1,1};
                int[] yDir = {-1,1,0,0};

                for(int j = 0; j < 4; j++){
```

```
                    int[] next = {cur[0] + xDir[j], cur[1] + yDi
r[j]};

                    if(next[0] >= 0 && next[1] >= 0 && next[0] <
 rows && next[1] < cols){
                        if(rooms[next[0]][next[1]] == Integer.MA
X_VALUE){
                            rooms[next[0]][next[1]] = distance;
                            queue.offer(next);
                        }
                    }
                }
            }
            distance ++;
        }
    }
}
```

Best Meeting Point

先说一下自己第一个不成功的尝试，那就是一维推导错了，以为是所有 X 坐标的平均值。

在一维情况下：

> - 只有一个点
> - 取当前点的位置；
> - 有两个点
> - 两点之间的任何位置都是等价的，等于区间大小；
> - 有三个点
> - 最外面两点之间的任意位置等价，距离和等于区间大小；
> - 于是剩下一个点，可以发现最优位置就是在点上；
> - 有 **N** 个点
> - 我们可以对于排序了的坐标从外向里配对消除，每一对的距离等于这对点形成的区间大小。

于是我我们可以总结为：

- 点的个数为偶数，则最里面 **pair** 之间的任意位置等价；
- 点的个数为奇数，则一定在最里面的中心点上；
- 因此，最优点就是当前维度的中位数。

于是一个最直观的解法就是，记录所有的 "1" 的 X/Y 坐标，选中位数，再把所有的地方过一遍。

扫哪里是 **1 +** 用两次 **quick select +** 计算总的距离

O(mn) + O(m) + O(n) + O(m) + O(n) = O(mn)

不过从提交速度上看，这个解法速度不理想，而且写个 **quick select** 代码量就上去了，面试时候有点浪费时间。

```
public class Solution {
    public int minTotalDistance(int[][] grid) {
        if(grid == null || grid.length == 0) return 0;
        List<Integer> listX = new ArrayList<>();
        List<Integer> listY = new ArrayList<>();

        for(int i = 0; i < grid.length; i++){
            for(int j = 0; j < grid[0].length; j++){
                if(grid[i][j] == 1){
                    listX.add(i);
                    listY.add(j);
                }
            }
        }

        int medianX = findMedian(listX, 0, listX.size() - 1);
        int medianY = findMedian(listY, 0, listY.size() - 1);
        int minDis = 0;

        for(int x : listX){
            minDis += Math.abs(medianX - x);
        }
        for(int y : listY){
            minDis += Math.abs(medianY - y);
        }

        return minDis;
    }

    private int findMedian(List<Integer> list, int start, int en
d){
        int pivot = list.get(start);
        int left = start;
        int right = end;

        while(left <= right){
            while(left <= right && list.get(left) <= pivot) left
```

```
++;
            while(left <= right && list.get(right) >= pivot) rig
ht--;

            if(left < right) swap(list, left, right);
        }
        swap(list, start, right);

        int medianIndex = list.size() / 2;

        if(right == medianIndex){
            return list.get(right);
        } else if(right < medianIndex){
            return findMedian(list, right + 1, end);
        } else {
            return findMedian(list, start, right - 1);
        }
    }

    private void swap(List<Integer> list, int a, int b){
        int temp = list.get(a);
        list.set(a, list.get(b));
        list.set(b, temp);
    }
}
```

论坛上还有一个很有趣的解法，虽然时间复杂度不好看，但是思路巧妙，简洁易懂。

核心思想在于，最优 **meeting point** 一定是在所有点的中间，而对于每个 **pair** ，他们到 **best meeting point** 的距离和就是 **pair** 两点的区间长度，所以假设以 **0** 为远点，用双指针检查每一对就可以了。

```java
public int minTotalDistance(int[][] grid) {
    int m = grid.length;
    int n = grid[0].length;

    List<Integer> I = new ArrayList<>(m);
    List<Integer> J = new ArrayList<>(n);

    for(int i = 0; i < m; i++){
        for(int j = 0; j < n; j++){
            if(grid[i][j] == 1){
                I.add(i);
                J.add(j);
            }
        }
    }

    return getMin(I) + getMin(J);
}

private int getMin(List<Integer> list){
    int ret = 0;

    Collections.sort(list);

    int i = 0;
    int j = list.size() - 1;
    while(i < j){
        ret += list.get(j--) - list.get(i++);
    }

    return ret;
}
```

如果进一步利用循环的特点，甚至排序都可以省了，**ArrayList** 里自动就是排好序的。不过会扫两次 **O(mn)**，就有一个取舍的问题在里面了。

```java
public class Solution {
    public int minTotalDistance(int[][] grid) {
        int m = grid.length, n = grid[0].length;

        List<Integer> I = new ArrayList<Integer>();
        List<Integer> J = new ArrayList<Integer>();

        for(int i = 0; i < m; i++) {
            for(int j = 0; j < n; j++) {
                if(grid[i][j] == 1) {
                    I.add(i);
                }
            }
        }
        for(int j = 0; j < n; j++) {
            for(int i = 0; i < m; i ++) {
                if(grid[i][j] == 1) {
                    J.add(j);
                }
            }
        }
        return minTotalDistance(I) + minTotalDistance(J);
    }

    public int minTotalDistance(List<Integer> grid) {
        int i = 0, j = grid.size() - 1, sum = 0;

        while(i < j) {
            sum += grid.get(j--) - grid.get(i++);
        }
        return sum;
    }

}
```

Shortest Distance from All Buildings

乍看之下是上一题的强化版，因为所谓“距离”的定义是一样的，都是曼哈顿距离。最主要的不同点在于，中间有障碍之后，问题就不再是简单的分开维度拆分分析了，因为对于每一个位置，都要考虑障碍物的具体位置。

因此不难看出，这题变成了一个 Graph 问题，因为对于矩阵中任意两点 A & B，其最短距离只有靠 BFS 去查。 矩阵就是一种 uniform weight, undirected graph.

一个非常土的办法是，先扫一遍矩阵，找到 k 个起始点；对于每一个起始点开始做 BFS，记录到每一个其他格子的距离，每次扫完了要存这个位置到起点的距离，并且每次扫完一个点要恢复矩阵的访问状态，免得下一次扫描出错，扫到第 k 个起点的时候，把每个 cell 里面的相对于其他店的距离加起来，保持一个 min 就可以了。

时间复杂度： 2k * O(mn) ，考虑到 k 可能等于整个矩阵，其实是 O(m^2 * n^2)

空间复杂度： k * O(mn)，考虑到 k 可能等于整个矩阵，其实时 O(m^2 * n^2)

我觉得这个思路可以不用写代码了，肯定要超时的。。

这个时间复杂度和暴力解 Walls and Gates 一样，不难想到，可以试着从每个 "1" 出发，去反过来探索矩阵。 假如我们记录了这 k 个起始点，我们可以依次 bfs 每一个起始点 / 该起始点的范围，一次一个点，一次扩张一轮，当 k 个点的 “势力范围” 第一次相交的时候，就是我们要找的最优起点。有了最优起点，O(mn) 的时间显然就可以算出最短距离。

- 问题一 **:** 如何保证对于固定起点的 **BFS** 过程中，不走回头路，而且相邻边界不重复添加；
- 问题二：如何解决多个起点 **BFS** 之间的覆盖问题？
- 问题三：对于某一个被若干个队列中边界点包围的位置，我们怎么判断每次试图访问的时候，是同一个起点的重复访问（此时跳过），还是另一个起点的回合点（此时保留）？

可以看到，在 Walls and Gates 中，这两个问题是不存在的，因为当一个位置被某个 BFS 发现之后，最短距离就确定了，后面的所有 BFS 也都不会再去访问了，一口气解决了 “相邻边界重复添加” 和 “不同起点覆盖搜索” 两个问题。

经过了半天的思考之后，依然没能找到接近 O(mn) 的解法，于是我就跑去 LC 论坛看答案了。你妹啊，你们用的也是我一开始的挫算法。。。。所以如果真是面试头一次碰到这种题，不要犹豫，就先撸个最暴力的办法再说。

痛定思痛，一次撸完AC，速度超过 89.66%

- 开个 **int[][]** 记录每个位置相对于每个点的最短距离，每一个 **(i , j)** 的值只会在一次探索中被更新一次，取值叠加；
- 用负数作为 **flag** ，代表探索进度。有两个明显的好处：
  - 剪枝，扫第 **5** 个起点的时候，对于 **flag = -2** 的位置，连扫的必要都没有；
  - 去重，每个起点的 **BFS** 都需要机制来避免重复添加，而 **flag** 是天然的状态指示，如果当前位置的 **flag** 和当前 **BFS** 的有效 **flag** 不相等，则直接跳过，起到了一个 **undirected graph bfs** 中的 **0, 1, 2** 状态功能~
- 每个起点的 **BFS** 都会更新其所有能碰到的 **“有效点”** 的最短距离，一个点在一次迭代中只会被访问一次。
- 因此最后只要扫一下 **flag** 矩阵，**flag == curMark** 的就是合理位置，找其对应的 **disMap[][]** 值，记录最小就可以了。
- 另一个优化是，可以在每一轮的 **BFS** 过程中检查当前最短距离，省去最后收尾扫描。不过对用时影响不大就是了，下面的代码没写那部分。

```
public int shortestDistance(int[][] grid) {
    if(grid == null || grid.length == 0) return 0;

    int rows = grid.length;
    int cols = grid[0].length;

    int[][] disMap = new int[rows][cols];
    int minDis = Integer.MAX_VALUE;

    int curMark = 0;

    for(int i = 0; i < rows; i++){
        for(int j = 0; j < cols; j++){
            if(grid[i][j] == 1){
                bfs(grid, disMap, i, j, curMark);
                curMark --;
```

```
                }
            }
        }

        for(int i = 0; i < rows; i++){
            for(int j = 0; j < cols; j++){
                if(grid[i][j] == curMark) minDis = Math.min(minD
is, disMap[i][j]);
            }
        }


        return minDis == Integer.MAX_VALUE ? -1 : minDis;
    }

    private void bfs(int[][] grid, int[][] disMap, int x, int y,
int curMark){
        int rows = grid.length;
        int cols = grid[0].length;
        int distance = 1;

        Queue<int[]> queue = new LinkedList<>();
        // Just found out using curMark can save the use of visi
ted
        queue.offer(new int[]{x, y});

        while(!queue.isEmpty()){
            int size = queue.size();
            for(int i = 0; i < size; i++){
                int[] cur = queue.poll();
                int curX = cur[0];
                int curY = cur[1];

                int[] xDirs = {0,0,1,-1};
                int[] yDirs = {-1,1,0,0};

                for(int j = 0; j < 4; j++){
                    int nextX = curX + xDirs[j];
                    int nextY = curY + yDirs[j];
```

```
                    if(nextX >= 0 && nextX < rows &&
                        nextY >= 0 && nextY < cols &&
                        grid[nextX][nextY] == curMark){

                        grid[nextX][nextY] = curMark - 1;
                        disMap[nextX][nextY] += distance;
                        queue.offer(new int[]{nextX, nextY});
                    }
                }
            }
            distance ++;
        }
    }
```

欧拉回路，Hierholzer算法

Reconstruct Itinerary

dietpepsi 你这个贱人。。我搞了半天才发现这题不是简单的 DFS，而是欧拉回路。。。

因为我们一定可以从 JFK 出发并回到 JFK ，则 JFK 一定是图中的奇点，有奇数的 in+out degrees;

Greedy DFS, building the route backwards when retreating.

这题其实和我之前用 DFS 处理 topological sort 的代码非常像，主要区别在于存 graph 的方式不同，这里是一个 String 直接连着对应的 next nodes，而且形式是 min heap:

> - 原题给的是 **edges**，所以图是自己用 **hashmap** 建的。
> - **min heap** 可以自动保证先访问 **lexicographical order** 较小的；
> - 同时 **poll** 出来的 **node** 自动删除，免去了用 **List** 的话要先 **collections.sort** 再 **remove** 的麻烦。
> - 这种以 **“edge”** 为重心的算法多靠 **heap**，比如 **dijkstra.**

```
public class Solution {
    public List<String> findItinerary(String[][] tickets) {
        LinkedList<String> list = new LinkedList<>();

        HashMap<String, PriorityQueue<String>> map = new HashMap
<>();
        for(String[] ticket : tickets){
            if(!map.containsKey(ticket[0])) map.put(ticket[0], n
ew PriorityQueue<String>());

            map.get(ticket[0]).add(ticket[1]);
        }

        dfs(list, "JFK", map);

        return new ArrayList<String>(list);
    }

    private void dfs(LinkedList<String> list, String airport, Ha
shMap<String, PriorityQueue<String>> map){
        while(map.containsKey(airport) && !map.get(airport).isEm
pty()){
            dfs(list, map.get(airport).poll(), map);
        }
        list.offerFirst(airport);
    }
}
```

密码锁破解序列

首先你要弄明白你面对的是一个神马密码锁，它的特性是这样的： 一个长度为n=4的密码框，一个键盘有k=10个按键，分别是0~9。 你输入0000，系统会验证0000是否是正确密码。 这时候你再输入一个1，不同的密码锁有不同的处理方式 一种会reset，也就是说前面的4个0消失了，你需要继续输入4个数供系统判断。 另一种不会reset，它会验证最近键入的4歌数字，即0001。 我们面对的是后一种。

如果是前一种的话，没啥好想的，破解序列就是4*10000 但是后一种，可以做到长度为10003的破解序列覆盖所有可能的10000个4位数。 题目就是让你找到这个序列。

我觉得这道题在题意明确的情况下，把所有状态看成点，状态之间的转移看成边是比较自然的。 这样就有两种看法，一种就是把4位看成状态，一种就是把3位看成状态。 把4位看成状态的图上面找Hamilton回路，很显然是本题的答案，因为访问了每一个节点一次且只有一次。 把3位看成状态的图上面找欧拉回路可能需要给面试官解释一下。但我觉得还是比较好解释的。 因为每一条边其实代表了一种4位的状态，于是就很好解释了。 那么上面的DFS找欧拉回路的算法就是相当简单有效的解法了。在删除边和查找下一个没有访问的边的复杂度是O(1)的情况下这个算法的复杂度是O(E)的，也就是O(k^n)的，de Buijin构造算法不会比这个复杂度更好。 我这个实现没有用LinkedList或者Hash来保存边的信息，所以每次都是循环所有可能的边，也就是O(k)查找边，所以总的复杂度是O(k^(n+1))。 考虑到 k = 10 n = 4 我觉得没啥问题。

A De Bruijn graph. Every four-digit sequence occurs exactly once if one traverses every edge exactly once and returns to one's starting point (an Eulerian cycle). Every three-digit sequence occurs exactly once if one visits every node exactly once (a Hamiltonian path).

AI, 迷宫生成

Google NYC 挺喜欢问这个的，估计是因为 NYC office 的人都搞 map..

普林斯顿的 **Robert Sedgewick** 大爷在他的算法课里面讲过几种：最简单的一种，随机方向 **DFS "挖墙".** 另外两个稍微 **fancy** 的一点，本质上就是把迷宫当做一个 **graph**，去做一个 **minimum spanning tree.**

https://en.wikipedia.org/wiki/Maze_generation_algorithm

面试 google 这个程度的，写个随机 DFS 生成就可以了~

AI, 迷宫寻路算法

http://www.1point3acres.com/bbs/thread-200439-1-1.html

假设你在一个迷宫里，你不知道迷宫大小，不知道自己方向。你只有以下3个API函数可以调用：

1. 检查是不是已经到了出口
2. 往前move一格（返回true表示成功move，返回false表示失败，不能移动，即撞墙）
3. 原地向左转90度。 要求写个函数把这个迷宫走出去。假设迷宫本身没有loop。正常情况来说楼主应该是可以顺利做出来的，无非就是DFS，唯一要小心的就是不要走圈子，或者说走回头路。

(G) Deep Copy 无向图成有向图

http://www.1point3acres.com/bbs/thread-199776-1-1.html

Deep copy一个无向图成有向图，方向是从value小的node指向value大的

括号与数学表达式的计算

往上加运算符也好，加括号也好，删除括号也好，最稳妥的方式都是 带 **StringBuilder** 从头扫的 **DFS + Backtracking.**

在没有任何优先级后顾之忧的情况下，可以以操作符为分界线，**Divide & Conquer.** 否则得存每个 **subproblem** 的前后缀用于做 **join.**

有乘法优先级顾虑时，需要在 **dfs** 参数里传上一次操作的值，用于恢复和重新计算；连续多个乘法可以自动叠加。

在只有一种括号的情况下，维护 **left / right count** 就可以 **one-pass** 知道到底有几个 **left / right** 括号没正确匹配

Dijkstra 的 **Shunting-Yard** 算法是生成 **RPN** 的大杀器，也是 **calculator** 类问题的通解。

Different Ways to Add Parentheses

先说一个自己的做法，写的是搜索，对于每一个 input string ，我们扫描并建立两个 list; 一个是 operands，一个是 operators. 每次 dfs 的时候挑一对数字和一个操作符，计算，修改 list 做 dfs，返回之后再把 list 改回来。

过了 15/25 个 test case 之后卡在了 "2*3-4*5" 上面，原因是多了一个额外的 -14，因为两个 subproblem 加括号的方式是一样的，顺序不同而已。

所以暴力搜索的问题在于，如果前面和后面加括号的方式是一样的，需要额外手段来判断“重复”。

而DFS 中想 cache 一个 List<>，远远不如 String 方便。

如果实在追求暴力到底，就干脆对每一个状态都搞序列化，记录当前的 operands & operators ，遇到重复的状态就直接返回了。

因此为了避免 dfs + backtracking 可能遇到的 overlap subproblem 返回多个结果的问题，可以先直接 divide & conquer，因为这个搜索结构是树状的，天生的 disjoint.

根据这个思想，我们可以以“操作符”为分隔，借鉴编译器和 reverse polish notation 中的 "expression tree" 来进行计算，结构如下：

这样左右子树都进行的完美的分隔，而且应为 input 为 string ，也非常容易对子问题进行记忆化搜索。

Divide & Conquer, 8ms，超过 41.50%.

```
public class Solution {
    public List<Integer> diffWaysToCompute(String input) {

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

        for(int i = 0; i < input.length(); i++){
            if(!Character.isDigit(input.charAt(i))){
                char operator = input.charAt(i);

                List<Integer> left = diffWaysToCompute(input.sub
string(0, i));
                List<Integer> right = diffWaysToCompute(input.su
bstring(i + 1));

                for(int num1 : left){
                    for(int num2 : right){
                        if(operator == '+') rst.add(num1 + num2)
;
                        if(operator == '-') rst.add(num1 - num2)
;
                        if(operator == '*') rst.add(num1 * num2)
;
                    }
                }
            }
        }

        if(rst.size() == 0) rst.add(Integer.parseInt(input));

        return rst;
    }
}
```

带记忆化搜索，3ms，超过 91.61%

```
public class Solution {
    public List<Integer> diffWaysToCompute(String input) {
        return helper(input, new HashMap<String, List<Integer>>(
```

```
));
    }

    private List<Integer> helper(String str, HashMap<String, Lis
t<Integer>> map){
        if(map.containsKey(str)) return map.get(str);

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

        for(int i = 0; i < str.length(); i++){
            char chr = str.charAt(i);
            if(!Character.isDigit(chr)){
                List<Integer> leftList = helper(str.substring(0,
 i), map);
                List<Integer> rightList = helper(str.substring(i
 + 1), map);

                for(int leftNum : leftList){
                    for(int rightNum : rightList){
                        if(chr == '+') list.add(leftNum + rightN
um);
                        if(chr == '-') list.add(leftNum - rightN
um);
                        if(chr == '*') list.add(leftNum * rightN
um);
                    }
                }
            }
        }

        if(list.size() == 0) list.add(Integer.parseInt(str));

        map.put(str, list);
        return list;
    }
}
```

Longest Valid Parentheses

首先最容易想到的是 naive 的 O(n^2) 解法，即以每一个 '(' 为起点向右扫，看到 '(' count ++，看到 ')' count -- ，什么时候 count = 0 代表当前 substring 是有效的，count 为负则直接 break.

注意这招只在只有一种括号的时候有效，不能用来做 "Valid Parentheses"，会在 "([)]" 这种 test case 上挂掉。

显然，因为时间复杂度过高这么写不能 AC，因为重复扫描实在太多了。

```
public class Solution {
    public int longestValidParentheses(String s) {
        if(s == null || s.length() == 0) return 0;

        int n = s.length();
        int max = 0;
        for(int i = 0; i < n; i++){
            char chr = s.charAt(i);
            if(chr == '('){
                int count = 1;
                int ptr = i + 1;
                while(ptr < n){
                    if(s.charAt(ptr) == '(') count ++;
                    if(s.charAt(ptr) == ')') count --;

                    if(count == 0) max = Math.max(max, ptr - i +
1);
                    if(count < 0) break;

                    ptr ++;
                }
            }
        }

        return max;
    }
}
```

仔细思考一下我们如何定义 "Valid Parenthese"，还有他们都长什么样。

Valid Parenthese 只有可能是这两种情况；

- 以一对括号为 **"kernel"** ，向外扩散的，如 **((()))**
- 多个 **"kernel"** 扩张出现，相互连续的，如 **(())()(())**

参考了 LC 论坛的一种解法，非常巧妙，利用了这样的一个隐藏性质：

- 用 **Stack** 做常规括号匹配，字符串扫描完毕之后，还存留在 **Stack** 中的 **index** 都是无法匹配的字符。
- 如果字符串正常从左向右扫的话，这个 **Stack** 中的元素 **index** 还是排好序的，元素 **index** 之间的 **gap**，就代表着可以匹配的括号。
- 同时要注意考虑字符串最两端作为起点和终点的可能性，用最右端做初始化，用最左端做收尾。

于是就有了下面这个 O(n) 的代码，空间复杂度为 O(n / 2);

```
public class Solution {
    public int longestValidParentheses(String s) {
        if(s == null || s.length() == 0) return 0;
        Stack<Integer> stack = new Stack<>();
        for(int i = 0; i < s.length(); i++){
            if(s.charAt(i) == '(') stack.push(i);
            else if(!stack.isEmpty() && s.charAt(stack.peek()) =
= '(') stack.pop();
            else stack.push(i);
        }
        // Whole string is matched
        //if(stack.isEmpty()) return s.length();

        int rightEnd = s.length();
        int max = 0;
        while(!stack.isEmpty()){
            int cur = stack.pop();
            max = Math.max(max, rightEnd - cur - 1);
            rightEnd = cur;
        }
        max = Math.max(max, rightEnd);

        return max;
    }
}
```

另一种解法是用 DP，空间占用大一倍，但是速度快，思路上非常接近于 Palindrome Partitioning.

- **dp[i] =** 以 **i** 结尾的括号的最长 **valid parenthese** 长度
  - 对于 **'('** 显然 **dp[i] = 0;**
  - 对于 **')'** ，就需要计算一下了。
- 先读 **dp[i - 1]** 的长度，然后根据长度找到最左面的目标位置 **leftPos**，看看是不是 **'('**，如果是，就可以 **dp[i] = dp[i - 1] + 2** 了，代表可以连续构造；

- 另一种情况是多个独立的括号序列相邻，所以每次如果当前位置可以构造括号，要再找一下 **dp[leftPos - 1]** 的长度，把相邻序列串起来。

速度超过85.05% ~

```
  public class Solution {
    public int longestValidParentheses(String s) {
        if(s == null || s.length() == 0) return 0;
        int n = s.length();
        int[] dp = new int[n]; // dp[i] = Maximum valid parenthe
ses length ends with i
        dp[0] = 0;
        int max = 0;

        for(int i = 1; i < s.length(); i++){
            if(s.charAt(i) == ')'){
                int len = dp[i - 1];
                int leftPos = i - len - 1;

                if(leftPos >= 0 && s.charAt(leftPos) == '(') {
                    dp[i] = dp[i - 1] + 2;
                    if(leftPos - 1 >= 0) dp[i] += dp[leftPos - 1
];
                } else {
                    dp[i] = 0;
                }

                max = Math.max(max, dp[i]);
            }
        }

        return max;
    }
}
```

(FB) 简化版，Remove Invalid Parenthese

http://www.1point3acres.com/bbs/thread-192179-1-1.html

"(a)()" -> "(a)()"

"((bc)" -> "(bc)"

")))a((" -> "a"

"(a(b)" ->"(ab)" or "a(b)"

简单说，就是在尽量保留有效括号的情况下，返回任意一种正确结果。

在只有一种括号的时候，是可以扫一遍通过 **+/-** 来找出非法括号的，不过以一个方向定义的 **+/-** 只能找出一种，需要两个方向都扫一遍。

- **"("** 为正 , **")"** 为负 ，从左向右可以找到非法的 **')'**
- **")"** 为正，**"("** 为负，从右向左可以找到非法的 **'('**

后面的就非常 trivial 了。

```
    private static String removeInvalid(String str){
        char[] chrArr = str.toCharArray();
        int val = 0;
        for(int i = 0; i < chrArr.length; i++){
            if(chrArr[i] == '(') val ++;
            else if(chrArr[i] == ')') val --;

            if(val < 0) {
                chrArr[i] = '#';
                val = 0;
            }
        }
        val = 0;
        for(int i = chrArr.length - 1; i >= 0; i--){
            if(chrArr[i] == ')') val ++;
            else if(chrArr[i] == '(') val --;

            if(val < 0){
                chrArr[i] = '#';
                val = 0;
```

```
            }
        }
        StringBuilder sb = new StringBuilder();
        for(int i = 0; i < chrArr.length; i++){
            if(chrArr[i] != '#') sb.append(chrArr[i]);
        }
        return sb.toString();
    }

    public static void main(String[] args) {
        String[] testcases = new String[]{"()()()()(((()()))()("
,
                                          "()AD(S)A()W)(S))()AS
D()",
                                          "))ASDAS()()()Q(((())
QWE)()",
                                          "S()((A)()D))W)Q())(E
Q()))()W(",
                                          "))ASD)QW(()S(Q)(WE)(
Q()(AS)D("};
        for(int i = 0; i < testcases.length; i++){
            System.out.println("Input  : " + testcases[i]);
            System.out.println("Output : " + removeInvalid(testc
ases[i]));
            System.out.println();
        }
    }
```

Remove Invalid Parentheses

这题的结构就和 **Different ways to add parenthese** 不一样，括号之间的相对位置是非常重要的，而且有连续性，不能像那题一样直接在操作符上建一个 **expression tree** 出来，**divide & conquer.**

首先继承上一题 Stack 的经验，我们很容易可以知道需要的最少 edit 次数，以及 invalid 字符位置：跑一遍算法，看 Stack 里剩几个元素，分别在哪就行了。

> 只有一个非法括号的情况：
>
> - **(( )) ( ) '(' (( )) ( )**
> - 可以发现对于 **'('** ，只能向右删下一个 **'('**，相邻的 **'('** 是重复解
> - **( ) (( )) ')' ( ) (( ))**
> - 同理，**')'** 只能往左边找**')'**，相邻 **')'** 为重复解。
>
> 有两个非法括号的情况：
>
> - **( ) (( )) ')' ( ) '(' (( )) ( )**
> - **( ) (( )) ( ) (( )) ( )**
> - **( ) (( )) ')' ( ) '(' (( )) ( )**
> - 可以发现 **')'** 有两个可能的位置，**'('** 有两个可能的位置，于是是 **4** 种组合。

于是这题就变成了一个搜索问题~

> - 问题**1**：既然在反复在字符串上进行修改，如何还能保证 **List<>** 里面的非法字符位置还是正确的？
> - 问题**2**：动态修改的 **string** ，动态变化的 **index**，要处理很多细节保证他们的 **match.**

觉得顺着这个思路越想细节越多。。于是参考了下论坛的 **DFS** 思路，在所有 **DFS** 解法中，我最喜欢这个，非常简洁易懂，而且不依赖什么 **trick**，便于和面试官交流。

- 先扫一遍原 **string** ，记录下需要移除的左括号数量和右括号数量，保证最后的解最优；

- 于是开始 **DFS**，对于每一个新的字符，我们都有两个选择：'留' 或者 '不留'，就像二叉树的分叉一样，留下了 **dfs + backtracking** 的空间。
- 于是当我们针对各种情况进行 **dfs** 的时候，我们一定可以考虑到所有可能的有效 **path**，接下来需要定义什么是 **“有效 path”**
- **base case** 是左右括号和开括号数量都为零，并且 **index** 读取完了所有字符，则把结果添加进去；
- 如果在任何时刻，左右括号和开括号的数量为负，我们都可以直接剪枝返回。

这种解法的主要优点是好理解，空间上因为只用了一个 **StringBuilder** 非常经济，相比 **BFS** 速度和空间上都优异很多。如果说进一步优化的空间在哪，那就是对于 **“重复状态”** 的缓存与记忆化搜索还有提高的空间。

于是很 **naive** 的尝试了下加入 **Set<>** 来记录已经访问过的 **StringBuilder** 状态试图剪枝，然而很容易就出了 **“没有任何返回结果”** 的 **bug** ，其原因是，这个 **dfs + backtracking** 的代码本来就是在一个神似 **binary tree** 的结构上做一个 **dfs** 穷举而已，并不是 **top-down recursion** 那种子问题覆盖的结构，所以不适合用记忆化搜索解决。非要用的话至少 **dfs** 的返回类型就不能是 **void** ，至少也得是个 **List<>** 或者 **Set<>** 之类的 **“子问题结果”** 嘛。

```
public class Solution {
    public List<String> removeInvalidParentheses(String s) {
        Set<String> set = new HashSet<>();

        int leftCount = 0;
        int rightCount = 0;
        int openCount = 0;

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

```
            if(s.charAt(i) == '(') leftCount++;
            if(s.charAt(i) == ')'){
                if(leftCount > 0) leftCount--;
                else rightCount ++;
            }
        }

        dfs(set, s, 0, leftCount, rightCount, openCount, new Str
ingBuilder());

        return new ArrayList<String>(set);
    }

    private void dfs(Set<String> set, String str, int index, int
 leftCount,
                     int rightCount, int openCount, StringBuilde
r sb){
        if(index == str.length() && leftCount == 0 && rightCount
 == 0 && openCount == 0){
            set.add(sb.toString());
            return;
        }

        if(index == str.length() || leftCount < 0 || rightCount
< 0 || openCount < 0) return;

        char chr = str.charAt(index);
        int len = sb.length();

        if(chr == '('){
            // Remove current '('
            dfs(set, str, index + 1, leftCount - 1, rightCount,
openCount, sb);
            // Keep current '('
            dfs(set, str, index + 1, leftCount, rightCount, open
Count + 1, sb.append(chr));
        } else if(chr == ')'){
            // Remove current ')'
            dfs(set, str, index + 1, leftCount, rightCount - 1,
openCount, sb);
```

```
            // Keep current ')'
            dfs(set, str, index + 1, leftCount, rightCount, open
Count - 1, sb.append(chr));
        } else {
            // Just keep the character
            dfs(set, str, index + 1, leftCount, rightCount, open
Count, sb.append(chr));
        }

        // Back-tracking
        sb.setLength(len);
    }
}
```

参考论坛思路的一个 **BFS** 写法，速度很慢只能超过 **14%**，搞的有点像 **word ladder** 了。。。 不过另一个好处是，既然这么高的时间复杂度都能过，我那个 **dfs** 的解法也不用太追求完美，这样 **index offset** 的问题更好解决，遇到个新 **string** 直接重扫就好了。

```
public List<String> removeInvalidParentheses(String s) {
        List<String> rst = new ArrayList<>();
        if(isValid(s)){
            rst.add(s);
            return rst;
        }
        Set<String> visited = new HashSet<>();
        Queue<String> queue = new LinkedList<>();

        boolean isFound = false;
        visited.add(s);
        queue.offer(s);
        while(!queue.isEmpty() && !isFound ){
            int size = queue.size();
            for(int i = 0; i < size; i++){
                String str = queue.poll();
                for(int pos = 0; pos < str.length(); pos++){
                    StringBuilder sb = new StringBuilder(str);
                    sb.deleteCharAt(pos);
```

```
                    String next = sb.toString();

                    if(visited.contains(next)) continue;

                    if(isValid(next)){
                        isFound = true;
                        rst.add(next);
                    } else {
                        queue.offer(next);
                    }

                    visited.add(next);
                }
            }
        }

        return rst;
    }

    private boolean isValid(String s){
        if(s == null || s.length() == 0) return true;
        int count = 0;
        for(int i = 0; i < s.length(); i++){
            if(s.charAt(i) == '(') count++;
            if(s.charAt(i) == ')') count--;

            if(count < 0) return false;
        }

        return (count == 0);
    }
```

Expression Add Operators

num = "105", target = 5;

返回 ["10-5", "1x0+5", "1+0x5", "1x05", "1-0x5"] 是错的

- 不应该直接把 "05" 当成数字；

- 乘法符号有优先级问题。

在 Different ways to add parentheses 里，这么干没有任何问题，因为默认是加括号的，左右子树可以完全分离，单独求值。

而这题里，用 divide & conquer 无可避免地遇到符号优先级问题。于是看了一圈 LC 论坛，大家的做法其实都是 DFS + backtracking ..

这题非要用 divide & conquer 也不是不行，但是极其麻烦，如图所示：

对于每一个节点，我们需要考虑其左子树的 String, value，还有抹去最后一步运算的 String 与 value .. 对于右子树，我们还需要其抹去第一步运算的 String 与 value 才能正确做 join.

所以说，珍爱生命，有乘法就远离 divide & conquer 吧。

一个思路不错的帖子，这种做法就不用考虑 **join**了，只要做乘法的时候看它的前一个元素就行；

流程：

- **dfs + backtracking** 思路
- **dfs** 函数中存 **"**左面表达式的当前值**"** 和 **"**上一步操作的结果**"** **(**用于处理优先级**)**
- 参数都用 **Long**，防止溢出，毕竟 **String** 可能是个大数
- 如果 **start** 位置是 **0** 而且当前循环的 **index** 还是 **0**，直接 **return**，因为一轮循环只能取一次 **0**；
- **start == 0** 的时候，直接考虑所有可能的起始数字扔进去，同时更新 **cumuVal** 和 **prevVal;**
- 除此之外，依次遍历所有可能的 **curVal parse**，按三种不同的可能操作做 **dfs + backtracking;**
    - 加法：直接算，**prevVal** 参数传 **curVal**，代表上一步是加法；
    - 减法：直接算，**prevVal** 参数传 **-curVal**，代表上一步是减法；
    - 乘法：要先减去 **prevVal** 抹去上一步的计算，然后加上 **prevVal** ***curVal*** 代表当前值；同时传的 ***preVal*** 参数也等于 ***prevVal*** **curVal.**
    - 乘法操作这种处理方式，在遇到连续乘法的时候可以看到是叠加的；但是前一步操作如果不是乘法，则可以优先计算乘法操作。

- **10-5-2x6** ，遇到乘法之前是 **cumuVal = 3, preVal = -2;** 新一轮 **dfs** 时 **curVal = 6,** 先退回前一步操作 **- prevVal**，得到 **5**，其实就是之前 **(10 - 5)** 的结果，然后加上当前结果 **(-2 x 5)**，**prevVal** 设成 **-10** 传递过去即可。

```
public class Solution {
    public List<String> addOperators(String num, int target) {
         List<String> rst = new ArrayList<>();
         dfs(rst, new StringBuilder(), num, target, 0, 0, 0);
         return rst;
    }

    private void dfs(List<String> rst, StringBuilder sb, String
num, int target, int startPos, long cumuVal, long prevVal){
        if(startPos == num.length() && cumuVal == target){
            rst.add(sb.toString());
            return;
        }

        for(int i = startPos; i < num.length(); i++){
            // We can pick one zero to start with; but not multi
ple
            if(num.charAt(startPos) == '0' && i != startPos) ret
urn;
            int len = sb.length();
            String curStr = num.substring(startPos, i + 1);
            long curVal = Long.parseLong(curStr);

            if(startPos == 0){
                dfs(rst, sb.append(curVal), num, target, i + 1,
curVal, curVal);
                sb.setLength(len);
            } else {
                dfs(rst, sb.append("+" + curStr), num, target, i
 + 1, cumuVal + curVal, curVal);
                sb.setLength(len);
                dfs(rst, sb.append("-" + curStr), num, target, i
 + 1, cumuVal - curVal, -curVal);
                sb.setLength(len);
```

```
                    dfs(rst, sb.append("*" + curStr), num, target, i
 + 1, cumuVal - prevVal + prevVal * curVal, prevVal * curVal);
                    sb.setLength(len);
                }
            }
        }
    }
```

Basic Calculator

计算器类问题，离不开 **Dijkstra** 的 **Shunting-yard algorithm** 和 **reverse polish notation.**

Input: 3 + 4 * 2 / (1 - 5) ^ 2 ^ 3

operator	precedence	associativity
^	4	Right
*	3	Left
/	3	Left
+	2	Left
–	2	Left

Token	Action	Output (in RPN)	Operator Stack	Notes
3	Add token to output	3		
+	Push token to stack	3	+	
4	Add token to output	3 4	+	
*	Push token to stack	3 4	* +	* has higher precedence than +
2	Add token to output	3 4 2	* +	
/	Pop stack to output	3 4 2 *	+	/ and * have same precedence
	Push token to stack	3 4 2 *	/ +	/ has higher precedence than +
(	Push token to stack	3 4 2 *	(/ +	
1	Add token to output	3 4 2 * 1	(/ +	
-	Push token to stack	3 4 2 * 1	- (/ +	
5	Add token to output	3 4 2 * 1 5	- (/ +	
)	Pop stack to output	3 4 2 * 1 5 -	(/ +	Repeated until "(" found
	Pop stack	3 4 2 * 1 5 -	/ +	Discard matching parenthesis
^	Push token to stack	3 4 2 * 1 5 -	^ / +	^ has higher precedence than /
2	Add token to output	3 4 2 * 1 5 - 2	^ / +	
^	Push token to stack	3 4 2 * 1 5 - 2	^ ^ / +	^ is evaluated right-to-left
3	Add token to output	3 4 2 * 1 5 - 2 3	^ ^ / +	
end	Pop entire stack to output	3 4 2 * 1 5 - 2 3 ^ ^ / +		

- 因为有负号，运算顺序不能颠倒；
- 数字可能是多数位的，也可能是0，不能 assume 都是一位数；
- 带括号的处理方式；

复现了一下 **Dijkstra** 的 **shunting yard algorithm**，因为只有 **“+”** 和 **“-”** ，运算符优先级上要简单一些。在这个代码基础上扩展任何新的运算符都很容易，加个判断函数就行了。

- 建个 **StringBuilder** 存输出的 **RPN**，一个 **Stack** 存运算符；
- 如果看到数字，直接添加到 **StringBuilder** 中；
- 如果看到 **"("** ，直接添加到 **Stack** 上；
- 如果看到 **")"**，把 **Stack** 上的所有运算符 **pop** 出来添加到 **StringBuilder**，直到遇见 **"("** 为止；
- 如果看到运算符，把 **Stack** 上所有 **"**大于等于当前运算符优先级**"** 的 **pop** 出来加到 **StringBuilder**，最后把自己放入 **Stack**，此时要么 **Stack** 为空，要么 **Stack.peek()** 的优先级比当前运算符小。
- 优先级 ：**"**乘除 **= 3"**， **"**加减 **= 2"**

```
public class Solution {
    public int calculate(String s) {
        // "(10+(  14+25+2)-0)+(0+18)"
        // multiple digits;
        // number zero;
        // have spaces;
        return solveRPN(getRPN(s));
    }

    // "123 \s 23 \s + \s 45 \s - \s 44 ..."
    private String getRPN(String s){
        StringBuilder sb = new StringBuilder();
        Stack<Character> stack = new Stack<>();
        for(int i = 0; i < s.length(); i++){
            char chr = s.charAt(i);
            if(chr == ' ') continue;

            if(Character.isDigit(chr)){
                sb.append(chr);
            } else {
                sb.append(' ');

                if(chr == '('){
                    stack.push(chr);
                } else if (chr == ')'){
```

```
                    while(!stack.isEmpty() && stack.peek() != '('
){
                        sb.append(stack.pop());
                        sb.append(' ');
                    }
                    stack.pop();
                } else {
                    // is "+" or "-" operator with same preceden
ce level
                    while(!stack.isEmpty() && stack.peek() != '('
){
                        sb.append(stack.pop());
                        sb.append(' ');
                    }
                    stack.push(chr);
                }
            }
        }
        while(!stack.isEmpty()){
            sb.append(' ');
            sb.append(stack.pop());
        }

        return sb.toString();
    }

    private int solveRPN(String s){
        String[] strs = s.split(" ");
        Stack<Integer> stack = new Stack<>();
        for(String str : strs){
            if(str.equals("")) continue;
            if("+-".indexOf(str) != -1){
                int num2 = stack.pop();
                int num1 = stack.pop();

                if(str.equals("+")) stack.push(num1 + num2);
                if(str.equals("-")) stack.push(num1 - num2);
            } else {
                stack.push(Integer.parseInt(str));
            }
```

```
        }

        return stack.pop();
    }
}
```

Basic Calculator II

Shunting Yard Algorithm 的做法同上，加一个优先级就行。

```
public class Solution {
    public int calculate(String s) {
        return evalRPN(getRPN(s));
    }

    private String getRPN(String s){
        StringBuilder sb = new StringBuilder();
        Stack<Character> stack = new Stack<>();

        for(int i = 0; i < s.length(); i++){
            char chr = s.charAt(i);
            if(chr == ' ') continue;

            if(Character.isDigit(chr)){
                sb.append(chr);
            } else {
                sb.append(' ');
                if(chr == '('){
                    stack.push(chr);
                } else if(chr == ')'){
                    while(stack.peek() != '('){
                        sb.append(stack.pop());
                        sb.append(' ');
                    }
                    stack.pop();
                } else {
                    while(!stack.isEmpty() && getPrecedence(chr)
 <= getPrecedence(stack.peek())){
```

```
                        sb.append(stack.pop());
                        sb.append(' ');
                    }
                    stack.push(chr);
                }
            }
        }
        while(!stack.isEmpty()){
            sb.append(' ');
            sb.append(stack.pop());
        }
        return sb.toString();
    }

    private int evalRPN(String s){
        String[] strs = s.split(" ");
        Stack<Integer> stack = new Stack<>();

        for(String str : strs){
            if(str.equals("")) continue;
            // Is operator
            if("+-*/".indexOf(str) != -1){
                int num2 = stack.pop();
                int num1 = stack.pop();

                if(str.equals("+")) stack.push(num1 + num2);
                if(str.equals("-")) stack.push(num1 - num2);
                if(str.equals("*")) stack.push(num1 * num2);
                if(str.equals("/")) stack.push(num1 / num2);
            } else {
                stack.push(Integer.parseInt(str));
            }
        }
        return stack.pop();
    }

    private int getPrecedence(char chr){
        if(chr == '*' || chr == '/') return 3;
        if(chr == '+' || chr == '-') return 2;
```

```
        return 0;
    }
}
```

Iterator 类

Zigzag Iterator

所谓的 Zigzag 其实等价于 circular ~ 实现 circular 的常见办法有两个：

- 建 array / arraylist，靠取 mod 的 index trick;
- 直接用内置的 deque 库，每次头尾操作就可以

```
public class ZigzagIterator {
    Deque<Iterator<Integer>> deque;

    public ZigzagIterator(List<Integer> v1, List<Integer> v2) {
        deque = new LinkedList<Iterator<Integer>>();
        if(v1.size() != 0) deque.offerFirst(v1.iterator());
        if(v2.size() != 0) deque.offerFirst(v2.iterator());
    }

    public int next() {
        Iterator<Integer> iter = deque.pollLast();
        int num = iter.next();
        if(iter.hasNext()) deque.offerFirst(iter);
        return num;
    }

    public boolean hasNext() {
        return (deque.size() != 0);
    }
}
```

Peeking Iterator

由于先做了 Flatten Nested List Iterator 的 peeking iterator 实现，做这题简直毫无压力。。。

```
// Java Iterator interface reference:
// https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.
html
class PeekingIterator implements Iterator<Integer> {

    Integer peek;
    Iterator<Integer> cur;

    public PeekingIterator(Iterator<Integer> iterator) {
        // initialize any member here.
        cur = iterator;
        peek = internalNext();
    }

    private Integer internalNext(){
        if(cur.hasNext()){
            return cur.next();
        } else {
            return null;
        }
    }

    // Returns the next element in the iteration without advanci
ng the iterator.
    public Integer peek() {
        return peek;
    }

    // hasNext() and next() should behave the same as in the Ite
rator interface.
    // Override them if needed.
    @Override
    public Integer next() {
        Integer tmp = peek;
        peek = internalNext();
        return tmp;
    }

    @Override
```

```
    public boolean hasNext() {
        return (peek != null);
    }
}
```

Flatten 2D Vector

依然是没啥难度。。迭代器好简单。。

```
public class Vector2D implements Iterator<Integer> {
    Iterator<List<Integer>> listIter;
    Iterator<Integer> cur;
    Integer peek;

    public Vector2D(List<List<Integer>> vec2d) {
        listIter = vec2d.iterator();
        peek = internalNext();
    }

    private Integer internalNext(){
        if(cur != null && cur.hasNext()){
            return cur.next();
        } else if(listIter.hasNext()){
            cur = listIter.next().iterator();
            return internalNext();
        } else {
            return null;
        }
    }

    @Override
    public Integer next() {
        Integer tmp = peek;
        peek = internalNext();
        return tmp;
    }

    @Override
    public boolean hasNext() {
        return (peek != null);
    }
}
```

Flatten Nested List Iterator

之前准备 LinkedIn 的时候做过，拿来重用下。

首先我不太认同 LC 的 test case 里没有重复的 hasNext() 调用的情况。。把执行逻辑全堆到 hasNext() 里是有问题的。

关键在于 internalNext() 的实现，还有巧妙递归调用自己减少代码量的实现方式。

- 如果 cur Iterator 还有货，就遍历；
- 遍历出来的是 Integer，就可以直接设 instance variable 了；
- 遍历出来的是 List，说明下面还有子树，先把当前的存 Stack 上，cur 指到下面，再调用自己一次；
- cur 没货了，从 stack 里取一个，接着搞；
- stack 也没货了，就只能返回 null 了 ...

```
public class NestedIterator implements Iterator<Integer> {

    private Integer peek;
    private Stack<Iterator<NestedInteger>> stack;
    private Iterator<NestedInteger> curIter;

    public NestedIterator(List<NestedInteger> nestedList) {
        curIter = nestedList.iterator();
        stack = new Stack<>();
        peek = internalNext();
    }

    private Integer internalNext(){
        if(curIter.hasNext()){
            NestedInteger node = curIter.next();
            if(node.isInteger()){
                return node.getInteger();
            } else {
                stack.push(curIter);
                curIter = node.getList().iterator();

                return internalNext();
            }
        } else if(!stack.isEmpty()){
            curIter = stack.pop();

            return internalNext();
        } else {
```

```
            return null;
        }
    }

    @Override
    public Integer next() {
        Integer rst = peek;
        peek = internalNext();
        return rst;
    }

    @Override
    public boolean hasNext() {
        return (peek != null);
    }
}
```

Binary Search Tree Iterator

顺着这个思路设计的话，BST iterator 也可以做。不过因为 TreeNode 不自带 iterator，往右子树跳的那一下，得自己手动写。

```
public class BSTIterator {

    Stack<TreeNode> stack;
    TreeNode cur;
    TreeNode peek;

    public BSTIterator(TreeNode root) {
        stack = new Stack<>();
        cur = root;
        peek = internalNext();
    }

    private TreeNode internalNext(){
        if(cur != null){
            if(cur.left != null){
                stack.push(cur);
                cur = cur.left;
```

```
                return internalNext();
            } else {
                TreeNode tmp = cur;
                cur = cur.right;
                return tmp;
            }
        } else if (!stack.isEmpty()){
            TreeNode tmp = stack.pop();
            cur = tmp.right;
            return tmp;
        } else {
            return null;
        }
    }

    /** @return whether we have a next smallest number */
    public boolean hasNext() {
        return (peek != null);
    }

    /** @return the next smallest number */
    public int next() {
        TreeNode node = peek;
        peek = internalNext();
        return node.val;
    }
}
```

这题的另一种画风比较常规的写法就是 in-order traversal ~

```
public class BSTIterator {
    Stack<TreeNode> stack;
    TreeNode cur;
    public BSTIterator(TreeNode root) {
        stack = new Stack<TreeNode>();
        cur = root;
    }

    /** @return whether we have a next smallest number */
    public boolean hasNext() {
        return (!stack.isEmpty() || cur != null);
    }

    /** @return the next smallest number */
    public int next() {
        while(cur != null){
            stack.push(cur);
            cur = cur.left;
        }
        TreeNode node = stack.pop();
        cur = node.right;
        return node.val;
    }
}
```

(FB) Binary Tree Post-order iterator

据群内小伙伴表示是 FB 高频提题，自己就写 test case 试试看。

其实这题就是考你到底会不会写只依靠 stack + cur + prev 指针的迭代写法，如果会了，这题就做出来了。

```
import java.util.*;

public class Main {
    private static class TreeNode {
        int val;
        TreeNode left, right;
```

```
        public TreeNode(int val){
            this.val = val;
        }
    }

    private static class PostOrderIterator implements Iterator<T
reeNode>{
        Stack<TreeNode> stack;
        TreeNode cur;
        TreeNode prev;
        public PostOrderIterator(TreeNode root){
            stack = new Stack<>();
            cur = root;
            prev = null;
            if(cur != null) stack.push(cur);
        }
        public boolean hasNext(){
            return (!stack.isEmpty());
        }
        public TreeNode next(){
            TreeNode rst = null;

            while(!stack.isEmpty()){
                cur = stack.peek();
                if(prev == null || prev.left == cur || prev.righ
t == cur){
                    if(cur.left != null){
                        stack.push(cur.left);
                    } else if(cur.right != null){
                        stack.push(cur.right);
                    }
                } else if(cur.left == prev){
                    if(cur.right != null) stack.push(cur.right);
                } else {
                    rst = cur;
                    stack.pop();
                }
                prev = cur;
                if(rst != null) break;
            }
```

```
            return rst;
        }
    }

    public static void main(String[] args){
        /*
        TreeNode node1 = new TreeNode(1);
        TreeNode node2 = new TreeNode(2);
        TreeNode node3 = new TreeNode(3);
        TreeNode node4 = new TreeNode(4);
        TreeNode node5 = new TreeNode(5);
        TreeNode node6 = new TreeNode(6);
        TreeNode node7 = new TreeNode(7);

        node7.left = node3;
        node7.right = node6;
        node3.left = node1;
        node3.right = node2;
        node6.left = node5;
        node5.right = node4;
        */

        TreeNode node1 = new TreeNode(1);
        TreeNode node2 = new TreeNode(2);
        TreeNode node3 = new TreeNode(3);
        TreeNode node4 = new TreeNode(4);
        TreeNode node5 = new TreeNode(5);
        TreeNode node6 = new TreeNode(6);
        TreeNode node7 = new TreeNode(7);
        TreeNode node8 = new TreeNode(8);
        TreeNode node9 = new TreeNode(9);
        TreeNode node10 = new TreeNode(10);

        node10.left = node5;
        node10.right = node9;
        node5.left = node2;
        node5.right = node4;
        node2.left = node1;
        node4.left = node3;
```

```
        node9.left = node6;
        node9.right = node8;
        node8.left = node7;

        PostOrderIterator iter = new PostOrderIterator(node10);

        while(iter.hasNext()){
            System.out.println(iter.next().val);
        }

    }
}
```

Majority Element

遍历一次，保存一个长度为**k - 1**的数组：

- 如果数组中包括这个数，则该数**count + 1**
- 如果数组中有空位，则放入概该，**count**置为**1**
- 如果数组中有数字**count**为**0**，则替换该数，**count**置为**1**
- 所有数字**count**减**1**

最后数组中剩下的数，就是**candidate**，每个统计一下就可以了

这个方法可以解决任意**1/k**的**majority number**

Majority Element

经典的投票算法。

```
public class Solution {
    public int majorityElement(int[] num) {

        int major=num[0], count = 1;
        for(int i=1; i<num.length;i++){
            if(count==0){
                count++;
                major=num[i];
            }else if(major==num[i]){
                count++;
            }else count--;

        }
        return major;
    }
}
```

Majority Element II

Moore's Voting Algorithm.

遍历一次，保存一个长度为k - 1的数组：

1. 如果数组中包括这个数，则该数count + 1
2. 如果数组中有空位，则放入该数，count置为1
3. 如果数组中有数字count为0，则替换该数，count置为1
4. 所有数字count减1

最后数组中剩下的数，就是candidate; 此时再扫一遍数组，确认每个 candidate 的真正出现次数，并根据时候符合最终要求添加到最终结果里。

在这个算法中，count = 0 并不一定代表这个数不是 majority 应该被“立刻”踢出去。

这个方法可以解决任意n/k的majority number

从思想上说，这种做法有点像蓄水池抽样，又有点像 find the celebrity，都是 streaming data 然后维护固定 size 进行淘汰的机制。

```
public class Solution {
```

```java
    public List<Integer> majorityElement(int[] nums) {
        List<Integer> list = new ArrayList<>();
        if(nums == null || nums.length == 0) return list;

        int count1 = 0;
        int count2 = 0;
        int num1 = 0;
        int num2 = 1;
        // 1 and 2 should be initialized to be different numbers

        for(int i = 0; i < nums.length; i++){
            if(nums[i] == num1){
                count1++;
            } else if(nums[i] == num2) {
                count2++;
            } else if(count1 == 0){
                num1 = nums[i];
                count1 = 1;
            } else if(count2 == 0){
                num2 = nums[i];
                count2 = 1;
            } else {
                count1 --;
                count2 --;
            }
        }
        int n = nums.length;

        count1 = 0;
        count2 = 0;
        for(int i = 0; i < nums.length; i++){
            if(nums[i] == num1) count1++;
            else if(nums[i] == num2) count2++;
        }

        if(count1 > n / 3) list.add(num1);
        if(count2 > n / 3) list.add(num2);

        return list;
    }
```

```
}
```

Majority Number III

同样的思路扩展开来的通用解法，不过注意这题说了只会有一个 majority element 所以最后 return 那里有所不同，但是这个算法完全可以找到所有 k - 1 个。

```
    public int majorityNumber(ArrayList<Integer> nums, int k) {
        // write your code
        int n = nums.size();

        List<Integer> numList = new ArrayList<>();
        List<Integer> countList = new ArrayList<>();

        if(nums == null || nums.size() == 0) return -1;

        for(int i = 0; i < n; i++){
            int num = nums.get(i);
            if(numList.size() < k - 1 && !numList.contains(num))
{
                numList.add(num);
                countList.add(1);
            } else {
                int index = numList.indexOf(num);
                if(index != -1){
                    // current number is in candidate list;
                    countList.set(index, countList.get(index) + 1
);
                } else {
                    int zeroIndex = countList.indexOf(0);
                    if(zeroIndex != -1){
                        numList.remove(zeroIndex);
                        countList.remove(zeroIndex);
                        numList.add(num);
                        countList.add(1);
                    } else {
                        for(int j = 0; j < countList.size(); j++
){
```

```
                                countList.set(j, countList.get(j) - 1
);
                            }
                        }
                    }
                }
            }

        //List<Integer> rst = new ArrayList<>();
        for(int i = 0; i < numList.size(); i++){
            int count = 0;
            for(int num : nums){
                if(num == numList.get(i)) count++;
            }
            //if(count > n / k) rst.add(numList.get(i));
            if(count > n / k) return numList.get(i);
        }
        return -1;
    }
```

(Google) Majority Element

http://www.1point3acres.com/bbs/thread-191900-1-1.html

之前google面经里有的关于majority element的题，就是一个排序数组有n个值，求所有出现次数等于或者超过n / k的值。 比如[1 1 2 2 2 2 3 4 5 5 5 5] k = 3 return [2,5]

解决办法就是依次检查所有可能的出现 **majority element** 的位置 **[n/k,2n/k, ... ,n]** ，在每个位置上根据当前元素做 **search for range**，**O(log n)**，如此重复最多 **k** 次即可。

Matrix Inplace Operations

当我们需要 **in-place** 处理矩阵的时候，最简便直接的思路往往是“多个 **pass**”的，即用一次 **pass** 做标记，第二次甚至第三次 **pass** 再做实际处理。

Set Matrix Zeroes

直接 in-place 的尝试一开始错了，试图每次看到一个 0 ，都把矩阵分为其左下和右下的子矩阵，递归处理，但是没有充分考虑到同一行上可能有其他 col 上的 0，会把这个 col 对应的下面位置也设成 0 的情况，因此是错误的。

- 先扫描第一行和第一列，记录第一行/列是否为 **0**；
- 在此之后，第一行与第一列起到标记作用，**header.**
- 扫描里面，在任何位置看到 **0** ，都把对应的 行/列 头设为 **0**
- 再次扫描里面，如果行或列的头位置为 **0** ，则设为 **0**；
- 此时第一行和列已经失去作为记录的作用，开始根据最开始的记录来设 **0.**

```
public class Solution {
    public void setZeroes(int[][] matrix) {
        if(matrix == null || matrix.length == 0) return;

        int rows = matrix.length;
        int cols = matrix[0].length;

        boolean firstRowZero = false;
        boolean firstColZero = false;

        for(int i = 0; i < rows; i++) if(matrix[i][0] == 0) firs
tColZero = true;
        for(int i = 0; i < cols; i++) if(matrix[0][i] == 0) firs
tRowZero = true;

        for(int i = 1; i < rows; i++){
            for(int j = 1; j < cols; j++){
                if(matrix[i][j] == 0) matrix[i][0] = matrix[0][j
] = 0;
            }
        }

        for(int i = 1; i < rows; i++){
            for(int j = 1; j < cols; j++){
                if(matrix[i][0] == 0 || matrix[0][j] == 0) matri
x[i][j] = 0;
            }
        }

        if(firstRowZero) for(int i = 0; i < cols; i++) matrix[0]
[i] = 0;
        if(firstColZero) for(int i = 0; i < rows; i++) matrix[i][
0] = 0;
    }
}
```

Game of Life

吸取了上一题的经验之后，这题很容易就一次 AC 了~

- 0 : 未访问，死
- 1 : 未访问，活
- 2 : 已访问，当前活，下轮活
- 3 : 已访问，当前活，下轮死
- -2 : 已访问，当前死，下轮活
- -3 : 已访问，当前死，下轮死

正确定义了状态之后就不会产生邻居在不同迭代周期时互相影响的问题了。

从以上两题可以发现，当我们需要 **in-place** 处理矩阵的时候，最简便直接的思路往往是“多个 **pass**”的，即用一次 **pass** 做标记，第二次甚至第三次 **pass** 再做实际处理。

```
public class Solution {
    public void gameOfLife(int[][] board) {
        for(int i = 0; i < board.length; i++){
            for(int j = 0; j < board[0].length; j++){
                board[i][j] = nextState(board, i, j);
            }
        }

        for(int i = 0; i < board.length; i++){
            for(int j = 0; j < board[0].length; j++){
                if(board[i][j] == 2 || board[i][j] == -2) board[
i][j] = 1;
                else board[i][j] = 0;
            }
        }
    }

    private int nextState(int[][] board, int row, int col){
        int[] xDirs = {0, 0, 1, -1, 1, -1, -1,  1};
        int[] yDirs = {1,-1, 0,  0, 1, -1,  1, -1};

        int aliveCount = 0;

        for(int i = 0; i < 8; i++){
```

```
                int x = row + xDirs[i];
                int y = col + yDirs[i];
                // legal position
                if(x >= 0 && x < board.length && y >= 0 && y < board[
0].length){
                    if(board[x][y] > 0) aliveCount ++;
                }
            }

            if(board[row][col] == 0){
                if(aliveCount == 3) return -2;
                else                return -3;
            } else {
                if(aliveCount == 2 || aliveCount == 3) return 2;
                else                                   return 3;
            }
        }
    }
```

Rotate Image

和剥洋葱差不多，一层一层从外向里；我第一种写的多一点，但是更值得学习和更好推广的是第二种。

```java
public class Solution {
    public void rotate(int[][] matrix) {
        if(matrix == null || matrix.length == 0) return;

        int n = matrix.length;

        for(int i = 0; i < n / 2; i++){
            for(int j = i; j < n - i - 1; j++){
                swap(matrix, i, j, j, n - 1 - i);
                swap(matrix, i, j, n - 1 - i, n - 1 - j);
                swap(matrix, i, j, n - 1 - j, i);
            }
        }
    }

    private void swap(int[][] matrix, int x1, int y1, int x2, int
 y2){
        int tmp = matrix[x1][y1];
        matrix[x1][y1] = matrix[x2][y2];
        matrix[x2][y2] = tmp;
    }
}
```

另一种更直观更好理解的方式是存两个指针，**a , b** , 分别代表着当前处理这层的“左上”和“右下”位置 **(**矩阵是正方形**)**，然后 **offset** 和各种 **index** 就好计算很多，也完全不需要考虑到 **base case** 上时候的各种特殊情况。

```
public class Solution {
    public void rotate(int[][] matrix) {
        if(matrix == null || matrix.length == 0) return;

        int n = matrix.length;
        int a = 0;
        int b = n - 1;

        while(a < b){
            for(int i = 0; i < (b - a); i++){
                swap(matrix, a, a + i, a + i, b);
                swap(matrix, a, a + i, b, b - i);
                swap(matrix, a, a + i, b - i, a);
            }
            ++a;
            --b;
        }
    }

    private void swap(int[][] matrix, int x1, int y1, int x2, int
 y2){
        int tmp = matrix[x1][y1];
        matrix[x1][y1] = matrix[x2][y2];
        matrix[x2][y2] = tmp;
    }
}
```

Spiral Matrix

维护四个边界，按顺序输出之后步步收缩。

特别注意处理“下”和“左”边界的时候，有可能当前这层只有一行，或者一列，已经输出过了不需要重复输出。所以这两条边的循环上要注意加一个判断条件。

```
public class Solution {
    public List<Integer> spiralOrder(int[][] matrix) {
        List<Integer> list = new ArrayList<>();
        if(matrix == null || matrix.length == 0) return list;

        int rowStart = 0;
        int rowEnd = matrix.length - 1;
        int colStart = 0;
        int colEnd = matrix[0].length - 1;

        while(rowStart <= rowEnd && colStart <= colEnd){
            // add top frame
            for(int i = colStart; i <= colEnd; i++) list.add(mat
rix[rowStart][i]);
            rowStart ++;
            // add right frame
            for(int i = rowStart; i <= rowEnd; i++) list.add(mat
rix[i][colEnd]);
            colEnd --;
            // add bot frame
            if(rowStart <= rowEnd) for(int i = colEnd; i >= colS
tart; i--) list.add(matrix[rowEnd][i]);
            rowEnd --;
            // add left frame
            if(colStart <= colEnd) for(int i = rowEnd; i >= rowS
tart; i--) list.add(matrix[i][colStart]);
            colStart ++;
        }

        return list;
    }
}
```

Spiral Matrix II

顺着同一个思路，于是这题可以轻松随意撸出来~~

```
public class Solution {
    public int[][] generateMatrix(int n) {
        if(n <= 0) return new int[0][0];

        int[][] matrix = new int[n][n];

        int rowStart = 0;
        int rowEnd = n - 1;
        int colStart = 0;
        int colEnd = n - 1;

        int curNum = 1;

        while(rowStart <= rowEnd && colStart <= colEnd){
            for(int i = colStart; i <= colEnd; i++) matrix[rowSt
art][i] = curNum++;
            rowStart ++;

            for(int i = rowStart; i <= rowEnd; i++) matrix[i][co
lEnd] = curNum++;
            colEnd --;

            if(rowStart <= rowEnd) for(int i = colEnd; i >= colS
tart; i--) matrix[rowEnd][i] = curNum++;
            rowEnd --;

            if(colStart <= colEnd) for(int i = rowEnd; i >= rowS
tart; i--) matrix[i][colStart] = curNum++;
            colStart ++;
        }

        return matrix;
    }
}
```

(G) 对角打印矩阵，左上到右下，按对角线长度降序打印

- 自定义一个函数，给定起点，打印对角线；
- 检查下 **rows / cols** 的长度，先把长的边长起点放进去；
- 而后依次轮流放入第一列 / 第一行的新起点，依次打印即可。

```
    public static void printMatrix(int[][] matrix){
        if(matrix == null || matrix.length == 0) return;
        int rows = matrix.length;
        int cols = matrix[0].length;

        int row = 0;
        int col = 0;
        if(cols > rows){
            for(int i = 0; i < cols - rows; i++){
                printDiagnal(matrix, 0, col++);
            }
        } else if(rows > cols){
            for(int i = 0; i < rows - cols; i++){
                printDiagnal(matrix, row++, 0);
            }
        }

        printDiagnal(matrix, row++, col++);

        while(row < rows || col < cols){
            if(rows - row < cols - col){
                printDiagnal(matrix, 0, col++);
            } else {
                printDiagnal(matrix, row++, 0);
            }
        }

    }

    private static void printDiagnal(int[][] matrix, int row, int
col){
        if(matrix == null || matrix.length == 0) return;
```

```
        int rows = matrix.length;
        int cols = matrix[0].length;

        while(row < rows && col < cols){
            System.out.print(" " + matrix[row++][col++]);
        }
        System.out.println();
    }

    public static void main(String[] args){
        int rows = 8;
        int cols = 5;

        int[][] matrix = new int[rows][cols];

        int num = 1;
        for(int i = 0; i < 8; i++){
            for(int j = 0; j < 5; j++){
                System.out.print(num + "    ");
                matrix[i][j] = num++;
            }
            System.out.println();
        }

        printMatrix(matrix);
    }
```

(G)数据结构

LRU Cache

一道非常亲切熟悉的题，当初刷 lintcode 的时候非常喜欢这道。这道题因为细节很多，但是操作重复性高，所以非常适合先把流程写下来，然后用 helper 函数实现。

一句话描述这题的话，就是一个 “头 + 尾 dummy node 的双向链表” + “HashMap”.

- 先沟通各种 **input/output**
- 思考流程
- 把流程用 **comments (另一种颜色 mark 笔写下来)**
- 把发现流程中重复率高的地方写成 **function**
- 模块化填充，改错。

```
public class LRUCache {
    private class Node{
        int key;
        int value;
        Node prev;
        Node next;
        public Node(int key, int value){
            this.key = key;
            this.value = value;
        }
    }

    private int capacity;
    private HashMap<Integer, Node> map;
    // Head : Most recently used
    private Node dummyHead;
```

```
    // Tail : Least recently used
    private Node dummyTail;

    public LRUCache(int capacity) {
        this.capacity = capacity;
        this.map = new HashMap<Integer, Node>();
        dummyHead = new Node(-1,-1);
        dummyTail = new Node(-1,-1);
        dummyHead.next = dummyTail;
        dummyTail.prev = dummyHead;
    }

    public int get(int key) {
        if(!map.containsKey(key)) return -1;

        // Get current node
        Node node = map.get(key);
        // Disconnect it from the linkedlist
        node.prev.next = node.next;
        node.next.prev = node.prev;
        // Move it to the front
        moveToFront(node);
        // Return value
        return node.value;
    }

    public void set(int key, int value) {
        if(map.containsKey(key)){
            // Get current node
            Node node = map.get(key);
            // Disconnect it from the linkedlist
            node.prev.next = node.next;
            node.next.prev = node.prev;
            // Move it to the front
            moveToFront(node);
            // set value
            node.value = value;
        } else {
            if(map.size() >= capacity){
                // Remove LRU element and its key
```

```
                removeLRU();
            }
            // Create new node
            Node node = new Node(key, value);
            // Save its key in map
            map.put(key, node);
            // Move it to the front
            moveToFront(node);
        }
    }

    private void moveToFront(Node node){
        Node oldHead = dummyHead.next;

        // Connect node to dummyHead
        dummyHead.next = node;
        node.prev = dummyHead;

        // Connect node to oldHead
        node.next = oldHead;
        oldHead.prev = node;
    }

    private void removeLRU(){
        Node node = dummyTail.prev;
        // Disconnect node from linkedlist
        node.prev.next = node.next;
        node.next.prev = node.prev;
        // Set null of its pointers
        node.next = null;
        node.prev = null;
        // Remove its key from hashmap
        map.remove(node.key);
    }
}
```

Find Median from Data Stream

也是非常经典的题。

- new PriorityQueue<>(Collections.reverseOrder());
- 把所有数字分成两个区间，一个区间为 <= median 的，存在 maxHeap 里；另一个为 >= median 的，存在 minHeap 里，这样我们每次在堆顶看到的都是 median candidates.
- 因此在插入新元素的时候，如果发现其 <= maxHeap top，则说明它位于左区间，插入 max; 反之则插入 minHeap 所代表的右区间。
- 每次新元素插入之后要保持平衡，堆顶的元素切换其实代表着区间分界线的转移。

```
public class MedianFinder {
    // max heap stores all elements smaller / equal to median
    PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(
Collections.reverseOrder());
    // min heap stores all elements bigger / equal to median
    PriorityQueue<Integer> minHeap = new PriorityQueue<Integer>(
);

    // Adds a number into the data structure.
    public void addNum(int num) {
        // Check heap top

        if(maxHeap.size() == 0 || num <= maxHeap.peek()){
            maxHeap.offer(num);
        } else {
            minHeap.offer(num);
        }

        // min/max heap's size would not differ more than one
        // Rebalance if necessary
        while(Math.abs(maxHeap.size() - minHeap.size()) > 1){
            if(maxHeap.size() > minHeap.size()){
                minHeap.offer(maxHeap.poll());
            } else {
                maxHeap.offer(minHeap.poll());
            }
        }
    }

    // Returns the median of current data stream
```

```
    public double findMedian() {
        int maxSize = maxHeap.size();
        int minSize = minHeap.size();

        if(maxSize == minSize) return (double) (maxHeap.peek() +
 minHeap.peek()) / 2;
        if(maxSize - 1 == minSize) return (double) maxHeap.peek(
);
        if(maxSize == minSize - 1) return (double) minHeap.peek(
);

        return 0;
    }
};
```

Sliding Window Maximum

操作特点：

- 只关心区域内的 **max;**
- 只要同一区域内 **max** 一样，输出就一样，不在意元素的出现顺序，因此这题有别于 **max/min stack** 的保存元素顺序要求。

这题暴力循环可以 O(nk)，hasheap 可以 O(n log k)，deque 可以 O(n).

我们每一步要执行下面三个操作：

- 添加当前元素 nums[i];
- 删除指定元素 nums[i - k];
- 找到当前 window max;

暴力解法中，我们可以直接维护一个 list，每次操作都进行扫描，这样的复杂度是：

- Add O(1)
- Remove O(k)
- max O(k)

另一种选择是，维护一个 sorted list，这样的复杂度是：

- Add O(k)
- Remove O(k)
- max O(1)

再观察题目特性可以发现，我们在维护 sorted list 上有很多可以取巧的地方：

- 因为我们只关心每个 window 上的 max，因此没有必要把 window 内的所有元素都留着，在 Add 的时候可以直接把小于新元素的值都 pop 掉；
- 而在 remove 上，对最终结果会产生影响的，只有我们 remove 的元素就是 max 的情形，因为其他无关元素在 add 的时候就被拿掉了；因此我们可以存一个元素的相对位置，以此来判断我们要删掉的元素是不是当前的 max.
- 因为我们维护的是 sorted array ，max 依然是 O(1).
- 由于每一个元素只会被 add 和 remove 一次，整个流程的均摊复杂度就是 O(1).

要点：

- 维护一个可以双向操作的 **sorted array.**
- **Deque** 里面要存 **index**，而不是值，不然会出现 **[8,1,2,8...]** 这种情况下，我们有可能会把刚加进去的 **8** 又给拿出来的 **bug.**

```
public class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        if(nums == null || nums.length == 0) return new int[0];

        int[] rst = new int[nums.length - k + 1];
        // Deque stores indexes of elements i, in descending ord
er of nums[i]
        // First : smallet element in current window
        // Last : biggest element in current window
        Deque<Integer> deque = new LinkedList<>();

        for(int i = 0; i < nums.length; i++){
            while(!deque.isEmpty() && nums[i] > nums[deque.peekF
irst()]){
                deque.pollFirst();
            }
            deque.offerFirst(i);
            // element to remove from sliding window
            if(i - k == deque.peekLast()) deque.pollLast();
            if(i - k + 1 >= 0) rst[i - k + 1] = nums[deque.peekL
ast()];
        }

        return rst;
    }
}
```

(G) Design / OOD 类算法题

Logger Rate Limiter

非常的 trivial ... 用一个 hashmap 做一个类似 inverted index 功能就可以了，需要注意的地方是如果这轮输出了 true，要同时更新 hashmap 里面的 timestamp，输出 false 的时候不更新，因为还没 print.

```
public class Logger {
    HashMap<String, Integer> map;
    /** Initialize your data structure here. */
    public Logger() {
        map = new HashMap<String, Integer>();
    }

    /** Returns true if the message should be printed in the giv
en timestamp, otherwise returns false.
        If this method returns false, the message will not be pr
inted.
        The timestamp is in seconds granularity. */
    public boolean shouldPrintMessage(int timestamp, String mess
age) {
        boolean rst = false;

        if(!map.containsKey(message) || timestamp - map.get(mess
age) >= 10) rst = true;
        if(rst) map.put(message, timestamp);

        return rst;
    }
}
```

Design Hit Counter

承接了 sliding window maximum 的灵感，维护一个所谓“固定大小”的 sorted deque，两端操作，从 peekLast() 开始看，小于当前起始 time interval 的就直接扔掉，也完全可以处理同一时间多个 hit 的情况。

不过貌似这写法速度不够快，126ms，只超过了 17.46%，一看论坛上快的解法都是用 int[] + circular buffer 的写法。

Follow up:

What if the number of hits per second could be very large? Does your design scale?

建 int[] array 显然就不现实了，不如 deque !

```
public class HitCounter {
    Deque<Integer> deque;
    /** Initialize your data structure here. */
    public HitCounter() {
        deque = new LinkedList<Integer>();
    }

    /** Record a hit.
        @param timestamp - The current timestamp (in seconds gra
nularity). */
    public void hit(int timestamp) {
        deque.offerFirst(timestamp);
    }

    /** Return the number of hits in the past 5 minutes.
        @param timestamp - The current timestamp (in seconds gra
nularity). */
    public int getHits(int timestamp) {
        if(timestamp <= 300) return deque.size();

        int startTime = timestamp - 300;
        while(!deque.isEmpty() && deque.peekLast() <= startTime)
 deque.pollLast();

        return deque.size();
    }
}
```

Design Tic-Tac-Toe

这题的 index 用法其实和 N-queens 的备忘录法一样，存行，列，各个对角线就行了。

```
public class TicTacToe {
    // array[][index]
    // index 0 : # of player 1's pieces
    // index 1 : # of player 2's pieces
    private int[] leftDiag;
```

```
    private int[] rightDiag;

    private int[][] rows;
    private int[][] cols;

    private int n;

    /** Initialize your data structure here. */
    public TicTacToe(int n) {
        this.n = n;
        leftDiag = new int[2];
        rightDiag = new int[2];
        rows = new int[n][2];
        cols = new int[n][2];
    }

    /** Player {player} makes a move at ({row}, {col}).
        @param row The row of the board.
        @param col The column of the board.
        @param player The player, can be either 1 or 2.
        @return The current winning condition, can be either:
                0: No one wins.
                1: Player 1 wins.
                2: Player 2 wins. */
    public int move(int row, int col, int player) {
        rows[row][player - 1] ++;
        cols[col][player - 1] ++;
        if(row == col) leftDiag[player - 1] ++;
        if(row + col == n - 1) rightDiag[player - 1]++;

        if(rows[row][player - 1] == n || cols[col][player - 1] =
= n ||
           leftDiag[player - 1] == n || rightDiag[player - 1] ==
 n){
            return player;
        } else {
            return 0;
        }
    }
}
```

看了下论坛，这题还可以进一步做空间优化，利用只有两个玩家的特点省掉一半空间。

```
public class TicTacToe {
    // array[][index]
    // index 0 : # of player 1's pieces
    // index 1 : # of player 2's pieces
    private int leftDiag;
    private int rightDiag;

    private int[] rows;
    private int[] cols;

    private int n;

    /** Initialize your data structure here. */
    public TicTacToe(int n) {
        this.n = n;
        leftDiag = 0;
        rightDiag = 0;
        rows = new int[n];
        cols = new int[n];
    }

    /** Player {player} makes a move at ({row}, {col}).
        @param row The row of the board.
        @param col The column of the board.
        @param player The player, can be either 1 or 2.
        @return The current winning condition, can be either:
                0: No one wins.
                1: Player 1 wins.
                2: Player 2 wins. */
    public int move(int row, int col, int player) {
        int delta = (player == 1) ? 1: -1;

        rows[row] += delta;
        cols[col] += delta;
        if(row == col) leftDiag += delta;
        if(row + col == n - 1) rightDiag += delta;
```

```
        if(Math.abs(rows[row]) == n || Math.abs(cols[col]) == n
||
           Math.abs(leftDiag) == n || Math.abs(rightDiag) == n){
            return player;
        } else {
            return 0;
        }
    }
}
```

Design Snake Game

先用 Deque 写了第一版，速度只超过了 2.35% .. 可能我每步都调用 String 复杂度太高了。

需要注意的细节：

- **food** 吃完之后还可以继续玩，蛇每次依然在动，所以要在合适的地方 **access food matrix** 免得越界。
- 吃自己身体会 **game over**，但是吃尾巴尖没事。。

用了另外一个 **deque** 存 **body index** 速度就快多了超过 **54%**，但是没什么大改动就不贴代码了。另外一个小改动是，其实用不着存 **score** 变量，因为 **score =** 蛇长度 **- 1**

```
public class SnakeGame {
    int[][] food;
    int foodIndex;
    // deque of String "x y" representing positions of snake
    Deque<String> snake;

    int score;
    int rows;
    int cols;
```

```
    /** Initialize your data structure here.
        @param width - screen width
        @param height - screen height
        @param food - A list of food positions
        E.g food = [[1,1], [1,0]] means the first food is positi
oned at [1,1], the second is at [1,0]. */
    public SnakeGame(int width, int height, int[][] food) {
        this.rows = height;
        this.cols = width;
        this.food = food;
        this.foodIndex = 0;
        this.score = 0;
        this.snake = new LinkedList<String>();

        snake.offerFirst("0,0");
    }

    /** Moves the snake.
        @param direction - 'U' = Up, 'L' = Left, 'R' = Right, 'D
' = Down
        @return The game's score after the move. Return -1 if ga
me over.
        Game over when snake crosses the screen boundary or bite
s its body. */
    public int move(String direction) {
        String[] coordinates = snake.peekFirst().split(",");
        int headX = Integer.parseInt(coordinates[0]);
        int headY = Integer.parseInt(coordinates[1]);

        if(direction.equals("U")) headX--;
        if(direction.equals("L")) headY--;
        if(direction.equals("R")) headY++;
        if(direction.equals("D")) headX++;

        // Hit wall
        if(headX < 0 || headX >= rows) return -1;
        if(headY < 0 || headY >= cols) return -1;

        // Hit itself
        String newPos = headX + "," + headY;
```

```
        // It's ok if you bite the tail
        if(snake.contains(newPos) && !snake.peekLast().equals(ne
wPos)) return -1;

        // No food, just move
        if(foodIndex >= food.length){
            snake.offerFirst(newPos);
            snake.pollLast();
            return score;
        }

        int foodX = food[foodIndex][0];
        int foodY = food[foodIndex][1];

        // Got food !
        if(headX == foodX && headY == foodY){
            snake.offerFirst(newPos);
            score ++;
            foodIndex ++;
            return score;
        } else {
            snake.offerFirst(newPos);
            snake.pollLast();
            return score;
        }
    }
}
```

Design Twitter

一个典型的 Observer pattern；为了降低耦合度，每个 user 维护自己的 tweet list 和 follow list，而 post tweet 只是负责自行生产，而不主动执行 push; 每次刷新的时候自己从 follow 列表抓 tweets 就好了。

这题一个需要注意的细节是，排序顺序是按照 timestamp，接口里给的 tweetId ，和时间顺序没有必然的联系，所以要自行维护全局时间戳。

除此之外就是比较直观的 merge k sorted list 问题了，为了方便处理，我们维护了一个 Tweet object 的 LinkedList，但同时 Tweet 存了一个自己的 prev 指针，方便进行多路归并的时候正确找到下一个元素。

其他需要注意的就是各种 corner case：关注自己，取关不存在的人，取关自己本来就没关注的人，自己或者自己的关注用户没有任何 post，等等。

```
public class Twitter {
    // Key : user
    // Value : list of tweets
    HashMap<Integer, LinkedList<Tweet>> tweetsMap;

    // Key : user
    // Value : list of userId's the user has been following
    HashMap<Integer, Set<Integer>> follows;

    int timestamp;

    private class Tweet implements Comparable<Tweet>{
        int timestamp;
        int tweetId;
        Tweet prev;

        public Tweet(int timestamp, int tweetId, Tweet prev){
            this.timestamp = timestamp;
            this.tweetId = tweetId;
            this.prev = prev;
        }
        public int compareTo(Tweet a){
            return a.timestamp - this.timestamp;
        }
    }

    /** Initialize your data structure here. */
    public Twitter() {
        tweetsMap = new HashMap<Integer, LinkedList<Tweet>>();
        follows = new HashMap<Integer, Set<Integer>>();
        timestamp = 0;
    }
```

```
    /** Compose a new tweet. */
    public void postTweet(int userId, int tweetId) {
        if(!tweetsMap.containsKey(userId)) tweetsMap.put(userId,
new LinkedList<Tweet>());

        Tweet prev = (tweetsMap.get(userId).size() == 0) ? null:
 tweetsMap.get(userId).peekFirst();
        tweetsMap.get(userId).offerFirst(new Tweet(timestamp, tw
eetId, prev));
        timestamp++;
    }

    /** Retrieve the 10 most recent tweet ids in the user's news
 feed. Each item in the news feed must be posted by users who th
e user followed or by the user herself. Tweets must be ordered f
rom most recent to least recent. */
    public List<Integer> getNewsFeed(int userId) {
        List<Integer> list = new ArrayList<>();
        PriorityQueue<Tweet> maxHeap = new PriorityQueue<Tweet>(
);

        if(tweetsMap.containsKey(userId) && tweetsMap.get(userId
).size() > 0)
            maxHeap.offer(tweetsMap.get(userId).peekFirst());

        if(follows.containsKey(userId)){
            for(int followee : follows.get(userId)){
                if(tweetsMap.containsKey(followee) && tweetsMap.
get(followee).size() > 0)
                    maxHeap.offer(tweetsMap.get(followee).peekFi
rst());
            }
        }

        for(int i = 0; i < 10 && !maxHeap.isEmpty(); i++){
            Tweet tweet = maxHeap.poll();
            list.add(tweet.tweetId);
            if(tweet.prev != null) maxHeap.offer(tweet.prev);
        }
```

```
        return list;
    }

    /** Follower follows a followee. If the operation is invalid
, it should be a no-op. */
    public void follow(int followerId, int followeeId) {
        // Can't follow yourself
        if(followerId == followeeId) return;
        if(!follows.containsKey(followerId)) follows.put(followe
rId, new HashSet<Integer>());

        follows.get(followerId).add(followeeId);
    }

    /** Follower unfollows a followee. If the operation is inval
id, it should be a no-op. */
    public void unfollow(int followerId, int followeeId) {
        if(!follows.containsKey(followerId) || !follows.get(foll
owerId).contains(followeeId)) return;

        follows.get(followerId).remove(followeeId);
    }
}
```

Design Phone Directory

我还以为要写个 trie 什么的。。这题是什么鬼。。写个 boolean[] 都能 AC.... 697 ms 好么。。

下面这个代码的 get 时间复杂度为 O(maxNumber)，显然太高了。

```
public class PhoneDirectory {
    boolean[] phoneBook;
    /** Initialize your data structure here
        @param maxNumbers - The maximum numbers that can be stor
ed in the phone directory. */
    public PhoneDirectory(int maxNumbers) {
        phoneBook = new boolean[maxNumbers];
    }

    /** Provide a number which is not assigned to anyone.
        @return - Return an available number. Return -1 if none
is available. */
    public int get() {
        for(int i = 0; i < phoneBook.length; i++){
            if(!phoneBook[i]){
                phoneBook[i] = true;
                return i;
            }
        }
        return -1;
    }

    /** Check if a number is available or not. */
    public boolean check(int number) {
        return !phoneBook[number];
    }

    /** Recycle or release a number. */
    public void release(int number) {
        phoneBook[number] = false;
    }
}
```

稍微快一点的做法是，用一个 Queue 维护目前所有空位，用 Set 维护目前所有已经使用的电话号码。主要 overhead 在于初始化的时候要花 O(n) 时间建队列，还有在 HashSet 里面做删除操作。

这种做法是靠均摊复杂度平摊 **cost** 的典型例子。

这题的另一种比较省空间的做法是用 **BitSet**，就看对 **API** 的熟悉程度了。

- **bitset.nextClearBit(smallestFreeIndex);**
- **bitset.clear(number);**
- **bitset.get(number) == false;**
- **bitset.set(smallestFreeIndex);**

```
public class PhoneDirectory {
    Set<Integer> set;
    Queue<Integer> available;
    /** Initialize your data structure here
        @param maxNumbers - The maximum numbers that can be stor
ed in the phone directory. */
    public PhoneDirectory(int maxNumbers) {
        set = new HashSet<>();
        available = new LinkedList<>();
        for(int i = 0; i < maxNumbers; i++){
            available.offer(i);
        }
    }

    /** Provide a number which is not assigned to anyone.
        @return - Return an available number. Return -1 if none
is available. */
    public int get() {
        if(available.size() == 0) return -1;
        int num  = available.poll();
        set.add(num);
        return num;
    }

    /** Check if a number is available or not. */
    public boolean check(int number) {
        return !set.contains(number);
    }

    /** Recycle or release a number. */
    public void release(int number) {
        if(!set.contains(number)) return;

        available.offer(number);
        // Amortized O(1) to remove in hashmap / hashset
        set.remove(number);
    }
}
```

(G) Design Battleship Game

(G) Design Elevator

(A) Design Parking lot

(LinkedIn) Data Stream as Disjoint Intervals

随机算法 & 数据结构

- **rd.nextInt(n)** 可以生成 **[0 - n)** 之间的整数，注意开区间。
- 一般要靠 **arraylist**，因为有 **get()** 函数。
- **remove** 任意位置是 **O(n)**，但是每次只 **remove** 末位的话，平均是 **O(1)**，在知道 **index** 的情况下 **swap** 一下就好了。

随机数的帖子

http://www.1point3acres.com/bbs/forum.php?mod=viewthread&tid=119376&highlight=%CB%E6%BB%FA

Insert Delete GetRandom O(1)

实现 O(1) 的 GetRaondom 可以通过如下方式：

- 生成范围随机数
- 对应的数据结构支持 **O(1) get**
- 可以维护连续 **key** 区间，保证随机数时间复杂度

于是一开始想了下自己维护一个 LinkedList，但是不支持 O(1) get 只能算了。。

ArrayList 的问题在于，它的任意位置 delete 操作是 O(n)，和自身长度成比例的，因为要 shift 元素，怎么能把这个操作变成 O(1) 呢？

这里要做个取巧的事，考虑到所有**value**都是 **integer**，我们可以直接 **swap**，这样删除 **“尾巴”** 就是 **O(1)** 的平均复杂度了。

- **rd.nextInt(n)** 可以生成 **[0 - n)** 之间的整数，注意开区间。

- 正好可以当 **index** 用。

```
import java.util.Random;

public class RandomizedSet {
    // Key : number
    // Value : corresponding index in arraylist
    Map<Integer, Integer> map;
    List<Integer> list;
    Random rd;
    /** Initialize your data structure here. */
    public RandomizedSet() {
        map = new HashMap<Integer, Integer>();
        list = new ArrayList<Integer>();
        rd = new Random();
    }

    /** Inserts a value to the set. Returns true if the set did
not already contain the specified element. */
    public boolean insert(int val) {
        // Set already has given value
        if(map.containsKey(val)) return false;

        map.put(val, list.size());
        list.add(val);

        return true;
    }

    /** Removes a value from the set. Returns true if the set co
ntained the specified element. */
    public boolean remove(int val) {
        if(!map.containsKey(val)) return false;

        int indexA = map.get(val);

        if(indexA != list.size() - 1) swap(list, map, indexA, li
st.size() - 1);

        map.remove(list.get(list.size() - 1));
```

```
        list.remove(list.size() - 1);

        return true;
    }

    /** Get a random element from the set. */
    public int getRandom() {
        return list.get(rd.nextInt(list.size()));
    }

    // a : list index of element a
    // b : list index of element b
    private void swap(List<Integer> list, Map<Integer, Integer>
map, int a, int b){
        int valA = list.get(a);
        int valB = list.get(b);
        int indexA = map.get(valA);
        int indexB = map.get(valB);

        list.set(a, valB);
        list.set(b, valA);
        map.put(valA, indexB);
        map.put(valB, indexA);
    }
}
```

Insert Delete GetRandom O(1) - Duplicates allowed

- **HashMap** 的 **value** 只存一个 **index** 就不够用了，得存一个 **Set<>**，记录相同元素所有的 **index**，反正 **add / remove** 的平均复杂度都是 **O(1)**。

```
import java.util.*;

public class RandomizedCollection {
    Map<Integer, Set<Integer>> map;
    List<Integer> list;
    Random rand;
```

```
    /** Initialize your data structure here. */
    public RandomizedCollection() {
        map = new HashMap<>();
        list = new ArrayList<>();
        rand = new Random();
    }

    /** Inserts a value to the collection. Returns true if the c
ollection did not already contain the specified element. */
    public boolean insert(int val) {
        boolean flag = false;

        if(!map.containsKey(val)){
            flag = true;
            map.put(val, new HashSet<Integer>());
        }

        list.add(val);
        map.get(val).add(list.size() - 1);

        return flag;
    }

    /** Removes a value from the collection. Returns true if the
 collection contained the specified element. */
    public boolean remove(int val) {
        boolean flag = false;

        if(map.containsKey(val) && map.get(val).size() > 0){
            flag = true;
            int indexA = map.get(val).iterator().next();
            int indexB = list.size() - 1;
            swap(map, list, indexA, indexB);

            map.get(val).remove(list.size() - 1);
            list.remove(list.size() - 1);
        }

        return flag;
    }
```

```
    /** Get a random element from the collection. */
    public int getRandom() {
        return list.get(rand.nextInt(list.size()));
    }

    // Swap elements at list index "a" and "b" in arraylist, and
 hashmap
    private void swap(Map<Integer, Set<Integer>> map, List<Integ
er> list, int indexA, int indexB){
        // O(1)
        int valA = list.get(indexA);
        int valB = list.get(indexB);

        // O(1) average
        map.get(valA).remove(indexA);
        map.get(valB).remove(indexB);

        // O(1) average
        map.get(valA).add(indexB);
        map.get(valB).add(indexA);

        // O(1)
        list.set(indexA, valB);
        list.set(indexB, valA);

    }
}
```

(G) Implement HashTable with get,set,delete,getRandom functions in O(1).

这个也是平均复杂度，所用的技巧和上一题完全一样，ArrayList + HashMap

(G) Get random number except blacklist

http://www.1point3acres.com/bbs/thread-78548-1-1.html

设计一个随机数产生器，有一个以列表形式保存的已经排序blacklist，输出的数字如果出现在其中就要剔除。（面试完之后，听人提醒这题很类似CTCI上的问题12.3） 这题面试官没让我写code，我说了几个答案他都觉得不满意，因为他想要一种“渐进的”方法解决此题

题目要求:

给定区间 [0，n]，和一个排好序的 blacklist，List<> 随机返回一个不在 blacklist 中的数。

分析：

这道题最简单的思路是先从 blacklist 的角度想。先生成一个[0, n] 之间的数，然后看这个数在不在 blacklist 里，如果在的话，再生成一次。

- 问题一：如果 **blacklist** 不是 **set** 的形式给的，至少要 **binary search;**
- 问题二：运气不好，或者 **blacklist** 非常大的话，我们可能要调用多次 **getRandom**，而这个 **API** 可能是非常贵的。

因此另一个思考角度是，从 **white list** 出发。

对于【0，10】，blacklist = 【1,2,3,7,8】来讲，white list = 【0,4,5,6,9,10】.因此假设我们有足够内存去维护 whitelist，我们可以直接生成并返回一个 white list 中的元素，这样可以保证一次 getRandom() 操作保证得到想要的元素。

假如 **white list** 内存放不下呢？

方法照旧。每次我们生成一个 white list 的合理 index 之后，我们要找的就是 “第 index + 1 个不在 blacklist 中的数”。这个可以通过扫 blacklist 实现，O(1) 的内存开销。

相当于实现了一个 **index -> whitelist[index]** 的 **mapping**，可以保证一次 **getRandom()** 就可以得到有效元素。

- **i 从 0 到 n 循环，如果大于 blacklist 最后一个数，或者小于 blacklist 当前数，都 count --;**

- 否则 **blackPtr ++ ;**
- 每次循环的时候，**count** 扣完了，当前 **i** 就是目标元素。

```
    static Random rand = new Random();

    public static int getRandom(int n, List<Integer> blackList){
        int totalNum = n + 1;
        int whiteListSize = totalNum - blackList.size();

        int index = rand.nextInt(whiteListSize);
        int count = index + 1;
        int blackPtr = 0;
        for(int i = 0; i <= n; i++){
            if(i > blackList.get(blackList.size() - 1) || i < bl
ackList.get(blackPtr)){
                count --;
            } else {
                blackPtr ++;
            }

            if(count == 0) return i;
        }

        return -1;
    }

    public static void main(String[] args){
        List<Integer> blackList = new ArrayList<Integer>();
        blackList.add(1);blackList.add(2);blackList.add(3);
        blackList.add(7);blackList.add(8);
        blackList.add(13);blackList.add(19);blackList.add(20);
        for(int i = 0; i < 50; i++){
            System.out.print(" , " + getRandom(20, blackList));
        }
    }
```

huixuan : 我之前还想过这个问题可以变形成另外一个问题：比如产生长度为**k**的随机序列，序列中数字的值在**[0, n]**中**sample**，但是不可以有重复。要求讨论两个**case: k**和**n**差不多大、**k**远小于**n**

Random number generator with weight

题目描述：给定几种元素，以及对应的 weight，按每个元素的概率分布符合其 weight 的方式生成随机元素。

如：【0,1,2,3,4】 的 weights 【0.1, 0.2, 0.2, 0.4, 0,1】 ，按 weight 返回 【0,4】

核心是 **cumulative distribution.** 计算出 **[0.1, 0.3, 0.5, 0.9, 1]** 的 **cdf** 之后，我们可以随机返回一个 **【0,1】** 之间的数，然后对于 **cdf array** 做 **binary search.**

我们要寻找的，就是第一个 **random number < cdf[i]** 所对应的元素 **num[i].**

Reservoir Sampling (Quora)

题目描述： **Given a data set of unknown size N, uniformly select k elements from the set such that each element has a 1/N probability of being chosen.**

http://www.geeksforgeeks.org/reservoir-sampling/

分析 **&** 证明：

- 核心是 **proof by induction.**
- 从 **k = 1** 开始，对于第 **i** 个元素，我们有 **1 / i** 的几率让当前元素成为 **candidate**，相应的，有 **(i - 1) / i** 的概率直接扔掉这个新元素。

- 假设我们有 **3** 个元素，那么第 **3** 个元素只会被 **check** 一次，有 **1/3** 的几率留下，**2/3** 的几率被扔掉；
- 而它的前一个元素，**2**，会被 **check** 两次，一次是自己，一次是 **3** 的时候；而它成为最终选中的元素需要同时满足两个条件：
  - 考虑到自己的时候，随到了 **keep : (1/i = 1/2** 概率**)**
  - 考虑后面元素的时候，没替换掉它，即后面元素都 **discard** 了：**(2/3** 概率**)**
  - 两项相乘，等于 **1/3.**
- 这个 **induction** 的推导，对于任意有效 **k** 和 **n** 也都适用，从后往前。
  - 倒数第二个元素选中的概率 **= (k / n - 1) * (n - 1 / n) = k / n**

O(n) 时间，**O(k)** 空间。算法和测试代码如下：

```
    static Random rand = new Random();

    public static int[] reserviorSampling(int k, int[] nums){
        if(k >= nums.length) return nums;

        int i = 0;
        int[] rst = new int[k];
        for(; i < k; i++){
            rst[i] = nums[i];
        }

        for(; i < nums.length; i++){
            // random is exclusive
            int num = rand.nextInt(i + 1);
            if(num < k) rst[num] = nums[i];
        }

        return rst;
    }

    public static void main(String[] args){
        int[] count = new int[10];
        for(int i = 0; i < 10000; i++){
            int[] sampled = reserviorSampling(5, new int[]{0,1,2,
3,4,5,6,7,8,9});
            for(int num : sampled) count[num]++;
        }

        for(int i = 0; i < 10; i++){
            System.out.println("Count of " + i + " " + count[i]
+ " times");
        }
    }
```

Linked List Random Node

刚写完这章的总结，leetcode 就搞了个蓄水池抽样的题出来。。好与时俱进啊！

```
import java.util.*;

public class Solution {
    ListNode head;
    ListNode rst;
    int count;
    Random rand;

    /** @param head The linked list's head. Note that the head i
s guanranteed to be not null, so it contains at least one node.
*/
    public Solution(ListNode head) {
        this.head = head;
        rand = new Random();
    }

    /** Returns a random node's value. */
    public int getRandom() {
        ListNode cur = head.next;
        rst = head;
        count = 2;
        while(cur != null){
            int num = rand.nextInt(count++);
            if(num == 0) rst = cur;
            cur = cur.next;
        }
        return rst.val;
    }
}
```

Shuffle an Array

Shuffle array, Fisher–Yates shuffle, Knuth 洗牌算法

代码和流程惊人的简单：

- 遍历每个 **i** ，随机从 **i** 还有 **i** 后面的区间抽一个数，和 **i** 交换；
- 重点是 **index =** 【**i, n - 1**】区间。每个元素有留在原位置的概率。

其原理非常类似于一群人在一个黑箱子里抽彩票，**for** 循环中的每次迭代相当于一个人，每次抽样范围 **index =** 箱子中剩下的所有彩票，其数量依次递减。到最后每个人虽然抽奖时箱子里彩票总数不同，但是获奖概率却是均等的。

```
import java.util.*;
public class Solution {
    int[] original;
    Random rand;
    public Solution(int[] nums) {
        original = nums;
        rand = new Random();
    }

    /** Resets the array to its original configuration and retur
n it. */
    public int[] reset() {
        return original;
    }

    /** Returns a random shuffling of the array. */
    public int[] shuffle() {
        int[] rst = Arrays.copyOf(original, original.length);

        for(int i = 0; i < rst.length; i++){
            int index = rand.nextInt(rst.length - i) + i;

            int temp = rst[i];
            rst[i] = rst[index];
            rst[index] = temp;
        }

        return rst;
    }
}
```

7/28, FB, I/O Buffer

Read N Characters Given Read4

这题的 API 和题目描述都有点模糊，重新理一下：

- int read4(char[] buf)
- 你扔一个 char[] 过去，函数写入最多 4 个 char 在上面，然后返回写入 char 的长度； 到文件末尾的时候返回 char 长度会小于 4.
- 于是我们的目标是把最终长度为 n 的字符串写入自己函数的 read(char[] buf) 里，然后返回实际长度。

主要的注意点就是，有的时候 **read4** 是短板，有的时候 **readN** 自己是短板**(**不需要**4**个那么多的字符**)**，在写入的时候要注意处理下。

```
    public int read(char[] buf, int n) {
        // file has less than n chars
        // n is not divisible by 4

        boolean EOF = false;
        char[] temp = new char[4];
        int curPtr = 0;

        while(curPtr < n && !EOF){
            int charCount = read4(temp);
            EOF = (charCount < 4);

            for(int i = 0; i < charCount && curPtr < n; i++){
                buf[curPtr++] = temp[i];
            }
        }

        return curPtr;
    }
```

Read N Characters Given Read4 II - Call multiple times

多次调用之后，这题的难点就变成了“如何处理剩余字符”。因为一次 **call** 拿到的字符很可能超过我们实际需要的，这时候就需要依赖外部 **buffer** 记录下来，每次新 **read() call** 的时候，先从缓存里拿。

两次 AC，但是这个代码的简洁性实在不忍直视。。。待我好好改改。。

```
public class Solution extends Reader4 {
    char[] leftOver = new char[4];
    int leftCount = 0;
    int leftPtr = 0;
    /**
     * @param buf Destination buffer
     * @param n   Maximum number of characters to read
```

```
     * @return    The number of characters read
     */
    public int read(char[] buf, int n) {
        int totalLen = 0; // always points to next pos to be fil
led
        boolean EOF = false;
        char[] temp = new char[4];
        while(!EOF && totalLen < n){
            while(leftCount > 0 && totalLen < n){
                buf[totalLen++] = leftOver[leftPtr++];
                leftCount--;
            }

            if(totalLen < n){
                int newLen = read4(temp);
                EOF = newLen < 4;

                if(totalLen + newLen <= n){
                    // did not exceed
                    for(int i = 0; i < newLen; i++) buf[totalLen
++] = temp[i];

                } else {
                    // points to next extra char
                    int ptr = 0;
                    // write needed char into buffer
                    while(totalLen < n) buf[totalLen++] = temp[p
tr++];
                    // write extra char into leftover
                    int size = newLen - ptr;
                    for(int i = 0; i < size; i++){
                        leftOver[i] = temp[ptr++];
                        leftCount = i + 1;
                    }
                    leftPtr = 0;
                }
            }
        }

        return totalLen;
```

```
    }
}
```

这个写法就巧妙简洁多了，而且原封不动就可以做原来的问题，非常巧妙。

- 我们只用一个 **buffer**，大小为 **4.** 因为一次读取的字符数量不会超过 **4**，而且每次 **read()** 之后剩余的字符数量也不会超过 **4.**
- 因此这一个全局 **buffer**，就代表了我们全部的可用字符，一个 **int** 存数量，一个 **int** 存起点。
- 需要写入 **buffer** 的时候就写，如果没有了，就向 **read4** 要，到时候 **ptr** 会自动记录下一轮的起始位置和剩余字符串数量。

```java
public class Solution extends Reader4 {
    /**
     * @param buf Destination buffer
     * @param n   Maximum number of characters to read
     * @return    The number of characters read
     */

    char[] temp = new char[4];
    int bufPtr = 0;
    int bufCount = 0;
    int bufEnd = 0;

    public int read(char[] buf, int n) {
        int curPtr = 0;
        while(curPtr < n){
            if(bufCount == 0){
                bufCount = read4(temp);
                bufEnd = bufCount;
                bufPtr = 0;
            }
            if(bufCount == 0) break;

            while(bufPtr < bufEnd && curPtr < n){
                buf[curPtr++] = temp[bufPtr++];
                bufCount --;
            }
        }
        return curPtr;
    }
}
```

7/28, FB, Simplify Path, H-Index I & II

Simplify Path

没啥特别好说的~ 用双端队列比 stack 省事很多。

收尾的时候注意下，如果字符串为空的话，需要留个 "/" 作为默认情况。

```
public class Solution {
    public String simplifyPath(String path) {
        String[] ops = path.split("/");
        Deque<String> deque = new LinkedList<String>();

        for(String str : ops){
            if(str.equals(".") || str.equals("")) continue;
            if(str.equals("..")) deque.pollFirst();
            else deque.offerFirst(str);
        }

        StringBuilder sb = new StringBuilder();

        while(!deque.isEmpty()){
            String str = deque.pollLast();
            sb.append('/');
            sb.append(str);
        }

        return (sb.length() == 0) ? "/" : sb.toString();
    }
}
```

H-Index

这题有官方答案

从图的方式理解的话，**h-index** 是图中最大正方形的边长。

h-index 的定义是，至少有 **h** 篇 **paper** 的 **citation** 数都大于等于 **h.**

对于给定 **array, h-index** 有且只有一个。

因此对于 n 篇 paper 的输入， h-index 的值一定在【0，n】区间。 一个需要特别考虑的例子是下面这个：

[0,0,4,4] 的 h-index 是 2.

一开始尝试了下排序然后从右往左扫，然而遇到上面那个 case 就比较麻烦，因为数组给定的 citations 之间间隔可能很大，而这种循环做法只考虑 citation 的值作为切分点是不对的。

但是排序 + 遍历依然可以做，代码如下：

```
public class Solution {
    public int hIndex(int[] citations) {
        if(citations == null || citations.length == 0) return 0;
        // Maxium hindex is n, number of papers
        int n = citations.length;
        Arrays.sort(citations);
        int hindex = 0;

        for(int i = n - 1; i >= 0; i--){
            if(citations[i] > hindex) hindex++;
        }

        return hindex;
    }
}
```

正确的做法是从 **N** 到 **0** 递减遍历一遍，才能保证得到的 **h-index** 是最大的，但是我们需要在扫描过程中知道对于每一个切分点 **val** 来讲，到底有多少个 **>= val** 的 **paper.**

换句话说，这是一种类似 “后缀和” 的计数算法，先记录所有可能的切分点；如果一个 **paper** 的引用次数大于 **N** ，则直接记录在 **count[n]** 上，代表 “我不管到底是什么，反正大于 **N** 了”，其他位置各自计算。然后从后向前扫，依次记录后缀和即可。

```
public class Solution {
    public int hIndex(int[] citations) {
        if(citations == null || citations.length == 0) return 0;
        // Maxium hindex is n, number of papers
        int n = citations.length;
        int[] counts = new int[n + 1];
        for(int i = 0; i < n; i++){
            if(citations[i] > n) counts[n]++;
            else counts[citations[i]]++;
        }

        int biggerCount = 0;
        for(int index = n; index >= 0; index--){
            biggerCount += counts[index];
            if(biggerCount >= index) return index;
        }

        return 0;
    }
}
```

H-Index II

试着用 left + 1 < right 的 binary search 搞这题，搞了好半天才 AC .. corner case 一大堆。。

之前用 left + 1 < right 是相对于“元素 index” 来讲的，为的是避免越界。这里面既然我们已知最终 h-index 区间 [0, N] 了，可以直接以这两点开始，以 pointer 重合结束。

需要注意的细节是，这里要做 **left = mid + 1** ，而不是 **right = mid - 1**，不然在 **[0]** 的 **testcase** 上会 **TLE.**

同时用 **mid + 1** 和 **mid - 1** 会在 **[1,2]** 的情况下 **WA.** 所以指针的推动和错位是要根据题意制定的；这题我们唯一能确定的只有两个情况：

- **c[mid] == N - mid** ，当前位置就是完美正方形，一定是解；
- **c[mid] < N - mid**，当前位置的 **bar** 太低，正解要在右边找，而且一定不会是 **mid** 的位置，**mid + 1;**

```
public class Solution {
    public int hIndex(int[] citations) {
        if(citations == null || citations.length == 0) return 0;

        int N = citations.length;
        int left = 0;
        int right = N;

        while(left < right){
            int mid = left + (right - left) / 2;

            if(citations[mid] == N - mid){
                return N - mid;
            } else if(citations[mid] < N - mid){
                left = mid + 1;
            } else {
                right = mid;
            }
        }

        return N - left;
    }
}
```

7/29, FB, Excel Sheet, Remove Duplicates

Excel Sheet Column Title

比较简单，一次 AC. 这个代码不算是最简洁的，但是面试将近，我的代码风格开始向“容易看懂 + 好交流 + 失误率低” 靠拢。。

26 - Z

52 - AZ

1000 - ALL

这里面的 'Z' 其实相当于 N 进制里面的 0；代表着 place holder.

可以看到填充顺序其实是从后往前，每一步取 % 26 的余数。只有 26 的倍数比较特殊，余数为 0 的时候，append Z 并且要减去 26，其他时候用 offset 就好了。

```java
public class Solution {
    public String convertToTitle(int n) {
        if(n <= 0) return "";

        StringBuilder sb = new StringBuilder();
        while(n > 0){
            int digit = n % 26;

            if(digit == 0){
                sb.append('Z');
                n -= 26;
            } else {
                char chr = (char)('A' + digit - 1);
                sb.append(chr);
            }

            n = n / 26;

        }
        return sb.reverse().toString();
    }
}
```

Excel Sheet Column Number

掌握了这个特性之后，解码就变得非常容易了。。

```java
public class Solution {
    public int titleToNumber(String s) {
        int num = 0;
        for(int i = 0; i < s.length(); i++){
            num *= 26;
            num += (int) s.charAt(i) - 'A' + 1;
        }
        return num;
    }
}
```

Remove Duplicates from Sorted Array II

60秒 AC .. 这么水题我都有点不好意思了。。。

```
public class Solution {
    public int removeDuplicates(int[] nums) {
        if(nums == null || nums.length == 0) return 0;
        if(nums.length <= 2) return nums.length;

        int ptr = 2;
        for(int i = 2; i < nums.length; i++){
            if(nums[i] == nums[ptr - 1] && nums[i] == nums[ptr -
2]) continue;
            else nums[ptr++] = nums[i];
        }
        return ptr;
    }
}
```

Remove Duplicates from Sorted List II

也挺简单的，但是多了几个细节：

- 这次是只要是 **duplicate** 就全跳过；
- 因此看到一截重复的就全部跳过，但同时要注意不要假设一段重复的后面不会接着出现另外一段；所以处理完一段直接 **continue;**
- **dummy node** 不能无脑连 **head** ，没准 **head** 里面的一个都不取，所以就建个 **dummy** 之后 **next** 先空着，自己一个一个加上去；
- 从 **dummy** 出发的 **cur** 连完了之后，要记得循环结束时切断尾巴，因为我们在循环中只跳过，没切断，最后的收尾要自己做一下。

```
public ListNode deleteDuplicates(ListNode head) {
    ListNode dummy = new ListNode(0);
    ListNode cur = dummy;
    while(head != null){
        if(head.next != null && head.val == head.next.val){
            int duplicateVal = head.val;
            while(head != null && head.val == duplicateVal)
head = head.next;

            // Can't assume next value we see isn't duplicat
e as well;
            continue;
        }

        cur.next = head;
        cur = cur.next;
        head = head.next;
    }
    // Need to cut the tail
    cur.next = null;

    return dummy.next;
}
```

Integer 的构造，操作，序列化

Integer to English Words

- 小于 **20** 的数要字典**;**
- 十几 **Tens** 的，需要字典；
- 多少个 **thousands** 的，需要字典，从右往左 **index** 递增；
- 以三位数为单位处理，任何三位数都可以用 **helper function + 字典解决，自带 hundred** 单位。
- **0** 在所有情况都代表空字符，除了 **num** 一开始就等于 **0** 的情况要返回 **"Zero".**

自己第一遍 AC 的版本太粗糙，就不放了。

这个版本就简洁了很多，从右向左，递归调用处理三位数的情况；

```
public class Solution {
    private final String[] LESS_THAN_20 = {"", "One", "Two", "Th
ree", "Four", "Five", "Six", "Seven", "Eight", "Nine", "Ten", "E
leven", "Twelve", "Thirteen", "Fourteen", "Fifteen", "Sixteen",
"Seventeen", "Eighteen", "Nineteen"};
    private final String[] TENS = {"", "Ten", "Twenty", "Thirty"
, "Forty", "Fifty", "Sixty", "Seventy", "Eighty", "Ninety"};
    private final String[] THOUSANDS = {"", "Thousand", "Million"
, "Billion"};

    public String numberToWords(int num) {
        if(num == 0) return "Zero";
        String rst = "";
        int highPtr = 0;
        while(num != 0){
            if(num % 1000 != 0){
                rst = helper(num % 1000) + THOUSANDS[highPtr] +
" " + rst;
            }
            num /= 1000;
            highPtr ++;
        }
        return rst.trim();
    }

    private String helper(int num){
        if(num == 0)
            return "";
        else if(num < 20)
            return LESS_THAN_20[num] + " ";
        else if(num < 100)
            return TENS[num / 10] + " " + helper(num % 10);
        else
            return LESS_THAN_20[num / 100] + ' ' + "Hundred" + '
 ' + helper(num % 100);
    }
}
```

Roman to Integer

Trivial problem. 没啥好讲的。。

```java
public class Solution {
    public int romanToInt(String s) {
        if(s.length() == 0) return 0;

        int num = getNum(s.charAt(0));
        int prev = num;
        for(int i = 1; i < s.length(); i++){
            int cur = getNum(s.charAt(i));
            if(cur > prev){
                num -= 2 * prev;
            }
            prev = cur;
            num += cur;
        }

        return num;
    }

    private int getNum(char chr){
        switch(chr){
            case 'I':
                return 1;
            case 'V':
                return 5;
            case 'X':
                return 10;
            case 'L':
                return 50;
            case 'C':
                return 100;
            case 'D':
                return 500;
            case 'M':
                return 1000;
            default:
                return 0;
        }
    }
}
```

Count and Say

这题怪怪的，我也不知道到底想考啥。。难道是迭代和递归么。。。。

```
public class Solution {
    public String countAndSay(int n) {
        if(n == 1) return "1";
        StringBuilder sb = new StringBuilder();
        sb.append("1");

        for(int i = 1; i < n; i++){
            String str = sb.toString();
            sb.setLength(0);
            int index = 0;
            while(index < str.length()){
                int num = str.charAt(index) - '0';
                int count = 1;
                while(index < str.length() - 1 && str.charAt(ind
ex) == str.charAt(index + 1)){
                    count ++;
                    index ++;
                }
                index ++;
                sb.append(count);
                sb.append(num);
            }
        }

        return sb.toString();
    }
}
```

Integer to Roman

之前那道 Roman to Integer 就弄了个 switch case ，这次可能性稍微多了点，直接开两个 1-1 onto mapping 当表查好了。

- 当可能的情况“有限”并“可数”的时候，可以自己用 **array** 去建 **1-1 mapping** 便于查询。

```
public class Solution {
    public String intToRoman(int num) {
        int[] nums =      {1000,  900,  500,  400,  100,   90,
50,   40,   10,    9,   5,    4,   1};
        String[] romans = { "M", "CM",  "D", "CD",  "C", "XC", "
L", "XL",  "X", "IX", "V", "IV", "I"};

        StringBuilder sb = new StringBuilder();
        while(num > 0){
            for(int i = 0; i < nums.length; i++){
                if(num >= nums[i]){
                    num -= nums[i];
                    sb.append(romans[i]);
                    break;
                }
            }
        }

        return sb.toString();
    }
}
```

Integer to English Words

- 小于 **20** 的数要字典;
- 十几 **Tens** 的，需要字典；
- 多少个 **thousands** 的，需要字典，从右往左 **index** 递增；
- 以三位数为单位处理，任何三位数都可以用 **helper function + 字典**解决，自带 **hundred** 单位。

- **0** 在所有情况都代表空字符，除了 **num** 一开始就等于 **0** 的情况要返回 **"Zero".**

自己第一遍 AC 的版本太粗糙，就不放了。

这个版本就简洁了很多，从右向左，递归调用处理三位数的情况；

```
public class Solution {
    private final String[] LESS_THAN_20 = {"", "One", "Two", "Th
ree", "Four", "Five", "Six", "Seven", "Eight", "Nine", "Ten", "E
leven", "Twelve", "Thirteen", "Fourteen", "Fifteen", "Sixteen",
"Seventeen", "Eighteen", "Nineteen"};
    private final String[] TENS = {"", "Ten", "Twenty", "Thirty"
, "Forty", "Fifty", "Sixty", "Seventy", "Eighty", "Ninety"};
    private final String[] THOUSANDS = {"", "Thousand", "Million"
, "Billion"};

    public String numberToWords(int num) {
        if(num == 0) return "Zero";
        String rst = "";
        int highPtr = 0;
        while(num != 0){
            if(num % 1000 != 0){
                rst = helper(num % 1000) + THOUSANDS[highPtr] +
" " + rst;
            }
            num /= 1000;
            highPtr ++;
        }
        return rst.trim();
    }

    private String helper(int num){
        if(num == 0)
            return "";
        else if(num < 20)
            return LESS_THAN_20[num] + " ";
        else if(num < 100)
            return TENS[num / 10] + " " + helper(num % 10);
        else
            return LESS_THAN_20[num / 100] + ' ' + "Hundred" + '
 ' + helper(num % 100);
    }
}
```

Frequency 类问题

Top K Frequent Elements

速度尚可，超过65.15%

```
public class Solution {
    private class Node implements Comparable<Node>{
        int key;
        int frequency;
        public Node(int key, int frequency){
            this.key = key;
            this.frequency = frequency;
        }
        public int compareTo(Node a){
            return a.frequency - this.frequency;
        }
    }

    public List<Integer> topKFrequent(int[] nums, int k) {
        PriorityQueue<Node> maxHeap = new PriorityQueue<Node>();
        HashMap<Integer, Node> map = new HashMap<>();
        List<Integer> list = new ArrayList<>();

        for(int i = 0; i < nums.length; i++){
            if(!map.containsKey(nums[i])){
                map.put(nums[i], new Node(nums[i], 1));
            } else {
                Node node = map.get(nums[i]);
                node.frequency ++;
                map.put(nums[i], node);
            }
        }

        for(Integer key : map.keySet()){
            maxHeap.offer(map.get(key));
```

```
        }

        for(int i = 0; i < k; i++){
            list.add(maxHeap.poll().key);
        }

        return list;
    }
}
```

大数据情况下， top K frequent

Missing Number 类，元素交换，数组环形跳转

Move Zeros

FB 的超高频，记得先演一演，给个 swap 的写法~

这种写法中，我们调用 swap 的次数取决于从数组第一个 0 开始，后面到底有多少个非 0 元素。

一次 swap = 2次数组写入 + 1次 temp variable 写入，总写入次数为 O(3n)，总数组写入次数为 O(2n).

```
public class Solution {
    public void moveZeroes(int[] nums) {
        if(nums == null || nums.length == 0) return;
        int ptr = 0;
        while(ptr < nums.length && nums[ptr] != 0) ptr ++;
        for(int i = ptr + 1; i < nums.length; i++){
            if(nums[i] != 0) swap(nums, i, ptr++);
        }
    }

    private void swap(int[] nums, int a, int b){
        int temp = nums[a];
        nums[a] = nums[b];
        nums[b] = temp;
    }
}
```

然后 follow-up 肯定是如何减少写入次数 (aka 别用 swap)，后面的演技就是这样……

其实我一直觉得这个写法才是最直观的，暴力点用 swap 的写法反而 tricky 一点。。

这种写法中，数组每一个位置一定会被不多不少写入一次，写入次数为 **O(n)**

```
    public void moveZeroes(int[] nums) {
        if(nums == null || nums.length == 0) return;
        int ptr = 0;
        for(int i = 0; i < nums.length; i++){
            if(nums[i] != 0) nums[ptr++] = nums[i];
        }
        for(; ptr < nums.length; ptr++){
            nums[ptr] = 0;
        }
    }
```

Missing Number

等差数列。 英文叫 arithmetic sequence，记一下，和老外面试官吹比用。。

```
public class Solution {
    public int missingNumber(int[] nums) {
        int N = nums.length;
        int sum = 0;
        for(int num : nums) sum += num;
        return  (N + 1) * N / 2 - sum;
    }
}
```

First Missing Positive

这题挺 tricky 的，到现在也是记的九章的答案。

思路就是不断试图把当前元素替换到其正确的位置上；如果目标位置元素已经正确则 **do nothing**，如果目标位置越界，也 **do nothing.** 每次交换之后指针位置不动，继续试图新一轮替换。这样一次循环之后，所有能被放到正确位置的元素，都会被放过去，而第一个位置和值不匹配的元素就是我们要寻找的目标。

注意要点：

- 当前元素的目标位置 **index** 有可能越上下界，多加两个判断条件；
- 如果真是 **swap** 了，不代表当前元素一定是正确位置，指针不要动(或者说在 **for loop** 里面要 **i--**)

```
public class Solution {
    public int firstMissingPositive(int[] nums) {
        for(int i = 0; i < nums.length; i++){
            if(nums[i] != i + 1 && nums[i] - 1 >= 0 && nums[i] -
1 < nums.length && nums[nums[i] - 1] != nums[i]){
                swap(nums, i, nums[i] - 1);
                i--;
            }
        }
        for(int i = 0; i < nums.length; i++){
            if(nums[i] != i + 1) return i + 1;
        }
        return nums.length + 1;
    }

    private void swap(int[] nums, int a, int b){
        int temp = nums[a];
        nums[a] = nums[b];
        nums[b] = temp;
    }
}
```

Missing Ranges

研究区间类问题的时候做过一次，挺有意思的。

这题的要点在于“不断更新 **lower**”，并且根据个新元素带来的区间终点决定是添加一个元素，还是一个区间。

```
public class Solution {
    public List<String> findMissingRanges(int[] nums, int lower,
int upper) {
        List<String> list = new ArrayList<>();
        for(int i = 0; i < nums.length; i++){
            int candidate = nums[i] - 1;

            if(candidate == lower) list.add("" + candidate);
            else if(candidate > lower) list.add("" + lower + "->"
 + candidate);

            lower = Math.max(lower, nums[i] + 1);
        }

        if(lower == upper) list.add("" + lower);
        else if(lower < upper) list.add("" + lower + "->" + uppe
r);

        return list;
    }
}
```

Find the Duplicate Number

这题是一个道理，我们依然用 first missing positive 的做法，把每个数试图换到它对应的位置上；然后 nums 数组最后的那个元素，一定是 duplicate element，因为其他元素都归位之后，它没地方去。因为如果最后一个元素是唯一元素的话，它一定会被置换到正确位置上。

好吧，写完了我才发现不让修改原数组= =

```
public class Solution {
    public int findDuplicate(int[] nums) {
        for(int i = 0; i < nums.length; i++){
            if(nums[i] != i + 1 && nums[i] - 1 >= 0 && nums[i] -
1 < nums.length && nums[nums[i] - 1] != nums[i]){
                swap(nums, i, nums[i] - 1);
                i--;
            }
        }

        return nums[nums.length - 1];
    }

    private void swap(int[] nums, int a, int b){
        int temp = nums[a];
        nums[a] = nums[b];
        nums[b] = temp;
    }
}
```

这题还有另一种很妖孽的写法，就是把它当成 LinkedList 做。。

https://discuss.leetcode.com/topic/25913/my-easy-understood-solution-with-o-n-time-and-o-1-space-without-modifying-the-array-with-clear-explanation

那么这道题又跟Find the Duplicate Number有什么关系呢？ 在table中每个元素存储的是table index，根据其跳转，其实就是把table中的元素作为指针来使用。就像我们在上题中也是在child node中存parent node的index，并做跳转。 我们结合Find the Duplicate Number这道题来看。这道题讨论区most voted的解法即是一个国人大神把它转化为了Linked List Cycle来做。 find linked list cycle是一道大家都很熟悉的题。而国人大神的解法就是把given array中的每一个元素当作指针来用跳转到下一个元素。 我们用A[1, 4, 3, 2, 5, 2, 6]来做例：

当我们扫到A[0]时，A[0] == 1，把它作为指针来使用，我们跳到A[1];

A[1] == 4, 跳到A[4];

A[4] == 5, 跳到A[5];

A[5] == 2, 跳到A[2];

A[2] == 3, 跳到A[3];

A[3] == 2, 开始循环...

这道题之所以可以把元素作为指针来使用当然也是题干出的巧。

这里要注意初始化问题，**fast** 要初始化成 **nums[nums[0]]**，不然死循环。

```
public class Solution {
    public int findDuplicate(int[] nums) {
        if (nums.length > 1)
        {
            int slow = nums[0];
            int fast = nums[nums[0]];
            while (slow != fast)
            {
                slow = nums[slow];
                fast = nums[nums[fast]];
            }

            fast = 0;
            while (fast != slow)
            {
                fast = nums[fast];
                slow = nums[slow];
            }
            return slow;
        }
        return -1;
    }
}
```

(G面经)

然后是一道关于按照固定的顺序重新排序，多少次排序才能回到初始顺序的题目，我们总是把第**x**个位置的元素移到第**y**个位置，比如【**0**，**1**，**2**，**3**】就是完全不换顺序，一次排序就回去了；【**3**，**2**，**1**，**0**】是位置**3**和位置**0**互换，位置**2**和位置**1**互换，两次排序就能回到原顺序；而【**1**，**2**，**3**，**0**】就是每次把一个位置上的元素往后移动一个位置，那显然转一圈就回来了。

举个例子，原来的input为[20, 34, 7, 9], 排序要求为[1, 3, 2, 0] 一次排序后： [20, 34, 7, 9] -> [34, 9, 7, 20] 二次排序后: [34, 9, 7, 20] -> [9, 20, 7, 34] 三次排序后：[9, 20, 7, 34] -> [20, 34, 7, 9] (此时和初始顺序一致，所以是用了三个iteration恢复原样)

排序规则[1, 3, 2, 0]的意思是把在位置1的元素排到第0个位置，位置3的元素排到第2个位置，位置2的元素还是在位置2，位置0的元素放到位置3去。 排序规则不变，每次只是Input被按照规则改变了顺序。

这个小技巧也可以用在原帖第二轮第二题上。

原帖中Heliuhun对于这道题给了一个很精妙的解法。我们来分析一下这个解法。 用A[a1, a2, a3, a4, a5], B[1,0,4,2,3]做例: 我们并不在乎A中的每个数到底是什么。我们想知道的是A中的每个数按照B来移动，最少经过多少次可以回到原位。 对于a2来说，它经历一次移动从1来到了0的位置。再经历一次移动从0回到了1的位置。 对于a4来说，它经历一次移动从3来到了2的位置。再经历一次移动从2来到了4的位置。再经历一次移动从4回到了3的位置。 可以看到A中的元素是根据B来跳转的，并且把B中的元素当作指针来使用。B中的元素形成了一个或多个封闭的环状“linked list”。 按照Heliuhun的思路，我们需要找出B中有多少封闭的环状“linked list”，再找出它们的最小公倍数。 为什么是最小公倍数呢？ 拿上面的例子来说，当a2经过两次移动后回到原位时，a4还需要一次移动才能回到原位。最少经过6次移动，a2，a4以及a1，a3，a5才能同时回到原位。

跳转序列可以拆成几个独立的环，同时也知道每个环的长度，然后求所有环长度的最小公倍数？比如**[1,0,4,2,3]**就是**0-1-0** 和 **2-3-4-2**两个环，所以**6**次**shuffle**后变回原来的数组

Least common multiple = 最小公倍数

8/10, Google Tag

Group Shifted Strings

挺简单的，和 group anagram 异曲同工。

唯一需要注意的就是 "az" 和 "ba" 同组，说明字母表是环形的。两个字母之间的差如果为负数，就把 diff 加上 26. "az" diff = -25 + 26 = 1; "ba" diff = 1;

```
public class Solution {
    public List<List<String>> groupStrings(String[] strings) {
        List<List<String>> rst = new ArrayList<>();
        // Key : diff1,diff2... single character is ""
        // Value : list of strings grouped together
        HashMap<String, List<String>> map = new HashMap<>();
        StringBuilder sb = new StringBuilder();

        for(int i = 0 ; i < strings.length; i++){
            String str = strings[i];
            sb.setLength(0);
            for(int j = 0; j < str.length() - 1; j++){
                int diff = str.charAt(j) - str.charAt(j + 1);
                if(diff < 0) diff += 26;
                sb.append(diff);
                sb.append(',');
            }
            String key = sb.toString();
            if(!map.containsKey(key)) map.put(key, new ArrayList
<String>());
            map.get(key).add(str);
        }

        Iterator<String> iter = map.keySet().iterator();
        while(iter.hasNext()){
            rst.add(map.get(iter.next()));
        }

        return rst;
    }
}
```

Perfect Squares

一开始看错了，以为是相乘，搞了一堆因式分解数因数的。。还往数论上想了半天。

后来发现 BFS 就能 AC.

前两次提交都在大数上 MLE，说明 BFS 剪枝不到位。

- 扫合理 **moves** 的时候，先从大的扫；
- 用个 **hashset** 存一下已经访问过的 **sum** 值，避免重复；

以上两招都可以很简便的减少内存和计算时间开销。

```java
public class Solution {
    public int numSquares(int n) {
        // All perfect squares less than n
        List<Integer> moves = new ArrayList<>();
        for(int i = 1; i <= n; i++){
            if(i * i <= n) moves.add(i * i);
            else break;
        }
        Set<Integer> visited = new HashSet<>();
        Queue<Integer> queue = new LinkedList<>();
        int lvl = 0;
        queue.offer(0);
        while(!queue.isEmpty()){
            int size = queue.size();
            lvl ++;
            for(int i = 0; i < size; i++){
                int sum = queue.poll();
                for(int j = moves.size() - 1; j >= 0; j--){
                    int next = moves.get(j);

                    if(sum + next > n || visited.contains(sum +
next)){
                        continue;
                    } else if(sum + next == n){
                        return lvl;
                    } else {
                        visited.add(sum + next);
                        queue.offer(sum + next);
                    }
                }
            }

        }
        return -1;
    }
}
```

再仔细观察一下这题:

- 我们要凑出来一个和正好是 **n** 的选择组合；
- 能选的元素是固定数量的 **perfect square** (有的会超)
- 一个元素可以选多次；

这就是背包啊！

```
public class Solution {
    public int numSquares(int n) {
        // All perfect squares less than n
        int[] dp = new int[n + 1];
        Arrays.fill(dp, n + 1);
        dp[0] = 0;

        for(int i = 1; i <= n; i++){
            for(int j = 1; j * j <= n; j++){
                if(i - j * j >= 0) dp[i] = Math.min(dp[i], dp[i
- j * j] + 1);
            }
        }

        return dp[n];
    }
}
```

Next Permutation

流程：

- **start** 从倒数第二个数起，往左扫，寻找到第一个 **nums[start] < nums[start + 1]** 的位置；
- 从 **start** 往后找，找到第一个比 **nums[start]** 小的前一个数 (也就是最小的大于等于 **nums[start]** 的数)
- 交换 **start , end**

- 把 **start + 1** 后面的区间翻转，**over.**

1: 右往左，找下落；

2：左往右，找 **threshold (**换过来的数，怎么说也得比 **nums[start]** 大**)**

3：交换，反转。

```
public class Solution {
    public void nextPermutation(int[] nums) {
        if(nums == null || nums.length <= 1) return;

        int start = nums.length - 2;
        while(start >= 0 && nums[start] >= nums[start + 1]) star
t--;

        if(start >= 0){
            int end = start + 1;
            while(end < nums.length && nums[start] < nums[end])
end++;
            end--;
            int temp = nums[start];
            nums[start] = nums[end];
            nums[end] = temp;
        }

        reverse(nums, start + 1);
    }

    private void reverse(int[] nums, int index){
        int end = nums.length - 1;
        while(index < end){
            int temp = nums[index];
            nums[index] = nums[end];
            nums[end] = temp;

            index++;
            end--;
        }
    }
}
```

Strobogrammatic Number II

为什么一个这么简单的 DFS 能超过 89% ..

注意：**index == 0** 并且 **i == 0** 的时候要跳过，免得在起始位置填上 **0 .**

```
public class Solution {
    public List<String> findStrobogrammatic(int n) {
        List<String> list = new ArrayList<>();
        char[] num1 = {'0','1','8','6','9'};
        char[] num2 = {'0','1','8','9','6'};
        char[] number = new char[n];

        dfs(list, number, num1, num2, 0);

        return list;
    }

    private void dfs(List<String> list, char[] number, char[] nu
m1, char[] num2, int index){
        int left = index;
        int right = number.length - index - 1;

        if(left > right){
            list.add(new String(number));
            return;
        }
        // We can fill in 0,1,8 only
        if(left == right){
            for(int i = 0; i < 3; i++){
                number[left] = num1[i];
                dfs(list, number, num1, num2, index + 1);
            }
        } else {
            for(int i = 0; i < num1.length; i++){
                if(index == 0 && i == 0) continue;
                number[left] = num1[i];
                number[right] = num2[i];
                dfs(list, number, num1, num2, index + 1);
            }
        }
    }
}
```

Strobogrammatic Number III

Google 面经里的 follow-up 是，给定一个上限 n ，输出所有上限范围内的数。

办法土了点，遍历所有 lowLen ~ highLen 区间的长度，生成所有可能的结果，考虑到区间可能是大数，我们就改一下，自己写一个 String compare 函数好了。

后来发现有点多余，可以直接用内置的 str1.compareTo(str2).

超过 81.92% ~

```
public class Solution {
    int count = 0;
    public int strobogrammaticInRange(String low, String high) {
        int lowLen = low.length();
        int highLen = high.length();

        char[] num1 = {'0','1','8','6','9'};
        char[] num2 = {'0','1','8','9','6'};

        for(int i = lowLen; i <= highLen; i++){
            char[] number = new char[i];
            dfs(number, num1, num2, 0, low, high);
        }

        return count;
    }

    private void dfs(char[] number, char[] num1, char[] num2, int
 index, String low, String high){
        int left = index;
        int right = number.length - index - 1;

        if(left > right){
            String num = new String(number);
            if(compare(low, num) <= 0 && compare(num, high) <= 0
) count++;
            return;
        } else if(left == right){
            for(int i = 0; i < 3; i++){
```

```
                    number[left] = num1[i];
                    dfs(number, num1, num2, index + 1, low, high);
                }
            } else {
                for(int i = 0; i < 5; i++){
                    if(index == 0 && i == 0) continue;
                    number[left] = num1[i];
                    number[right] = num2[i];
                    dfs(number, num1, num2, index + 1, low, high);
                }
            }
        }

        // -1 : str1 is bigger
        // 1 : str 2 is bigger
        // 0 : equal
        private int compare(String str1, String str2){
            if(str1.length() > str2.length()) return 1;
            else if(str1.length() < str2.length()) return -1;
            else {
                for(int i = 0; i < str1.length(); i++){
                    int digit1 = str1.charAt(i) - '0';
                    int digit2 = str2.charAt(i) - '0';

                    if(digit1 != digit2) return (digit1 > digit2) ? 1
 : -1;
                }
            }
            // Equal
            return 0;
        }

    }
```

Sort Transformed Array

我很喜欢这题，挺有创意~

- 注意点 **1 : a = 0** 的时候是直线，不能无脑按照抛物线的方式处理；
- 注意点 **2 :** 对称轴 **-b / 2 * a** 的时候括号顺序是 **(double) -b / (2 * a)**

第一次 AC 的代码~ 太粗糙，得改改。

```
public class Solution {
    public int[] sortTransformedArray(int[] nums, int a, int b,
int c) {
        int[] rst = new int[nums.length];

        // Is a straight line
        if(a == 0){
            if(b > 0){
                for(int i = 0; i < nums.length; i++){
                    rst[i] = a * nums[i] * nums[i] + b * nums[i]
 + c;
                }
            } else {
                for(int i = 0; i < nums.length; i++){
                    rst[nums.length - i - 1] = a * nums[i] * num
s[i] + b * nums[i] + c;
                }
            }
            return rst;
        }


        double symmetricAxis = -((double) b / (2*a));

        int left = 0;
        int right = nums.length - 1;
        int index = (a < 0) ? 0: nums.length - 1;
        int step = (a < 0) ? 1: -1;

        while(left <= right){
            double leftDis = Math.abs(nums[left] - symmetricAxis
);
```

```
            double rightDis = Math.abs(nums[right] - symmetricAx
is);

            if(leftDis > rightDis){
                rst[index] = a * nums[left] * nums[left] + b * n
ums[left] + c;
                left ++;
            } else {
                rst[index] = a * nums[right] * nums[right] + b *
 nums[right] + c;
                right --;
            }

            index += step;
        }

        return rst;
    }
}
```

这题比较简洁的写法如下，明天学习一个。

这种写法的核心是只看 **a** 的 **“**正负**”**，无所谓 **0.**

- **If a > 0; the smallest number must be at two ends of orgin array;**
- **If a < 0; the largest number must be at two ends of orgin array;**
- 换句话说，**a** 的符号可以直接确定 **two pointer** 两端一定有一个 **“**最大**/**最小**”** 的值，每次 **O(1)** 计算一下就好，也能处理直线的情况。

```
public class Solution {
    public int[] sortTransformedArray(int[] nums, int a, int b,
int c) {
        int n = nums.length;
        int[] sorted = new int[n];
        int i = 0, j = n - 1;
        int index = a >= 0 ? n - 1 : 0;
        while (i <= j) {
            if (a >= 0) {
                sorted[index--] = quad(nums[i], a, b, c) >= quad
(nums[j], a, b, c)
                                        ? quad(nums[i++], a, b, c)
                                        : quad(nums[j--], a, b, c);
            } else {
                sorted[index++] = quad(nums[i], a, b, c) >= quad
(nums[j], a, b, c)
                                        ? quad(nums[j--], a, b, c)
                                        : quad(nums[i++], a, b, c);
            }
        }
        return sorted;
    }

    private int quad(int x, int a, int b, int c) {
        return a * x * x + b * x + c;
    }
}
```

(高频) Rearrange String k Distance Apart

Rearrange String k Distance Apart

第一次写的时候，用的是自定义 tuple(char, count, timestamp) + heap 的贪心法，每次取 count 最大的元素，count 一样取 timestamp 最小的。

但是挂在了 "aaabc" k = 2，上，正解是 "abaca"。

换句话说，条件允许的时候我们要从堆顶拿元素，但是堆顶元素非法的时候，不一定代表此时无解。

因此我们需要在此基础上加一个 **queue**，来维护这次循环寻找元素时候的解；如果在任何时刻 **maxHeap** 里没东西了，但是 **queue** 里还有，都可以说明此时真的没有合法元素了，返回 **""**.

速度尚可，超过 **48%**

```
public class Solution {
    private class Tuple{
        char chr;
        int count;
        int timestamp;
        public Tuple(char chr, int count){
            this.chr = chr;
            this.count = count;
        }
    }

    private class TupleComparator implements Comparator<Tuple>{
        public int compare(Tuple a, Tuple b){
            if(a.count != b.count) return b.count - a.count;
            else return a.timestamp - b.timestamp;
        }
    }
```

```
    public String rearrangeString(String str, int k) {
        int[] hash = new int[26];
        PriorityQueue<Tuple> maxHeap = new PriorityQueue<Tuple>(
new TupleComparator());
        for(int i = 0; i < str.length(); i++){
            hash[str.charAt(i) - 'a']++;
        }
        for(int i = 0; i < 26; i++){
            if(hash[i] > 0) maxHeap.offer(new Tuple((char)(i + '
a'),hash[i]));
        }

        StringBuilder sb = new StringBuilder();
        Queue<Tuple> queue = new LinkedList<>();

        int time = 0;

        while(!maxHeap.isEmpty()){
            Tuple tuple = maxHeap.poll();
            time ++;
            if(tuple.timestamp != 0 && time - tuple.timestamp <
k){
                queue.offer(tuple);
                continue;
            }
            sb.append(tuple.chr);
            tuple.count --;
            tuple.timestamp = time;
            if(tuple.count != 0) maxHeap.offer(tuple);

            while(!queue.isEmpty()) maxHeap.offer(queue.poll());
        }


        return !queue.isEmpty() ? "": sb.toString();
    }
}
```

思路来自
https://discuss.leetcode.com/topic/48125/java_solution_in_12_ms-o-n-time-and-space

注意，下面的代码和画图有 bug.

"aabbccd"

k = 3

返回了 abcacdb, c 冲突了，正解是 abcabcd

改良版的写法建立在下图的逻辑上：

> - 我们在一个 size = str.length 的 char[] 上维护 size = k 的 bin 若干个
> - 对于每个 bin ，里面都放置着最多 k 个 distinct character;
> - 每个 bin 里面元素的顺序存在一种 consistent ordering，即一定会严格按照我们定义的 Tuple 顺序由大到小填充。因此 bin 与 bin 之间一定不会违反 distance constraint;
> - 同时我们的遍历顺序是顺序依次遍历每个 bin，避免类似 AAAABBBBCCCCDDDD ，k = 3 的时候，实际 bin size 应该是 4 ，无脑从头填充会出 bug 的问题。

这种写法适合不适合写自己填空格的 follow-up，有待研究。。至少 AAABBBCCCDDD 这种填充时候需要额外判断一下 “这个 space 到底是不是必要的” 这类问题。

12ms，超过 90.16%

```
private class Tuple implements Comparable<Tuple>{
    char chr;
    int freq;
    public Tuple(char chr, int freq){
        this.chr = chr;
        this.freq = freq;
    }
    public int compareTo(Tuple b){
        if(this.freq != b.freq) return b.freq - this.freq;
        else return this.chr - b.chr;
    }
}


public String rearrangeString(String str, int k) {
    if(k < 2) return str;
```

```
        Tuple[] count = new Tuple[26];

        for(int i = 0; i < str.length(); i++){
            char chr = str.charAt(i);
            int index = (int) chr - 'a';

            if(count[index] == null) count[index] = new Tuple(ch
r, 0);
            count[index].freq++;
        }

        ArrayList<Tuple> list = new ArrayList<>();
        for(int i = 0; i < 26; i++){
            if(count[i] != null) list.add(count[i]);
        }
        Collections.sort(list);

        char[] rst = new char[str.length()];
        int binIndex = 0;
        int ptr = 0;
        for(int index = 0; index < list.size(); index++){
            Tuple tuple = list.get(index);
            for(int i = 0; i < tuple.freq; i++){
                rst[ptr] = tuple.chr;
                if(ptr > 0 && rst[ptr] == rst[ptr - 1]) return ""
;

                ptr += k;
                if(ptr >= rst.length) ptr = ++ binIndex;
            }
        }

        return new String(rst);
    }
```

(FB) 不改顺序的 cooldown 输出

第二轮，老外面试官，给一个String, 如AABACCDCD, 插入''使同一个字母间隔为*k:* 如果*k=3: A - - - A B - - A C - - - C D - - C D,* 一开始理解有误，认为是要先*shuffle*字母顺序然后插入"，花了不少时间，然后面试官提示字母顺序不变。

刚刚面完，欧洲小哥，发上来攒人品 Tasks: 1, 1, 2, 1

cooldown: 2

Output: 7 (order is 1 - - 1 2 - 1)

Tasks: 1, 2, 3, 1, 2, 3

(cooldown): 3

Output: 7 (order is 1 2 3 - 1 2 3)

Tasks: 1, 2, 3 ,4, 5, 6, 2, 4, 6, 1, 2, 4

(cooldown): 6

Output: 18 (1 2 3 4 5 6 - - 2 - 4 - 6 1 - 2 - 4)

```
    public static String schedule(int[] tasks, int k){
        // Key : task number
        // Value : index of last appearance
        HashMap<Integer, Integer> map = new HashMap<>();
        StringBuilder sb = new StringBuilder();
        int timestamp = 0;

        for(int i = 0; i < tasks.length; i++){
            int task = tasks[i];
            if(!map.containsKey(task)){
                sb.append("" + task);
            } else {
                while(timestamp - map.get(task) <= k){
                    sb.append("_");
                    timestamp ++;
                }
                sb.append("" + task);
            }
            map.put(task, timestamp++);
        }

        return sb.toString();
    }

    public static void main(String[] args){
        int[] tasks1 = new int[]{1,2,3,1,2,3};
        int cooldown1 = 3;

        int[] tasks2 = new int[]{1, 2, 3 ,4, 5, 6, 2, 4, 6, 1, 2
, 4};
        int cooldown2 = 6;

        int[] tasks3 = new int[]{1,2,3,1,2,3};
        int cooldown4 = 3;

        System.out.println(schedule(tasks2, cooldown2));
```

http://www.1point3acres.com/bbs/forum.php?mod=viewthread&tid=137479&extra=page%3D1%26filter%3Dsortid%26sortid%3D311&page=1

给定任务AABCB, 冷却时间k（相同任务之间的最短距离时间），任务顺序不能变，问完成任务的总时间。

例子：AABCB, k = 3, A _ _ A B C _ B, 时间为8.

解法：用hashtable保存上次的时间。

Followup1：如果k很小，怎么优化？ 解法：之前的hashtable的大小是unique task的大小，如果k很小，可以只维护k那么大的hashtable

Followup2: 如果可以改变任务的顺序，最短的任务时间是多少？ 例子：AABBC, K=2, AB*ABC, 时间为6.

解法：根据每个任务出现的频率排序，优先处理频率高的。但是具体细节没有时间讨论。

(FB) 变种：任务调度，生成“最优”的任务序列，可以用空格占位；

- 问题一：返回最优长度
- 问题二：返回带空格的最优序列
- **follow up：https://instant.1point3acres.com/thread/167190** 在这个基础上，已知**cooldown**会很小，可以视作**constant**，**task**的**type**会很多，让我减少空间复杂度 **(**这个是 **task** 顺序已经给好不需要动的情况**)**

Abstract Algebra

以下内容是一本我从哥们那“借”(并拒绝归还……)的书，“A Book of Abstract Algebra” by Charles C. Pinter.

就当休息期间写个学习笔记吧。还是不动笔墨不读书的学习方式最好。

Chap1 -- Why Abstract Algebra ?

So the first difference between the elementary and the more advanced course in algebra is that, whereas earlier we concentrated on "technique", we will now develop that branch of mathematics called "algebra" -- in a systematic way.

Ideas and general principles will take precedence over problem solving.

人类对“代数”的认识，长期以来一直处于一个“解方程”的状态 -- 数学家们互相约战，出题，看谁在规定时间内解的多，胜者就可以拿奖金。

“抽象”也是一个相对的概念。 One person's abstract is another's bread and butter.

Chap2 -- Operations

Chap3 -- The Definition of Groups