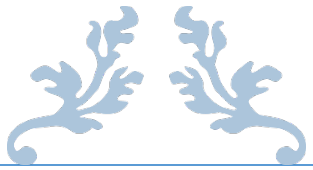


LEETCODE FIRST 400 Q&A

simple solutions

1.01

FELOMENG

Contents

1. Two Sum

Given an array of integers, return **indices** of the two numbers such that they add up to a specific target.
You may assume that each input would have ***exactly*** one solution, and you may not use the *same* element twice.
Example:

```
Given nums = [2, 7, 11, 15], target = 9,
Because nums[0] + nums[1] = 2 + 7 = 9,
return [0, 1].
```

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> m = new HashMap<Integer,Integer>();
        for(int i = 0;i < nums.length;++i) {
            int num = target - nums[i];
            if(m.containsKey(num))
                return new int[]{m.get(num), i };
            m.put(nums[i], i);
        }
        throw new RuntimeException("no answer!");
    }
}
```

思路：查找时，建立索引（Hash查找）或进行排序（二分查找）。本题缓存可在找的过程中建立索引，故一个循环可以求出解（总是使用未使用元素查找使用元素，可以保证每一对都被检索到）。Indexing/ordering is the first step to search questions.

2. Add Two Numbers

You are given two **non-empty** linked lists representing two non-negative integers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.
You may assume the two numbers do not contain any leading zero, except the number 0 itself.
Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)
Output: 7 -> 0 -> 8

```
public class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        int carry = 0;
        ListNode lResult = new ListNode(0);
        ListNode lPointer = lResult;
        while (l1 != null || l2 != null) {
            int n1 = 0, n2 = 0;
            if (l1 != null) {
                n1 = l1.val;
                l1 = l1.next;
            }
            if (l2 != null) {
                n2 = l2.val;
                l2 = l2.next;
            }
            int temp = n1 + n2 + carry;
            carry = temp / 10;
            temp %= 10;
            lPointer.next = new ListNode(temp);
            lPointer = lPointer.next;
        }
        if (carry > 0) {
            lPointer.next = new ListNode(carry);
        }
        return lResult.next;
    }
}
```

思路：数字进位问题，该位有效值为值%10，进位值为值/10。可以使用一个变量记录进位值。

3. Longest Substring Without Repeating Characters

Given a string, find the length of the **longest substring** without repeating characters.
Examples:
Given `"abcabcbb"`, the answer is `"abc"`, which the length is 3.
Given `"bbbbb"`, the answer is `"b"`, with the length of 1.
Given `"pwwkew"`, the answer is `"wke"`, with the length of 3. Note that the answer must be a **substring**, `"pwke"` is a *subsequence* and not a substring.

```
class Solution {
    public int lengthOfLongestSubstring(String s) {
        char[] sc = s.toCharArray();
        Map<Character, Integer> cm = new HashMap<Character, Integer>();
        int j = 0, maxLen = 0;
        for(int i = 0;i < sc.length; ++i) {
            char cur = sc[i];
            if(cm.containsKey(cur)) {
                maxLen = Math.max(i - j, maxLen);
                j = Math.max(j, cm.get(cur) + 1);
            }
```

```
        }
        return Math.max(sc.length - j, maxLen);
    }
}
```

思路：其实只需要前面出现过的重复字符的下标即可算出此段不重复子段的长度，核心操作其实是向前检索重复字符。需要注意的是最后循环完成后，需要再算一下没有计算的那段的长度，在这些子段中取最长的。

4. Median of Two Sorted Arrays

There are two sorted arrays **nums1** and **nums2** of size m and n respectively.
Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).
Example 1:

```
nums1 = [1, 3]
nums2 = [2]
The median is 2.0
```

Example 2:

```
nums1 = [1, 2]
nums2 = [3, 4]
The median is (2 + 3)/2 = 2.5
```

```
public class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        int m = nums1.length, n = nums2.length;
        int l = (m + n + 1) >> 1;
        int r = (m + n + 2) >> 1;
        return (getkth(nums1, 0, nums2, 0, l) + getkth(nums1, 0, nums2, 0, r)) / 2.0;
    }

    public double getkth(int[] A, int aStart, int[] B, int bStart, int k) {
        if (aStart == A.length) return B[bStart + k - 1];
        if (bStart == B.length) return A[aStart + k - 1];
        if (k == 1) return Math.min(A[aStart], B[bStart]);

        int aMid = Integer.MAX_VALUE, bMid = Integer.MAX_VALUE;
        if (aStart + k/2 - 1 < A.length) aMid = A[aStart + k/2 - 1];
        if (bStart + k/2 - 1 < B.length) bMid = B[bStart + k/2 - 1];

        if (aMid < bMid)
            return getkth(A, aStart + k/2, B, bStart,       k - k/2);
        else
            return getkth(A, aStart,       B, bStart + k/2, k - k/2);
    }
}
```

5. Longest Palindromic Substring

Given a string **s**, find the longest palindromic substring in **s**. You may assume that the maximum length of **s** is 1000.
Example:

```
Input: "babad"
Output: "bab"
Note: "aba" is also a valid answer.
```

Example:

```
Input: "cbbd"
Output: "bb"
```

```
public class Solution {
    private int lo, maxLen;
    public String longestPalindrome(String s) {
        int len = s.length();
        if (len < 2)
            return s;
        for (int i = 0; i < len - 1; i++) {
            extendPalindrome(s, i, i);
            extendPalindrome(s, i, i + 1);
        }
        return s.substring(lo, lo + maxLen);
    }
    private void extendPalindrome(String s, int j, int k) {
        while (j >= 0 && k < s.length() && s.charAt(j) == s.charAt(k)) {
            j--;
            k++;
        }
        if (maxLen < k - j - 1) {
            lo = j + 1;
            maxLen = k - j - 1;
```

```
    }
}
```

6. ZigZag Conversion

The string `"PAYPALISHIRING"` is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)

```
P   A   H   N
A P L S I I G
Y   I   R
```

And then read line by line: `"PAHNAPLSIIGYIR"`
Write the code that will take a string and make this conversion given a number of rows:

```
string convert(string text, int nRows);
```

`convert("PAYPALISHIRING", 3)` should return `"PAHNAPLSIIGYIR"`.

```
public class Solution {
    public String convert(String s, int numRows) {
        char[] c = s.toCharArray();
        int len = c.length;
        StringBuffer[] sb = new StringBuffer[numRows];
        for (int i = 0; i < sb.length; i++)
            sb[i] = new StringBuffer();
        int i = 0;
        while (i < len) {
            for (int idx = 0; idx < numRows && i < len; idx++)
                sb[idx].append(c[i++]);
            for (int idx = numRows - 2; idx >= 1 && i < len; idx--)
                sb[idx].append(c[i++]);
        }
        for (int idx = 1; idx < sb.length; idx++)
            sb[0].append(sb[idx]);
        return sb[0].toString();
    }
}
```

7. Reverse Integer

Reverse digits of an integer.
Example1: `x = 123, return 321`
Example2: `x = -123, return -321`
click to show spoilers.
Have you thought about this?
Here are some good questions to ask before coding. Bonus points for you if you have already thought through this!
If the integer's last digit is 0, what should the output be? ie, cases such as 10, 100.
Did you notice that the reversed integer might overflow? Assume the input is a 32-bit integer, then the reverse of 1000000003 overflows. How should you handle such cases?
For the purpose of this problem, assume that your function returns 0 when the reversed integer overflows.
Note:
The input is assumed to be a 32-bit signed integer. Your function should **return 0 when the reversed integer overflows**.

```
public class Solution {
        public int reverse(int x) {
                int result = 0;
                while (x != 0) {
                        int tail = x % 10;
                        int newResult = result * 10 + tail;
                        if ((newResult - tail) / 10 != result)
                                return 0;
                        result = newResult;
                        x = x / 10;
                }
                return result;
        }
}
```

8. String to Integer (atoi)

Implement `atoi` to convert a string to an integer.
Hint: Carefully consider all possible input cases. If you want a challenge, please do not see below and ask yourself what are the possible input cases.
Notes: It is intended for this problem to be specified vaguely (ie, no given input specs). You are responsible to gather all the input requirements up front.
spoilers alert... click to show requirements for atoi.
Requirements for atoi:
The function first discards as many whitespace characters as necessary until the first non-whitespace character is found. Then, starting from this character, takes an optional initial plus or minus sign followed by as many numerical digits as possible, and interprets them as a numerical value.
The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function.

If the first sequence of non-whitespace characters in str is not a valid integral number, or if no such sequence exists because either str is empty or it contains only whitespace characters, no conversion is performed.
If no valid conversion could be performed, a zero value is returned. If the correct value is out of the range of representable values, INT_MAX (2147483647) or INT_MIN (-2147483648) is returned.

```
public class Solution {
    public int myAtoi(String str) {
        if (str == null || str.length() == 0)
                return 0;//
        str = str.trim();
        char firstChar = str.charAt(0);
        int sign = 1, start = 0, len = str.length();
        long sum = 0;
        if (firstChar == '+') {
                sign = 1;
                start++;
        } else if (firstChar == '-') {
                sign = -1;
                start++;
        }
        for (int i = start; i < len; i++) {
                if (!Character.isDigit(str.charAt(i)))
                        return (int) sum * sign;
                sum = sum * 10 + str.charAt(i) - '0';
                if (sign == 1 && sum > Integer.MAX_VALUE)
                        return Integer.MAX_VALUE;
                if (sign == -1 && (-1) * sum < Integer.MIN_VALUE)
                        return Integer.MIN_VALUE;
        }

        return (int) sum * sign;
    }
}
```

9. Palindrome Number

Determine whether an integer is a palindrome. Do this without extra space.
click to show spoilers.
Some hints:
Could negative integers be palindromes? (ie, -1)
If you are thinking of converting the integer to string, note the restriction of using extra space.
You could also try reversing an integer. However, if you have solved the problem "Reverse Integer", you know that the reversed integer might overflow. How would you handle such case?
There is a more generic way of solving this problem.

```
public class Solution {
    public boolean isPalindrome(int x) {
        if (x<0 || (x!=0 && x%10==0)) return false;
        int rev = 0;
        while (x>rev){
        rev = rev*10 + x%10;
        x = x/10;
        }
        return (x==rev || x==rev/10);
    }
}
```

10. Regular Expression Matching

Implement regular expression matching with support for `'.'` and `'*'`.

```
'.' Matches any single character.
'*' Matches zero or more of the preceding element.
The matching should cover the entire input string (not partial).
The function prototype should be:
bool isMatch(const char *s, const char *p)
Some examples:
isMatch("aa","a") ? false
isMatch("aa","aa") ? true
isMatch("aaa","aa") ? false
isMatch("aa", "a*") ? true
isMatch("aa", ".*") ? true
isMatch("ab", ".*") ? true
isMatch("aab", "c*a*b") ? true
```

```
public class Solution {
    public boolean isMatch(String str, String regex) {
        boolean[][] dp = new boolean[str.length() + 1][regex.length() + 1];
        dp[0][0] = true;
        for(int i = 1; i < regex.length() + 1; i++) {
            if(regex.charAt(i - 1) == '*') dp[0][i] = dp[0][i-2];
        }
```

```
        for(int i = 1; i < dp.length; i++) {
            for(int j = 1; j < dp[0].length; j++) {
                if(match(str.charAt(i-1), regex.charAt(j-1))) {
                    dp[i][j] = dp[i-1][j-1];
                }else{
                    if(regex.charAt(j-1) == '*') {
                        dp[i][j] = dp[i][j-2];
                        if(match(str.charAt(i-1), regex.charAt(j-2))) {
                            dp[i][j] |= dp[i-1][j];
                        }
                    }
                }
            }
        }

        return dp[str.length()][regex.length()];
    }

    private boolean match(char c1, char r) {
        return c1 == r || r == '.';
    }
}
```

11. Container With Most Water

Given n non-negative integers a_1, a_2, ..., a_n, where each represents a point at coordinate (i, a_i). n vertical lines are drawn such that the two endpoints of line i is at (i, a_i) and (i, 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water.

Note: You may not slant the container and n is at least 2.

```
public class Solution {
    public int maxArea(int[] height) {
        int n = height.length;
        int max = 0;
        int i = 0, j = n - 1;
        while(i < j) {
            max = Math.max(Math.min(height[i], height[j]) * (j - i), max);
            if(height[i] < height[j]) {
                ++i;
            } else {
                --j;
            }
        }
        return max;
    }
}
```

12. Integer to Roman

Given an integer, convert it to a roman numeral.

Input is guaranteed to be within the range from 1 to 3999.

```
public class Solution {
    public String intToRoman(int num) {
        String M[] = {"", "M", "MM", "MMM"};
        String C[] = {"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"};
        String X[] = {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"};
        String I[] = {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"};
        return M[num/1000] + C[(num%1000)/100] + X[(num%100)/10] + I[num%10];
    }
}
```

13. Roman to Integer

Given a roman numeral, convert it to an integer.

Input is guaranteed to be within the range from 1 to 3999.

```
public class Solution {
    public int romanToInt(String s) {
        int sum=0;
        if(s.indexOf("IV")!=-1){sum-=2;}
        if(s.indexOf("IX")!=-1){sum-=2;}
        if(s.indexOf("XL")!=-1){sum-=20;}
        if(s.indexOf("XC")!=-1){sum-=20;}
        if(s.indexOf("CD")!=-1){sum-=200;}
        if(s.indexOf("CM")!=-1){sum-=200;}
        char c[]=s.toCharArray();
        int count=0;
        for(;count<=s.length()-1;count++){
            if(c[count]=='M') sum+=1000;
            if(c[count]=='D') sum+=500;
            if(c[count]=='C') sum+=100;
            if(c[count]=='L') sum+=50;
            if(c[count]=='X') sum+=10;
```

```
            if(c[count]=='V') sum+=5;
            if(c[count]=='I') sum+=1;
        }
        return sum;
    }
}
```

14. Longest Common Prefix

Write a function to find the longest common prefix string amongst an array of strings.

```
public class Solution {
        public String longestCommonPrefix(String[] strs) {
                if (strs.length == 0)
                        return "";
                if (strs.length == 1)
                        return strs[0];
                StringBuilder sb = new StringBuilder();
                int n = Integer.MAX_VALUE;
                boolean finished = false;
                for (int i = 0; i < strs.length; ++i)
                        n = Math.min(strs[i].length(), n);
                for (int i = 0; i < n; ++i) {
                        char c = strs[0].charAt(i);
                        for (int j = 1; j < strs.length; ++j) {
                                if (strs[j].charAt(i) != c) {
                                        finished = true;
                                        break;
                                }
                        }
                        if (finished)
                                break;
                        sb.append(c);
                }
                return sb.toString();
        }
}
```

15. 3Sum

Given an array *S* of *n* integers, are there elements *a*, *b*, *c* in *S* such that *a* + *b* + *c* = 0? Find all unique triplets in the array which gives the sum of zero.

Note: The solution set must not contain duplicate triplets.

```
For example, given array S = [-1, 0, 1, 2, -1, -4],
A solution set is:
[ [-1, 0, 1],
  [-1, -1, 2]]
```

```
public class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        if(nums.length < 3) return result;
        Arrays.sort(nums);
        int i = 0;
        while(i < nums.length - 2) {
            if(nums[i] > 0) break;
            int j = i + 1;
            int k = nums.length - 1;
            while(j < k) {
                int sum = nums[i] + nums[j] + nums[k];
                if(sum == 0) result.add(Arrays.asList(nums[i], nums[j], nums[k]));
                if(sum <= 0) while(nums[j] == nums[++j] && j < k);
                if(sum >= 0) while(nums[k--] == nums[k] && j < k);
            }
            while(nums[i] == nums[++i] && i < nums.length - 2);
        }
        return result;
    }
}
```

16. 3Sum Closest

Given an array *S* of *n* integers, find three integers in *S* such that the sum is closest to a given number, target. Return the sum of the three integers. You may assume that each input would have exactly one solution.

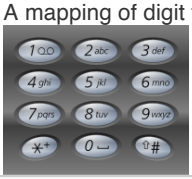
```
    For example, given array S = {-1 2 1 -4}, and target = 1.
    The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).
```

```
public class Solution {
    public int threeSumClosest(int[] nums, int target) {
        Arrays.sort(nums);
        int closest = Integer.MAX_VALUE;
        int i = 0;
```

```
        while(i < nums.length - 2) {
            int j = i + 1;
            int k = nums.length - 1;
            while(j < k) {
                int sum = nums[i] + nums[j] + nums[k];
                if(closest == Integer.MAX_VALUE
                   || Math.abs(closest - target) > Math.abs(sum - target)) {
                    closest = sum;
                }
                if(sum == target) return sum;
                if(sum <= target) while(nums[j] == nums[++j] && j < k);
                if(sum >= target) while(nums[k--] == nums[k] && j < k);
            }
            while(nums[i] == nums[++i] && i < nums.length - 2);
        }
        return closest;
    }
}
```

17. Letter Combinations of a Phone Number

Given a digit string, return all possible letter combinations that the number could represent.
A mapping of digit to letters (just like on the telephone buttons) is given below.



```
Input:Digit string "23"
Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].
```

Note:
Although the above answer is in lexicographical order, your answer could be in any order you want.

```
public class Solution {
    public List<String> letterCombinations(String digits) {
        Map<String, List<String>> digitalMap = new HashMap<String, List<String>>();
        digitalMap.put("2",
                new ArrayList<String>(Arrays.asList("a", "b", "c")));
        digitalMap.put("3",
                new ArrayList<String>(Arrays.asList("d", "e", "f")));
        digitalMap.put("4",
                new ArrayList<String>(Arrays.asList("g", "h", "i")));
        digitalMap.put("5",
                new ArrayList<String>(Arrays.asList("j", "k", "l")));
        digitalMap.put("6",
                new ArrayList<String>(Arrays.asList("m", "n", "o")));
        digitalMap.put("7",
                new ArrayList<String>(Arrays.asList("p", "q", "r", "s")));
        digitalMap.put("8",
                new ArrayList<String>(Arrays.asList("t", "u", "v")));
        digitalMap.put("9",
                new ArrayList<String>(Arrays.asList("w", "x", "y", "z")));
        List<String> resultList = new ArrayList<String>();
        getLetterCombinations(digits.toCharArray(), new StringBuilder(), 0,
                digitalMap, resultList);
        return resultList;
    }
    private static void getLetterCombinations(char[] c, StringBuilder sbSoFar,
            int curP, Map<String, List<String>> digitalMap,
            List<String> resultList) {
        if (curP == c.length) {
            if (sbSoFar.length() > 0)
                resultList.add(sbSoFar.toString());
            return;
        }
        char cur = c[curP];
        for (String letter : digitalMap.get(String.valueOf(cur))) {
            StringBuilder sb = new StringBuilder();
            sb.append(sbSoFar);
            sb.append(letter);
            getLetterCombinations(c, sb, curP + 1, digitalMap, resultList);
        }
    }
}
```

18. 4Sum

Given an array *S* of *n* integers, are there elements *a*, *b*, *c*, and *d* in *S* such that *a* + *b* + *c* + *d* = target? Find all unique quadruplets in the array which gives the sum of target.

Note: The solution set must not contain duplicate quadruplets.

```
For example, given array S = [1, 0, -1, 0, -2, 2], and target = 0.
A solution set is:
[ [-1,  0, 0, 1],
  [-2, -1, 1, 2],
  [-2,  0, 0, 2]]
```

```
public class Solution {
    public List<List<Integer>> fourSum(int[] nums, int target) {
        int n = nums.length;
        Arrays.sort(nums);
        return kSum(nums, target, 4, 0, n);
    }
    private static ArrayList<List<Integer>> kSum(int[] nums, int target, int k,
            int index, int len) {
        ArrayList<List<Integer>> res = new ArrayList<List<Integer>>();
        if (index >= len)
            return res;
        int max = nums[nums.length - 1];
        if (k * nums[index] > target || k * max < target)
            return res;
        if (k == 2) {
            int i = index, j = len - 1;
            while (i < j) {
                if (target - nums[i] == nums[j]) {
                    List<Integer> temp = new ArrayList<>();
                    temp.add(nums[i]);
                    temp.add(target - nums[i]);
                    res.add(temp);
                    while (i < j && nums[i] == nums[i + 1])
                        i++;
                    while (i < j && nums[j - 1] == nums[j])
                        j--;
                    i++;
                    j--;
                } else if (target - nums[i] > nums[j])
                    i++;
                else
                    j--;
            }
        } else {
            for (int i = index; i < len - k + 1; i++) {
                ArrayList<List<Integer>> temp = kSum(nums, target - nums[i],
                        k - 1, i + 1, len);
                if (temp != null) {
                    for (List<Integer> t : temp)
                        t.add(0, nums[i]);
                    res.addAll(temp);
                }
                while (i < len - 1 && nums[i] == nums[i + 1])
                    ++i;
            }
        }
        return res;
    }
}
```

19. Remove Nth Node From End of List

Given a linked list, remove the n^{th} node from the end of list and return its head.
For example,

```
   Given linked list: 1->2->3->4->5, and n = 2.
   After removing the second node from the end, the linked list becomes 1->2->3->5.
```

Note:
Given *n* will always be valid.
Try to do this in one pass.

```
public class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        if(head.next == null) {
            return null;
        }
        int i = 0;
        ListNode p1 = head, p2 = head;
        while(p1.next != null) {
            p1 = p1.next;
            ++i;
```

```
                p2 = p2.next;
            }
        }
        if(i == n - 1) {
            head = head.next;
        } else {
            p2.next = p2.next.next;
        }
        return head;
    }
}
```

20. Valid Parentheses

Given a string containing just the characters `'('`, `')'`, `'{'`, `'}'`, `'['` and `']'`, determine if the input string is valid.
The brackets must close in the correct order, `"()"` and `"()[]{}"` are all valid but `"(]"` and `"([)]"` are not.

```
public class Solution {
    public boolean isValid(String s) {
        Map<Character,Character> pM = new HashMap<Character, Character>();
        pM.put('(', ')');
        pM.put('{', '}');
        pM.put('[', ']');
        Stack<Character> pS = new Stack<Character>();
        for(Character c: s.toCharArray()) {
            if(pM.containsKey(c)) {
                pS.push(pM.get(c));
            } else {
                if(pS.isEmpty() || c != pS.pop())
                    return false;
            }
        }
        if(!pS.isEmpty())
            return false;
        return true;
    }
}
```

21. Merge Two Sorted Lists

Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.

```
public class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        if(l1 == null)
            return l2;
        if(l2 == null)
            return l1;
        if(l1.val < l2.val) {
            l1.next = mergeTwoLists(l1.next, l2);
            return l1;
        } else {
            l2.next = mergeTwoLists(l1, l2.next);
            return l2;
        }
    }
}
```

22. Generate Parentheses

Given *n* pairs of parentheses, write a function to generate all combinations of well-formed parentheses.
For example, given *n* = 3, a solution set is:

```
[ "((()))",
  "(()())",
  "(())()",
  "()(())",
  "()()()"]
```

```
public class Solution {
    public List<String> generateParenthesis(int n) {
        List<String> list = new ArrayList<String>();
        backtrack(list, "", 0, 0, n);
        return list;
    }
    public void backtrack(List<String> list, String str, int open, int close,
            int max) {
        if (str.length() == max * 2) {
            list.add(str);
            return;
        }
        if (open < max)
            backtrack(list, str + "(", open + 1, close, max);
```

```
            backtrack(list, str + ")", open, close + 1, max);
    }
}
```

23. Merge k Sorted Lists

Merge *k* sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

```
public class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        if (lists == null || lists.length == 0)
            return null;
        Queue<ListNode> nodeList = new PriorityQueue<ListNode>(lists.length,
                (m, n) -> (m.val - n.val));
        for (ListNode l : lists) {
            if (l != null) {
                nodeList.add(l);
            }
        }
        ListNode node = new ListNode(0);
        ListNode last = node;
        while (!nodeList.isEmpty()) {
            last.next = nodeList.poll();
            last = last.next;
            if (last.next != null)
                nodeList.add(last.next);
        }
        return node.next;
    }
}
```

24. Swap Nodes in Pairs

Given a linked list, swap every two adjacent nodes and return its head.
For example,
Given `1->2->3->4`, you should return the list as `2->1->4->3`.
Your algorithm should use only constant space. You may **not** modify the values in the list, only nodes itself can be changed.

```
public class Solution {
    public ListNode swapPairs(ListNode head) {
        ListNode p = head;
        if(p == null || p.next == null)
            return head;
        ListNode newHead = p.next;
        p.next = p.next.next;
        newHead.next = p;
        p = newHead.next.next;
        newHead.next.next = swapPairs(p);
        return newHead;
    }
}
```

25. Reverse Nodes in k-Group

Given a linked list, reverse the nodes of a linked list *k* at a time and return its modified list.
k is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of *k* then left-out nodes in the end should remain as it is.
You may not alter the values in the nodes, only nodes itself may be changed.
Only constant memory is allowed.
For example,
Given this linked list: `1->2->3->4->5`
For *k* = 2, you should return: `2->1->4->3->5`
For *k* = 3, you should return: `3->2->1->4->5`

```
public class Solution {
    public ListNode reverseKGroup(ListNode head, int k) {
        if (head==null||head.next==null||k<2) return head;
            ListNode dummy = new ListNode(0);
            dummy.next = head;

            ListNode tail = dummy, prev = dummy,temp;
            int count;
            while(true){
                count =k;
                while(count>0&&tail!=null){
                    count--;
                    tail=tail.next;
                }
                if (tail==null) break;
                head=prev.next;
                while(prev.next!=tail){
                    temp=prev.next;//Assign
                    prev.next=temp.next;//Delete
```

```
                tail.next=temp;//Insert

            }
            tail=head;
            prev=head;

        }
        return dummy.next;

    }
}
```

26. Remove Duplicates from Sorted Array

Given a sorted array, remove the duplicates in place such that each element appear only *once* and return the new length.
Do not allocate extra space for another array, you must do this in place with constant memory.
For example,
Given input array *nums* = `[1,1,2]`,
Your function should return length = `2`, with the first two elements of *nums* being `1` and `2` respectively. It doesn't matter what you leave beyond the new length.

```
public class Solution {
    public int removeDuplicates(int[] nums) {
        Integer temp = null, cur = null;
        int j = 0;
        for(int i = 0;i < nums.length; ++i) {
            cur = nums[i];
            if(temp == null || temp.intValue() != cur.intValue()) {
                nums[j] = cur;
                ++j;
            }
            temp = cur;
        }
        return j;
    }
}
```

27. Remove Element

Given an array and a value, remove all instances of that value in place and return the new length.
Do not allocate extra space for another array, you must do this in place with constant memory.
The order of elements can be changed. It doesn't matter what you leave beyond the new length.
Example:
Given input array *nums* = `[3,2,2,3]`, *val* = `3`
Your function should return length = 2, with the first two elements of *nums* being 2.

```
public class Solution {
    public int removeElement(int[] nums, int val) {
        int cur;
        int j = 0;
        for(int i = 0;i < nums.length; ++i) {
            cur = nums[i];
            if(val != cur) {
                nums[j] = cur;
                ++j;
            }
        }
        return j;
    }
}
```

28. Implement strStr()

Implement strStr().
Returns the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

```
public class Solution {
        public int strStr(String haystack, String needle) {
                if (haystack == null || needle == null)
                        return -1;
                char[] cn = needle.toCharArray();
                if (cn.length == 0)
                        return 0;
                char[] ch = haystack.toCharArray();
                for (int i = 0; i < ch.length - cn.length + 1; ++i) {
                        for (int j = 0; j < cn.length; ++j) {
                                if (ch[i + j] != cn[j])
                                        break;
                                else if (j == cn.length - 1)
                                        return i;
                        }
                }
                return -1;
        }
```

It can be solved using KMP.

```
    void makeNext(char patternChars[], int next[]) {
        int q, k;
        int n = patternChars.length;
        next[0] = 0;
        for (q = 1, k = 0; q < n; ++q) {
            while (k > 0 && patternChars[q] != patternChars[k])
                k = next[k - 1];
            if (patternChars[q] == patternChars[k]) {
                k++;
            }
            next[q] = k;
        }
    }
```

29. Divide Two Integers

Divide two integers without using multiplication, division and mod operator.
If it is overflow, return MAX_INT.

```
public class Solution {
        public int divide(int dividend, int divisor) {
                if (divisor == 0)
                        throw new java.lang.ArithmeticException("/ by zero");
                long result = divideLong(dividend, divisor);
                return result > Integer.MAX_VALUE ? Integer.MAX_VALUE : (int) result;
        }
        public long divideLong(long dividend, long divisor) {
                boolean negative = dividend < 0 != divisor < 0;
                if (dividend < 0)
                        dividend = -dividend;
                if (divisor < 0)
                        divisor = -divisor;
                if (dividend < divisor)
                        return 0;
                long sum = divisor;
                long divide = 1;
                while ((sum + sum) <= dividend) {
                        sum += sum;
                        divide += divide;
                }
                return negative ? -(divide + divideLong((dividend - sum), divisor))
                                : (divide + divideLong((dividend - sum), divisor));
        }
}
```

30. Substring with Concatenation of All Words

You are given a string, **s**, and a list of words, **words**, that are all of the same length. Find all starting indices of substring(s) in **s** that is a concatenation of each word in **words** exactly once and without any intervening characters.
For example, given:
s: `"barfoothefoobarman"`
words: `["foo", "bar"]`
You should return the indices: `[0,9]`.
(order does not matter).

```
public class Solution {
    public List<Integer> findSubstring(String s, String[] words) {
        if (words.length == 0 || words[0].length() == 0)
            return new ArrayList<>();
        HashMap<String, Integer> map = new HashMap<>();
        for (String word : words)
            map.put(word, map.getOrDefault(word, 0) + 1);
        List<Integer> list = new ArrayList<>();
        int gap = words[0].length();
        int nlen = words.length * gap;
        for (int k = 0; k < gap; k++) {
            HashMap<String, Integer> wordmap = new HashMap<>(map);
            for (int i = k, j = 0; i < s.length() - nlen + 1
                    && i + j <= s.length() - gap;) {
                String temp = s.substring(i + j, i + j + gap);
                if (wordmap.containsKey(temp)) {
                    wordmap.put(temp, wordmap.get(temp) - 1);
                    if (wordmap.get(temp) == 0)
                        wordmap.remove(temp);
                    if (wordmap.isEmpty())
                        list.add(i);
                    j += gap;
                } else {
                    if (j == 0)
                        i += gap;
                    else {
```

```
                        wordmap.put(s.substring(i, i + gap),
                                wordmap.getOrDefault(s.substring(i, i + gap), 0)
                                        + 1);
                        i += gap;
                        j -= gap;
                    }
                }
            }
        }
        return list;
    }
}
```

31. Next Permutation

Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers.
If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).
The replacement must be in-place, do not allocate extra memory.
Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.
`1,2,3` →`1,3,2`
`3,2,1` →`1,2,3`
`1,1,5` →`1,5,1`

```
public class Solution {
        public void nextPermutation(int[] nums) {
                int i;
                for (i = nums.length - 1; i > 0; --i) {
                        if (nums[i] > nums[i - 1]) {
                                break;
                        }
                }
                if (i > 0) {
                        for (int j = nums.length - 1; j >= i; --j) {
                                if (nums[j] > nums[i - 1]) {
                                        swap(nums, j, i - 1);
                                        break;
                                }
                        }
                }
                reverse(nums, i, nums.length - 1);
        }

        private static void swap(int[] nums, int i, int j) {
                int temp = nums[i];
                nums[i] = nums[j];
                nums[j] = temp;
        }

        private static void reverse(int[] nums, int i, int j) {
                while (i < j) {
                        swap(nums, i++, j--);
                }
        }
}
```

32. Longest Valid Parentheses

Given a string containing just the characters `'('` and `')'`, find the length of the longest valid (well-formed) parentheses substring.
For `"(()"`, the longest valid parentheses substring is `"()"`, which has length = 2.
Another example is `")()())"`, where the longest valid parentheses substring is `"()()"`, which has length = 4.

```
public class Solution {
    public int longestValidParentheses(String s) {
        Stack<Integer> stack = new Stack<Integer>();
        int max=0;
        int left = -1;
        for(int j=0;j<s.length();j++){
            if(s.charAt(j)=='(') stack.push(j);
            else {
                if (stack.isEmpty()) left=j;
                else{
                    stack.pop();
                    if(stack.isEmpty()) max=Math.max(max,j-left);
                    else max=Math.max(max,j-stack.peek());
                }
            }
        }
        return max;
    }
}
```

33. Search in Rotated Sorted Array

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand

(i.e., `0 1 2 4 5 6 7` might become `4 5 6 7 0 1 2`).
You are given a target value to search. If found in the array return its index, otherwise return -1.
You may assume no duplicate exists in the array.

```
public class Solution {
    public int search(int[] nums, int target) {
        int l = 0, h = nums.length - 1;
        if (h < 0)
            return -1;
        int n0 = nums[0];
        while (l <= h) {
            int mid = (l + h) >>> 1, m = nums[mid];
            if (target == m)
                return mid;
            else if (m < n0 == target < n0 && target < m
                    || target >= n0 && m < n0)
                h = mid - 1;
            else
                l = mid + 1;
        }
        return -1;
    }
}
```

34. Search for a Range

Given an array of integers sorted in ascending order, find the starting and ending position of a given target value.
Your algorithm's runtime complexity must be in the order of *O*(log *n*).
If the target is not found in the array, return `[-1, -1]`.
For example,
Given `[5, 7, 7, 8, 8, 10]` and target value 8,
return `[3, 4]`.

```
public class Solution {
    public int[] searchRange(int[] A, int target) {
        int start = Solution.firstGreaterEqual(A, target);
        if (start == A.length || A[start] != target) {
            return new int[] { -1, -1 };
        }
        return new int[] { start,
                Solution.firstGreaterEqual(A, target + 1) - 1 };
    }
    private static int firstGreaterEqual(int[] A, int target) {
        int low = 0, high = A.length;
        while (low < high) {
            int mid = low + ((high - low) >> 1);
            if (A[mid] < target) {
                low = mid + 1;
            } else {
                high = mid;
            }
        }
        return low;
    }
}
```

35. Search Insert Position

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.
You may assume no duplicates in the array.
Here are few examples.
`[1,3,5,6]`, 5 →2
`[1,3,5,6]`, 2 →1
`[1,3,5,6]`, 7 →4
`[1,3,5,6]`, 0 →0

```
public class Solution {
        public int searchInsert(int[] nums, int target) {
                int n = nums.length;
                if (n == 0) {
                        return 0;
                }
                int left = 0, right = n - 1;
                int mid = 0;
                while (left <= right) {
                        mid = left + (right - left) / 2;
                        if (target > nums[mid]) {
                                left = Math.min(mid + 1, right);
                        } else if (target < nums[mid]) {
                                right = Math.max(mid - 1, left);
                        } else {
```

```
                        }
                        if (left == right) {
                                if (target <= nums[left]) {
                                        return left;
                                }
                                return right + 1;
                        }
                }
                return -1;
        }
}
```

36. Valid Sudoku

Determine if a Sudoku is valid, according to: Sudoku Puzzles - The Rules.
The Sudoku board could be partially filled, where empty cells are filled with the character `'.'`.

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

A partially filled sudoku which is valid.

Note:
A valid Sudoku board (partially filled) is not necessarily solvable. Only the filled cells need to be validated.

```
public class Solution {
    public boolean isValidSudoku(char[][] board) {
        Set<String> seen = new HashSet<>();
        for (int i = 0; i < 9; ++i) {
            for (int j = 0; j < 9; ++j) {
                char number = board[i][j];
                if (number != '.')
                    if (!seen.add(number + " in row " + i)
                            || !seen.add(number + " in column " + j)
                            || !seen.add(number + " in block " + i / 3 + "-"
                                    + j / 3))
                        return false;
            }
        }
        return true;
    }
}
```

37. Sudoku Solver

Write a program to solve a Sudoku puzzle by filling the empty cells.
Empty cells are indicated by the character `'.'`.
You may assume that there will be only one unique solution.

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

A sudoku puzzle...

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

...and its solution numbers marked in red.

```
public class Solution {
    public void solveSudoku(char[][] board) {
        if (board == null || board.length == 0)
            return;
        solve(board);
    }
    public boolean solve(char[][] board) {
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[0].length; j++) {
                if (board[i][j] == '.') {
                    for (char c = '1'; c <= '9'; c++) {
                        if (isValid(board, i, j, c)) {
                            board[i][j] = c;
                            if (solve(board))
                                return true;
                            else
                                board[i][j] = '.';
                        }
                    }
                    return false;
                }
            }
        }
        return true;
```

```
    }
    private boolean isValid(char[][] board, int row, int col, char c) {
        for (int i = 0; i < 9; i++) {
            if (board[i][col] != '.' && board[i][col] == c)
                return false;
            if (board[row][i] != '.' && board[row][i] == c)
                return false;
            if (board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] != '.'
                    && board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] == c)
                return false;
        }
        return true;
    }
}
```

38. Count and Say

The count-and-say sequence is the sequence of integers with the first five terms as following:

```
1.     1
2.     11
3.     21
4.     1211
5.     111221
```

`1` is read off as `"one 1"` or `11`.
`11` is read off as `"two 1s"` or `21`.
`21` is read off as `"one 2"`, then `one 1"` or `1211`.
Given an integer *n*, generate the n^{th} term of the count-and-say sequence.
Note: Each term of the sequence of integers will be represented as a string.
Example 1:

```
Input: 1
Output: "1"
```

Example 2:

```
Input: 4
Output: "1211"
```

```
public class Solution {
        public String countAndSay(int n) {
                if (n == 1) {
                        return "1";
                }
                return countAndSay(countAndSay(n - 1));
        }
        public String countAndSay(String str) {
                StringBuilder sb = new StringBuilder();
                char[] cs = str.toCharArray();
                int i = 0, j = 0;
                for (; i < cs.length; i = j) {
                        char cur = cs[i];
                        j = i + 1;
                        while (j < cs.length && cur == cs[j]) {
                                ++j;
                        }
                        sb.append(j - i);
                        sb.append(cur);
                }
                return sb.toString();
        }
}
```

39. Combination Sum

Given a **set** of candidate numbers (***C***) **(without duplicates)** and a target number (***T***), find all unique combinations in ***C*** where the candidate numbers sums to ***T***.
The **same** repeated number may be chosen from ***C*** unlimited number of times.
Note:

- All numbers (including target) will be positive integers.
- The solution set must not contain duplicate combinations.

For example, given candidate set `[2, 3, 6, 7]` and target `7`,
A solution set is:

```
[ [7],
  [2, 2, 3]]
```

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

```
        int n = candidates.length;
        for (int i = 0; i < n; ++i) {
            List<Integer> candidateL = new ArrayList<Integer>();
            candidateL.add(candidates[i]);
            backTrack(candidates, i, candidates[i], candidateL, target, r);
        }
        return r;
    }
    private static void backTrack(int[] nums, int i, int sum,
            List<Integer> candidateL, int target, List<List<Integer>> r) {
        if (sum > target)
            return;
        if (sum == target) {
            List<Integer> newL = new ArrayList<Integer>();
            newL.addAll(candidateL);
            r.add(newL);
            return;
        }
        for (int j = i; j < nums.length; ++j) {
            candidateL.add(Integer.valueOf(nums[j]));
            sum += nums[j];
            backTrack(nums, j, sum, candidateL, target, r);
            candidateL.remove(Integer.valueOf(nums[j]));
            sum -= nums[j];
        }
    }
}
```

40. Combination Sum II

Given a collection of candidate numbers (***C***) and a target number (***T***), find all unique combinations in ***C*** where the candidate numbers sums to ***T***.
Each number in ***C*** may only be used **once** in the combination.
Note:

- All numbers (including target) will be positive integers.
- The solution set must not contain duplicate combinations.

For example, given candidate set `[10, 1, 2, 7, 6, 1, 5]` and target `8`,
A solution set is:

```
[ [1, 7],
  [1, 2, 5],
  [2, 6],
  [1, 1, 6]]
```

```
public class Solution {
    public List<List<Integer>> combinationSum2(int[] cand, int target) {
        Arrays.sort(cand);
        List<List<Integer>> res = new ArrayList<List<Integer>>();
        List<Integer> path = new ArrayList<>();
        dfs_com(cand, 0, target, path, res);
        return res;
    }
    void dfs_com(int[] cand, int cur, int target, List<Integer> path,
            List<List<Integer>> res) {
        if (target == 0) {
            res.add(new ArrayList<>(path));
            return;
        }
        if (target < 0)
            return;
        for (int i = cur; i < cand.length; i++) {
            if (i > cur && cand[i] == cand[i - 1])
                continue;
            path.add(path.size(), cand[i]);
            dfs_com(cand, i + 1, target - cand[i], path, res);
            path.remove(path.size() - 1);
        }
    }
}
```

41. First Missing Positive

Given an unsorted integer array, find the first missing positive integer.
For example,
Given `[1,2,0]` return `3`,
and `[3,4,-1,1]` return `2`.
Your algorithm should run in *O*(*n*) time and uses constant space.

```
public class Solution {
        public int firstMissingPositive(int[] nums) {
```

```
                while (i < nums.length) {
                        if (nums[i] == i + 1 || nums[i] <= 0 || nums[i] > nums.length)
                                i++;
                        else if (nums[nums[i] - 1] != nums[i])
                                swap(nums, i, nums[i] - 1);
                        else
                                i++;
                }
                i = 0;
                while (i < nums.length && nums[i] == i + 1)
                        i++;
                return i + 1;
        }
        private void swap(int[] A, int i, int j) {
                int temp = A[i];
                A[i] = A[j];
                A[j] = temp;
        }
}
```

42. Trapping Rain Water

Given *n* non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

For example,

Given `[0,1,0,2,1,0,1,3,2,1,2,1]`, return `6`.

The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped. **Thanks Marcos** for contributing this image!

```
public class Solution {
        public int trap(int[] A) {
                if (A.length < 3)
                        return 0;
                int ans = 0;
                int l = 0, r = A.length - 1;
                while (l < r && A[l] <= A[l + 1])
                        l++;
                while (l < r && A[r] <= A[r - 1])
                        r--;
                while (l < r) {
                        int left = A[l];
                        int right = A[r];
                        if (left <= right)
                                while (l < r && left >= A[++l])
                                        ans += left - A[l];
                        else
                                while (l < r && A[--r] <= right)
                                        ans += right - A[r];
                }
                return ans;
        }
}
```

43. Multiply Strings

Given two non-negative integers `num1` and `num2` represented as strings, return the product of `num1` and `num2`.

Note:

1. The length of both `num1` and `num2` is < 110.
2. Both `num1` and `num2` contains only digits `0-9`.
3. Both `num1` and `num2` does not contain any leading zero.
4. You **must not use any built-in BigInteger library** or **convert the inputs to integer** directly.

```
public class Solution {
        public String multiply(String num1, String num2) {
                int m = num1.length(), n = num2.length();
                int[] pos = new int[m + n];
                for (int i = m - 1; i >= 0; i--) {
                        for (int j = n - 1; j >= 0; j--) {
                                int mul = (num1.charAt(i) - '0') * (num2.charAt(j) - '0');
                                int p1 = i + j, p2 = i + j + 1;
                                int sum = mul + pos[p2];
                                pos[p1] += sum / 10;
                                pos[p2] = (sum) % 10;
                        }
```

```
                }
                StringBuilder sb = new StringBuilder();
                for (int p : pos)
                        if (!(sb.length() == 0 && p == 0))
                                sb.append(p);
                return sb.length() == 0 ? "0" : sb.toString();
        }
}
```

44. Wildcard Matching

Implement wildcard pattern matching with support for `'?'` and `'*'`.

```
'?' Matches any single character.
'*' Matches any sequence of characters (including the empty sequence).
The matching should cover the entire input string (not partial).
The function prototype should be:
bool isMatch(const char *s, const char *p)
Some examples:
isMatch("aa","a") ? false
isMatch("aa","aa") ? true
isMatch("aaa","aa") ? false
isMatch("aa", "*") ? true
isMatch("aa", "a*") ? true
isMatch("ab", "?*") ? true
isMatch("aab", "c*a*b") ? false
```

```
public class Solution {
    public boolean isMatch(String s, String p) {
        int sp = 0, pp = 0, match = 0, starIdx = -1;
        while (sp < s.length()) {
            if (pp < p.length()
                    && (p.charAt(pp) == '?' || s.charAt(sp) == p.charAt(pp))) {
                sp++;
                pp++;
            }
            else if (pp < p.length() && p.charAt(pp) == '*') {
                starIdx = pp;
                match = sp;
                pp++;
            }
            else if (starIdx != -1) {
                pp = starIdx + 1;
                match++;
                sp = match;
            }
            else
                return false;
        }
        while (pp < p.length() && p.charAt(pp) == '*')
            pp++;
        return pp == p.length();
    }
}
public class Solution {
    public boolean isMatch(String s, String p) {
        int m = s.length(), n = p.length();
        boolean[][] dp = new boolean[m + 1][n + 1];
        dp[0][0] = true;
        for (int i = 1; i <= m; i++)
            dp[i][0] = false;
        for (int j = 1; j <= n; j++) {
            if (p.charAt(j - 1) == '*')
                dp[0][j] = true;
            else
                break;
        }
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (p.charAt(j - 1) != '*')
                    dp[i][j] = dp[i - 1][j - 1]
                            && (s.charAt(i - 1) == p.charAt(j - 1)
                                    || p.charAt(j - 1) == '?');
                else
                    dp[i][j] = dp[i - 1][j] || dp[i][j - 1];
            }
        }
        return dp[m][n];
    }
```

45. Jump Game II

Given an array of non-negative integers, you are initially positioned at the first index of the array.
Each element in the array represents your maximum jump length at that position.
Your goal is to reach the last index in the minimum number of jumps.
For example:
Given array A = `[2,3,1,1,4]`
The minimum number of jumps to reach the last index is `2`. (Jump `1` step from index 0 to 1, then `3` steps to the last index.)
Note:
You can assume that you can always reach the last index.

```
public class Solution {
        public int jump(int[] nums) {
                int step = 0;
                int curMaxP = 0;
                int farthest = 0;
                if (nums.length == 1) {
                        return 0;
                }
                for (int i = 0; i < nums.length;) {
                        if (nums[i] + i >= nums.length - 1)
                                return step + 1;
                        for (int j = i; j < nums.length && j <= nums[i] + i; ++j) {
                                if (farthest <= nums[j] + j) {
                                        farthest = nums[j] + j;
                                        curMaxP = j;
                                }
                        }
                        ++step;
                        i = curMaxP == i ? curMaxP + 1 : curMaxP;
                }
                return -1;
        }
}
```

46. Permutations

Given a collection of **distinct** numbers, return all possible permutations.
For example,
`[1,2,3]` have the following permutations:

```
[ [1,2,3],
  [1,3,2],
  [2,1,3],
  [2,3,1],
  [3,1,2],
  [3,2,1]]
```

```
public class Solution {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> list = new ArrayList<>();
        backtrack(list, new ArrayList<>(), nums);
        return list;
    }
    private static void backtrack(List<List<Integer>> list,
            List<Integer> tempList, int[] nums) {
        if (tempList.size() == nums.length) {
            list.add(new ArrayList<>(tempList));
        } else {
            for (int i = 0; i < nums.length; i++) {
                if (tempList.contains(nums[i]))
                    continue; // element already exists, skip
                tempList.add(nums[i]);
                backtrack(list, tempList, nums);
                tempList.remove(tempList.size() - 1);
            }
        }
    }
}
```

47. Permutations II

Given a collection of numbers that might contain duplicates, return all possible unique permutations.
For example,
`[1,1,2]` have the following unique permutations:

```
[ [1,1,2],
  [1,2,1],
  [2,1,1]]
```

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

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

48. Rotate Image

You are given an *n* x *n* 2D matrix representing an image.
Rotate the image by 90 degrees (clockwise).
Follow up:
Could you do this in-place?

```
public class Solution {
    public void rotate(int[][] matrix) {
        if(matrix.length ==0 ||matrix.length != matrix[0].length ) return;
        int n = matrix.length;
        int temp, offset;
        for(int i=0;i<=n/2; ++i) {
            int first = i;
            int last = n-i-1;
            for(int j=first;j<last;++j) {
                offset = j-i;
                temp = matrix[first][j];
                matrix[first][j] = matrix[last-offset][first];
                matrix[last-offset][first] = matrix[last][last-offset];
                matrix[last][last-offset] = matrix[j][last];
                matrix[j][last] = temp;
            }
        }
    }
}
```

49. Group Anagrams

Given an array of strings, group anagrams together.
For example, given: `["eat", "tea", "tan", "ate", "nat", "bat"]`,
Return:

```
[ ["ate", "eat","tea"],
  ["nat","tan"],
  ["bat"]]
```

Note: All inputs will be in lower-case.

```
public class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
        List<List<String>> r = new ArrayList<List<String>>();
        Map<String,List<String>> result = new HashMap<String,List<String>>();
        for(String str:strs) {
            char[] strC = str.toCharArray();
            Arrays.sort(strC);
            String key = new String(strC);
            List<String> l = result.getOrDefault(key,new ArrayList<String>());
            l.add(str);
            result.put(key, l);
        }
        r.addAll(result.values());
        return r;
    }
}
```

50. Pow(x, n)

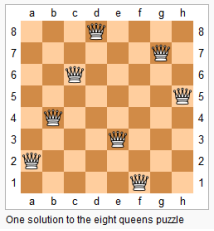
Implement pow(*x*, *n*).

```
public class Solution {
    public double myPow(double x, int n) {
```

```
        long m = (long)n;
        if(n == 0)
            return 1;
        if(n<0){
            m = -m;
            x = 1/x;
        }
        return (m%2 == 0) ? myPow(x*x, (int)(m/2)) : x*myPow(x*x, (int)(m/2));
    }
}
```

51. N-Queens

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



One solution to the eight queens puzzle

Given an integer *n*, return all distinct solutions to the *n*-queens puzzle.
Each solution contains a distinct board configuration of the *n*-queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space respectively.
For example,
There exist two distinct solutions to the 4-queens puzzle:

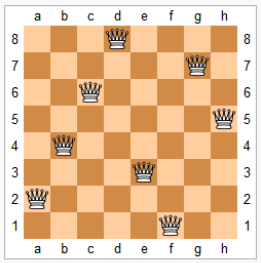
```
[[".Q..",  // Solution 1
  "...Q",
  "Q...",
  "..Q."],
 ["..Q.",  // Solution 2
  "Q...",
  "...Q",
  ".Q.."]]
```

```
public class Solution {
    public List<List<String>> solveNQueens(int n) {
        List<List<String>> r = new ArrayList<List<String>>();
        boolean[] col = new boolean[n];
        boolean[] lr = new boolean[n * 2];
        boolean[] rl = new boolean[n * 2];
        backtrack(0, n, col, lr, rl, r, new ArrayList<String>());
        return r;
    }
    private static void backtrack(int row, int n, boolean[] col, boolean[] lr,
            boolean[] rl, List<List<String>> r, List<String> l) {
        if (row == n) {
            List<String> resultL = new ArrayList<String>();
            resultL.addAll(l);
            r.add(resultL);
        }
        if (row == n)
            return;
        for (int i = 0; i < n; ++i) {
            int lrp = row - i + n;
            int rlp = n * 2 - i - row - 1;
            if (col[i] || lr[lrp] || rl[rlp]) {
                continue;
            }
            StringBuilder sb = new StringBuilder();
            for (int i1 = 0; i1 < n; ++i1) {
                if (i == i1) {
                    sb.append("Q");
                    continue;
                }
                sb.append(".");
            }
            l.add(sb.toString());
            col[i] = true;
            lr[lrp] = true;
            rl[rlp] = true;
            backtrack(row + 1, n, col, lr, rl, r, l);
            l.remove(l.size() - 1);
            col[i] = false;
            lr[lrp] = false;
            rl[rlp] = false;
```

```
        }
    }
}
```

52. N-Queens II

Follow up for N-Queens problem.
Now, instead outputting board configurations, return the total number of distinct solutions.



One solution to the eight queens puzzle

```
public class Solution {
    public int totalNQueens(int n) {
        return solveNQueens(n).size();
    }
    private void helper(int r, boolean[] cols, boolean[] d1, boolean[] d2,
            String[] board, List<String[]> res) {
        if (r == board.length)
            res.add(board.clone());
        else {
            for (int c = 0; c < board.length; c++) {
                int id1 = r - c + board.length,
                        id2 = 2 * board.length - r - c - 1;
                if (!cols[c] && !d1[id1] && !d2[id2]) {
                    char[] row = new char[board.length];
                    Arrays.fill(row, '.');
                    row[c] = 'Q';
                    board[r] = new String(row);
                    cols[c] = true;
                    d1[id1] = true;
                    d2[id2] = true;
                    helper(r + 1, cols, d1, d2, board, res);
                    cols[c] = false;
                    d1[id1] = false;
                    d2[id2] = false;
                }
            }
        }
    }
    public List<String[]> solveNQueens(int n) {
        List<String[]> res = new ArrayList<>();
        helper(0, new boolean[n], new boolean[2 * n], new boolean[2 * n],
                new String[n], res);
        return res;
    }
}
```

53. Maximum Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest sum.
For example, given the array `[-2,1,-3,4,-1,2,1,-5,4]`,
the contiguous subarray `[4,-1,2,1]` has the largest sum = `6`.
click to show more practice.
More practice:
If you have figured out the O(*n*) solution, try coding another solution using the divide and conquer approach, which is more subtle.

```
public class Solution {
    public int maxSubArray(int[] nums) {
        int max = Integer.MIN_VALUE;
        int sum = 0;
        for(int i = 0;i < nums.length;++i) {
            int cur = nums[i];
            sum += cur;
            max = Math.max(max, sum);
            if(sum < 0) {
                sum = 0;
                continue;
            }
        }
        return max;
    }
}
public class Solution {
```

```
    private class ArrayContext {
        int max;
        int lMax;
        int rMax;
        int sum;
    }
    public ArrayContext getArrayContext(int[] nums, int l, int r) {
        ArrayContext ctx = new ArrayContext();
        if (l == r) {
            ctx.max = nums[l];
            ctx.lMax = nums[l];
            ctx.rMax = nums[l];
            ctx.sum = nums[l];
        } else {
            int m = (l + r) / 2;
            ArrayContext lCtx = getArrayContext(nums, l, m);
            ArrayContext rCtx = getArrayContext(nums, m + 1, r);
            ctx.max = Math.max(Math.max(lCtx.max, rCtx.max),
                    lCtx.rMax + rCtx.lMax);
            ctx.lMax = Math.max(lCtx.lMax, lCtx.sum + rCtx.lMax);
            ctx.rMax = Math.max(rCtx.rMax, rCtx.sum + lCtx.rMax);
            ctx.sum = lCtx.sum + rCtx.sum;
        }
        return ctx;
    }
    public int maxSubArray(int[] nums) {
        if (nums.length == 0) {
            return 0;
        }
        ArrayContext ctx = getArrayContext(nums, 0, nums.length - 1);
        return ctx.max;
    }
}
```

54. Spiral Matrix

Given a matrix of *m* x *n* elements (*m* rows, *n* columns), return all elements of the matrix in spiral order.
For example,
Given the following matrix:

```
[[ 1, 2, 3 ],
 [ 4, 5, 6 ],
 [ 7, 8, 9 ]]
```

You should return `[1,2,3,6,9,8,7,4,5]`.

```
public class Solution {
        public List<Integer> spiralOrder(int[][] matrix) {
                List<Integer> res = new ArrayList<Integer>();
                if (matrix.length == 0)
                        return res;
                int rowBegin = 0;
                int rowEnd = matrix.length - 1;
                int colBegin = 0;
                int colEnd = matrix[0].length - 1;
                while (rowBegin <= rowEnd && colBegin <= colEnd) {
                        for (int j = colBegin; j <= colEnd; j++)
                                res.add(matrix[rowBegin][j]);
                        rowBegin++;
                        for (int j = rowBegin; j <= rowEnd; j++)
                                res.add(matrix[j][colEnd]);
                        colEnd--;
                        if (rowBegin <= rowEnd)
                                for (int j = colEnd; j >= colBegin; j--)
                                        res.add(matrix[rowEnd][j]);
                        rowEnd--;
                        if (colBegin <= colEnd)
                                for (int j = rowEnd; j >= rowBegin; j--)
                                        res.add(matrix[j][colBegin]);
                        colBegin++;
                }
                return res;
        }
}
```

55. Jump Game

Given an array of non-negative integers, you are initially positioned at the first index of the array.
Each element in the array represents your maximum jump length at that position.
Determine if you are able to reach the last index.
For example:

A = `[3,2,1,0,4]`, return `false`.

```
public class Solution {
    public boolean canJump(int[] nums) {
            int max = 0;
            for(int i=0;i<nums.length;i++){
                if(i>max) {return false;}
                max = Math.max(nums[i]+i,max);
            }
            return true;
    }
}
```

56. Merge Intervals

Given a collection of intervals, merge all overlapping intervals.
For example,
Given `[1,3],[2,6],[8,10],[15,18]`,
return `[1,6],[8,10],[15,18]`.

```
public class Solution {
    public List<Interval> merge(List<Interval> intervals) {
        if (intervals.size() <= 1)
            return intervals;
        intervals.sort((i1, i2) -> Integer.compare(i1.start, i2.start));
        List<Interval> result = new LinkedList<Interval>();
        int start = intervals.get(0).start;
        int end = intervals.get(0).end;
        for (Interval interval : intervals) {
            if (interval.start <= end)
                end = Math.max(end, interval.end);
            else {
                result.add(new Interval(start, end));
                start = interval.start;
                end = interval.end;
            }
        }
        result.add(new Interval(start, end));
        return result;
    }
}
```

57. Insert Interval

Given a set of *non-overlapping* intervals, insert a new interval into the intervals (merge if necessary).
You may assume that the intervals were initially sorted according to their start times.
Example 1:
Given intervals `[1,3],[6,9]`, insert and merge `[2,5]` in as `[1,5],[6,9]`.
Example 2:
Given `[1,2],[3,5],[6,7],[8,10],[12,16]`, insert and merge `[4,9]` in as `[1,2],[3,10],[12,16]`.
This is because the new interval `[4,9]` overlaps with `[3,5],[6,7],[8,10]`.

```
public class Solution {
    public List<Interval> insert(List<Interval> intervals,
            Interval newInterval) {
        int i = 0;
        while (i < intervals.size() && intervals.get(i).end < newInterval.start)
            i++;
        while (i < intervals.size()
                && intervals.get(i).start <= newInterval.end) {
            newInterval = new Interval(
                    Math.min(intervals.get(i).start, newInterval.start),
                    Math.max(intervals.get(i).end, newInterval.end));
            intervals.remove(i);
        }
        intervals.add(i, newInterval);
        return intervals;
    }
}
```

58. Length of Last Word

Given a string *s* consists of upper/lower-case alphabets and empty space characters `' '`, return the length of last word in the string.
If the last word does not exist, return 0.
Note: A word is defined as a character sequence consists of non-space characters only.
For example,
Given *s* = `"Hello World"`,
return `5`.

```
public class Solution {
    public int lengthOfLastWord(String s) {
        return s.trim().length()-s.trim().lastIndexOf(" ")-1;
    }
}
public class Solution {
```

```
        public int lengthOfLastWord(String s) {
                int lenIndex = s.length() - 1;
                int len = 0;
                for (int i = lenIndex; i >= 0 && s.charAt(i) == ' '; i--)
                        lenIndex--;
                for (int i = lenIndex; i >= 0 && s.charAt(i) != ' '; i--)
                        len++;
                return len;
        }
}
```

59. Spiral Matrix II

Given an integer *n*, generate a square matrix filled with elements from 1 to n^2 in spiral order.
For example,
Given *n* = 3,
You should return the following matrix:

```
[[ 1, 2, 3 ],
 [ 8, 9, 4 ],
 [ 7, 6, 5 ]]
```

```
public class Solution {
        public int[][] generateMatrix(int n) {
                int[][] ret = new int[n][n];
                int left = 0, top = 0;
                int right = n - 1, down = n - 1;
                int count = 1;
                while (left <= right) {
                        for (int j = left; j <= right; j++)
                                ret[top][j] = count++;
                        top++;
                        for (int i = top; i <= down; i++)
                                ret[i][right] = count++;
                        right--;
                        for (int j = right; j >= left; j--)
                                ret[down][j] = count++;
                        down--;
                        for (int i = down; i >= top; i--)
                                ret[i][left] = count++;
                        left++;
                }
                return ret;
        }
}
```

60. Permutation Sequence

The set `[1,2,3,…,n]` contains a total of *n*! unique permutations.
By listing and labeling all of the permutations in order,
We get the following sequence (ie, for *n* = 3):

1. `"123"`
2. `"132"`
3. `"213"`
4. `"231"`
5. `"312"`
6. `"321"`

Given *n* and *k*, return the k^{th} permutation sequence.
Note: Given *n* will be between 1 and 9 inclusive.

```
public class Solution {
        public String getPermutation(int n, int k) {
                int pos = 0;
                List<Integer> numbers = new ArrayList<>();
                int[] factorial = new int[n + 1];
                StringBuilder sb = new StringBuilder();
                int sum = 1;
                factorial[0] = 1;
                for (int i = 1; i <= n; i++) {
                        sum *= i;
                        factorial[i] = sum;
                }
                for (int i = 1; i <= n; i++) {
                        numbers.add(i);
                }
                k--;
                for (int i = 1; i <= n; i++) {
                        int index = k / factorial[n - i];
                        sb.append(String.valueOf(numbers.get(index)));
                        numbers.remove(index);
                        k -= index * factorial[n - i];
```

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

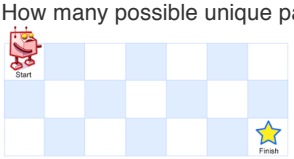
61. Rotate List

Given a list, rotate the list to the right by *k* places, where *k* is non-negative.
For example:
Given `1->2->3->4->5->NULL` and *k* = `2`,
return `4->5->1->2->3->NULL`.

```
public class Solution {
    public ListNode rotateRight(ListNode head, int k) {
        if (head == null || head.next == null)
            return head;
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode fast = dummy, slow = dummy;
        int i;
        for (i = 0; fast.next != null; i++)
            fast = fast.next;
        for (int j = i - k % i; j > 0; j--)
            slow = slow.next;
        fast.next = dummy.next;
        dummy.next = slow.next;
        slow.next = null;
        return dummy.next;
    }
}
```

62. Unique Paths

A robot is located at the top-left corner of a *m* x *n* grid (marked 'Start' in the diagram below).
The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).
How many possible unique paths are there?



Above is a 3 x 7 grid. How many possible unique paths are there?
Note: *m* and *n* will be at most 100.

```
public class Solution {
    public int uniquePaths(int m, int n) {
        int table[][] = new int[m][n];
        for(int i = 0; i < m;i++) {
            table[i][0] = 1;
        }
        for(int i = 1; i < n;i++) {
            table[0][i] = 1;
        }
        for(int row = 1; row < m;row++) {
            for(int column = 1; column < n;column++) {
                table[row][column] = table[row-1][column] + table[row][column - 1];
            }
        }
        return table[m-1][n-1];
    }
}
```

63. Unique Paths II

Follow up for "Unique Paths":
Now consider if some obstacles are added to the grids. How many unique paths would there be?
An obstacle and empty space is marked as `1` and `0` respectively in the grid.
For example,
There is one obstacle in the middle of a 3x3 grid as illustrated below.

```
[ [0,0,0],
  [0,1,0],
  [0,0,0]]
```

The total number of unique paths is `2`.
Note: *m* and *n* will be at most 100.

```
public class Solution {
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {
        int table[][] = obstacleGrid;
        int m = obstacleGrid.length;
        int n = obstacleGrid[0].length;
        for (int row = 0; row < m; row++) {
```

```
                if (table[row][column] == 1) {
                    table[row][column] = 0;
                } else if (column == 0 && row == 0 && table[row][column] != 1) {
                    table[row][column] = 1;
                } else if (column == 0) {
                    table[row][column] = table[row - 1][column];
                } else if (row == 0) {
                    table[row][column] = table[row][column - 1];
                } else
                    table[row][column] = table[row - 1][column]
                            + table[row][column - 1];
            }
        }
        return table[m - 1][n - 1];
    }
}
```

64. Minimum Path Sum

Given a *m* x *n* grid filled with non-negative numbers, find a path from top left to bottom right which *minimizes* the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

```
public class Solution {
    public int minPathSum(int[][] grid) {
        int m = grid.length;
        if(m == 0)
            return 0;
        int n = grid[0].length;
        for(int i = 0; i < m; ++i) {
            for(int j = 0; j < n; ++j) {
                if(i == 0 && j != 0)
                    grid[i][j] = grid[i][j-1] + grid[i][j];
                else if(i != 0 && j == 0)
                    grid[i][j] = grid[i-1][j] + grid[i][j];
                if(i != 0 && j != 0)
                    grid[i][j] = Math.min(grid[i][j-1], grid[i-1][j]) + grid[i][j];
            }
        }
        return grid[m-1][n-1];
    }
}
```

65. Valid Number

Validate if a given string is numeric.

Some examples:

`"0"` => `true`
`" 0.1 "` => `true`
`"abc"` => `false`
`"1 a"` => `false`
`"2e10"` => `true`

Note: It is intended for the problem statement to be ambiguous. You should gather all requirements up front before implementing one.

```
public class Solution {
    public boolean isNumber(String s) {
        if (s == null)
            return false;
        s = s.trim().toLowerCase();
        int n = s.length();
        if (n == 0)
            return false;
        int signCount = 0;
        boolean hasE = false;
        boolean hasNum = false;
        boolean hasPoint = false;
        for (int i = 0; i < n; i++) {
            char c = s.charAt(i);
            if (!isValid(c))
                return false;
            if (c >= '0' && c <= '9')
                hasNum = true;
            if (c == 'e') {
                if (hasE || !hasNum)
                    return false;
                if (i == n - 1)
                    return false;
                hasE = true;
            }
            if (c == '.') {
                if (hasPoint || hasE)
```

```
                return false;
            if (i == n - 1 && !hasNum)
                return false;
            hasPoint = true;
        }
        if (c == '+' || c == '-') {
            if (signCount == 2)
                return false;
            if (i == n - 1)
                return false;
            if (i > 0 && s.charAt(i - 1) != 'e')
                return false;
            signCount++;
        }
    }
    return true;
  }
  boolean isValid(char c) {
    return c == '.' || c == '+' || c == '-' || c == 'e'
            || c >= '0' && c <= '9';
  }
}
```

66. Plus One

Given a non-negative integer represented as a **non-empty** array of digits, plus one to the integer.
You may assume the integer do not contain any leading zero, except the number 0 itself.
The digits are stored such that the most significant digit is at the head of the list.

```
public class Solution {
    public int[] plusOne(int[] digits) {
        int n = digits.length;
        for(int i=n-1; i>=0; i--) {
            if(digits[i] < 9) {
                digits[i]++;
                return digits;
            }
            digits[i] = 0;
        }
        int[] newNumber = new int [n+1];
        newNumber[0] = 1;//only for 9999 like
        return newNumber;
    }
}
```

67. Add Binary

Given two binary strings, return their sum (also a binary string).
For example,
a = `"11"`
b = `"1"`
Return `"100"`.

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

68. Text Justification

Given an array of words and a length *L*, format the text such that each line has exactly *L* characters and is fully (left and right) justified.
You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces `' '` when necessary so that each line has exactly *L* characters.
Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line do not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right.
For the last line of text, it should be left justified and no extra space is inserted between words.
For example,
words: `["This", "is", "an", "example", "of", "text", "justification."]`
L: `16`.
Return the formatted lines as:

```
[  "This    is    an",
   "example  of text",
   "justification.  "]
```

Note: Each word is guaranteed not to exceed *L* in length.

click to show corner cases.

Corner Cases:

- A line other than the last line might contain only one word. What should you do in this case?
  In this case, that line should be left-justified.

```
public class Solution {
    public List<String> fullJustify(String[] words, int maxWidth) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < maxWidth; ++i)
            sb.append(" ");
        String pads = sb.toString();
        List<String> strs = new ArrayList<>();
        for (int i = 0, sum = 0, j = 0; i < words.length; i = j) {
            for (j = i + 1, sum = words[i].length(); j < words.length
                    && sum + j - i + words[j].length() <= maxWidth; ++j)
                sum += words[j].length();
            StringBuilder l = new StringBuilder();
            int n = j - 1 - i;
            int m = (j == words.length || 0 == n) ? 1 : ((maxWidth - sum) / n);
            int b = (j == words.length) ? 0 : (maxWidth - sum - m * n);
            for (int k = i; k < j - 1; ++k) {
                l.append(words[k]);
                l.append(pads.substring(0, (k - i < b) ? (m + 1) : m));
            }
            l.append(words[j - 1]);
            if (j == words.length || 0 == n) {
                l.append(pads.substring(0, maxWidth - sum - n));
            }
            strs.add(l.toString());
        }
        return strs;
    }
}
```

69. Sqrt(x)

Implement `int sqrt(int x)`.

Compute and return the square root of *x*.

```
public class Solution {
    public int mySqrt(int x) {
        if (x < 1) return 0;
        int left = 1, right = x;
        long mid;
        while(left+1 < right) {
            mid = left + (right-left) /2;
            if(mid*mid > x) {
                right = (int)mid;
            } else if(mid*mid < x) {
                left = (int)mid;
            } else
                return (int)mid;
        }
        return left;
    }
}
```

70. Climbing Stairs

You are climbing a stair case. It takes *n* steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Note: Given *n* will be a positive integer.

```
public class Solution {
    public int climbStairs(int n) {
        if(n<=1)
            return 1;
        if(n==2)
            return 2;
        int[] ways = new int[n];
        ways[0]=1;
        ways[1]=2;
        for(int i = 2;i<n;++i)
            ways[i] = ways[i-1] + ways[i-2];
        return ways[n-1];
    }
}
```

71. Simplify Path

Given an absolute path for a file (Unix-style), simplify it.
For example,
path = `"/home/"`, => `"/home"`
path = `"/a/./b/../../c/"`, => `"/c"`
click to show corner cases.
Corner Cases:

- Did you consider the case where **path** = `"/../"`?
  In this case, you should return `"/"`.
- Another corner case is the path might contain multiple slashes `'/'` together, such as `"/home//foo/"`.
  In this case, you should ignore redundant slashes and return `"/home/foo"`.

```
public class Solution {
    public String simplifyPath(String path) {
        while (path.contains("//")) {
            path = path.replace("//", "/");
        }
        String[] paths = path.split("/");
        List<String> pathList = new LinkedList<String>();
        for (String p : paths) {
            if (p.length() < 1)
                continue;
            if (p.equals(".")) {
            } else if (p.equals("..")) {
                if (pathList.size() > 0)
                    pathList.remove(pathList.size() - 1);
            } else
                pathList.add(p);
        }
        StringBuilder sb = new StringBuilder();
        sb.append("/");
        for (int i = 0; i < pathList.size(); ++i) {
            sb.append(pathList.get(i));
            if (i != pathList.size() - 1)
                sb.append("/");
        }
        return sb.toString();
    }
}
```

72. Edit Distance

Given two words *word1* and *word2*, find the minimum number of steps required to convert *word1* to *word2*. (each operation is counted as 1 step.)
You have the following 3 operations permitted on a word:
a) Insert a character
b) Delete a character
c) Replace a character

```
public class Solution {
    public int minDistance(String word1, String word2) {
        int m = word1.length();
        int n = word2.length();
        int[][] cost = new int[m + 1][n + 1];
        for(int i = 0; i <= m; i++)
            cost[i][0] = i;
        for(int i = 1; i <= n; i++)
            cost[0][i] = i;
        for(int i = 0; i < m; i++) {
            for(int j = 0; j < n; j++) {
                if(word1.charAt(i) == word2.charAt(j))
                    cost[i + 1][j + 1] = cost[i][j];
                else {
                    int a = cost[i][j];
                    int b = cost[i][j + 1];
                    int c = cost[i + 1][j];
                    cost[i + 1][j + 1] = a < b ? (a < c ? a : c) : (b < c ? b : c);
                    cost[i + 1][j + 1]++;
                }
            }
        }
        return cost[m][n];
    }
}
```

73. Set Matrix Zeroes

Given a *m* x *n* matrix, if an element is 0, set its entire row and column to 0. Do it in place.
click to show follow up.
Follow up:

Did you use extra space?
A straight forward solution using O(*mn*) space is probably a bad idea.
A simple improvement uses O(*m* + *n*) space, but still not the best solution.
Could you devise a constant space solution?

```
public class Solution {
    public void setZeroes(int[][] matrix) {
        boolean fr = false,fc = false;
        for(int i = 0; i < matrix.length; i++) {
            for(int j = 0; j < matrix[0].length; j++) {
                if(matrix[i][j] == 0) {
                    if(i == 0) fr = true;
                    if(j == 0) fc = true;
                    matrix[0][j] = 0;
                    matrix[i][0] = 0;
                }
            }
        }
        for(int i = 1; i < matrix.length; i++) {
            for(int j = 1; j < matrix[0].length; j++) {
                if(matrix[i][0] == 0 || matrix[0][j] == 0) {
                    matrix[i][j] = 0;
                }
            }
        }
        if(fr) {
            for(int j = 0; j < matrix[0].length; j++) {
                matrix[0][j] = 0;
            }
        }
        if(fc) {
            for(int i = 0; i < matrix.length; i++) {
                matrix[i][0] = 0;
            }
        }
    }
}
```

74. Search a 2D Matrix

Write an efficient algorithm that searches for a value in an *m* x *n* matrix. This matrix has the following properties:

- Integers in each row are sorted from left to right.
- The first integer of each row is greater than the last integer of the previous row.

For example,
Consider the following matrix:

```
[ [1,   3,  5,  7],
  [10, 11, 16, 20],
  [23, 30, 34, 50]]
```

Given **target** = `3`, return `true`.

```
public class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        int n = matrix.length;
        if(n == 0)
            return false;
        int m = matrix[0].length;
        if(m == 0)
            return false;
        int l = 0, r = m * n - 1;
        while (l != r){
            int mid = (l + r - 1) >> 1;
            if (matrix[mid / m][mid % m] < target)
                l = mid + 1;
            else
                r = mid;
        }
        return matrix[r / m][r % m] == target;
    }
}
```

75. Sort Colors

Given an array with *n* objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue.
Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.
Note:
You are not suppose to use the library's sort function for this problem.
click to show follow up.

Follow up:
A rather straight forward solution is a two-pass algorithm using counting sort.
First, iterate the array counting number of 0's, 1's, and 2's, then overwrite array with total number of 0's, then 1's and followed by 2's.
Could you come up with an one-pass algorithm using only constant space?

```
public class Solution {
    public void sortColors(int[] nums) {
        int r = 0;
        int w = 0;
        for(int i=0;i<nums.length;++i) {
            if(0==nums[i])
                ++r;
            if(1==nums[i])
                ++w;
        }
        for(int i=0;i<r;++i)
            nums[i] = 0;
        for(int i=r;i<r+w;++i)
            nums[i] = 1;
        for(int i=r+w;i<nums.length;++i)
            nums[i] = 2;
    }
}
```

76. Minimum Window Substring

Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity O(n).
For example,
S = `"ADOBECODEBANC"`
T = `"ABC"`
Minimum window is `"BANC"`.
Note:
If there is no such window in S that covers all characters in T, return the empty string `""`.
If there are multiple such windows, you are guaranteed that there will always be only one unique minimum window in S.

```
public class Solution {
    public String minWindow(String s, String t) {
        int[] map = new int[256];
        for (char c : t.toCharArray())
            map[c]++;
        int counter = t.length(), begin = 0, end = 0, d = Integer.MAX_VALUE,
                head = 0;
        while (end < s.length()) {
            if (map[s.charAt(end++)]-- > 0)
                counter--; // in t
            while (counter == 0) { // valid
                if (end - begin < d) {
                    head = begin;
                    d = end - head;
                }
                if (map[s.charAt(begin++)]++ == 0)
                    counter++; // make it invalid
            }
        }
        return d == Integer.MAX_VALUE ? "" : s.substring(head, head + d);
    }
}
```

77. Combinations

Given two integers *n* and *k*, return all possible combinations of *k* numbers out of 1 ... *n*.
For example,
If *n* = 4 and *k* = 2, a solution is:

```
[ [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],]
```

```
public class Solution {
    public List<List<Integer>> combine(int n, int k) {
        List<List<Integer>> combs = new ArrayList<List<Integer>>();
        combine(combs, new ArrayList<Integer>(), 1, n, k);
        return combs;
    }
    public static void combine(List<List<Integer>> combs, List<Integer> comb,
            int start, int n, int k) {
        if (k == 0) {
            combs.add(new ArrayList<Integer>(comb));
            return;
```

```
        }
        for (int i = start; i <= n; i++) {
            comb.add(i);
            combine(combs, comb, i + 1, n, k - 1);
            comb.remove(comb.size() - 1);
        }
    }
}
```

78. Subsets

Given a set of **distinct** integers, *nums*, return all possible subsets.
Note: The solution set must not contain duplicate subsets.
For example,
If ***nums*** = `[1,2,3]`, a solution is:

```
[ [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []]
```

```
public class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> list = new ArrayList<>();
        backtrack(list, new ArrayList<>(), nums, 0);
        return list;
    }
    private void backtrack(List<List<Integer>> list, List<Integer> tempList,
            int[] nums, int start) {
        list.add(new ArrayList<>(tempList));
        for (int i = start; i < nums.length; i++) {
            tempList.add(nums[i]);
            backtrack(list, tempList, nums, i + 1);
            tempList.remove(tempList.size() - 1);
        }
    }
}
```

79. Word Search

Given a 2D board and a word, find if the word exists in the grid.
The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.
For example,
Given **board** =

```
[ ['A','B','C','E'],
  ['S','F','C','S'],
  ['A','D','E','E']]
```

word = `"ABCCED"`, -> returns `true`,
word = `"SEE"`, -> returns `true`,
word = `"ABCB"`, -> returns `false`.

```
public class Solution {
    public boolean exist(char[][] board, String word) {
        if (word == null || word.length() == 0) {
            return true;
        }
        char[] chs = word.toCharArray();
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[0].length; j++) {
                if (dfs(board, chs, 0, i, j)) {
                    return true;
                }
            }
        }
        return false;
    }
    private boolean dfs(char[][] board, char[] words, int idx, int x, int y) {
        if (idx == words.length) {
            return true;
        }
        if (x < 0 || x == board.length || y < 0 || y == board[0].length) {
            return false;
        }
        if (board[x][y] != words[idx]) {
```

```
        }
        board[x][y] ^= 256;
        boolean exist = dfs(board, words, idx + 1, x, y + 1)
                || dfs(board, words, idx + 1, x, y - 1)
                || dfs(board, words, idx + 1, x + 1, y)
                || dfs(board, words, idx + 1, x - 1, y);
        board[x][y] ^= 256;
        return exist;
    }
}
```

80. Remove Duplicates from Sorted Array II

Follow up for "Remove Duplicates":
What if duplicates are allowed at most *twice*?
For example,
Given sorted array *nums* = `[1,1,1,2,2,3]`,
Your function should return length = `5`, with the first five elements of *nums* being `1`, `1`, `2`, `2` and `3`. It doesn't matter what you leave beyond the new length.

```
public class Solution {
        public int removeDuplicates(int[] nums) {
                int i = 0;
                for (int n : nums)
                        if (i < 2 || n > nums[i - 2])
                                nums[i++] = n;
                return i;
        }
}
```

81. Search in Rotated Sorted Array II

Follow up for "Search in Rotated Sorted Array":
What if *duplicates* are allowed?
Would this affect the run-time complexity? How and why?
Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.
(i.e., `0 1 2 4 5 6 7` might become `4 5 6 7 0 1 2`).
Write a function to determine if a given target is in the array.
The array may contain duplicates.

```
public class Solution {
    public boolean search(int[] nums, int target) {
        int start = 0, end = nums.length - 1, mid = -1;
        while (start <= end) {
            mid = (start + end) / 2;
            if (nums[mid] == target) {
                return true;
            }
            if (nums[mid] < nums[end] || nums[mid] < nums[start]) {
                if (target > nums[mid] && target <= nums[end]) {
                    start = mid + 1;
                } else {
                    end = mid - 1;
                }
            } else if (nums[mid] > nums[start] || nums[mid] > nums[end]) {
                if (target < nums[mid] && target >= nums[start]) {
                    end = mid - 1;
                } else {
                    start = mid + 1;
                }
            } else {
                end--;
            }
        }
        return false;
    }
}
```

82. Remove Duplicates from Sorted List II

Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only *distinct* numbers from the original list.
For example,
Given `1->2->3->3->4->4->5`, return `1->2->5`.
Given `1->1->1->2->3`, return `2->3`.

```
public class Solution {
        public ListNode deleteDuplicates(ListNode head) {
                if (head == null)
                        return null;
                ListNode dummyNode = new ListNode(0);
                dummyNode.next = head;
                ListNode pre = dummyNode;
                ListNode cur = head;
```

```
                        while (cur.next != null && cur.val == cur.next.val) {
                                cur = cur.next;
                        }
                        if (pre.next == cur) {
                                pre = pre.next;
                        } else {
                                pre.next = cur.next;
                        }
                        cur = cur.next;
                }
                return dummyNode.next;
        }
}
```

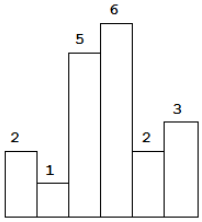
83. Remove Duplicates from Sorted List

Given a sorted linked list, delete all duplicates such that each element appear only *once*.
For example,
Given `1->1->2`, return `1->2`.
Given `1->1->2->3->3`, return `1->2->3`.

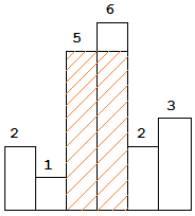
```
public class Solution {
        public ListNode deleteDuplicates(ListNode head) {
                ListNode dummyNode = new ListNode(0);
                dummyNode.next = head;
                while (head != null && head.next != null) {
                        while (head.val == head.next.val) {
                                head.next = head.next.next;
                                if (head.next == null)
                                        break;
                        }
                        head = head.next;
                }
                return dummyNode.next;
        }
}
```

84. Largest Rectangle in Histogram

Given *n* non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.



Above is a histogram where width of each bar is 1, given height = `[2,1,5,6,2,3]`.



The largest rectangle is shown in the shaded area, which has area = `10` unit.
For example,
Given heights = `[2,1,5,6,2,3]`,
return `10`.

```
public class Solution {
    public int largestRectangleArea(int[] heights) {
        if (heights == null || heights.length == 0) {
            return 0;
        }
        int[] lessFromLeft = new int[heights.length];
        int[] lessFromRight = new int[heights.length];
        lessFromRight[heights.length - 1] = heights.length;
        lessFromLeft[0] = -1;
        for (int i = 1; i < heights.length; i++) {
            int p = i - 1;
            while (p >= 0 && heights[p] >= heights[i]) {
                p = lessFromLeft[p];
            }
            lessFromLeft[i] = p;
        }
        for (int i = heights.length - 2; i >= 0; i--) {
            int p = i + 1;
```

```
            while (p < heights.length && heights[p] >= heights[i]) {
                p = lessFromRight[p];
            }
            lessFromRight[i] = p;
        }
        int maxArea = 0;
        for (int i = 0; i < heights.length; i++) {
            maxArea = Math.max(maxArea,
                    heights[i] * (lessFromRight[i] - lessFromLeft[i] - 1));
        }
        return maxArea;
    }
}
public class Solution {
    public int largestRectangleArea(int[] heights) {
        int len = heights.length;
        Stack<Integer> s = new Stack<Integer>();
        int maxArea = 0;
        for (int i = 0; i <= len; i++) {
            int h = (i == len ? 0 : heights[i]);
            if (s.isEmpty() || h >= heights[s.peek()]) {
                s.push(i);
            } else {
                int tp = s.pop();
                maxArea = Math.max(maxArea,
                        heights[tp] * (s.isEmpty() ? i : i - 1 - s.peek()));
                i--;
            }
        }
        return maxArea;
    }
}
```

85. Maximal Rectangle

Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle containing only 1's and return its area.
For example, given the following matrix:

```
1 0 1 0 0
1 0 1 1 1
1 1 1 1 1
1 0 0 1 0
```

Return 6.

```
public class Solution {
    public int maximalRectangle(char[][] matrix) {
        if (matrix.length == 0)
            return 0;
        int m = matrix.length;
        int n = matrix[0].length;
        int[] left = new int[n], right = new int[n], height = new int[n];
        for (int i = 0; i < n; ++i)
            right[i] = n;
        int maxA = 0;
        for (int i = 0; i < m; ++i) {
            int cur_left = 0, cur_right = n;
            for (int j = 0; j < n; j++) {
                if (matrix[i][j] == '1')
                    height[j]++;
                else
                    height[j] = 0;
            }
            for (int j = 0; j < n; j++) {
                if (matrix[i][j] == '1')
                    left[j] = Math.max(left[j], cur_left);
                else {
                    left[j] = 0;
                    cur_left = j + 1;
                }
            }
            for (int j = n - 1; j >= 0; j--) {
                if (matrix[i][j] == '1')
                    right[j] = Math.min(right[j], cur_right);
                else {
                    right[j] = n;
                    cur_right = j;
                }
            }
            for (int j = 0; j < n; j++)
```

```
        }
        return maxA;
    }
}
```

86. Partition List

Given a linked list and a value *x*, partition it such that all nodes less than *x* come before nodes greater than or equal to *x*.
You should preserve the original relative order of the nodes in each of the two partitions.
For example,
Given `1->4->3->2->5->2` and *x* = 3,
return `1->2->2->4->3->5`.

```
public class Solution {
    public ListNode partition(ListNode head, int x) {
        ListNode dummy1 = new ListNode(0), dummy2 = new ListNode(0);
        ListNode curr1 = dummy1, curr2 = dummy2;
        while (head != null) {
            if (head.val < x) {
                curr1.next = head;
                curr1 = head;
            } else {
                curr2.next = head;
                curr2 = head;
            }
            head = head.next;
        }
        curr2.next = null;
        curr1.next = dummy2.next;
        return dummy1.next;
    }
}
```

87. Scramble String

Given a string *s1*, we may represent it as a binary tree by partitioning it to two non-empty substrings recursively.
Below is one possible representation of *s1* = `"great"`:

```
    great
   /    \
  gr    eat
 / \    /  \
g   r  e   at
           / \
          a   t
```

To scramble the string, we may choose any non-leaf node and swap its two children.
For example, if we choose the node `"gr"` and swap its two children, it produces a scrambled string `"rgeat"`.

```
    rgeat
   /    \
  rg    eat
 / \    /  \
r   g  e   at
           / \
          a   t
```

We say that `"rgeat"` is a scrambled string of `"great"`.
Similarly, if we continue to swap the children of nodes `"eat"` and `"at"`, it produces a scrambled string `"rgtae"`.

```
    rgtae
   /    \
  rg    tae
 / \    /  \
r   g  ta  e
       / \
      t   a
```

We say that `"rgtae"` is a scrambled string of `"great"`.
Given two strings *s1* and *s2* of the same length, determine if *s2* is a scrambled string of *s1*.

```
public class Solution {
    public boolean isScramble(String s1, String s2) {
        if (s1.equals(s2))
            return true;
        int[] letters = new int[26];
        for (int i = 0; i < s1.length(); i++) {
            letters[s1.charAt(i) - 'a']++;
            letters[s2.charAt(i) - 'a']--;
        }
        for (int i = 0; i < 26; i++)
            if (letters[i] != 0)
                return false;
```

```
        for (int i = 1; i < s1.length(); i++) {
            if (isScramble(s1.substring(0, i), s2.substring(0, i))
                    && isScramble(s1.substring(i), s2.substring(i)))
                return true;
            if (isScramble(s1.substring(0, i), s2.substring(s2.length() - i))
                    && isScramble(s1.substring(i),
                            s2.substring(0, s2.length() - i)))
                return true;
        }
        return false;
    }
}
```

88. Merge Sorted Array

Given two sorted integer arrays *nums1* and *nums2*, merge *nums2* into *nums1* as one sorted array.

Note:

You may assume that *nums1* has enough space (size that is greater or equal to *m* + *n*) to hold additional elements from *nums2*. The number of elements initialized in *nums1* and *nums2* are *m* and *n* respectively.

```
public class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        for (int i = nums1.length - 1; i >= n; --i) {
            nums1[i] = nums1[i - n];
        }
        int k = 0;
        for (int i = n, j = 0; i < m + n || j < n;) {
            if (i >= m + n) {
                nums1[k++] = nums2[j++];
            } else if (j >= n) {
                nums1[k++] = nums1[i++];
            } else {
                nums1[k++] = nums1[i] > nums2[j] ? nums2[j++] : nums1[i++];
            }
        }
    }
}
```

89. Gray Code

The gray code is a binary numeral system where two successive values differ in only one bit.

Given a non-negative integer *n* representing the total number of bits in the code, print the sequence of gray code. A gray code sequence must begin with 0.

For example, given *n* = 2, return `[0,1,3,2]`. Its gray code sequence is:

```
00 - 0
01 - 1
11 - 3
10 - 2
```

Note:

For a given *n*, a gray code sequence is not uniquely defined.

For example, `[0,2,3,1]` is also a valid gray code sequence according to the above definition.

For now, the judge is able to judge based on one instance of gray code sequence. Sorry about that.

```
public class Solution {
        public List<Integer> grayCode(int n) {
                List<Integer> result = new LinkedList<>();
                for (int i = 0; i < 1 << n; i++)
                        result.add(i ^ i >> 1);
                return result;
        }
}
```

90. Subsets II

Given a collection of integers that might contain duplicates, ***nums***, return all possible subsets.

Note: The solution set must not contain duplicate subsets.

For example,

If ***nums*** = `[1,2,2]`, a solution is:

```
[ [2],
  [1],
  [1,2,2],
  [2,2],
  [1,2],
  []]
```

```
public class Solution {
    public List<List<Integer>> subsetsWithDup(int[] nums) {
        List<List<Integer>> list = new ArrayList<>();
        Arrays.sort(nums);
        backtrack(list, new ArrayList<>(), nums, 0);
        return list;
```

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

91. Decode Ways

A message containing letters from `A-Z` is being encoded to numbers using the following mapping:

```
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

Given an encoded message containing digits, determine the total number of ways to decode it.
For example,
Given encoded message `"12"`, it could be decoded as `"AB"` (1 2) or `"L"` (12).
The number of ways decoding `"12"` is 2.

```
public class Solution {
        public int numDecodings(String s) {
                int len = s.length();
                if (len == 0)
                        return 0;
                int[] dp = new int[len + 1];
                dp[0] = 1;
                dp[1] = s.charAt(0) == '0' ? 0 : 1;
                for (int i = 2; i <= len; i++) {
                        char c0 = s.charAt(i - 1);
                        char c1 = s.charAt(i - 2);
                        if (c0 != '0')
                                dp[i] += dp[i - 1];
                        if (c1 == '1' || (c1 == '2' && c0 <= '6'))
                                dp[i] += dp[i - 2];
                }
                return dp[len];
        }
}
```

92. Reverse Linked List II

Reverse a linked list from position *m* to *n*. Do it in-place and in one-pass.
For example:
Given `1->2->3->4->5->NULL`, *m* = 2 and *n* = 4,
return `1->4->3->2->5->NULL`.
Note:
Given *m*, *n* satisfy the following condition:
1 ? *m* ? *n* ? length of list.

```
public class Solution {
    public ListNode reverseBetween(ListNode head, int m, int n) {
        if (head == null)
            return null;
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode pre = dummy;
        for (int i = 0; i < m - 1; i++)
            pre = pre.next;
        ListNode start = pre.next;
        ListNode then = start.next;
        for (int i = 0; i < n - m; i++) {
            start.next = then.next;
            then.next = pre.next;
            pre.next = then;
            then = start.next;
        }
        return dummy.next;
    }
}
```

93. Restore IP Addresses

Given a string containing only digits, restore it by returning all possible valid IP address combinations.
For example:
Given `"`

return `["255.255.11.135", "255.255.111.35"]`. (Order does not matter)

```
public class Solution {
    public List<String> restoreIpAddresses(String s) {
        List<String> ret = new ArrayList<>();
        StringBuffer ip = new StringBuffer();
        for (int a = 1; a < 4; ++a)
            for (int b = 1; b < 4; ++b)
                for (int c = 1; c < 4; ++c)
                    for (int d = 1; d < 4; ++d) {
                        if (a + b + c + d == s.length()) {
                            int n1 = Integer.parseInt(s.substring(0, a));
                            int n2 = Integer.parseInt(s.substring(a, a + b));
                            int n3 = Integer
                                    .parseInt(s.substring(a + b, a + b + c));
                            int n4 = Integer.parseInt(s.substring(a + b + c));
                            if (n1 <= 255 && n2 <= 255 && n3 <= 255
                                    && n4 <= 255) {
                                ip.append(n1).append('.').append(n2).append('.')
                                        .append(n3).append('.').append(n4);
                                if (ip.length() == s.length() + 3)
                                    ret.add(ip.toString());
                                ip.delete(0, ip.length());
                            }
                        }
                    }
        return ret;
    }
}
```

94. Binary Tree Inorder Traversal

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

For example:

Given binary tree `[1,null,2,3]`,

```
   1
    \
     2
    /
   3
```

return `[1,3,2]`.

Note: Recursive solution is trivial, could you do it iteratively?

```
public class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> r = new ArrayList<Integer>();
        if (root != null)
            inorder(root, r);
        return r;
    }
    private static void inorder(TreeNode root, List<Integer> r) {
        if (root.left != null)
            inorder(root.left, r);
        r.add(root.val);
        if (root.right != null)
            inorder(root.right, r);
    }
}
```

95. Unique Binary Search Trees II

Given an integer *n*, generate all structurally unique **BST's** (binary search trees) that store values 1...*n*.

For example,

Given *n* = 3, your program should return all 5 unique BST's shown below.

```
   1         3     3      2      1
    \       /     /      / \      \
     3     2     1      1   3      2
    /     /       \                 \
   2     1         2                 3
```

```
public class Solution {
    public List<TreeNode> generateTrees(int n) {
        if (n == 0)
            return new ArrayList<TreeNode>();
        return genTrees(1, n);
    }
    public List<TreeNode> genTrees(int start, int end) {
        List<TreeNode> list = new ArrayList<TreeNode>();
        if (start > end) {
            list.add(null);
```

```
            return list;
        }
        if (start == end) {
            list.add(new TreeNode(start));
            return list;
        }
        List<TreeNode> left, right;
        for (int i = start; i <= end; i++) {
            left = genTrees(start, i - 1);
            right = genTrees(i + 1, end);
            for (TreeNode lnode : left) {
                for (TreeNode rnode : right) {
                    TreeNode root = new TreeNode(i);
                    root.left = lnode;
                    root.right = rnode;
                    list.add(root);
                }
            }
        }
        return list;
    }
}
```

96. Unique Binary Search Trees

Given *n*, how many structurally unique **BST's** (binary search trees) that store values 1...*n*?
For example,
Given *n* = 3, there are a total of 5 unique BST's.

```
   1         3     3      2      1
    \       /     /      / \      \
     3     2     1      1   3      2
    /     /       \                 \
   2     1         2                 3
```

```
public class Solution {
    public int numTrees(int n) {
        if (n <= 1)
            return 1;
        int[] c = new int[n + 1];
        c[0] = c[1] = 1;
        for (int i = 2; i <= n; ++i) {
            for (int j = 1; j <= i; ++j) {
                c[i] += c[j - 1] * c[i - j];
            }
        }
        return c[n];
    }
}
```

97. Interleaving String

Given *s1*, *s2*, *s3*, find whether *s3* is formed by the interleaving of *s1* and *s2*.
For example,
Given:
s1 = `"aabcc"`,
s2 = `"dbbca"`,
When *s3* = `"aadbbcbcac"`, return true.
When *s3* = `"aadbbbaccc"`, return false.

```
public class Solution {
    public boolean isInterleave(String s1, String s2, String s3) {
        if ((s1.length()+s2.length())!=s3.length()) return false;
        boolean[][] matrix = new boolean[s2.length()+1][s1.length()+1];
        matrix[0][0] = true;
        for (int i = 1; i < matrix[0].length; i++)
            matrix[0][i] = matrix[0][i-1]&&(s1.charAt(i-1)==s3.charAt(i-1));
        for (int i = 1; i < matrix.length; i++)
            matrix[i][0] = matrix[i-1][0]&&(s2.charAt(i-1)==s3.charAt(i-1));
        for (int i = 1; i < matrix.length; i++){
            for (int j = 1; j < matrix[0].length; j++){
                matrix[i][j] = (matrix[i-1][j]&&(s2.charAt(i-1)==s3.charAt(i+j-1)))
                        || (matrix[i][j-1]&&(s1.charAt(j-1)==s3.charAt(i+j-1)));
            }
        }
        return matrix[s2.length()][s1.length()];
    }
}
```

98. Validate Binary Search Tree

Given a binary tree, determine if it is a valid binary search tree (BST).
Assume a BST is defined as follows:

- The left subtree of a node contains only nodes with keys **less than** the node's key.
- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

Example 1:

```
  2
 / \
1   3
```

Binary tree `[2,1,3]`, return true.

Example 2:

```
  1
 / \
2   3
```

Binary tree `[1,2,3]`, return false.

```
public class Solution {
    public boolean isValidBST(TreeNode root) {
        return isValidBST(root, Long.MIN_VALUE, Long.MAX_VALUE);
    }
    public boolean isValidBST(TreeNode root, long minVal, long maxVal) {
        if (root == null)
            return true;
        if (root.val >= maxVal || root.val <= minVal)
            return false;
        return isValidBST(root.left, minVal, root.val)
                && isValidBST(root.right, root.val, maxVal);
    }
}
```

99. Recover Binary Search Tree

Two elements of a binary search tree (BST) are swapped by mistake.
Recover the tree without changing its structure.

Note:
A solution using O(*n*) space is pretty straight forward. Could you devise a constant space solution?

```
public class Solution {
    TreeNode firstElement = null;
    TreeNode secondElement = null;
    TreeNode prevElement = new TreeNode(Integer.MIN_VALUE);
    public void recoverTree(TreeNode root) {
        traverse(root);
        int temp = firstElement.val;
        firstElement.val = secondElement.val;
        secondElement.val = temp;
    }
    private void traverse(TreeNode root) {
        if (root == null)
            return;
        traverse(root.left);
        if (firstElement == null && prevElement.val >= root.val) {
            firstElement = prevElement;
        }
        if (firstElement != null && prevElement.val >= root.val) {
            secondElement = root;
        }
        prevElement = root;
        traverse(root.right);
    }
}
```

100.Same Tree

Given two binary trees, write a function to check if they are equal or not.
Two binary trees are considered equal if they are structurally identical and the nodes have the same value.

```
public class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {
        if (p == null && q == null)
            return true;
        if (p == null || q == null || p.val != q.val)
            return false;
        boolean result = true;
        if (p.left != null && q.left != null) {
            result = isSameTree(p.left, q.left);
        } else if (p.left != null || q.left != null) {
            return false;
        }
        if (!result) {
            return result;
```

```
        }
        if (p.right != null && q.right != null) {
            result = isSameTree(p.right, q.right);
        } else if (p.right != null || q.right != null) {
            return false;
        }
        return result;
    }
}
```

101.Symmetric Tree

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).
For example, this binary tree `[1,2,2,3,4,4,3]` is symmetric:

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

But the following `[1,2,2,null,3,null,3]` is not:

```
    1
   / \
  2   2
   \   \
   3    3
```

Note:
Bonus points if you could solve it both recursively and iteratively.

```
public class Solution {
    public boolean isSymmetric(TreeNode root) {
        return root == null || isSymmetricHelp(root.left, root.right);
    }
    private boolean isSymmetricHelp(TreeNode left, TreeNode right) {
        if (left == null || right == null)
            return left == right;
        if (left.val != right.val)
            return false;
        return isSymmetricHelp(left.left, right.right)
                && isSymmetricHelp(left.right, right.left);
    }
}
public class Solution {
    public boolean isSymmetric(TreeNode root) {
        if (root == null)
            return true;
        Stack<TreeNode> stack = new Stack<TreeNode>();
        TreeNode left, right;
        if (root.left != null) {
            if (root.right == null)
                return false;
            stack.push(root.left);
            stack.push(root.right);
        } else if (root.right != null) {
            return false;
        }
        while (!stack.empty()) {
            if (stack.size() % 2 != 0)
                return false;
            right = stack.pop();
            left = stack.pop();
            if (right.val != left.val)
                return false;
            if (left.left != null) {
                if (right.right == null)
                    return false;
                stack.push(left.left);
                stack.push(right.right);
            } else if (right.right != null) {
                return false;
            }
            if (left.right != null) {
                if (right.left == null)
                    return false;
                stack.push(left.right);
                stack.push(right.left);
            } else if (right.left != null) {
```

```
            }
        }
        return true;
    }
}
```

102.Binary Tree Level Order Traversal

Given a binary tree, return the *level order* traversal of its nodes' values. (ie, from left to right, level by level).
For example:
Given binary tree `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

return its level order traversal as:

```
[ [3],
  [9,20],
  [15,7]]
```

```
public class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        List<List<Integer>> wrapList = new LinkedList<List<Integer>>();
        if (root == null)
            return wrapList;
        queue.offer(root);
        while (!queue.isEmpty()) {
            int levelNum = queue.size();
            List<Integer> subList = new LinkedList<Integer>();
            for (int i = 0; i < levelNum; i++) {
                if (queue.peek().left != null)
                    queue.offer(queue.peek().left);
                if (queue.peek().right != null)
                    queue.offer(queue.peek().right);
                subList.add(queue.poll().val);
            }
            wrapList.add(subList);
        }
        return wrapList;
    }
}
```

103.Binary Tree Zigzag Level Order Traversal

Given a binary tree, return the *zigzag level order* traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).
For example:
Given binary tree `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

return its zigzag level order traversal as:

```
[ [3],
  [20,9],
  [15,7]]
```

```
public class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> sol = new ArrayList<>();
        travel(root, sol, 0);
        return sol;
    }
    private void travel(TreeNode curr, List<List<Integer>> sol, int level) {
        if (curr == null)
            return;
        if (sol.size() <= level) {
            List<Integer> newLevel = new LinkedList<>();
            sol.add(newLevel);
        }
        List<Integer> collection = sol.get(level);
        if (level % 2 == 0)
```

```
        else
            collection.add(0, curr.val);
        travel(curr.left, sol, level + 1);
        travel(curr.right, sol, level + 1);
    }
}
```

104.Maximum Depth of Binary Tree

Given a binary tree, find its maximum depth.
The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

```
public class Solution {
    public int maxDepth(TreeNode root) {
        return maxDepth(root, 0);
    }
    public int maxDepth(TreeNode root, int level) {
        if (root == null) {
            return level;
        }
        return Math.max(maxDepth(root.left, level + 1),
                maxDepth(root.right, level + 1));
    }
}
```

105.Construct Binary Tree from Preorder and Inorder Traversal

Given preorder and inorder traversal of a tree, construct the binary tree.
Note:
You may assume that duplicates do not exist in the tree.

```
public class Solution {
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        return helper(0, 0, inorder.length - 1, preorder, inorder);
    }
    public TreeNode helper(int preStart, int inStart, int inEnd, int[] preorder,
            int[] inorder) {
        if (preStart > preorder.length - 1 || inStart > inEnd) {
            return null;
        }
        TreeNode root = new TreeNode(preorder[preStart]);
        int inIndex = 0;
        for (int i = inStart; i <= inEnd; i++) {
            if (inorder[i] == root.val) {
                inIndex = i;
            }
        }
        root.left = helper(preStart + 1, inStart, inIndex - 1, preorder,
                inorder);
        root.right = helper(preStart + inIndex - inStart + 1, inIndex + 1,
                inEnd, preorder, inorder);
        return root;
    }
}
```

106.Construct Binary Tree from Inorder and Postorder Traversal

Given inorder and postorder traversal of a tree, construct the binary tree.
Note:
You may assume that duplicates do not exist in the tree.

```
public class Solution {
    public TreeNode buildTree(int[] inorder, int[] postorder) {
        return helper(postorder.length - 1, 0, inorder.length - 1, inorder,
                postorder);
    }
    private TreeNode helper(int postend, int instart, int inend, int[] inorder,
            int[] postorder) {
        if (postend < 0 || instart > inend)
            return null;
        TreeNode root = new TreeNode(postorder[postend]);
        int inindex = 0;
        for (int i = instart; i <= inend; i++) {
            if (root.val == inorder[i])
                inindex = i;
        }
        root.right = helper(postend - 1, inindex + 1, inend, inorder,
                postorder);
        root.left = helper(postend + inindex - inend - 1, instart, inindex - 1,
                inorder, postorder);
        return root;
    }
}
```

107.Binary Tree Level Order Traversal II

Given a binary tree, return the *bottom-up level order* traversal of its nodes' values. (ie, from left to right, level by level from leaf to root).

For example:

Given binary tree `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

return its bottom-up level order traversal as:

```
[ [15,7],
  [9,20],
  [3]]
```

```
public class Solution {
    public List<List<Integer>> levelOrderBottom(TreeNode root) {
        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        List<List<Integer>> wrapList = new LinkedList<List<Integer>>();
        if (root == null)
            return wrapList;
        queue.offer(root);
        while (!queue.isEmpty()) {
            int levelNum = queue.size();
            List<Integer> subList = new LinkedList<Integer>();
            for (int i = 0; i < levelNum; i++) {
                if (queue.peek().left != null)
                    queue.offer(queue.peek().left);
                if (queue.peek().right != null)
                    queue.offer(queue.peek().right);
                subList.add(queue.poll().val);
            }
            wrapList.add(0, subList);
        }
        return wrapList;
    }
}
```

108.Convert Sorted Array to Binary Search Tree

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

```
public class Solution {
    public TreeNode sortedArrayToBST(int[] nums) {
        if (nums.length == 0) {
            return null;
        }
        TreeNode head = helper(nums, 0, nums.length - 1);
        return head;
    }
    private static TreeNode helper(int[] nums, int low, int high) {
        if (low > high) {
            return null;
        }
        int mid = (low + high) / 2;
        TreeNode node = new TreeNode(nums[mid]);
        node.left = helper(nums, low, mid - 1);
        node.right = helper(nums, mid + 1, high);
        return node;
    }
}
```

109.Convert Sorted List to Binary Search Tree

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

```
public class Solution {
    private ListNode node;
    public TreeNode sortedListToBST(ListNode head) {
        if (head == null) {
            return null;
        }
        int size = 0;
        ListNode runner = head;
        node = head;
        while (runner != null) {
            runner = runner.next;
            size++;
        }
        return inorderHelper(0, size - 1);
```

```
    }
    public TreeNode inorderHelper(int start, int end) {
        if (start > end) {
            return null;
        }
        int mid = start + (end - start) / 2;
        TreeNode left = inorderHelper(start, mid - 1);
        TreeNode treenode = new TreeNode(node.val);
        treenode.left = left;
        node = node.next;
        TreeNode right = inorderHelper(mid + 1, end);
        treenode.right = right;
        return treenode;
    }
}
```

110.Balanced Binary Tree

Given a binary tree, determine if it is height-balanced.
For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of *every* node never differ by more than 1.

```
public class Solution {
    public boolean isBalanced(TreeNode root) {
        return height(root)!=-1;
    }
    public int height(TreeNode node){
        if(node==null)
            return 0;
        int lH=height(node.left);
        if(lH==-1)
            return -1;
        int rH=height(node.right);
        if(rH==-1)
            return -1;
        if(lH-rH<-1 || lH-rH>1)
            return -1;
        return Math.max(lH,rH)+1;
    }
}
```

111.Minimum Depth of Binary Tree

Given a binary tree, find its minimum depth.
The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

```
public class Solution {
    public int minDepth(TreeNode root) {
        if (root == null)
            return 0;
        int left = minDepth(root.left);
        int right = minDepth(root.right);
        return (left == 0 || right == 0) ? left + right + 1
                : Math.min(left, right) + 1;
    }
}
```

112.Path Sum

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.
For example:
Given the below binary tree and `sum = 22`,

```
              5
             / \
            4   8
           /   / \
          11  13  4
         /  \      \
        7    2      1
```

return true, as there exist a root-to-leaf path `5->4->11->2` which sum is 22.

```
public class Solution {
    public boolean hasPathSum(TreeNode root, int sum) {
        return hasPathSum(root, 0, sum);
    }
    private boolean hasPathSum(TreeNode root, int curSum, int sum) {
        if (root == null) {
            return false;
        }
        curSum += root.val;
        if (root.left == null && root.right == null) {
            if (curSum == sum) {
```

```
                return true;
            } else {
                return false;
            }
        }
        return hasPathSum(root.left, curSum, sum)
                || hasPathSum(root.right, curSum, sum);
    }
}
```

113.Path Sum II

Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.
For example:
Given the below binary tree and `sum = 22`,

```
              5
             / \
            4   8
           /   / \
          11  13  4
         /  \    / \
        7    2  5   1
```

return

```
[  [5,4,11,2],
   [5,8,4,5]]
```

```
public class Solution {
    public List<List<Integer>> pathSum(TreeNode root, int sum) {
        List<List<Integer>> resultList = new LinkedList<List<Integer>>();
        List<Integer> initList = new LinkedList<Integer>();
        pathSum(root, resultList, initList, 0, sum);
        return resultList;
    }
    private void pathSum(TreeNode root, List<List<Integer>> resultList,
            List<Integer> curList, int sum, int target) {
        if (root == null) {
            return;
        }
        curList.add(root.val);
        sum += root.val;
        if (root.left == null && root.right == null && sum == target) {
            resultList.add(curList);
        } else {
            List<Integer> newList = new LinkedList<Integer>();
            newList.addAll(curList);
            pathSum(root.left, resultList, curList, sum, target);
            pathSum(root.right, resultList, newList, sum, target);
        }
    }
}
```

114.Flatten Binary Tree to Linked List

Given a binary tree, flatten it to a linked list in-place.
For example,
Given

```
         1
        / \
       2   5
      / \   \
     3   4   6
```

The flattened tree should look like:

```
   1
    \
     2
      \
       3
        \
         4
          \
           5
            \
             6
```

Hints:
If you notice carefully in the flattened tree, each node's right child points to the next node of a pre-order traversal.

```
public class Solution {
    public void flatten(TreeNode root) {
        List<TreeNode> r = new ArrayList<TreeNode>();
        preOrderTraverse(root, r);
        TreeNode dummy = new TreeNode(0);
        TreeNode p = dummy;
        for (TreeNode n : r) {
            dummy.right = n;
            dummy = dummy.right;
            dummy.left = null;
        }
        root = p.right;
    }
    private static void preOrderTraverse(TreeNode root, List<TreeNode> r) {
        if (root == null) {
            return;
        }
        r.add(root);
        preOrderTraverse(root.left, r);
        preOrderTraverse(root.right, r);
    }
}
```

115.Distinct Subsequences

Given a string **S** and a string **T**, count the number of distinct subsequences of **S** which equals **T**.
A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, `"ACE"` is a subsequence of `"ABCDE"` while `"AEC"` is not).
Here is an example:
S = `"rabbbit"`, **T** = `"rabbit"`
Return `3`.

```
public class Solution {
    public int numDistinct(String s, String t) {
        int[][] mem = new int[t.length() + 1][s.length() + 1];
        for (int j = 0; j <= s.length(); j++)
            mem[0][j] = 1;
        for (int i = 0; i < t.length(); i++)
            for (int j = 0; j < s.length(); j++)
                if (t.charAt(i) == s.charAt(j))
                    mem[i + 1][j + 1] = mem[i][j] + mem[i + 1][j];
                else
                    mem[i + 1][j + 1] = mem[i + 1][j];
        return mem[t.length()][s.length()];
    }
}
```

116.Populating Next Right Pointers in Each Node

Given a binary tree

```
    struct TreeLinkNode {
      TreeLinkNode *left;
      TreeLinkNode *right;
      TreeLinkNode *next;
    }
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to `NULL`.
Initially, all next pointers are set to `NULL`.
Note:

- You may only use constant extra space.
- You may assume that it is a perfect binary tree (ie, all leaves are at the same level, and every parent has two children).

For example,
Given the following perfect binary tree,

```
         1
       /  \
      2    3
     / \  / \
    4  5  6  7
```

After calling your function, the tree should look like:

```
         1 -> NULL
       /  \
      2 -> 3 -> NULL
     / \  / \
    4->5->6->7 -> NULL
```

```
public class Solution {
    public void connect(TreeLinkNode root) {
        if (root == null)
            return;
        TreeLinkNode pre = root;
        TreeLinkNode cur = null;
        while (pre.left != null) {
            cur = pre;
            while (cur != null) {
                cur.left.next = cur.right;
                if (cur.next != null)
                    cur.right.next = cur.next.left;
                cur = cur.next;
            }
            pre = pre.left;
        }
    }
}
```

117.Populating Next Right Pointers in Each Node II

Follow up for problem "*Populating Next Right Pointers in Each Node*".
What if the given tree could be any binary tree? Would your previous solution still work?
Note:

- You may only use constant extra space.

For example,
Given the following binary tree,

```
         1
       /  \
      2    3
     / \    \
    4   5    7
```

After calling your function, the tree should look like:

```
         1 -> NULL
       /  \
      2 -> 3 -> NULL
     / \    \
    4-> 5 -> 7 -> NULL
```

```
public class Solution {
    public void connect(TreeLinkNode root) {
        TreeLinkNode head = null; // head of the next level
        TreeLinkNode prev = null; // the leading node on the next level
        TreeLinkNode cur = root; // current node of current level
        while (cur != null) {
            while (cur != null) { // iterate on the current level left child
                if (cur.left != null) {
                    if (prev != null) {
                        prev.next = cur.left;
                    } else {
                        head = cur.left;
                    }
                    prev = cur.left;
                }
                if (cur.right != null) {
                    if (prev != null) {
                        prev.next = cur.right;
                    } else {
                        head = cur.right;
                    }
                    prev = cur.right;
                }
                cur = cur.next;
            }
            cur = head;
            head = null;
            prev = null;
        }
    }
}
```

118.Pascal's Triangle

Given *numRows*, generate the first *numRows* of Pascal's triangle.
For example, given *numRows* = 5,
Return

```
[
```

```
    [1],
   [1,1],
  [1,2,1],
 [1,3,3,1],
[1,4,6,4,1]
]
```

```
public class Solution {
    public List<List<Integer>> generate(int numRows) {
        List<List<Integer>> allrows = new ArrayList<List<Integer>>();
        ArrayList<Integer> row = new ArrayList<Integer>();
        for (int i = 0; i < numRows; i++) {
            row.add(0, 1);
            for (int j = 1; j < row.size() - 1; j++)
                row.set(j, row.get(j) + row.get(j + 1));
            allrows.add(new ArrayList<Integer>(row));
        }
        return allrows;
    }
}
```

119.Pascal's Triangle II

Given an index *k*, return the k^{th} row of the Pascal's triangle.
For example, given *k* = 3,
Return `[1,3,3,1]`.
Note:
Could you optimize your algorithm to use only *O*(*k*) extra space?

```
public class Solution {
    public List<Integer> getRow(int rowIndex) {
        ArrayList<Integer> row = new ArrayList<Integer>();
        for (int i = 0; i < rowIndex + 1; i++) {
            row.add(0, 1);
            for (int j = 1; j < row.size() - 1; j++)
                row.set(j, row.get(j) + row.get(j + 1));
        }
        return row;
    }
}
```

120.Triangle

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.
For example, given the following triangle

```
[     [2],
    [3,4],
   [6,5,7],
  [4,1,8,3]]
```

The minimum path sum from top to bottom is `11` (i.e., 2 + 3 + 5 + 1 = 11).
Note:
Bonus point if you are able to do this using only *O*(*n*) extra space, where *n* is the total number of rows in the triangle.

```
public class Solution {
    public int minimumTotal(List<List<Integer>> triangle) {
        for (int i = triangle.size() - 2; i >= 0; i--)
            for (int j = 0; j <= i; j++)
                triangle.get(i).set(j,
                        triangle.get(i).get(j)
                                + Math.min(triangle.get(i + 1).get(j),
                                        triangle.get(i + 1).get(j + 1)));
        return triangle.get(0).get(0);
    }
}
```

121.Best Time to Buy and Sell Stock

Say you have an array for which the i^{th} element is the price of a given stock on day *i*.
If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.
Example 1:

```
Input: [7, 1, 5, 3, 6, 4]
Output: 5
max. difference = 6-1 = 5 (not 7-1 = 6, as selling price needs to be larger than buying price)
```

Example 2:

```
Input: [7, 6, 4, 3, 1]
Output: 0
In this case, no transaction is done, i.e. max profit = 0.
```

```
public class Solution {
    public int maxProfit(int[] prices) {
        int maxCur = 0, maxSoFar = 0;
        for (int i = 1; i < prices.length; i++) {
            maxCur = Math.max(0, maxCur += prices[i] - prices[i - 1]);
            maxSoFar = Math.max(maxCur, maxSoFar);
        }
        return maxSoFar;
    }
}
```

122.Best Time to Buy and Sell Stock II

Say you have an array for which the i^{th} element is the price of a given stock on day *i*.
Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

```
public class Solution {
    public int maxProfit(int[] prices) {
        int maxCur = 0, maxSoFar = 0;
        for (int i = 1; i < prices.length; i++) {
            maxCur = Math.max(0, prices[i] - prices[i - 1]);
            maxSoFar += maxCur;
        }
        return maxSoFar;
    }
}
```

123.Best Time to Buy and Sell Stock III

Say you have an array for which the i^{th} element is the price of a given stock on day *i*.
Design an algorithm to find the maximum profit. You may complete at most *two* transactions.
Note:
You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

```
public class Solution {
    public int maxProfit(int[] prices) {
        int buy1 = Integer.MIN_VALUE, buy2 = Integer.MIN_VALUE;
        int sell1 = 0, sell2 = 0;
        for (int i : prices) {
            sell2 = Math.max(sell2, buy2 + i);
            buy2 = Math.max(buy2, sell1 - i);
            sell1 = Math.max(sell1, buy1 + i);
            buy1 = Math.max(buy1, -i);
        }
        return sell2;
    }
}
```

124.Binary Tree Maximum Path Sum

Given a binary tree, find the maximum path sum.
For this problem, a path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The path must contain **at least one node** and does not need to go through the root.
For example:
Given the below binary tree,

```
       1
      / \
     2   3
```

Return 6.

```
public class Solution {
    int maxValue;
    public int maxPathSum(TreeNode root) {
        maxValue = Integer.MIN_VALUE;
        maxPathDown(root);
        return maxValue;
    }
    private int maxPathDown(TreeNode node) {
        if (node == null)
            return 0;
        int left = Math.max(0, maxPathDown(node.left));
        int right = Math.max(0, maxPathDown(node.right));
        maxValue = Math.max(maxValue, left + right + node.val);
        return Math.max(left, right) + node.val;
    }
}
```

125.Valid Palindrome

Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

For example,
`"A man, a plan, a canal: Panama"` is a palindrome.
`"race a car"` is *not* a palindrome.
Note:
Have you consider that the string might be empty? This is a good question to ask during an interview.
For the purpose of this problem, we define empty string as valid palindrome.

```
public class Solution {
    public boolean isPalindrome(String s) {
        if (s.isEmpty()) {
            return true;
        }
        int head = 0, tail = s.length() - 1;
        char cHead, cTail;
        while (head < tail) {
            cHead = s.charAt(head);
            cTail = s.charAt(tail);
            if (!Character.isLetterOrDigit(cHead)) {
                head++;
            } else if (!Character.isLetterOrDigit(cTail)) {
                tail--;
            } else {
                if (Character.toLowerCase(cHead) != Character
                        .toLowerCase(cTail)) {
                    return false;
                }
                head++;
                tail--;
            }
        }
        return true;
    }
}
```

126.Word Ladder II

Given two words (*beginWord* and *endWord*), and a dictionary's word list, find all shortest transformation sequence(s) from *beginWord* to *endWord*, such that:

1. Only one letter can be changed at a time
2. Each transformed word must exist in the word list. Note that *beginWord* is *not* a transformed word.

For example,
Given:
beginWord = `"hit"`
endWord = `"cog"`
wordList = `["hot","dot","dog","lot","log","cog"]`
Return

```
  [   ["hit","hot","dot","dog","cog"],
    ["hit","hot","lot","log","cog"]]
```

Note:

- Return an empty list if there is no such transformation sequence.
- All words have the same length.
- All words contain only lowercase alphabetic characters.
- You may assume no duplicates in the word list.
- You may assume *beginWord* and *endWord* are non-empty and are not the same.

```
public class Solution {
    public List<List<String>> findLadders(String beginWord, String endWord,
            List<String> wordList) {
        Set<String> start = new HashSet<>();
        Set<String> end = new HashSet<>();
        Set<String> dict = new HashSet<>();
        start.add(beginWord);
        end.add(endWord);
        dict.addAll(wordList);
        HashMap<String, List<String>> map = new HashMap<>();
        List<List<String>> res = new ArrayList<>();
        if (!dict.contains(endWord)) {
            return res;
        }
        buildMap(start, end, false, dict, map);
        List<String> path = new ArrayList<>();
        path.add(beginWord);
        genPath(beginWord, endWord, res, map, path);
        return res;
    }
    private void genPath(String start, String end, List<List<String>> ans,
```

```
        if (start.equals(end)) {
            ans.add(new ArrayList<>(temp));
            return;
        }
        if (!map.containsKey(start))
            return;
        for (String s : map.get(start)) {
            temp.add(s);
            genPath(s, end, ans, map, temp);
            temp.remove(temp.size() - 1);
        }
    }
    private void buildMap(Set<String> start, Set<String> end,
            boolean reverse, Set<String> dict,
            HashMap<String, List<String>> map) {
        if (start.size() == 0)
            return;
        if (start.size() > end.size()) {
            buildMap(end, start, !reverse, dict, map);
            return;
        }
        dict.removeAll(start);
        boolean finished = false;
        HashSet<String> next = new HashSet<>();
        for (String word : start) {
            char[] arr = word.toCharArray();
            for (int i = 0; i < arr.length; i++) {
                char old = arr[i];
                for (char c = 'a'; c <= 'z'; c++) {
                    if (c == old)
                        continue;
                    arr[i] = c;
                    String newString = new String(arr);
                    if (dict.contains(newString)) {
                        if (end.contains(newString))
                            finished = true;
                        else
                            next.add(newString);
                        String parent = reverse ? newString : word;
                        String child = reverse ? word : newString;
                        List<String> neighbor = map.getOrDefault(parent,
                                new ArrayList<String>());
                        neighbor.add(child);
                        map.put(parent, neighbor);
                    }
                }
                arr[i] = old;
            }
        }
        if (!finished)
            buildMap(next, end, reverse, dict, map);
    }
}
```

127.Word Ladder

Given two words (*beginWord* and *endWord*), and a dictionary's word list, find the length of shortest transformation sequence from *beginWord* to *endWord*, such that:

1. Only one letter can be changed at a time.
2. Each transformed word must exist in the word list. Note that *beginWord* is *not* a transformed word.

For example,
Given:
beginWord = `"hit"`
endWord = `"cog"`
wordList = `["hot","dot","dog","lot","log","cog"]`
As one shortest transformation is `"hit" -> "hot" -> "dot" -> "dog" -> "cog"`,
return its length `5`.
Note:

- Return 0 if there is no such transformation sequence.
- All words have the same length.
- All words contain only lowercase alphabetic characters.
- You may assume no duplicates in the word list.
- You may assume *beginWord* and *endWord* are non-empty and are not the same.

```
public class Solution {
    public int ladderLength(String beginWord, String endWord,
            List<String> wordList) {
        if (!wordList.contains(endWord))
```

```
            return 0;
        Set<String> startSet = new HashSet<String>(),
                endSet = new HashSet<String>(), dictSet = new HashSet<String>();
        startSet.add(beginWord);
        endSet.add(endWord);
        int len = 1;
        for (String temp : wordList) {
            dictSet.add(temp);
        }
        while (!startSet.isEmpty() && !endSet.isEmpty()) {
            if (startSet.size() > endSet.size()) {
                Set<String> tmpSet = startSet;
                startSet = endSet;
                endSet = tmpSet;
            }
            Set<String> tmp = new HashSet<String>();
            for (String word : startSet) {
                char[] charArr = word.toCharArray();
                for (int i = 0; i < word.length(); i++) {
                    for (char c = 'a'; c <= 'z'; c++) {
                        char replace = charArr[i];
                        charArr[i] = c;
                        String s = new String(charArr);
                        if (endSet.contains(s))
                            return len + 1;
                        if (dictSet.contains(s)) {
                            tmp.add(s);
                            dictSet.remove(s);
                        }
                        charArr[i] = replace;// change it back
                    }
                }
            }
            startSet = tmp;
            len++;
        }
        return 0;
    }
}
```

128.Longest Consecutive Sequence

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.
For example,
Given `[100, 4, 200, 1, 3, 2]`,
The longest consecutive elements sequence is `[1, 2, 3, 4]`. Return its length: `4`.
Your algorithm should run in O(*n*) complexity.

```
public class Solution {
    public int longestConsecutive(int[] num) {
        int res = 0;
        HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
        for (int n : num) {
            if (!map.containsKey(n)) {
                int left = (map.containsKey(n - 1)) ? map.get(n - 1) : 0;
                int right = (map.containsKey(n + 1)) ? map.get(n + 1) : 0;
                int sum = left + right + 1;
                map.put(n, sum);
                res = Math.max(res, sum);
                map.put(n - left, sum);
                map.put(n + right, sum);
            } else {
                continue;
            }
        }
        return res;
    }
}
```

129.Sum Root to Leaf Numbers

Given a binary tree containing digits from `0-9` only, each root-to-leaf path could represent a number.
An example is the root-to-leaf path `1->2->3` which represents the number `123`.
Find the total sum of all root-to-leaf numbers.
For example,

```
    1
   / \
  2   3
```

The root-to-leaf path `1->2` represents the number `12`.

Return the sum = 12 + 13 = `25`.

```
public class Solution {
    public int sumNumbers(TreeNode root) {
        return sum(root, 0);
    }
    public int sum(TreeNode n, int s) {
        if (n == null)
            return 0;
        if (n.right == null && n.left == null)
            return s * 10 + n.val;
        return sum(n.left, s * 10 + n.val) + sum(n.right, s * 10 + n.val);
    }
}
```

130.Surrounded Regions

Given a 2D board containing `'X'` and `'O'` (the **letter** O), capture all regions surrounded by `'X'`.
A region is captured by flipping all `'O'`s into `'X'`s in that surrounded region.
For example,

```
X X X X
X O O X
X X O X
X O X X
```

After running your function, the board should be:

```
X X X X
X X X X
X X X X
X O X X
```

```
public class Solution {
    public void solve(char[][] board) {
        if (board.length < 2 || board[0].length < 2)
            return;
        int m = board.length, n = board[0].length;
        for (int i = 0; i < m; i++) {
            if (board[i][0] == 'O')
                boundaryDFS(board, i, 0);
            if (board[i][n - 1] == 'O')
                boundaryDFS(board, i, n - 1);
        }
        for (int j = 0; j < n; j++) {
            if (board[0][j] == 'O')
                boundaryDFS(board, 0, j);
            if (board[m - 1][j] == 'O')
                boundaryDFS(board, m - 1, j);
        }
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (board[i][j] == 'O')
                    board[i][j] = 'X';
                else if (board[i][j] == '*')
                    board[i][j] = 'O';
            }
        }
     }
    private void boundaryDFS(char[][] board, int i, int j) {
        if (i < 0 || i > board.length - 1 || j < 0 || j > board[0].length - 1)
            return;
        if (board[i][j] == 'O')
            board[i][j] = '*';
        if (i > 1 && board[i - 1][j] == 'O')
            boundaryDFS(board, i - 1, j);
        if (i < board.length - 2 && board[i + 1][j] == 'O')
            boundaryDFS(board, i + 1, j);
        if (j > 1 && board[i][j - 1] == 'O')
            boundaryDFS(board, i, j - 1);
        if (j < board[i].length - 2 && board[i][j + 1] == 'O')
            boundaryDFS(board, i, j + 1);
    }
}
```

131.Palindrome Partitioning

Given a string *s*, partition *s* such that every substring of the partition is a palindrome.
Return all possible palindrome partitioning of *s*.
For example, given *s* = `"aab"`,
Return

```
[ ["aa","b"],
  ["a","a","b"]]
```

```
public class Solution {
    List<List<String>> resultLst;
    ArrayList<String> currLst;
    public List<List<String>> partition(String s) {
        resultLst = new ArrayList<List<String>>();
        currLst = new ArrayList<String>();
        backTrack(s, 0);
        return resultLst;
    }
    public void backTrack(String s, int l) {
        if (currLst.size() > 0
                && l >= s.length()) {
            List<String> r = (ArrayList<String>) currLst.clone();
            resultLst.add(r);
        }
        for (int i = l; i < s.length(); i++) {
            if (isPalindrome(s, l, i)) {
                if (l == i)
                    currLst.add(Character.toString(s.charAt(i)));
                else
                    currLst.add(s.substring(l, i + 1));
                backTrack(s, i + 1);
                currLst.remove(currLst.size() - 1);
            }
        }
    }
    public boolean isPalindrome(String str, int l, int r) {
        if (l == r)
            return true;
        while (l < r) {
            if (str.charAt(l) != str.charAt(r))
                return false;
            l++;
            r--;
        }
        return true;
    }
}
```

132.Palindrome Partitioning II

Given a string *s*, partition *s* such that every substring of the partition is a palindrome.
Return the minimum cuts needed for a palindrome partitioning of *s*.
For example, given *s* = `"aab"`,
Return `1` since the palindrome partitioning `["aa","b"]` could be produced using 1 cut.

```
public class Solution {
    public int minCut(String s) {
        char[] c = s.toCharArray();
        int n = c.length;
        int[] cut = new int[n];
        boolean[][] pal = new boolean[n][n];
        for(int i = 0; i < n; i++) {
            int min = i;
            for(int j = 0; j <= i; j++) {
                if(c[j] == c[i] && (i - j < 2 || pal[j + 1][i - 1])) {
                    pal[j][i] = true;
                    min = j == 0 ? 0 : Math.min(min, cut[j - 1] + 1);
                }
            }
            cut[i] = min;
        }
        return cut[n - 1];
    }
}
```

```
public class Solution {
    public int minCut(String s) {
        int n = s.length();
        int[] cut = new int[n + 1];
        for (int i = 0; i <= n; i++)
            cut[i] = i - 1;
        for (int i = 0; i < n; i++) {
            for (int j = 0; i - j >= 0 && i + j < n
                    && s.charAt(i - j) == s.charAt(i + j); j++) // odd
                cut[i + j + 1] = Math.min(cut[i + j + 1], 1 + cut[i - j]);
```

```
                    && s.charAt(i - j + 1) == s.charAt(i + j); j++) // even
                cut[i + j + 1] = Math.min(cut[i + j + 1], 1 + cut[i - j + 1]);
        }
        return cut[n];
    }
}
```

133.Clone Graph

Clone an undirected graph. Each node in the graph contains a `label` and a list of its `neighbors`.

OJ's undirected graph serialization:

Nodes are labeled uniquely.

We use `#` as a separator for each node, and `,` as a separator for node label and each neighbor of the node.

As an example, consider the serialized graph `{0,1,2#1,2#2,2}`.

The graph has a total of three nodes, and therefore contains three parts as separated by `#`.

1. First node is labeled as `0`. Connect node `0` to both nodes `1` and `2`.
2. Second node is labeled as `1`. Connect node `1` to node `2`.
3. Third node is labeled as `2`. Connect node `2` to node `2` (itself), thus forming a self-cycle.

Visually, the graph looks like the following:

```
       1
      / \
     /   \
    0 --- 2
         / \
         \_/
```

```
public class Solution {
    public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {
        if (node == null)
            return null;
        UndirectedGraphNode newNode = new UndirectedGraphNode(node.label);
        HashMap<Integer, UndirectedGraphNode> map = new HashMap<>();
        map.put(newNode.label, newNode);
        LinkedList<UndirectedGraphNode> queue = new LinkedList<>();
        queue.add(node);
        while (!queue.isEmpty()) {
            UndirectedGraphNode n = queue.pop();
            for (UndirectedGraphNode neighbor : n.neighbors) {
                if (!map.containsKey(neighbor.label)) {
                    map.put(neighbor.label,
                            new UndirectedGraphNode(neighbor.label));
                    queue.add(neighbor);
                }
                map.get(n.label).neighbors.add(map.get(neighbor.label));
            }
        }
        return newNode;
    }
}
```

134.Gas Station

There are *N* gas stations along a circular route, where the amount of gas at station *i* is `gas[i]`.

You have a car with an unlimited gas tank and it costs `cost[i]` of gas to travel from station *i* to its next station (*i*+1). You begin the journey with an empty tank at one of the gas stations.

Return the starting gas station's index if you can travel around the circuit once, otherwise return -1.

Note:

The solution is guaranteed to be unique.

```
public class Solution {
    public int canCompleteCircuit(int[] gas, int[] cost) {
        int start = gas.length, end = 0, sum = 0;
        do
            sum += sum > 0 ? gas[end] - cost[end++]
                    : gas[--start] - cost[start];
        while (start != end);
        return sum >= 0 ? start : -1;
    }
}
```

135.Candy

There are *N* children standing in a line. Each child is assigned a rating value.

You are giving candies to these children subjected to the following requirements:

- Each child must have at least one candy.
- Children with a higher rating get more candies than their neighbors.

What is the minimum candies you must give?

```
public class Solution {
    public int candy(int[] ratings) {
        int len = ratings.length;
```

```
        candy[0] = 1;
        for (int i = 1; i < len; ++i) {
            if (ratings[i] > ratings[i - 1]) {
                candy[i] = candy[i - 1] + 1;
            } else {
                candy[i] = 1;
            }
        }
        int total = candy[len - 1];
        for (int i = len - 2; i >= 0; --i) {
            if (ratings[i] > ratings[i + 1] && candy[i] <= candy[i + 1]) {
                candy[i] = candy[i + 1] + 1;
            }
            total += candy[i];
        }
        return total;
    }
}
```

136.Single Number

Given an array of integers, every element appears *twice* except for one. Find that single one.

Note:

Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

```
public class Solution {
    public int singleNumber(int[] nums) {
        int result = 0;
        for(int num:nums) {
            result ^= num;
        }
        return result;
    }
}
```

137.Single Number II

Given an array of integers, every element appears *three* times except for one, which appears exactly once. Find that single one.

Note:

Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

```
public class Solution {
    public int singleNumber(int[] nums) {
        int ones = 0, twos = 0;
        for (int num : nums) {
            ones = (ones ^ num) & ~twos;
            twos = (twos ^ num) & ~ones;
        }
        return ones;
    }
}
```

138.Copy List with Random Pointer

A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null.

Return a deep copy of the list.

```
public class Solution {
    public RandomListNode copyRandomList(RandomListNode head) {
        if (head == null)
            return null;
        Map<RandomListNode, RandomListNode> map = new HashMap<RandomListNode, RandomListNode>();
        RandomListNode node = head;
        while (node != null) {
            map.put(node, new RandomListNode(node.label));
            node = node.next;
        }
        node = head;
        while (node != null) {
            map.get(node).next = map.get(node.next);
            map.get(node).random = map.get(node.random);
            node = node.next;
        }
        return map.get(head);
    }
}
```

139.Word Break

Given a **non-empty** string *s* and a dictionary *wordDict* containing a list of **non-empty** words, determine if *s* can be segmented into a space-separated sequence of one or more dictionary words. You may assume the dictionary does not contain duplicate words.

For example, given

s = `"leetcode"`,

dict = `["leet", "code"]`.

Return true because `"leetcode"` can be segmented as `"leet code"`.

```
public class Solution {
```

```
    public boolean wordBreak(String s, List<String> wordDict) {
        boolean[] f = new boolean[s.length() + 1];
        f[0] = true;
        for (int i = 1; i <= s.length(); i++) {
            for (int j = 0; j < i; j++) {
                if (f[j] && wordDict.contains(s.substring(j, i))) {
                    f[i] = true;
                    break;
                }
            }
        }
        return f[s.length()];
    }
}
```

140.Word Break II

Given a **non-empty** string *s* and a dictionary *wordDict* containing a list of **non-empty** words, add spaces in *s* to construct a sentence where each word is a valid dictionary word. You may assume the dictionary does not contain duplicate words.
Return all such possible sentences.
For example, given
s = `"catsanddog"`,
dict = `["cat", "cats", "and", "sand", "dog"]`.
A solution is `["cats and dog", "cat sand dog"]`.

```
public class Solution {
    public List<String> wordBreak(String s, List<String> wordDict) {
        return DFS(s, wordDict, new HashMap<String, LinkedList<String>>());
    }
    List<String> DFS(String s, List<String> wordDict,
            HashMap<String, LinkedList<String>> map) {
        if (map.containsKey(s))
            return map.get(s);
        LinkedList<String> res = new LinkedList<String>();
        if (s.length() == 0) {
            res.add("");
            return res;
        }
        for (String word : wordDict) {
            if (s.startsWith(word)) {
                List<String> sublist = DFS(s.substring(word.length()), wordDict,
                        map);
                for (String sub : sublist)
                    res.add(word + (sub.isEmpty() ? "" : " ") + sub);
            }
        }
        map.put(s, res);
        return res;
    }
}
```

141.Linked List Cycle

Given a linked list, determine if it has a cycle in it.
Follow up:
Can you solve it without using extra space?

```
public class Solution {
    public boolean hasCycle(ListNode head) {
        if (head == null)
            return false;
        ListNode walker = head;
        ListNode runner = head;
        while (runner.next != null && runner.next.next != null) {
            walker = walker.next;
            runner = runner.next.next;
            if (walker == runner)
                return true;
        }
        return false;
    }
}
```

142.Linked List Cycle II

Given a linked list, return the node where the cycle begins. If there is no cycle, return `null`.
Note: Do not modify the linked list.
Follow up:
Can you solve it without using extra space?

```
public class Solution {
    public ListNode detectCycle(ListNode head) {
        if (head == null || head.next == null)
            return null;
```

```
        pointer1 = pointer2 = pointer3 = head;
        Boolean inited = false;
        for (; pointer2 != null;) {
            pointer1 = pointer1.next;
            pointer2 = pointer2.next;
            if (pointer2 == null)
                return null;
            else
                pointer2 = pointer2.next;
            if (pointer1 == pointer2 && inited)
                break;
            if (!inited)
                inited = true;
        }
        if (pointer2 == null)
            return null;
        while (pointer1 != pointer3) {
            pointer1 = pointer1.next;
            pointer3 = pointer3.next;
        }
        return pointer1;
    }
}
```

143.Reorder List

Given a singly linked list L: $L_0?L_1?\dots?L_{n-1}?L_n$,
reorder it to: $L_0?L_n?L_1?L_{n-1}?L_2?L_{n-2}?\dots$
You must do this in-place without altering the nodes' values.
For example,
Given `{1,2,3,4}`, reorder it to `{1,4,2,3}`.

```
public class Solution {
    public void reorderList(ListNode head) {
        if (head == null || head.next == null)
            return;
        ListNode p1 = head;
        ListNode p2 = head;
        while (p2.next != null && p2.next.next != null) {
            p1 = p1.next;
            p2 = p2.next.next;
        }
        ListNode preMiddle = p1;
        ListNode preCurrent = p1.next;
        while (preCurrent.next != null) {
            ListNode current = preCurrent.next;
            preCurrent.next = current.next;
            current.next = preMiddle.next;
            preMiddle.next = current;
        }
        p1 = head;
        p2 = preMiddle.next;
        while (p1 != preMiddle) {
            preMiddle.next = p2.next;
            p2.next = p1.next;
            p1.next = p2;
            p1 = p2.next;
            p2 = preMiddle.next;
        }
    }
}
```

144.Binary Tree Preorder Traversal

Given a binary tree, return the *preorder* traversal of its nodes' values.
For example:
Given binary tree `{1,#,2,3}`,

```
   1
    \
     2
    /
   3
```

return `[1,2,3]`.
Note: Recursive solution is trivial, could you do it iteratively?

```
public class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        Stack<TreeNode> stack = new Stack<>();
        List<Integer> traversal = new ArrayList<>();
        if (root != null) {
```

```
            while (!stack.isEmpty()) {
                TreeNode curr = stack.pop();
                traversal.add(curr.val);
                if (curr.right != null) {
                    stack.push(curr.right);
                }
                if (curr.left != null) {
                    stack.push(curr.left);
                }
            }
        }
        return traversal;
    }
}
```

145.Binary Tree Postorder Traversal

Given a binary tree, return the *postorder* traversal of its nodes' values.
For example:
Given binary tree `{1,#,2,3}`,

```
   1
    \
     2
    /
   3
```

return `[3,2,1]`.
Note: Recursive solution is trivial, could you do it iteratively?

```
public class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        LinkedList<Integer> ans = new LinkedList<>();
        Stack<TreeNode> stack = new Stack<>();
        if (root == null)
            return ans;
        stack.push(root);
        while (!stack.isEmpty()) {
            TreeNode cur = stack.pop();
            ans.addFirst(cur.val);
            if (cur.left != null) {
                stack.push(cur.left);
            }
            if (cur.right != null) {
                stack.push(cur.right);
            }
        }
        return ans;
    }
}
```

146.LRU Cache

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: `get` and `put`.
`get(key)` - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.
`put(key, value)` - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.
Follow up:
Could you do both operations in **O(1)** time complexity?
Example:

```
LRUCache cache = new LRUCache( 2 /* capacity */ );
cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // returns 1
cache.put(3, 3);    // evicts key 2
cache.get(2);       // returns -1 (not found)
cache.put(4, 4);    // evicts key 1
cache.get(1);       // returns -1 (not found)
cache.get(3);       // returns 3
cache.get(4);       // returns 4
```

```
public class LRUCache {
    private LinkedHashMap<Integer, Integer> map;
    private final int CAPACITY;
    public LRUCache(int capacity) {
        CAPACITY = capacity;
        map = new LinkedHashMap<Integer, Integer>(capacity, 0.75f, true) {
            @Override
            protected boolean removeEldestEntry(Map.Entry eldest) {
```

```
            }
        };
    }
    public int get(int key) {
        return map.getOrDefault(key, -1);
    }
    public void put(int key, int value) {
        map.put(key, value);
    }
}
```

147.Insertion Sort List

Sort a linked list using insertion sort.

```
public class Solution {
    public ListNode insertionSortList(ListNode head) {
        if( head == null ){
            return head;
        }

        ListNode helper = new ListNode(0);
        ListNode cur = head;
        ListNode pre = helper;
        ListNode next = null;
        while( cur != null ){
            next = cur.next;
            while( pre.next != null && pre.next.val < cur.val ){
                pre = pre.next;
            }
            cur.next = pre.next;
            pre.next = cur;
            pre = helper;
            cur = next;
        }

        return helper.next;
    }
}
```

148.Sort List

Sort a linked list in *O*(*n* log *n*) time using constant space complexity.

```
public class Solution {
    public ListNode sortList(ListNode head) {
        if (head == null || head.next == null)
            return head;
        ListNode prev = null, slow = head, fast = head;
        while (fast != null && fast.next != null) {
            prev = slow;
            slow = slow.next;
            fast = fast.next.next;
        }
        prev.next = null;
        ListNode l1 = sortList(head);
        ListNode l2 = sortList(slow);
        return merge(l1, l2);
    }
    ListNode merge(ListNode l1, ListNode l2) {
        ListNode l = new ListNode(0), p = l;
        while (l1 != null && l2 != null) {
            if (l1.val < l2.val) {
                p.next = l1;
                l1 = l1.next;
            } else {
                p.next = l2;
                l2 = l2.next;
            }
            p = p.next;
        }
        if (l1 != null)
            p.next = l1;
        if (l2 != null)
            p.next = l2;
        return l.next;
    }
}
```

149.Max Points on a Line

Given *n* points on a 2D plane, find the maximum number of points that lie on the same straight line.

```
public class Solution {
    private int gcd(int a, int b) {
```

```
            return b;
        return gcd(b % a, a);
    }
    public int maxPoints(Point[] points) {
        if (points.length <= 0)
            return 0;
        if (points.length <= 2)
            return points.length;
        int result = 0;
        for (int i = 0; i < points.length; i++) {
            Map<String, Integer> hm = new HashMap<>();
            int samex = 1;
            int samey = 1;
            int samep = 0;
            boolean sameSome = false;
            for (int j = 0; j < points.length; j++) {
                if (j != i) {
                    if ((points[j].x == points[i].x)
                            && (points[j].y == points[i].y))
                        samep++;
                    if (points[j].x == points[i].x) {
                        samex++;
                        sameSome = true;
                    }
                    if (points[j].y == points[i].y) {
                        samey++;
                        sameSome = true;
                    }
                    if(sameSome) {
                        sameSome = false;
                        continue;
                    }
                    int numerator = points[j].y - points[i].y;
                    int denaminator = points[j].x - points[i].x;
                    int gcd = gcd(numerator, denaminator);
                    String hashStr = (numerator / gcd) + "_"
                            + (denaminator / gcd);
                    hm.put(hashStr, hm.getOrDefault(hashStr, 1) + 1);
                    result = Math.max(result, hm.get(hashStr) + samep);
                }
            }
            result = Math.max(result, Math.max(samex, samey));
        }
        return result;
    }
}
```

150.Evaluate Reverse Polish Notation

Evaluate the value of an arithmetic expression in Reverse Polish Notation.
Valid operators are `+`, `-`, `*`, `/`. Each operand may be an integer or another expression.
Some examples:

```
  ["2", "1", "+", "3", "*"] -> ((2 + 1) * 3) -> 9
  ["4", "13", "5", "/", "+"] -> (4 + (13 / 5)) -> 6
```

```
public class Solution {
    public int evalRPN(String[] tokens) {
        int a, b;
        Stack<Integer> S = new Stack<Integer>();
        for (String s : tokens) {
            if (s.equals("+")) {
                S.add(S.pop() + S.pop());
            } else if (s.equals("/")) {
                b = S.pop();
                a = S.pop();
                S.add(a / b);
            } else if (s.equals("*")) {
                S.add(S.pop() * S.pop());
            } else if (s.equals("-")) {
                b = S.pop();
                a = S.pop();
                S.add(a - b);
            } else {
                S.add(Integer.parseInt(s));
            }
        }
        return S.pop();
    }
```

151.Reverse Words in a String

Given an input string, reverse the string word by word.
For example,
Given s = "`the sky is blue`",
return "`blue is sky the`".

Update (2015-02-12):
For C programmers: Try to solve it *in-place* in *O*(1) space.
click to show clarification.

Clarification:

- What constitutes a word?
  A sequence of non-space characters constitutes a word.
- Could the input string contain leading or trailing spaces?
  Yes. However, your reversed string should not contain leading or trailing spaces.
- How about multiple spaces between two words?
  Reduce them to a single space in the reversed string.

```
public class Solution {
    public String reverseWords(String s) {
        String[] strs = s.split(" ");
        StringBuilder sb = new StringBuilder();
        for (int i = strs.length - 1; i >= 0; --i) {
            if (strs[i].length() > 0) {
                sb.append(strs[i]);
                sb.append(" ");
            }
        }
        return sb.toString().trim();
    }
}
```

```
public class Solution {
    public String reverseWords(String s) {
        if (s == null)
            return null;
        char[] a = s.toCharArray();
        int n = a.length;
        reverse(a, 0, n - 1);
        reverseWords(a, n);
        return cleanSpaces(a, n);
    }
    void reverseWords(char[] a, int n) {
        int i = 0, j = 0;
        while (i < n) {
            while (i < j || i < n && a[i] == ' ')
                i++; // skip spaces
            while (j < i || j < n && a[j] != ' ')
                j++; // skip non spaces
            reverse(a, i, j - 1); // reverse the word
        }
    }
    String cleanSpaces(char[] a, int n) {
        int i = 0, j = 0;
        while (j < n) {
            while (j < n && a[j] == ' ')
                j++; // skip spaces
            while (j < n && a[j] != ' ')
                a[i++] = a[j++]; // keep non spaces
            while (j < n && a[j] == ' ')
                j++; // skip spaces
            if (j < n)
                a[i++] = ' '; // keep only one space
        }
        return new String(a).substring(0, i);
    }
    private void reverse(char[] a, int i, int j) {
        while (i < j) {
            char t = a[i];
            a[i++] = a[j];
            a[j--] = t;
        }
    }
}
```

152.Maximum Product Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest product.
For example, given the array `[2,3,-2,4]`,
the contiguous subarray `[2,3]` has the largest product = `6`.

```
public class Solution {
```

```
    public int maxProduct(int[] nums) {
        int maxProduct = nums[0], tmp = 0;
        for (int i = 1, max = maxProduct, min = maxProduct; i < nums.length; i++) {
            if (nums[i] < 0) {
                tmp = min;
                min = max;
                max = tmp;
            }
            max = Math.max(nums[i], max * nums[i]);
            min = Math.min(nums[i], min * nums[i]);
            maxProduct = Math.max(maxProduct, max);
        }
        return maxProduct;
    }
}
```

153.Find Minimum in Rotated Sorted Array

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.
(i.e., `0 1 2 4 5 6 7` might become `4 5 6 7 0 1 2`).
Find the minimum element.
You may assume no duplicate exists in the array.

```
public class Solution {
    public int findMin(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }
        if (nums.length == 1) {
            return nums[0];
        }
        int start = 0, end = nums.length - 1;
        while (start < end) {
            int mid = (start + end) / 2;
            if (mid > 0 && nums[mid] < nums[mid - 1]) {
                return nums[mid];
            }
            if (nums[start] <= nums[mid] && nums[mid] > nums[end]) {
                start = mid + 1;
            } else {
                end = mid - 1;
            }
        }
        return nums[start];
    }
}
```

154.Find Minimum in Rotated Sorted Array II

Follow up for "Find Minimum in Rotated Sorted Array":
What if *duplicates* are allowed?
Would this affect the run-time complexity? How and why?
Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.
(i.e., `0 1 2 4 5 6 7` might become `4 5 6 7 0 1 2`).
Find the minimum element.
The array may contain duplicates.

```
public class Solution {
    public int findMin(int[] nums) {
        int l = 0, r = nums.length - 1;
        while (l < r) {
            int mid = (l + r) / 2;
            if (nums[mid] < nums[r]) {
                r = mid;
            } else if (nums[mid] > nums[r]) {
                l = mid + 1;
            } else {
                r--;
            }
        }
        return nums[l];
    }
}
```

155.Min Stack

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

- push(x) -- Push element x onto stack.
- pop() -- Removes the element on top of the stack.
- top() -- Get the top element.
- getMin() -- Retrieve the minimum element in the stack.

```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();   --> Returns -3.
minStack.pop();
minStack.top();      --> Returns 0.
minStack.getMin();   --> Returns -2.
```

```
public class MinStack {
    Stack<Integer> stack = new Stack<Integer>();
    Stack<Integer> minStack = new Stack<Integer>();
    public void push(int x) {
        stack.push(x);
        if (minStack.isEmpty())
            minStack.push(x);
        else
            if (minStack.peek() > x)
                minStack.push(x);
            else
                minStack.push(minStack.peek());
     }
    public void pop() {
        if (!stack.isEmpty()) {
            stack.pop();
            minStack.pop();
        }
    }
    public int top() {
        return stack.peek();
    }
    public int getMin() {
        return minStack.peek();
    }
}
```

156.Binary Tree Upside Down

Given a binary tree where all the right nodes are either leaf nodes with a sibling (a left node that shares the same parent node) or empty, flip it upside down and turn it into a tree where the original right nodes turned into left leaf nodes. Return the new root.
For example:
Given a binary tree `{1,2,3,4,5}`,

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

return the root of the binary tree `[4,5,2,#,#,3,1]`.

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

confused what `"{1,#,2,3}"` means? > read more on how binary tree is serialized on OJ.

```
public class Solution {
    public TreeNode upsideDownBinaryTree(TreeNode root) {
        if (root == null || root.left == null) {
            return root;
        }
        TreeNode newRoot = upsideDownBinaryTree(root.left);
        root.left.left = root.right;
        root.left.right = root;
        root.left = null;
        root.right = null;
        return newRoot;
    }
}
public class Solution {
    public TreeNode upsideDownBinaryTree(TreeNode root) {
        TreeNode curr = root;
        TreeNode next = null;
        TreeNode temp = null;
        TreeNode prev = null;
```

```
            next = curr.left;
            curr.left = temp;
            temp = curr.right;
            curr.right = prev;
            prev = curr;
            curr = next;
        }
        return prev;
    }
}
```

157.Read N Characters Given Read4

The API: `int read4(char *buf)` reads 4 characters at a time from a file.
The return value is the actual number of characters read. For example, it returns 3 if there is only 3 characters left in the file.
By using the `read4` API, implement the function `int read(char *buf, int n)` that reads *n* characters from the file.
Note:
The `read` function will only be called once for each test case.

```
public class Solution extends Reader4 {
    public int read(char[] buf, int n) {
        char[] tmp = new char[4];
        int res = 0, count = -1;
        while (res < n) {
            count = read4(tmp);
            if (count == 0)
                break;
            int r = n - res < count ? n - res : count;
            for (int i = 0; i < r; i++) {
                buf[res++] = tmp[i];
            }
        }
        return res;
    }
}
```

158.Read N Characters Given Read4 II - Call multiple times

The API: `int read4(char *buf)` reads 4 characters at a time from a file.
The return value is the actual number of characters read. For example, it returns 3 if there is only 3 characters left in the file.
By using the `read4` API, implement the function `int read(char *buf, int n)` that reads *n* characters from the file.
Note:
The `read` function may be called multiple times.

```
public class Solution extends Reader4 {
    private int buffPtr = 0;
    private int buffCnt = 0;
    private char[] buff = new char[4];
    public int read(char[] buf, int n) {
        int ptr = 0;
        while (ptr < n) {
            if (buffPtr == 0) {
                buffCnt = read4(buff);
            }
            if (buffCnt == 0)
                break;
            while (ptr < n && buffPtr < buffCnt) {
                buf[ptr++] = buff[buffPtr++];
            }
            if (buffPtr >= buffCnt)
                buffPtr = 0;
        }
        return ptr;
    }
}
```

159.Longest Substring with At Most Two Distinct Characters

Given a string, find the length of the longest substring T that contains at most 2 distinct characters.
For example, Given s = “eceba”,
T is "ece" which its length is 3.

```
public class Solution {
    public int lengthOfLongestSubstringTwoDistinct(String s) {
        if (s.isEmpty())
            return 0;
        int max = 1;
        int p1 = 0, p2 = 0;
        int last = 1;
        char[] chars = s.toCharArray();
        for (int i = 1; i < chars.length; i++) {
            if (p1 != p2 && chars[i] != chars[p1] && chars[i] != chars[p2]) {
                if (last > max)
                    max = last;
```

```
                p1 = p2;
                p2 = i;
            } else {
                if (chars[i] == chars[p1]) {
                    p1 = p1 == p2 ? i : p2;
                }
                last++;
                p2 = i;
            }
        }
        if (last > max)
            max = last;
        return max;
    }
}
```

160.Intersection of Two Linked Lists

Write a program to find the node at which the intersection of two singly linked lists begins.

For example, the following two linked lists:

```
A:          a1 → a2
                   ↘
                     c1 → c2 → c3
                   ↗
B:     b1 → b2 → b3
```

begin to intersect at node c1.

Notes:

- If the two linked lists have no intersection at all, return `null`.
- The linked lists must retain their original structure after the function returns.
- You may assume there are no cycles anywhere in the entire linked structure.
- Your code should preferably run in O(n) time and use only O(1) memory.

```
public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        if (headA == null || headB == null)
            return null;
        ListNode a = headA;
        ListNode b = headB;
        while (a != b) {
            a = a == null ? headB : a.next;
            b = b == null ? headA : b.next;
        }
        return a;
    }
}
```

161.One Edit Distance

Given two strings S and T, determine if they are both one edit distance apart.

```
public class Solution {
    public boolean isOneEditDistance(String s, String t) {
        for (int i = 0; i < Math.min(s.length(), t.length()); i++) {
            if (s.charAt(i) != t.charAt(i)) {
                if (s.length() == t.length())
                    return s.substring(i + 1).equals(t.substring(i + 1));
                else if (s.length() < t.length())
                    return s.substring(i).equals(t.substring(i + 1));
                else
                    return t.substring(i).equals(s.substring(i + 1));
            }
        }
        return Math.abs(s.length() - t.length()) == 1;
    }
}
```

162.Find Peak Element

A peak element is an element that is greater than its neighbors.
Given an input array where `num[i] ≠ num[i+1]`, find a peak element and return its index.
The array may contain multiple peaks, in that case return the index to any one of the peaks is fine.
You may imagine that `num[-1] = num[n] = -∞`.
For example, in array `[1, 2, 3, 1]`, 3 is a peak element and your function should return the index number 2.
click to show spoilers.
Note:
Your solution should be in logarithmic complexity.

```
    public int findPeakElement(int[] nums) {
        return search(nums, 0, nums.length - 1);
    }
    public int search(int[] nums, int l, int r) {
        if (l == r)
            return l;
        int mid = (l + r) / 2;
        if (nums[mid] > nums[mid + 1])
            return search(nums, l, mid);
        return search(nums, mid + 1, r);
    }
}
public class Solution {
    public int findPeakElement(int[] nums) {
        int l = 0, r = nums.length - 1;
        while (l < r) {
            int mid = (l + r) / 2;
            if (nums[mid] > nums[mid + 1])
                r = mid;
            else
                l = mid + 1;
        }
        return l;
    }
}
```

163.Missing Ranges

Given a sorted integer array where **the range of elements are in the inclusive range [*lower*, *upper*]**, return its missing ranges.
For example, given `[0, 1, 3, 50, 75]`, *lower* = 0 and *upper* = 99, return `["2", "4->49", "51->74", "76->99"]`.

```
public class Solution {
    public List<String> findMissingRanges(int[] nums, int lower, int upper) {
        List<String> list = new ArrayList<String>();
        int preMiss = lower;
        for (int num : nums) {
            if (num == Integer.MAX_VALUE) {
                if (upper == num)
                    --upper;
                break;
            }
            preMiss = getNext(preMiss, num, list);
        }
        if (preMiss == upper)
            list.add(preMiss + "");
        else if (preMiss < upper)
            list.add(preMiss + "->" + upper);
        return list;
    }
    private static int getNext(int miss, int num, List<String> list) {
        if (miss < num) {
            if (miss == num - 1)
                list.add(miss + "");
            else
                list.add(miss + "->" + (num - 1));
        }
        return num + 1;
    }
}
```

164.Maximum Gap

Given an unsorted array, find the maximum difference between the successive elements in its sorted form.
Try to solve it in linear time/space.
Return 0 if the array contains less than 2 elements.
You may assume all elements in the array are non-negative integers and fit in the 32-bit signed integer range.

```
public class Solution {
    public int maximumGap(int[] nums) {
        if (nums == null || nums.length < 2)
            return 0;
        int min = nums[0];
        int max = nums[0];
        for (int i : nums) {
            min = Math.min(min, i);
            max = Math.max(max, i);
        }
        int gap = (int) Math.ceil((double) (max - min) / (nums.length - 1));
        int[] bucketsMIN = new int[nums.length - 1];
        int[] bucketsMAX = new int[nums.length - 1];
        Arrays.fill(bucketsMIN, Integer.MAX_VALUE);
        Arrays.fill(bucketsMAX, Integer.MIN_VALUE);
```

```
            if (i == min || i == max)
                continue;
            int idx = (i - min) / gap;
            bucketsMIN[idx] = Math.min(i, bucketsMIN[idx]);
            bucketsMAX[idx] = Math.max(i, bucketsMAX[idx]);
        }
        int maxGap = Integer.MIN_VALUE;
        int previous = min;
        for (int i = 0; i < nums.length - 1; i++) {
            if (bucketsMIN[i] == Integer.MAX_VALUE
                    && bucketsMAX[i] == Integer.MIN_VALUE)
                continue;
            maxGap = Math.max(maxGap, bucketsMIN[i] - previous);
            previous = bucketsMAX[i];
        }
        maxGap = Math.max(maxGap, max - previous);
        return maxGap;
    }
}
```

165.Compare Version Numbers

Compare two version numbers *version1* and *version2*.
If *version1* > *version2* return 1, if *version1* < *version2* return -1, otherwise return 0.
You may assume that the version strings are non-empty and contain only digits and the `.` character.
The `.` character does not represent a decimal point and is used to separate number sequences.
For instance, `2.5` is not "two and a half" or "half way to version three", it is the fifth second-level revision of the second first-level revision.
Here is an example of version numbers ordering:

```
0.1 < 1.1 < 1.2 < 13.37
```

```
public class Solution {
    public int compareVersion(String version1, String version2) {
        String[] levels1 = version1.split("\\.");
        String[] levels2 = version2.split("\\.");
        int length = Math.max(levels1.length, levels2.length);
        for (int i = 0; i < length; i++) {
            Integer v1 = i < levels1.length ? Integer.parseInt(levels1[i]) : 0;
            Integer v2 = i < levels2.length ? Integer.parseInt(levels2[i]) : 0;
            int compare = v1.compareTo(v2);
            if (compare != 0) {
                return compare;
            }
        }
        return 0;
    }
}
```

166.Fraction to Recurring Decimal

Given two integers representing the numerator and denominator of a fraction, return the fraction in string format.
If the fractional part is repeating, enclose the repeating part in parentheses.
For example,

- Given numerator = 1, denominator = 2, return "0.5".
- Given numerator = 2, denominator = 1, return "2".
- Given numerator = 2, denominator = 3, return "0.(6)".

```
public class Solution {
    public String fractionToDecimal(int numerator, int denominator) {
        if (numerator == 0) {
            return "0";
        }
        StringBuilder fraction = new StringBuilder();
        if (numerator < 0 ^ denominator < 0) {
            fraction.append("-");
        }
        long dividend = Math.abs(Long.valueOf(numerator));
        long divisor = Math.abs(Long.valueOf(denominator));
        fraction.append(String.valueOf(dividend / divisor));
        long remainder = dividend % divisor;
        if (remainder == 0) {
            return fraction.toString();
        }
        fraction.append(".");
        Map<Long, Integer> map = new HashMap<>();
        while (remainder != 0) {
            if (map.containsKey(remainder)) {
                fraction.insert(map.get(remainder), "(");
                fraction.append(")");
```

```
            }
            map.put(remainder, fraction.length());
            remainder *= 10;
            fraction.append(String.valueOf(remainder / divisor));
            remainder %= divisor;
        }
        return fraction.toString();
    }
}
```

167.Two Sum II - Input array is sorted

Given an array of integers that is already ***sorted in ascending order***, find two numbers such that they add up to a specific target number.

The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2. Please note that your returned answers (both index1 and index2) are not zero-based.

You may assume that each input would have *exactly* one solution and you may not use the *same* element twice.

```
Input: numbers={2, 7, 11, 15}, target=9
Output: index1=1, index2=2
```

```
public class Solution {
    public int[] twoSum(int[] nums, int target) {
        int[] indice = new int[2];
        if (nums == null || nums.length < 2)
            return indice;
        int left = 0, right = nums.length - 1;
        while (left < right) {
            int v = nums[left] + nums[right];
            if (v == target) {
                indice[0] = left + 1;
                indice[1] = right + 1;
                break;
            } else if (v > target) {
                right--;
            } else {
                left++;
            }
        }
        return indice;
    }
}
```

168.Excel Sheet Column Title

Given a positive integer, return its corresponding column title as appear in an Excel sheet.

For example:

```
1 -> A
2 -> B
3 -> C
...
26 -> Z
27 -> AA
28 -> AB
```

```
public class Solution {
    public String convertToTitle(int n) {
        return n == 0 ? "" : convertToTitle(--n / 26) + (char) ('A' + (n % 26));
    }
}
```

169.Majority Element

Given an array of size *n*, find the majority element. The majority element is the element that appears **more than** `⌊ n/2 ⌋` times.

You may assume that the array is non-empty and the majority element always exist in the array.

```
public class Solution {
    public int majorityElement(int[] nums) {
        int major = nums[0], count = 1;
        for (int i = 1; i < nums.length; i++) {
            if (count == 0) {
                count++;
                major = nums[i];
            } else if (major == nums[i])
                count++;
            else
                count--;
        }
        return major;
    }
}
```

170.Two Sum III - Data structure design

Design and implement a TwoSum class. It should support the following operations: `add` and `find`.

`add` - Add the number to an internal data structure.
`find` - Find if there exists any pair of numbers which sum is equal to the value.
For example,

```
add(1); add(3); add(5);
find(4) -> true
find(7) -> false
```

```
public class TwoSum {

    private List<Integer> list = new ArrayList<Integer>();
    private Map<Integer, Integer> map = new HashMap<Integer, Integer>();
    public TwoSum() {
    }
    public void add(int number) {
        if (map.containsKey(number))
            map.put(number, map.get(number) + 1);
        else {
            map.put(number, 1);
            list.add(number);
        }
    }
    public boolean find(int value) {
        for (int i = 0; i < list.size(); i++) {
            int num1 = list.get(i), num2 = value - num1;
            if ((num1 == num2 && map.get(num1) > 1)
                    || (num1 != num2 && map.containsKey(num2)))
                return true;
        }
        return false;
    }
}
```

171.Excel Sheet Column Number

Related to question Excel Sheet Column Title
Given a column title as appear in an Excel sheet, return its corresponding column number.
For example:

```
    A -> 1
    B -> 2
    C -> 3
    ...
    Z -> 26
    AA -> 27
    AB -> 28
```

```
public class Solution {
    public int titleToNumber(String s) {
        int result = 0;
        for (int i = 0; i < s.length(); i++)
            result = result * 26 + (s.charAt(i) - 'A' + 1);
        return result;
    }
}
```

172.Factorial Trailing Zeroes

Given an integer *n*, return the number of trailing zeroes in *n*!.
Note: Your solution should be in logarithmic time complexity.

```
public class Solution {
    public int trailingZeroes(int n) {
        return n == 0 ? 0 : n / 5 + trailingZeroes(n / 5);
    }
}
```

173.Binary Search Tree Iterator

Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST.
Calling `next()` will return the next smallest number in the BST.
Note: `next()` and `hasNext()` should run in average O(1) time and uses O(*h*) memory, where *h* is the height of the tree.

```
public class BSTIterator {
    private Stack<TreeNode> stack = new Stack<TreeNode>();
    public BSTIterator(TreeNode root) {
        pushAll(root);
    }
    public boolean hasNext() {
        return !stack.isEmpty();
    }
    public int next() {
        TreeNode tmpNode = stack.pop();
        pushAll(tmpNode.right);
```

```
    }
    private void pushAll(TreeNode node) {
        for (; node != null; stack.push(node), node = node.left)
            ;
    }
}
```

174.Dungeon Game

The demons had captured the princess (**P**) and imprisoned her in the bottom-right corner of a dungeon. The dungeon consists of M x N rooms laid out in a 2D grid. Our valiant knight (**K**) was initially positioned in the top-left room and must fight his way through the dungeon to rescue the princess.

The knight has an initial health point represented by a positive integer. If at any point his health point drops to 0 or below, he dies immediately.

Some of the rooms are guarded by demons, so the knight loses health (*negative* integers) upon entering these rooms; other rooms are either empty (*0's*) or contain magic orbs that increase the knight's health (*positive* integers).

In order to reach the princess as quickly as possible, the knight decides to move only rightward or downward in each step.

Write a function to determine the knight's minimum initial health so that he is able to rescue the princess.

For example, given the dungeon below, the initial health of the knight must be at least **7** if he follows the optimal path `RIGHT-> RIGHT -> DOWN -> DOWN`.

-2 (K)	-3	3
-5	-10	1
10	30	-5 (P)

Notes:

- The knight's health has no upper bound.
- Any room can contain threats or power-ups, even the first room the knight enters and the bottom-right room where the princess is imprisoned.

```
public class Solution {
    public int calculateMinimumHP(int[][] dungeon) {
        if (dungeon == null || dungeon.length == 0 || dungeon[0].length == 0)
            return 0;
        int m = dungeon.length;
        int n = dungeon[0].length;
        int[][] health = new int[m][n];
        health[m - 1][n - 1] = Math.max(1 - dungeon[m - 1][n - 1], 1);
        for (int i = m - 2; i >= 0; i--)
            health[i][n - 1] = Math
                    .max(health[i + 1][n - 1] - dungeon[i][n - 1], 1);
        for (int j = n - 2; j >= 0; j--)
            health[m - 1][j] = Math
                    .max(health[m - 1][j + 1] - dungeon[m - 1][j], 1);
        for (int i = m - 2; i >= 0; i--) {
            for (int j = n - 2; j >= 0; j--) {
                int down = Math.max(health[i + 1][j] - dungeon[i][j], 1);
                int right = Math.max(health[i][j + 1] - dungeon[i][j], 1);
                health[i][j] = Math.min(right, down);
            }
        }
        return health[0][0];
    }
}
```

175.Combine Two Tables

```
Create table If Not Exists Person (PersonId int, FirstName varchar(255), LastName varchar(255));
Create table If Not Exists Address (AddressId int, PersonId int, City varchar(255),
State varchar(255));
Truncate table Person;
insert into Person (PersonId, LastName, FirstName) values ('1', 'Wang', 'Allen');
Truncate table Address;
insert into Address (AddressId, PersonId, City, State) values ('1', '2', 'New York City', 'New
York');
```

Table: `Person`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| PersonId    | int     |
| FirstName   | varchar |
| LastName    | varchar |
+-------------+---------+
PersonId is the primary key column for this table.
```

Table: `Address`

```
| Column Name | Type    |
+-------------+---------+
| AddressId   | int     |
| PersonId    | int     |
| City        | varchar |
| State       | varchar |
+-------------+---------+
AddressId is the primary key column for this table.
```

Write a SQL query for a report that provides the following information for each person in the Person table, regardless if there is an address for each of those people:

```
FirstName, LastName, City, State
```

```
SELECT Person.FirstName,
  Person.LastName,
  Address.City,
  Address.State
FROM Person
LEFT JOIN Address
ON Person.PersonId = Address.PersonId
```

176.Second Highest Salary

```
Create table If Not Exists Employee (Id int, Salary int);
Truncate table Employee;
insert into Employee (Id, Salary) values ('1', '100');
insert into Employee (Id, Salary) values ('2', '200');
insert into Employee (Id, Salary) values ('3', '300');
```

Write a SQL query to get the second highest salary from the `Employee` table.

```
+----+--------+
| Id | Salary |
+----+--------+
| 1  | 100    |
| 2  | 200    |
| 3  | 300    |
+----+--------+
```

For example, given the above Employee table, the query should return `200` as the second highest salary. If there is no second highest salary, then the query should return `null`.

```
+---------------------+
| SecondHighestSalary |
+---------------------+
| 200                 |
+---------------------+
```

```
SELECT MAX(Salary) SecondHighestSalary
FROM Employee
WHERE Salary <
  (SELECT MAX(Salary) FROM Employee)
```

177.Nth Highest Salary

Write a SQL query to get the n^{th} highest salary from the `Employee` table.

```
+----+--------+
| Id | Salary |
+----+--------+
| 1  | 100    |
| 2  | 200    |
| 3  | 300    |
+----+--------+
```

For example, given the above Employee table, the n^{th} highest salary where $n = 2$ is `200`. If there is no n^{th} highest salary, then the query should return `null`.

```
+------------------------+
| getNthHighestSalary(2) |
+------------------------+
| 200                    |
+------------------------+
```

```
CREATE FUNCTION getNthHighestSalary(N INT) RETURNS INT
BEGIN
DECLARE M INT;
SET M=N-1;
  RETURN (
      SELECT DISTINCT Salary FROM Employee ORDER BY Salary DESC LIMIT M, 1
```

```
  );
END
```

178.Rank Scores

```
Create table If Not Exists Scores (Id int, Score DECIMAL(3,2));
Truncate table Scores;
insert into Scores (Id, Score) values ('1', '3.5');
insert into Scores (Id, Score) values ('2', '3.65');
insert into Scores (Id, Score) values ('3', '4.0');
insert into Scores (Id, Score) values ('4', '3.85');
insert into Scores (Id, Score) values ('5', '4.0');
insert into Scores (Id, Score) values ('6', '3.65');
```

Write a SQL query to rank scores. If there is a tie between two scores, both should have the same ranking. Note that after a tie, the next ranking number should be the next consecutive integer value. In other words, there should be no "holes" between ranks.

```
+----+-------+
| Id | Score |
+----+-------+
| 1  | 3.50  |
| 2  | 3.65  |
| 3  | 4.00  |
| 4  | 3.85  |
| 5  | 4.00  |
| 6  | 3.65  |
+----+-------+
```

For example, given the above `Scores` table, your query should generate the following report (order by highest score):

```
+-------+------+
| Score | Rank |
+-------+------+
| 4.00  | 1    |
| 4.00  | 1    |
| 3.85  | 2    |
| 3.65  | 3    |
| 3.65  | 3    |
| 3.50  | 4    |
+-------+------+
```

```
SELECT
  Score,
  @rank := @rank + (@prev <> (@prev := Score)) Rank
FROM
  Scores,
  (SELECT @rank := 0, @prev := -1) init
ORDER BY Score desc
```

179.Largest Number

Given a list of non negative integers, arrange them such that they form the largest number.

For example, given `[3, 30, 34, 5, 9]`, the largest formed number is `9534330`.

Note: The result may be very large, so you need to return a string instead of an integer.

```
public class Solution {
    public String largestNumber(int[] num) {
        String[] array = Arrays.stream(num).mapToObj(String::valueOf)
                .toArray(String[]::new);
        Arrays.sort(array,
                (String s1, String s2) -> (s2 + s1).compareTo(s1 + s2));
        return Arrays.stream(array).reduce((x, y) -> x.equals("0") ? y : x + y)
                .get();
    }
}
```

180.Consecutive Numbers

```
Create table If Not Exists Logs (Id int, Num int);
Truncate table Logs;
insert into Logs (Id, Num) values ('1', '1');
insert into Logs (Id, Num) values ('2', '1');
insert into Logs (Id, Num) values ('3', '1');
insert into Logs (Id, Num) values ('4', '2');
insert into Logs (Id, Num) values ('5', '1');
insert into Logs (Id, Num) values ('6', '2');
insert into Logs (Id, Num) values ('7', '2');
```

Write a SQL query to find all numbers that appear at least three times consecutively.

```
+----+-----+
| Id | Num |
+----+-----+
| 1  |  1  |
| 2  |  1  |
```

```
| 3  |  1  |
| 4  |  2  |
| 5  |  1  |
| 6  |  2  |
| 7  |  2  |
+----+-----+
```

For example, given the above `Logs` table, `1` is the only number that appears consecutively for at least three times.

```
+-----------------+
| ConsecutiveNums |
+-----------------+
| 1               |
+-----------------+
```

```
SELECT DISTINCT l1.Num ConsecutiveNums
FROM Logs l1,
  Logs l2,
  Logs l3
WHERE l1.Id=l2.Id-1
AND l2.Id  =l3.Id-1
AND l1.Num =l2.Num
AND l2.Num =l3.Num
```

181.Employees Earning More Than Their Managers

```
Create table If Not Exists Employee (Id int, Name varchar(255), Salary int, ManagerId int);
Truncate table Employee;
insert into Employee (Id, Name, Salary, ManagerId) values ('1', 'Joe', '70000', '3');
insert into Employee (Id, Name, Salary, ManagerId) values ('2', 'Henry', '80000', '4');
insert into Employee (Id, Name, Salary, ManagerId) values ('3', 'Sam', '60000', 'None');
insert into Employee (Id, Name, Salary, ManagerId) values ('4', 'Max', '90000', 'None');
```

The `Employee` table holds all employees including their managers. Every employee has an Id, and there is also a column for the manager Id.

```
+----+-------+--------+-----------+
| Id | Name  | Salary | ManagerId |
+----+-------+--------+-----------+
| 1  | Joe   | 70000  | 3         |
| 2  | Henry | 80000  | 4         |
| 3  | Sam   | 60000  | NULL      |
| 4  | Max   | 90000  | NULL      |
+----+-------+--------+-----------+
```

Given the `Employee` table, write a SQL query that finds out employees who earn more than their managers. For the above table, Joe is the only employee who earns more than his manager.

```
+----------+
| Employee |
+----------+
| Joe      |
+----------+
```

```
SELECT E1.Name Employee
FROM Employee AS E1,
  Employee    AS E2
WHERE E1.ManagerId = E2.Id
AND E1.Salary      > E2.Salary
```

182.Duplicate Emails

```
Create table If Not Exists Person (Id int, Email varchar(255));
Truncate table Person;
insert into Person (Id, Email) values ('1', 'a@b.com');
insert into Person (Id, Email) values ('2', 'c@d.com');
insert into Person (Id, Email) values ('3', 'a@b.com');
```

Write a SQL query to find all duplicate emails in a table named `Person`.

```
+----+---------+
| Id | Email   |
+----+---------+
| 1  | a@b.com |
| 2  | c@d.com |
| 3  | a@b.com |
+----+---------+
```

For example, your query should return the following for the above table:

```
+---------+
| Email   |
```

```
+---------+
| a@b.com |
+---------+
```

Note: All emails are in lowercase.

```
select Email
from Person
group by Email
having count(*) > 1
```

183.Customers Who Never Order

```
Create table If Not Exists Customers (Id int, Name varchar(255));
Create table If Not Exists Orders (Id int, CustomerId int);
Truncate table Customers;
insert into Customers (Id, Name) values ('1', 'Joe');
insert into Customers (Id, Name) values ('2', 'Henry');
insert into Customers (Id, Name) values ('3', 'Sam');
insert into Customers (Id, Name) values ('4', 'Max');
Truncate table Orders;
insert into Orders (Id, CustomerId) values ('1', '3');
insert into Orders (Id, CustomerId) values ('2', '1');
```

Suppose that a website contains two tables, the `Customers` table and the `Orders` table. Write a SQL query to find all customers who never order anything.

Table: `Customers`.

```
+----+-------+
| Id | Name  |
+----+-------+
| 1  | Joe   |
| 2  | Henry |
| 3  | Sam   |
| 4  | Max   |
+----+-------+
```

Table: `Orders`.

```
+----+------------+
| Id | CustomerId |
+----+------------+
| 1  | 3          |
| 2  | 1          |
+----+------------+
```

Using the above tables as example, return the following:

```
+-----------+
| Customers |
+-----------+
| Henry     |
| Max       |
+-----------+
```

```
select customers.name as 'Customers'
from customers
where customers.id not in
(
    select customerid from orders
)
```

184.Department Highest Salary

```
Create table If Not Exists Employee (Id int, Name varchar(255), Salary int, DepartmentId int);
Create table If Not Exists Department (Id int, Name varchar(255));
Truncate table Employee;
insert into Employee (Id, Name, Salary, DepartmentId) values ('1', 'Joe', '70000', '1');
insert into Employee (Id, Name, Salary, DepartmentId) values ('2', 'Henry', '80000', '2');
insert into Employee (Id, Name, Salary, DepartmentId) values ('3', 'Sam', '60000', '2');
insert into Employee (Id, Name, Salary, DepartmentId) values ('4', 'Max', '90000', '1');
Truncate table Department;
insert into Department (Id, Name) values ('1', 'IT');
insert into Department (Id, Name) values ('2', 'Sales');
```

The `Employee` table holds all employees. Every employee has an Id, a salary, and there is also a column for the department Id.

```
+----+-------+--------+--------------+
| Id | Name  | Salary | DepartmentId |
+----+-------+--------+--------------+
| 1  | Joe   | 70000  | 1            |
| 2  | Henry | 80000  | 2            |
```

```
| 4  | Max   | 90000  | 1            |
+----+-------+--------+--------------+
```

The `Department` table holds all departments of the company.

```
+----+----------+
| Id | Name     |
+----+----------+
| 1  | IT       |
| 2  | Sales    |
+----+----------+
```

Write a SQL query to find employees who have the highest salary in each of the departments. For the above tables, Max has the highest salary in the IT department and Henry has the highest salary in the Sales department.

```
+------------+----------+--------+
| Department | Employee | Salary |
+------------+----------+--------+
| IT         | Max      | 90000  |
| Sales      | Henry    | 80000  |
+------------+----------+--------+
```

```sql
SELECT
    Department.name AS 'Department',
    Employee.name AS 'Employee',
    Salary
FROM
    Employee
        JOIN
    Department ON Employee.DepartmentId = Department.Id
WHERE
    (Employee.DepartmentId , Salary) IN
    (   SELECT
            DepartmentId, MAX(Salary)
        FROM
            Employee
        GROUP BY DepartmentId
    )
```

185.Department Top Three Salaries

```sql
Create table If Not Exists Employee (Id int, Name varchar(255), Salary int, DepartmentId int);
Create table If Not Exists Department (Id int, Name varchar(255));
Truncate table Employee;
insert into Employee (Id, Name, Salary, DepartmentId) values ('1', 'Joe', '70000', '1');
insert into Employee (Id, Name, Salary, DepartmentId) values ('2', 'Henry', '80000', '2');
insert into Employee (Id, Name, Salary, DepartmentId) values ('3', 'Sam', '60000', '2');
insert into Employee (Id, Name, Salary, DepartmentId) values ('4', 'Max', '90000', '1');
Truncate table Department;
insert into Department (Id, Name) values ('1', 'IT');
insert into Department (Id, Name) values ('2', 'Sales');
```

The `Employee` table holds all employees. Every employee has an Id, and there is also a column for the department Id.

```
+----+-------+--------+--------------+
| Id | Name  | Salary | DepartmentId |
+----+-------+--------+--------------+
| 1  | Joe   | 70000  | 1            |
| 2  | Henry | 80000  | 2            |
| 3  | Sam   | 60000  | 2            |
| 4  | Max   | 90000  | 1            |
| 5  | Janet | 69000  | 1            |
| 6  | Randy | 85000  | 1            |
+----+-------+--------+--------------+
```

The `Department` table holds all departments of the company.

```
+----+----------+
| Id | Name     |
+----+----------+
| 1  | IT       |
| 2  | Sales    |
+----+----------+
```

Write a SQL query to find employees who earn the top three salaries in each of the department. For the above tables, your SQL query should return the following rows.

```
+------------+----------+--------+
| Department | Employee | Salary |
```

```
| IT         | Max      | 90000  |
| IT         | Randy    | 85000  |
| IT         | Joe      | 70000  |
| Sales      | Henry    | 80000  |
| Sales      | Sam      | 60000  |
+------------+----------+--------+
```

```sql
SELECT
    d.Name AS 'Department', e1.Name AS 'Employee', e1.Salary
FROM
    Employee e1
        JOIN
    Department d ON e1.DepartmentId = d.Id
WHERE
    3 > (SELECT
            COUNT(DISTINCT e2.Salary)
        FROM
            Employee e2
        WHERE
            e2.Salary > e1.Salary
                AND e1.DepartmentId = e2.DepartmentId
        )
```

186.Reverse Words in a String II

Given an input string, reverse the string word by word. A word is defined as a sequence of non-space characters.
The input string does not contain leading or trailing spaces and the words are always separated by a single space.
For example,
Given s = "`the sky is blue`",
return "`blue is sky the`".
Could you do it *in-place* without allocating extra space?

```java
public class Solution {
    public void reverseWords(char[] s) {
        reverse(s, 0, s.length - 1);
        int start = 0;
        for (int i = 0; i < s.length; i++) {
            if (s[i] == ' ') {
                reverse(s, start, i - 1);
                start = i + 1;
            }
        }
        reverse(s, start, s.length - 1);
    }
    public void reverse(char[] s, int start, int end) {
        while (start < end) {
            char temp = s[start];
            s[start] = s[end];
            s[end] = temp;
            start++;
            end--;
        }
    }
}
```

187.Repeated DNA Sequences

All DNA is composed of a series of nucleotides abbreviated as A, C, G, and T, for example: "ACGAATTCCG". When studying DNA, it is sometimes useful to identify repeated sequences within the DNA.
Write a function to find all the 10-letter-long sequences (substrings) that occur more than once in a DNA molecule.
For example,

```
Given s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT",
Return:
["AAAAACCCCC", "CCCCCAAAAA"].
```

```java
public class Solution {
    public List<String> findRepeatedDnaSequences(String s) {
        Set seen = new HashSet(), repeated = new HashSet();
        for (int i = 0; i + 9 < s.length(); i++) {
            String ten = s.substring(i, i + 10);
            if (!seen.add(ten))
                repeated.add(ten);
        }
        return new ArrayList(repeated);
    }
}
```

188.Best Time to Buy and Sell Stock IV

Say you have an array for which the i^{th} element is the price of a given stock on day *i*.
Design an algorithm to find the maximum profit. You may complete at most **k** transactions.

Note:
You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

```
public class Solution {
    public int maxProfit(int k, int[] prices) {
        int n = prices.length;
        if (n <= 1)
            return 0;
        if (k >= n / 2) {
            int maxPro = 0;
            for (int i = 1; i < n; i++) {
                if (prices[i] > prices[i - 1])
                    maxPro += prices[i] - prices[i - 1];
            }
            return maxPro;
        }
        int[][] dp = new int[k + 1][n];
        for (int i = 1; i <= k; i++) {
            int localMax = dp[i - 1][0] - prices[0];
            for (int j = 1; j < n; j++) {
                dp[i][j] = Math.max(dp[i][j - 1], prices[j] + localMax);
                localMax = Math.max(localMax, dp[i - 1][j] - prices[j]);
            }
        }
        return dp[k][n - 1];
    }
}
```

189.Rotate Array

Rotate an array of *n* elements to the right by *k* steps.
For example, with *n* = 7 and *k* = 3, the array `[1,2,3,4,5,6,7]` is rotated to `[5,6,7,1,2,3,4]`.
Note:
Try to come up as many solutions as you can, there are at least 3 different ways to solve this problem.
[show hint]
Hint:
Could you do it in-place with O(1) extra space?
Related problem: Reverse Words in a String II

```
public class Solution {
    public void rotate(int[] nums, int k) {
        k %= nums.length;
        reverse(nums, 0, nums.length - 1);
        reverse(nums, 0, k - 1);
        reverse(nums, k, nums.length - 1);
    }
    public void reverse(int[] nums, int start, int end) {
        while (start < end) {
            int temp = nums[start];
            nums[start] = nums[end];
            nums[end] = temp;
            start++;
            end--;
        }
    }
}
```

190.Reverse Bits

Reverse bits of a given 32 bits unsigned integer.
For example, given input 43261596 (represented in binary as **00000010100101000001111010011100**), return 964176192 (represented in binary as **00111001011110000010100101000000**).
Follow up:
If this function is called many times, how would you optimize it?
Related problem: Reverse Integer

```
public class Solution {
    public int reverseBits(int n) {
        int result = 0;
        for (int i = 0; i < 32; i++) {
            result += n & 1;
            n >>>= 1;
            if (i < 31)
                result <<= 1;
        }
        return result;
    }
}
```

191.Number of 1 Bits

Write a function that takes an unsigned integer and returns the number of '1' bits it has (also known as the Hamming weight).
For example, the 32-bit integer '11' has binary representation `00000000000000000000000000001011`, so the function should return 3.

```
public class Solution {
```

```
    public int hammingWeight(int n) {
        int bits = 0;
        int mask = 1;
        for (int i = 0; i < 32; i++) {
            if ((n & mask) != 0) {
                bits++;
            }
            mask <<= 1;
        }
        return bits;
    }
}
```

192.Word Frequency

Write a bash script to calculate the frequency of each word in a text file `words.txt`.
For simplicity sake, you may assume:

- `words.txt` contains only lowercase characters and space `' '` characters.
- Each word must consist of lowercase characters only.
- Words are separated by one or more whitespace characters.

For example, assume that `words.txt` has the following content:

```
the day is sunny the the
the sunny is is
```

Your script should output the following, sorted by descending frequency:

```
the 4
is 3
sunny 2
day 1
```

Note:
Don't worry about handling ties, it is guaranteed that each word's frequency count is unique.
[show hint]
Hint:
Could you write it in one-line using Unix pipes?
Solution

```
cat words.txt | tr -s ' ' '\n' | sort | uniq -c | sort -r | awk '{ print $2, $1 }'
```

tr -s: truncate the string with target string, but only remaining one instance (e.g. multiple whitespaces)
sort: To make the same string successive so that `uniq` could count the same string fully and correctly.
uniq -c: uniq is used to filter out the repeated lines which are successive, -c means counting
sort -r: -r means sorting in descending order
awk '{ print $2, $1 }': To format the output, see here.

193.Valid Phone Numbers

Given a text file `file.txt` that contains list of phone numbers (one per line), write a one liner bash script to print all valid phone numbers.
You may assume that a valid phone number must appear in one of the following two formats: (xxx) xxx-xxxx or xxx-xxx-xxxx. (x means a digit)
You may also assume each line in the text file must not contain leading or trailing white spaces.
For example, assume that `file.txt` has the following content:

```
987-123-4567
123 456 7890
(123) 456-7890
```

Your script should output the following valid phone numbers:

```
987-123-4567
(123) 456-7890
```

Solution
Using `grep`:

```
grep -P '^(\d{3}-|\(\d{3}\) )\d{3}-\d{4}$' file.txt
```

Using `sed`:

```
sed -n -r '/^([0-9]{3}-|\([0-9]{3}\) )[0-9]{3}-[0-9]{4}$/p' file.txt
```

Using `awk`:

```
awk '/^([0-9]{3}-|\([0-9]{3}\) )[0-9]{3}-[0-9]{4}$/' file.txt
```

194.Transpose File

Given a text file `file.txt`, transpose its content.
You may assume that each row has the same number of columns and each field is separated by the `' '` character.
For example, if `file.txt` has the following content:

```
name age
alice 21
ryan 30
```

Output the following:

```
name alice ryan
age 21 30
```

Solution

```
awk '
{
    for (i = 1; i <= NF; i++) {
        if(NR == 1) {
            s[i] = $i;
        } else {
            s[i] = s[i] " " $i;
        }
    }
}
END {
    for (i = 1; s[i] != ""; i++) {
        print s[i];
    }
}' file.txt
```

Bash:

```
ncol=`head -n1 file.txt | wc -w`
for i in `seq 1 $ncol`
do
    echo `cut -d' ' -f$i file.txt`
done
```

195.Tenth Line

How would you print just the 10th line of a file?
For example, assume that `file.txt` has the following content:

```
Line 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
Line 8
Line 9
Line 10
```

Your script should output the tenth line, which is:

```
Line 10
```

[show hint]

Hint:

1. If the file contains less than 10 lines, what should you output?
2. There's at least three different solutions. Try to explore all possibilities.

Solution

```
# Solution 1
cnt=0
while read line && [ $cnt -le 10 ]; do
  let 'cnt = cnt + 1'
  if [ $cnt -eq 10 ]; then
    echo $line
    exit 0
  fi
done < file.txt
# Solution 2
awk 'FNR == 10 {print }'  file.txt
# OR
awk 'NR == 10' file.txt
# Solution 3
sed -n 10p file.txt
# Solution 4
tail -n+10 file.txt|head -1
```

196.Delete Duplicate Emails

Write a SQL query to delete all duplicate email entries in a table named `Person`, keeping only unique emails based on its *smallest* **Id**.

```
+----+------------------+
| Id | Email            |
+----+------------------+
| 1  | john@example.com |
| 2  | bob@example.com  |
| 3  | john@example.com |
```

```
Id is the primary key column for this table.
```

For example, after running your query, the above `Person` table should have the following rows:

```
+----+------------------+
| Id | Email            |
+----+------------------+
| 1  | john@example.com |
| 2  | bob@example.com  |
+----+------------------+
```

```
DELETE p1 FROM Person p1,
    Person p2
WHERE
    p1.Email = p2.Email AND p1.Id > p2.Id
```

197.Rising Temperature

```
Create table If Not Exists Weather (Id int, Date date, Temperature int);
Truncate table Weather;
insert into Weather (Id, Date, Temperature) values ('1', '2015-01-01', '10');
insert into Weather (Id, Date, Temperature) values ('2', '2015-01-02', '25');
insert into Weather (Id, Date, Temperature) values ('3', '2015-01-03', '20');
insert into Weather (Id, Date, Temperature) values ('4', '2015-01-04', '30');
```

Given a `Weather` table, write a SQL query to find all dates' Ids with higher temperature compared to its previous (yesterday's) dates.

```
+---------+------------+------------------+
| Id(INT) | Date(DATE) | Temperature(INT) |
+---------+------------+------------------+
|       1 | 2015-01-01 |               10 |
|       2 | 2015-01-02 |               25 |
|       3 | 2015-01-03 |               20 |
|       4 | 2015-01-04 |               30 |
+---------+------------+------------------+
```

For example, return the following Ids for the above Weather table:

```
+----+
| Id |
+----+
|  2 |
|  4 |
+----+
```

```
SELECT
    weather.id AS 'Id'
FROM
    weather
        JOIN
    weather w ON DATEDIFF(weather.date, w.date) = 1
        AND weather.Temperature > w.Temperature
```

198.House Robber

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight **without alerting the police**.

```
public class Solution {
    public int rob(int[] nums) {
        int prevMax = 0;
        int currMax = 0;
        for (int x : nums) {
            int temp = currMax;
            currMax = Math.max(prevMax + x, currMax);
            prevMax = temp;
        }
        return currMax;
    }
}
```

199.Binary Tree Right Side View

Given a binary tree, imagine yourself standing on the *right* side of it, return the values of the nodes you can see ordered from top to bottom.

For example:

Given the following binary tree,

```
   1            <---
 /   \
2     3         <---
```

```
 \     \
  5     4       <---
```

You should return `[1, 3, 4]`.

```
public class Solution {
    public List<Integer> rightSideView(TreeNode root) {
        List<Integer> result = new ArrayList<Integer>();
        if (root == null) {
            return result;
        }
        LinkedList<TreeNode> queue = new LinkedList<TreeNode>();
        queue.offer(root);
        while (!queue.isEmpty()) {
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                TreeNode curr = queue.poll();
                if (i == size - 1) {
                    result.add(curr.val);
                }
                if (curr.left != null) {
                    queue.offer(curr.left);
                }
                if (curr.right != null) {
                    queue.offer(curr.right);
                }
            }
        }
        return result;
    }
}
```

200.Number of Islands

Given a 2d grid map of `'1'`s (land) and `'0'`s (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

```
11110
11010
11000
00000
```

Answer: 1

Example 2:

```
11000
11000
00100
00011
```

Answer: 3

```
public class Solution {
    private int n;
    private int m;
    public int numIslands(char[][] grid) {
        int count = 0;
        n = grid.length;
        if (n == 0)
            return 0;
        m = grid[0].length;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++)
                if (grid[i][j] == '1') {
                    DFSMarking(grid, i, j);
                    ++count;
                }
        }
        return count;
    }
    private void DFSMarking(char[][] grid, int i, int j) {
        if (i < 0 || j < 0 || i >= n || j >= m || grid[i][j] != '1')
            return;
        grid[i][j] = '0';
        DFSMarking(grid, i + 1, j);
        DFSMarking(grid, i - 1, j);
        DFSMarking(grid, i, j + 1);
        DFSMarking(grid, i, j - 1);
    }
```

201.Bitwise AND of Numbers Range

Given a range [m, n] where 0 <= m <= n <= 2147483647, return the bitwise AND of all numbers in this range, inclusive.
For example, given the range [5, 7], you should return 4.

```
public class Solution {
    public int rangeBitwiseAnd(int m, int n) {
        while (m < n)
            n = n & (n - 1);
        return n;
    }
}
```

202.Happy Number

Write an algorithm to determine if a number is "happy".
A happy number is a number defined by the following process: Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1. Those numbers for which this process ends in 1 are happy numbers.
Example: 19 is a happy number

- $1^2 + 9^2 = 82$
- $8^2 + 2^2 = 68$
- $6^2 + 8^2 = 100$
- $1^2 + 0^2 + 0^2 = 1$

```
public class Solution {
    public boolean isHappy(int n) {
        Set<Integer> inLoop = new HashSet<Integer>();
        int squareSum, remain;
        while (inLoop.add(n)) {
            squareSum = 0;
            while (n > 0) {
                remain = n % 10;
                squareSum += remain * remain;
                n /= 10;
            }
            if (squareSum == 1)
                return true;
            else
                n = squareSum;
        }
        return false;
    }
}
```

203.Remove Linked List Elements

Remove all elements from a linked list of integers that have value ***val***.
Example
Given: 1 --> 2 --> 6 --> 3 --> 4 --> 5 --> 6, ***val*** = 6
Return: 1 --> 2 --> 3 --> 4 --> 5

```
public class Solution {
    public ListNode removeElements(ListNode head, int val) {
        if (head == null)
            return null;
        head.next = removeElements(head.next, val);
        return head.val == val ? head.next : head;
    }
}
```

```
public class Solution {
    public ListNode removeElements(ListNode head, int val) {
        ListNode fakeHead = new ListNode(-1);
        fakeHead.next = head;
        ListNode curr = head, prev = fakeHead;
        while (curr != null) {
            if (curr.val == val) {
                prev.next = curr.next;
            } else {
                prev = prev.next;
            }
            curr = curr.next;
        }
        return fakeHead.next;
    }
}
```

204.Count Primes

Description:
Count the number of prime numbers less than a non-negative number, ***n***.

```
public class Solution {
    public int countPrimes(int n) {
        boolean[] notPrime = new boolean[n];
```

```
        int count = 0;
        for (int i = 2; i < n; i++) {
            if (notPrime[i] == false) {
                count++;
                for (int j = 2; i * j < n; j++) {
                    notPrime[i * j] = true;
                }
            }
        }
        return count;
    }
}
```

205.Isomorphic Strings

Given two strings ***s*** and ***t***, determine if they are isomorphic.
Two strings are isomorphic if the characters in ***s*** can be replaced to get ***t***.
All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character but a character may map to itself.
For example,
Given `"egg"`, `"add"`, return true.
Given `"foo"`, `"bar"`, return false.
Given `"paper"`, `"title"`, return true.
Note:
You may assume both ***s*** and ***t*** have the same length.

```
public class Solution {
    public boolean isIsomorphic(String s, String t) {
        int[] m = new int[512];
        for (int i = 0; i < s.length(); i++) {
            if (m[s.charAt(i)] != m[t.charAt(i) + 256])
                return false;
            m[s.charAt(i)] = m[t.charAt(i) + 256] = i + 1;
        }
        return true;
    }
}
```

206.Reverse Linked List

Reverse a singly linked list.
click to show more hints.
Hint:
A linked list can be reversed either iteratively or recursively. Could you implement both?

```
public class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode curr = head;
        while (curr != null) {
            ListNode nextTemp = curr.next;
            curr.next = prev;
            prev = curr;
            curr = nextTemp;
        }
        return prev;
    }
}
```

207.Course Schedule

There are a total of *n* courses you have to take, labeled from `0` to `n - 1`.
Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`
Given the total number of courses and a list of prerequisite **pairs**, is it possible for you to finish all courses?
For example:

```
2, [[1,0]]
```

There are a total of 2 courses to take. To take course 1 you should have finished course 0. So it is possible.

```
2, [[1,0],[0,1]]
```

There are a total of 2 courses to take. To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.
Note:

1. The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about how a graph is represented.
2. You may assume that there are no duplicate edges in the input prerequisites.

click to show more hints.
Hints:

1. This problem is equivalent to finding if a cycle exists in a directed graph. If a cycle exists, no topological ordering exists and therefore it will be impossible to take all courses.

2. Topological Sort via DFS - A great video tutorial (21 minutes) on Coursera explaining the basic concepts of Topological Sort.
3. Topological sort could also be done via BFS.

```
public class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        int[][] matrix = new int[numCourses][numCourses]; // i -> j
        int[] indegree = new int[numCourses];
        for (int i = 0; i < prerequisites.length; i++) {
            int ready = prerequisites[i][0];
            int pre = prerequisites[i][1];
            if (matrix[pre][ready] == 0)
                indegree[ready]++;
            matrix[pre][ready] = 1;
        }
        int count = 0;
        Queue<Integer> queue = new LinkedList();
        for (int i = 0; i < indegree.length; i++) {
            if (indegree[i] == 0)
                queue.offer(i);
        }
        while (!queue.isEmpty()) {
            int course = queue.poll();
            count++;
            for (int i = 0; i < numCourses; i++) {
                if (matrix[course][i] != 0) {
                    if (--indegree[i] == 0)
                        queue.offer(i);
                }
            }
        }
        return count == numCourses;
    }
}
```

208.Implement Trie (Prefix Tree)

Implement a trie with `insert`, `search`, and `startsWith` methods.

Note:

You may assume that all inputs are consist of lowercase letters `a-z`.

```
class TrieNode {
    public boolean isWord;
    public TrieNode[] children = new TrieNode[26];
}
public class Trie {
    private TrieNode root;
    public Trie() {
        root = new TrieNode();
    }
    public void insert(String word) {
        TrieNode ws = root;
        for (int i = 0; i < word.length(); i++) {
            char ch = word.charAt(i);
            if (ws.children[ch - 'a'] == null) {
                ws.children[ch - 'a'] = new TrieNode();
            }
            ws = ws.children[ch - 'a'];
        }
        ws.isWord = true;
    }
    public boolean search(String word) {
        TrieNode ws = searchHelper(word);
        return ws != null && ws.isWord;
    }
    public boolean startsWith(String prefix) {
        return searchHelper(prefix) != null;
    }
    public TrieNode searchHelper(String key) {
        TrieNode ws = root;
        for (int i = 0; i < key.length() && ws != null; i++) {
            char ch = key.charAt(i);
            ws = ws.children[ch - 'a'];
        }
        return ws;
    }
}
```

209.Minimum Size Subarray Sum

Given an array of **n** positive integers and a positive integer **s**, find the minimal length of a **contiguous** subarray of which the sum >= **s**. If there isn't one, return 0 instead.

For example, given the array `[2,3,1,2,4,3]` and `s = 7`,
the subarray `[4,3]` has the minimal length under the problem constraint.
click to show more practice.
More practice:
If you have figured out the *O*(*n*) solution, try coding another solution of which the time complexity is *O*(*n* log *n*).

```
public class Solution {
    public int minSubArrayLen(int s, int[] nums) {
        if (nums == null || nums.length == 0)
            return 0;
        int i = 0, j = 0, sum = 0, min = Integer.MAX_VALUE;
        while (j < nums.length) {
            sum += nums[j++];
            while (sum >= s) {
                min = Math.min(min, j - i);
                sum -= nums[i++];
            }
        }
        return min == Integer.MAX_VALUE ? 0 : min;
    }
}
```

210.Course Schedule II

There are a total of *n* courses you have to take, labeled from `0` to `n - 1`.
Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`
Given the total number of courses and a list of prerequisite **pairs**, return the ordering of courses you should take to finish all courses.
There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.
For example:

```
2, [[1,0]]
```

There are a total of 2 courses to take. To take course 1 you should have finished course 0. So the correct course order is `[0,1]`

```
4, [[1,0],[2,0],[3,1],[3,2]]
```

There are a total of 4 courses to take. To take course 3 you should have finished both courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0. So one correct course order is `[0,1,2,3]`. Another correct ordering is`[0,2,1,3]`.
Note:

1. The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about how a graph is represented.
2. You may assume that there are no duplicate edges in the input prerequisites.

click to show more hints.
Hints:

1. This problem is equivalent to finding the topological order in a directed graph. If a cycle exists, no topological ordering exists and therefore it will be impossible to take all courses.
2. Topological Sort via DFS - A great video tutorial (21 minutes) on Coursera explaining the basic concepts of Topological Sort.
3. Topological sort could also be done via BFS.

```
public class Solution {
    public int[] findOrder(int numCourses, int[][] prerequisites) {
        List<List<Integer>> adj = new ArrayList<List<Integer>>(numCourses);
        for (int i = 0; i < numCourses; i++)
            adj.add(i, new ArrayList<>());
        for (int i = 0; i < prerequisites.length; i++)
            adj.get(prerequisites[i][1]).add(prerequisites[i][0]);
        int[] visited = new int[numCourses];
        Stack<Integer> stack = new Stack<>();
        for (int i = 0; i < numCourses; i++) {
            if (!topologicalSort(adj, i, stack, visited))
                return new int[0];
        }
        int i = 0;
        int[] result = new int[numCourses];
        while (!stack.isEmpty()) {
            result[i++] = stack.pop();
        }
        return result;
    }
    private boolean topologicalSort(List<List<Integer>> adj, int v,
            Stack<Integer> stack, int[] visited) {
        if (visited[v] == 2)
            return true;
        if (visited[v] == 1)
            return false;
```

```
        for (Integer u : adj.get(v)) {
            if (!topologicalSort(adj, u, stack, visited))
                return false;
        }
        visited[v] = 2;
        stack.push(v);
        return true;
    }
}
```

211.Add and Search Word - Data structure design

Design a data structure that supports the following two operations:

```
void addWord(word)
bool search(word)
```

search(word) can search a literal word or a regular expression string containing only letters `a-z` or `.`. A `.` means it can represent any one letter.
For example:

```
addWord("bad")
addWord("dad")
addWord("mad")
search("pad") -> false
search("bad") -> true
search(".ad") -> true
search("b..") -> true
```

Note:
You may assume that all words are consist of lowercase letters `a-z`.
click to show hint.
You should be familiar with how a Trie works. If not, please work on this problem: Implement Trie (Prefix Tree) first.

```
public class WordDictionary {
    public class TrieNode {
        public TrieNode[] children = new TrieNode[26];
        public String item = "";
    }
    private TrieNode root = new TrieNode();
    public void addWord(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            if (node.children[c - 'a'] == null) {
                node.children[c - 'a'] = new TrieNode();
            }
            node = node.children[c - 'a'];
        }
        node.item = word;
    }
    public boolean search(String word) {
        return match(word.toCharArray(), 0, root);
    }
    private boolean match(char[] chs, int k, TrieNode node) {
        if (k == chs.length)
            return !node.item.equals("");
        if (chs[k] != '.') {
            return node.children[chs[k] - 'a'] != null
                    && match(chs, k + 1, node.children[chs[k] - 'a']);
        } else {
            for (int i = 0; i < node.children.length; i++) {
                if (node.children[i] != null) {
                    if (match(chs, k + 1, node.children[i])) {
                        return true;
                    }
                }
            }
        }
        return false;
    }
}
```

212.Word Search II

Given a 2D board and a list of words from the dictionary, find all words in the board.
Each word must be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.
For example,
Given **words** = `["oath","pea","eat","rain"]` and **board** =

```
[ ['o','a','a','n'],
```

```
  ['i','h','k','r'],
  ['i','f','l','v']]
```

Return `["eat","oath"]`.

Note:

You may assume that all inputs are consist of lowercase letters `a-z`.

click to show hint.

You would need to optimize your backtracking to pass the larger test. Could you stop backtracking earlier?

If the current candidate does not exist in all words' prefix, you could stop backtracking immediately. What kind of data structure could answer such query efficiently? Does a hash table work? Why or why not? How about a Trie? If you would like to learn how to implement a basic trie, please work on this problem: Implement Trie (Prefix Tree) first.

```
public class Solution {
    public List<String> findWords(char[][] board, String[] words) {
        List<String> res = new ArrayList<>();
        TrieNode root = buildTrie(words);
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[0].length; j++) {
                dfs(board, i, j, root, res);
            }
        }
        return res;
    }
    public void dfs(char[][] board, int i, int j, TrieNode p,
            List<String> res) {
        char c = board[i][j];
        if (c == '#' || p.next[c - 'a'] == null)
            return;
        p = p.next[c - 'a'];
        if (p.word != null) { // found one
            res.add(p.word);
            p.word = null; // de-duplicate
        }
        board[i][j] = '#';
        if (i > 0)
            dfs(board, i - 1, j, p, res);
        if (j > 0)
            dfs(board, i, j - 1, p, res);
        if (i < board.length - 1)
            dfs(board, i + 1, j, p, res);
        if (j < board[0].length - 1)
            dfs(board, i, j + 1, p, res);
        board[i][j] = c;
    }
    public TrieNode buildTrie(String[] words) {
        TrieNode root = new TrieNode();
        for (String w : words) {
            TrieNode p = root;
            for (char c : w.toCharArray()) {
                int i = c - 'a';
                if (p.next[i] == null)
                    p.next[i] = new TrieNode();
                p = p.next[i];
            }
            p.word = w;
        }
        return root;
    }
    class TrieNode {
        TrieNode[] next = new TrieNode[26];
        String word;
    }
}
```

213.House Robber II

Note: This is an extension of House Robber.

After robbing those houses on that street, the thief has found himself a new place for his thievery so that he will not get too much attention. This time, all houses at this place are **arranged in a circle.** That means the first house is the neighbor of the last one. Meanwhile, the security system for these houses remain the same as for those in the previous street.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight **without alerting the police**.

```
public class Solution {
    public int rob(int[] nums) {
        if (nums.length == 1)
            return nums[0];
        return Math.max(rob(nums, 0, nums.length - 2),
                rob(nums, 1, nums.length - 1));
```

```
    public int rob(int[] nums, int start, int end) {
        int prevMax = 0;
        int currMax = 0;
        for (int i = start; i <= end; ++i) {
            int x = nums[i];
            int temp = currMax;
            currMax = Math.max(prevMax + x, currMax);
            prevMax = temp;
        }
        return currMax;
    }
}
```

214.Shortest Palindrome

Given a string S, you are allowed to convert it to a palindrome by adding characters in front of it. Find and return the shortest palindrome you can find by performing this transformation.
For example:
Given `"aacecaaa"`, return `"aaacecaaa"`.
Given `"abcd"`, return `"dcbabcd"`.

```
public class Solution {
    public String shortestPalindrome(String s) {
        int n = s.length();
        String rev = new StringBuilder(s).reverse().toString();
        String s_new = s + "#" + rev;
        int m = s_new.length();
        int[] f = new int[m];
        for (int i = 1; i < m; i++) {
            int t = f[i - 1];
            while (t > 0 && s_new.charAt(i) != s_new.charAt(t))
                t = f[t - 1];
            if (s_new.charAt(i) == s_new.charAt(t))
                ++t;
            f[i] = t;
        }
        return rev.substring(0, n - f[m - 1]) + s;
    }
}
```

215.Kth Largest Element in an Array

Find the **k**th largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.
For example,
Given `[3,2,1,5,6,4]` and k = 2, return 5.
Note:
You may assume k is always valid, 1 ? k ? array's length.

```
public class Solution {
    public int findKthLargest(int[] nums, int k) {
        final int N = nums.length;
        Arrays.sort(nums);
        return nums[N - k];
    }
}
public class Solution {
    public int findKthLargest(int[] nums, int k) {
        final PriorityQueue<Integer> pq = new PriorityQueue<>();
        for (int val : nums) {
            pq.offer(val);
            if (pq.size() > k) {
                pq.poll();
            }
        }
        return pq.peek();
    }
}
```

216.Combination Sum III

Find all possible combinations of ***k*** numbers that add up to a number ***n***, given that only numbers from 1 to 9 can be used and each combination should be a unique set of numbers.
Example 1:
Input: ***k*** = 3, ***n*** = 7
Output:

```
[[1,2,4]]
```

Example 2:
Input: ***k*** = 3, ***n*** = 9
Output:

```
[[1,2,6], [1,3,5], [2,3,4]]
```

```
public class Solution {
    public List<List<Integer>> combinationSum3(int k, int n) {
        List<List<Integer>> ans = new ArrayList<>();
        combination(ans, new ArrayList<Integer>(), k, 1, n);
        return ans;
    }
    private void combination(List<List<Integer>> ans, List<Integer> comb, int k,
            int start, int n) {
        if (comb.size() == k && n == 0) {
            List<Integer> li = new ArrayList<Integer>(comb);
            ans.add(li);
            return;
        }
        for (int i = start; i <= 9; i++) {
            comb.add(i);
            combination(ans, comb, k, i + 1, n - i);
            comb.remove(comb.size() - 1);
        }
    }
}
```

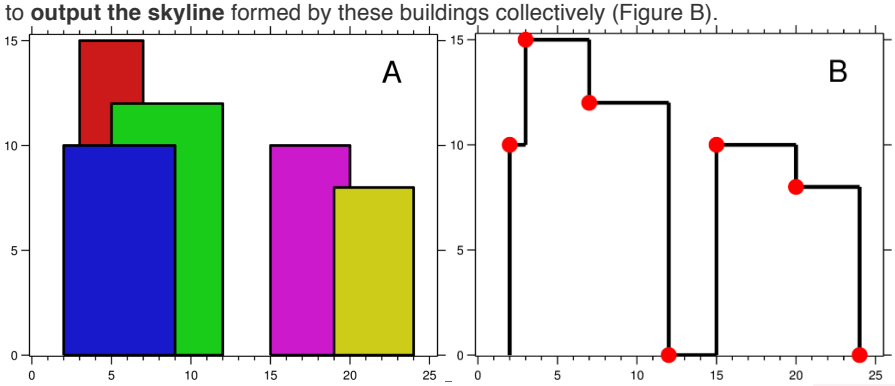
217.Contains Duplicate

Given an array of integers, find if the array contains any duplicates. Your function should return true if any value appears at least twice in the array, and it should return false if every element is distinct.

```
public class Solution {
    public boolean containsDuplicate(int[] nums) {
        Set<Integer> numSet = new HashSet<Integer>();
        for (int num : nums) {
            if (!numSet.add(num))
                return true;
        }
        return false;
    }
}
```

218.The Skyline Problem

A city's skyline is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Now suppose you are **given the locations and height of all the buildings** as shown on a cityscape photo (Figure A), write a program to **output the skyline** formed by these buildings collectively (Figure B).



The geometric information of each building is represented by a triplet of integers `[Li, Ri, Hi]`, where `Li` and `Ri` are the x coordinates of the left and right edge of the ith building, respectively, and `Hi` is its height. It is guaranteed that `0 ? Li, Ri ? INT_MAX`, `0 < Hi ? INT_MAX`, and `Ri - Li > 0`. You may assume all buildings are perfect rectangles grounded on an absolutely flat surface at height 0.

For instance, the dimensions of all buildings in Figure A are recorded as: `[ [2 9 10], [3 7 15], [5 12 12], [15 20 10], [19 24 8] ]`.

The output is a list of "**key points**" (red dots in Figure B) in the format of `[ [x1,y1], [x2, y2], [x3, y3], ... ]` that uniquely defines a skyline. **A key point is the left endpoint of a horizontal line segment**. Note that the last key point, where the rightmost building ends, is merely used to mark the termination of the skyline, and always has zero height. Also, the ground in between any two adjacent buildings should be considered part of the skyline contour.

For instance, the skyline in Figure B should be represented as:`[ [2 10], [3 15], [7 12], [12 0], [15 10], [20 8], [24, 0] ]`.

Notes:

- The number of buildings in any input list is guaranteed to be in the range `[0, 10000]`.
- The input list is already sorted in ascending order by the left x position `Li`.
- The output list must be sorted by the x position.
- There must be no consecutive horizontal lines of equal height in the output skyline. For instance, `[...[2 3], [4 5], [7 5], [11 5], [12 7]...]` is not acceptable; the three lines of height 5 should be merged into one in the final output as such: `[...[2 3], [4 5], [12 7], ...]`

```
public class Solution {
    public List<int[]> getSkyline(int[][] buildings) {
```

```
        List<int[]> heights = new ArrayList<>();
        for (int[] b : buildings) {
            heights.add(new int[] { b[0], -b[2] });
            heights.add(new int[] { b[1], b[2] });
        }
        Collections.sort(heights,
                (a, b) -> (a[0] == b[0]) ? a[1] - b[1] : a[0] - b[0]);
        TreeMap<Integer, Integer> heightMap = new TreeMap<>(
                Collections.reverseOrder());
        heightMap.put(0, 1);
        int prevHeight = 0;
        List<int[]> skyLine = new LinkedList<>();
        for (int[] h : heights) {
            if (h[1] < 0) {
                heightMap.put(-h[1], heightMap.getOrDefault(-h[1], 0) + 1);
            } else {
                Integer cnt = heightMap.get(h[1]);
                if (cnt == 1) {
                    heightMap.remove(h[1]);
                } else {
                    heightMap.put(h[1], cnt - 1);
                }
            }
            int currHeight = heightMap.firstKey();
            if (prevHeight != currHeight) {
                skyLine.add(new int[] { h[0], currHeight });
                prevHeight = currHeight;
            }
        }
        return skyLine;
    }
}
```

219.Contains Duplicate II

Given an array of integers and an integer *k*, find out whether there are two distinct indices *i* and *j* in the array such that **nums[i] = nums[j]** and the **absolute** difference between *i* and *j* is at most *k*.

```
public class Solution {
    public boolean containsNearbyDuplicate(int[] nums, int k) {
        Set<Integer> set = new HashSet<Integer>();
        for (int i = 0; i < nums.length; i++) {
            if (i > k)
                set.remove(nums[i - k - 1]);
            if (!set.add(nums[i]))
                return true;
        }
        return false;
    }
}
```

220.Contains Duplicate III

Given an array of integers, find out whether there are two distinct indices *i* and *j* in the array such that the **absolute** difference between **nums[i]** and **nums[j]** is at most *t* and the **absolute** difference between *i* and *j* is at most *k*.

```
public class Solution {
    public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {
        if (k < 1 || t < 0)
            return false;
        Map<Long, Long> map = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            long remappedNum = (long) nums[i] - Integer.MIN_VALUE;
            long bucket = remappedNum / ((long) t + 1);
            if (map.containsKey(bucket)
                    || (map.containsKey(bucket - 1)
                            && remappedNum - map.get(bucket - 1) <= t)
                    || (map.containsKey(bucket + 1)
                            && map.get(bucket + 1) - remappedNum <= t))
                return true;
            if (map.entrySet().size() >= k) {
                long lastBucket = ((long) nums[i - k] - Integer.MIN_VALUE)
                        / ((long) t + 1);
                map.remove(lastBucket);
            }
            map.put(bucket, remappedNum);
        }
        return false;
    }
}
```

221.Maximal Square

Given a 2D binary matrix filled with 0's and 1's, find the largest square containing only 1's and return its area.

For example, given the following matrix:

```
1 0 1 0 0
1 0 1 1 1
1 1 1 1 1
1 0 0 1 0
```

Return 4.

```
public class Solution {
    public int maximalSquare(char[][] matrix) {
        if (matrix.length == 0)
            return 0;
        int m = matrix.length, n = matrix[0].length, result = 0;
        int[][] b = new int[m + 1][n + 1];
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (matrix[i - 1][j - 1] == '1') {
                    b[i][j] = Math.min(Math.min(b[i][j - 1], b[i - 1][j - 1]),
                            b[i - 1][j]) + 1;
                    result = Math.max(b[i][j], result); // update result
                }
            }
        }
        return result * result;
    }
}
```

222.Count Complete Tree Nodes

Given a **complete** binary tree, count the number of nodes.

Definition of a complete binary tree from Wikipedia:

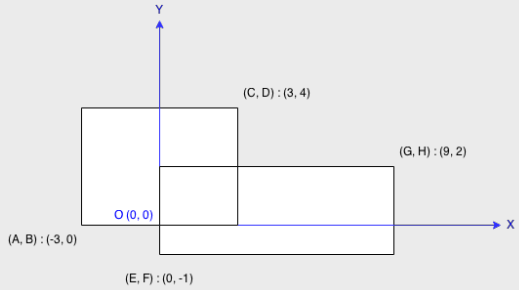
In a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and 2^h nodes inclusive at the last level h.

```
class Solution {
    int height(TreeNode root) {
        return root == null ? -1 : 1 + height(root.left);
    }
    public int countNodes(TreeNode root) {
        int nodes = 0, h = height(root);
        while (root != null) {
            if (height(root.right) == h - 1) {
                nodes += 1 << h;
                root = root.right;
            } else {
                nodes += 1 << h - 1;
                root = root.left;
            }
            h--;
        }
        return nodes;
    }
}
```

223.Rectangle Area

Find the total area covered by two **rectilinear** rectangles in a **2D** plane.

Each rectangle is defined by its bottom left corner and top right corner as shown in the figure.



Assume that the total area is never beyond the maximum possible value of **int**.

```
public class Solution {
    public int computeArea(int A, int B, int C, int D, int E, int F, int G,
            int H) {
        int left = Math.max(A, E), right = Math.max(Math.min(C, G), left);
        int bottom = Math.max(B, F), top = Math.max(Math.min(D, H), bottom);
        return (C - A) * (D - B) - (right - left) * (top - bottom)
                + (G - E) * (H - F);
    }
}
```

224.Basic Calculator

The expression string may contain open `(` and closing parentheses `)`, the plus `+` or minus sign `-`, **non-negative** integers and empty spaces .
You may assume that the given expression is always valid.
Some examples:

```
"1 + 1" = 2
" 2-1 + 2 " = 3
"(1+(4+5+2)-3)+(6+8)" = 23
```

Note: Do not use the `eval` built-in library function.

```
public class Solution {
    public static int calculate(String s) {
        int len = s.length(), sign = 1, result = 0;
        Stack<Integer> stack = new Stack<Integer>();
        for (int i = 0; i < len; i++) {
            if (Character.isDigit(s.charAt(i))) {
                int sum = s.charAt(i) - '0';
                while (i + 1 < len && Character.isDigit(s.charAt(i + 1))) {
                    sum = sum * 10 + s.charAt(i + 1) - '0';
                    i++;
                }
                result += sum * sign;
            } else if (s.charAt(i) == '+')
                sign = 1;
            else if (s.charAt(i) == '-')
                sign = -1;
            else if (s.charAt(i) == '(') {
                stack.push(result);
                stack.push(sign);
                result = 0;
                sign = 1;
            } else if (s.charAt(i) == ')') {
                result = result * stack.pop() + stack.pop();
            }
        }
        return result;
    }
}
```

225.Implement Stack using Queues

Implement the following operations of a stack using queues.

- push(x) -- Push element x onto stack.
- pop() -- Removes the element on top of the stack.
- top() -- Get the top element.
- empty() -- Return whether the stack is empty.

Notes:

- You must use *only* standard operations of a queue -- which means only `push to back`, `peek/pop from front`, `size`, and `is empty` operations are valid.
- Depending on your language, queue may not be supported natively. You may simulate a queue by using a list or deque (double-ended queue), as long as you use only standard operations of a queue.
- You may assume that all operations are valid (for example, no pop or top operations will be called on an empty stack).

```
public class MyStack {
    Queue<Integer> queue;
    public MyStack() {
        this.queue = new LinkedList<Integer>();
    }
    public void push(int x) {
        queue.add(x);
        for (int i = 0; i < queue.size() - 1; i++) {
            queue.add(queue.poll());
        }
    }
    public int pop() {
        return queue.poll();
    }
    public int top() {
        return queue.peek();
    }
    public boolean empty() {
        return queue.isEmpty();
    }
}
```

226.Invert Binary Tree

Invert a binary tree.

```
     4
   /   \
  2     7
 / \   / \
1   3 6   9
```

to

```
     4
   /   \
  7     2
 / \   / \
9   6 3   1
```

Trivia:

This problem was inspired by this original tweet by Max Howell:

> Google: 90% of our engineers use the software you wrote (Homebrew), but you can't invert a binary tree on a whiteboard so fuck off.

```
public class Solution {
    public TreeNode invertTree(TreeNode root) {
        if (root == null)
            return null;
        final Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        while (!queue.isEmpty()) {
            final TreeNode node = queue.poll();
            final TreeNode left = node.left;
            node.left = node.right;
            node.right = left;
            if (node.left != null) {
                queue.offer(node.left);
            }
            if (node.right != null) {
                queue.offer(node.right);
            }
        }
        return root;
    }
}
```

```
public class Solution {
    public TreeNode invertTree(TreeNode root) {
        if (root == null)
            return null;
        TreeNode right = invertTree(root.right);
        TreeNode left = invertTree(root.left);
        root.left = right;
        root.right = left;
        return root;
    }
}
```

227. Basic Calculator II

Implement a basic calculator to evaluate a simple expression string.

The expression string contains only **non-negative** integers, `+`, `-`, `*`, `/` operators and empty spaces . The integer division should truncate toward zero.

You may assume that the given expression is always valid.

Some examples:

```
"3+2*2" = 7
" 3/2 " = 1
" 3+5 / 2 " = 5
```

Note: Do not use the `eval` built-in library function.

```
public class Solution {
    public int calculate(String s) {
        int len;
        if (s == null || (len = s.length()) == 0)
            return 0;
        Stack<Integer> stack = new Stack<Integer>();
        int num = 0;
        char sign = '+';
        for (int i = 0; i < len; i++) {
            if (Character.isDigit(s.charAt(i))) {
                num = num * 10 + s.charAt(i) - '0';
            }
            if ((!Character.isDigit(s.charAt(i)) && ' ' != s.charAt(i))
```

```
                if (sign == '-') {
                    stack.push(-num);
                }
                if (sign == '+') {
                    stack.push(num);
                }
                if (sign == '*') {
                    stack.push(stack.pop() * num);
                }
                if (sign == '/') {
                    stack.push(stack.pop() / num);
                }
                sign = s.charAt(i);
                num = 0;
            }
        }
        int re = 0;
        for (int i : stack) {
            re += i;
        }
        return re;
    }
}
```

228.Summary Ranges

Given a sorted integer array without duplicates, return the summary of its ranges.
For example, given `[0,1,2,4,5,7]`, return `["0->2","4->5","7"].`

```
public class Solution {
    public List<String> summaryRanges(int[] nums) {
        StringBuilder sb = new StringBuilder();
        List<String> result = new LinkedList<String>();
        int lastNum = 0;
        boolean addP = false;
        for (int i = 0; i < nums.length; ++i) {
            int temp = nums[i];
            if (sb.length() == 0) {
                sb.append(temp);
                lastNum = temp;
                continue;
            }
            if (lastNum == temp - 1) {
                addP = true;
                lastNum = temp;
            } else {
                if (addP) {
                    sb.append("->");
                    addP = false;
                    sb.append(lastNum);
                }
                result.add(sb.toString());
                sb = new StringBuilder();
                --i;
            }
        }
        if (sb.length() > 0) {
            if (addP) {
                sb.append("->");
                sb.append(lastNum);
            }
            result.add(sb.toString());
        }
        return result;
    }
}
```

229.Majority Element II

Given an integer array of size *n*, find all elements that appear more than `⌊ n/3 ⌋` times. The algorithm should run in linear time and in O(1) space.

```
public class Solution {
    public List<Integer> majorityElement(int[] nums) {
        if (nums == null || nums.length == 0)
            return new ArrayList<Integer>();
        List<Integer> result = new ArrayList<Integer>();
        int number1 = nums[0], number2 = nums[0], count1 = 0, count2 = 0,
                len = nums.length;
        for (int i = 0; i < len; i++) {
            if (nums[i] == number1)
                count1++;
```

```
                count2++;
            else if (count1 == 0) {
                number1 = nums[i];
                count1 = 1;
            } else if (count2 == 0) {
                number2 = nums[i];
                count2 = 1;
            } else {
                count1--;
                count2--;
            }
        }
        count1 = 0;
        count2 = 0;
        for (int i = 0; i < len; i++) {
            if (nums[i] == number1)
                count1++;
            else if (nums[i] == number2)
                count2++;
        }
        if (count1 > len / 3)
            result.add(number1);
        if (count2 > len / 3)
            result.add(number2);
        return result;
    }
}
```

230.Kth Smallest Element in a BST

Given a binary search tree, write a function `kthSmallest` to find the **k**th smallest element in it.

Note:

You may assume k is always valid, 1 ? k ? BST's total elements.

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?

```
public class Solution {
    public int kthSmallest(TreeNode root, int k) {
        int count = countNodes(root.left);
        if (k <= count)
            return kthSmallest(root.left, k);
        else if (k > count + 1)
            return kthSmallest(root.right, k - 1 - count);
        return root.val;
    }
    public int countNodes(TreeNode n) {
        if (n == null)
            return 0;
        return 1 + countNodes(n.left) + countNodes(n.right);
    }
}
```

231.Power of Two

Given an integer, write a function to determine if it is a power of two.

```
public class Solution {
    public boolean isPowerOfTwo(int n) {
        if (n <= 0)
            return false;
        return (n & (n - 1)) == 0;
    }
}
public class Solution {
    public boolean isPowerOfTwo(int n) {
        return n > 0 && Integer.bitCount(n) == 1;
    }
}
```

232.Implement Queue using Stacks

Implement the following operations of a queue using stacks.

- push(x) -- Push element x to the back of queue.
- pop() -- Removes the element from in front of queue.
- peek() -- Get the front element.
- empty() -- Return whether the queue is empty.

Notes:

- You must use *only* standard operations of a stack -- which means only `push to top`, `peek/pop from top`, `size`, and `is empty`operations are valid.

- Depending on your language, stack may not be supported natively. You may simulate a stack by using a list or deque (double-ended queue), as long as you use only standard operations of a stack.
- You may assume that all operations are valid (for example, no pop or peek operations will be called on an empty queue).

```
public class MyQueue {
    private int front;
    private Stack<Integer> s1 = new Stack<>();
    private Stack<Integer> s2 = new Stack<>();
    public MyQueue() {
    }
    public void push(int x) {
        if (s1.empty())
            front = x;
        s1.push(x);
    }
    public int pop() {
        if (s2.isEmpty())
            while (!s1.isEmpty())
                s2.push(s1.pop());
        return s2.pop();
    }
    public int peek() {
        if (!s2.isEmpty())
            return s2.peek();
        return front;
    }
    public boolean empty() {
        return s1.isEmpty() && s2.isEmpty();
    }
}
```

233.Number of Digit One

Given an integer n, count the total number of digit 1 appearing in all non-negative integers less than or equal to n.
For example:
Given n = 13,
Return 6, because digit 1 occurred in the following numbers: 1, 10, 11, 12, 13.

```
public class Solution {
    public int countDigitOne(int n) {
        int countr = 0;
        for (long i = 1; i <= n; i *= 10) {
            long divider = i * 10;
            countr += (n / divider) * i
                    + Math.min(Math.max(n % divider - i + 1, 0L), i);
        }
        return countr;
    }
}
```

234.Palindrome Linked List

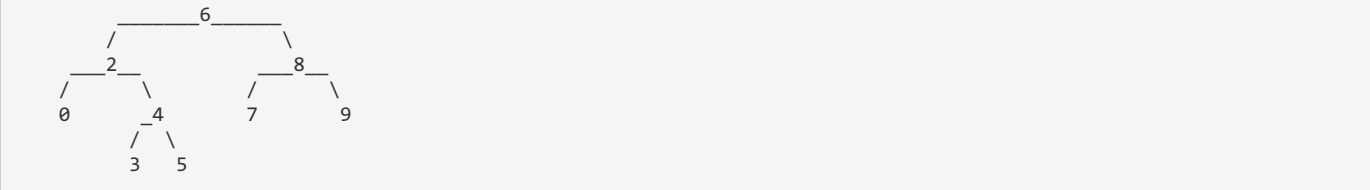
Given a singly linked list, determine if it is a palindrome.
Follow up:
Could you do it in O(n) time and O(1) space?

```
public class Solution {
    public boolean isPalindrome(ListNode head) {
        ListNode fast = head, slow = head;
        while (fast != null && fast.next != null) {
            fast = fast.next.next;
            slow = slow.next;
        }
        if (fast != null) // odd nodes: let right half smaller
            slow = slow.next;
        slow = reverse(slow);
        fast = head;
        while (slow != null) {
            if (fast.val != slow.val)
                return false;
            fast = fast.next;
            slow = slow.next;
        }
        return true;
    }
    public ListNode reverse(ListNode head) {
        ListNode prev = null;
        while (head != null) {
            ListNode next = head.next;
            head.next = prev;
            prev = head;
            head = next;
        }
```

```
        return prev;
    }
}
```

235.Lowest Common Ancestor of a Binary Search Tree

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.
According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow **a node to be a descendant of itself**).”

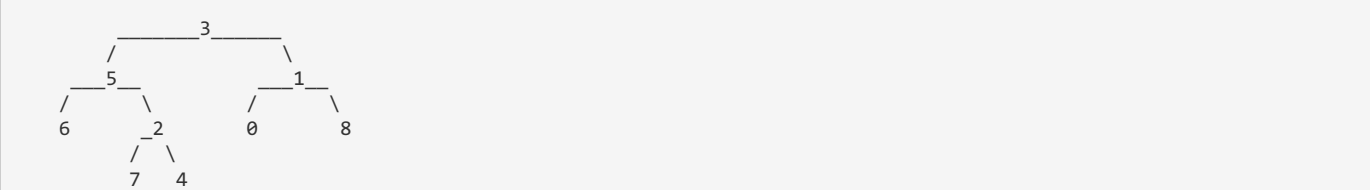


For example, the lowest common ancestor (LCA) of nodes `2` and `8` is `6`. Another example is LCA of nodes `2` and `4` is `2`, since a node can be a descendant of itself according to the LCA definition.

```
public class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p,
            TreeNode q) {
        while ((root.val - p.val) * (root.val - q.val) > 0)
            root = p.val < root.val ? root.left : root.right;
        return root;
    }
}
```

236.Lowest Common Ancestor of a Binary Tree

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.
According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow **a node to be a descendant of itself**).”



For example, the lowest common ancestor (LCA) of nodes `5` and `1` is `3`. Another example is LCA of nodes `5` and `4` is `5`, since a node can be a descendant of itself according to the LCA definition.

```
public class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p,
            TreeNode q) {
        if (root == null || root == p || root == q)
            return root;
        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);
        return left == null ? right : right == null ? left : root;
    }
}
```

237.Delete Node in a Linked List

Write a function to delete a node (except the tail) in a singly linked list, given only access to that node.
Supposed the linked list is `1 -> 2 -> 3 -> 4` and you are given the third node with value `3`, the linked list should become `1 -> 2 -> 4` after calling your function.

```
public class Solution {
    public void deleteNode(ListNode node) {
        node.val = node.next.val;
        node.next = node.next.next;
    }
}
```

238.Product of Array Except Self

Given an array of *n* integers where *n* > 1, `nums`, return an array `output` such that `output[i]` is equal to the product of all the elements of `nums` except `nums[i]`.
Solve it **without division** and in O(*n*).
For example, given `[1,2,3,4]`, return `[24,12,8,6]`.
Follow up:
Could you solve it with constant space complexity? (Note: The output array **does not** count as extra space for the purpose of space complexity analysis.)

```
public class Solution {
    public int[] productExceptSelf(int[] nums) {
        int n = nums.length;
        int[] res = new int[n];
        res[0] = 1;
        for (int i = 1; i < n; i++) {
```

```
        }
        int right = 1;
        for (int i = n - 1; i >= 0; i--) {
            res[i] *= right;
            right *= nums[i];
        }
        return res;
    }
}
```

239.Sliding Window Maximum

Given an array *nums*, there is a sliding window of size *k* which is moving from the very left of the array to the very right. You can only see the *k* numbers in the window. Each time the sliding window moves right by one position.
For example,
Given *nums* = `[1,3,-1,-3,5,3,6,7]`, and *k* = 3.

```
Window position                Max
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

Therefore, return the max sliding window as `[3,3,5,5,6,7]`.
Note:
You may assume *k* is always valid, ie: 1 ≤ k ≤ input array's size for non-empty array.
Follow up:
Could you solve it in linear time?

```
public class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        if (nums == null || k <= 0)
            return new int[0];
        int n = nums.length;
        int[] r = new int[n - k + 1];
        int ri = 0;
        Deque<Integer> q = new ArrayDeque<>();
        for (int i = 0; i < nums.length; i++) {
            while (!q.isEmpty() && q.peek() < i - k + 1)
                q.poll();
            while (!q.isEmpty() && nums[q.peekLast()] < nums[i])
                q.pollLast();
            q.offer(i);
            if (i >= k - 1)
                r[ri++] = nums[q.peek()];
        }
        return r;
    }
}
```

240.Search a 2D Matrix II

Write an efficient algorithm that searches for a value in an *m* x *n* matrix. This matrix has the following properties:

- Integers in each row are sorted in ascending from left to right.
- Integers in each column are sorted in ascending from top to bottom.

For example,
Consider the following matrix:

```
[ [1,   4,  7, 11, 15],
  [2,   5,  8, 12, 19],
  [3,   6,  9, 16, 22],
  [10, 13, 14, 17, 24],
  [18, 21, 23, 26, 30]]
```

Given **target** = `5`, return `true`.
Given **target** = `20`, return `false`.

```
public class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        if (matrix == null || matrix.length < 1 || matrix[0].length < 1) {
            return false;
        }
        int col = matrix[0].length - 1;
        int row = 0;
        while (col >= 0 && row <= matrix.length - 1) {
            if (target == matrix[row][col]) {
                return true;
            } else if (target < matrix[row][col]) {
                col--;
```

```
            } else if (target > matrix[row][col]) {
                row++;
            }
        }
        return false;
    }
}
```

241.Different Ways to Add Parentheses

Given a string of numbers and operators, return all possible results from computing all the different possible ways to group numbers and operators. The valid operators are `+`, `-` and `*`.

Example 1

Input: `"2-1-1"`.

```
((2-1)-1) = 0
(2-(1-1)) = 2
```

Output: `[0, 2]`

Example 2

Input: `"2*3-4*5"`

```
(2*(3-(4*5))) = -34
((2*3)-(4*5)) = -14
((2*(3-4))*5) = -10
(2*((3-4)*5)) = -10
(((2*3)-4)*5) = 10
```

Output: `[-34, -14, -10, -10, 10]`

```
public class Solution {
    public List<Integer> diffWaysToCompute(String input) {
        List<Integer> ret = new LinkedList<Integer>();
        for (int i = 0; i < input.length(); i++) {
            if (input.charAt(i) == '-' || input.charAt(i) == '*'
                    || input.charAt(i) == '+') {
                String part1 = input.substring(0, i);
                String part2 = input.substring(i + 1);
                List<Integer> part1Ret = diffWaysToCompute(part1);
                List<Integer> part2Ret = diffWaysToCompute(part2);
                for (Integer p1 : part1Ret) {
                    for (Integer p2 : part2Ret) {
                        int c = 0;
                        switch (input.charAt(i)) {
                        case '+':
                            c = p1 + p2;
                            break;
                        case '-':
                            c = p1 - p2;
                            break;
                        case '*':
                            c = p1 * p2;
                            break;
                        }
                        ret.add(c);
                    }
                }
            }
        }
        if (ret.size() == 0) {
            ret.add(Integer.valueOf(input));
        }
        return ret;
    }
}
```

242.Valid Anagram

Given two strings *s* and *t*, write a function to determine if *t* is an anagram of *s*.
For example,
s = "anagram", *t* = "nagaram", return true.
s = "rat", *t* = "car", return false.
Note:
You may assume the string contains only lowercase alphabets.
Follow up:
What if the inputs contain unicode characters? How would you adapt your solution to such case?

```
public class Solution {
    public boolean isAnagram(String s, String t) {
        if (s.length() != t.length()) {
```

```
            return false;
        }
        int[] counter = new int[26];
        for (int i = 0; i < s.length(); i++) {
            counter[s.charAt(i) - 'a']++;
            counter[t.charAt(i) - 'a']--;
        }
        for (int count : counter) {
            if (count != 0) {
                return false;
            }
        }
        return true;
    }
}
```

243.Shortest Word Distance

Given a list of words and two words *word1* and *word2*, return the shortest distance between these two words in the list.
For example,
Assume that words = `["practice", "makes", "perfect", "coding", "makes"]`.
Given *word1* = `“coding”`, *word2* = `“practice”`, return 3.
Given *word1* = `"makes"`, *word2* = `"coding"`, return 1.
Note:
You may assume that *word1* **does not equal to** *word2*, and *word1* and *word2* are both in the list.

```
public class Solution {
    public int shortestDistance(String[] words, String word1, String word2) {
        int i1 = -1, i2 = -1;
        int minDistance = words.length;
        for (int i = 0; i < words.length; i++) {
            if (words[i].equals(word1)) {
                i1 = i;
            } else if (words[i].equals(word2)) {
                i2 = i;
            }
            if (i1 != -1 && i2 != -1) {
                minDistance = Math.min(minDistance, Math.abs(i1 - i2));
            }
        }
        return minDistance;
    }
}
```

244.Shortest Word Distance II

This is a **follow up** of Shortest Word Distance. The only difference is now you are given the list of words and your method will be called *repeatedly* many times with different parameters. How would you optimize it?
Design a class which receives a list of words in the constructor, and implements a method that takes two words *word1* and *word2* and return the shortest distance between these two words in the list.
For example,
Assume that words = `["practice", "makes", "perfect", "coding", "makes"]`.
Given *word1* = `“coding”`, *word2* = `“practice”`, return 3.
Given *word1* = `"makes"`, *word2* = `"coding"`, return 1.
Note:
You may assume that *word1* **does not equal to** *word2*, and *word1* and *word2* are both in the list.

```
public class WordDistance {
    private Map<String, List<Integer>> map;
    public WordDistance(String[] words) {
        map = new HashMap<String, List<Integer>>();
        for (int i = 0; i < words.length; i++) {
            String w = words[i];
            if (map.containsKey(w)) {
                map.get(w).add(i);
            } else {
                List<Integer> list = new ArrayList<Integer>();
                list.add(i);
                map.put(w, list);
            }
        }
    }
    public int shortest(String word1, String word2) {
        List<Integer> list1 = map.get(word1);
        List<Integer> list2 = map.get(word2);
        int ret = Integer.MAX_VALUE;
        for (int i = 0, j = 0; i < list1.size() && j < list2.size();) {
            int index1 = list1.get(i), index2 = list2.get(j);
            if (index1 < index2) {
                ret = Math.min(ret, index2 - index1);
                i++;
```

```
                ret = Math.min(ret, index1 - index2);
                j++;
            }
        }
        return ret;
    }
}
```

245.Shortest Word Distance III

This is a **follow up** of Shortest Word Distance. The only difference is now *word1* could be the same as *word2*.
Given a list of words and two words *word1* and *word2*, return the shortest distance between these two words in the list.
word1 and *word2* may be the same and they represent two individual words in the list.
For example,
Assume that words = `["practice", "makes", "perfect", "coding", "makes"]`.
Given *word1* = “`makes`”, *word2* = “`coding`”, return 1.
Given *word1* = `"makes"`, *word2* = `"makes"`, return 3.
Note:
You may assume *word1* and *word2* are both in the list.

```
public class Solution {
    public int shortestWordDistance(String[] words, String word1,
            String word2) {
        long dist = Integer.MAX_VALUE, i1 = dist, i2 = -dist;
        for (int i = 0; i < words.length; i++) {
            if (words[i].equals(word1))
                i1 = i;
            if (words[i].equals(word2)) {
                if (word1.equals(word2))
                    i1 = i2;
                i2 = i;
            }
            dist = Math.min(dist, Math.abs(i1 - i2));
        }
        return (int) dist;
    }
}
```

246.Strobogrammatic Number

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).
Write a function to determine if a number is strobogrammatic. The number is represented as a string.
For example, the numbers "69", "88", and "818" are all strobogrammatic.

```
public class Solution {
    public boolean isStrobogrammatic(String num) {
        for (int i = 0, j = num.length() - 1; i <= j; i++, j--)
            if (!"00 11 88 696".contains(num.charAt(i) + "" + num.charAt(j)))
                return false;
        return true;
    }
}
```

247.Strobogrammatic Number II

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).
Find all strobogrammatic numbers that are of length = n.
For example,
Given n = 2, return `["11","69","88","96"]`.

```
public class Solution {
    public List<String> findStrobogrammatic(int n) {
        return helper(n, n);
    }
    List<String> helper(int n, int m) {
        if (n == 0)
            return new ArrayList<String>(Arrays.asList(""));
        if (n == 1)
            return new ArrayList<String>(Arrays.asList("0", "1", "8"));
        List<String> list = helper(n - 2, m);
        List<String> res = new ArrayList<String>();
        for (int i = 0; i < list.size(); i++) {
            String s = list.get(i);
            if (n != m)
                res.add("0" + s + "0");
            res.add("1" + s + "1");
            res.add("6" + s + "9");
            res.add("8" + s + "8");
            res.add("9" + s + "6");
        }
        return res;
    }
}
```

248.Strobogrammatic Number III

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).
Write a function to count the total strobogrammatic numbers that exist in the range of low <= num <= high.
For example,
Given low = "50", high = "100", return 3. Because 69, 88, and 96 are three strobogrammatic numbers.
Note:
Because the range might be a large number, the *low* and *high* numbers are represented as string.

```
public class Solution {
    private static final char[][] pairs = { { '0', '0' }, { '1', '1' },
            { '6', '9' }, { '8', '8' }, { '9', '6' } };
    public int strobogrammaticInRange(String low, String high) {
        int[] count = { 0 };
        for (int len = low.length(); len <= high.length(); len++) {
            char[] c = new char[len];
            dfs(low, high, c, 0, len - 1, count);
        }
        return count[0];
    }
    public void dfs(String low, String high, char[] c, int left, int right,
            int[] count) {
        if (left > right) {
            String s = new String(c);
            if ((s.length() == low.length() && s.compareTo(low) < 0)
                    || (s.length() == high.length() && s.compareTo(high) > 0)) {
                return;
            }
            count[0]++;
            return;
        }
        for (char[] p : pairs) {
            c[left] = p[0];
            c[right] = p[1];
            if (c.length != 1 && c[0] == '0') {
                continue;
            }
            if (left == right && p[0] != p[1]) {
                continue;
            }
            dfs(low, high, c, left + 1, right - 1, count);
        }
    }
}
```

249.Group Shifted Strings

Given a string, we can "shift" each of its letter to its successive letter, for example: `"abc" -> "bcd"`. We can keep "shifting" which forms the sequence:

```
"abc" -> "bcd" -> ... -> "xyz"
```

Given a list of strings which contains only lowercase alphabets, group all strings that belong to the same shifting sequence.
For example, given: `["abc", "bcd", "acef", "xyz", "az", "ba", "a", "z"]`,
A solution is:

```
[ ["abc","bcd","xyz"],
  ["az","ba"],
  ["acef"],
  ["a","z"]]
```

```
public class Solution {
    public List<List<String>> groupStrings(String[] strings) {
        HashMap<String, ArrayList<String>> map = new HashMap<String, ArrayList<String>>();
        for (String s : strings) {
            StringBuilder keySb = new StringBuilder();
            for (int i = 1; i < s.length(); i++)
                keySb.append(String.format("%2d",
                        (s.charAt(i) - s.charAt(i - 1) + 26) % 26));
            String key = keySb.toString();
            if (!map.containsKey(key))
                map.put(key, new ArrayList<String>());
            map.get(key).add(s);
        }
        return new ArrayList<List<String>>(map.values());
    }
}
```

250.Count Univalue Subtrees

Given a binary tree, count the number of uni-value subtrees.
A Uni-value subtree means all nodes of the subtree have the same value.

For example:
Given binary tree,

```
              5
             / \
            1   5
           / \   \
          5   5   5
```

return `4`.

```
public class Solution {
    public int countUnivalSubtrees(TreeNode root) {
        int[] count = new int[1];
        helper(root, count);
        return count[0];
    }
    private boolean helper(TreeNode node, int[] count) {
        if (node == null) {
            return true;
        }
        boolean left = helper(node.left, count);
        boolean right = helper(node.right, count);
        if (left && right) {
            if (node.left != null && node.val != node.left.val) {
                return false;
            }
            if (node.right != null && node.val != node.right.val) {
                return false;
            }
            count[0]++;
            return true;
        }
        return false;
    }
}
```

251.Flatten 2D Vector

Implement an iterator to flatten a 2d vector.
For example,
Given 2d vector =

```
[ [1,2],
  [3],
  [4,5,6]]
```

By calling *next* repeatedly until *hasNext* returns false, the order of elements returned by *next* should be: `[1,2,3,4,5,6]`.
Follow up:
As an added challenge, try to code it using only iterators in C++ or iterators in Java.

```
public class Vector2D implements Iterator<Integer> {
    private Iterator<List<Integer>> i;
    private Iterator<Integer> j;
    public Vector2D(List<List<Integer>> vec2d) {
        i = vec2d.iterator();
    }
    @Override
    public Integer next() {
        hasNext();
        return j.next();
    }
    @Override
    public boolean hasNext() {
        while ((j == null || !j.hasNext()) && i.hasNext())
            j = i.next().iterator();
        return j != null && j.hasNext();
    }
}
```

252.Meeting Rooms

Given an array of meeting time intervals consisting of start and end times `[[s1,e1],[s2,e2],...]` ($s_i < e_i$), determine if a person could attend all meetings.
For example,
Given `[[0, 30],[5, 10],[15, 20]]`,
return `false`.

```
public class Solution {
    public boolean canAttendMeetings(Interval[] intervals) {
        if (intervals == null)
            return false;
        Arrays.sort(intervals, (a, b) -> (a.start - b.start));
```

```
            if (intervals[i].start < intervals[i - 1].end)
                return false;
        return true;
    }
}
```

253.Meeting Rooms II

Given an array of meeting time intervals consisting of start and end times `[[s1,e1],[s2,e2],...]` ($s_i < e_i$), find the minimum number of conference rooms required.
For example,
Given `[[0, 30],[5, 10],[15, 20]]`,
return `2`.

```
public class Solution {
    public int minMeetingRooms(Interval[] intervals) {
        int[] starts = new int[intervals.length];
        int[] ends = new int[intervals.length];
        for (int i = 0; i < intervals.length; i++) {
            starts[i] = intervals[i].start;
            ends[i] = intervals[i].end;
        }
        Arrays.sort(starts);
        Arrays.sort(ends);
        int rooms = 0;
        int endsItr = 0;
        for (int i = 0; i < starts.length; i++) {
            if (starts[i] < ends[endsItr])
                rooms++;
            else
                endsItr++;
        }
        return rooms;
    }
}
```

254.Factor Combinations

Numbers can be regarded as product of its factors. For example,

```
8 = 2 x 2 x 2;
  = 2 x 4.
```

Write a function that takes an integer *n* and return all possible combinations of its factors.
Note:

1. You may assume that *n* is always positive.
2. Factors should be greater than 1 and less than *n*.

Examples:
input: `1`
output:

```
[]
```

input: `37`
output:

```
[]
```

input: `12`
output:

```
[ [2, 6],
  [2, 2, 3],
  [3, 4]]
```

input: `32`
output:

```
[ [2, 16],
  [2, 2, 8],
  [2, 2, 2, 4],
  [2, 2, 2, 2, 2],
  [2, 4, 4],
  [4, 8]]
```

```
public class Solution {
    public List<List<Integer>> getFactors(int n) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
```

```
        return result;
    }
    public void helper(List<List<Integer>> result, List<Integer> item, int n,
            int start) {
        if (n <= 1) {
            if (item.size() > 1) {
                result.add(new ArrayList<Integer>(item));
            }
            return;
        }
        for (int i = start; i <= n; ++i) {
            if (n % i == 0) {
                item.add(i);
                helper(result, item, n / i, i);
                item.remove(item.size() - 1);
            }
        }
    }
}
```

255.Verify Preorder Sequence in Binary Search Tree

Given an array of numbers, verify whether it is the correct preorder traversal sequence of a binary search tree.
You may assume each number in the sequence is unique.
Follow up:
Could you do it using only constant space complexity?

```
public class Solution {
    public boolean verifyPreorder(int[] preorder) {
        int low = Integer.MIN_VALUE;
        Stack<Integer> path = new Stack<>();
        for (int p : preorder) {
            if (p < low)
                return false;
            while (!path.empty() && p > path.peek())
                low = path.pop();
            path.push(p);
        }
        return true;
    }
}
public class Solution {
    public boolean verifyPreorder(int[] preorder) {
        int low = Integer.MIN_VALUE, i = -1;
        for (int p : preorder) {
            if (p < low)
                return false;
            while (i >= 0 && p > preorder[i])
                low = preorder[i--];
            preorder[++i] = p;
        }
        return true;
    }
}
```

256.Paint House

There are a row of *n* houses, each house can be painted with one of the three colors: red, blue or green. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.
The cost of painting each house with a certain color is represented by a `n x 3` cost matrix. For example, `costs[0][0]` is the cost of painting house 0 with color red; `costs[1][2]` is the cost of painting house 1 with color green, and so on... Find the minimum cost to paint all houses.
Note:
All costs are positive integers.

```
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]);
    }
}
```

257.Binary Tree Paths

Given a binary tree, return all root-to-leaf paths.

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

All root-to-leaf paths are:

```
["1->2->5", "1->3"]
```

```java
public class Solution {
    public List<String> binaryTreePaths(TreeNode root) {
        List<String> answer = new ArrayList<String>();
        if (root != null)
            searchBT(root, "", answer);
        return answer;
    }
    private void searchBT(TreeNode root, String path, List<String> answer) {
        if (root.left == null && root.right == null)
            answer.add(path + root.val);
        if (root.left != null)
            searchBT(root.left, path + root.val + "->", answer);
        if (root.right != null)
            searchBT(root.right, path + root.val + "->", answer);
    }
}
```

258.Add Digits

Given a non-negative integer `num`, repeatedly add all its digits until the result has only one digit.
For example:
Given `num = 38`, the process is like: `3 + 8 = 11`, `1 + 1 = 2`. Since `2` has only one digit, return it.
Follow up:
Could you do it without any loop/recursion in O(1) runtime?
Ref: Digit Root: https://en.wikipedia.org/wiki/Digital_root#Congruence_formula

```java
public class Solution {
    public int addDigits(int num) {
        return 1 + (num - 1) % 9;
    }
}
```

259.3Sum Smaller

Given an array of *n* integers *nums* and a *target*, find the number of index triplets `i, j, k` with `0 <= i < j < k < n` that satisfy the condition `nums[i] + nums[j] + nums[k] < target`.
For example, given *nums* = `[-2, 0, 1, 3]`, and *target* = 2.
Return 2. Because there are two triplets which sums are less than 2:

```
[-2, 0, 1]
[-2, 0, 3]
```

Follow up:
Could you solve it in $O(n^2)$ runtime?

```java
public class Solution {
    public int threeSumSmaller(int[] nums, int target) {
        Arrays.sort(nums);
        int sum = 0;
        for (int i = 0; i < nums.length - 2; i++) {
            sum += twoSumSmaller(nums, i + 1, target - nums[i]);
        }
        return sum;
    }
    private int twoSumSmaller(int[] nums, int startIndex, int target) {
        int sum = 0;
        int left = startIndex;
        int right = nums.length - 1;
        while (left < right) {
            if (nums[left] + nums[right] < target) {
                sum += right - left;
                left++;
            } else
                right--;
        }
        return sum;
    }
}
```

260.Single Number III

Given an array of numbers `nums`, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once.

Given `nums = [1, 2, 1, 3, 2, 5]`, return `[3, 5]`.

Note:

1. The order of the result is not important. So in the above example, `[5, 3]` is also correct.
2. Your algorithm should run in linear runtime complexity. Could you implement it using only constant space complexity?

```
public class Solution {
    public int[] singleNumber(int[] nums) {
        int diff = 0;
        for (int num : nums) {
            diff ^= num;
        }
        diff &= -diff;
        int[] rets = { 0, 0 };
        for (int num : nums) {
            if ((num & diff) == 0)
                rets[0] ^= num;
            else
                rets[1] ^= num;
        }
        return rets;
    }
}
```

261.Graph Valid Tree

Given `n` nodes labeled from `0` to `n - 1` and a list of undirected edges (each edge is a pair of nodes), write a function to check whether these edges make up a valid tree.

For example:

Given `n = 5` and `edges = [[0, 1], [0, 2], [0, 3], [1, 4]]`, return `true`.

Given `n = 5` and `edges = [[0, 1], [1, 2], [2, 3], [1, 3], [1, 4]]`, return `false`.

Note: you can assume that no duplicate edges will appear in `edges`. Since all edges are undirected, `[0, 1]` is the same as `[1, 0]`and thus will not appear together in `edges`.

```
public class Solution {
    public boolean validTree(int n, int[][] edges) {
        int[] nums = new int[n];
        Arrays.fill(nums, -1);
        for (int i = 0; i < edges.length; i++) {
            int x = find(nums, edges[i][0]);
            int y = find(nums, edges[i][1]);
            if (x == y)
                return false;
            nums[x] = y;
        }
        return edges.length == n - 1;
    }
    int find(int nums[], int i) {
        if (nums[i] == -1)
            return i;
        return find(nums, nums[i]);
    }
}
```

262.Trips and Users

```
Create table If Not Exists Trips (Id int, Client_Id int, Driver_Id int,
City_Id int, Status ENUM('completed', 'cancelled_by_driver', 'cancelled_by_client'),
Request_at varchar(50));
Create table If Not Exists Users (Users_Id int,
Banned varchar(50), Role ENUM('client', 'driver', 'partner'));
Truncate table Trips;
insert into Trips (Id, Client_Id, Driver_Id, City_Id, Status,
Request_at) values ('1', '1', '10', '1', 'completed', '2013-10-01');
insert into Trips (Id, Client_Id, Driver_Id, City_Id, Status,
Request_at) values ('2', '2', '11', '1', 'cancelled_by_driver', '2013-10-01');
insert into Trips (Id, Client_Id, Driver_Id, City_Id, Status,
Request_at) values ('3', '3', '12', '6', 'completed', '2013-10-01');
insert into Trips (Id, Client_Id, Driver_Id, City_Id, Status,
Request_at) values ('4', '4', '13', '6', 'cancelled_by_client', '2013-10-01');
insert into Trips (Id, Client_Id, Driver_Id, City_Id, Status,
Request_at) values ('5', '1', '10', '1', 'completed', '2013-10-02');
insert into Trips (Id, Client_Id, Driver_Id, City_Id, Status,
Request_at) values ('6', '2', '11', '6', 'completed', '2013-10-02');
insert into Trips (Id, Client_Id, Driver_Id, City_Id, Status,
Request_at) values ('7', '3', '12', '6', 'completed', '2013-10-02');
insert into Trips (Id, Client_Id, Driver_Id, City_Id, Status,
Request_at) values ('8', '2', '12', '12', 'completed', '2013-10-03');
insert into Trips (Id, Client_Id, Driver_Id, City_Id, Status,
Request_at) values ('9', '3', '10', '12', 'completed', '2013-10-03');
insert into Trips (Id, Client_Id, Driver_Id, City_Id, Status,
Request_at) values ('10', '4', '13', '12', 'cancelled_by_driver', '2013-10-03');
```

```
insert into Users (Users_Id, Banned, Role) values ('1', 'No', 'client');
insert into Users (Users_Id, Banned, Role) values ('2', 'Yes', 'client');
insert into Users (Users_Id, Banned, Role) values ('3', 'No', 'client');
insert into Users (Users_Id, Banned, Role) values ('4', 'No', 'client');
insert into Users (Users_Id, Banned, Role) values ('10', 'No', 'driver');
insert into Users (Users_Id, Banned, Role) values ('11', 'No', 'driver');
insert into Users (Users_Id, Banned, Role) values ('12', 'No', 'driver');
insert into Users (Users_Id, Banned, Role) values ('13', 'No', 'driver');
```

The `Trips` table holds all taxi trips. Each trip has a unique Id, while Client_Id and Driver_Id are both foreign keys to the Users_Id at the `Users` table. Status is an ENUM type of ('completed', 'cancelled_by_driver', 'cancelled_by_client').

```
+----+-----------+-----------+---------+--------------------+----------+
| Id | Client_Id | Driver_Id | City_Id |        Status      |Request_at|
+----+-----------+-----------+---------+--------------------+----------+
| 1  |     1     |    10     |    1    |     completed      |2013-10-01|
| 2  |     2     |    11     |    1    | cancelled_by_driver|2013-10-01|
| 3  |     3     |    12     |    6    |     completed      |2013-10-01|
| 4  |     4     |    13     |    6    | cancelled_by_client|2013-10-01|
| 5  |     1     |    10     |    1    |     completed      |2013-10-02|
| 6  |     2     |    11     |    6    |     completed      |2013-10-02|
| 7  |     3     |    12     |    6    |     completed      |2013-10-02|
| 8  |     2     |    12     |    12   |     completed      |2013-10-03|
| 9  |     3     |    10     |    12   |     completed      |2013-10-03|
| 10 |     4     |    13     |    12   | cancelled_by_driver|2013-10-03|
+----+-----------+-----------+---------+--------------------+----------+
```

The `Users` table holds all users. Each user has an unique Users_Id, and Role is an ENUM type of ('client', 'driver', 'partner').

```
+----------+--------+--------+
| Users_Id | Banned |  Role  |
+----------+--------+--------+
|    1     |   No   | client |
|    2     |   Yes  | client |
|    3     |   No   | client |
|    4     |   No   | client |
|    10    |   No   | driver |
|    11    |   No   | driver |
|    12    |   No   | driver |
|    13    |   No   | driver |
+----------+--------+--------+
```

Write a SQL query to find the cancellation rate of requests made by unbanned clients between **Oct 1, 2013** and **Oct 3, 2013**. For the above tables, your SQL query should return the following rows with the cancellation rate being rounded to *two* decimal places.

```
+------------+-------------------+
|     Day    | Cancellation Rate |
+------------+-------------------+
| 2013-10-01 |       0.33        |
| 2013-10-02 |       0.00        |
| 2013-10-03 |       0.50        |
+------------+-------------------+
```

```
select
t.Request_at Day,
round(sum(case when t.Status like 'cancelled_%' then 1 else 0 end)/count(*),2) Rate
from Trips t
inner join Users u
on t.Client_Id = u.Users_Id and u.Banned='No'
where t.Request_at between '2013-10-01' and '2013-10-03'
group by t.Request_at
```

263.Ugly Number

Write a program to check whether a given number is an ugly number.
Ugly numbers are positive numbers whose prime factors only include `2, 3, 5`. For example, `6, 8` are ugly while `14` is not ugly since it includes another prime factor `7`.
Note that `1` is typically treated as an ugly number.

```
public class Solution {
    public boolean isUgly(int num) {
        for (int i = 2; i < 6 && num > 0; i++)
            while (num % i == 0)
                num /= i;
        return num == 1;
    }
}
```

264.Ugly Number II

Write a program to find the `n`-th ugly number.
Ugly numbers are positive numbers whose prime factors only include `2, 3, 5`. For example, `1, 2, 3, 4, 5, 6, 8, 9, 10, 12` is the sequence of the first `10` ugly numbers.

Note that `1` is typically treated as an ugly number, and *n* **does not exceed 1690**.

```
public class Solution {
    public int nthUglyNumber(int n) {
        int[] ugly = new int[n];
        ugly[0] = 1;
        int index2 = 0, index3 = 0, index5 = 0;
        int factor2 = 2, factor3 = 3, factor5 = 5;
        for (int i = 1; i < n; i++) {
            int min = Math.min(Math.min(factor2, factor3), factor5);
            ugly[i] = min;
            if (factor2 == min)
                factor2 = 2 * ugly[++index2];
            if (factor3 == min)
                factor3 = 3 * ugly[++index3];
            if (factor5 == min)
                factor5 = 5 * ugly[++index5];
        }
        return ugly[n - 1];
    }
}
```

265.Paint House II

There are a row of *n* houses, each house can be painted with one of the *k* colors. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by a `n x k` cost matrix. For example, `costs[0][0]` is the cost of painting house 0 with color 0; `costs[1][2]` is the cost of painting house 1 with color 2, and so on... Find the minimum cost to paint all houses.

Note:

All costs are positive integers.

Follow up:

Could you solve it in *O*(*nk*) runtime?

```
public class Solution {
    public int minCostII(int[][] costs) {
        if (costs == null || costs.length == 0)
            return 0;
        int n = costs.length, k = costs[0].length;
        int min1 = -1, min2 = -1;
        for (int i = 0; i < n; i++) {
            int last1 = min1, last2 = min2;
            min1 = -1;
            min2 = -1;
            for (int j = 0; j < k; j++) {
                if (j != last1)
                    costs[i][j] += last1 < 0 ? 0 : costs[i - 1][last1];
                else
                    costs[i][j] += last2 < 0 ? 0 : costs[i - 1][last2];
                if (min1 < 0 || costs[i][j] < costs[i][min1]) {
                    min2 = min1;
                    min1 = j;
                } else if (min2 < 0 || costs[i][j] < costs[i][min2])
                    min2 = j;
            }
        }
        return costs[n - 1][min1];
    }
}
```

266.Palindrome Permutation

Given a string, determine if a permutation of the string could form a palindrome.

For example,

`"code"` -> False, `"aab"` -> True, `"carerac"` -> True.

```
public class Solution {
    public boolean canPermutePalindrome(String s) {
        int[] map = new int[128];
        int count = 0;
        for (int i = 0; i < s.length(); i++) {
            map[s.charAt(i)]++;
            if (map[s.charAt(i)] % 2 == 0)
                count--;
            else
                count++;
        }
        return count <= 1;
    }
}
```

267.Palindrome Permutation II

Given a string `s`, return all the palindromic permutations (without duplicates) of it. Return an empty list if no palindromic permutation

For example:
Given `s = "aabb"`, return `["abba", "baab"]`.
Given `s = "abc"`, return `[]`.

```
public class Solution {
    public List<String> generatePalindromes(String s) {
        int odd = 0;
        String mid = "";
        List<String> res = new ArrayList<>();
        List<Character> list = new ArrayList<>();
        Map<Character, Integer> map = new HashMap<>();
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            map.put(c, map.containsKey(c) ? map.get(c) + 1 : 1);
            odd += map.get(c) % 2 != 0 ? 1 : -1;
        }
        if (odd > 1)
            return res;
        for (Map.Entry<Character, Integer> entry : map.entrySet()) {
            char key = entry.getKey();
            int val = entry.getValue();
            if (val % 2 != 0)
                mid += key;
            for (int i = 0; i < val / 2; i++)
                list.add(key);
        }
        getPerm(list, mid, new boolean[list.size()], new StringBuilder(), res);
        return res;
    }
    void getPerm(List<Character> list, String mid, boolean[] used,
            StringBuilder sb, List<String> res) {
        if (sb.length() == list.size()) {
            res.add(sb.toString() + mid + sb.reverse().toString());
            sb.reverse();
            return;
        }
        for (int i = 0; i < list.size(); i++) {
            if (i > 0 && list.get(i) == list.get(i - 1) && !used[i - 1])
                continue;
            if (!used[i]) {
                used[i] = true;
                sb.append(list.get(i));
                getPerm(list, mid, used, sb, res);
                used[i] = false;
                sb.deleteCharAt(sb.length() - 1);
            }
        }
    }
}
```

268.Missing Number

Given an array containing *n* distinct numbers taken from `0, 1, 2, ..., n`, find the one that is missing from the array.
For example,
Given *nums* = `[0, 1, 3]` return `2`.
Note:
Your algorithm should run in linear runtime complexity. Could you implement it using only constant extra space complexity?

```
public class Solution {
    public int missingNumber(int[] nums) {
        int xor = 0, i = 0;
        for (i = 0; i < nums.length; i++)
            xor = xor ^ i ^ nums[i];
        return xor ^ i;
    }
}
```

269.Alien Dictionary

There is a new alien language which uses the latin alphabet. However, the order among letters are unknown to you. You receive a list of **non-empty** words from the dictionary, where **words are sorted lexicographically by the rules of this new language**. Derive the order of letters in this language.
Example 1:
Given the following words in dictionary,

```
[ "wrt",
  "wrf",
  "er",
  "ett",
  "rftt"]
```

The correct order is: `"wertf"`

Example 2:
Given the following words in dictionary,

```
[ "z",
  "x"]
```

The correct order is: `"zx"`.
Example 3:
Given the following words in dictionary,

```
[ "z",
  "x",
  "z"]
```

The order is invalid, so return `""`.
Note:

1. You may assume all letters are in lowercase.
2. You may assume that if a is a prefix of b, then a must appear before b in the given dictionary.
3. If the order is invalid, return an empty string.
4. There may be multiple valid order of letters, return any one of them is fine.

```
public class Solution {
    public String alienOrder(String[] words) {
        Map<Character, Set<Character>> map = new HashMap<Character, Set<Character>>();
        Map<Character, Integer> degree = new HashMap<Character, Integer>();
        String result = "";
        if (words == null || words.length == 0)
            return result;
        for (String s : words) {
            for (char c : s.toCharArray())
                degree.put(c, 0);
        }
        for (int i = 0; i < words.length - 1; i++) {
            String cur = words[i];
            String next = words[i + 1];
            int length = Math.min(cur.length(), next.length());
            for (int j = 0; j < length; j++) {
                char c1 = cur.charAt(j);
                char c2 = next.charAt(j);
                if (c1 != c2) {
                    Set<Character> set = new HashSet<Character>();
                    if (map.containsKey(c1))
                        set = map.get(c1);
                    if (!set.contains(c2)) {
                        set.add(c2);
                        map.put(c1, set);
                        degree.put(c2, degree.get(c2) + 1);
                    }
                    break;
                }
            }
        }
        Queue<Character> q = new LinkedList<Character>();
        for (char c : degree.keySet()) {
            if (degree.get(c) == 0)
                q.add(c);
        }
        while (!q.isEmpty()) {
            char c = q.remove();
            result += c;
            if (map.containsKey(c)) {
                for (char c2 : map.get(c)) {
                    degree.put(c2, degree.get(c2) - 1);
                    if (degree.get(c2) == 0)
                        q.add(c2);
                }
            }
        }
        if (result.length() != degree.size())
            return "";
        return result;
    }
}
```

270.Closest Binary Search Tree Value

Given a non-empty binary search tree and a target value, find the value in the BST that is closest to the target.
Note:

- Given target value is a floating point.

```
public class Solution {
    public int closestValue(TreeNode root, double target) {
        int a = root.val;
        TreeNode kid = target < a ? root.left : root.right;
        if (kid == null)
            return a;
        int b = closestValue(kid, target);
        return Math.abs(a - target) < Math.abs(b - target) ? a : b;
    }
}
```

271.Encode and Decode Strings

Design an algorithm to encode **a list of strings** to **a string**. The encoded string is then sent over the network and is decoded back to the original list of strings.
Machine 1 (sender) has the function:

```
string encode(vector<string> strs) {
  // ... your code
  return encoded_string;
}
```

Machine 2 (receiver) has the function:

```
vector<string> decode(string s) {
  //... your code
  return strs;
}
```

So Machine 1 does:

```
string encoded_string = encode(strs);
```

and Machine 2 does:

```
vector<string> strs2 = decode(encoded_string);
```

`strs2` in Machine 2 should be the same as `strs` in Machine 1.
Implement the `encode` and `decode` methods.
Note:

- The string may contain any possible characters out of 256 valid ascii characters. Your algorithm should be generalized enough to work on any possible characters.
- Do not use class member/global/static variables to store states. Your encode and decode algorithms should be stateless.
- Do not rely on any library method such as `eval` or serialize methods. You should implement your own encode/decode algorithm.

```
public class Codec {
        public String encode(List<String> strs) {
                StringBuilder sb = new StringBuilder();
                for (String s : strs)
                        sb.append(s.length()).append('/').append(s);
                return sb.toString();
        }
        public List<String> decode(String s) {
                List<String> ret = new ArrayList<String>();
                int i = 0;
                while (i < s.length()) {
                        int slash = s.indexOf('/', i);
                        int size = Integer.valueOf(s.substring(i, slash));
                        ret.add(s.substring(slash + 1, slash + size + 1));
                        i = slash + size + 1;
                }
                return ret;
        }
}
```

272.Closest Binary Search Tree Value II

Given a non-empty binary search tree and a target value, find *k* values in the BST that are closest to the target.
Note:

- Given target value is a floating point.
- You may assume *k* is always valid, that is: *k* ≤ total nodes.
- You are guaranteed to have only one unique set of *k* values in the BST that are closest to the target.

Follow up:
Assume that the BST is balanced, could you solve it in less than *O*(*n*) runtime (where *n* = total nodes)?

```
public class Solution {
    public List<Integer> closestKValues(TreeNode root, double target, int k) {
        List<Integer> res = new ArrayList<>();
        Stack<Integer> preStack = new Stack<>();
        Stack<Integer> sucStack = new Stack<>();
```

```
        inorder(root, target, true, sucStack);
        while (k-- > 0) {
            if (preStack.isEmpty())
                res.add(sucStack.pop());
            else if (sucStack.isEmpty())
                res.add(preStack.pop());
            else if (Math.abs(preStack.peek() - target) < Math
                    .abs(sucStack.peek() - target))
                res.add(preStack.pop());
            else
                res.add(sucStack.pop());
        }
        return res;
    }
    void inorder(TreeNode root, double target, boolean reverse,
            Stack<Integer> stack) {
        if (root == null)
            return;
        inorder(reverse ? root.right : root.left, target, reverse, stack);
        if ((reverse && root.val <= target) || (!reverse && root.val > target))
            return;
        stack.push(root.val);
        inorder(reverse ? root.left : root.right, target, reverse, stack);
    }
}
```

273.Integer to English Words

Convert a non-negative integer to its english words representation. Given input is guaranteed to be less than $2^{31} - 1$.
For example,

```
123 -> "One Hundred Twenty Three"
12345 -> "Twelve Thousand Three Hundred Forty Five"
1234567 -> "One Million Two Hundred Thirty Four Thousand Five Hundred Sixty Seven"
```

```
public class Solution {
    private final String[] LESS_THAN_20 = { "", "One", "Two", "Three", "Four",
            "Five", "Six", "Seven", "Eight", "Nine", "Ten", "Eleven", "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";
        int i = 0;
        String words = "";
        while (num > 0) {
            if (num % 1000 != 0)
                words = helper(num % 1000) + THOUSANDS[i] + " " + words;
            num /= 1000;
            i++;
        }
        return words.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);
    }
}
```

274.H-Index

Given an array of citations (each citation is a non-negative integer) of a researcher, write a function to compute the researcher's h-index.
According to the definition of h-index on Wikipedia: "A scientist has index *h* if *h* of his/her *N* papers have **at least** *h* citations each, and the other *N* – *h* papers have **no more than** *h* citations each."
For example, given `citations = [3, 0, 6, 1, 5]`, which means the researcher has `5` papers in total and each of them had received `3, 0, 6, 1, 5` citations respectively. Since the researcher has `3` papers with **at least** `3` citations each and the remaining two with **no more than** `3` citations each, his h-index is `3`.
Note: If there are several possible values for `h`, the maximum one is taken as the h-index.

```
public class Solution {
    public int hIndex(int[] citations) {
```

```
        int[] papers = new int[n + 1];
        for (int c : citations)
            papers[Math.min(n, c)]++;
        int k = n;
        for (int s = papers[n]; k > s; s += papers[k])
            k--;
        return k;
    }
}
```

275.H-Index II

Follow up for H-Index: What if the `citations` array is sorted in ascending order? Could you optimize your algorithm?

```
public class Solution {
    public int hIndex(int[] citations) {
        if (citations == null || citations.length == 0)
            return 0;
        int l = 0, r = citations.length;
        int n = citations.length;
        while (l < r) {
            int mid = l + (r - l) / 2;
            if (citations[mid] == n - mid)
                return n - mid;
            if (citations[mid] < citations.length - mid)
                l = mid + 1;
            else
                r = mid;
        }
        return n - l;
    }
}
```

276.Paint Fence

There is a fence with n posts, each post can be painted with one of the k colors.
You have to paint all the posts such that no more than two adjacent fence posts have the same color.
Return the total number of ways you can paint the fence.
Note:
n and k are non-negative integers.

```
public class Solution {
    public int numWays(int n, int k) {
        if (n == 0)
            return 0;
        else if (n == 1)
            return k;
        int diffColorCounts = k * (k - 1);
        int sameColorCounts = k;
        for (int i = 2; i < n; i++) {
            int temp = diffColorCounts;
            diffColorCounts = (diffColorCounts + sameColorCounts) * (k - 1);
            sameColorCounts = temp;
        }
        return diffColorCounts + sameColorCounts;
    }
}
```

277.Find the Celebrity

Suppose you are at a party with `n` people (labeled from `0` to `n - 1`) and among them, there may exist one celebrity. The definition of a celebrity is that all the other `n - 1` people know him/her but he/she does not know any of them.
Now you want to find out who the celebrity is or verify that there is not one. The only thing you are allowed to do is to ask questions like: "Hi, A. Do you know B?" to get information of whether A knows B. You need to find out the celebrity (or verify there is not one) by asking as few questions as possible (in the asymptotic sense).
You are given a helper function `bool knows(a, b)` which tells you whether A knows B. Implement a function `int findCelebrity(n)`, your function should minimize the number of calls to `knows`.
Note: There will be exactly one celebrity if he/she is in the party. Return the celebrity's label if there is a celebrity in the party. If there is no celebrity, return `-1`.

```
public class Solution extends Relation {
    public int findCelebrity(int n) {
        int candidate = 0;
        for (int i = 1; i < n; i++) {
            if (knows(candidate, i))
                candidate = i;
        }
        for (int i = 0; i < n; i++) {
            if (i != candidate && (knows(candidate, i) || !knows(i, candidate)))
                return -1;
        }
        return candidate;
    }
}
```

278.First Bad Version

You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

Suppose you have `n` versions `[1, 2, ..., n]` and you want to find out the first bad one, which causes all the following ones to be bad.

You are given an API `bool isBadVersion(version)` which will return whether `version` is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

```
public class Solution extends VersionControl {
    public int firstBadVersion(int n) {
        int left = 1;
        int right = n;
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (isBadVersion(mid)) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left;
    }
}
```

279.Perfect Squares

Given a positive integer *n*, find the least number of perfect square numbers (for example, `1, 4, 9, 16, ...`) which sum to *n*.

For example, given *n* = `12`, return `3` because `12 = 4 + 4 + 4`; given *n* = `13`, return `2` because `13 = 4 + 9`.

```
public class Solution {
    public int numSquares(int n) {
        int[] dp = new int[n + 1];
        Arrays.fill(dp, Integer.MAX_VALUE);
        dp[0] = 0;
        for (int i = 1; i <= n; ++i) {
            int min = Integer.MAX_VALUE;
            int j = 1;
            while (i - j * j >= 0) {
                min = Math.min(min, dp[i - j * j] + 1);
                ++j;
            }
            dp[i] = min;
        }
        return dp[n];
    }
}
```

280.Wiggle Sort

Given an unsorted array `nums`, reorder it **in-place** such that `nums[0] <= nums[1] >= nums[2] <= nums[3]...`.

For example, given `nums = [3, 5, 2, 1, 6, 4]`, one possible answer is `[1, 6, 2, 5, 3, 4]`.

```
public class Solution {
    public void wiggleSort(int[] nums) {
        for (int i = 0; i < nums.length - 1; i++)
            if ((i % 2 == 0) == (nums[i] > nums[i + 1]))
                swap(nums, i, i + 1);
    }
    private static void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
}
```

281.Zigzag Iterator

Given two 1d vectors, implement an iterator to return their elements alternately.

For example, given two 1d vectors:

```
v1 = [1, 2]
v2 = [3, 4, 5, 6]
```

By calling *next* repeatedly until *hasNext* returns `false`, the order of elements returned by *next* should be: `[1, 3, 2, 4, 5, 6]`.

Follow up: What if you are given `k` 1d vectors? How well can your code be extended to such cases?

Clarification for the follow up question - Update (2015-09-18):

The "Zigzag" order is not clearly defined and is ambiguous for `k > 2` cases. If "Zigzag" does not look right to you, replace "Zigzag" with "Cyclic". For example, given the following input:

```
[1,2,3]
[4,5,6,7]
[8,9]
```

```
public class ZigzagIterator implements Iterator<Integer> {
    LinkedList<Iterator<Integer>> list;
    public ZigzagIterator(List<Integer> v1, List<Integer> v2) {
        list = new LinkedList<Iterator<Integer>>();
        if (!v1.isEmpty())
            list.add(v1.iterator());
        if (!v2.isEmpty())
            list.add(v2.iterator());
    }
    @Override
    public Integer next() {
        Iterator<Integer> poll = list.remove();
        int result = (Integer) poll.next();
        if (poll.hasNext())
            list.add(poll);
        return result;
    }
    @Override
    public boolean hasNext() {
        return !list.isEmpty();
    }
}
```

282.Expression Add Operators

Given a string that contains only digits `0-9` and a target value, return all possibilities to add **binary** operators (not unary) `+`, `-`, or `*`between the digits so they evaluate to the target value.

Examples:

```
"123", 6 -> ["1+2+3", "1*2*3"]
"232", 8 -> ["2*3+2", "2+3*2"]
"105", 5 -> ["1*0+5","10-5"]
"00", 0 -> ["0+0", "0-0", "0*0"]
"3456237490", 9191 -> []
```

```
public class Solution {
    public List<String> addOperators(String num, int target) {
        List<String> rst = new ArrayList<String>();
        if (num == null || num.length() == 0)
            return rst;
        helper(rst, "", num, target, 0, 0, 0);
        return rst;
    }
    public void helper(List<String> rst, String path, String num, int target,
            int pos, long eval, long multed) {
        if (pos == num.length()) {
            if (target == eval)
                rst.add(path);
            return;
        }
        for (int i = pos; i < num.length(); i++) {
            if (i != pos && num.charAt(pos) == '0')
                break;
            long cur = Long.parseLong(num.substring(pos, i + 1));
            if (pos == 0) {
                helper(rst, path + cur, num, target, i + 1, cur, cur);
            } else {
                helper(rst, path + "+" + cur, num, target, i + 1, eval + cur,
                        cur);
                helper(rst, path + "-" + cur, num, target, i + 1, eval - cur,
                        -cur);
                helper(rst, path + "*" + cur, num, target, i + 1,
                        eval - multed + multed * cur, multed * cur);
            }
        }
    }
}
```

283.Move Zeroes

Given an array `nums`, write a function to move all `0`'s to the end of it while maintaining the relative order of the non-zero elements.
For example, given `nums = [0, 1, 0, 3, 12]`, after calling your function, `nums` should be `[1, 3, 12, 0, 0]`.

Note:

1. You must do this **in-place** without making a copy of the array.
2. Minimize the total number of operations.

```
public class Solution {
    public void moveZeroes(int[] nums) {
        int temp;
        for (int lastNonZeroFoundAt = 0, cur = 0; cur < nums.length; cur++) {
            if (nums[cur] != 0) {
```

```
                nums[lastNonZeroFoundAt++] = nums[cur];
                nums[cur] = temp;
            }
        }
    }
}
```

284.Peeking Iterator

Given an Iterator class interface with methods: `next()` and `hasNext()`, design and implement a PeekingIterator that support the `peek()` operation -- it essentially peek() at the element that will be returned by the next call to next().

Here is an example. Assume that the iterator is initialized to the beginning of the list: `[1, 2, 3]`.
Call `next()` gets you 1, the first element in the list.
Now you call `peek()` and it returns 2, the next element. Calling `next()` after that ***still*** return 2.
You call `next()` the final time and it returns 3, the last element. Calling `hasNext()` after that should return false.
Follow up: How would you extend your design to be generic and work with all types, not just integer?

```
class PeekingIterator implements Iterator<Integer> {
    private Integer next = null;
    private Iterator<Integer> iter;
    public PeekingIterator(Iterator<Integer> iterator) {
        iter = iterator;
        if (iter.hasNext())
            next = iter.next();
    }
    public Integer peek() {
        return next;
    }
    @Override
    public Integer next() {
        Integer res = next;
        next = iter.hasNext() ? iter.next() : null;
        return res;
    }
    @Override
    public boolean hasNext() {
        return next != null;
    }
}
```

285.Inorder Successor in BST

Given a binary search tree and a node in it, find the in-order successor of that node in the BST.
Note: If the given node has no in-order successor in the tree, return `null`.

```
public class Solution {
    public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {
        if (root == null)
            return null;
        if (root.val <= p.val) {
            return inorderSuccessor(root.right, p);
        } else {
            TreeNode left = inorderSuccessor(root.left, p);
            return (left != null) ? left : root;
        }
    }
}
```

286.Walls and Gates

You are given a *m x n* 2D grid initialized with these three possible values.

1. `-1` - A wall or an obstacle.
2. `0` - A gate.
3. `INF` - Infinity means an empty room. We use the value `2^31 - 1 = 2147483647` to represent `INF` as you may assume that the distance to a gate is less than `2147483647`.

Fill each empty room with the distance to its *nearest* gate. If it is impossible to reach a gate, it should be filled with `INF`.
For example, given the 2D grid:

```
INF  -1  0  INF
INF INF INF  -1
INF  -1 INF  -1
  0  -1 INF INF
```

After running your function, the 2D grid should be:

```
  3  -1   0   1
  2   2   1  -1
  1  -1   2  -1
  0  -1   3   4
```

```
public class Solution {
    public void wallsAndGates(int[][] rooms) {
```

```
            return;
        Queue<int[]> queue = new LinkedList<>();
        for (int i = 0; i < rooms.length; i++) {
            for (int j = 0; j < rooms[0].length; j++) {
                if (rooms[i][j] == 0)
                    queue.add(new int[] { i, j });
            }
        }
        while (!queue.isEmpty()) {
            int[] top = queue.remove();
            int row = top[0], col = top[1];
            if (row > 0 && rooms[row - 1][col] == Integer.MAX_VALUE) {
                rooms[row - 1][col] = rooms[row][col] + 1;
                queue.add(new int[] { row - 1, col });
            }
            if (row < rooms.length - 1
                    && rooms[row + 1][col] == Integer.MAX_VALUE) {
                rooms[row + 1][col] = rooms[row][col] + 1;
                queue.add(new int[] { row + 1, col });
            }
            if (col > 0 && rooms[row][col - 1] == Integer.MAX_VALUE) {
                rooms[row][col - 1] = rooms[row][col] + 1;
                queue.add(new int[] { row, col - 1 });
            }
            if (col < rooms[0].length - 1
                    && rooms[row][col + 1] == Integer.MAX_VALUE) {
                rooms[row][col + 1] = rooms[row][col] + 1;
                queue.add(new int[] { row, col + 1 });
            }
        }
    }
}
```

287.Find the Duplicate Number

Given an array *nums* containing *n* + 1 integers where each integer is between 1 and *n* (inclusive), prove that at least one duplicate number must exist. Assume that there is only one duplicate number, find the duplicate one.

Note:

1. You **must not** modify the array (assume the array is read only).
2. You must use only constant, *O*(1) extra space.
3. Your runtime complexity should be less than $O(n^2)$.
4. There is only one duplicate number in the array, but it could be repeated more than once.

```
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;
    }
}
```

288.Unique Word Abbreviation

An abbreviation of a word follows the form <first letter><number><last letter>. Below are some examples of word abbreviations:

```
a) it                      --> it    (no abbreviation)
     1
b) d|o|g                   --> d1g
              1    1  1
     1---5----0----5--8
c) i|nternationalizatio|n  --> i18n
              1
     1---5----0
d) l|ocalizatio|n          --> l10n
```

Assume you have a dictionary and given a word, find whether its abbreviation is unique in the dictionary. A word's abbreviation is unique if no *other* word from the dictionary has the same abbreviation.

Example:

```
isUnique("dear") -> false
isUnique("cart") -> true
isUnique("cane") -> false
isUnique("make") -> true
```

```
public class ValidWordAbbr {
    private final Map<String, Boolean> abbrDict = new HashMap<>();
    private final Set<String> dict;
    public ValidWordAbbr(String[] dictionary) {
        dict = new HashSet<>(Arrays.asList(dictionary));
        for (String s : dict) {
            String abbr = toAbbr(s);
            abbrDict.put(abbr, !abbrDict.containsKey(abbr));
        }
    }
    public boolean isUnique(String word) {
        String abbr = toAbbr(word);
        Boolean hasAbbr = abbrDict.get(abbr);
        return hasAbbr == null || (hasAbbr && dict.contains(word));
    }
    private String toAbbr(String s) {
        int n = s.length();
        if (n <= 2) {
            return s;
        }
        return s.charAt(0) + Integer.toString(n - 2) + s.charAt(n - 1);
    }
}
```

289.Game of Life

According to the Wikipedia's article: "The **Game of Life**, also known simply as **Life**, is a cellular automaton devised by the British mathematician John Horton Conway in 1970."
Given a *board* with *m* by *n* cells, each cell has an initial state *live* (1) or *dead* (0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules (taken from the above Wikipedia article):

1. Any live cell with fewer than two live neighbors dies, as if caused by under-population.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by over-population..
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

Write a function to compute the next state (after one update) of the board given its current state.
Follow up:

1. Could you solve it in-place? Remember that the board needs to be updated at the same time: You cannot update some cells first and then use their updated values to update other cells.
2. In this question, we represent the board using a 2D array. In principle, the board is infinite, which would cause problems when the active area encroaches the border of the array. How would you address these problems?

```
public class Solution {
    public void gameOfLife(int[][] board) {
        if (board == null || board.length == 0)
            return;
        int m = board.length, n = board[0].length;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                int lives = liveNeighbors(board, m, n, i, j);
                if (board[i][j] == 1 && lives >= 2 && lives <= 3) {
                    board[i][j] = 3;
                }
                if (board[i][j] == 0 && lives == 3) {
                    board[i][j] = 2;
                }
            }
        }
        for (int i = 0; i < m; i++)
            for (int j = 0; j < n; j++)
                board[i][j] >>= 1;
    }
    public int liveNeighbors(int[][] board, int m, int n, int i, int j) {
        int lives = 0;
        for (int x = Math.max(i - 1, 0); x <= Math.min(i + 1, m - 1); x++)
            for (int y = Math.max(j - 1, 0); y <= Math.min(j + 1, n - 1); y++)
                lives += board[x][y] & 1;
        lives -= board[i][j] & 1;
        return lives;
    }
}
```

290.Word Pattern

Given a `pattern` and a string `str`, find if `str` follows the same pattern.
Here **follow** means a full match, such that there is a bijection between a letter in `pattern` and a **non-empty** word in `str`.

Examples:

1. pattern = `"abba"`, str = `"dog cat cat dog"` should return true.
2. pattern = `"abba"`, str = `"dog cat cat fish"` should return false.
3. pattern = `"aaaa"`, str = `"dog cat cat dog"` should return false.
4. pattern = `"abba"`, str = `"dog dog dog dog"` should return false.

Notes:

You may assume `pattern` contains only lowercase letters, and `str` contains lowercase letters separated by a single space.

```
public class Solution {
    public boolean wordPattern(String pattern, String str) {
        String[] words = str.split(" ");
        if (words.length != pattern.length())
            return false;
        Map<Object, Integer> index = new HashMap<Object, Integer>();
        for (Integer i = 0; i < words.length; ++i)
            if (index.put(pattern.charAt(i), i) != index.put(words[i], i))
                return false;
        return true;
    }
}
```

291.Word Pattern II

Given a `pattern` and a string `str`, find if `str` follows the same pattern.

Here **follow** means a full match, such that there is a bijection between a letter in `pattern` and a **non-empty** substring in `str`.

Examples:

1. pattern = `"abab"`, str = `"redblueredblue"` should return true.
2. pattern = `"aaaa"`, str = `"asdasdasdasd"` should return true.
3. pattern = `"aabb"`, str = `"xyzabcxzyabc"` should return false.

Notes:

You may assume both `pattern` and `str` contains only lowercase letters.

```
public class Solution {
    public boolean wordPatternMatch(String pattern, String str) {
        Map<Character, String> map = new HashMap<>();
        Set<String> set = new HashSet<>();
        return isMatch(str, 0, pattern, 0, map, set);
    }
    boolean isMatch(String str, int i, String pat, int j,
            Map<Character, String> map, Set<String> set) {
        if (i == str.length() && j == pat.length())
            return true;
        if (i == str.length() || j == pat.length())
            return false;
        char c = pat.charAt(j);
        if (map.containsKey(c)) {
            String s = map.get(c);
            if (!str.startsWith(s, i))
                return false;
            return isMatch(str, i + s.length(), pat, j + 1, map, set);
        }
        for (int k = i; k < str.length(); k++) {
            String p = str.substring(i, k + 1);
            if (set.contains(p))
                continue;
            map.put(c, p);
            set.add(p);
            if (isMatch(str, k + 1, pat, j + 1, map, set))
                return true;
            map.remove(c);
            set.remove(p);
        }
        return false;
    }
}
```

292.Nim Game

You are playing the following Nim Game with your friend: There is a heap of stones on the table, each time one of you take turns to remove 1 to 3 stones. The one who removes the last stone will be the winner. You will take the first turn to remove the stones.

Both of you are very clever and have optimal strategies for the game. Write a function to determine whether you can win the game given the number of stones in the heap.

For example, if there are 4 stones in the heap, then you will never win the game: no matter 1, 2, or 3 stones you remove, the last stone will always be removed by your friend.

```
public class Solution {
    public boolean canWinNim(int n) {
        return (n % 4 != 0);
    }
}
```

293.Flip Game

You are playing the following Flip Game with your friend: Given a string that contains only these two characters: `+` and `-`, you and your friend take turns to flip two **consecutive** `"++"` into `"--"`. The game ends when a person can no longer make a move and therefore the other person will be the winner.

Write a function to compute all possible states of the string after one valid move.

For example, given `s = "++++"`, after one move, it may become one of the following states:

```
[ "--++",
  "+--+",
  "++--"]
```

If there is no valid move, return an empty list `[]`.

```
public class Solution {
    public List<String> generatePossibleNextMoves(String s) {
        List<String> list = new ArrayList<>();
        for (int i = -1; (i = s.indexOf("++", i + 1)) >= 0;)
            list.add(s.substring(0, i) + "--" + s.substring(i + 2));
        return list;
    }
}
```

294.Flip Game II

You are playing the following Flip Game with your friend: Given a string that contains only these two characters: `+` and `-`, you and your friend take turns to flip two **consecutive** `"++"` into `"--"`. The game ends when a person can no longer make a move and therefore the other person will be the winner.

Write a function to determine if the starting player can guarantee a win.

For example, given `s = "++++"`, return true. The starting player can guarantee a win by flipping the middle `"++"` to become `"+--+"`.

Follow up:

Derive your algorithm's runtime complexity.

```
public class Solution {
    public boolean canWin(String s) {
        if (s == null || s.length() < 2)
            return false;
        Map<String, Boolean> map = new HashMap<>();
        return canWin(s, map);
    }
    public boolean canWin(String s, Map<String, Boolean> map) {
        if (map.containsKey(s))
            return map.get(s);
        for (int i = 0; i < s.length() - 1; i++) {
            if (s.charAt(i) == '+' && s.charAt(i + 1) == '+') {
                String opponent = s.substring(0, i) + "--" + s.substring(i + 2);
                if (!canWin(opponent, map)) {
                    map.put(s, true);
                    return true;
                }
            }
        }
        map.put(s, false);
        return false;
    }
}
```

295.Find Median from Data Stream

Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.

Examples:

`[2,3,4]` , the median is `3`

`[2,3]`, the median is `(2 + 3) / 2 = 2.5`

Design a data structure that supports the following two operations:

- void addNum(int num) - Add a integer number from the data stream to the data structure.
- double findMedian() - Return the median of all elements so far.

For example:

```
addNum(1)
addNum(2)
findMedian() -> 1.5
addNum(3)
findMedian() -> 2
```

```
public class MedianFinder {
    private Queue<Long> small = new PriorityQueue<>(),
            large = new PriorityQueue<>();
    public void addNum(int num) {
        large.add((long) num);
        small.add(-large.poll());
        if (large.size() < small.size())
```

```
            large.add(-small.poll());
    }
    public double findMedian() {
        return large.size() > small.size() ? large.peek()
                : (large.peek() - small.peek()) / 2.0;
    }
}
```

296.Best Meeting Point

A group of two or more people wants to meet and minimize the total travel distance. You are given a 2D grid of values 0 or 1, where each 1 marks the home of someone in the group. The distance is calculated using Manhattan Distance, where distance(p1, p2) = `|p2.x - p1.x| + |p2.y - p1.y|`.
For example, given three people living at `(0,0)`, `(0,4)`, and `(2,2)`:

```
1 - 0 - 0 - 0 - 1
|   |   |   |   |
0 - 0 - 0 - 0 - 0
|   |   |   |   |
0 - 0 - 1 - 0 - 0
```

The point `(0,2)` is an ideal meeting point, as the total travel distance of 2+2+2=6 is minimal. So return 6.

```
public class Solution {
    public int minTotalDistance(int[][] grid) {
        List<Integer> rows = collectRows(grid);
        List<Integer> cols = collectCols(grid);
        return minDistance1D(rows) + minDistance1D(cols);
    }
    private int minDistance1D(List<Integer> points) {
        int distance = 0;
        int i = 0;
        int j = points.size() - 1;
        while (i < j) {
            distance += points.get(j) - points.get(i);
            i++;
            j--;
        }
        return distance;
    }
    private List<Integer> collectRows(int[][] grid) {
        List<Integer> rows = new ArrayList<>();
        for (int row = 0; row < grid.length; row++)
            for (int col = 0; col < grid[0].length; col++)
                if (grid[row][col] == 1)
                    rows.add(row);
        return rows;
    }
    private List<Integer> collectCols(int[][] grid) {
        List<Integer> cols = new ArrayList<>();
        for (int col = 0; col < grid[0].length; col++)
            for (int row = 0; row < grid.length; row++)
                if (grid[row][col] == 1)
                    cols.add(col);
        return cols;
    }
}
```

297.Serialize and Deserialize Binary Tree

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.
Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.
For example, you may serialize the following tree

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

as `"[1,2,3,null,null,4,5]"`, just the same as how LeetCode OJ serializes a binary tree. You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.
Note: Do not use class member/global/static variables to store states. Your serialize and deserialize algorithms should be stateless.

```
public class Codec {
    public String serialize(TreeNode root) {
        StringBuilder sb = new StringBuilder();
        serialize(root, sb);
        return sb.toString().trim();
```

```
    private static void serialize(TreeNode root, StringBuilder sb) {
        if (root == null) {
            sb.append("# ");
            return;
        }
        sb.append(root.val);
        sb.append(" ");
        serialize(root.left, sb);
        serialize(root.right, sb);
    }
    public TreeNode deserialize(String data) {
        String[] vals = data.split(" ");
        return deserialize(new ValHolder(vals, -1));
    }
    class ValHolder {
        String[] vals;
        int i;
        int n;
        public ValHolder(String[] vals, int i) {
            this.vals = vals;
            this.i = i;
            n = vals.length;
        }
    }
    private static TreeNode deserialize(ValHolder vh) {
        String cur = vh.vals[++vh.i];
        if ("#".equals(cur))
            return null;
        TreeNode root = new TreeNode(Integer.valueOf(cur));
        if (vh.i < vh.n - 1)
            root.left = deserialize(vh);
        if (vh.i < vh.n - 1)
            root.right = deserialize(vh);
        return root;
    }
}
```

298.Binary Tree Longest Consecutive Sequence

Given a binary tree, find the length of the longest consecutive sequence path.
The path refers to any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The longest consecutive path need to be from parent to child (cannot be the reverse).
For example,

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

Longest consecutive sequence path is `3-4-5`, so return `3`.

```
   2
    \
     3
    /
   2
  /
 1
```

Longest consecutive sequence path is `2-3`,not`3-2-1`, so return `2`.

```
public class Solution {
    public int longestConsecutive(TreeNode root) {
        return dfs(root, null, 0);
    }
    private int dfs(TreeNode p, TreeNode parent, int length) {
        if (p == null)
            return length;
        length = (parent != null && p.val == parent.val + 1) ? length + 1 : 1;
        return Math.max(length,
                Math.max(dfs(p.left, p, length), dfs(p.right, p, length)));
    }
}
```

299.Bulls and Cows

You are playing the following Bulls and Cows game with your friend: You write down a number and ask your friend to guess what the number is. Each time your friend makes a guess, you provide a hint that indicates how many digits in said guess match your

secret number exactly in both digit and position (called "bulls") and how many digits match the secret number but locate in the wrong position (called "cows"). Your friend will use successive guesses and hints to eventually derive the secret number.
For example:

```
Secret number:  "1807"
Friend's guess: "7810"
```

Hint: `1` bull and `3` cows. (The bull is `8`, the cows are `0`, `1` and `7`.)
Write a function to return a hint according to the secret number and friend's guess, use `A` to indicate the bulls and `B` to indicate the cows. In the above example, your function should return `"1A3B"`.
Please note that both secret number and friend's guess may contain duplicate digits, for example:

```
Secret number:  "1123"
Friend's guess: "0111"
```

In this case, the 1st `1` in friend's guess is a bull, the 2nd or 3rd `1` is a cow, and your function should return `"1A1B"`.
You may assume that the secret number and your friend's guess only contain digits, and their lengths are always equal.

```
public class Solution {
    public String getHint(String secret, String guess) {
        int bulls = 0;
        int cows = 0;
        int[] numbers = new int[10];
        for (int i = 0; i < secret.length(); i++) {
            if (secret.charAt(i) == guess.charAt(i))
                bulls++;
            else {
                if (numbers[secret.charAt(i) - '0']++ < 0)
                    cows++;
                if (numbers[guess.charAt(i) - '0']-- > 0)
                    cows++;
            }
        }
        return bulls + "A" + cows + "B";
    }
}
```

300.Longest Increasing 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(n^2) complexity.
Follow up: Could you improve it to O(n log n) time complexity?

```
public class Solution {
    public int lengthOfLIS(int[] nums) {
        int[] dp = new int[nums.length];
        int len = 0;
        for (int num : nums) {
            int i = Arrays.binarySearch(dp, 0, len, num);
            if (i < 0)
                i = -(i + 1);
            dp[i] = num;
            if (i == len)
                len++;
        }
        return len;
    }
}
```

301.Remove Invalid Parentheses

Remove the minimum number of invalid parentheses in order to make the input string valid. Return all possible results.
Note: The input string may contain letters other than the parentheses `(` and `)`.
Examples:

```
"()())()" -> ["()()()", "(())()"]
"(a)())()" -> ["(a)()()", "(a())()"]
")(" -> [""]
```

```
public class Solution {
    public List<String> removeInvalidParentheses(String s) {
        List<String> ans = new ArrayList<>();
        remove(s, ans, 0, 0, new char[] { '(', ')' });
        return ans;
    }
    public void remove(String s, List<String> ans, int last_i, int last_j,
            char[] par) {
        for (int stack = 0, i = last_i; i < s.length(); ++i) {
```

```
                stack++;
            if (s.charAt(i) == par[1])
                stack--;
            if (stack >= 0)
                continue;
            for (int j = last_j; j <= i; ++j)
                if (s.charAt(j) == par[1]
                        && (j == last_j || s.charAt(j - 1) != par[1]))
                    remove(s.substring(0, j) + s.substring(j + 1, s.length()),
                            ans, i, j, par);
            return;
        }
        String reversed = new StringBuilder(s).reverse().toString();
        if (par[0] == '(')
            remove(reversed, ans, 0, 0, new char[] { ')', '(' });
        else
            ans.add(reversed);
    }
}
```

302.Smallest Rectangle Enclosing Black Pixels

An image is represented by a binary matrix with `0` as a white pixel and `1` as a black pixel. The black pixels are connected, i.e., there is only one black region. Pixels are connected horizontally and vertically. Given the location `(x, y)` of one of the black pixels, return the area of the smallest (axis-aligned) rectangle that encloses all black pixels.
For example, given the following image:

```
[ "0010",
  "0110",
  "0100"]
```

and `x = 0`, `y = 2`,
Return `6`.

```
public class Solution {
    public int minArea(char[][] image, int x, int y) {
        int m = image.length, n = image[0].length;
        int left = searchColumns(image, 0, y, 0, m, true);
        int right = searchColumns(image, y + 1, n, 0, m, false);
        int top = searchRows(image, 0, x, left, right, true);
        int bottom = searchRows(image, x + 1, m, left, right, false);
        return (right - left) * (bottom - top);
    }
    private int searchColumns(char[][] image, int i, int j, int top, int bottom,
            boolean whiteToBlack) {
        while (i != j) {
            int k = top, mid = (i + j) / 2;
            while (k < bottom && image[k][mid] == '0')
                ++k;
            if (k < bottom == whiteToBlack)
                j = mid;
            else
                i = mid + 1;
        }
        return i;
    }
    private int searchRows(char[][] image, int i, int j, int left, int right,
            boolean whiteToBlack) {
        while (i != j) {
            int k = left, mid = (i + j) / 2;
            while (k < right && image[mid][k] == '0')
                ++k;
            if (k < right == whiteToBlack)
                j = mid;
            else
                i = mid + 1;
        }
        return i;
    }
}
```

303.Range Sum Query - Immutable

Given an integer array *nums*, find the sum of the elements between indices *i* and *j* (*i* ≤ *j*), inclusive.
Example:

```
Given nums = [-2, 0, 3, -5, 2, -1]
sumRange(0, 2) -> 1
sumRange(2, 5) -> -1
sumRange(0, 5) -> -3
```

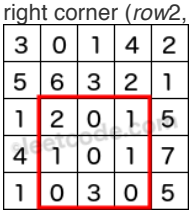
Note:

1. You may assume that the array does not change.
2. There are many calls to *sumRange* function.

```
public class NumArray {
    private int[] sum;
    public NumArray(int[] nums) {
        sum = new int[nums.length + 1];
        for (int i = 0; i < nums.length; i++) {
            sum[i + 1] = sum[i] + nums[i];
        }
    }
    public int sumRange(int i, int j) {
        return sum[j + 1] - sum[i];
    }
}
```

304.Range Sum Query 2D - Immutable

Given a 2D matrix *matrix*, find the sum of the elements inside the rectangle defined by its upper left corner (*row*1, *col*1) and lower right corner (*row2*, *col2*).



The above rectangle (with the red border) is defined by (row1, col1) = **(2, 1)** and (row2, col2) = **(4, 3)**, which contains sum = **8**.

Example:

```
Given matrix = [
  [3, 0, 1, 4, 2],
  [5, 6, 3, 2, 1],
  [1, 2, 0, 1, 5],
  [4, 1, 0, 1, 7],
  [1, 0, 3, 0, 5]]
sumRegion(2, 1, 4, 3) -> 8
sumRegion(1, 1, 2, 2) -> 11
sumRegion(1, 2, 2, 4) -> 12
```

Note:

1. You may assume that the matrix does not change.
2. There are many calls to *sumRegion* function.
3. You may assume that *row*1 ≤ *row2* and *col*1 ≤ *col2*.

```
public class NumMatrix {
    private int[][] dp;
    public NumMatrix(int[][] matrix) {
        if (matrix.length == 0 || matrix[0].length == 0)
            return;
        dp = new int[matrix.length + 1][matrix[0].length + 1];
        for (int r = 0; r < matrix.length; r++) {
            for (int c = 0; c < matrix[0].length; c++) {
                dp[r + 1][c + 1] = dp[r + 1][c] + dp[r][c + 1] + matrix[r][c]
                        - dp[r][c];
            }
        }
    }
    public int sumRegion(int row1, int col1, int row2, int col2) {
        return dp[row2 + 1][col2 + 1] - dp[row1][col2 + 1] - dp[row2 + 1][col1]
                + dp[row1][col1];
    }
}
```

305.Number of Islands II

A 2d grid map of `m` rows and `n` columns is initially filled with water. We may perform an *addLand* operation which turns the water at position (row, col) into a land. Given a list of positions to operate, **count the number of islands after each *addLand* operation**. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example:

Given `m = 3, n = 3`, `positions = [[0,0], [0,1], [1,2], [2,1]]`.
Initially, the 2d grid `grid` is filled with water. (Assume 0 represents water and 1 represents land).

```
0 0 0
0 0 0
0 0 0
```

Operation #1: addLand(0, 0) turns the water at grid[0][0] into a land.

```
1 0 0
0 0 0   Number of islands = 1
```

```
0 0 0
```

Operation #2: addLand(0, 1) turns the water at grid[0][1] into a land.

```
1 1 0
0 0 0   Number of islands = 1
0 0 0
```

Operation #3: addLand(1, 2) turns the water at grid[1][2] into a land.

```
1 1 0
0 0 1   Number of islands = 2
0 0 0
```

Operation #4: addLand(2, 1) turns the water at grid[2][1] into a land.

```
1 1 0
0 0 1   Number of islands = 3
0 1 0
```

We return the result as an array: `[1, 1, 2, 3]`

Challenge:

Can you do it in time complexity O(k log mn), where k is the length of the `positions`?

```
public class Solution {
    int[][] dirs = { { 0, 1 }, { 1, 0 }, { -1, 0 }, { 0, -1 } };
    public List<Integer> numIslands2(int m, int n, int[][] positions) {
        List<Integer> result = new ArrayList<>();
        if (m <= 0 || n <= 0)
            return result;
        int count = 0;
        int[] roots = new int[m * n];
        Arrays.fill(roots, -1);
        for (int[] p : positions) {
            int root = n * p[0] + p[1];
            roots[root] = root;
            count++;
            for (int[] dir : dirs) {
                int x = p[0] + dir[0];
                int y = p[1] + dir[1];
                int nb = n * x + y;
                if (x < 0 || x >= m || y < 0 || y >= n || roots[nb] == -1)
                    continue;
                int rootNb = findIsland(roots, nb);
                if (root != rootNb) {
                    roots[root] = rootNb;
                    root = rootNb;
                    count--;
                }
            }
            result.add(count);
        }
        return result;
    }
    public int findIsland(int[] roots, int id) {
        while (id != roots[id])
            id = roots[id];
        return id;
    }
}
```

306.Additive Number

Additive number is a string whose digits can form additive sequence.

A valid additive sequence should contain **at least** three numbers. Except for the first two numbers, each subsequent number in the sequence must be the sum of the preceding two.

For example:

`"112358"` is an additive number because the digits can form an additive sequence: `1, 1, 2, 3, 5, 8`.

```
1 + 1 = 2, 1 + 2 = 3, 2 + 3 = 5, 3 + 5 = 8
```

`"199100199"` is also an additive number, the additive sequence is: `1, 99, 100, 199`.

```
1 + 99 = 100, 99 + 100 = 199
```

Note: Numbers in the additive sequence **cannot** have leading zeros, so sequence `1, 2, 03` or `1, 02, 3` is invalid.

Given a string containing only digits `'0'-'9'`, write a function to determine if it's an additive number.

Follow up:

How would you handle overflow for very large input integers?

```
public class Solution {
```

```
    public boolean isAdditiveNumber(String num) {
        int n = num.length();
        for (int i = 1; i <= n / 2; ++i)
            for (int j = 1; Math.max(j, i) <= n - i - j; ++j)
                if (isValid(i, j, num))
                    return true;
        return false;
    }
    private boolean isValid(int i, int j, String num) {
        if (num.charAt(0) == '0' && i > 1)
            return false;
        if (num.charAt(i) == '0' && j > 1)
            return false;
        String sum;
        Long x1 = Long.parseLong(num.substring(0, i));
        Long x2 = Long.parseLong(num.substring(i, i + j));
        for (int start = i + j; start != num.length(); start += sum.length()) {
            x2 = x2 + x1;
            x1 = x2 - x1;
            sum = x2.toString();
            if (!num.startsWith(sum, start))
                return false;
        }
        return true;
    }
}
```

307.Range Sum Query - Mutable

Given an integer array *nums*, find the sum of the elements between indices *i* and *j* (*i* ≤ *j*), inclusive.
The *update(i, val)* function modifies *nums* by updating the element at index *i* to *val*.

Example:

```
Given nums = [1, 3, 5]
sumRange(0, 2) -> 9
update(1, 2)
sumRange(0, 2) -> 8
```

Note:

1. The array is only modifiable by the *update* function.
2. You may assume the number of calls to *update* and *sumRange* function is distributed evenly.

```
public class NumArray {
    int[] tree;
    int n;
    public NumArray(int[] nums) {
        if (nums.length > 0) {
            n = nums.length;
            tree = new int[n * 2];
            buildTree(nums);
        }
    }
    private void buildTree(int[] nums) {
        for (int i = n, j = 0; i < 2 * n; i++, j++)
            tree[i] = nums[j];
        for (int i = n - 1; i > 0; --i)
            tree[i] = tree[i * 2] + tree[i * 2 + 1];
    }
    public void update(int pos, int val) {
        pos += n;
        tree[pos] = val;
        while (pos > 0) {
            int left = pos;
            int right = pos;
            if (pos % 2 == 0)
                right = pos + 1;
            else
                left = pos - 1;
            tree[pos / 2] = tree[left] + tree[right];
            pos /= 2;
        }
    }
    public int sumRange(int l, int r) {
        l += n;
        r += n;
        int sum = 0;
        while (l <= r) {
            if ((l % 2) == 1) {
                sum += tree[l];
                l++;
```

```
            }
            if ((r % 2) == 0) {
                sum += tree[r];
                r--;
            }
            l /= 2;
            r /= 2;
        }
        return sum;
    }
}
public class NumArray {
    int[] nums;
    int[] BIT;
    int n;
    public NumArray(int[] nums) {
        this.nums = nums;
        n = nums.length;
        BIT = new int[n + 1];
        for (int i = 0; i < n; i++)
            init(i, nums[i]);
    }
    public void init(int i, int val) {
        i++;
        while (i <= n) {
            BIT[i] += val;
            i += (i & -i);
        }
    }
    void update(int i, int val) {
        int diff = val - nums[i];
        nums[i] = val;
        init(i, diff);
    }
    public int getSum(int i) {
        int sum = 0;
        i++;
        while (i > 0) {
            sum += BIT[i];
            i -= (i & -i);
        }
        return sum;
    }
    public int sumRange(int i, int j) {
        return getSum(j) - getSum(i - 1);
    }
}
```

308.Range Sum Query 2D - Mutable

Given a 2D matrix *matrix*, find the sum of the elements inside the rectangle defined by its upper left corner (*row*1, *col*1) and lower right corner (*row*2, *col*2).

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

The above rectangle (with the red border) is defined by (row1, col1) = **(2, 1)** and (row2, col2) = **(4, 3)**, which contains sum = **8**.

Example:

```
Given matrix = [
  [3, 0, 1, 4, 2],
  [5, 6, 3, 2, 1],
  [1, 2, 0, 1, 5],
  [4, 1, 0, 1, 7],
  [1, 0, 3, 0, 5]]
sumRegion(2, 1, 4, 3) -> 8
update(3, 2, 2)
sumRegion(2, 1, 4, 3) -> 10
```

Note:

1. The matrix is only modifiable by the *update* function.
2. You may assume the number of calls to *update* and *sumRegion* function is distributed evenly.
3. You may assume that *row*1 ≤ *row*2 and *col*1 ≤ *col*2.

```
public class NumMatrix {
    int[][] tree;
    int[][] nums;
```

```
    int m;
    int n;
    public NumMatrix(int[][] matrix) {
        if (matrix.length == 0 || matrix[0].length == 0)
            return;
        m = matrix.length;
        n = matrix[0].length;
        tree = new int[m + 1][n + 1];
        nums = new int[m][n];
        for (int i = 0; i < m; i++)
            for (int j = 0; j < n; j++)
                update(i, j, matrix[i][j]);
    }
    public void update(int row, int col, int val) {
        if (m == 0 || n == 0)
            return;
        int delta = val - nums[row][col];
        nums[row][col] = val;
        for (int i = row + 1; i <= m; i += i & (-i))
            for (int j = col + 1; j <= n; j += j & (-j))
                tree[i][j] += delta;
    }
    public int sumRegion(int row1, int col1, int row2, int col2) {
        if (m == 0 || n == 0)
            return 0;
        return sum(row2 + 1, col2 + 1) + sum(row1, col1) - sum(row1, col2 + 1)
                - sum(row2 + 1, col1);
    }
    public int sum(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 += tree[i][j];
            }
        }
        return sum;
    }
}
```

309.Best Time to Buy and Sell Stock with Cooldown

Say you have an array for which the i^{th} element is the price of a given stock on day *i*.
Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times) with the following restrictions:

- You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).
- After you sell your stock, you cannot buy stock on next day. (ie, cooldown 1 day)

Example:

```
prices = [1, 2, 3, 0, 2]
maxProfit = 3
transactions = [buy, sell, cooldown, buy, sell]
```

```
public class Solution {
    public int maxProfit(int[] prices) {
        int sell = 0, prev_sell = 0, buy = Integer.MIN_VALUE, prev_buy;
        for (int price : prices) {
            prev_buy = buy;
            buy = Math.max(prev_sell - price, prev_buy);
            prev_sell = sell;
            sell = Math.max(prev_buy + price, prev_sell);
        }
        return sell;
    }
}
```

310.Minimum Height Trees

For a undirected graph with tree characteristics, we can choose any node as the root. The result graph is then a rooted tree. Among all possible rooted trees, those with minimum height are called minimum height trees (MHTs). Given such a graph, write a function to find all the MHTs and return a list of their root labels.

Format

The graph contains `n` nodes which are labeled from `0` to `n - 1`. You will be given the number `n` and a list of undirected `edges` (each edge is a pair of labels).

You can assume that no duplicate edges will appear in `edges`. Since all edges are undirected, `[0, 1]` is the same as `[1, 0]` and thus will not appear together in `edges`.

Example 1:

Given `n = 4`, `edges = [[1, 0], [1, 2], [1, 3]]`

```
        0
```

```
      1
     / \
    2   3
```

return `[1]`

Example 2:

Given `n = 6`, `edges = [[0, 3], [1, 3], [2, 3], [4, 3], [5, 4]]`

```
    0  1  2
     \ | /
       3
       |
       4
       |
       5
```

return `[3, 4]`

Note:

(1) According to the definition of tree on Wikipedia: “a tree is an undirected graph in which any two vertices are connected by *exactly* one path. In other words, any connected graph without simple cycles is a tree.”

(2) The height of a rooted tree is the number of edges on the longest downward path between the root and a leaf.

```java
public class Solution {
    public List<Integer> findMinHeightTrees(int n, int[][] edges) {
        if (n == 1)
            return Collections.singletonList(0);
        List<Set<Integer>> adj = new ArrayList<>(n);
        for (int i = 0; i < n; ++i)
            adj.add(new HashSet<>());
        for (int[] edge : edges) {
            adj.get(edge[0]).add(edge[1]);
            adj.get(edge[1]).add(edge[0]);
        }
        List<Integer> leaves = new ArrayList<>();
        for (int i = 0; i < n; ++i)
            if (adj.get(i).size() == 1)
                leaves.add(i);
        while (n > 2) {
            n -= leaves.size();
            List<Integer> newLeaves = new ArrayList<>();
            for (int i : leaves) {
                int j = adj.get(i).iterator().next();
                adj.get(j).remove(i);
                if (adj.get(j).size() == 1)
                    newLeaves.add(j);
            }
            leaves = newLeaves;
        }
        return leaves;
    }
}
```

311.Sparse Matrix Multiplication

Given two sparse matrices **A** and **B**, return the result of **AB**.

You may assume that **A**'s column number is equal to **B**'s row number.

Example:

```
A = [
  [ 1, 0, 0],
  [-1, 0, 3]
]
B = [
  [ 7, 0, 0 ],
  [ 0, 0, 0 ],
  [ 0, 0, 1 ]
]
     |  1 0 0 |   | 7 0 0 |   |  7 0 0 |
AB = | -1 0 3 | x | 0 0 0 | = | -7 0 3 |
                  | 0 0 1 |
```

```java
public class Solution {
    public int[][] multiply(int[][] A, int[][] B) {
        int m = A.length, n = A[0].length, nB = B[0].length;
        int[][] C = new int[m][nB];
        for (int i = 0; i < m; i++)
            for (int k = 0; k < n; k++)
                if (A[i][k] != 0) {
                    for (int j = 0; j < nB; j++)
```

```
                        C[i][j] += A[i][k] * B[k][j];
        return C;
    }
}
```

312.Burst Balloons

Given `n` balloons, indexed from `0` to `n-1`. Each balloon is painted with a number on it represented by array `nums`. You are asked to burst all the balloons. If the you burst balloon `i` you will get `nums[left] * nums[i] * nums[right]` coins. Here `left` and `right` are adjacent indices of `i`. After the burst, the `left` and `right` then becomes adjacent.
Find the maximum coins you can collect by bursting the balloons wisely.
Note:
(1) You may imagine `nums[-1] = nums[n] = 1`. They are not real therefore you can not burst them.
(2) 0 ≤ `n` ≤ 500, 0 ≤ `nums[i]` ≤ 100
Example:
Given `[3, 1, 5, 8]`
Return `167`

```
    nums = [3,1,5,8] --> [3,5,8] -->   [3,8]   -->  [8]  --> []
   coins =  3*1*5      +  3*5*8    +  1*3*8      + 1*8*1   = 167
```

```
public class Solution {
    public int maxCoins(int[] nums) {
        int[] nnums = new int[nums.length + 2];
        int n = 1;
        for (int x : nums)
            if (x > 0)
                nnums[n++] = x;
        nnums[0] = nnums[n++] = 1;
        int[][] memo = new int[n][n];
        return burst(memo, nnums, 0, n - 1);
    }
    public int burst(int[][] memo, int[] nums, int left, int right) {
        if (left + 1 == right)
            return 0;
        if (memo[left][right] > 0)
            return memo[left][right];
        int ans = 0;
        for (int i = left + 1; i < right; ++i)
            ans = Math.max(ans, nums[left] * nums[i] * nums[right]
                    + burst(memo, nums, left, i) + burst(memo, nums, i, right));
        memo[left][right] = ans;
        return ans;
    }
}
public class Solution {
    public int maxCoins(int[] nums) {
        int[] nnums = new int[nums.length + 2];
        int n = 1;
        for (int x : nums)
            if (x > 0)
                nnums[n++] = x;
        nnums[0] = nnums[n++] = 1;
        int[][] dp = new int[n][n];
        for (int k = 2; k < n; ++k)
            for (int left = 0; left < n - k; ++left) {
                int right = left + k;
                for (int i = left + 1; i < right; ++i)
                    dp[left][right] = Math.max(dp[left][right],
                            nnums[left] * nnums[i] * nnums[right] + dp[left][i]
                                    + dp[i][right]);
            }
        return dp[0][n - 1];
    }
}
```

313.Super Ugly Number

Write a program to find the nth super ugly number.
Super ugly numbers are positive numbers whose all prime factors are in the given prime list `primes` of size `k`. For example, `[1, 2, 4, 7, 8, 13, 14, 16, 19, 26, 28, 32]` is the sequence of the first 12 super ugly numbers given `primes` = `[2, 7, 13, 19]` of size 4.
Note:
(1) `1` is a super ugly number for any given `primes`.
(2) The given numbers in `primes` are in ascending order.
(3) 0 < `k` ≤ 100, 0 < `n` ≤ 10^6, 0 < `primes[i]` < 1000.
(4) The nth super ugly number is guaranteed to fit in a 32-bit signed integer.

```
public class Solution {
    public int nthSuperUglyNumber(int n, int[] primes) {
        int[] ugly = new int[n];
```

```
        int[] val = new int[primes.length];
        Arrays.fill(val, 1);
        int next = 1;
        for (int i = 0; i < n; i++) {
            ugly[i] = next;
            next = Integer.MAX_VALUE;
            for (int j = 0; j < primes.length; j++) {
                if (val[j] == ugly[i])
                    val[j] = ugly[idx[j]++] * primes[j];
                next = Math.min(next, val[j]);
            }
        }
        return ugly[n - 1];
    }
}
```

314.Binary Tree Vertical Order Traversal

Given a binary tree, return the *vertical order* traversal of its nodes' values. (ie, from top to bottom, column by column).
If two nodes are in the same row and column, the order should be from **left to right**.

Examples:

Given binary tree `[3,9,20,null,null,15,7]`,

```
   3
  /\
 /  \
 9  20
    /\
   /  \
  15   7
```

return its vertical order traversal as:

```
[ [9],
  [3,15],
  [20],
  [7]]
```

Given binary tree `[3,9,8,4,0,1,7]`,

```
     3
    /\
   /  \
   9   8
  /\  /\
 /  \/  \
 4  01   7
```

return its vertical order traversal as:

```
[ [4],
  [9],
  [3,0,1],
  [8],
  [7]]
```

Given binary tree `[3,9,8,4,0,1,7,null,null,null,2,5]` (0's right child is 2 and 1's left child is 5),

```
     3
    /\
   /  \
   9   8
  /\  /\
 /  \/  \
 4  01   7
    /\
   /  \
   5   2
```

return its vertical order traversal as:

```
[ [4],
  [9,5],
  [3,0,1],
  [8,2],
  [7]]
```

```
    public List<List<Integer>> verticalOrder(TreeNode root) {
        List<List<Integer>> cols = new ArrayList<>();
        if (root == null)
            return cols;
        int[] range = new int[] { 0, 0 };
        getRange(root, range, 0);
        for (int i = range[0]; i <= range[1]; i++)
            cols.add(new ArrayList<Integer>());
        Queue<TreeNode> queue = new LinkedList<>();
        Queue<Integer> colQueue = new LinkedList<>();
        queue.add(root);
        colQueue.add(-range[0]);
        while (!queue.isEmpty()) {
            TreeNode node = queue.poll();
            int col = colQueue.poll();
            cols.get(col).add(node.val);
            if (node.left != null) {
                queue.add(node.left);
                colQueue.add(col - 1);
            }
            if (node.right != null) {
                queue.add(node.right);
                colQueue.add(col + 1);
            }
        }
        return cols;
    }
    public void getRange(TreeNode root, int[] range, int col) {
        if (root == null) {
            return;
        }
        range[0] = Math.min(range[0], col);
        range[1] = Math.max(range[1], col);
        getRange(root.left, range, col - 1);
        getRange(root.right, range, col + 1);
    }
}
```

315.Count of Smaller Numbers After Self

You are given an integer array *nums* and you have to return a new *counts* array. The *counts* array has the property where `counts[i]` is the number of smaller elements to the right of `nums[i]`.

Example:

```
Given nums = [5, 2, 6, 1]
To the right of 5 there are 2 smaller elements (2 and 1).
To the right of 2 there is only 1 smaller element (1).
To the right of 6 there is 1 smaller element (1).
To the right of 1 there is 0 smaller element.
```

Return the array `[2, 1, 1, 0]`.

```
public class Solution {
    public List<Integer> countSmaller(int[] nums) {
        if (nums == null || nums.length == 0)
            return new ArrayList<Integer>();
        Num[] arr = new Num[nums.length];
        for (int i = 0; i < nums.length; i++)
            arr[i] = new Num(nums[i], i);
        int[] smaller = new int[nums.length];
        mergeCount(smaller, arr, 0, nums.length - 1);
        List<Integer> res = new ArrayList<Integer>();
        for (int count : smaller)
            res.add(count);
        return res;
    }
    private void mergeCount(int[] smaller, Num[] nums, int l, int r) {
        if (l >= r)
            return;
        int mid = l + (r - l) / 2;
        mergeCount(smaller, nums, l, mid);
        mergeCount(smaller, nums, mid + 1, r);
        Num[] cache = new Num[r - l + 1];
        int j = mid + 1, k = mid + 1;
        for (int i = l, idx = 0; i <= mid; i++, idx++) {
            while (j <= r && nums[i].val > nums[j].val)
                j++;
            while (k <= r && nums[k].val < nums[i].val)
                cache[idx++] = nums[k++];
            cache[idx] = nums[i];
```

```
        }
        System.arraycopy(cache, 0, nums, l, k - l);
    }
    class Num {
        int val, idx;
        public Num(int val, int idx) {
            this.val = val;
            this.idx = idx;
        }
    }
}
```

316.Remove Duplicate Letters

Given a string which contains only lowercase letters, remove duplicate letters so that every letter appear once and only once. You must make sure your result is the smallest in lexicographical order among all possible results.

Example:

Given `"bcabc"`
Return `"abc"`
Given `"cbacdcbc"`
Return `"acdb"`

```
public class Solution {
    public String removeDuplicateLetters(String s) {
        int[] cnt = new int[26];
        int pos = 0;
        for (int i = 0; i < s.length(); i++)
            cnt[s.charAt(i) - 'a']++;
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) < s.charAt(pos))
                pos = i;
            if (--cnt[s.charAt(i) - 'a'] == 0)
                break;
        }
        return s.length() == 0 ? ""
                : s.charAt(pos) + removeDuplicateLetters(s.substring(pos + 1)
                        .replaceAll("" + s.charAt(pos), ""));
    }
}
```

317.Shortest Distance from All Buildings

You want to build a house on an *empty* land which reaches all buildings in the shortest amount of distance. You can only move up, down, left and right. You are given a 2D grid of values **0**, **1** or **2**, where:

- Each **0** marks an empty land which you can pass by freely.
- Each **1** marks a building which you cannot pass through.
- Each **2** marks an obstacle which you cannot pass through.

For example, given three buildings at `(0,0)`, `(0,4)`, `(2,2)`, and an obstacle at `(0,2)`:

```
1 - 0 - 2 - 0 - 1
|   |   |   |   |
0 - 0 - 0 - 0 - 0
|   |   |   |   |
0 - 0 - 1 - 0 - 0
```

The point `(1,2)` is an ideal empty land to build a house, as the total travel distance of 3+3+1=7 is minimal. So return 7.

Note:

There will be at least one building. If it is not possible to build such house according to the above rules, return -1.

```
public class Solution {
    public int shortestDistance(int[][] grid) {
        if (grid == null || grid[0].length == 0)
            return 0;
        final int[] shift = new int[] { 0, 1, 0, -1, 0 };
        int row = grid.length, col = grid[0].length;
        int[][] distance = new int[row][col];
        int[][] reach = new int[row][col];
        int buildingNum = 0;
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                if (grid[i][j] == 1) {
                    buildingNum++;
                    Queue<int[]> myQueue = new LinkedList<int[]>();
                    myQueue.offer(new int[] { i, j });
                    boolean[][] isVisited = new boolean[row][col];
                    int level = 1;
                    while (!myQueue.isEmpty()) {
                        int qSize = myQueue.size();
                        for (int q = 0; q < qSize; q++) {
                            int[] curr = myQueue.poll();
                            for (int k = 0; k < 4; k++) {
```

```
                                int nextCol = curr[1] + shift[k + 1];
                                if (nextRow >= 0 && nextRow < row
                                        && nextCol >= 0 && nextCol < col
                                        && grid[nextRow][nextCol] == 0
                                        && !isVisited[nextRow][nextCol]) {
                                    distance[nextRow][nextCol] += level;
                                    reach[nextRow][nextCol]++;
                                    isVisited[nextRow][nextCol] = true;
                                    myQueue.offer(
                                            new int[] { nextRow, nextCol });
                                }
                            }
                        }
                        level++;
                    }
                }
            }
        }
        int shortest = Integer.MAX_VALUE;
        for (int i = 0; i < row; i++)
            for (int j = 0; j < col; j++)
                if (grid[i][j] == 0 && reach[i][j] == buildingNum)
                    shortest = Math.min(shortest, distance[i][j]);
        return shortest == Integer.MAX_VALUE ? -1 : shortest;
    }
}
```

318.Maximum Product of Word Lengths

Given a string array `words`, find the maximum value of `length(word[i]) * length(word[j])` where the two words do not share common letters. You may assume that each word will contain only lower case letters. If no such two words exist, return 0.

Example 1:

Given `["abcw", "baz", "foo", "bar", "xtfn", "abcdef"]`

Return `16`

The two words can be `"abcw", "xtfn"`.

Example 2:

Given `["a", "ab", "abc", "d", "cd", "bcd", "abcd"]`

Return `4`

The two words can be `"ab", "cd"`.

Example 3:

Given `["a", "aa", "aaa", "aaaa"]`

Return `0`

No such pair of words.

```
public class Solution {
    public static int maxProduct(String[] words) {
        if (words == null || words.length == 0)
            return 0;
        int len = words.length;
        int[] value = new int[len];
        for (int i = 0; i < len; i++) {
            String tmp = words[i];
            value[i] = 0;
            for (int j = 0; j < tmp.length(); j++) {
                value[i] |= 1 << (tmp.charAt(j) - 'a');
            }
        }
        int maxProduct = 0;
        for (int i = 0; i < len; i++)
            for (int j = i + 1; j < len; j++) {
                if ((value[i] & value[j]) == 0
                        && (words[i].length() * words[j].length() > maxProduct))
                    maxProduct = words[i].length() * words[j].length();
            }
        return maxProduct;
    }
}
```

319.Bulb Switcher

There are *n* bulbs that are initially off. You first turn on all the bulbs. Then, you turn off every second bulb. On the third round, you toggle every third bulb (turning on if it's off or turning off if it's on). For the *i*th round, you toggle every *i* bulb. For the *n*th round, you only toggle the last bulb. Find how many bulbs are on after *n* rounds.

Example:

```
Given n = 3.
At first, the three bulbs are [off, off, off].
After first round, the three bulbs are [on, on, on].
After second round, the three bulbs are [on, off, on].
```

```
After third round, the three bulbs are [on, off, off].
So you should return 1, because there is only one bulb is on.
```

```
public class Solution {
    public int bulbSwitch(int n) {
        return (int) Math.sqrt(n);
    }
}
```

320.Generalized Abbreviation

Write a function to generate the generalized abbreviations of a word.
Example:
Given word = `"word"`, return the following list (order does not matter):

```
["word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "1o1d", "1or1", "w1r1", "1o2", "2r1", "3d", "w3", "4"]
```

```
public class Solution {
    public List<String> generateAbbreviations(String word) {
        List<String> ans = new ArrayList<String>();
        backtrack(ans, new StringBuilder(), word, 0, 0);
        return ans;
    }
    private void backtrack(List<String> ans, StringBuilder builder, String word,
            int i, int k) {
        int len = builder.length();
        if (i == word.length()) {
            if (k != 0)
                builder.append(k);
            ans.add(builder.toString());
        } else {
            backtrack(ans, builder, word, i + 1, k + 1);
            if (k != 0)
                builder.append(k);
            builder.append(word.charAt(i));
            backtrack(ans, builder, word, i + 1, 0);
        }
        builder.setLength(len);
    }
}
```

321.Create Maximum Number

Given two arrays of length `m` and `n` with digits `0-9` representing two numbers. Create the maximum number of length `k <= m + n`from digits of the two. The relative order of the digits from the same array must be preserved. Return an array of the `k` digits. You should try to optimize your time and space complexity.
Example 1:
nums1 = `[3, 4, 6, 5]`
nums2 = `[9, 1, 2, 5, 8, 3]`
k = `5`
return `[9, 8, 6, 5, 3]`
Example 2:
nums1 = `[6, 7]`
nums2 = `[6, 0, 4]`
k = `5`
return `[6, 7, 6, 0, 4]`
Example 3:
nums1 = `[3, 9]`
nums2 = `[8, 9]`
k = `3`
return `[9, 8, 9]`

```
public class Solution {
    public int[] maxNumber(int[] nums1, int[] nums2, int k) {
        int n = nums1.length;
        int m = nums2.length;
        int[] ans = new int[k];
        for (int i = Math.max(0, k - m); i <= k && i <= n; ++i) {
            int[] candidate = merge(maxArray(nums1, i), maxArray(nums2, k - i),
                    k);
            if (greater(candidate, 0, ans, 0))
                ans = candidate;
        }
        return ans;
    }
    private int[] merge(int[] nums1, int[] nums2, int k) {
        int[] ans = new int[k];
        for (int i = 0, j = 0, r = 0; r < k; ++r)
            ans[r] = greater(nums1, i, nums2, j) ? nums1[i++] : nums2[j++];
        return ans;
```

```
    }
    public boolean greater(int[] nums1, int i, int[] nums2, int j) {
        while (i < nums1.length && j < nums2.length && nums1[i] == nums2[j]) {
            i++;
            j++;
        }
        return j == nums2.length || (i < nums1.length && nums1[i] > nums2[j]);
    }
    public int[] maxArray(int[] nums, int k) {
        int n = nums.length;
        int[] ans = new int[k];
        for (int i = 0, j = 0; i < n; ++i) {
            while (n - i + j > k && j > 0 && ans[j - 1] < nums[i])
                j--;
            if (j < k)
                ans[j++] = nums[i];
        }
        return ans;
    }
}
```

322.Coin Change

You are given coins of different denominations and a total amount of money *amount*. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return `-1`.

Example 1:

coins = `[1, 2, 5]`, amount = `11`
return `3` (11 = 5 + 5 + 1)

Example 2:

coins = `[2]`, amount = `3`
return `-1`.

Note:

You may assume that you have an infinite number of each kind of coin.

```
public class Solution {
    public int coinChange(int[] coins, int amount) {
        if (amount < 1)
            return 0;
        return coinChange(coins, amount, new int[amount]);
    }
    private int coinChange(int[] coins, int rem, int[] count) {
        if (rem < 0)
            return -1;
        if (rem == 0)
            return 0;
        if (count[rem - 1] != 0)
            return count[rem - 1];
        int min = Integer.MAX_VALUE;
        for (int coin : coins) {
            int res = coinChange(coins, rem - coin, count);
            if (res >= 0 && res < min)
                min = 1 + res;
        }
        count[rem - 1] = (min == Integer.MAX_VALUE) ? -1 : min;
        return count[rem - 1];
    }
}
```

```
public class Solution {
    public int coinChange(int[] coins, int amount) {
        int max = amount + 1;
        int[] dp = new int[amount + 1];
        Arrays.fill(dp, max);
        dp[0] = 0;
        for (int i = 1; i <= amount; i++) {
            for (int j = 0; j < coins.length; j++) {
                if (coins[j] <= i) {
                    dp[i] = Math.min(dp[i], dp[i - coins[j]] + 1);
                }
            }
        }
        return dp[amount] > amount ? -1 : dp[amount];
    }
}
```

323.Number of Connected Components in an Undirected Graph

Given `n` nodes labeled from `0` to `n - 1` and a list of undirected edges (each edge is a pair of nodes), write a function to find the number of connected components in an undirected graph.

Example 1:

```
     0          3
     |          |
     1 --- 2    4
```

Given `n = 5` and `edges = [[0, 1], [1, 2], [3, 4]]`, return `2`.

Example 2:

```
     0           4
     |           |
     1 --- 2 --- 3
```

Given `n = 5` and `edges = [[0, 1], [1, 2], [2, 3], [3, 4]]`, return `1`.

Note:

You can assume that no duplicate edges will appear in `edges`. Since all edges are undirected, `[0, 1]` is the same as `[1, 0]` and thus will not appear together in `edges`.

```
public class Solution {
    public int countComponents(int n, int[][] edges) {
        int[] roots = new int[n];
        for (int i = 0; i < n; i++)
            roots[i] = i;
        for (int[] e : edges) {
            int root1 = find(roots, e[0]);
            int root2 = find(roots, e[1]);
            if (root1 != root2) {
                roots[root1] = root2; // union
                n--;
            }
        }
        return n;
    }
    public int find(int[] roots, int id) {
        while (roots[id] != id) {
            roots[id] = roots[roots[id]]; // optional: path compression
            id = roots[id];
        }
        return id;
    }
}
```

```
public class Solution {
    public int countComponents(int n, int[][] edges) {
        int count = 0;
        int m = edges.length;
        Map<Integer, List<Integer>> eM = new HashMap<>();
        for (int i = 0; i < m; ++i) {
            List<Integer> tmpL = eM.getOrDefault(edges[i][0],
                    new ArrayList<Integer>());
            List<Integer> tmpL2 = eM.getOrDefault(edges[i][1],
                    new ArrayList<Integer>());
            tmpL.add(edges[i][1]);
            tmpL2.add(edges[i][0]);
            eM.put(edges[i][0], tmpL);
            eM.put(edges[i][1], tmpL2);
        }
        Set<Integer> consumed = new HashSet<Integer>();
        for (int i = 0; i < n; ++i) {
            if (consumed.contains(i))
                continue;
            consumed.add(i);
            ++count;
            List<Integer> l = eM.getOrDefault(i, new ArrayList<Integer>());
            while (!l.isEmpty()) {
                int j = l.get(0);
                l.remove(0);
                if (consumed.contains(j))
                    continue;
                consumed.add(j);
                l.addAll(eM.getOrDefault(j, new ArrayList<Integer>()));
            }
        }
        return count;
    }
}
```

324.Wiggle Sort II

Given an unsorted array `nums`, reorder it such that `nums[0] < nums[1] > nums[2] < nums[3]...`.

Example:
(1) Given `nums = [1, 5, 1, 1, 6, 4]`, one possible answer is `[1, 4, 1, 5, 1, 6]`.
(2) Given `nums = [1, 3, 2, 2, 3, 1]`, one possible answer is `[2, 3, 1, 3, 1, 2]`.
Note:
You may assume all input has valid answer.
Follow Up:
Can you do it in O(n) time and/or in-place with O(1) extra space?

```
public class Solution {
    public void wiggleSort(int[] nums) {
        if (nums == null || nums.length == 0)
            return;
        int len = nums.length;
        int median = findMedian(0, len - 1, len / 2, nums);
        int left = 0, right = len - 1, i = 0;
        while (i <= right) {
            int mappedCurIndex = newIndex(i, len);
            if (nums[mappedCurIndex] > median) {
                int mappedLeftIndex = newIndex(left, len);
                swap(mappedLeftIndex, mappedCurIndex, nums);
                left++;
                i++;
            } else if (nums[mappedCurIndex] < median) {
                int mappedRightIndex = newIndex(right, len);
                swap(mappedCurIndex, mappedRightIndex, nums);
                right--;
            } else
                i++;
        }
    }
    public int newIndex(int index, int len) {
        return (1 + 2 * index) % (len | 1);
    }
    public int findMedian(int start, int end, int k, int[] nums) {
        if (start > end)
            return Integer.MAX_VALUE;
        int pivot = nums[end];
        int indexOfWall = start;
        for (int i = start; i < end; i++) {
            if (nums[i] <= pivot) {
                swap(i, indexOfWall, nums);
                indexOfWall++;
            }
        }
        swap(indexOfWall, end, nums);
        if (indexOfWall == k)
            return nums[indexOfWall];
        else if (indexOfWall < k)
            return findMedian(indexOfWall + 1, end, k, nums);
        else
            return findMedian(start, indexOfWall - 1, k, nums);
    }
    public void swap(int i, int j, int[] nums) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
}
```

325.Maximum Size Subarray Sum Equals k

Given an array *nums* and a target value *k*, find the maximum length of a subarray that sums to *k*. If there isn't one, return 0 instead.
Note:
The sum of the entire *nums* array is guaranteed to fit within the 32-bit signed integer range.
Example 1:
Given *nums* = `[1, -1, 5, -2, 3]`, *k* = `3`,
return `4`. (because the subarray `[1, -1, 5, -2]` sums to 3 and is the longest)
Example 2:
Given *nums* = `[-2, -1, 2, 1]`, *k* = `1`,
return `2`. (because the subarray `[-1, 2]` sums to 1 and is the longest)
Follow Up:
Can you do it in O(*n*) time?

```
public class Solution {
    public int maxSubArrayLen(int[] nums, int k) {
        int sum = 0, max = 0;
        HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
        for (int i = 0; i < nums.length; i++) {
            sum = sum + nums[i];
```

```
                max = i + 1;
            else if (map.containsKey(sum - k))
                max = Math.max(max, i - map.get(sum - k));
            if (!map.containsKey(sum))
                map.put(sum, i);
        }
        return max;
    }
}
```

326.Power of Three

Given an integer, write a function to determine if it is a power of three.

Follow up:

Could you do it without using any loop / recursion?

```
public class Solution {
    public boolean isPowerOfThree(int n) {
        if (n < 1) {
            return false;
        }
        while (n % 3 == 0) {
            n /= 3;
        }
        return n == 1;
    }
}
```

327.Count of Range Sum

Given an integer array `nums`, return the number of range sums that lie in `[lower, upper]` inclusive.

Range sum `S(i, j)` is defined as the sum of the elements in `nums` between indices `i` and `j` (`i` ? `j`), inclusive.

Note:

A naive algorithm of $O(n^2)$ is trivial. You MUST do better than that.

Example:

Given *nums* = `[-2, 5, -1]`, *lower* = `-2`, *upper* = `2`,

Return `3`.

The three ranges are : `[0, 0]`, `[2, 2]`, `[0, 2]` and their respective sums are: `-2, -1, 2`.

```
public class Solution {
    public int countRangeSum(int[] nums, int lower, int upper) {
        int n = nums.length;
        long[] sums = new long[n + 1];
        for (int i = 0; i < n; ++i)
            sums[i + 1] = sums[i] + nums[i];
        return countWhileMergeSort(sums, 0, n + 1, lower, upper);
    }
    private int countWhileMergeSort(long[] sums, int start, int end, int lower,
            int upper) {
        if (end - start <= 1)
            return 0;
        int mid = (start + end) / 2;
        int count = countWhileMergeSort(sums, start, mid, lower, upper)
                + countWhileMergeSort(sums, mid, end, lower, upper);
        int j = mid, k = mid, t = mid;
        long[] cache = new long[end - start];
        for (int i = start, r = 0; i < mid; ++i, ++r) {
            while (k < end && sums[k] - sums[i] < lower)
                k++;
            while (j < end && sums[j] - sums[i] <= upper)
                j++;
            while (t < end && sums[t] < sums[i])
                cache[r++] = sums[t++];
            cache[r] = sums[i];
            count += j - k;
        }
        System.arraycopy(cache, 0, sums, start, t - start);
        return count;
    }
}
```

328.Odd Even Linked List

Given a singly linked list, group all odd nodes together followed by the even nodes. Please note here we are talking about the node number and not the value in the nodes.

You should try to do it in place. The program should run in O(1) space complexity and O(nodes) time complexity.

Example:

Given `1->2->3->4->5->NULL`,

return `1->3->5->2->4->NULL`.

Note:

The relative order inside both the even and odd groups should remain as it was in the input.

The first node is considered odd, the second node even and so on ...

```
public class Solution {
```

```
    public ListNode oddEvenList(ListNode head) {
        if (head == null)
            return null;
        ListNode odd = head, even = head.next, evenHead = even;
        while (even != null && even.next != null) {
            odd.next = even.next;
            odd = odd.next;
            even.next = odd.next;
            even = even.next;
        }
        odd.next = evenHead;
        return head;
    }
}
```

329.Longest Increasing Path in a Matrix

Given an integer matrix, find the length of the longest increasing path.
From each cell, you can either move to four directions: left, right, up or down. You may NOT move diagonally or move outside of the boundary (i.e. wrap-around is not allowed).
Example 1:

```
nums = [
  [9,9,4],
  [6,6,8],
  [2,1,1]
]
```

Return `4`
The longest increasing path is `[1, 2, 6, 9]`.
Example 2:

```
nums = [
  [3,4,5],
  [3,2,6],
  [2,2,1]
]
```

Return `4`
The longest increasing path is `[3, 4, 5, 6]`. Moving diagonally is not allowed.

```
public class Solution {
    private static final int[][] dirs = { { 0, 1 }, { 1, 0 }, { 0, -1 },
            { -1, 0 } };
    private int m, n;
    public int longestIncreasingPath(int[][] matrix) {
        if (matrix.length == 0)
            return 0;
        m = matrix.length;
        n = matrix[0].length;
        int[][] cache = new int[m][n];
        int ans = 0;
        for (int i = 0; i < m; ++i)
            for (int j = 0; j < n; ++j)
                ans = Math.max(ans, dfs(matrix, i, j, cache));
        return ans;
    }
    private int dfs(int[][] matrix, int i, int j, int[][] cache) {
        if (cache[i][j] != 0)
            return cache[i][j];
        for (int[] d : dirs) {
            int x = i + d[0], y = j + d[1];
            if (0 <= x && x < m && 0 <= y && y < n
                    && matrix[x][y] > matrix[i][j])
                cache[i][j] = Math.max(cache[i][j], dfs(matrix, x, y, cache));
        }
        return ++cache[i][j];
    }
}
```

330.Patching Array

Given a sorted positive integer array *nums* and an integer *n*, add/patch elements to the array such that any number in range `[1, n]`inclusive can be formed by the sum of some elements in the array. Return the minimum number of patches required.
Example 1:
nums = `[1, 3]`, *n* = `6`
Return `1`.
Combinations of *nums* are `[1], [3], [1,3]`, which form possible sums of: `1, 3, 4`.
Now if we add/patch `2` to *nums*, the combinations are: `[1], [2], [3], [1,3], [2,3], [1,2,3]`.
Possible sums are `1, 2, 3, 4, 5, 6`, which now covers the range `[1, 6]`.
So we only need `1` patch.

Example 2:
nums = `[1, 5, 10]`, *n* = `20`
Return `2`.
The two patches can be `[2, 4]`.
Example 3:
nums = `[1, 2, 2]`, *n* = `5`
Return `0`.

```
public class Solution {
    public int minPatches(int[] nums, int n) {
        int patches = 0, i = 0;
        long miss = 1;
        while (miss <= n) {
            if (i < nums.length && nums[i] <= miss)
                miss += nums[i++];
            else {
                miss += miss;
                patches++;
            }
        }
        return patches;
    }
}
```

331.Verify Preorder Serialization of a Binary Tree

One way to serialize a binary tree is to use pre-order traversal. When we encounter a non-null node, we record the node's value. If it is a null node, we record using a sentinel value such as `#`.

```
     _9_
    /   \
   3     2
  / \   / \
 4   1  #  6
/ \ / \   / \
# # # #   # #
```

For example, the above binary tree can be serialized to the string `"9,3,4,#,#,1,#,#,2,#,6,#,#"`, where `#` represents a null node.
Given a string of comma separated values, verify whether it is a correct preorder traversal serialization of a binary tree. Find an algorithm without reconstructing the tree.
Each comma separated value in the string must be either an integer or a character `'#'` representing `null` pointer.
You may assume that the input format is always valid, for example it could never contain two consecutive commas such as `"1,,3"`.
Example 1:
`"9,3,4,#,#,1,#,#,2,#,6,#,#"`
Return `true`
Example 2:
`"1,#"`
Return `false`
Example 3:
`"9,#,#,1"`
Return `false`

```
public class Solution {
    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;
    }
}
```

332.Reconstruct Itinerary

Given a list of airline tickets represented by pairs of departure and arrival airports `[from, to]`, reconstruct the itinerary in order. All of the tickets belong to a man who departs from `JFK`. Thus, the itinerary must begin with `JFK`.
Note:

1. If there are multiple valid itineraries, you should return the itinerary that has the smallest lexical order when read as a single string. For example, the itinerary `["JFK", "LGA"]` has a smaller lexical order than `["JFK", "LGB"]`.
2. All airports are represented by three capital letters (IATA code).
3. You may assume all tickets form at least one valid itinerary.

Example 1:
`tickets` = `[["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]`
Return `["JFK", "MUC", "LHR", "SFO", "SJC"]`.
Example 2:
`tickets` = `[["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],["ATL","SFO"]]`

Return `["JFK","ATL","JFK","SFO","ATL","SFO"]`.
Another possible reconstruction is `["JFK","SFO","ATL","JFK","ATL","SFO"]`. But it is larger in lexical order.

```
public class Solution {
    public List<String> findItinerary(String[][] tickets) {
        Map<String, PriorityQueue<String>> targets = new HashMap<>();
        for (String[] ticket : tickets)
            targets.computeIfAbsent(ticket[0], k -> new PriorityQueue<>())
                    .add(ticket[1]);
        List<String> route = new LinkedList<>();
        Stack<String> stack = new Stack<>();
        stack.push("JFK");
        while (!stack.empty()) {
            while (targets.containsKey(stack.peek())
                    && !targets.get(stack.peek()).isEmpty())
                stack.push(targets.get(stack.peek()).poll());
            route.add(0, stack.pop());
        }
        return route;
    }
}
```

333.Largest BST Subtree

Given a binary tree, find the largest subtree which is a Binary Search Tree (BST), where largest means subtree with largest number of nodes in it.
Note:
A subtree must include all of its descendants.
Here's an example:

```
    10
    / \
   5  15
  / \   \
 1   8   7
```

The Largest BST Subtree in this case is the highlighted one.
The return value is the subtree's size, which is 3.
Follow up:
Can you figure out ways to solve it with O(n) time complexity?

```
public class Solution {
    public int largestBSTSubtree(TreeNode root) {
        if (root == null)
            return 0;
        if (root.left == null && root.right == null)
            return 1;
        if (isValid(root, null, null))
            return countNode(root);
        return Math.max(largestBSTSubtree(root.left),
                largestBSTSubtree(root.right));
    }
    public boolean isValid(TreeNode root, Integer min, Integer max) {
        if (root == null)
            return true;
        if (min != null && min >= root.val)
            return false;
        if (max != null && max <= root.val)
            return false;
        return isValid(root.left, min, root.val)
                && isValid(root.right, root.val, max);
    }
    public int countNode(TreeNode root) {
        if (root == null)
            return 0;
        if (root.left == null && root.right == null)
            return 1;
        return 1 + countNode(root.left) + countNode(root.right);
    }
}
```

334.Increasing Triplet Subsequence

Given an unsorted array return whether an increasing subsequence of length 3 exists or not in the array.
Formally the function should:
Return true if there exists *i, j, k*
such that *arr[i] < arr[j] < arr[k]* given $0 \le i < j < k \le n$-1 else return false.
Your algorithm should run in O(*n*) time complexity and O(*1*) space complexity.
Examples:
Given `[1, 2, 3, 4, 5]`,
return `true`

Given `[5, 4, 3, 2, 1]`,
return `false`.

```
public class Solution {
    public boolean increasingTriplet(int[] nums) {
        int small = Integer.MAX_VALUE, big = Integer.MAX_VALUE;
        for (int n : nums) {
            if (n <= small)
                small = n;
            else if (n <= big)
                big = n;
            else
                return true;
        }
        return false;
    }
}
```

335.Self Crossing

You are given an array *x* of `n` positive numbers. You start at point `(0,0)` and moves `x[0]` metres to the north, then `x[1]` metres to the west, `x[2]` metres to the south, `x[3]` metres to the east and so on. In other words, after each move your direction changes counter-clockwise.
Write a one-pass algorithm with `O(1)` extra space to determine, if your path crosses itself, or not.
Example 1:

```
Given x = [2, 1, 1, 2],
?????
?   ?
???????>
    ?
Return true (self crossing)
```

Example 2:

```
Given x = [1, 2, 3, 4],
????????
?       ?
?
?
?????????????>
Return false (not self crossing)
```

Example 3:

```
Given x = [1, 1, 1, 1],
?????
?   ?
?????>
Return true (self crossing)
```

```
public class Solution {
    public boolean isSelfCrossing(int[] x) {
        for (int i = 3, l = x.length; i < l; i++) {
            if (x[i] >= x[i - 2] && x[i - 1] <= x[i - 3])
                return true;
            else if (i >= 4 && x[i - 1] == x[i - 3]
                    && x[i] + x[i - 4] >= x[i - 2])
                return true;
            else if (i >= 5 && x[i - 2] >= x[i - 4]
                    && x[i] + x[i - 4] >= x[i - 2] && x[i - 1] <= x[i - 3]
                    && x[i - 1] + x[i - 5] >= x[i - 3])
                return true;
        }
        return false;
    }
}
```

336.Palindrome Pairs

Given a list of **unique** words, find all pairs of ***distinct*** indices `(i, j)` in the given list, so that the concatenation of the two words, i.e. `words[i] + words[j]` is a palindrome.
Example 1:
Given `words` = `["bat", "tab", "cat"]`
Return `[[0, 1], [1, 0]]`
The palindromes are `["battab", "tabbat"]`
Example 2:
Given `words` = `["abcd", "dcba", "lls", "s", "sssll"]`
Return `[[0, 1], [1, 0], [3, 2], [2, 4]]`
The palindromes are `["dcbaabcd", "abcddcba", "slls", "llssssll"]`

```
public class Solution {
```

```
        List<List<Integer>> ret = new ArrayList<>();
        if (words == null || words.length < 2)
            return ret;
        Map<String, Integer> map = new HashMap<String, Integer>();
        for (int i = 0; i < words.length; i++)
            map.put(words[i], i);
        for (int i = 0; i < words.length; i++) {
            for (int j = 0; j <= words[i].length(); j++) {
                String str1 = words[i].substring(0, j);
                String str2 = words[i].substring(j);
                if (isPalindrome(str1)) {
                    String str2rvs = new StringBuilder(str2).reverse()
                            .toString();
                    if (map.containsKey(str2rvs) && map.get(str2rvs) != i) {
                        List<Integer> list = new ArrayList<Integer>();
                        list.add(map.get(str2rvs));
                        list.add(i);
                        ret.add(list);
                    }
                }
                if (isPalindrome(str2)) {
                    String str1rvs = new StringBuilder(str1).reverse()
                            .toString();
                    if (map.containsKey(str1rvs) && map.get(str1rvs) != i
                            && str2.length() != 0) {
                        List<Integer> list = new ArrayList<Integer>();
                        list.add(i);
                        list.add(map.get(str1rvs));
                        ret.add(list);
                    }
                }
            }
        }
        return ret;
    }
    private boolean isPalindrome(String str) {
        int left = 0;
        int right = str.length() - 1;
        while (left <= right)
            if (str.charAt(left++) != str.charAt(right--))
                return false;
        return true;
    }
}
```

337.House Robber III

The thief has found himself a new place for his thievery again. There is only one entrance to this area, called the "root." Besides the root, each house has one and only one parent house. After a tour, the smart thief realized that "all houses in this place forms a binary tree". It will automatically contact the police if two directly-linked houses were broken into on the same night.
Determine the maximum amount of money the thief can rob tonight without alerting the police.

Example 1:

```
     3
    / \
   2   3
    \   \
     3   1
```

Maximum amount of money the thief can rob = 3 + 3 + 1 = **7**.

Example 2:

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

Maximum amount of money the thief can rob = 4 + 5 = **9**.

```
public class Solution {
    public int rob(TreeNode root) {
        int[] nums = robMax(root);
        return Math.max(nums[0], nums[1]);
    }
    int[] robMax(TreeNode root) {
        int[] nums = { 0, 0 };
        if (root != null) {
            int[] leftNums = robMax(root.left);
            int[] rightNums = robMax(root.right);
```

```
            nums[0] = Math.max(leftNums[1], leftNums[0])
                    + Math.max(rightNums[1], rightNums[0]);
        }
        return nums;
    }
}
```

338.Counting Bits

Given a non negative integer number **num**. For every numbers **i** in the range **0 ≤ i ≤ num** calculate the number of 1's in their binary representation and return them as an array.

Example:

For `num = 5` you should return `[0,1,1,2,1,2]`.

Follow up:

- It is very easy to come up with a solution with run time **O(n*sizeof(integer))**. But can you do it in linear time **O(n)** /possibly in a single pass?
- Space complexity should be **O(n)**.
- Can you do it like a boss? Do it without using any builtin function like **__builtin_popcount** in c++ or in any other language.

```
public class Solution {
    public int[] countBits(int num) {
        int[] ans = new int[num + 1];
        for (int i = 1; i <= num; ++i)
            ans[i] = ans[i & (i - 1)] + 1;
        return ans;
    }
}
```

339.Nested List Weight Sum

Given a nested list of integers, return the sum of all integers in the list weighted by their depth.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

Example 1:

Given the list `[[1,1],2,[1,1]]`, return **10**. (four 1's at depth 2, one 2 at depth 1)

Example 2:

Given the list `[1,[4,[6]]]`, return **27**. (one 1 at depth 1, one 4 at depth 2, and one 6 at depth 3; 1 + 4*2 + 6*3 = 27)

```
public class Solution {
    public int depthSum(List<NestedInteger> nestedList) {
        return depthSum(nestedList, 1);
    }
    public int depthSum(List<NestedInteger> list, int depth) {
        int sum = 0;
        for (NestedInteger n : list) {
            if (n.isInteger())
                sum += n.getInteger() * depth;
            else
                sum += depthSum(n.getList(), depth + 1);
        }
        return sum;
    }
}
```

340.Longest Substring with At Most K Distinct Characters

Given a string, find the length of the longest substring T that contains at most *k* distinct characters.

For example, Given s = `“eceba”` and k = 2,

T is "ece" which its length is 3.

```
public class Solution {
    public int lengthOfLongestSubstringKDistinct(String s, int k) {
        int[] count = new int[256];
        int num = 0, i = 0, res = 0;
        for (int j = 0; j < s.length(); j++) {
            if (count[s.charAt(j)]++ == 0)
                num++;
            if (num > k) {
                while (--count[s.charAt(i++)] > 0)
                    ;
                num--;
            }
            res = Math.max(res, j - i + 1);
        }
        return res;
    }
}
```

341.Flatten Nested List Iterator

Given a nested list of integers, implement an iterator to flatten it.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

Example 1:

Given the list `[[1,1],2,[1,1]]`,

By calling *next* repeatedly until *hasNext* returns false, the order of elements returned by *next* should be: `[1,1,2,1,1]`.

Example 2:
Given the list `[1,[4,[6]]]`,
By calling *next* repeatedly until *hasNext* returns false, the order of elements returned by *next* should be: `[1,4,6]`.

```
public class NestedIterator implements Iterator<Integer> {
    Deque<NestedInteger> s;
    public NestedIterator(List<NestedInteger> nestedList) {
        s = new ArrayDeque<>(nestedList == null ? Arrays.asList() : nestedList);
    }
    @Override
    public Integer next() {
        return s.pollFirst().getInteger();
    }
    @Override
    public boolean hasNext() {
        while (!s.isEmpty() && !s.peekFirst().isInteger()) {
            List<NestedInteger> list = s.pollFirst().getList();
            for (int i = list.size() - 1; i >= 0; i--)
                s.addFirst(list.get(i));
        }
        return !s.isEmpty();
    }
}
```

342.Power of Four

Given an integer (signed 32 bits), write a function to check whether it is a power of 4.
Example:
Given num = 16, return true. Given num = 5, return false.
Follow up: Could you solve it without loops/recursion?

```
public class Solution {
    public boolean isPowerOfFour(int num) {
        return Integer.toString(num, 4).matches("10*");
    }
}
```

343.Integer Break

Given a positive integer *n*, break it into the sum of **at least** two positive integers and maximize the product of those integers. Return the maximum product you can get.
For example, given *n* = 2, return 1 (2 = 1 + 1); given *n* = 10, return 36 (10 = 3 + 3 + 4).
Note: You may assume that *n* is not less than 2 and not larger than 58.

```
public class Solution {
    public int integerBreak(int n) {
        if (n == 2)
            return 1;
        if (n == 3)
            return 2;
        int product = 1;
        while (n > 4) {
            product *= 3;
            n -= 3;
        }
        product *= n;
        return product;
    }
}
```

344.Reverse String

Write a function that takes a string as input and returns the string reversed.
Example:
Given s = "hello", return "olleh".

```
public class Solution {
    public String reverseString(String s) {
        char[] word = s.toCharArray();
        int i = 0;
        int j = s.length() - 1;
        while (i < j) {
            char temp = word[i];
            word[i] = word[j];
            word[j] = temp;
            i++;
            j--;
        }
        return new String(word);
    }
}
```

345.Reverse Vowels of a String

Write a function that takes a string as input and reverse only the vowels of a string.
Example 1:
Given s = "hello", return "holle".

Example 2:
Given s = "leetcode", return "leotcede".
Note:
The vowels does not include the letter "y".

```
public class Solution {
    public String reverseVowels(String s) {
        char[] chars = s.toCharArray();
        Stack<Character> charStack = new Stack<Character>();
        for (int i = 0; i < chars.length; ++i)
            if (isVowel(chars[i]))
                charStack.push(chars[i]);
        for (int i = 0; i < chars.length; ++i) {
            if (isVowel(chars[i]))
                chars[i] = charStack.pop();
        return new String(chars);
    }
    private boolean isVowel(char c) {
        c = Character.toLowerCase(c);
        if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u')
            return true;
        return false;
    }
}
```

346.Moving Average from Data Stream

Given a stream of integers and a window size, calculate the moving average of all integers in the sliding window.
For example,

```
MovingAverage m = new MovingAverage(3);
m.next(1) = 1
m.next(10) = (1 + 10) / 2
m.next(3) = (1 + 10 + 3) / 3
m.next(5) = (10 + 3 + 5) / 3
```

```
public class MovingAverage {
    private double previousSum = 0.0;
    private int maxSize;
    private Queue<Integer> currentWindow;
    public MovingAverage(int size) {
        currentWindow = new LinkedList<Integer>();
        maxSize = size;
    }
    public double next(int val) {
        if (currentWindow.size() == maxSize)
            previousSum -= currentWindow.remove();
        previousSum += val;
        currentWindow.add(val);
        return previousSum / currentWindow.size();
    }
}
```

347.Top K Frequent Elements

Given a non-empty array of integers, return the ***k*** most frequent elements.
For example,
Given `[1,1,1,2,2,3]` and k = 2, return `[1,2]`.
Note:

- You may assume *k* is always valid, 1 ? *k* ? number of unique elements.
- Your algorithm's time complexity **must be** better than O(*n* log *n*), where *n* is the array's size.

```
public class Solution {
    public List<Integer> topKFrequent(int[] nums, int k) {
        List<Integer>[] bucket = new List[nums.length + 1];
        Map<Integer, Integer> frequencyMap = new HashMap<Integer, Integer>();
        for (int n : nums) {
            frequencyMap.put(n, frequencyMap.getOrDefault(n, 0) + 1);
        }
        for (int key : frequencyMap.keySet()) {
            int frequency = frequencyMap.get(key);
            if (bucket[frequency] == null) {
                bucket[frequency] = new ArrayList<>();
            }
            bucket[frequency].add(key);
        }
        List<Integer> res = new ArrayList<>();
        for (int pos = bucket.length - 1; pos >= 0 && res.size() < k; pos--) {
            if (bucket[pos] != null) {
                res.addAll(bucket[pos]);
            }
        }
```

```
        return res.subList(0, k);
    }
}
```

348.Design Tic-Tac-Toe

Design a Tic-tac-toe game that is played between two players on a *n* x *n* grid.
You may assume the following rules:

1. A move is guaranteed to be valid and is placed on an empty block.
2. Once a winning condition is reached, no more moves is allowed.
3. A player who succeeds in placing *n* of their marks in a horizontal, vertical, or diagonal row wins the game.

Example:

```
Given n = 3, assume that player 1 is "X" and player 2 is "O" in the board.
TicTacToe toe = new TicTacToe(3);
toe.move(0, 0, 1); -> Returns 0 (no one wins)
|X| | |
| | | |    // Player 1 makes a move at (0, 0).
| | | |
toe.move(0, 2, 2); -> Returns 0 (no one wins)
|X| |O|
| | | |    // Player 2 makes a move at (0, 2).
| | | |
toe.move(2, 2, 1); -> Returns 0 (no one wins)
|X| |O|
| | | |    // Player 1 makes a move at (2, 2).
| | |X|
toe.move(1, 1, 2); -> Returns 0 (no one wins)
|X| |O|
| |O| |    // Player 2 makes a move at (1, 1).
| | |X|
toe.move(2, 0, 1); -> Returns 0 (no one wins)
|X| |O|
| |O| |    // Player 1 makes a move at (2, 0).
|X| |X|
toe.move(1, 0, 2); -> Returns 0 (no one wins)
|X| |O|
|O|O| |    // Player 2 makes a move at (1, 0).
|X| |X|
toe.move(2, 1, 1); -> Returns 1 (player 1 wins)
|X| |O|
|O|O| |    // Player 1 makes a move at (2, 1).
|X|X|X|
```

```
public class TicTacToe {
    private int[] rows;
    private int[] cols;
    private int diagonal;
    private int antiDiagonal;
    public TicTacToe(int n) {
        rows = new int[n];
        cols = new int[n];
    }
    public int move(int row, int col, int player) {
        int toAdd = player == 1 ? 1 : -1;
        rows[row] += toAdd;
        cols[col] += toAdd;
        if (row == col)
            diagonal += toAdd;
        if (col == (cols.length - row - 1))
            antiDiagonal += toAdd;
        int size = rows.length;
        if (Math.abs(rows[row]) == size || Math.abs(cols[col]) == size
                || Math.abs(diagonal) == size
                || Math.abs(antiDiagonal) == size) {
            return player;
        }
        return 0;
    }
}
```

349.Intersection of Two Arrays

Given two arrays, write a function to compute their intersection.
Example:
Given *nums1* = `[1, 2, 2, 1]`, *nums2* = `[2, 2]`, return `[2]`.
Note:

- Each element in the result must be unique.
- The result can be in any order.

```
public class Solution {
```

```
    public int[] intersection(int[] nums1, int[] nums2) {
        Set<Integer> set = new HashSet<>();
        Set<Integer> intersect = new HashSet<>();
        for (int i = 0; i < nums1.length; i++)
            set.add(nums1[i]);
        for (int i = 0; i < nums2.length; i++)
            if (set.contains(nums2[i]))
                intersect.add(nums2[i]);
        int[] result = new int[intersect.size()];
        int i = 0;
        for (Integer num : intersect)
            result[i++] = num;
        return result;
    }
}
```

350.Intersection of Two Arrays II

Given two arrays, write a function to compute their intersection.

Example:

Given *nums1* = `[1, 2, 2, 1]`, *nums2* = `[2, 2]`, return `[2, 2]`.

Note:

- Each element in the result should appear as many times as it shows in both arrays.
- The result can be in any order.

Follow up:

- What if the given array is already sorted? How would you optimize your algorithm?
- What if *nums1*'s size is small compared to *nums2*'s size? Which algorithm is better?
- What if elements of *nums2* are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?

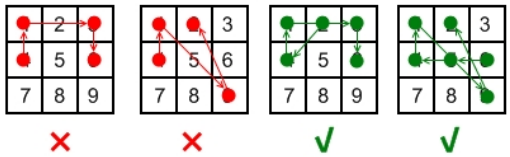
```
public class Solution {
    public int[] intersect(int[] nums1, int[] nums2) {
        Map<Integer, Integer> set = new HashMap<>();
        Map<Integer, Integer> intersect = new HashMap<>();
        int count = 0;
        for (int i = 0; i < nums1.length; i++)
            set.put(nums1[i], set.getOrDefault(nums1[i], 0) + 1);
        for (int i = 0; i < nums2.length; i++) {
            if (set.containsKey(nums2[i]) && set.get(nums2[i]) > intersect
                    .getOrDefault(nums2[i], 0)) {
                intersect.put(nums2[i],
                        intersect.getOrDefault(nums2[i], 0) + 1);
                ++count;
            }
        }
        int[] result = new int[count];
        int j = 0;
        for (Integer num : intersect.keySet())
            for (int i = 0; i < intersect.get(num); ++i)
                result[j++] = num;
        return result;
    }
}
```

351.Android Unlock Patterns

Given an Android **3x3** key lock screen and two integers **m** and **n**, where 1 ≤ m ≤ n ≤ 9, count the total number of unlock patterns of the Android lock screen, which consist of minimum of **m** keys and maximum **n** keys.

Rules for a valid pattern:

1. Each pattern must connect at least **m** keys and at most **n** keys.
2. All the keys must be distinct.
3. If the line connecting two consecutive keys in the pattern passes through any other keys, the other keys must have previously selected in the pattern. No jumps through non selected key is allowed.
4. The order of keys used matters.



Explanation:

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

Invalid move: `4 - 1 - 3 - 6`
Line 1 - 3 passes through key 2 which had not been selected in the pattern.

Invalid move: `4 - 1 - 9 - 2`
Line 1 - 9 passes through key 5 which had not been selected in the pattern.
Valid move: `2 - 4 - 1 - 3 - 6`
Line 1 - 3 is valid because it passes through key 2, which had been selected in the pattern
Valid move: `6 - 5 - 4 - 1 - 9 - 2`
Line 1 - 9 is valid because it passes through key 5, which had been selected in the pattern.
Example:
Given **m** = 1, **n** = 1, return 9.

```
public class Solution {
    int DFS(boolean vis[], int[][] skip, int cur, int remain) {
        if (remain < 0)
            return 0;
        if (remain == 0)
            return 1;
        vis[cur] = true;
        int rst = 0;
        for (int i = 1; i <= 9; ++i)
            if (!vis[i] && (skip[cur][i] == 0 || (vis[skip[cur][i]])))
                rst += DFS(vis, skip, i, remain - 1);
        vis[cur] = false;
        return rst;
    }
    public int numberOfPatterns(int m, int n) {
        int skip[][] = new int[10][10];
        skip[1][3] = skip[3][1] = 2;
        skip[1][7] = skip[7][1] = 4;
        skip[3][9] = skip[9][3] = 6;
        skip[7][9] = skip[9][7] = 8;
        skip[1][9] = skip[9][1] = skip[2][8] = skip[8][2] = skip[3][7] = skip[7][3] = skip[4][6] = skip[6][4] =
5;
        boolean vis[] = new boolean[10];
        int rst = 0;
        for (int i = m; i <= n; ++i) {
            rst += DFS(vis, skip, 1, i - 1) * 4;
            rst += DFS(vis, skip, 2, i - 1) * 4;
            rst += DFS(vis, skip, 5, i - 1);
        }
        return rst;
    }
}
```

352.Data Stream as Disjoint Intervals

Given a data stream input of non-negative integers a_1, a_2, ..., a_n, ..., summarize the numbers seen so far as a list of disjoint intervals.
For example, suppose the integers from the data stream are 1, 3, 7, 2, 6, ..., then the summary will be:

```
[1, 1]
[1, 1], [3, 3]
[1, 1], [3, 3], [7, 7]
[1, 3], [7, 7]
[1, 3], [6, 7]
```

Follow up:
What if there are lots of merges and the number of disjoint intervals are small compared to the data stream's size?

```
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) {
            tree.get(l).end = tree.get(h).end;
            tree.remove(h);
        } else if (l != null && tree.get(l).end + 1 >= val) {
            tree.get(l).end = Math.max(tree.get(l).end, val);
        } else if (h != null && h == val + 1) {
            tree.put(val, new Interval(val, tree.get(h).end));
            tree.remove(h);
        } else
            tree.put(val, new Interval(val, val));
    }
    public List<Interval> getIntervals() {
        return new ArrayList<>(tree.values());
```

}

353. Design Snake Game

Design a Snake game that is played on a device with screen size = *width* x *height*. Play the game online if you are not familiar with the game.
The snake is initially positioned at the top left corner (0,0) with length = 1 unit.
You are given a list of food's positions in row-column order. When a snake eats the food, its length and the game's score both increase by 1.
Each food appears one by one on the screen. For example, the second food will not appear until the first food was eaten by the snake.
When a food does appear on the screen, it is guaranteed that it will not appear on a block occupied by the snake.
Example:

```
Given width = 3, height = 2, and food = [[1,2],[0,1]].
Snake snake = new Snake(width, height, food);
Initially the snake appears at position (0,0) and the food at (1,2).
|S| | |
| | |F|
snake.move("R"); -> Returns 0
| |S| |
| | |F|
snake.move("D"); -> Returns 0
| | | |
| |S|F|
snake.move("R"); -> Returns 1 (Snake eats the first food and right after that, the second food appears at
(0,1) )
| |F| |
| |S|S|
snake.move("U"); -> Returns 1
| |F|S|
| | |S|
snake.move("L"); -> Returns 2 (Snake eats the second food)
| |S|S|
| | |S|
snake.move("U"); -> Returns -1 (Game over because snake collides with border)
```

```java
public class SnakeGame {
    Set<Integer> set;
    Deque<Integer> body;
    int score;
    int[][] food;
    int foodIndex;
    int width;
    int height;
    public SnakeGame(int width, int height, int[][] food) {
        this.width = width;
        this.height = height;
        this.food = food;
        set = new HashSet<>();
        set.add(0);
        body = new LinkedList<>();
        body.offerLast(0);
    }
    public int move(String direction) {
        if (score == -1)
            return -1;
        int rowHead = body.peekFirst() / width;
        int colHead = body.peekFirst() % width;
        switch (direction) {
        case "U":
            rowHead--;
            break;
        case "D":
            rowHead++;
            break;
        case "L":
            colHead--;
            break;
        default:
            colHead++;
        }
        int head = rowHead * width + colHead;
        set.remove(body.peekLast());
        if (rowHead < 0 || rowHead == height || colHead < 0 || colHead == width
                || set.contains(head))
            return score = -1;
        set.add(head);
        body.offerFirst(head);
```

```
        if (foodIndex < food.length && rowHead == food[foodIndex][0]
                && colHead == food[foodIndex][1]) {
            set.add(body.peekLast());
            foodIndex++;
            return ++score;
        }
        body.pollLast();
        return score;
    }
}
```

354.Russian Doll Envelopes

You have a number of envelopes with widths and heights given as a pair of integers `(w, h)`. One envelope can fit into another if and only if both the width and height of one envelope is greater than the width and height of the other envelope.
What is the maximum number of envelopes can you Russian doll? (put one inside other)

Example:

Given envelopes = `[[5,4],[6,4],[6,7],[2,3]]`, the maximum number of envelopes you can Russian doll is `3` ([2,3] => [5,4] => [6,7]).

```
public class Solution {
    public int maxEnvelopes(int[][] envelopes) {
        if (envelopes == null || envelopes.length == 0 || envelopes[0] == null
                || envelopes[0].length != 2)
            return 0;
        Arrays.sort(envelopes, new Comparator<int[]>() {
            public int compare(int[] arr1, int[] arr2) {
                if (arr1[0] == arr2[0])
                    return arr2[1] - arr1[1];
                else
                    return arr1[0] - arr2[0];
            }
        });
        int dp[] = new int[envelopes.length];
        int len = 0;
        for (int[] envelope : envelopes) {
            int index = Arrays.binarySearch(dp, 0, len, envelope[1]);
            if (index < 0)
                index = -(index + 1);
            dp[index] = envelope[1];
            if (index == len)
                len++;
        }
        return len;
    }
}
```

355.Design Twitter

Design a simplified version of Twitter where users can post tweets, follow/unfollow another user and is able to see the 10 most recent tweets in the user's news feed. Your design should support the following methods:

1. **postTweet(userId, tweetId)**: Compose a new tweet.
2. **getNewsFeed(userId)**: 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 the user followed or by the user herself. Tweets must be ordered from most recent to least recent.
3. **follow(followerId, followeeId)**: Follower follows a followee.
4. **unfollow(followerId, followeeId)**: Follower unfollows a followee.

Example:

```
Twitter twitter = new Twitter();
// User 1 posts a new tweet (id = 5).
twitter.postTweet(1, 5);
// User 1's news feed should return a list with 1 tweet id -> [5].
twitter.getNewsFeed(1);
// User 1 follows user 2.
twitter.follow(1, 2);
// User 2 posts a new tweet (id = 6).
twitter.postTweet(2, 6);
// User 1's news feed should return a list with 2 tweet ids -> [6, 5].
// Tweet id 6 should precede tweet id 5 because it is posted after tweet id 5.
twitter.getNewsFeed(1);
// User 1 unfollows user 2.
twitter.unfollow(1, 2);
// User 1's news feed should return a list with 1 tweet id -> [5],
// since user 1 is no longer following user 2.
twitter.getNewsFeed(1);
```

```
public class Twitter {
    private static int timeStamp = 0;
    private Map<Integer, User> userMap;
    private class Tweet {
        public int id;
```

```
    public int time;
    public Tweet next;
    public Tweet(int id) {
        this.id = id;
        time = timeStamp++;
        next = null;
    }
}
public class User {
    public int id;
    public Set<Integer> followed;
    public Tweet tweet_head;
    public User(int id) {
        this.id = id;
        followed = new HashSet<>();
        follow(id); // first follow itself
        tweet_head = null;
    }
    public void follow(int id) {
        followed.add(id);
    }
    public void unfollow(int id) {
        followed.remove(id);
    }
    public void post(int id) {
        Tweet t = new Tweet(id);
        t.next = tweet_head;
        tweet_head = t;
    }
}
public Twitter() {
    userMap = new HashMap<Integer, User>();
}
public void postTweet(int userId, int tweetId) {
    if (!userMap.containsKey(userId)) {
        User u = new User(userId);
        userMap.put(userId, u);
    }
    userMap.get(userId).post(tweetId);
}
public List<Integer> getNewsFeed(int userId) {
    List<Integer> res = new LinkedList<>();
    if (!userMap.containsKey(userId))
        return res;
    Set<Integer> users = userMap.get(userId).followed;
    PriorityQueue<Tweet> q = new PriorityQueue<Tweet>(users.size(),
            (a, b) -> (b.time - a.time));
    for (int user : users) {
        Tweet t = userMap.get(user).tweet_head;
        if (t != null)
            q.add(t);
    }
    int n = 0;
    while (!q.isEmpty() && n < 10) {
        Tweet t = q.poll();
        res.add(t.id);
        n++;
        if (t.next != null)
            q.add(t.next);
    }
    return res;
}
public void follow(int followerId, int followeeId) {
    if (!userMap.containsKey(followerId)) {
        User u = new User(followerId);
        userMap.put(followerId, u);
    }
    if (!userMap.containsKey(followeeId)) {
        User u = new User(followeeId);
        userMap.put(followeeId, u);
    }
    userMap.get(followerId).follow(followeeId);
}
public void unfollow(int followerId, int followeeId) {
    if (!userMap.containsKey(followerId) || followerId == followeeId)
        return;
    userMap.get(followerId).unfollow(followeeId);
```

```
}
```

356.Line Reflection

Given n points on a 2D plane, find if there is such a line parallel to y-axis that reflect the given points.

Example 1:

Given *points* = `[[1,1],[-1,1]]`, return `true`.

Example 2:

Given *points* = `[[1,1],[-1,-1]]`, return `false`.

Follow up:

Could you do better than O(n^2)?

```
public class Solution {
    public boolean isReflected(int[][] points) {
        int max = Integer.MIN_VALUE;
        int min = Integer.MAX_VALUE;
        HashSet<String> set = new HashSet<>();
        for (int[] p : points) {
            max = Math.max(max, p[0]);
            min = Math.min(min, p[0]);
            String str = p[0] + "a" + p[1];
            set.add(str);
        }
        int sum = max + min;
        for (int[] p : points) {
            String str = (sum - p[0]) + "a" + p[1];
            if (!set.contains(str))
                return false;
        }
        return true;
    }
}
```

357.Count Numbers with Unique Digits

Given a **non-negative** integer n, count all numbers with unique digits, x, where $0 \le x < 10^n$.

Example:

Given n = 2, return 91. (The answer should be the total numbers in the range of $0 \le x < 100$, excluding `[11,22,33,44,55,66,77,88,99]`)

```
public class Solution {
    public int countNumbersWithUniqueDigits(int n) {
        if (n == 0)
            return 1;
        int res = 10;
        int uniqueDigits = 9;
        int availableNumber = 9;
        while (n-- > 1 && availableNumber > 0) {
            uniqueDigits = uniqueDigits * availableNumber;
            res += uniqueDigits;
            availableNumber--;
        }
        return res;
    }
}
```

358.Rearrange String k Distance Apart

Given a non-empty string **s** and an integer **k**, rearrange the string such that the same characters are at least distance **k** from each other.

All input strings are given in lowercase letters. If it is not possible to rearrange the string, return an empty string `""`.

Example 1:

```
s = "aabbcc", k = 3
Result: "abcabc"
The same letters are at least distance 3 from each other.
```

Example 2:

```
s = "aaabc", k = 3
Answer: ""
It is not possible to rearrange the string.
```

Example 3:

```
s = "aaadbbcc", k = 2
Answer: "abacabcd"
Another possible answer is: "abcabcda"
The same letters are at least distance 2 from each other.
```

```
public class Solution {
    public String rearrangeString(String str, int k) {
        int length = str.length();
        int[] count = new int[26];
```

```
        for (int i = 0; i < length; i++)
            count[str.charAt(i) - 'a']++;
        StringBuilder sb = new StringBuilder();
        for (int index = 0; index < length; index++) {
            int candidatePos = findValidMax(count, valid, index);
            if (candidatePos == -1)
                return "";
            count[candidatePos]--;
            valid[candidatePos] = index + k;
            sb.append((char) ('a' + candidatePos));
        }
        return sb.toString();
    }
    private int findValidMax(int[] count, int[] valid, int index) {
        int max = Integer.MIN_VALUE;
        int candidatePos = -1;
        for (int i = 0; i < count.length; i++) {
            if (count[i] > 0 && count[i] > max && index >= valid[i]) {
                max = count[i];
                candidatePos = i;
            }
        }
        return candidatePos;
    }
}
```

359.Logger Rate Limiter

Design a logger system that receive stream of messages along with its timestamps, each message should be printed if and only if it is **not printed in the last 10 seconds**.

Given a message and a timestamp (in seconds granularity), return true if the message should be printed in the given timestamp, otherwise returns false.

It is possible that several messages arrive roughly at the same time.

Example:

```
Logger logger = new Logger();
// logging string "foo" at timestamp 1
logger.shouldPrintMessage(1, "foo"); returns true;
// logging string "bar" at timestamp 2
logger.shouldPrintMessage(2,"bar"); returns true;
// logging string "foo" at timestamp 3
logger.shouldPrintMessage(3,"foo"); returns false;
// logging string "bar" at timestamp 8
logger.shouldPrintMessage(8,"bar"); returns false;
// logging string "foo" at timestamp 10
logger.shouldPrintMessage(10,"foo"); returns false;
// logging string "foo" at timestamp 11
logger.shouldPrintMessage(11,"foo"); returns true;
```

```
public class Logger {
    private Map<String, Integer> ok = new HashMap<>();
    public boolean shouldPrintMessage(int timestamp, String message) {
        if (timestamp < ok.getOrDefault(message, 0))
            return false;
        ok.put(message, timestamp + 10);
        return true;
    }
}
```

360.Sort Transformed Array

Given a **sorted** array of integers *nums* and integer values *a*, *b* and *c*. Apply a function of the form f(*x*) = $ax^2 + bx + c$ to each element *x* in the array.

The returned array must be in **sorted order**.

Expected time complexity: **O(*n*)**

Example:

```
nums = [-4, -2, 2, 4], a = 1, b = 3, c = 5,
Result: [3, 9, 15, 33]
nums = [-4, -2, 2, 4], a = -1, b = 3, c = 5
Result: [-23, -5, 1, 7]
```

```
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) {
```

```
                        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;
    }
}
```

361.Bomb Enemy

Given a 2D grid, each cell is either a wall `'W'`, an enemy `'E'` or empty `'0'` (the number zero), return the maximum enemies you can kill using one bomb.
The bomb kills all the enemies in the same row and column from the planted point until it hits the wall since the wall is too strong to be destroyed.
Note that you can only put the bomb at an empty cell.
Example:

```
For the given grid
0 E 0 0
E 0 W E
0 E 0 0
return 3. (Placing a bomb at (1,1) kills 3 enemies)
```

```
public class Solution {
    public int maxKilledEnemies(char[][] grid) {
        int rowNum = grid.length;
        if (rowNum == 0)
            return 0;
        int colNum = grid[0].length;
        int[][] fromBottom = new int[rowNum][colNum];
        int[][] fromRight = new int[rowNum][colNum];
        for (int i = rowNum - 1; i >= 0; i--) {
            for (int j = colNum - 1; j >= 0; j--) {
                int enemy = grid[i][j] == 'E' ? 1 : 0;
                if (grid[i][j] != 'W') {
                    fromBottom[i][j] = (i == rowNum - 1) ? enemy
                            : fromBottom[i + 1][j] + enemy;
                    fromRight[i][j] = (j == colNum - 1) ? enemy
                            : fromRight[i][j + 1] + enemy;
                } else {
                    fromBottom[i][j] = 0;
                    fromRight[i][j] = 0;
                }
            }
        }
        int[] fromTop = new int[colNum];
        int[] fromLeft = new int[rowNum];
        int max = 0;
        for (int i = 0; i < rowNum; i++) {
            for (int j = 0; j < colNum; j++) {
                if (grid[i][j] != '0') {
                    fromTop[j] = grid[i][j] == 'W' ? 0 : fromTop[j] + 1;
                    fromLeft[i] = grid[i][j] == 'W' ? 0 : fromLeft[i] + 1;
                } else {
                    int num = fromTop[j] + fromLeft[i] + fromBottom[i][j]
                            + fromRight[i][j];
                    max = Math.max(num, max);
                }
            }
        }
        return max;
    }
}
```

362.Design Hit Counter

Design a hit counter which counts the number of hits received in the past 5 minutes.
Each function accepts a timestamp parameter (in seconds granularity) and you may assume that calls are being made to the system in chronological order (ie, the timestamp is monotonically increasing). You may assume that the earliest timestamp starts at 1.
It is possible that several hits arrive roughly at the same time.
Example:

```
// hit at timestamp 1.
counter.hit(1);
// hit at timestamp 2.
counter.hit(2);
// hit at timestamp 3.
counter.hit(3);
// get hits at timestamp 4, should return 3.
counter.getHits(4);
// hit at timestamp 300.
counter.hit(300);
// get hits at timestamp 300, should return 4.
counter.getHits(300);
// get hits at timestamp 301, should return 3.
counter.getHits(301);
```

Follow up:
What if the number of hits per second could be very large? Does your design scale?

```
public class HitCounter {
    private int[] times;
    private int[] hits;
    public HitCounter() {
        times = new int[300];
        hits = new int[300];
    }
    public void hit(int timestamp) {
        int index = timestamp % 300;
        if (times[index] != timestamp) {
            times[index] = timestamp;
            hits[index] = 1;
        } else
            hits[index]++;
    }
    public int getHits(int timestamp) {
        int total = 0;
        for (int i = 0; i < 300; i++) {
            if (timestamp - times[i] < 300) {
                total += hits[i];
            }
        }
        return total;
    }
}
```

363.Max Sum of Rectangle No Larger Than K

Given a non-empty 2D matrix *matrix* and an integer *k*, find the max sum of a rectangle in the *matrix* such that its sum is no larger than *k*.

Example:

```
Given matrix = [
  [1,  0, 1],
  [0, -2, 3]]
k = 2
```

The answer is `2`. Because the sum of rectangle `[[0, 1], [-2, 3]]` is 2 and 2 is the max number no larger than k (k = 2).

Note:

1. The rectangle inside the matrix must have an area > 0.
2. What if the number of rows is much larger than the number of columns?

```
public class Solution {
    public int maxSumSubmatrix(int[][] matrix, int target) {
        int row = matrix.length;
        if (row == 0)
            return 0;
        int col = matrix[0].length;
        int m = Math.min(row, col);
        int n = Math.max(row, col);
        boolean colIsBig = col > row;
        int res = Integer.MIN_VALUE;
        for (int i = 0; i < m; i++) {
            int[] array = new int[n];
            for (int j = i; j >= 0; j--) {
                int val = 0;
                TreeSet<Integer> set = new TreeSet<Integer>();
                set.add(0);
                for (int k = 0; k < n; k++) {
                    array[k] = array[k]
                            + (colIsBig ? matrix[j][k] : matrix[k][j]);
                    val = val + array[k];
```

```
                    Integer subres = set.ceiling(val - target);
                    if (null != subres)
                        res = Math.max(res, val - subres);
                    set.add(val);
                }
            }
        }
        return res;
    }
}
```

364.Nested List Weight Sum II

Given a nested list of integers, return the sum of all integers in the list weighted by their depth.
Each element is either an integer, or a list -- whose elements may also be integers or other lists.
Different from the previous question where weight is increasing from root to leaf, now the weight is defined from bottom up. i.e., the leaf level integers have weight 1, and the root level integers have the largest weight.

Example 1:

Given the list `[[1,1],2,[1,1]]`, return **8**. (four 1's at depth 1, one 2 at depth 2)

Example 2:

Given the list `[1,[4,[6]]]`, return **17**. (one 1 at depth 3, one 4 at depth 2, and one 6 at depth 1; 1*3 + 4*2 + 6*1 = 17)

```
public class Solution {
    public int depthSumInverse(List<NestedInteger> nestedList) {
        return depthSum(nestedList, getDepth(nestedList, 0));
    }
    private static int getDepth(List<NestedInteger> nestedList, int dep) {
        int depth = dep + 1;
        int max = depth;
        for (NestedInteger n : nestedList)
            if (!n.isInteger())
                max = Math.max(max, getDepth(n.getList(), depth));
        return max;
    }
    public int depthSum(List<NestedInteger> list, int depth) {
        int sum = 0;
        for (NestedInteger n : list) {
            if (n.isInteger())
                sum += n.getInteger() * depth;
            else
                sum += depthSum(n.getList(), depth - 1);
        }
        return sum;
    }
}
```

365.Water and Jug Problem

You are given two jugs with capacities *x* and *y* litres. There is an infinite amount of water supply available. You need to determine whether it is possible to measure exactly *z* litres using these two jugs.
If *z* liters of water is measurable, you must have *z* liters of water contained within **one or both buckets** by the end.
Operations allowed:

- Fill any of the jugs completely with water.
- Empty any of the jugs.
- Pour water from one jug into another till the other jug is completely full or the first jug itself is empty.

Example 1: (From the famous *"Die Hard"* example)

```
Input: x = 3, y = 5, z = 4
Output: True
```

Example 2:

```
Input: x = 2, y = 6, z = 5
Output: False
```

```
public class Solution {
    public boolean canMeasureWater(int x, int y, int z) {
        if (x + y < z)
            return false;
        if (x == z || y == z || x + y == z)
            return true;
        return z % GCD(x, y) == 0;
    }
    public int GCD(int a, int b) {
        while (b != 0) {
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
```

}

366.Find Leaves of Binary Tree

Given a binary tree, collect a tree's nodes as if you were doing this: Collect and remove all leaves, repeat until the tree is empty.

Example:

Given binary tree

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

Returns `[4, 5, 3], [2], [1]`.

Explanation:

1. Removing the leaves `[4, 5, 3]` would result in this tree:

```
          1
         /
        2
```

2. Now removing the leaf `[2]` would result in this tree:

```
          1
```

3. Now removing the leaf `[1]` would result in the empty tree:

```
          []
```

Returns `[4, 5, 3], [2], [1]`.

```
public class Solution {
    public List<List<Integer>> findLeaves(TreeNode root) {
        List<List<Integer>> list = new ArrayList<>();
        findLeavesHelper(list, root);
        return list;
    }
    private int findLeavesHelper(List<List<Integer>> list, TreeNode root) {
        if (root == null)
            return -1;
        int leftLevel = findLeavesHelper(list, root.left);
        int rightLevel = findLeavesHelper(list, root.right);
        int level = Math.max(leftLevel, rightLevel) + 1;
        if (list.size() == level)
            list.add(new ArrayList<>());
        list.get(level).add(root.val);
        root.left = root.right = null;
        return level;
    }
}
```

367.Valid Perfect Square

Given a positive integer *num*, write a function which returns True if *num* is a perfect square else False.

Note: Do not use any built-in library function such as `sqrt`.

Example 1:

```
Input: 16
Returns: True
```

Example 2:

```
Input: 14
Returns: False
```

```
public class Solution {
    public boolean isPerfectSquare(int num) {
        int low = 1, high = num;
        while (low <= high) {
            long mid = (low + high) >>> 1;
            if (mid * mid == num)
                return true;
            else if (mid * mid < num)
                low = (int) mid + 1;
            else
                high = (int) mid - 1;
        }
        return false;
    }
}
```

368.Largest Divisible Subset

Given a set of **distinct** positive integers, find the largest subset such that every pair (S_i, S_j) of elements in this subset satisfies: S_i % S_j = 0 or S_j % S_i = 0.

If there are multiple solutions, return any subset is fine.

Example 1:

```
nums: [1,2,3]
Result: [1,2] (of course, [1,3] will also be ok)
```

Example 2:

```
nums: [1,2,4,8]
Result: [1,2,4,8]
```

```java
public class Solution {
    public List<Integer> largestDivisibleSubset(int[] nums) {
        int n = nums.length;
        int[] count = new int[n];
        int[] pre = new int[n];
        Arrays.sort(nums);
        int max = 0, index = -1;
        for (int i = 0; i < n; i++) {
            count[i] = 1;
            pre[i] = -1;
            for (int j = i - 1; j >= 0; j--) {
                if (nums[i] % nums[j] == 0) {
                    if (1 + count[j] > count[i]) {
                        count[i] = count[j] + 1;
                        pre[i] = j;
                    }
                }
            }
            if (count[i] > max) {
                max = count[i];
                index = i;
            }
        }
        List<Integer> res = new ArrayList<>();
        while (index != -1) {
            res.add(nums[index]);
            index = pre[index];
        }
        return res;
    }
}
```

369.Plus One Linked List

Given a non-negative integer represented as **non-empty** a singly linked list of digits, plus one to the integer.

You may assume the integer do not contain any leading zero, except the number 0 itself.

The digits are stored such that the most significant digit is at the head of the list.

Example:

```
Input:
1->2->3
Output:
1->2->4
```

```java
public class Solution {
    public ListNode plusOne(ListNode head) {
        if (DFS(head) == 0) {
            return head;
        } else {
            ListNode newHead = new ListNode(1);
            newHead.next = head;
            return newHead;
        }
    }
    public int DFS(ListNode head) {
        if (head == null)
            return 1;
        int carry = DFS(head.next);
        if (carry == 0)
            return 0;
        int val = head.val + 1;
        head.val = val % 10;
        return val / 10;
    }
}
```

370.Range Addition

Assume you have an array of length ***n*** initialized with all **0**'s and are given ***k*** update operations.
Each operation is represented as a triplet: **[startIndex, endIndex, inc]** which increments each element of subarray **A[startIndex ... endIndex]** (startIndex and endIndex inclusive) with **inc**.
Return the modified array after all ***k*** operations were executed.
Example:

```
Given:
    length = 5,
    updates = [
        [1,  3,  2],
        [2,  4,  3],
        [0,  2, -2]]
Output:
    [-2, 0, 3, 5, 3]
```

Explanation:

```
Initial state:
[ 0, 0, 0, 0, 0 ]
After applying operation [1, 3, 2]:
[ 0, 2, 2, 2, 0 ]
After applying operation [2, 4, 3]:
[ 0, 2, 5, 5, 3 ]
After applying operation [0, 2, -2]:
[-2, 0, 3, 5, 3 ]
```

```java
public class Solution {
    public int[] getModifiedArray(int length, int[][] updates) {
        int[] res = new int[length];
        for (int[] update : updates) {
            int value = update[2];
            int start = update[0];
            int end = update[1];
            res[start] += value;
            if (end < length - 1)
                res[end + 1] -= value;
        }
        int sum = 0;
        for (int i = 0; i < length; i++) {
            sum += res[i];
            res[i] = sum;
        }
        return res;
    }
}
```

371.Sum of Two Integers

Calculate the sum of two integers *a* and *b*, but you are **not allowed** to use the operator `+` and `-`.
Example:
Given *a* = 1 and *b* = 2, return 3.

```java
public class Solution {
    public int getSum(int a, int b) {
        if (a == 0)
            return b;
        if (b == 0)
            return a;
        while (b != 0) {
            int carry = a & b;
            a = a ^ b;
            b = carry << 1;
        }
        return a;
    }
}
```

372.Super Pow

Your task is to calculate a^b mod 1337 where *a* is a positive integer and *b* is an extremely large positive integer given in the form of an array.
Example1:

```
a = 2
b = [3]
Result: 8
```

Example2:

```
a = 2
b = [1,0]
```

```
Result: 1024
```

```java
public class Solution {
    public int superPow(int a, int[] b) {
        if (a % 1337 == 0)
            return 0;
        int p = 0;
        for (int i : b)
            p = (p * 10 + i) % 1140;// 6*190=1140, 7*191-1337
        if (p == 0)
            p += 1440;
        return power(a, p, 1337);
    }
    public int power(int a, int n, int mod) {
        a %= mod;
        int ret = 1;
        while (n != 0) {
            if ((n & 1) != 0)
                ret = ret * a % mod;
            a = a * a % mod;
            n >>= 1;
        }
        return ret;
    }
}
```

373.Find K Pairs with Smallest Sums

You are given two integer arrays **nums1** and **nums2** sorted in ascending order and an integer **k**.
Define a pair **(u,v)** which consists of one element from the first array and one element from the second array.
Find the k pairs $(u_1,v_1),(u_2,v_2)$ ...(u_k,v_k) with the smallest sums.

Example 1:

```
Given nums1 = [1,7,11], nums2 = [2,4,6],  k = 3
Return: [1,2],[1,4],[1,6]
The first 3 pairs are returned from the sequence:
[1,2],[1,4],[1,6],[7,2],[7,4],[11,2],[7,6],[11,4],[11,6]
```

Example 2:

```
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]
```

Example 3:

```
Given nums1 = [1,2], nums2 = [3],  k = 3
Return: [1,3],[2,3]
All possible pairs are returned from the sequence:
[1,3],[2,3]
```

```java
public class Solution {
    public List<int[]> kSmallestPairs(int[] nums1, int[] nums2, int k) {
        PriorityQueue<int[]> que = new PriorityQueue<>(
                (a, b) -> a[0] + a[1] - b[0] - b[1]);
        List<int[]> res = new ArrayList<>();
        if (nums1.length == 0 || nums2.length == 0 || k == 0)
            return res;
        for (int i = 0; i < nums1.length && i < k; i++)
            que.offer(new int[] { nums1[i], nums2[0], 0 });
        while (k-- > 0 && !que.isEmpty()) {
            int[] cur = que.poll();
            res.add(new int[] { cur[0], cur[1] });
            if (cur[2] == nums2.length - 1)
                continue;
            que.offer(new int[] { cur[0], nums2[cur[2] + 1], cur[2] + 1 });
        }
        return res;
    }
}
```

374.Guess Number Higher or Lower

We are playing the Guess Game. The game is as follows:
I pick a number from **1** to ***n***. You have to guess which number I picked.
Every time you guess wrong, I'll tell you whether the number is higher or lower.
You call a pre-defined API `guess(int num)` which returns 3 possible results (`-1`, `1`, or `0`):

```
-1 : My number is lower
```

```
 0 : Congrats! You got it!
```

Example:

```
n = 10, I pick 6.
Return 6.
```

```
public class Solution extends GuessGame {
    public int guessNumber(int n) {
        int low = 1;
        int high = n;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            int res = guess(mid);
            if (res == 0)
                return mid;
            else if (res < 0)
                high = mid - 1;
            else
                low = mid + 1;
        }
        return -1;
    }
}
```

375.Guess Number Higher or Lower II

We are playing the Guess Game. The game is as follows:
I pick a number from **1** to **n**. You have to guess which number I picked.
Every time you guess wrong, I'll tell you whether the number I picked is higher or lower.
However, when you guess a particular number x, and you guess wrong, you pay **$x**. You win the game when you guess the number I picked.

Example:

```
n = 10, I pick 8.
First round:  You guess 5, I tell you that it's higher. You pay $5.
Second round: You guess 7, I tell you that it's higher. You pay $7.
Third round:  You guess 9, I tell you that it's lower. You pay $9.
Game over. 8 is the number I picked.
You end up paying $5 + $7 + $9 = $21.
```

Given a particular **n ≥ 1**, find out how much money you need to have to guarantee a **win**.

```
public class Solution {
    public int getMoneyAmount(int n) {
        int[][] dp = new int[n + 1][n + 1];
        for (int len = 2; len <= n; len++) {
            for (int start = 1; start <= n - len + 1; start++) {
                int minres = Integer.MAX_VALUE;
                for (int piv = start + (len - 1) / 2; piv < start + len
                        - 1; piv++) {
                    int res = piv + Math.max(dp[start][piv - 1],
                            dp[piv + 1][start + len - 1]);
                    minres = Math.min(res, minres);
                }
                dp[start][start + len - 1] = minres;
            }
        }
        return dp[1][n];
    }
}
```

376.Wiggle Subsequence

A sequence of numbers is called a **wiggle sequence** if the differences between successive numbers strictly alternate between positive and negative. The first difference (if one exists) may be either positive or negative. A sequence with fewer than two elements is trivially a wiggle sequence.
For example, `[1,7,4,9,2,5]` is a wiggle sequence because the differences (6,-3,5,-7,3) are alternately positive and negative. In contrast, `[1,4,7,2,5]` and `[1,7,4,5,5]` are not wiggle sequences, the first because its first two differences are positive and the second because its last difference is zero.
Given a sequence of integers, return the length of the longest subsequence that is a wiggle sequence. A subsequence is obtained by deleting some number of elements (eventually, also zero) from the original sequence, leaving the remaining elements in their original order.

Examples:

```
Input: [1,7,4,9,2,5]
Output: 6
The entire sequence is a wiggle sequence.
Input: [1,17,5,10,13,15,10,5,16,8]
Output: 7
```

```
Input: [1,2,3,4,5,6,7,8,9]
Output: 2
```

Follow up:
Can you do it in O(*n*) time?

```
public class Solution {
    public int wiggleMaxLength(int[] nums) {
        if (nums.length < 2)
            return nums.length;
        int down = 1, up = 1;
        for (int i = 1; i < nums.length; i++) {
            if (nums[i] > nums[i - 1])
                up = down + 1;
            else if (nums[i] < nums[i - 1])
                down = up + 1;
        }
        return Math.max(down, up);
    }
}
```

```
public class Solution {
    public int wiggleMaxLength(int[] nums) {
        if (nums.length < 2)
            return nums.length;
        int[] up = new int[nums.length];
        int[] down = new int[nums.length];
        up[0] = down[0] = 1;
        for (int i = 1; i < nums.length; i++) {
            if (nums[i] > nums[i - 1]) {
                up[i] = down[i - 1] + 1;
                down[i] = down[i - 1];
            } else if (nums[i] < nums[i - 1]) {
                down[i] = up[i - 1] + 1;
                up[i] = up[i - 1];
            } else {
                down[i] = down[i - 1];
                up[i] = up[i - 1];
            }
        }
        return Math.max(down[nums.length - 1], up[nums.length - 1]);
    }
}
```

377.Combination Sum IV

Given an integer array with all positive numbers and no duplicates, find the number of possible combinations that add up to a positive integer target.

Example:

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

Follow up:
What if negative numbers are allowed in the given array?
How does it change the problem?
What limitation we need to add to the question to allow negative numbers?

```
public class Solution {
    private int[] dp;
    public int combinationSum4(int[] nums, int target) {
        dp = new int[target + 1];
        Arrays.fill(dp, -1);
        dp[0] = 1;
        return helper(nums, target);
    }
    private int helper(int[] nums, int target) {
        if (dp[target] != -1)
            return dp[target];
        int res = 0;
        for (int i = 0; i < nums.length; i++)
            if (target >= nums[i])
```

```
                res += helper(nums, target - nums[i]);
        dp[target] = res;
        return res;
    }
}
```

378.Kth Smallest Element in a Sorted Matrix

Given a *n* x *n* matrix where each of the rows and columns are sorted in ascending order, find the kth smallest element in the matrix.
Note that it is the kth smallest element in the sorted order, not the kth distinct element.

Example:

```
matrix = [
   [ 1,  5,  9],
   [10, 11, 13],
   [12, 13, 15]],
k = 8,
return 13.
```

Note:

You may assume k is always valid, 1 ? k ? n^2.

```
public class Solution {
    public int kthSmallest(int[][] matrix, int k) {
        int lo = matrix[0][0],
                hi = matrix[matrix.length - 1][matrix[0].length - 1] + 1;
        while (lo < hi) {
            int mid = lo + (hi - lo) / 2;
            int count = 0, j = matrix[0].length - 1;
            for (int i = 0; i < matrix.length; i++) {
                while (j >= 0 && matrix[i][j] > mid)
                    j--;
                count += (j + 1);
            }
            if (count < k)
                lo = mid + 1;
            else
                hi = mid;
        }
        return lo;
    }
}
```

379.Design Phone Directory

Design a Phone Directory which supports the following operations:

1. `get`: Provide a number which is not assigned to anyone.
2. `check`: Check if a number is available or not.
3. `release`: Recycle or release a number.

Example:

```
// Init a phone directory containing a total of 3 numbers: 0, 1, and 2.
PhoneDirectory directory = new PhoneDirectory(3);
// It can return any available phone number. Here we assume it returns 0.
directory.get();
// Assume it returns 1.
directory.get();
// The number 2 is available, so return true.
directory.check(2);
// It returns 2, the only number that is left.
directory.get();
// The number 2 is no longer available, so return false.
directory.check(2);
// Release number 2 back to the pool.
directory.release(2);
// Number 2 is available again, return true.
directory.check(2);
```

```
public class PhoneDirectory {
    Set<Integer> used = new HashSet<Integer>();
    Queue<Integer> available = new LinkedList<Integer>();
    int max;
    public PhoneDirectory(int maxNumbers) {
        max = maxNumbers;
        for (int i = 0; i < maxNumbers; i++)
            available.offer(i);
    }
    public int get() {
        Integer ret = available.poll();
        if (ret == null)
            return -1;
        used.add(ret);
```

```
        return ret;
    }
    public boolean check(int number) {
        if (number >= max || number < 0)
            return false;
        return !used.contains(number);
    }
    public void release(int number) {
        if (used.remove(number))
            available.offer(number);
    }
}
```

380.Insert Delete GetRandom O(1)

Design a data structure that supports all following operations in *average* **O(1)** time.

1. `insert(val)`: Inserts an item val to the set if not already present.
2. `remove(val)`: Removes an item val from the set if present.
3. `getRandom`: Returns a random element from current set of elements. Each element must have the **same probability** of being returned.

Example:

```
// Init an empty set.
RandomizedSet randomSet = new RandomizedSet();
// Inserts 1 to the set. Returns true as 1 was inserted successfully.
randomSet.insert(1);
// Returns false as 2 does not exist in the set.
randomSet.remove(2);
// Inserts 2 to the set, returns true. Set now contains [1,2].
randomSet.insert(2);
// getRandom should return either 1 or 2 randomly.
randomSet.getRandom();
// Removes 1 from the set, returns true. Set now contains [2].
randomSet.remove(1);
// 2 was already in the set, so return false.
randomSet.insert(2);
// Since 2 is the only number in the set, getRandom always return 2.
randomSet.getRandom();
```

```
public class RandomizedSet {
    ArrayList<Integer> nums;
    HashMap<Integer, Integer> locs;
    java.util.Random rand = new java.util.Random();
    public RandomizedSet() {
        nums = new ArrayList<Integer>();
        locs = new HashMap<Integer, Integer>();
    }
    public boolean insert(int val) {
        boolean contain = locs.containsKey(val);
        if (contain)
            return false;
        locs.put(val, nums.size());
        nums.add(val);
        return true;
    }
    public boolean remove(int val) {
        boolean contain = locs.containsKey(val);
        if (!contain)
            return false;
        int loc = locs.get(val);
        if (loc < nums.size() - 1) {
            int lastone = nums.get(nums.size() - 1);
            nums.set(loc, lastone);
            locs.put(lastone, loc);
        }
        locs.remove(val);
        nums.remove(nums.size() - 1);
        return true;
    }
    public int getRandom() {
        return nums.get(rand.nextInt(nums.size()));
    }
}
```

381.Insert Delete GetRandom O(1) - Duplicates allowed

Design a data structure that supports all following operations in *average* **O(1)** time.

Note: Duplicate elements are allowed.

1. `insert(val)`: Inserts an item val to the collection.
2. `remove(val)`: Removes an item val from the collection if present.

3. `getRandom`: Returns a random element from current collection of elements. The probability of each element being returned is **linearly related** to the number of same value the collection contains.

Example:

```
// Init an empty collection.
RandomizedCollection collection = new RandomizedCollection();
// Inserts 1 to the collection. Returns true as the collection did not contain 1.
collection.insert(1);
// Inserts another 1 to the collection. Returns false as the collection contained 1. Collection now contains
[1,1].
collection.insert(1);
// Inserts 2 to the collection, returns true. Collection now contains [1,1,2].
collection.insert(2);
// getRandom should return 1 with the probability 2/3, and returns 2 with the probability 1/3.
collection.getRandom();
// Removes 1 from the collection, returns true. Collection now contains [1,2].
collection.remove(1);
// getRandom should return 1 and 2 both equally likely.
collection.getRandom();
```

```
public class RandomizedCollection {
    ArrayList<Integer> nums;
    HashMap<Integer, Set<Integer>> locs;
    java.util.Random rand = new java.util.Random();
    public RandomizedCollection() {
        nums = new ArrayList<Integer>();
        locs = new HashMap<Integer, Set<Integer>>();
    }
    public boolean insert(int val) {
        boolean contain = locs.containsKey(val);
        if (!contain)
            locs.put(val, new LinkedHashSet<Integer>());
        locs.get(val).add(nums.size());
        nums.add(val);
        return !contain;
    }
    public boolean remove(int val) {
        boolean contain = locs.containsKey(val);
        if (!contain)
            return false;
        int loc = locs.get(val).iterator().next();
        locs.get(val).remove(loc);
        if (loc < nums.size() - 1) {
            int lastone = nums.get(nums.size() - 1);
            nums.set(loc, lastone);
            locs.get(lastone).remove(nums.size() - 1);
            locs.get(lastone).add(loc);
        }
        nums.remove(nums.size() - 1);
        if (locs.get(val).isEmpty())
            locs.remove(val);
        return true;
    }
    public int getRandom() {
        return nums.get(rand.nextInt(nums.size()));
    }
}
```

382.Linked List Random Node

Given a singly linked list, return a random node's value from the linked list. Each node must have the **same probability** of being chosen.

Follow up:

What if the linked list is extremely large and its length is unknown to you? Could you solve this efficiently without using extra space?

Example:

```
// Init a singly linked list [1,2,3].
ListNode head = new ListNode(1);
head.next = new ListNode(2);
head.next.next = new ListNode(3);
Solution solution = new Solution(head);
// getRandom() should return either 1, 2, or 3 randomly. Each element should have equal probability of
returning.
solution.getRandom();
```

```
public class Solution {
    ListNode head = null;
    Random r = new Random();
    public Solution(ListNode head) {
```

```
    }
    public int getRandom() {
        int result = this.head.val;
        ListNode node = this.head.next;
        int k = 1;
        int i = 1;
        while (node != null) {
            double x = r.nextDouble();
            double y = k / (k + i * 1.0);
            if (x <= y) {
                result = node.val;
            }
            i++;
            node = node.next;
        }
        return result;
    }
}
```

383.Ransom Note

Given an arbitrary ransom note string and another string containing letters from all the magazines, write a function that will return true if the ransom note can be constructed from the magazines ; otherwise, it will return false.
Each letter in the magazine string can only be used once in your ransom note.
Note:
You may assume that both strings contain only lowercase letters.

```
canConstruct("a", "b") -> false
canConstruct("aa", "ab") -> false
canConstruct("aa", "aab") -> true
```

```
public class Solution {
    public boolean canConstruct(String ransomNote, String magazine) {
        int[] arr = new int[26];
        for (int i = 0; i < magazine.length(); i++)
            arr[magazine.charAt(i) - 'a']++;
        for (int i = 0; i < ransomNote.length(); i++)
            if (--arr[ransomNote.charAt(i) - 'a'] < 0)
                return false;
        return true;
    }
}
```

384.Shuffle an Array

Shuffle a set of numbers without duplicates.
Example:

```
// Init an array with set 1, 2, and 3.
int[] nums = {1,2,3};
Solution solution = new Solution(nums);
// Shuffle the array [1,2,3] and return its result. Any permutation of [1,2,3] must equally likely to be
returned.
solution.shuffle();
// Resets the array back to its original configuration [1,2,3].
solution.reset();
// Returns the random shuffling of array [1,2,3].
solution.shuffle();
```

```
public class Solution {
    private int[] nums;
    private Random random;
    public Solution(int[] nums) {
        this.nums = nums;
        random = new Random();
    }

    public int[] reset() {
        return nums;
    }

    public int[] shuffle() {
        if(nums == null) return null;
        int[] a = nums.clone();
        for(int j = 1; j < a.length; j++) {
            int i = random.nextInt(j + 1);
            swap(a, i, j);
        }
        return a;
    }
```

```
    private void swap(int[] a, int i, int j) {
        int t = a[i];
        a[i] = a[j];
        a[j] = t;
    }
}
```

385.Mini Parser

Given a nested list of integers represented as a string, implement a parser to deserialize it.
Each element is either an integer, or a list -- whose elements may also be integers or other lists.
Note: You may assume that the string is well-formed:

- String is non-empty.
- String does not contain white spaces.
- String contains only digits `0-9`, `[`, `-`, `,`, `]`.

Example 1:

```
Given s = "324",
You should return a NestedInteger object which contains a single integer 324.
```

Example 2:

```
Given s = "[123,[456,[789]]]",
Return a NestedInteger object containing a nested list with 2 elements:
1. An integer containing value 123.
2. A nested list containing two elements:
    i.  An integer containing value 456.
    ii. A nested list with one element:
         a. An integer containing value 789.
```

```
public class Solution {
    public NestedInteger deserialize(String s) {
        if (s.isEmpty())
            return null;
        if (s.charAt(0) != '[')
            return new NestedInteger(Integer.valueOf(s));
        Stack<NestedInteger> stack = new Stack<>();
        NestedInteger curr = null;
        int l = 0;
        for (int r = 0; r < s.length(); r++) {
            char ch = s.charAt(r);
            if (ch == '[') {
                if (curr != null) {
                    stack.push(curr);
                }
                curr = new NestedInteger();
                l = r + 1;
            } else if (ch == ']') {
                String num = s.substring(l, r);
                if (!num.isEmpty())
                    curr.add(new NestedInteger(Integer.valueOf(num)));
                if (!stack.isEmpty()) {
                    NestedInteger pop = stack.pop();
                    pop.add(curr);
                    curr = pop;
                }
                l = r + 1;
            } else if (ch == ',') {
                if (s.charAt(r - 1) != ']') {
                    String num = s.substring(l, r);
                    curr.add(new NestedInteger(Integer.valueOf(num)));
                }
                l = r + 1;
            }
        }
        return curr;
    }
}
```

386.Lexicographical Numbers

Given an integer *n*, return 1 - *n* in lexicographical order.
For example, given 13, return: [1,10,11,12,13,2,3,4,5,6,7,8,9].
Please optimize your algorithm to use less time and space. The input size may be as large as 5,000,000.

```
public class Solution {
    public List<Integer> lexicalOrder(int n) {
        List<Integer> list = new ArrayList<>(n);
        int curr = 1;
        for (int i = 1; i <= n; i++) {
            list.add(curr);
```

```
            if (curr * 10 <= n) {
                curr *= 10;
            } else if (curr % 10 != 9 && curr + 1 <= n) {
                curr++;
            } else {
                while ((curr / 10) % 10 == 9) {
                    curr /= 10;
                }
                curr = curr / 10 + 1;
            }
        }
        return list;
    }
}
```

387.First Unique Character in a String

Given a string, find the first non-repeating character in it and return it's index. If it doesn't exist, return -1.

Examples:

```
s = "leetcode"
return 0.
s = "loveleetcode",
return 2.
```

Note: You may assume the string contain only lowercase letters.

```
public class Solution {
    public int firstUniqChar(String s) {
        int freq[] = new int[26];
        for (int i = 0; i < s.length(); i++)
            freq[s.charAt(i) - 'a']++;
        for (int i = 0; i < s.length(); i++)
            if (freq[s.charAt(i) - 'a'] == 1)
                return i;
        return -1;
    }
}
```

388.Longest Absolute File Path

Suppose we abstract our file system by a string in the following manner:
The string `"dir\n\tsubdir1\n\tsubdir2\n\t\tfile.ext"` represents:

```
dir
    subdir1
    subdir2
        file.ext
```

The directory `dir` contains an empty sub-directory `subdir1` and a sub-directory `subdir2` containing a file `file.ext`.
The string `"dir\n\tsubdir1\n\t\tfile1.ext\n\t\tsubsubdir1\n\tsubdir2\n\t\tsubsubdir2\n\t\t\tfile2.ext"` represents:

```
dir
    subdir1
        file1.ext
        subsubdir1
    subdir2
        subsubdir2
            file2.ext
```

The directory `dir` contains two sub-directories `subdir1` and `subdir2`. `subdir1` contains a file `file1.ext` and an empty second-level sub-directory `subsubdir1`. `subdir2` contains a second-level sub-directory `subsubdir2` containing a file `file2.ext`.
We are interested in finding the longest (number of characters) absolute path to a file within our file system. For example, in the second example above, the longest absolute path is `"dir/subdir2/subsubdir2/file2.ext"`, and its length is `32` (not including the double quotes).
Given a string representing the file system in the above format, return the length of the longest absolute path to file in the abstracted file system. If there is no file in the system, return `0`.

Note:

- The name of a file contains at least a `.` and an extension.
- The name of a directory or sub-directory will not contain a `.`.

Time complexity required: `O(n)` where `n` is the size of the input string.
Notice that `a/aa/aaa/file1.txt` is not the longest file path, if there is another path `aaaaaaaaaaaaaaaaaaaa/sth.png`.

```
public class Solution {
    public int lengthLongestPath(String input) {
        Stack<Integer> stack = new Stack<>();
        stack.push(0);
        int maxLen = 0;
        for (String s : input.split("\n")) {
            int lev = s.lastIndexOf("\t") + 1;
            while (lev + 1 < stack.size())
                stack.pop();
```

```
            int len = stack.peek() + s.length() - lev + 1;
            stack.push(len);
            if (s.contains("."))
                maxLen = Math.max(maxLen, len - 1);
        }
        return maxLen;
    }
}
```

389.Find the Difference

Given two strings ***s*** and ***t*** which consist of only lowercase letters.
String ***t*** is generated by random shuffling string ***s*** and then add one more letter at a random position.
Find the letter that was added in ***t***.
Example:

```
Input:
s = "abcd"
t = "abcde"
Output:e
Explanation:
'e' is the letter that was added.
```

```
public class Solution {
    public char findTheDifference(String s, String t) {
        int charCodeS = 0, charCodeT = 0;
        for (int i = 0; i < s.length(); ++i)
            charCodeS += (int) s.charAt(i);
        for (int i = 0; i < t.length(); ++i)
            charCodeT += (int) t.charAt(i);
        return (char) (charCodeT - charCodeS);
    }
}
```

390.Elimination Game

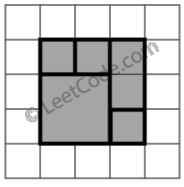
There is a list of sorted integers from 1 to *n*. Starting from left to right, remove the first number and every other number afterward until you reach the end of the list.
Repeat the previous step again, but this time from right to left, remove the right most number and every other number from the remaining numbers.
We keep repeating the steps again, alternating left to right and right to left, until a single number remains.
Find the last number that remains starting with a list of length *n*.
Example:

```
Input:
n = 9,
1 2 3 4 5 6 7 8 9
2 4 6 8
2 6
6
Output:6
```

```
public class Solution {
    public int lastRemaining(int n) {
        boolean left = true;
        int remaining = n;
        int step = 1;
        int head = 1;
        while (remaining > 1) {
            if (left || remaining % 2 == 1)
                head = head + step;
            remaining = remaining / 2;
            step = step * 2;
            left = !left;
        }
        return head;
    }
}
```

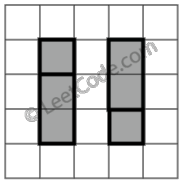
391.Perfect Rectangle

Given N axis-aligned rectangles where N > 0, determine if they all together form an exact cover of a rectangular region.
Each rectangle is represented as a bottom-left point and a top-right point. For example, a unit square is represented as [1,1,2,2]. (coordinate of bottom-left point is (1, 1) and top-right point is (2, 2)).



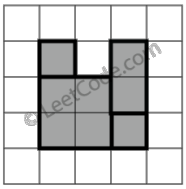
Example 1:

```
rectangles = [
  [1,1,3,3],
  [3,1,4,2],
  [3,2,4,4],
  [1,3,2,4],
  [2,3,3,4]]
Return true. All 5 rectangles together form an exact cover of a rectangular region.
```



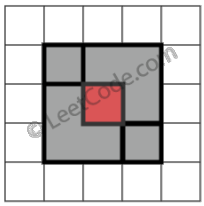
Example 2:

```
rectangles = [
  [1,1,2,3],
  [1,3,2,4],
  [3,1,4,2],
  [3,2,4,4]]
Return false. Because there is a gap between the two rectangular regions.
```



Example 3:

```
rectangles = [
  [1,1,3,3],
  [3,1,4,2],
  [1,3,2,4],
  [3,2,4,4]]
Return false. Because there is a gap in the top center.
```



Example 4:

```
rectangles = [
  [1,1,3,3],
  [3,1,4,2],
  [1,3,2,4],
  [2,2,4,4]]
Return false. Because two of the rectangles overlap with each other.
```

```java
public class Solution {
    public boolean isRectangleCover(int[][] rectangles) {
        if (rectangles.length == 0 || rectangles[0].length == 0)
            return false;
        int x1 = Integer.MAX_VALUE;
        int x2 = Integer.MIN_VALUE;
        int y1 = Integer.MAX_VALUE;
        int y2 = Integer.MIN_VALUE;
        HashSet<String> set = new HashSet<String>();
        int area = 0;
        for (int[] rect : rectangles) {
            x1 = Math.min(rect[0], x1);
            y1 = Math.min(rect[1], y1);
            x2 = Math.max(rect[2], x2);
            y2 = Math.max(rect[3], y2);
            area += (rect[2] - rect[0]) * (rect[3] - rect[1]);
            String s1 = rect[0] + " " + rect[1];
```

```
            String s2 = rect[0] + " " + rect[3];
            String s3 = rect[2] + " " + rect[3];
            String s4 = rect[2] + " " + rect[1];
            if (!set.add(s1))
                set.remove(s1);
            if (!set.add(s2))
                set.remove(s2);
            if (!set.add(s3))
                set.remove(s3);
            if (!set.add(s4))
                set.remove(s4);
        }
        if (!set.contains(x1 + " " + y1) || !set.contains(x1 + " " + y2)
                || !set.contains(x2 + " " + y1) || !set.contains(x2 + " " + y2)
                || set.size() != 4)
            return false;
        return area == (x2 - x1) * (y2 - y1);
    }
}
```

392.Is Subsequence

Given a string **s** and a string **t**, check if **s** is subsequence of **t**.
You may assume that there is only lower case English letters in both **s** and **t**. **t** is potentially a very long (length ~= 500,000) string, and **s**is a short string (<=100).
A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, `"ace"` is a subsequence of `"abcde"` while `"aec"` is not).
Example 1:
s = `"abc"`, **t** = `"ahbgdc"`
Return `true`.
Example 2:
s = `"axc"`, **t** = `"ahbgdc"`
Return `false`.
Follow up:
If there are lots of incoming S, say S1, S2, ... , Sk where k >= 1B, and you want to check one by one to see if T has its subsequence. In this scenario, how would you change your code?

```
public class Solution {
    public boolean isSubsequence(String s, String t) {
        if (s.length() == 0)
            return true;
        int indexS = 0, indexT = 0;
        while (indexT < t.length()) {
            if (t.charAt(indexT) == s.charAt(indexS)) {
                indexS++;
                if (indexS == s.length())
                    return true;
            }
            indexT++;
        }
        return false;
    }
}
```

393.UTF-8 Validation

A character in UTF8 can be from **1 to 4 bytes** long, subjected to the following rules:

1. For 1-byte character, the first bit is a 0, followed by its unicode code.
2. For n-bytes character, the first n-bits are all one's, the n+1 bit is 0, followed by n-1 bytes with most significant 2 bits being 10.

This is how the UTF-8 encoding would work:

```
   Char. number range  |        UTF-8 octet sequence
      (hexadecimal)    |              (binary)
   --------------------+---------------------------------------------
   0000 0000-0000 007F | 0xxxxxxx
   0000 0080-0000 07FF | 110xxxxx 10xxxxxx
   0000 0800-0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx
   0001 0000-0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
```

Given an array of integers representing the data, return whether it is a valid utf-8 encoding.
Note:
The input is an array of integers. Only the **least significant 8 bits** of each integer is used to store the data. This means each integer represents only 1 byte of data.
Example 1:

```
data = [197, 130, 1], which represents the octet sequence: 11000101 10000010 00000001.
Return true.
It is a valid utf-8 encoding for a 2-bytes character followed by a 1-byte character.
```

```
data = [235, 140, 4], which represented the octet sequence: 11101011 10001100 00000100.
Return false.
The first 3 bits are all one's and the 4th bit is 0 means it is a 3-bytes character.
The next byte is a continuation byte which starts with 10 and that's correct.
But the second continuation byte does not start with 10, so it is invalid.
```

```
public class Solution {
    public boolean validUtf8(int[] data) {
        if (data == null || data.length == 0)
            return false;
        boolean isValid = true;
        for (int i = 0; i < data.length; i++) {
            if (data[i] > 255)
                return false;
            int numberOfBytes = 0;
            if ((data[i] & 128) == 0) {
                numberOfBytes = 1;
            } else if ((data[i] & 224) == 192) {
                numberOfBytes = 2;
            } else if ((data[i] & 240) == 224) {
                numberOfBytes = 3;
            } else if ((data[i] & 248) == 240) {
                numberOfBytes = 4;
            } else {
                return false;
            }
            for (int j = 1; j < numberOfBytes; j++) {
                if (i + j >= data.length)
                    return false;
                if ((data[i + j] & 192) != 128)
                    return false;
            }
            i = i + numberOfBytes - 1;
        }
        return isValid;
    }
}
```

394.Decode String

Given an encoded string, return it's decoded string.
The encoding rule is: `k[encoded_string]`, where the *encoded_string* inside the square brackets is being repeated exactly *k* times.
Note that *k* is guaranteed to be a positive integer.
You may assume that the input string is always valid; No extra white spaces, square brackets are well-formed, etc.
Furthermore, you may assume that the original data does not contain any digits and that digits are only for those repeat numbers, *k*.
For example, there won't be input like `3a` or `2[4]`.
Examples:

```
s = "3[a]2[bc]", return "aaabcbc".
s = "3[a2[c]]", return "accaccacc".
s = "2[abc]3[cd]ef", return "abcabccdcdcdef".
```

```
public class Solution {
    public String decodeString(String s) {
        String res = "";
        Stack<Integer> countStack = new Stack<>();
        Stack<String> resStack = new Stack<>();
        int idx = 0;
        while (idx < s.length()) {
            if (Character.isDigit(s.charAt(idx))) {
                int count = 0;
                while (Character.isDigit(s.charAt(idx))) {
                    count = 10 * count + (s.charAt(idx) - '0');
                    idx++;
                }
                countStack.push(count);
            } else if (s.charAt(idx) == '[') {
                resStack.push(res);
                res = "";
                idx++;
            } else if (s.charAt(idx) == ']') {
                StringBuilder temp = new StringBuilder(resStack.pop());
                int repeatTimes = countStack.pop();
                for (int i = 0; i < repeatTimes; i++) {
                    temp.append(res);
                }
                res = temp.toString();
                idx++;
```

```
                res += s.charAt(idx++);
            }
        }
        return res;
    }
}
```

395.Longest Substring with At Least K Repeating Characters

Find the length of the longest substring ***T*** of a given string (consists of lowercase letters only) such that every character in ***T*** appears no less than *k* times.

Example 1:

```
Input:s = "aaabb", k = 3
Output:3
The longest substring is "aaa", as 'a' is repeated 3 times.
```

Example 2:

```
Input:s = "ababbc", k = 2
Output:5
The longest substring is "ababb", as 'a' is repeated 2 times and 'b' is repeated 3 times.
```

```
public class Solution {
    public int longestSubstring(String s, int k) {
        char[] str = s.toCharArray();
        int[] counts = new int[26];
        int h, i, j, idx, max = 0, unique, noLessThanK;
        for (h = 1; h <= 26; h++) {
            Arrays.fill(counts, 0);
            i = 0;
            j = 0;
            unique = 0;
            noLessThanK = 0;
            while (j < str.length) {
                if (unique <= h) {
                    idx = str[j] - 'a';
                    if (counts[idx] == 0)
                        unique++;
                    counts[idx]++;
                    if (counts[idx] == k)
                        noLessThanK++;
                    j++;
                } else {
                    idx = str[i] - 'a';
                    if (counts[idx] == k)
                        noLessThanK--;
                    counts[idx]--;
                    if (counts[idx] == 0)
                        unique--;
                    i++;
                }
                if (unique == h && unique == noLessThanK)
                    max = Math.max(j - i, max);
            }
        }
        return max;
    }
}
```

396.Rotate Function

Given an array of integers `A` and let *n* to be its length.

Assume B_k to be an array obtained by rotating the array `A` *k* positions clock-wise, we define a "rotation function" `F` on `A` as follow: $F(k) = 0 * B_k[0] + 1 * B_k[1] + ... + (n-1) * B_k[n-1]$.

Calculate the maximum value of `F(0), F(1), ..., F(n-1)`.

Note:

n is guaranteed to be less than 10^5.

Example:

```
A = [4, 3, 2, 6]
F(0) = (0 * 4) + (1 * 3) + (2 * 2) + (3 * 6) = 0 + 3 + 4 + 18 = 25
F(1) = (0 * 6) + (1 * 4) + (2 * 3) + (3 * 2) = 0 + 4 + 6 + 6 = 16
F(2) = (0 * 2) + (1 * 6) + (2 * 4) + (3 * 3) = 0 + 6 + 8 + 9 = 23
F(3) = (0 * 3) + (1 * 2) + (2 * 6) + (3 * 4) = 0 + 2 + 12 + 12 = 26
So the maximum value of F(0), F(1), F(2), F(3) is F(3) = 26.
```

```
public class Solution {
    public int maxRotateFunction(int[] A) {
        if (A.length == 0)
            return 0;
```

```
        int sum = 0, iteration = 0, len = A.length;
        for (int i = 0; i < len; i++) {
            sum += A[i];
            iteration += (A[i] * i);
        }
        int max = iteration;
        for (int j = 1; j < len; j++) {
            iteration = iteration - sum + A[j - 1] * len;
            max = Math.max(max, iteration);
        }
        return max;
    }
}
```

397.Integer Replacement

Given a positive integer *n* and you can do operations as follow:

1. If *n* is even, replace *n* with `n/2`.
2. If *n* is odd, you can replace *n* with either `n + 1` or `n - 1`.

What is the minimum number of replacements needed for *n* to become 1?

Example 1:

```
Input:8
Output:3
Explanation:
8 -> 4 -> 2 -> 1
```

Example 2:

```
Input:7
Output:4
Explanation:
7 -> 8 -> 4 -> 2 -> 1 or 7 -> 6 -> 3 -> 2 -> 1
```

```
public class Solution {
    public int integerReplacement(int n) {
        if (n == Integer.MAX_VALUE)
            return 32;
        int count = 0;
        while (n > 1) {
            if (n % 2 == 0)
                n /= 2;
            else {
                if ((n + 1) % 4 == 0 && (n - 1 != 2))
                    n++;
                else
                    n--;
            }
            count++;
        }
        return count;
    }
}
```

398.Random Pick Index

Given an array of integers with possible duplicates, randomly output the index of a given target number. You can assume that the given target number must exist in the array.

Note:

The array size can be very large. Solution that uses too much extra space will not pass the judge.

Example:

```
int[] nums = new int[] {1,2,3,3,3};
Solution solution = new Solution(nums);
// pick(3) should return either index 2, 3, or 4 randomly. Each index should have equal probability of
returning.
solution.pick(3);
// pick(1) should return 0. Since in the array only nums[0] is equal to 1.
solution.pick(1);
```

```
public class Solution {
    int[] nums;
    Random rnd;
    public Solution(int[] nums) {
        this.nums = nums;
        this.rnd = new Random();
    }
    public int pick(int target) {
        int result = -1;
        int count = 0;
        for (int i = 0; i < nums.length; i++) {
```

```
            if (nums[i] != target)
                continue;
            if (rnd.nextInt(++count) == 0)
                result = i;
        }
        return result;
    }
}
```

399.Evaluate Division

Equations are given in the format `A / B = k`, where `A` and `B` are variables represented as strings, and `k` is a real number (floating point number). Given some queries, return the answers. If the answer does not exist, return `-1.0`.

Example:

Given `a / b = 2.0, b / c = 3.0.`
queries are: `a / c = ?, b / a = ?, a / e = ?, a / a = ?, x / x = ? .`
return `[6.0, 0.5, -1.0, 1.0, -1.0 ].`
The input is: `vector<pair<string, string>> equations, vector<double>& values, vector<pair<string, string>> queries` , where `equations.size() == values.size()`, and the values are positive. This represents the equations. Return `vector<double>`.
According to the example above:

```
equations = [ ["a", "b"], ["b", "c"] ],
values = [2.0, 3.0],
queries = [ ["a", "c"], ["b", "a"], ["a", "e"], ["a", "a"], ["x", "x"] ].
```

The input is always valid. You may assume that evaluating the queries will result in no division by zero and there is no contradiction.

```
public class Solution {
    public double[] calcEquation(String[][] equations, double[] values,
            String[][] queries) {
        double[] results = new double[queries.length];
        Map<String, List<String>> graph = buildGraph(equations, values);
        for (int i = 0; i < queries.length; i++) {
            String[] query = queries[i];
            if (query[0].equals(query[1]) && !graph.containsKey(query[0])) {
                results[i] = -1.0;
            } else {
                double result = computeResultDFS(graph, query);
                results[i] = result;
            }
        }
        return results;
    }
    public double computeResultDFS(Map<String, List<String>> graph,
            String[] query) {
        String dividend = query[0];
        String divisor = query[1];
        Set<String> visited = new HashSet<>();
        double result = 1.0;
        result = dfs(dividend, divisor, graph, visited);
        if (result < 0)
            result = -1.;
        return result;
    }
    public double dfs(String start, String end, Map<String, List<String>> graph,
            Set<String> visited) {
        if (start.equals(end))
            return 1.0;
        if (visited.contains(start))
            return -1.0;
        visited.add(start);
        List<String> edgesList = graph.get(start);
        if (edgesList == null || edgesList.isEmpty())
            return -1.0;
        double result = 1.0;
        for (String edge : edgesList) {
            String[] tokens = edge.split(" ");
            String next = tokens[0];
            double val = Double.parseDouble(tokens[1]);
            result = val * dfs(next, end, graph, visited);
            if (result > 0)
                return result;
        }
        return result;
    }
    public Map<String, List<String>> buildGraph(String[][] equations,
            double[] values) {
        Map<String, List<String>> graph = new HashMap<>();
        for (int i = 0; i < equations.length; i++) {
```

```
            String endNode = equations[i][1];
            double value = values[i];
            String endEdge = endNode + " " + value;
            String startEdge = startNode + " " + (1 / value);
            if (graph.containsKey(startNode)) {
                graph.get(startNode).add(endEdge);
            } else {
                List<String> edges = new ArrayList<>();
                edges.add(endEdge);
                graph.put(startNode, edges);
            }
            if (graph.containsKey(endNode)) {
                graph.get(endNode).add(startEdge);
            } else {
                List<String> edges = new ArrayList<>();
                edges.add(startEdge);
                graph.put(endNode, edges);
            }
        }
        return graph;
    }
}
```

400.Nth Digit

Find the n^{th} digit of the infinite integer sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...

Note:

n is positive and will fit within the range of a 32-bit signed integer ($n < 2^{31}$).

Example 1:

```
Input:3
Output:3
```

Example 2:

```
Input:11
Output:0
Explanation:
The 11th digit of the sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ... is a 0, which is part of the number 10.
```

```
public class Solution {
    public int findNthDigit(int n) {
        int len = 1;
        long count = 9;
        int start = 1;
        while (n > len * count) {
            n -= len * count;
            len += 1;
            count *= 10;
            start *= 10;
        }
        start += (n - 1) / len;
        String s = Integer.toString(start);
        return Character.getNumericValue(s.charAt((n - 1) % len));
    }
}
```