目录

CHAPTER 1: String

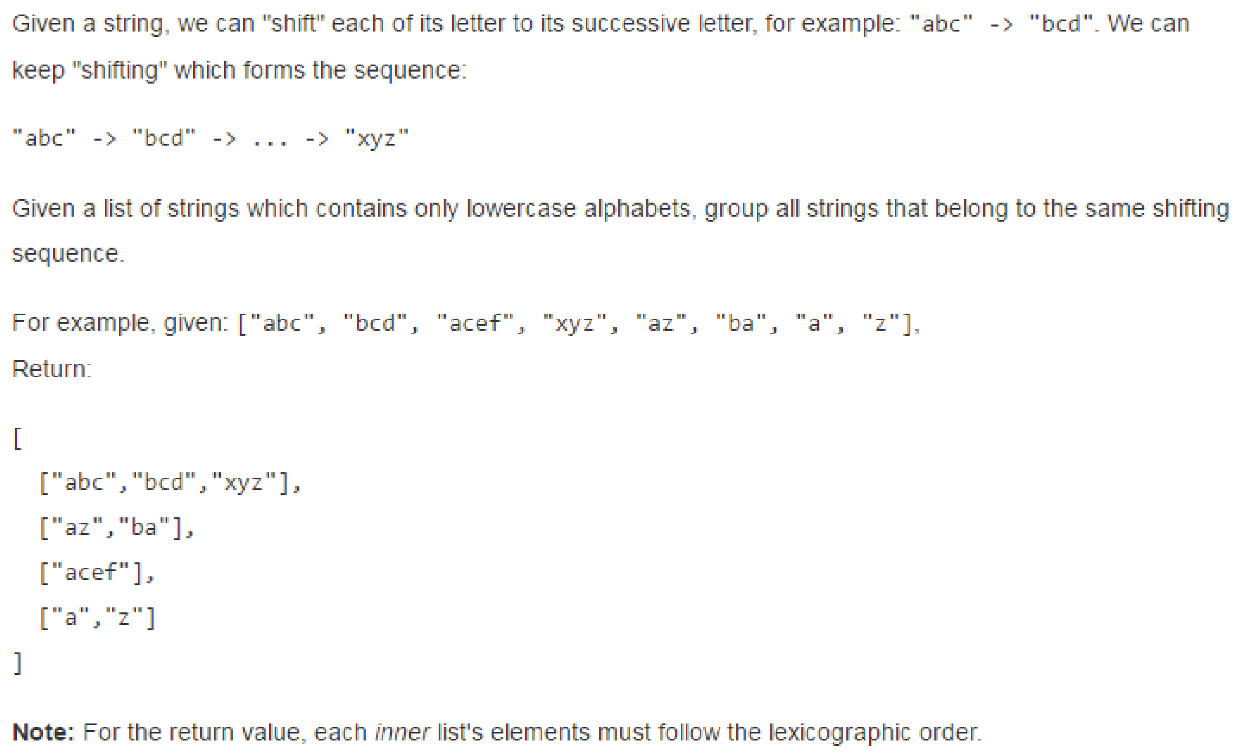
1.1 Read N Character Given Read 4 (Easy->Leetcode No.157)
Solution: See CleanCodeHandbook-1 Question 15

1.2 Read N Characters Given Read 4 II-Call Multiple times (Hard->Leetcode No.158)
Solution: See CleanCodeHandbook-1 Question 16

1.3 Longest Substring with At Most Two Distinct Characters (Hard->Leetcode No.159)
Solution: See CleanCodeHandbook-1 Question 11

1.4 One Edit Distance (Medium->Leetcode No.161)
Solution: See CleanCodeHandbook-1 Question 14

1.5 Reverse Words in a String II (Medium->Leetcode No.186)
Solution: See CleanCodeHandbook-1 Question 7

1.6 Group Shifted String (Easy->Leetcode No.249)


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"]`,
Return:

```
[
  ["abc","bcd","xyz"],
  ["az","ba"],
  ["acef"],
  ["a","z"]
]
```

Note: For the return value, each *inner* list's elements must follow the lexicographic order.

Solution:
```
public class Solution {
public List<List<String>> groupStrings(String[] strings) {
   HashMap<String,List<String>> map=new HashMap<>();
   Arrays.sort(strings);
   for(String s:strings){
      String key=pattern(s);
      if(map.containsKey(key)) map.get(key).add(s);
      else map.put(key,new LinkedList<String>(Arrays.asList(s)));
```

```
    }
    return new LinkedList<List<String>>(map.values());
}
public String pattern(String s){
    StringBuilder ss=new StringBuilder();
    ss.append(0);
    for(int i=0;i<s.length()-1;i++){
       if(s.charAt(i)<s.charAt(i+1)){
           ss.append(s.charAt(i)-s.charAt(i+1)+26);
       }
       else ss.append(s.charAt(i)-s.charAt(i+1));
    }
    return ss.toString();
}
```

1.7 Encode and Decode Strings (Medium->Leetcode No.271)

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.

Solution:

```
public class Codec {

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

  // Decodes a single string to a list of strings.
  public List<String> decode(String s) {
    List<String> res = new ArrayList<>();
    if(s == null || s.length() == 0) {
      return res;
    }

    int index = s.indexOf("#");
    int len = Integer.valueOf(s.substring(0, index));
    String t = s.substring(index + 1, index + 1 + len);
    res.add(t);
    res.addAll(decode(s.substring(index+1+len)));
    return res;
  }
}
```

1.8 Flip Game (Easy->Leetcode No.293)

Problem Description:

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:

```
[
  "--++",
  "+--+",
  "++--"
]
```

Solution:

```
public class Solution {
    public List<String> generatePossibleNextMoves(String s) {
      List<String> result = new ArrayList<>();
      if (s == null || s.length() < 2) {
        return result;
      }

      for (int i = 0; i < s.length() - 1; i++) {
            if (s.charAt(i) == '+' && s.charAt(i + 1) == '+') {
                String s1 = s.substring(0, i);
                String s2 = "--";
                String s3 = s.substring(i + 2);
                String temp = s1 + s2 + s3;
                result.add(temp);
            }
        }

        return result;
    }
}
```

1.9 longest Substring with At Most K Distinct Characters (Hard->Leetcode No.340)

Solution: See CleanCodeHandbook-1 Question 11 -> Further Thoughts

CHAPTER 2: Heap

2.1 Meeting Rooms II (Medium->Leetcode No.253)

Given an array of meeting time intervals consisting of start and end times [[s1,e1],[s2,e2],...] find the minimum number of conference rooms required.

```
public int minMeetingRooms(Interval[] intervals) {
   if(intervals==null||intervals.length==0)
      return 0;

   Arrays.sort(intervals, new Comparator<Interval>(){
      public int compare(Interval i1, Interval i2){
         return i1.start-i2.start;
      }
   });

   PriorityQueue<Integer> queue = new PriorityQueue<Integer>();
   int count=1;
   queue.offer(intervals[0].end);

   for(int i=1; i<intervals.length; i++){
      if(intervals[i].start<queue.peek()){
         count++;

      }else{
         queue.poll();
      }

      queue.offer(intervals[i].end);
   }

   return count;
}
```

CHAPTER 3: Hash Table

3.1 Two Sum III- Data Structure Design(Easy->Leetcode No.170)

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.

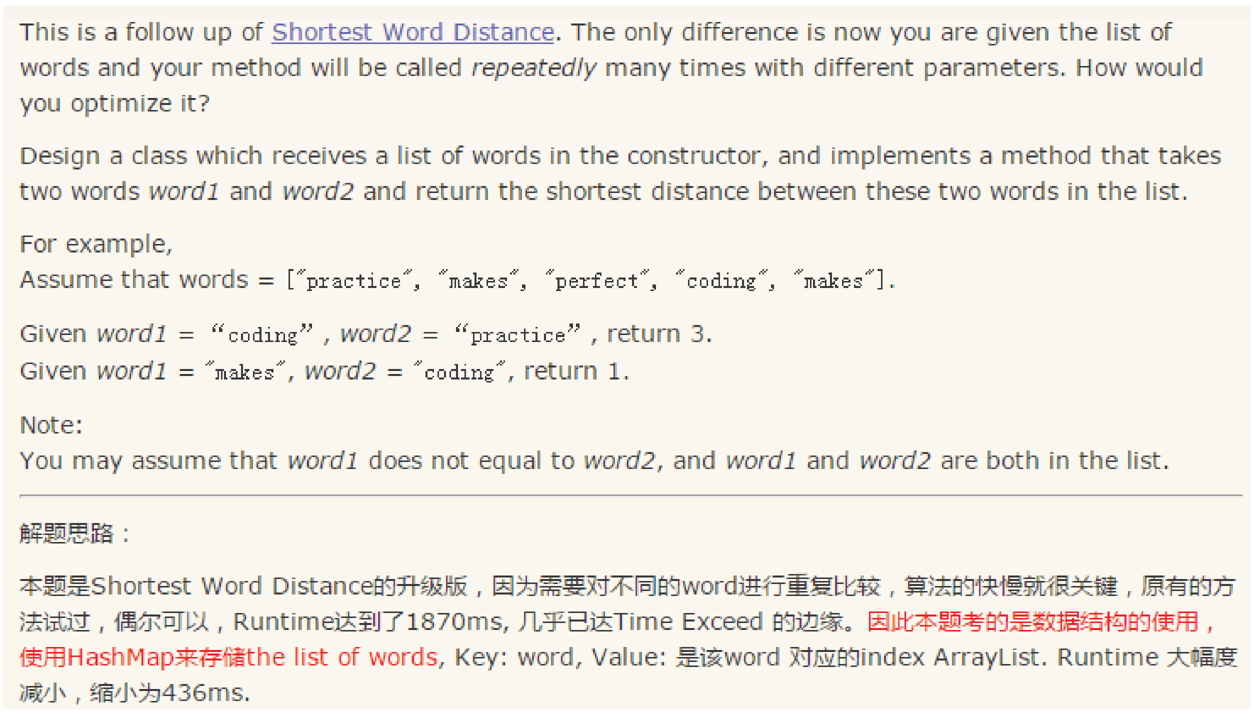
For example,

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

Solution:

```
public class TwoSum{
        private HashMap<Integer, Integer> elements = new HashMap<>();
        public void add(int number){
                if(elements.containsKey(number)){
                        map.put(number, map.get(number)+1);
                }else{
                        elements.put(number, 1);
                }
        }
        public boolean find (int value){
                for(Integer i : elements.keySet()){
                        int target = value-i;
                        if(elements.containsKey(target)){
                                if(i==target && elements.get(target)<2) continue;
                        }
                        return true;
                }
                return false;
        }
}
```

3.2 Shortest Word Distance II (Medium->Leetcode No.244)



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.

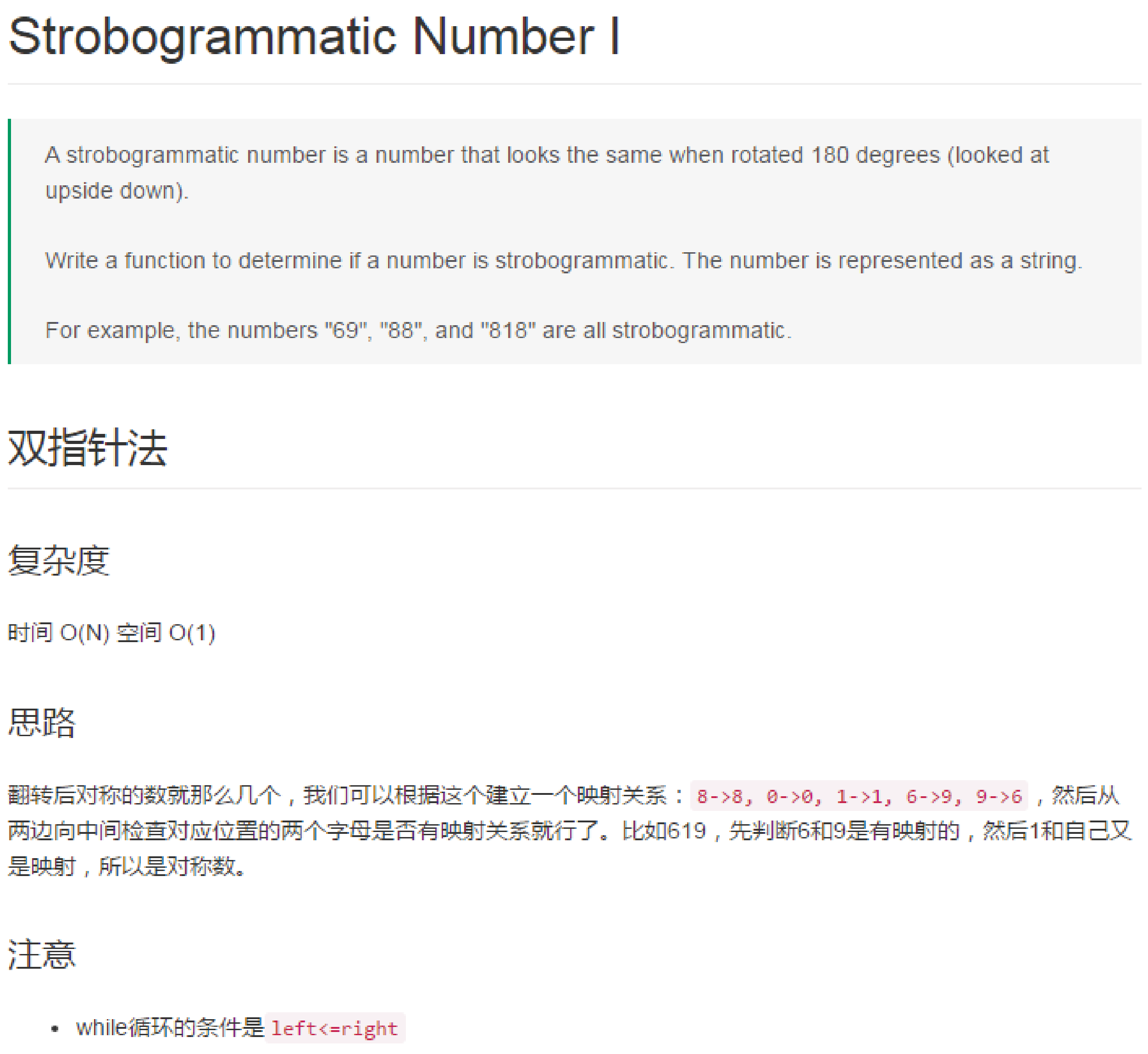
解题思路：

本题是Shortest Word Distance的升级版，因为需要对不同的word进行重复比较，算法的快慢就很关键，原有的方法试过，偶尔可以，Runtime达到了1870ms, 几乎已达Time Exceed 的边缘。因此本题考的是数据结构的使用，使用HashMap来存储the list of words, Key: word, Value: 是该word 对应的index ArrayList. Runtime 大幅度减小，缩小为436ms.

Solution:

```
public class WordDistance{
        private HashMap<String, List<Integer>> map;
        public WordDistance(String[] words){
                map = new HashMap<>();
                for(int i=0;i<words.length;i++){
                        String tmp = words[i];
                        if(map.containsKey(tmp)){
                                map.get(tmp).add(i);
                        }else{
                                List<Integer> list = new ArrayList<>();
                                list.add(i);
                                map.put(tmp,list);
                        }
                }
        }

        public int shorstest(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++;
                        }else{
                                ret = Math.min(ret, index1-index2);
                        j++;
                    }
                }
                return ret;
        }
}
```

3.3 Strobogrammatic Number (Easy -> Leetcode No.246)



Strobogrammatic Number I

> 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.

双指针法

复杂度

时间 O(N) 空间 O(1)

思路

翻转后对称的数就那么几个，我们可以根据这个建立一个映射关系：`8->8, 0->0, 1->1, 6->9, 9->6`，然后从两边向中间检查对应位置的两个字母是否有映射关系就行了。比如619，先判断6和9是有映射的，然后1和自己又是映射，所以是对称数。

注意

- while循环的条件是`left<=right`

Solution

```
public boolean isStrobogrammatic(String num){
        HashMap<Character, Character> map = new HashMap<>();
        map.put('1','1');
        map.put('0','0');
        map.put('6','9');
        map.put('9','6');
        map.put('8','8');
        int left = 0, right=num.length()-1;
        while(left<=right){
                if(!map.containsKey(num.charAt(left))  ||  map.get(num.charAt(left))!=num.charAt(right))
                        return false;
                 left++;
                 right--;
        }
        return true;
}
```

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"]`.

中间插入法

复杂度

时间 O(N) 空间 O(1)

思路

找出所有的可能，必然是深度优先搜索。但是每轮搜索如何建立临时的字符串呢？因为数是"对称"的，我们插入一个字母就知道对应位置的另一个字母是什么，所以我们可以从中间插入来建立这个临时的字符串。这样每次从中间插入两个"对称"的字符，之前插入的就被挤到两边去了。这里有几个边界条件要考虑：

1. 如果是第一个字符，即临时字符串为空时进行插入时，不能插入'0'，因为没有0开头的数字
2. 如果n=1的话，第一个字符则可以是'0'
3. 如果只剩下一个带插入的字符，这时候不能插入'6'或'9'，因为他们不能和自己产生映射，翻转后就不是自己了

这样，当深度优先搜索时遇到这些情况，则要相应的跳过

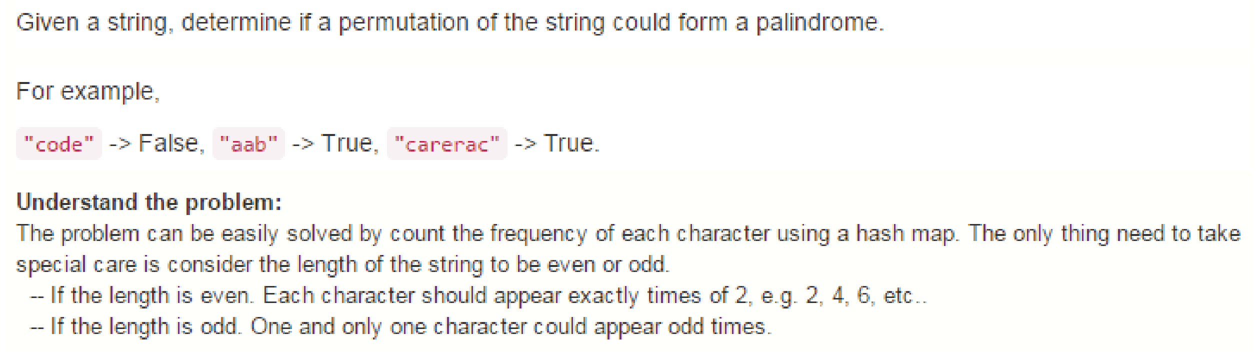
注意

- 为了实现从中间插入，我们用`StringBuilder`在length/2的位置`insert`就行了

Solution:
```
public class Solution{
	char[] table = {'1','0','6','8','9'};
	List<String> result;
	public List<String> findStrobogrammatic(int n){
		result = new ArrayList<>();
		build(n,"");
		return result;
	}
```

```
        public void build(int n, String tmp){
                if(n==tmp.length()){
                        result.add(tmp);
                        return;
                }

                boolean last = n-tmp.length()==1;
                for(int i=0;i<table.length;i++){
                        char c = table[i];
                      // 第一个字符不能是‘0’，但 n=1 除外。只插入一个字符时不能插入‘6’
和‘9’
                        if((n!=1 && tmp.length()==0 && c=='0') || (last && (c=='6' || c=='9'))) continue;
                        StringBuilder sb = new StringBuilder(tmp);
                        // 插入字符 c 和它的对应字符
                        append(last, c, sb);
                        build(n, sb.toString());
                }
        }

        public void append(boolean last, char c, StringBuilder sb){
                if(c=='6') sb.insert(sb.length()/2, "69");
                else if(c=='9') sb.insert(sb.length()/2,"96");
                else sb.insert(sb.length()/2, last?c:""+c+c);
        }
}
```

3.4 Palindrome Permutation (Easy->Leetcode No.266)



Given a string, determine if a permutation of the string could form a palindrome.

For example,

`"code"` -> False, `"aab"` -> True, `"carerac"` -> True.

Understand the problem:
The problem can be easily solved by count the frequency of each character using a hash map. The only thing need to take special care is consider the length of the string to be even or odd.
  -- If the length is even. Each character should appear exactly times of 2, e.g. 2, 4, 6, etc..
  -- If the length is odd. One and only one character could appear odd times.

Solution

```
public class Solution{
        public boolean canPermutePalindrome(String s){
                if(s==null || s.length()<=1) return true;
                HashMap<Character, Integer> map = new HashMap<>();
                for(int i=0;i<s.length();i++){
                        char letter = s.charAt(i);
                        if(map.containsKey(letter)){
                                map.put(letter, map.get(letter)+1)
                        }else map.put(letter,1);
                }
                int  tolerance = 0;
                Iterator it = map.entrySet().iterator();
                while(it.hasNext()){
                        Map.Entry entry = (Map.Entry) it.next();
                        if((int)entry.getValue()%2!=0) tolerance++;
                }
                if(s.length()%2==0) return tolerance==0;
                else return tolerance==1;
        }
}
public boolean canPermutePalindrome(String s) {
        BitSet bs = new BitSet();
        for (byte b : s.getBytes())
                bs.flip(b);
        return bs.cardinality() < 2;
}
```

3.5 Unique Word Abbreviation (Easy->Leetcode No.288)

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.

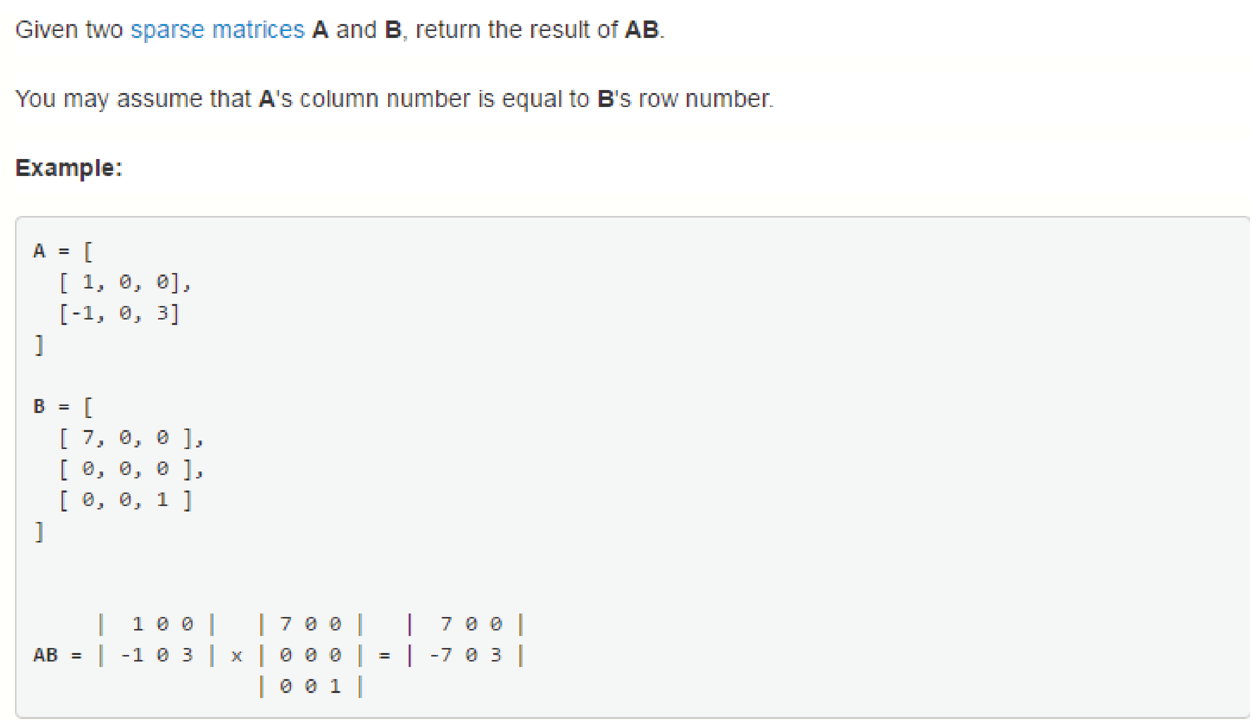
Example:

```
Given dictionary = [ "deer", "door", "cake", "card" ]

isUnique("dear") -> false
isUnique("cart") -> true
isUnique("cane") -> false
isUnique("make") -> true
```

Solution

```
public class ValidWordAbbr{
    private HashMap<String, Integer> dict;
    private HashMap<String, Integer> abbrDict;
    public  ValidWordAbbr(String[] dictionary){
        dict = = new HashMap<>();
        abbrDict = = new HashMap<>();
        for(String s:dictionary){
            if(dict.containsKey(s)) dict.put(s,dict.get(s)+1);
            else dict.put(s,1);
            String abbr = abbreviation(s);
            if(abbrDict.containsKey(abbr)) abbrDict.put(abbr, abbrDict.get(abbr)+1);
            else abbrDict.put(abbr,1);
        }
    }
    public boolean isUnique(String word){
        if(word==null || word.length()==0) return true;
        String abbr = abbreviation(word);
        if(!abbrDict.containsKey(abbr)) return true;
        else{
            if(dict.containsKey(word) && abbrDict.get(abbr)==dict.get(word)) return true;
        }
        return false;
    }

    public String abbreviation(String word){
        if(word.length()<=2) return word;
        StringBuilder sb = new StringBuilder();
        sb.append(word.charAt(0));
        sb.append(word.length()-2);
        sb.append(word.charAt(word.length()-1));
        return sb.toString();
    }
}
```

3.6 Sparse Matrix Multiplication (Medium->Leetcode No.311)



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 |
```

Solution

```
public class Solution {
  public int[][] multiply(int[][] A, int[][] B) {
    if (A == null || A.length == 0 ||
      B == null || B.length == 0) {
      return new int[0][0];
    }

    int m = A.length;
    int n = A[0].length;
    int l = B[0].length;

    int[][] C = new int[m][l];

    // Step 1: convert the sparse A to dense format
    Map<Integer, Map<Integer, Integer>> denseA = new HashMap<>();
    for (int i = 0; i < m; i++) {
      for (int j = 0; j < n; j++) {
        if (A[i][j] != 0) {
          if (!denseA.containsKey(i)) {
            denseA.put(i, new HashMap<>());
          }
          denseA.get(i).put(j, A[i][j]);
        }
      }
    }
```

```
        // Step 2: convert the sparse B to dense format
        Map<Integer, Map<Integer, Integer>> denseB = new HashMap<>();
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < l; j++) {
                if (B[i][j] != 0) {
                    if (!denseB.containsKey(i)) {
                        denseB.put(i, new HashMap<>());
                    }
                    denseB.get(i).put(j, B[i][j]);
                }
            }
        }

        // Step3: calculate the denseA * denseB
        for (int i : denseA.keySet()) {
            for (int j : denseA.get(i).keySet()) {
                if (!denseB.containsKey(j)) {
                    continue;
                }

                for (int k : denseB.get(j).keySet()) {
                    C[i][k] += denseA.get(i).get(j) * denseB.get(j).get(k);
                }
            }
        }

        return C;
    }
}
```
Another solution using a table
```
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++) {
                        if (B[k][j] != 0) C[i][j] += A[i][k] * B[k][j];
                    }
                }
            }
        }
        return C;
    }
}
```
3.7 Binary Tree Vertical Order Traversal (Medium->Leetcode No.314)

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 `[3,9,20,4,5,2,7]`,

```
    _3_
   /   \
  9     20
 / \   / \
4   5 2   7
```

return its vertical order traversal as:

```
[
  [4],
  [9],
  [3,5,2],
  [20],
  [7]
]
```

题解：

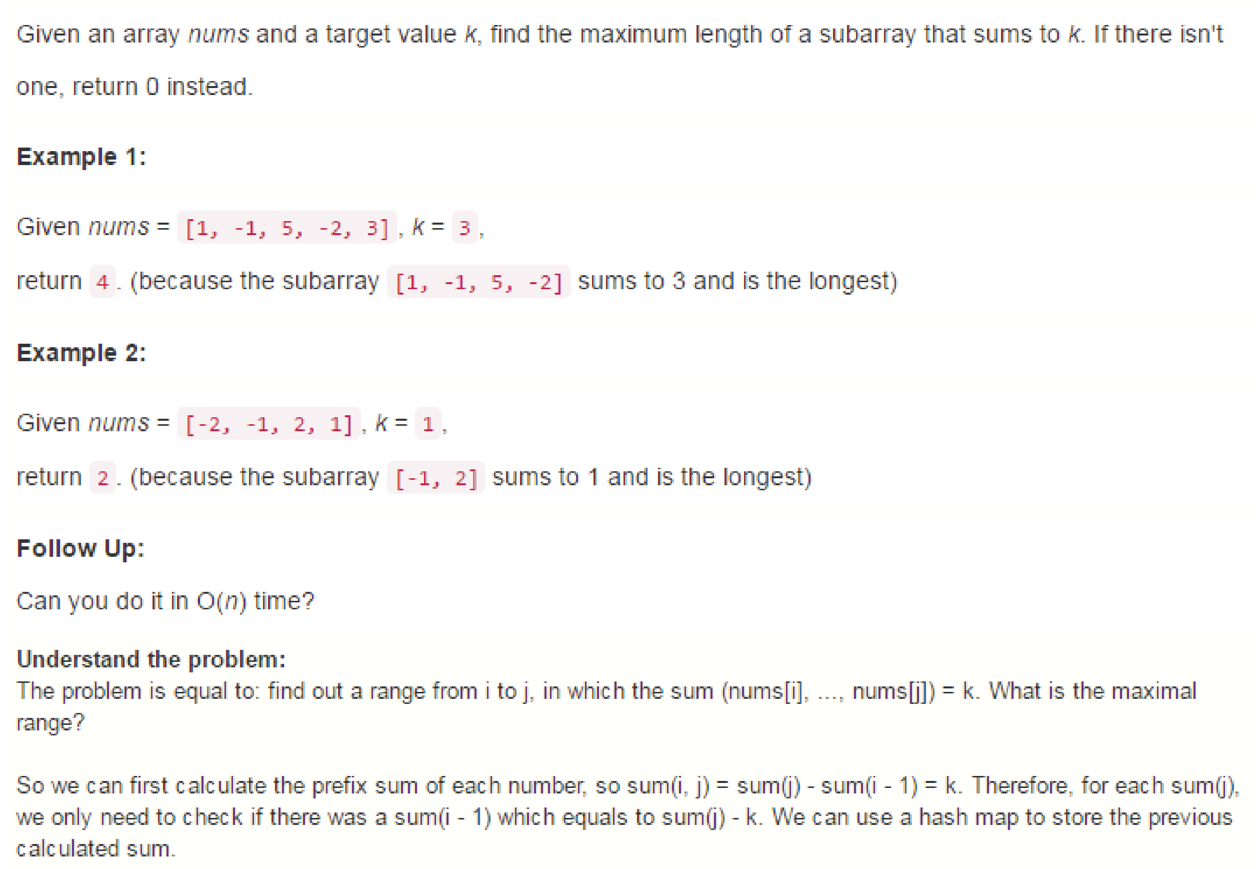
二叉树Vertical order traversal。这道题意思很简单但例子举得不够好，假如上面第二个例子里5还有右子树的话，就会和20在一条column里。总的来说就是假定一个node的column是 i，那么它的左子树column就是i - 1，右子树column就是i + 1。我们可以用decorator模式建立一个TreeColumnNode，包含一个TreeNode，以及一个column value，然后用level order traversal进行计算就可以了，计算的时候用一个HashMap保存column value以及相同value的点。也要设置一个min column value和一个max column value，方便最后按照从小到大顺序获取hashmap里的值输出。这道题Discuss区Yavinci大神写得非常棒，放在reference里。

Solution Reference
Reference:
https://leetcode.com/discuss/75054/5ms-java-clean-solution
https://leetcode.com/discuss/73113/using-hashmap-bfs-java-solution

https://leetcode.com/discuss/74022/hashmap-bfs-solution-in-java
https://leetcode.com/discuss/73893/java-level-order-traversal-solution
http://algorithms.tutorialhorizon.com/print-the-binary-tree-in-vertical-order-path/
http://www.geeksforgeeks.org/print-binary-tree-vertical-order/
http://www.geeksforgeeks.org/print-binary-tree-vertical-order-set-2/

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    private class TreeColumnNode{
        public TreeNode treeNode;
        int col;
        public TreeColumnNode(TreeNode node, int col) {
            this.treeNode = node;
            this.col = col;
        }
    }

    public List<List<Integer>> verticalOrder(TreeNode root) {
        List<List<Integer>> res = new ArrayList<>();
        if(root == null) {
            return res;
        }
        Queue<TreeColumnNode> queue = new LinkedList<>();
        Map<Integer, List<Integer>> map = new HashMap<>();
        queue.offer(new TreeColumnNode(root, 0));
        int curLevel = 1;
        int nextLevel = 0;
        int min = 0;
        int max = 0;

        while(!queue.isEmpty()) {
            TreeColumnNode node = queue.poll();
            if(map.containsKey(node.col)) {
                map.get(node.col).add(node.treeNode.val);
            } else {
                map.put(node.col, new
ArrayList<Integer>(Arrays.asList(node.treeNode.val)));
            }
            curLevel--;

            if(node.treeNode.left != null) {
                queue.offer(new TreeColumnNode(node.treeNode.left, node.col - 1));
                nextLevel++;
                min = Math.min(node.col - 1, min);
            }
            if(node.treeNode.right != null) {
                queue.offer(new TreeColumnNode(node.treeNode.right, node.col + 1));
                nextLevel++;
                max = Math.max(node.col + 1, max);
            }
            if(curLevel == 0) {
                curLevel = nextLevel;
                nextLevel = 0;
            }
        }

        for(int i = min; i <= max; i++) {
            res.add(map.get(i));
        }

        return res;
    }
}
```

3.8 Maximum Size Subarray Sum Equals K (Easy->leetcode No.325)



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.

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?

Understand the problem:
The problem is equal to: find out a range from i to j, in which the sum (nums[i], ..., nums[j]) = k. What is the maximal range?

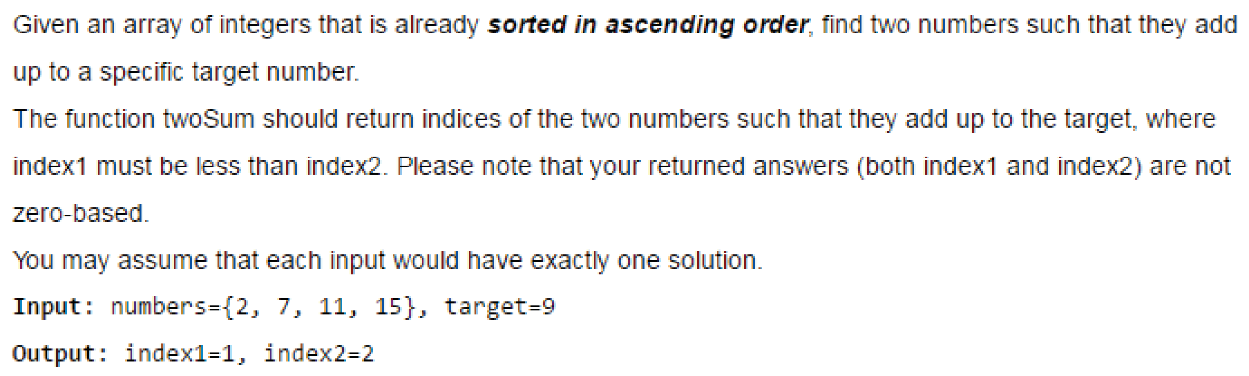
So we can first calculate the prefix sum of each number, so sum(i, j) = sum(j) - sum(i - 1) = k. Therefore, for each sum(j), we only need to check if there was a sum(i - 1) which equals to sum(j) - k. We can use a hash map to store the previous calculated sum.

Solution

```
public class Solution{
        public int maxSubArrayLen(int[] nums, int k){
                if(nums==null || nums.length==0) return 0;
                int sum = 0;
                int maxLen = 0;
                Map<Integer, Integer> map = new HashMap<>();
               map.put(0,-1) // if the maximal range starts from 0, we need to calculate sum(j)-sum (i-1)
                for(int i=0;i<nums.length;i++){
                        sum+=nums[i];
                        if(!map.containsKey(sum)) map.put(sum,i);
                        if(map.containsKey(sum-k)) maxLen = Math.max(maxLen,i- map.get(sum-k));
                }
                return maxLen;
        }
}
```

CHAPTER 4: Binary Search

4.1 Two Sum II - Input array is sorted (Medium->Leetcode No.167)



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.

Input: `numbers={2, 7, 11, 15}, target=9`

Output: `index1=1, index2=2`

Solution

```
public class Solution{
        public int[]  twoSum(int[] nums, int target){
                if(nums==null || nums.length==0) return null;
                int i=0, j = nums.length-1;
                while(i<j){
                        int tmp = nums[i]+nums[j];
                        if(tmp<target) i++;
                        else if(tmp>target) j--;
                        else return new int[]{i+1,j+1};
                }
                return null;
        }
}
```

4.2 Closest Binary Search Tree Value(Easy->Leetcode No.270)

> 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. You are guaranteed to have only one unique value in the BST that is closest to the target.

递归法

复杂度

时间 O(logN) 空间 O(H)

思路

根据二叉树的性质，我们知道当遍历到某个根节点时，最近的那个节点要么是在子树里面，要么就是根节点本身。所以我们根据这个递归，返回子树中最近的节点，和根节点中更近的那个就行了。

```java
public class Solution {
    public int closestValue(TreeNode root, double target) {
        // 选出子树的根节点
        TreeNode kid = target < root.val ? root.left : root.right;
        // 如果没有子树，也就是递归到底时，直接返回当前节点值
        if(kid == null) return root.val;
        // 找出子树中最近的那个节点
        int closest = closestValue(kid, target);
        // 返回根节点和子树最近节点中，更近的那个节点
        return Math.abs(root.val - target) < Math.abs(closest - target) ? root.val : closest;
    }
}
```

迭代法

复杂度

时间 O(logN) 空间 O(H)

思路

记录一个最近的值，然后沿着二叉搜索的路径一路比较下去，并更新这个最近值就行了。因为我们知道离目标数最接近的数肯定在二叉搜索的路径上。

代码

```
public class Solution {
    public int closestValue(TreeNode root, double target) {
        int closest = root.val;
        while(root != null){
            // 如果该节点的离目标更近，则更新到目前为止的最近值
            closest = Math.abs(closest - target) < Math.abs(root.val - target) ? closest : root.
            // 二叉搜索
            root = target < root.val ? root.left : root.right;
        }
        return closest;
    }
}
```

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)?
>
> Hint:
>
> Consider implement these two helper functions: getPredecessor(N), which returns the next smaller node to N. getSuccessor(N), which returns the next larger node to N.

中序遍历法

复杂度

时间 O(N) 空间 Max(O(K),O(H))

思路

二叉搜索树的中序遍历就是顺序输出二叉搜索树，所以我们只要中序遍历二叉搜索树，同时维护一个大小为K的队列，前K个数直接加入队列，之后每来一个新的数（较大的数），如果该数和目标的差，相比于队头的数离目标的差来说，更小，则将队头拿出来，将新数加入队列。如果该数的差更大，则直接退出并返回这个队列，因为后面的数更大，差值也只会更大。

Solution

```java
public class Solution {
    public List<Integer> closestKValues(TreeNode root, double target, int k)
{
        Queue<Integer> klist = new LinkedList<Integer>();
        Stack<TreeNode> stk = new Stack<TreeNode>();
        // 迭代中序遍历二叉搜索树的代码
        while(root != null){
            stk.push(root);
            root = root.left;
        }
        while(!stk.isEmpty()){
            TreeNode curr = stk.pop();
            // 维护一个大小为 k 的队列
            // 队列不到 k 时直接加入
            if(klist.size() < k){
                klist.offer(curr.val);
            } else {
            // 队列到 k 时，判断下新的数是否更近，更近就加入队列并去掉队头
                int first = klist.peek();
                if(Math.abs(first - target) > Math.abs(curr.val - target)){
                    klist.poll();
                    klist.offer(curr.val);
                } else {
                // 如果不是更近则直接退出，后面的数只会更大
                    break;
                }
            }
            // 中序遍历的代码
            if(curr.right != null){
                curr = curr.right;
                while(curr != null){
                    stk.push(curr);
                    curr = curr.left;
                }
            }
        }
        // 强制转换成 List，是用 LinkedList 实现的所以可以转换
        return (List<Integer>)klist;
    }
}
```

4.3 Smallest Rectangle Enclosing Black Pixels (Hard->Leetcode No.302)

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.

Solution

```
public int minArea(char[][] image, int x, int y) {
    int m = image.length, n = image[0].length;
    int colMin = binarySearch(image, true, 0, y, 0, m, true);
    int colMax = binarySearch(image, true, y + 1, n, 0, m, false);
    int rowMin = binarySearch(image, false, 0, x, colMin, colMax, true);
    int rowMax = binarySearch(image, false, x + 1, m, colMin, colMax, false);
    return (rowMax - rowMin) * (colMax - colMin);
}

public int binarySearch(char[][] image, boolean horizontal, int lower, int upper, int min, int
max, boolean goLower) {
    while(lower < upper) {
        int mid = lower + (upper - lower) / 2;
        boolean inside = false;
        for(int i = min; i < max; i++) {
            if((horizontal ? image[i][mid] : image[mid][i]) == '1') {
                inside = true;
                break;
            }
        }
        if(inside == goLower) {
            upper = mid;
        } else {
            lower = mid + 1;
        }
    }
    return lower;
}
```

CHAPTER 5: Dynamic Programming

5.1 Paint House (Medium --> Leetcode No.256)

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.

Solution

The basic idea is when we have painted the first i houses, and want to paint the i+1 th house, we have 3 choices: paint it either red, or green, or blue. If we choose to paint it red, we have the follow deduction:

```
paintCurrentRed = min(paintPreviousGreen,paintPreviousBlue) + costs[i+1][0]
```

Same for the green and blue situation. And the initialization is set to costs[0], so we get the code:

```
public class Solution {
public int minCost(int[][] costs) {
    if(costs.length==0) return 0;
    int lastR = costs[0][0];
    int lastG = costs[0][1];
    int lastB = costs[0][2];
    for(int i=1; i<costs.length; i++){
        int curR = Math.min(lastG,lastB)+costs[i][0];
        int curG = Math.min(lastR,lastB)+costs[i][1];
        int curB = Math.min(lastR,lastG)+costs[i][2];
        lastR = curR;
        lastG = curG;
        lastB = curB;
    }
    return Math.min(Math.min(lastR,lastG),lastB);
}
```

}

5.2 Paint House II (Hard-->Leetcode No.265)

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.

Solution

```
public int minCostII(int[][] costs) {
    if(costs==null || costs.length==0)
        return 0;

    int preMin=0;
    int preSecond=0;
    int preIndex=-1;

    for(int i=0; i<costs.length; i++){
        int currMin=Integer.MAX_VALUE;
        int currSecond = Integer.MAX_VALUE;
        int currIndex = 0;

        for(int j=0; j<costs[0].length; j++){
            if(preIndex==j){
                costs[i][j] += preSecond;
            }else{
                costs[i][j] += preMin;
            }

            if(currMin>costs[i][j]){
                currSecond = currMin;
                currMin=costs[i][j];
                currIndex = j;
            } else if(currSecond>costs[i][j] ){
                currSecond = costs[i][j];
            }
        }

        preMin=currMin;
        preSecond=currSecond;
        preIndex =currIndex;
    }

    int result = Integer.MAX_VALUE;
    for(int j=0; j<costs[0].length; j++){
        if(result>costs[costs.length-1][j]){
            result = costs[costs.length-1][j];
        }
    }
    return result;
}
```

5.3 Paint Fence (Easy->Leetcode No.276)

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.

Solution 1: with O(N) time complexity and O(N) space complexity

```
public int numWays(int n, int k) {
    if (n == 0 || k == 0) return 0;
    if (n == 1) return k;
    // same[i] means the ith post has the same color with the (i-1)th post.
    int[] same = new int[n];
    // diff[i] means the ith post has a different color with the (i-1)th post.
    int[] diff = new int[n];
    same[0] = same[1] = k;
    diff[0] = k;
    diff[1] = k * (k - 1);
    for (int i = 2; i < n; ++i) {
        same[i] = diff[i-1];
        diff[i] = (k - 1) * same[i-1] + (k - 1) * diff[i-1];
    }
    return same[n-1] + diff[n-1];
}
```

Solution 2: with O(N) time complexity and O(1) space complexity

```
public class Solution {
    public int numWays(int n, int k) {
        if(n == 0 || k == 0) return 0;
        int[] dp = new int[n];
        for(int i = 0; i < n; i++){
            if(i == 0){
                dp[i] = k;
            }
            else if(i == 1){
                dp[i] = k*k;
            }
            else{
                dp[i] = dp[i-1]*(k-1) + dp[i-2]*(k-1);
            }
        }
        return dp[n-1];
    }
}
```

CHAPTER 6: Depth-first Search

6.1 Graph Valid Tree (Medium->Leetcode No.261)

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.

Hint:

Given n = 5 and edges = [[0, 1], [1, 2], [3, 4]], what should your return? Is this case a valid tree? Show More Hint 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.

并查集

复杂度

时间 O(N^M) 空间 O(1)

思路

判断输入的边是否能构成一个树，我们需要确定两件事：

1. 这些边是否构成环路，如果有环则不能构成树
2. 这些边是否能将所有节点连通，如果有不能连通的节点则不能构成树

因为不需要知道具体的树长什么样子，只要知道连通的关系，所以并查集相比深度优先搜索是更好的方法。我们定义一个并查集的数据结构，并提供标准的四个接口：

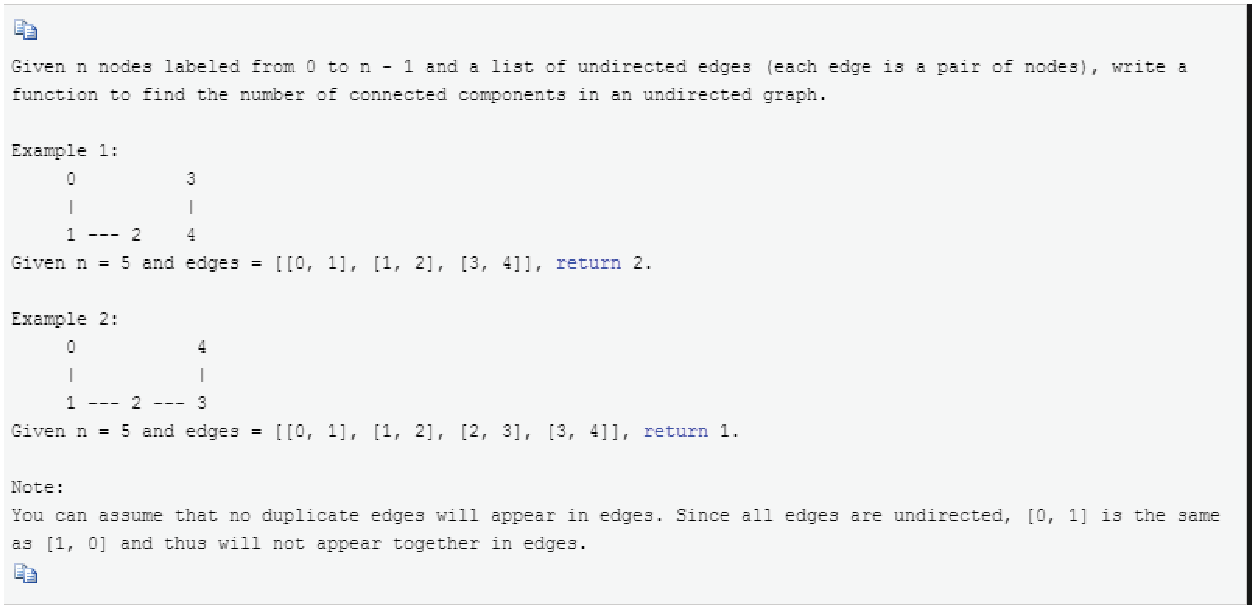
- `union` 将两个节点放入一个集合中
- `find` 找到该节点所属的集合编号
- `areConnected` 判断两个节点是否是一个集合
- `count` 返回该并查集中有多少个独立的集合

具体并查集的原理，参见这篇文章。简单来讲，就是先构建一个数组，节点0到节点n-1，刚开始都各自独立的属于自己的集合。这时集合的编号是节点号。然后，每次union操作时，我们把整个并查集中，所有和第一个节点所属集合号相同的节点的集合号，都改成第二个节点的集合号。这样就将一个集合的节点归属到同一个集合号下了。我们遍历一遍输入，把所有边加入我们的并查集中，加的同时判断是否有环路。最后如果并查集中只有一个集合，则说明可以构建树。

Solution:

```
public class Solution{
        public boolean validTree(int n, int[][] edges){
                UnionFind uf = new UnionFind(n);
                for(int i=0;i<edges.length;i++){
                        if(!uf.union(edge[i][0],edges[i][1])) return false;
                }
                return fu.count()==1;
        }
}

public class UnionFind{
        int[] ids;
        int cnt;
        public UnionFind(int size){
                this.ids = new int[size];
                for(int i=0;i<this.ids.length;i++){
                        this.ids[i] = i;
                }
                this.cnt = size;
        }
        public boolean union(int m, int n){
                int src = find(m);
                int des = find(n);
                if(src!=des){
                        for(int i=0;i<ids.length;i++){
                                if(ids[i]==src) ids[i]=des;
                        }
                        cnt--;
                        return true;
                }else return false;
        }
        public int find(int m){
                return ids[m];
        }
        public boolean areConnected(int m,int n){ return find(m)==find(n);}
        public int count(){return cnt;}
}
```

6.2 Number of Connected Components in an Undirected Graph (Medium->Leetcode No.323)



```
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.
```

Solution 1: Depth-First Search

```
public class Solution{
        public int countCompnents(int n, int[][] edges){
                if(n<=1) return n;
                List<List<Integer>> adjlist = new ArrayList<List<Integer>>();
                for(int i=0;i<n;i++) adjlist.add(new ArrayList<>());
                for(int[] edge:edges){
                        adjlist.get(edge[0]).add(edge[1]);
                        adjlist.get(edge[1]).add(edge[0]);
                }
                boolean[] visited = new boolean[n];
                int count = 0;
                for(int i=0;i<i;i++){
                        if(!visited[i]){
                                count++;
                                dfs(visited, i, adjlist);
                        }
                }
                return count;
        }
        public void dfs(boolean[] visited, int index, List<List<Integer>> adjlist){
                visited[index]=true;
                for(int i:adjlist.get(index)){
                        if(!visited[i]) dfs(visited, i, adjlist);
                }
        }
}
```

Solution 2: Breadth-First Search

```
public class Solution{
        public int countComponents(int n, int[][] edges){
```

```
        if(n<=1) return n;
        List<List<Integer>> adjlist = new Arraylist<List<Integer>>();
        for(int i=0;i<n;i++) adjlist.add(new ArrayList<Integer>());
        for(int[] edge:edges){
            adjlist.get(edge[0]).add(edge[1]);
            adjlist.get(edge[1]).add(edge[0]);
        }
        boolean[] visited = new boolean[n];
        int count = 0;
        for(int i=0;i<n;i++){
            if(!visited[i]){
                count++;
                Queue<Integer> queue = new LinkedList<Integer>();
                queue.offer(i);
                while(!queue.isEmpty()){
                    int index = queue.poll();
                    visited[index] = true;
                    for(int next:adjlist.get(index)){
                        if(!visited[next]) queue.offer(next);
                    }
                }
            }
        }
    }
    return count;
}
```

6.3 Nested List Weight Sum (Easy->Leetcode No.339)

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)

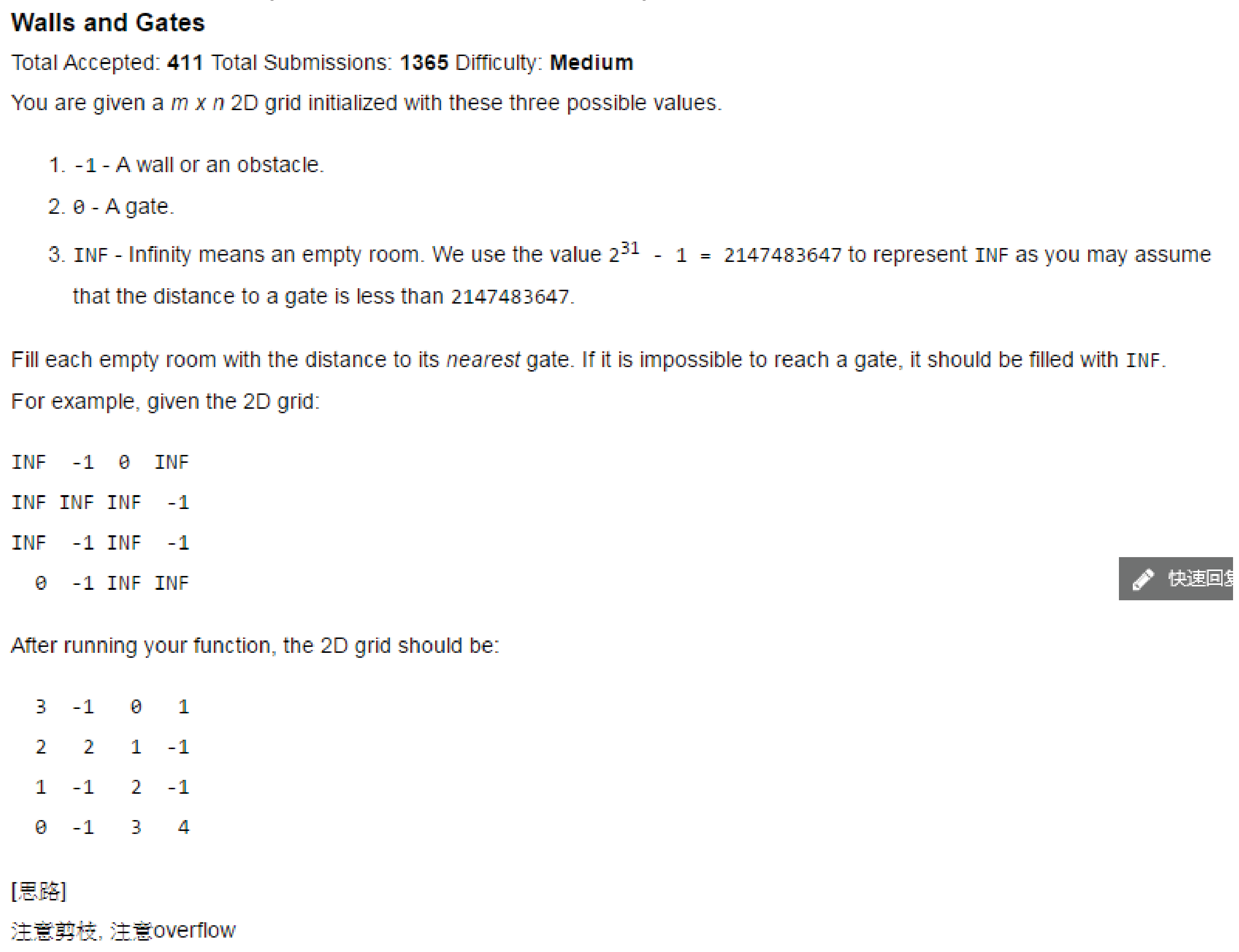
Solution: Depth-First Search

```
public class Solution{
    public int depthSum(List<NestedInteger> nestedList){
        return helper(nestedList, 1);
    }
    public int helper(List<NestedInteger> list, int depth){
        int ret = 0;
        for(NestedInteger e:list){
            ret+=e.isInteger()?e.getInteger()*depth:helper(e.getLIst(), depth+1);
        }
```

```
		return ret;
		}
}
```

CHAPTER 7: Breadth-first Search

7.1 Walls and Gates (Medium-> Leetcode No.286)



Walls and Gates

Total Accepted: **411** Total Submissions: **1365** Difficulty: **Medium**

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
```

[思路]

注意剪枝, 注意overflow

7.2 Shortest Distance from All Buildings (Hard->Leetcode No.317)

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.

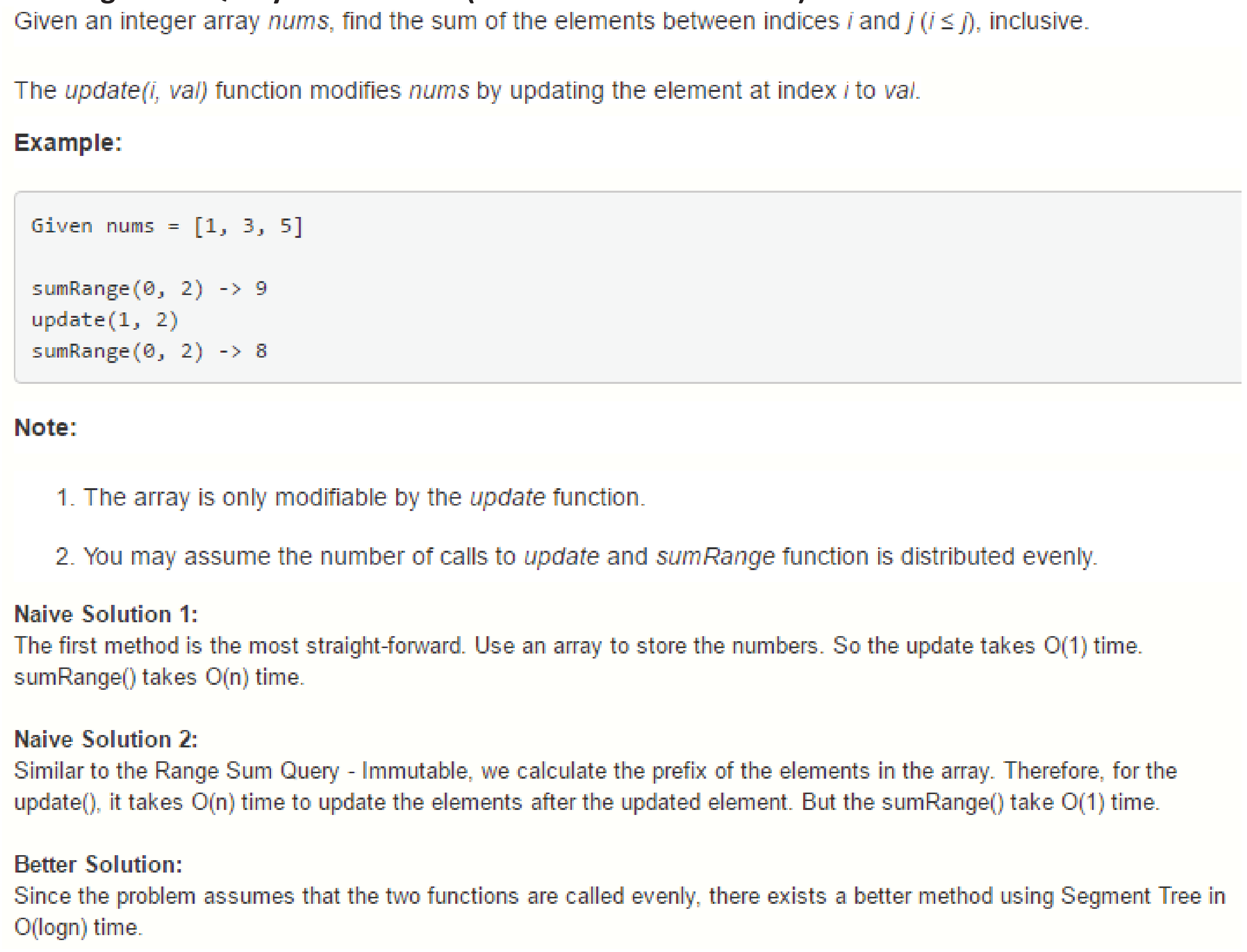
Understand the problem:
A BFS problem. Search from each building and calculate the distance to the building. One thing to note is an empty land must be reachable by all buildings. To achieve this, maintain an array of counters. Each time we reach a empty land from a building, increase the counter. Finally, a reachable point must have the counter equaling to the number of buildings.

```
public class Solution {

/*
- beats 98% 9ms JAVA BFS
- basically use lee-algroithm, bfs each 1 to find out min distance to each 0, accumulate this
distances to each 0 location: distance[][], finally find out min value from distance[][]
- In the case of cannot find a house to each all house:
    - not all 0s can reach each house: reachCount[][] to count the # of house each 0 can reach,
only >= houseCount valid
    - [~~improve speed~~] : not all house can reach each house, in this case, we cannot build
such house,
        -count reachable house #, if < houseCount, return -1 directly!!!!!
*/

public int shortestDistance(int[][] grid) {

    if (grid.length == 0 || grid[0].length == 0) {
        return -1;
    }
    int m = grid.length;
    int n = grid[0].length;
    int[][] distance = new int[m][n];  //accumulated distance of each house (1) to each 0
    int[][] reachCount = new int[m][n]; //count reachable house for each 0
    int houseCount = 0;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid[i][j] == 1) {
                houseCount++;
            }
        }
    }
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid[i][j] == 1) {
                //houseCount++;
                if (!bfs(grid, distance, reachCount, houseCount, m, n, i, j)) {
                    return -1;
                }
            }
        }
    }
    int minDistance = Integer.MAX_VALUE;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid[i][j] == 0 && reachCount[i][j] == houseCount) {
                minDistance = Math.min(minDistance, distance[i][j]);
            }
        }
    }
    return minDistance == Integer.MAX_VALUE ? -1 : minDistance;

}

private boolean bfs(int[][] grid, int[][] distance, int[][] reachCount, int houseCount, int m,
int n, int x, int y) {

    int[][] visited = new int[m][n];
    Queue<int[]> q = new LinkedList<int[]>();
    q.offer(new int[]{x, y});
    int[] dx = new int[]{0, 0, -1, 1};
    int[] dy = new int[]{1, -1, 0, 0};
    int level = 0;
    int count = 0;
    while (!q.isEmpty()) {
        int size = q.size();
        level++;
        for (int i = 0; i < size; i++) {//level by level
            int[] curr = q.poll();
            for (int k = 0; k < 4; k++) { //visit all neighbors & accumulate distance
```

```
                int nx = curr[0] + dx[k];
                int ny = curr[1] + dy[k];
                if (nx >=0 && ny >= 0 && nx < m && ny < n  && visited[nx][ny] == 0) {
                    if (grid[nx][ny] == 0) {
                        distance[nx][ny] += level;
                        visited[nx][ny] = 1;
                        reachCount[nx][ny]++;
                        q.offer(new int[]{nx, ny});
                    } else if (grid[nx][ny] == 1) {
                        count++;
                        visited[nx][ny] = 1;
                    }
                }
            }
        }
    }
    return count == houseCount;
}
```

CHAPTER 8: Binary Indexed Tree/Segment Tree

8.1 Range Sum Query 2D -Mutable (Hard-> LeetCode No.308)



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.

Naive Solution 1:
The first method is the most straight-forward. Use an array to store the numbers. So the update takes O(1) time. sumRange() takes O(n) time.

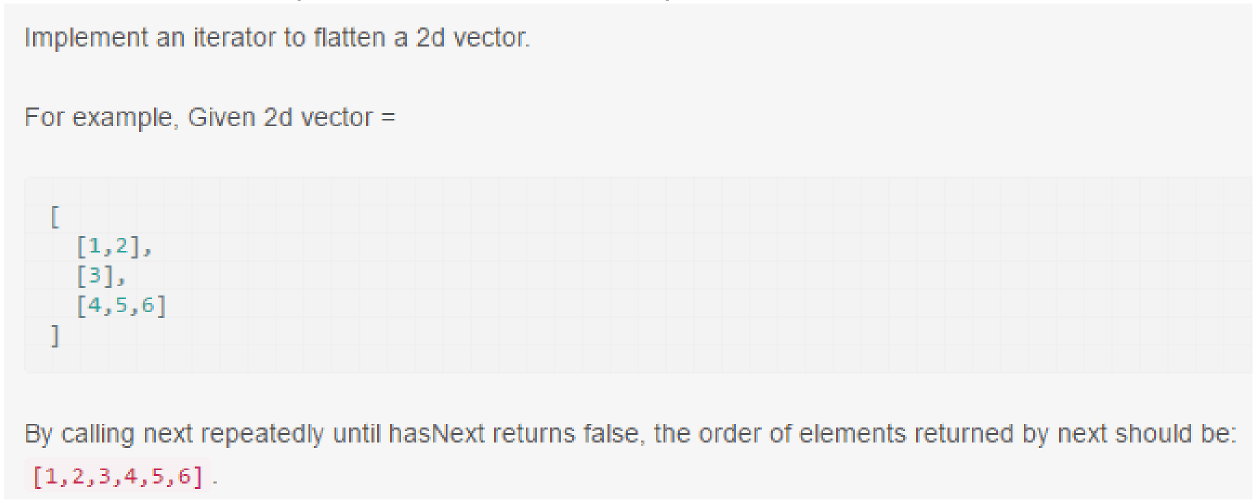
Naive Solution 2:
Similar to the Range Sum Query - Immutable, we calculate the prefix of the elements in the array. Therefore, for the update(), it takes O(n) time to update the elements after the updated element. But the sumRange() take O(1) time.

Better Solution:
Since the problem assumes that the two functions are called evenly, there exists a better method using Segment Tree in O(logn) time.

Solution is the same with the 307 in CleanCodeHandbook

CHAPTER 9: Design

9.1 Flatten 2D Vector (Medium->Leetcode No.251)



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]`.

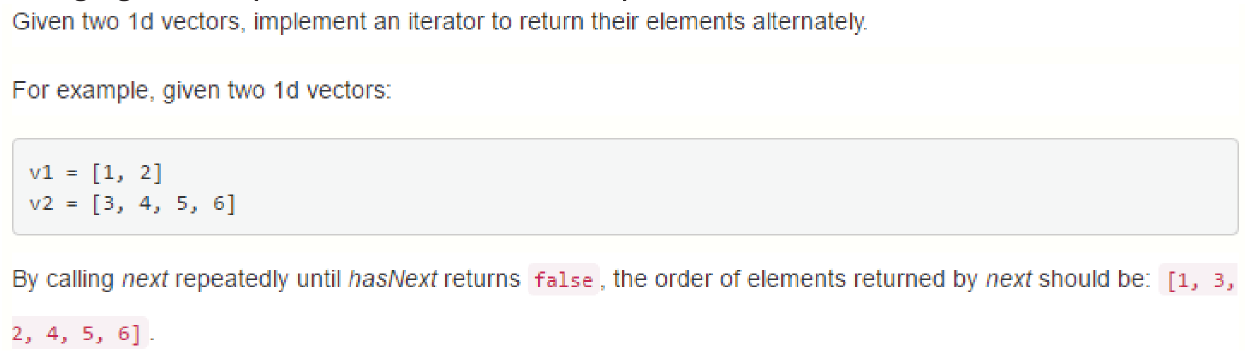
Solution 1: 用一个 List 包含每个 List 的迭代器，然后再记录一个变量，用来表示当前用到了第几个迭代器，这种方法复杂度是 O(N) and O(1).

```
public class Vector2D{
        List<Iterator<Integer>> its;
        Int cur = 0;
        Pubic Vector2D(List<List<Integer>> vec2d){
                this.its = new ArrayList<List<>>();
                for(List<Integer> list:vec2d){
                        if(list.size()>0) this.its.add(list.iterator());
                }
        }
        public int next(){
                Integer result = its.get(cur).next();
                If(!its.get(cur).hasNext()) cur++;
                Return result;
        }
        Public boolean hasNext(){
                return cur<its.size() && its.get(cur).hasNext();
        }
}
```

Solution 2: 双迭代器，维护两个迭代器，一个是输入的 List<List<Integer>>迭代器，它负责遍历 List<Integer>的迭代器，另一个是 List<Integer>的迭代器，他负责记录当前到哪一个 List 的迭代器，每次 next 时，我们先调用一下 hasNext 确保当前 list 的迭代器有下一个值。

```
Public class Vector2D{
        Iterator<List<Ingeter>> its;
        Iterator<Integer> curr;
        Public Vector2D(List<List<Integer>> vec2d){
                Its = vec2d.iterator();
        }
        Public int next(){
                hasNext();
                return curr.next();
        }
        Public boolean hasNext(){
                While((curr==null || !curr.hasNext()) && it.hasNext()) curr = it.next().iterator();
                Return curr!=null && curr.hasNext();
        }
}
```

9.2 Zigzag Iterator (Medium->Leetcode No.281)


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]`.

```
public class ZigzagIterator {
    private List<Integer> v1;
    private List<Integer> v2;
    private int i;
    private int j;
    private int listId;

    public ZigzagIterator(List<Integer> v1, List<Integer> v2) {
        this.v1 = v1;
        this.v2 = v2;
        this.i = 0;
        this.j = 0;
        this.listId = 0;
    }

    public int next() {
        int result = 0;
        if (i >= v1.size()) {
            result = v2.get(j);
            j++;
        } else if (j >= v2.size()) {
            result = v1.get(i);
            i++;
        } else {
            if (listId == 0) {
                result = v1.get(i);
                i++;
                listId = 1;
            } else {
                result = v2.get(j);
                j++;
                listId = 0;
            }
        }
        return result;
    }

    public boolean hasNext() {
        return i < v1.size() || j < v2.size();
    }
}
```

9.3 Moving Average From Data Stream (Easy -> Leetcode No.346)

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 {
    Queue<Integer> queue;
    int n;
    int sum;

    /** Initialize your data structure here. */
    public MovingAverage(int size) {
        queue = new LinkedList<>();
        n = size;
        sum = 0;
    }

    public double next(int val) {
        queue.offer(val);
        double result = 0;
        sum += val;
        if (queue.size() <= n) {
            result = (double) sum / queue.size();
        } else {
            int remove = queue.poll();
            sum -= remove;
            result = (double) sum / n;
        }

        return result;
    }
}
```

```
/**
```

9.4 Design Tic-Tac-Toe (Medium->Leetcode No.348)

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|
```

```java
/** Player {player} makes a move at ({row}, {col}).
    @param row The row of the board.
    @param col The column of the board.
    @param player The player, can be either 1 or 2.
    @return The current winning condition, can be either:
            0: No one wins.
            1: Player 1 wins.
            2: Player 2 wins. */
public int move(int row, int col, int player) {
    int 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;
}
```

```
public class TicTacToe {
    private int[][] rows;
    private int[][] cols;
    private int[] diag;
    private int[] xdiag;
    private int n;

    /** Initialize your data structure here. */
    public TicTacToe(int n) {
        this.n = n;
        rows = new int[2][n];
        cols = new int[2][n];
        diag = new int[2];
        xdiag = new int[2];
    }

    /** Player {player} makes a move at ({row}, {col}).
        @param row The row of the board.
        @param col The column of the board.
        @param player The player, can be either 1 or 2.
        @return The current winning condition, can be either:
                0: No one wins.
                1: Player 1 wins.
                2: Player 2 wins. */
    public int move(int row, int col, int player) {
        int p = player == 1 ? 0 : 1;

        rows[p][row]++;
        cols[p][col]++;

        if (row == col) {
            diag[p]++;
        }

        // X-diagonal
        if (row + col == n - 1) {
            xdiag[p]++;
        }

        // If any of them equals to n, return 1
        if (rows[p][row] == n || cols[p][col] == n ||
            diag[p] == n || xdiag[p] == n) {
            return p + 1;
        }

        return 0;
    }
}

/**
```

```
* Your TicTacToe object will be instantiated and called as such:
* TicTacToe obj = new TicTacToe(n);
* int param_1 = obj.move(row,col,player);
*/
```

CHAPTER 11: Array

11.1 Missing Range (Medium->leetcode No.163)

Given a sorted integer array where the range of elements are [*lower*, *upper*] inclusive, 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"].`

```
private String outputRange(int n, int m) {
    return (n == m)?String.valueOf(n) : n + "->" + m;
}

public List<String> findMissingRanges(int[] nums, int lower, int upper) {
    List<String> ranges = new ArrayList<String>();
    if (nums.length == 0) {  //Empty array misses the range lower->upper.
        ranges.add(outputRange(lower, upper));
        return ranges;
    }
    int prev;
    if (nums[0] - lower > 0) {    //Handles lower boundary. Notice "inclusive".
        ranges.add(outputRange(lower, nums[0] - 1));
        prev = nums[0];
    } else {
        prev = lower;
    }
    for (int cur : nums) {
        if (cur - prev > 1) {
            ranges.add(outputRange(prev + 1, cur - 1)); //Misses range if distance > 1.
        }
        prev = cur;
    }
    if (upper - prev > 0) {  //Handles the upper boundary.
        ranges.add(outputRange(prev + 1, upper));
    }

    return ranges;
}
```

11.2 Shortest Word Distance (Easy->leetcode NO.243)

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.

```
public class Solution {
    public int shortestDistance(String[] words, String word1, String word2) {
        int posA = -1;
        int posB = -1;

        int minDistance = Integer.MAX_VALUE;

        for (int i = 0; i < words.length; i++) {
            if (words[i].equals(word1)) {
                posA = i;
            }

            if (words[i].equals(word2)) {
                posB = i;
            }

            if (posA != -1 && posB != -1) {
                minDistance = Math.min(minDistance, Math.abs(posA - posB));
            }
        }

        return minDistance;
    }
}
```

11.3 Shortest Word Distance III (Medium->leetcode No.245)

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.

```
public class Solution {
    public int shortestWordDistance(String[] words, String word1, String word2)
{
        int posA = -1;
        int posB = -1;
        int minDistance = Integer.MAX_VALUE;

        for (int i = 0; i < words.length; i++) {
            String word = words[i];

            if (word.equals(word1)) {
                posA = i;
            } else if (word.equals(word2)) {
                posB = i;
            }
```

```
                if (posA != -1 && posB != -1 && posA != posB) {
                    minDistance = Math.min(minDistance, Math.abs(posA - posB));
                }

                if (word1.equals(word2)) {
                    posB = posA;
                }
            }

            return minDistance;
        }
}
```

11.4 3sum Smaller (Medium->leetcode No.259)

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]
```

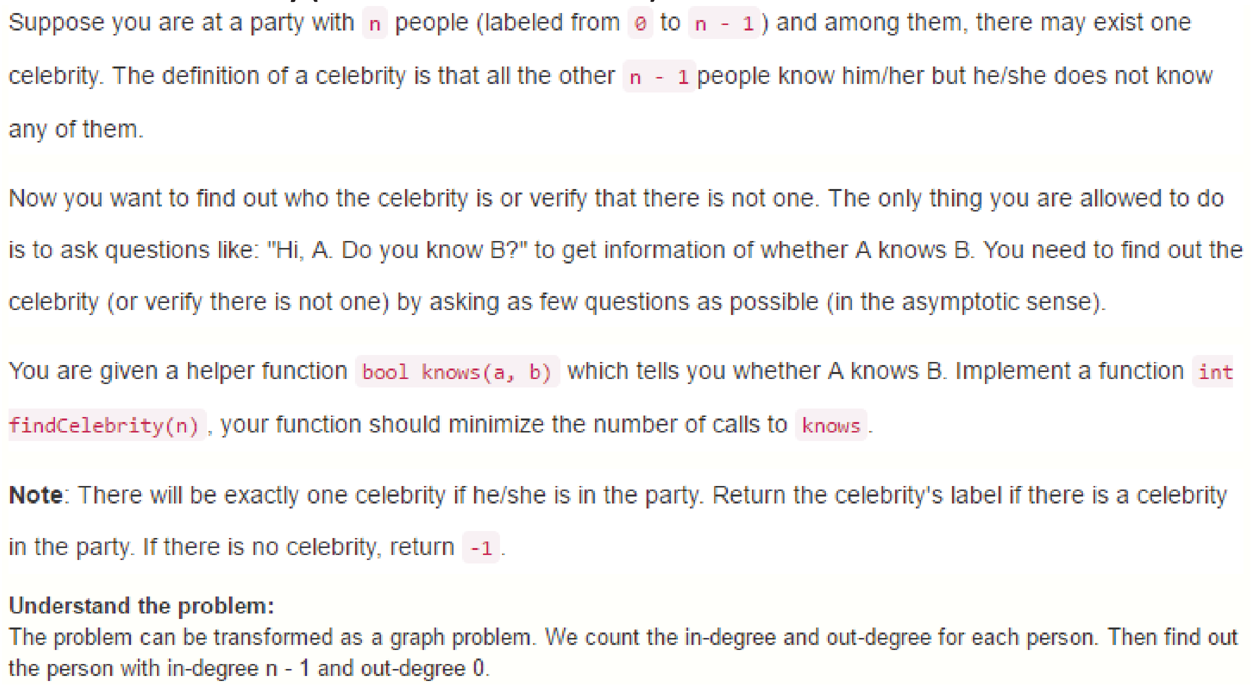
Follow up:

Could you solve it in $O(n^2)$ runtime?

Solution

```
public class Solution {
    public int threeSumSmaller(int[] nums, int target) {
        int result = 0;
        Arrays.sort(nums);
        for(int i = 0; i <= nums.length-3; i++) {
            int lo = i+1;
            int hi = nums.length-1;
            while(lo < hi) {
                if(nums[i] + nums[lo] + nums[hi] < target) {
                    result += hi - lo;
                    lo++;
                } else {
                    hi--;
                }
            }
        }
        return result;
    }
}
```

11.5 Find the Celebrity (Mediu->leetcode No.277)



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`.

Understand the problem:
The problem can be transformed as a graph problem. We count the in-degree and out-degree for each person. Then find out the person with in-degree n - 1 and out-degree 0.

Solution 1 :

Analysis:
Time O(n^2).
Space O(n).

```java
/* The knows API is defined in the parent class Relation.
      boolean knows(int a, int b); */

public class Solution extends Relation {
    public int findCelebrity(int n) {
        if (n <= 1) {
            return -1;
        }

        int[] inDegree = new int[n];
        int[] outDegree = new int[n];

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (i != j && knows(i, j)) {
                    outDegree[i]++;
                    inDegree[j]++;
                }
            }
        }

        for (int i = 0; i < n; i++) {
            if (inDegree[i] == n - 1 && outDegree[i] == 0) {
                return i;
            }
        }

        return -1;
    }
}
```

```
/* The knows API is defined in the parent class Relation.
      boolean knows(int a, int b); */

public class Solution extends Relation {
    public int findCelebrity(int n) {
        if (n <= 1) {
            return -1;
        }

        int left = 0;
        int right = n - 1;

        // Step 1: Find the potential candidate
        while (left < right) {
            if (knows(left, right)) {
                left++;
            } else {
                right--;
            }
        }

        // Step 2: Validate the candidate
        int candidate = right;
        for (int i = 0; i < n; i++) {
            if (i != candidate && (!knows(i, candidate) || knows(candidate, i))) {
                return -1;
            }
        }

        return candidate;
    }
}
```

11.6 Wiggle Sort (Medium->leetcode No.280)

> 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]` .

复杂度

时间 O(NlogN) 空间 O(1)

思路

根据题目的定义，摇摆排序的方法将会很多种。我们可以先将数组排序，这时候从第3个元素开始，将第3个元素和第2个元素交换。然后再从第5个元素开始，将第5个元素和第4个元素交换，以此类推。就能满足题目要求。

代码

```
public class Solution {
    public void wiggleSort(int[] nums) {
        // 先将数组排序
        Arrays.sort(nums);
        // 将数组中一对一对交换
        for(int i = 2; i < nums.length; i+=2){
            int tmp = nums[i-1];
            nums[i-1] = nums[i];
            nums[i] = tmp;
        }
    }
}
```

```java
public class Solution {
    public void wiggleSort(int[] nums) {
        for(int i = 1; i < nums.length; i++){
            // 需要交换的情况：奇数时nums[i] < nums[i - 1]或偶数时nums[i] > nums[i - 1]
            if((i % 2 == 1 && nums[i] < nums[i-1]) || (i % 2 == 0 && nums[i] > nums[i-1])){
                int tmp = nums[i-1];
                nums[i-1] = nums[i];
                nums[i] = tmp;
            }
        }
    }
}
```

复杂度

时间 O(N) 空间 O(1)

思路

题目对摇摆排序的定义有两部分：

1. 如果i是奇数，`nums[i] >= nums[i - 1]`
2. 如果i是偶数，`nums[i] <= nums[i - 1]`

所以我们只要遍历一遍数组，把不符合的情况交换一下就行了。具体来说，如果nums[i] > nums[i - 1]， 则交换以后肯定有nums[i] <= nums[i - 1]。

CHAPTER 12: Math

12.1 Strobogrammatic Number II (Medium->leetcode NO.247)

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"]`.

Understand the problem:

The problem is an extension of the last problem. It could be solved by using recursion.
There are two things need to be careful:
In each recursion, it should recurse with n - 2 not n / 2
Secondly, noticed that the strobogrammatic number should not contain leading "0"s.

```
public class Solution {
    public List<String> findStrobogrammatic(int n) {
        List<String> one = Arrays.asList("0", "1", "8"), two = Arrays.asList(""), r = two;
        if(n%2 == 1)
            r = one;
        for(int i=(n%2)+2; i<=n; i+=2){
            List<String> newList = new ArrayList<>();
            for(String str : r){
                if(i != n)
                    newList.add("0" + str + "0");
                newList.add("1" + str + "1");
                newList.add("6" + str + "9");
                newList.add("8" + str + "8");
                newList.add("9" + str + "6");
            }
            r = newList;
        }
        return r;
    }

}
```

12.2 Strobogrammatic Number III (Medium->leetcode No.248)

Leetcode: 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.

Understand the problem:
The idea would be very close to the previous problem. So we find all the strobogrammatic numbers between the length of low and high. Note that when the n == low or n == high, we need to compare and make sure the strobogrammatic number we find is within the range.

```
char[][] pairs = {{'0', '0'}, {'1', '1'}, {'6', '9'}, {'8', '8'}, {'9', '6'}};
int count = 0;

public int strobogrammaticInRange(String low, String high) {
    for(int len = low.length(); len <= high.length(); len++) {
        dfs(low, high, new char[len], 0, len - 1);
    }
    return count;
}

public void dfs(String low, String high, char[] c, int left, int right) {
    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++;
        return;
    }

    for(char[] p : pairs) {
        c[left] = p[0];
        c[right] = p[1];
        if(c.length != 1 && c[0] == '0') continue;
        if(left < right || left == right && p[0] == p[1]) dfs(low, high, c, left + 1, right -
1);
    }
}
```

12.3 Best Meeting Point (Hard ->leetcode No.296)

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.

复杂度

时间 O(NM) 空间 O(NM)

思路

为了保证总长度最小，我们只要保证每条路径尽量不要重复就行了，比如`1->2->3<-4`这种一维的情况，如果起点是1，2和4，那`2->3`和`1->2->3`这两条路径就有重复了。为了尽量保证右边的点向左走，左边的点向右走，那我们就应该去这些点中间的点作为交点。由于是曼哈顿距离，我们可以分开计算横坐标和纵坐标，结果是一样的。所以我们算出各个横坐标到中点横坐标的距离，加上各个纵坐标到中点纵坐标的距离，就是结果了。

```java
public class Solution {
    public int minTotalDistance(int[][] grid) {
        List<Integer> ipos = new ArrayList<Integer>();
        List<Integer> jpos = new ArrayList<Integer>();
        // 统计出有哪些横纵坐标
        for(int i = 0; i < grid.length; i++){
            for(int j = 0; j < grid[0].length; j++){
                if(grid[i][j] == 1){
                    ipos.add(i);
                    jpos.add(j);
                }
            }
        }
        int sum = 0;
        // 计算纵坐标到纵坐标中点的距离，这里不需要排序，因为之前统计时是按照i的顺序
        for(Integer pos : ipos){
            sum += Math.abs(pos - ipos.get(ipos.size() / 2));
        }
        // 计算横坐标到横坐标中点的距离，这里需要排序，因为统计不是按照j的顺序
        Collections.sort(jpos);
        for(Integer pos : jpos){
            sum += Math.abs(pos - jpos.get(jpos.size() / 2));
        }
        return sum;
    }
}
```

12.4 Line Reflection (Medium->leetcode No.356)

Given n points on a 2D plane, find if there is such a line parallel to y-axis that reflect the given set of 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)?

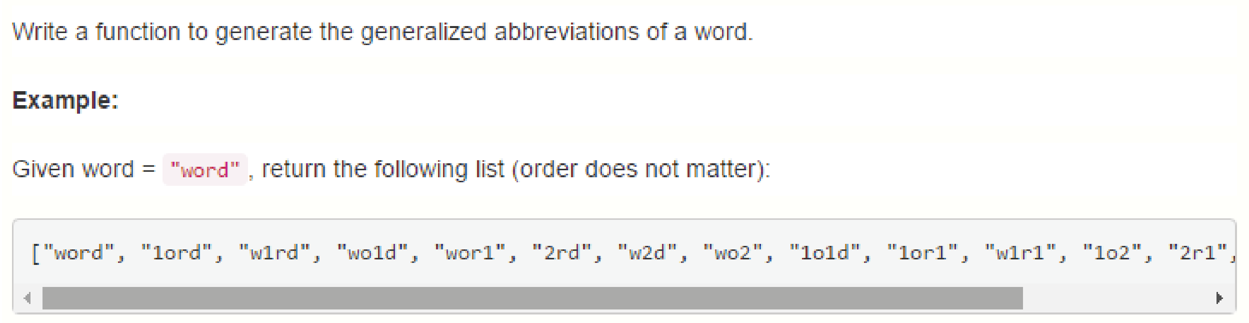
Hint:

1. Find the smallest and largest x-value for all points.
2. If there is a line then it should be at y = (minX + maxX) / 2.
3. For each point, make sure that it has a reflected point in the opposite side.

```
 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){
        //int[] arr = {sum-p[0],p[1]};
        String str = (sum-p[0]) + "a" + p[1];
        if( !set.contains(str))
            return false;

    }
    return true;
}
```

CHAPTER 13: Bit Manipulation

13.1 Generalized Abbreviation (Medium->leetcode NO.320)



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",
```

```
public class Solution {
    public List<String> generateAbbreviations(String word) {
        List<String> result = new ArrayList<>();

        result.add(word);
        generateHelper(0, word, result);

        return result;
    }

    private void generateHelper(int start, String s, List<String> result) {
        if (start >= s.length()) {
            return;
        }

        for (int i = start; i < s.length(); i++) {
            for (int j = 1; i + j <= s.length(); j++) {
                String num = Integer.toString(j);
                String abbr = s.substring(0, i) + num + s.substring(i + j);
                result.add(abbr);
                generateHelper(i + 1 + num.length(), abbr, result); // skip 1b
            }
        }
    }
}
```

13.2 Number of Islands II (Hard->leetcode NO.305)

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]`

The idea is simple. To represent a list of islands, we use **trees**. i.e., a list of roots. This helps us find the identifier of an island faster. If `roots[c] = p` means the parent of node c is p, we can climb up the parent chain to find out the identifier of an island, i.e., which island this point belongs to:

```
Do root[root[roots[c]]]... until root[c] == c;
```

To transform the two dimension problem into the classic UF, perform a linear mapping:

```
int id = n * x + y;
```

Initially assume every cell are in non-island set `{-1}`. When point A is added, we create a new root, i.e., a new island. Then, check if any of its 4 neighbors belong to the same island. If not, `union` the neighbor by setting the root to be the same. Remember to skip non-island cells.

UNION operation is only changing the root parent so the running time is `O(1)`.

FIND operation is proportional to the depth of the tree. If N is the number of points added, the average running time is `O(logN)`, and a sequence of `4N` operations take `O(NlogN)`. If there is no balancing, the worse case could be `O(N^2)`.

Remember that one island could have different `roots[node]` value for each node. Because `roots[node]` is the parent of the node, not the highest root of the island. To find the actually root, we have to climb up the tree by calling **findIsland** function.

Here I've attached my solution. There can be at least two improvements: `union by rank` & `path compression`. However I suggest first finish the basis, then discuss the improvements.

```
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;                      // number of islands
    int[] roots = new int[m * n];       // one island = one tree
    Arrays.fill(roots, -1);

    for(int[] p : positions) {
        int root = n * p[0] + p[1];     // assume new point is isolated island
        roots[root] = root;             // add new island
        count++;
```

```java
        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) {        // if neighbor is in another island
                roots[root] = rootNb;   // union two islands
                root = rootNb;          // current tree root = joined tree root
                count--;
            }
        }

        result.add(count);
    }
    return result;
}

public int findIsland(int[] roots, int id) {
    while(id != roots[id]) {
        roots[id] = roots[roots[id]];
        id = roots[id];

    }
    return id;
}
```

CHAPTER 14: Others

14.1 Binary Tree Upside Down (medium->leetcode No.156)

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
```

快速回复

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

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

Solution1

这题有一个重要的限制就是，整个数的任何一个右孩子都不会再生枝节，基本就是一个梳子的形状。对于树类型的题目，首先可以想到一种递归的思路：**把左子树继续颠倒，颠倒完后，原来的那个左孩子的左右孩子指针分别指向原来的根节点以及原来的右兄弟节点即可。**

[java]

```java
public TreeNode UpsideDownBinaryTree(TreeNode root) {
    if (root == null)
        return null;
    TreeNode parent = root, left = root.left, right = root.right;
    if (left != null) {
        TreeNode ret = UpsideDownBinaryTree(left);
        left.left = right;
        left.right = parent;
        return ret;
    }
    return root;
}
```

第二个思路是直接**用迭代代替递归**，做起来也不麻烦，并且效率会更高，因为省去了递归所用的栈空间。

[java]

```java
public TreeNode UpsideDownBinaryTree(TreeNode root) {
    TreeNode node = root, parent = null, right = null;
    while (node != null) {
        TreeNode left = node.left;
        node.left = right;
        right = node.right;
        node.right = parent;
        parent = node;
        node = left;
    }
    return parent;
}
```

14.2 Count Univalue Subtrees (Medium->leetcode NO.250)

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.

726 views

```java
public class Solution {
    public int countUnivalSubtrees(TreeNode root) {
        int[] arr = new int[1];
        postOrder(arr, root);
        return arr[0];
    }
    public boolean postOrder (int[] arr, TreeNode node) {
        if (node == null) return true;
        boolean left = postOrder(arr, node.left);
        boolean right = postOrder(arr, node.right);
        if (left && right) {
            if (node.left != null && node.left.val != node.val) return false;
            if (node.right != null && node.right.val != node.val) return false;
            arr[0]++;
            return true;
        }
        return false;
    }
}
```

14.3 Meeting Rooms (Easy->leetcode no.252)

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.

Solution

```
/**
 * Definition for an interval.
 * public class Interval {
 *     int start;
 *     int end;
 *     Interval() { start = 0; end = 0; }
 *     Interval(int s, int e) { start = s; end = e; }
 * }
 */
public class Solution {
    public boolean canAttendMeetings(Interval[] intervals) {
        if (intervals == null || intervals.length ==0) {
            return true;
        }

        // Sort according to the start time
        Arrays.sort(intervals, new IntervalComparator());

        Interval prev = intervals[0];
        for (int i = 1; i < intervals.length; i++) {
            Interval curr = intervals[i];
            if (isOverlapped(prev, curr)) {
                return false;
            }
            prev = curr;
        }

        return true;
    }
    public class IntervalComparator implements Comparator<Interval> {
        @Override
        public int compare(Interval a, Interval b) {
            return a.start - b.start;
        }
    }

    private boolean isOverlapped(Interval a, Interval b) {
        return a.end > b.start;
    }
}
```

14.4 Factor Combinations (Medium->leetcode NO.254)

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. Each combination's factors must be sorted ascending, for example: The factors of 2 and 6 is `[2, 6]`, not `[6, 2]`.
2. You may assume that *n* is always positive.
3. Factors should be greater than 1 and less than *n*.

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. Each combination's factors must be sorted ascending, for example: The factors of 2 and 6 is `[2, 6]`, not `[6, 2]`.
2. You may assume that *n* is always positive.
3. 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]
]
```

用 dfs 方法,

```
public class Solution {
    public List<List<Integer>> getFactors(int n) {
        List<List<Integer>> res = new ArrayList<List<Integer>>();
        if (n <= 2) {
            return res;
        }
        List<Integer> tem = new ArrayList<Integer>();
        helper(res, tem,n,2);
        return res;
    }
    public void helper(List<List<Integer>> res, List<Integer> tem, int m,int 
div) {
        if (m <= 1) {
            if (tem.size() > 1) {
                res.add(new ArrayList<Integer>(tem));
            }
            return;
        }
        for (int i = div; i <= m; i++) {
            if (m % i == 0) {
                tem.add(i);
                helper(res, tem,m / i, i);
                tem.remove(tem.size() - 1);
            }

        }
    }
}
```

14.5 Verify Preorder Sequence in Binary Search Tree (Medium ->leetcode No.255)

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.

Solution 1

```
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;
}
```

Solution 2: O(n) time complexity and O(1) space complexity

```
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;
}
```

`Solution 3: O(nlogn) time complexity and O(1) space complexity`

```
public boolean verifyPreorder(int[] preorder) {
    return verify(preorder, 0, preorder.length - 1, Integer.MIN_VALUE, Integer.MAX_VALUE);
}

private boolean verify(int[] preorder, int start, int end, int min, int max) {
    if (start > end) {
        return true;
    }
    int root = preorder[start];
    if (root > max || root < min) {
        return false;
    }

    int rightIndex = start;
    while (rightIndex <= end && preorder[rightIndex] <= root) {
        rightIndex++;
    }
    return verify(preorder, start + 1, rightIndex - 1, min, root) && verify(preorder,
rightIndex, end, root, max);
}
```

14.6 Palindrome Permutation ii (Medium->leetcode No.267)

Given a string `s`, return all the palindromic permutations (without duplicates) of it. Return an empty list if no palindromic permutation could be form.

For example:

Given `s = "aabb"`, return `["abba", "baab"]`.

Given `s = "abc"`, return `[]`.

Hint:

1. If a palindromic permutation exists, we just need to generate the first half of the string.
2. To generate all distinct permutations of a (half of) string, use a similar approach from: Permutations II or Next Permutation.

Solution

```
public List<String> generatePalindromes(String s) {
    int[] map = new int[256];
    for(int i=0;i<s.length();i++){
        map[s.charAt(i)]++;
    }
    int j=0,count=0;
    for(int i=0;i<256;i++){
        if(count== 0 && map[i] %2 == 1){
            j= i;
            count++;
        }else if(map[i] % 2==1){
            return new ArrayList<String>();
        }
    }
    String cur = "";
    if(j != 0){
        cur = ""+ (char)j;
        map[j]--;
    }
    List<String> res = new ArrayList<String>();
    DFS(res,cur,map,s.length());
    return res;
}
public void DFS(List<String> res,String cur,int[] map,int len){
    if(cur.length()== len){
        res.add(new String(cur));
    }else {
        for(int i=0;i<map.length;i++){
            if(map[i] <= 0) continue;
            map[i] = map[i] - 2;
            cur = (char)i + cur + (char)i;
            DFS(res,cur,map,len);
            cur = cur.substring(1,cur.length()-1);
            map[i] = map[i]+2;
        }
    }
}
```

14.7 Alien Dictionary (Hard->leetcode NO.269)

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 words from the dictionary, wherewords are sorted lexicographically by the rules of this new language. Derive the order of letters in this language.

For example,
Given the following words in dictionary,

```
[
  "wrt",
  "wrf",
  "er",
  "ett",
  "rftt"
]
```

The correct order is: "wertf".

Note:

1. You may assume all letters are in lowercase.
2. If the order is invalid, return an empty string.
3. There may be multiple valid order of letters, return any one of them is fine.

> The key to this problem is:
>
> > *A topological ordering is possible if and only if the graph has no directed cycles*
>
> Let's build a graph and perform a DFS. The following states made things easier.
>
> 1. `visited[i] = -1`. Not even exist.
> 2. `visited[i] = 0`. Exist. Non-visited.
> 3. `visited[i] = 1`. Visiting.
> 4. `visited[i] = 2`. Visited.

Solution

```java
private final int N = 26;
public String alienOrder(String[] words) {
    boolean[][] adj = new boolean[N][N];
    int[] visited = new int[N];
    buildGraph(words, adj, visited);

    StringBuilder sb = new StringBuilder();
    for(int i = 0; i < N; i++) {
        if(visited[i] == 0) {                 // unvisited
            if(!dfs(adj, visited, sb, i)) return "";
        }
    }
    return sb.reverse().toString();
}
```

```
public boolean dfs(boolean[][] adj, int[] visited, StringBuilder sb, int i) {
    visited[i] = 1;                                 // 1 = visiting
    for(int j = 0; j < N; j++) {
        if(adj[i][j]) {                             // connected
            if(visited[j] == 1) return false;       // 1 => 1, cycle
            if(visited[j] == 0) {                   // 0 = unvisited
                if(!dfs(adj, visited, sb, j)) return false;
            }
        }
    }
    visited[i] = 2;                                // 2 = visited
    sb.append((char) (i + 'a'));
    return true;
}

public void buildGraph(String[] words, boolean[][] adj, int[] visited) {
    Arrays.fill(visited, -1);                      // -1 = not even existed
    for(int i = 0; i < words.length; i++) {
        for(char c : words[i].toCharArray()) visited[c - 'a'] = 0;
        if(i > 0) {
            String w1 = words[i - 1], w2 = words[i];
            int len = Math.min(w1.length(), w2.length());
            for(int j = 0; j < len; j++) {
                char c1 = w1.charAt(j), c2 = w2.charAt(j);
                if(c1 != c2) {
                    adj[c1 - 'a'][c2 - 'a'] = true;
                    break;
                }
            }
        }
    }
}
```

14.8 Closest Binary Search Tree Value II (Hard->leetcode No.272)

Given a non-empty binary search tree and a target value, find k values in the BST that are closest to the target.

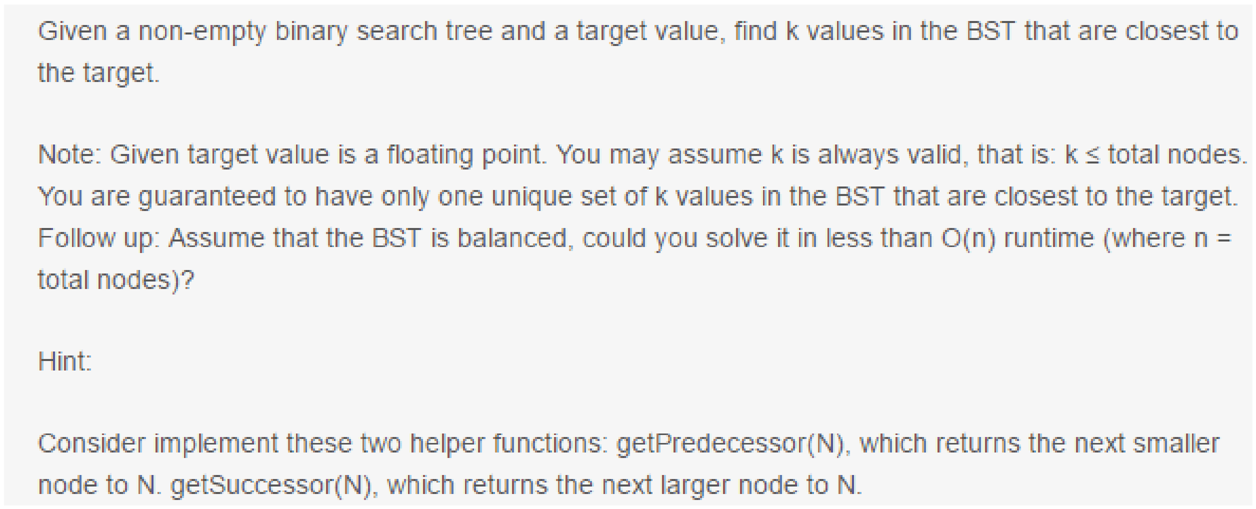
Note: Given target value is a floating point. You may assume k is always valid, that is: k ≤ total nodes. You are guaranteed to have only one unique set of k values in the BST that are closest to the target. Follow up: Assume that the BST is balanced, could you solve it in less than O(n) runtime (where n = total nodes)?

Hint:

Consider implement these two helper functions: getPredecessor(N), which returns the next smaller node to N. getSuccessor(N), which returns the next larger node to N.

```
public class Solution {
    public List<Integer> closestKValues(TreeNode root, double target, int k)
{
        Queue<Integer> klist = new LinkedList<Integer>();
        Stack<TreeNode> stk = new Stack<TreeNode>();
        // 迭代中序遍历二叉搜索树的代码
        while(root != null){
            stk.push(root);
            root = root.left;
        }
        while(!stk.isEmpty()){
            TreeNode curr = stk.pop();
            // 维护一个大小为 k 的队列
            // 队列不到 k 时直接加入
            if(klist.size() < k){
                klist.offer(curr.val);
            } else {
            // 队列到 k 时，判断下新的数是否更近，更近就加入队列并去掉队头
                int first = klist.peek();
                if(Math.abs(first - target) > Math.abs(curr.val - target)){
                    klist.poll();
                    klist.offer(curr.val);
                } else {
                // 如果不是更近则直接退出，后面的数只会更大
                    break;
                }
            }
            // 中序遍历的代码
            if(curr.right != null){
                curr = curr.right;
                while(curr != null){
                    stk.push(curr);
                    curr = curr.left;
                }
            }
        }
        // 强制转换成 List，是用 LinkedList 实现的所以可以转换
        return (List<Integer>)klist;
    }
}
```

14.9 Inorder Successor in BST (Medium->leetcode No.285)



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)?

Hint:

Consider implement these two helper functions: getPredecessor(N), which returns the next smaller node to N. getSuccessor(N), which returns the next larger node to N.

Solution

```
public class Solution{
        Public TreeNode inorderSuccessor(TreeNode root, TreeNode p){
                If(root==null || p==null) return null;
                If(p.right!=null) return findMin(p.right);  // case 1

                TreeNode succ = null;
                TreeNode q = root;
                While(q!=null){
                        If(q.val>p.val) {
                                Succ = q;
                                q = q.left;
                        }else if(q.val<p.val) q = q.right;
                        else break;
                }
                Return succ;
        }

        Public TreeNode findMin(TreeNode root){
                TreeNode p = root;
                While(p.left!=null) p = p.left;
                Return p;
        }
}
```

14.10 Word Pattern II (Hard->leetcode No.291)

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{
        Map<Character, String> map;
        Set<String> set;
        boolean result;

        public boolean wordPattenMatch(String pattern, String str){
                map = new HashMap<>();
                set = new HashSet<>();
                result = false;
                helper(pattern, str, 0.0);
                return result;
        }

        Public void helper(String pattern String str, int i, int j){
                If(i==pattern.length() && j==str.length()){
                        result = true;
                        return result;
                }
                If(i>=pattern.length() || j>=str.length()) return;
                char c = pattern.charAt(i);
                for(int cut = j+1; cut<str.length();cut++){
                        String substr = str.substring(j,cut);
                        If(!set.contains(substr) && !map.containsKey(c)){
                                set.add(substr);
                                map.put(c, substr);
                                helper(pattern, str, i+1, cut);
                                map.remove(c);
                                set.remove(substr);
                        }else if(map.containsKey(c) && map.get(c).equals(substr)) helper(pattern, str,
                        i+1,cut)
```

```
                    }
            }
    }
```

Solution

```
public class Solution {
Map<Character,String> map =new HashMap();
Set<String> set =new HashSet();
public boolean wordPatternMatch(String pattern, String str) {
    if(pattern.isEmpty()) return str.isEmpty();
    if(map.containsKey(pattern.charAt(0))){
        String value= map.get(pattern.charAt(0));
        if(str.length()<value.length() || !str.substring(0,value.length()).equals(value))
return false;
        if(wordPatternMatch(pattern.substring(1),str.substring(value.length()))) return true;
    }else{
        for(int i=1;i<=str.length();i++){
            if(set.contains(str.substring(0,i))) continue;
            map.put(pattern.charAt(0),str.substring(0,i));
            set.add(str.substring(0,i));
            if(wordPatternMatch(pattern.substring(1),str.substring(i))) return true;
            set.remove(str.substring(0,i));
            map.remove(pattern.charAt(0));
        }
    }
    return false;
}
```

14.11 Flip Game II (Medium->leetcode NO.294)

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.

Understand the problem:
At first glance, backtracking seems to be the only feasible solution to this problem. We can basically try every possible move for the first player (Let's call him 1P from now on), and recursively check if the second player 2P has any chance to win. If 2P is guaranteed to lose, then we know the current move 1P takes must be the winnin
move. The implementation is actually very simple:

```
public boolean canWin(String s) {
    if (s == null) return false;
    return canWin(s.toCharArray());
}
private boolean canWin(char[] chars) {
   for (int i = 0; i < chars.length - 1; i++) {
        if (chars[i] == '+' && chars[i+1] == '+') {
            chars[i] = chars[i+1] = '-';
            boolean win = !canWin(chars);
            chars[i] = chars[i+1] = '+'; //backtrack.
            if (win)
                return true;
        }
    }
    return false;
}
```

14.12 Binary Tree Lonest Consecutive Sequence (Medium->leetcode NO.298)

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 {
    private int max = 0;
    public int longestConsecutive(TreeNode root) {
        if(root == null) return 0;
        helper(root, 0, root.val);
        return max;
    }

    public void helper(TreeNode root, int cur, int target){
        if(root == null) return;
        if(root.val == target) cur++;
        else cur = 1;
        max = Math.max(cur, max);
        helper(root.left, cur, root.val + 1);
        helper(root.right, cur, root.val + 1);
    }
}
```

14.13 Generalized Abbreviation (Medium->leetcode NO.320)

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"]
```

Solution 1:

The idea is simple:

1. *for each character of the "word", we can either abbreviate it or not abbreviate it.*
2. *if we abbreviate, we can abbreviate one character, or two characters,...,or all characters of the remaining string.*
3. *if we abbreviate the first k characters, the k + 1 character must not be abbreviated; otherwise, we should abbreviate these k + 1 characters rather than just the first k*
4. *After each phase, deal with the remaining string using the same way*
5. *parse the length of abbreviation number by yourself can be a little faster than using java lib.*

```
public List<String> generateAbbreviations(String word) {
    List<String> res = new ArrayList<String>();
    if (word.length() == 0) {
        res.add(word);
        return res;
    }
    char[] c = word.toCharArray();
    char[] aux = new char[word.length()]; //auxiliary array for generating abbreviation
    generateAbbreviations(aux, 0, c, 0, res);
    return res;
}
private void generateAbbreviations(char[] aux, int p1, char[] c, int p2, List<String> res){
    // not abbr
    aux[p1] = c[p2];
    if (p2 == c.length - 1) res.add(new String(aux, 0, p1 + 1));
    else generateAbbreviations(aux, p1 + 1, c, p2 + 1, res);
    // abbr
    for (int i = 1; i <= c.length - p2; i++){
        int l = i;//length of abbreviation
        int p = p1;
        while(l >= 10){
            aux[p++] = (char)(l / 10 + '0');
            l = l - l / 10 * 10;
        }
        aux[p++] = (char)(l + '0');
        if (p2 + i == c.length) res.add(new String(aux, 0, p)); // abbr all the remaining
        else if (p2 + i == c.length - 1) { // abbr all the remaining except the last character
            aux[p++] = c[p2 + i];// the following character cannot be abbreviated
            res.add(new String(aux, 0, p));
        }
        else {
            aux[p++] = c[p2 + i];// the following character cannot be abbreviated
            generateAbbreviations(aux, p, c, p2 + i + 1, res);
        }
    }
}
```

Solution 2:

For each char `c[i]`, either abbreviate it or not.

1. Abbreviate: count accumulate `num` of abbreviating chars, but don't append it yet.
2. Not Abbreviate: append accumulated `num` as well as current char `c[i]`.
3. In the end append remaining `num`.
4. Using `StringBuilder` can decrease `36.4%` time.

This comes to the pattern I find powerful:

```
int len = sb.length(); // decision point
... backtracking logic ...
sb.setLength(len);     // reset to decision point
```

Similarly, check out remove parentheses and add operators.

```
public List<String> generateAbbreviations(String word) {
    List<String> res = new ArrayList<>();
    DFS(res, new StringBuilder(), word.toCharArray(), 0, 0);
    return res;
}

public void DFS(List<String> res, StringBuilder sb, char[] c, int i, int num) {
    int len = sb.length();
    if(i == c.length) {
        if(num != 0) sb.append(num);
        res.add(sb.toString());
    } else {
        DFS(res, sb, c, i + 1, num + 1);               // abbr c[i]

        if(num != 0) sb.append(num);                   // not abbr c[i]
        DFS(res, sb.append(c[i]), c, i + 1, 0);
    }
    sb.setLength(len);
}
```

14.14 Patching Array (Hard->leetcode No.330)

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.

Let miss be the smallest number that can not be formed by the sum of elements in the array. All elements in [0, miss) can be formed. The miss value starts with 1. If the next number nums[i] <=miss, then the boundary is increased to be [0, miss+nums[i]), because all numbers between the boundaries can be formed; if next number nums[i]>miss, that means there is a gap and we need to insert a number, inserting miss itself is a the choice because its boundary doubles and cover every number between the boundaries [0, miss+miss).

Here is an example.
Given nums=[1, 4, 10] and n=50.

```
miss=1;
i=0, nums[i]<=miss, then miss=1+1=2
i=1, nums[i]>2, then miss = miss+miss = 4
i=1, nums[i]<=miss, then miss = miss+num[i] = 8
i=2, nums[i]>miss, then miss = miss+miss = 16
i=2, nums[i]>miss, then miss = miss+miss = 32
i=2, nums[i]>miss, then miss = miss+miss = 64
64 > 50. Done! 4 elements are added!
```

Think about this example: nums = [1, 2, 3, 9]. We naturally want to iterate through nums from left to right and see what we would discover. After we encountered 1, we know 1...1 is patched completely. After encountered 2, we know 1...3 (1+2) is patched completely. After we encountered 3, we know 1...6 (1+2+3) is patched completely. After we encountered 9, the smallest number we can get is 9. So we must patch a new number here so that we don't miss 7, 8. To have 7, the numbers we can patch is 1, 2, 3 ... 7. Any number greater than 7 won't help here. Patching 8 will not help you get 7. So we have 7 numbers (1...7) to choose from. I hope you can see number 7 works best here because if we chose number 7, we can move all the way up to 1+2+3+7 = 13. (1...13 is patched completely) and it makes us reach n as quickly as possible. After we patched 7 and reach 13, we can consider last element 9 in nums. Having 9 makes us reach 13+9 = 22, which means 1...22 is completely patched. If we still did't reach n, we can then patch 23, which makes 1...45 (22+23) completely patched. We continue until we reach n.

```
public int minPatches(int[] nums, int n) {
    long miss = 1;
    int count = 0;
    int i = 0;

    while(miss <= n){
        if(i<nums.length && nums[i] <= miss){
            miss = miss + nums[i];
            i++;
        }else{
            miss += miss;
            count++;
        }
    }

    return count;
}
```

14.15 Verify Preorder Serialization of A binary Tree (Medium->leetcode NO.331)

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.

```
public boolean isValidSerialization(String preorder) {
    LinkedList<String> stack = new LinkedList<String>();
    String[] arr = preorder.split(",");

    for(int i=0; i<arr.length; i++){
        stack.add(arr[i]);

        while(stack.size()>=3
            && stack.get(stack.size()-1).equals("#")
            && stack.get(stack.size()-2).equals("#")
            && !stack.get(stack.size()-3).equals("#")){

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

            stack.add("#");
        }

    }

    if(stack.size()==1 && stack.get(0).equals("#"))
        return true;
    else
        return false;
}
Solution 2: without any space complexity
```

```
public boolean isValidSerialization(String preorder) {
    String[] chars = preorder.split(",");

    int sentinel = 0;
    int node = 0;
    for (int i = chars.length - 1; i >= 0; i--)
    {
        if (chars[i].equals("#"))sentinel++;
        else node ++;
        if (sentinel - node < 1) return false;
    }
    return sentinel - node == 1;
}
```

14.16 Largest BST Subtree (Medium->leetcode NO.333)

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.

Hint:

1. You can recursively use algorithm similar to 98. Validate Binary Search Tree at each node of the tree, which will result in O(nlogn) time complexity.

Follow up:

Can you figure out ways to solve it with O(n) time complexity?

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);
}
```

```
Solution 2:
```

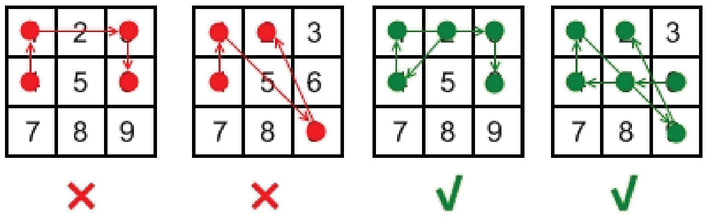
```
public class Solution {
public int largestBSTSubtree(TreeNode root) {
    int ret = isValidBST(root, Integer.MIN_VALUE, Integer.MAX_VALUE);
    if(ret >= 0){
        return ret;
    }
    return Math.max(largestBSTSubtree(root.left), largestBSTSubtree(root.right));
}
private int isValidBST(TreeNode node, int min, int max){
    if(node == null){
        return 0;
    }
    if(node.val < min || node.val > max){
        return -1;
    }
    int left = isValidBST(node.left, min, node.val);
    int right = isValidBST(node.right, node.val, max);
    return (left >= 0 && right >= 0)? left + right + 1 : -1;
}
```

14.17 Android Unlock Patterns (Medium->leetcode NO.351)

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.

Credits:
Special thanks to @elmirap for adding this problem and creating all test cases.

Solution 1

```
private int[][] jumps;
private boolean[] visited;
```

```
public int numberOfPatterns(int m, int n) {
    jumps = new int[10][10];
    jumps[1][3] = jumps[3][1] = 2;
    jumps[4][6] = jumps[6][4] = 5;
    jumps[7][9] = jumps[9][7] = 8;
    jumps[1][7] = jumps[7][1] = 4;
    jumps[2][8] = jumps[8][2] = 5;
    jumps[3][9] = jumps[9][3] = 6;
    jumps[1][9] = jumps[9][1] = jumps[3][7] = jumps[7][3] = 5;
    visited = new boolean[10];
    int count = 0;
    count += DFS(1, 1, 0, m, n) * 4; // 1, 3, 7, 9 are symmetrical
    count += DFS(2, 1, 0, m, n) * 4; // 2, 4, 6, 8 are symmetrical
    count += DFS(5, 1, 0, m, n);
    return count;
}

private int DFS(int num, int len, int count, int m, int n) {
    if (len >= m) count++; // only count if moves are larger than m
    len++;
    if (len > n) return count;
    visited[num] = true;
    for (int next = 1; next <= 9; next++) {
        int jump = jumps[num][next];
        if (!visited[next] && (jump == 0 || visited[jump])) {
            count = DFS(next, len, count, m, n);
        }
    }
    visited[num] = false; // backtracking
    return count;
}
```

Solution 2:

```
public class Solution {
    // cur: the current position
    // remain: the steps remaining
    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] is not visited and (two numbers are adjacent or skip number is
already visited)
            if(!vis[i] && (skip[i][cur] == 0 || (vis[skip[i][cur]]))) {
                rst += DFS(vis, skip, i, remain - 1);
            }
        }
        vis[cur] = false;
        return rst;
    }

    public int numberOfPatterns(int m, int n) {
        // Skip array represents number to skip between two pairs
        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;
        // DFS search each length from m to n
        for(int i = m; i <= n; ++i) {
            rst += DFS(vis, skip, 1, i - 1) * 4;    // 1, 3, 7, 9 are symmetric
            rst += DFS(vis, skip, 2, i - 1) * 4;    // 2, 4, 6, 8 are symmetric
            rst += DFS(vis, skip, 5, i - 1);        // 5
        }
        return rst;
    }
}
```

14.18 Design Snake Game (Medium->leetcode NO.353)

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)
```

```
Public class SnakeGame{
class Position{
        int x;
        int y;
        public Position(int x,int y){
            this.x = x;
            this.y = y;
        }
        public boolean isEqual(Position p){
            return this.x==p.x && this.y == p.y ;
        }
    }
    int len;
    int rows ,cols;

    int[][] food;
    LinkedList<Position> snake;

    /** Initialize your data structure here.
```

```
        @param width - screen width
        @param height - screen height
        @param food - A list of food positions
        E.g food = [[1,1], [1,0]] means the first food is positioned at [1,1], the second is
at [1,0]. */
    public SnakeGame(int width, int height, int[][] food) {
        this.rows = height;
        this.cols = width;
        this.food = food;

        snake = new LinkedList<Position>();
        snake.add(new Position(0,0));
        len = 0;
    }

    /** Moves the snake.
        @param direction - 'U' = Up, 'L' = Left, 'R' = Right, 'D' = Down
        @return The game's score after the move. Return -1 if game over.
        Game over when snake crosses the screen boundary or bites its body. */
    public int move(String direction) {
        //if(len>=food.length) return len;

        Position cur = new Position(snake.get(0).x,snake.get(0).y);

        switch(direction){
        case "U":
            cur.x--;  break;
        case "L":
            cur.y--; break;
        case "R":
            cur.y++;   break;
        case "D":
            cur.x++;   break;
        }

        if(cur.x<0 || cur.x>= rows || cur.y<0 || cur.y>=cols) return -1;


        for(int i=1;i<snake.size()-1;i++){
            Position next = snake.get(i);
            if(next.isEqual(cur)) return -1;
        }
        snake.addFirst(cur);
        if(len<food.length){
            Position p = new Position(food[len][0],food[len][1]);
            if(cur.isEqual(p)){
                len++;
            }
        }
        while(snake.size()>len+1) snake.removeLast();
        return len;
    }
}

/**
 * Your SnakeGame object will be instantiated and called as such:
 * SnakeGame obj = new SnakeGame(width, height, food);
 * int param_1 = obj.move(direction);
 */
```

Solution 2:

```
public class SnakeGame {
//Here we use int to keep the location of snake body, value = x*width + y
//And LinkedList has the method boolean contains, to check if collide with snake body
Deque<Integer> snakeBody = new LinkedList();
int width = 0;
int height = 0;
int[][] food;
int count = 0;
boolean gameOver = false;

public SnakeGame(int width, int height, int[][] food) {
    this.width = width;
    this.height = height;
    this.food = food;
    snakeBody.addLast(0);
}

/** Moves the snake.
    @param direction - 'U' = Up, 'L' = Left, 'R' = Right, 'D' = Down
    @return The game's score after the move. Return -1 if game over.
    Game over when snake crosses the screen boundary or bites its body. */
public int move(String direction) {
    if(gameOver) return -1;
    int curValue = snakeBody.getFirst();
    int lastValue = snakeBody.removeLast();
    int[] head = new int[2];
    head[0] = curValue/width;
    head[1] = curValue%width;
    switch(direction) {
        case "U": head[0]--;break;
        case "L": head[1]--;break;
        case "R": head[1]++;break;
        case "D": head[0]++;break;
    }

if(head[0]<0||head[1]<0||head[0]>=height||head[1]>=width||snakeBody.contains(valueOf(head))) {
```

```
            gameOver = true;
            return -1;
        }
        snakeBody.addFirst(valueOf(head));
        if(count<food.length&&food[count][0]==head[0]&&food[count][1]==head[1]) {
            snakeBody.addLast(lastValue);
            count++;
        }
        return snakeBody.size()-1;
}

public int valueOf(int[] head) {
        return head[0]*width+head[1];
}
}
```

14.19 Logger Rate Limiter (Easy->leetcode NO.359)

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

Solution: non-concurrency

```
public class Logger {
HashMap<String,Integer> map;
/** Initialize your data structure here. */
public Logger() {
    map=new HashMap<>();
}

/** Returns true if the message should be printed in the given timestamp, otherwise returns
false. The timestamp is in seconds granularity. */
public boolean shouldPrintMessage(int timestamp, String message) {
//update timestamp of the message if the message is coming in for the first time,or the last
coming time is earlier than 10 seconds from now
    if(!map.containsKey(message)||timestamp-map.get(message)>=10){
        map.put(message,timestamp);
        return true;
    }
    return false;
}
```

Soltion 2: with Concurrency Class

```
import java.util.concurrent.*;

public class Logger {
    ConcurrentHashMap<String, Integer> lastPrintTime;

    /** Initialize your data structure here. */
    public Logger() {
        lastPrintTime = new ConcurrentHashMap<String, Integer>();
    }

    /** Returns true if the message should be printed in the given timestamp, otherwise
returns false. The timestamp is in seconds granularity. */
    public boolean shouldPrintMessage(int timestamp, String message) {
        Integer last = lastPrintTime.get(message);

        return last == null && lastPrintTime.putIfAbsent(message, timestamp) == null
               || last != null && timestamp - last >= 10 && lastPrintTime.replace(message,
last, timestamp);

    }
}
```

14.20 Sort Transformed Array (Medium ->leetcode NO.360)

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]
```

Solution

```
public int[] sortTransformedArray(int[] nums, int a, int b, int c) {
    Integer[] transformed = Arrays.stream(nums)
                                  .map(x -> a*x*x + b*x + c)
                                  .boxed().toArray(Integer[]::new);
    Arrays.sort(transformed);
    return Arrays.stream(transformed).mapToInt(i -> i).toArray();
}
```

the problem seems to have many cases a>0, a=0,a<0, (when a=0, b>0, b<0). However, they can be combined into just 2 cases: a>0 or a<0

1.a>0, two ends in original array are bigger than center if you learned middle school math before.

2.a<0, center is bigger than two ends.

so use two pointers i, j and do a merge-sort like process. depending on sign of a, you may want to start from the beginning or end of the transformed array. For a==0 case, it does not matter what b's sign is. The function is monotonically increasing or decreasing. you can start with either beginning or end.

```
public class Solution {
    public int[] sortTransformedArray(int[] nums, int a, int b, int c) {
        int n = nums.length;
        int[] sorted = new int[n];
        int i = 0, j = n - 1;
        int index = a >= 0 ? n - 1 : 0;
        while (i <= j) {
            if (a >= 0) {
                sorted[index--] = quad(nums[i], a, b, c) >= quad(nums[j], a, b, c) ?
quad(nums[i++], a, b, c) : quad(nums[j--], a, b, c);
            } else {
                sorted[index++] = quad(nums[i], a, b, c) >= quad(nums[j], a, b, c) ?
quad(nums[j--], a, b, c) : quad(nums[i++], a, b, c);
            }
        }
        return sorted;
    }

    private int quad(int x, int a, int b, int c) {
        return a * x * x + b * x + c;
    }
}
```

14.21 Bomb Enemy (medium->leetcode No.361)

Given a 2D grid, each cell is either a wall `'Y'`, an enemy `'X'` 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 X 0 0
X 0 Y X
0 X 0 0

return 3. (Placing a bomb at (1,1) kills 3 enemies)
```

```java
public int maxKilledEnemies(char[][] grid) {
   if(grid == null || grid.length == 0 ||  grid[0].length == 0) return 0;
   int max = 0;
   int row = 0;
   int[] col = new int[grid[0].length];
   for(int i = 0; i<grid.length; i++){
       for(int j = 0; j<grid[0].length;j++){
           if(grid[i][j] == 'W') continue;
           if(j == 0 || grid[i][j-1] == 'W'){
                row = killedEnemiesRow(grid, i, j);
           }
           if(i == 0 || grid[i-1][j] == 'W'){
                col[j] = killedEnemiesCol(grid,i,j);
           }
           if(grid[i][j] == '0'){
               max = (row + col[j] > max) ? row + col[j] : max;
           }
       }

   }

   return max;
}
```

```java
//calculate killed enemies for row i from column j
private int killedEnemiesRow(char[][] grid, int i, int j){
    int num = 0;
    while(j <= grid[0].length-1 && grid[i][j] != 'W'){
        if(grid[i][j] == 'E') num++;
        j++;
    }
    return num;
}
//calculate killed enemies for  column j from row i
private int killedEnemiesCol(char[][] grid, int i, int j){
    int num = 0;
    while(i <= grid.length -1 && grid[i][j] != 'W'){
        if(grid[i][j] == 'E') num++;
        i++;
    }
    return num;
}
```