

Contents

1.1 Regular Expression Matching (Medium->Leetcode No.10)	3
1.2 Substring With Concatenation for All Words (hard -> LeetCode No.30)	4
1.3 Longest Valid Parentheses (hard -> LeetCode No.32)	5
1.4 Wildcard Matching (hard -> LeetCode No.44)	6
1.5 Valid Number (hard -> LeetCode No.65)	7
1.6 Text Justification (hard -> LeetCode No.68)	8
1.7 Edit Distance (hard -> LeetCode No.72)	9
1.8 Minimum Window Substring (hard ->Leetcode No.76)	11
1.9 Scramble String (hard -> Leetcode No.87)	12
1.10 Interleaving String (hard->Leetcode No.97)	13
1.11 Distinct Subsequences (hard->Leetcode No.115)	15
1.12 Word Ladder II (hard -> Leetcode No.126)	15
1.13 Shortest Palindrome (hard->Leetcode No.214)	17
1.14 Integer to English Words (hard->Leetcode No.273)	18
1.15 Palindrome Paris (hard->Leetcode No.336)	19
1.16 Median of Two Sorted Arrays (hard->Leetcode No.4)	21
1.17 Reverse Nodes in K-Group (Hard->Leetcode No.25)	22
1.18 Search in Rotated Sorted Array (hard -> leetcode No.33)	23
1.19 Sudoku Solver (hard->leetcode No.37)	24
1.20 First Missing positive (hard->leetcode No.41)	25
1.21 Trapping Rain Water (hard->leetcode No.42)	26
1.22 Jump Game II (hard->Leetcode No.45)	27
1.23 N-Queens (hard->leetcode No.51)	27
1.24 N-Queens II(hard->leetcode No.52)	29
1.25 Merge Intervals (hard->Leetcode No.56)	31
1.26 Insert Interval (hard->leetcode No.57)	32
1.27 Largest Rectangle in Histogram (hard->leetcode No.84)	32
1.28 Maximal Rectangle (hard->leetcode NO.85)	34
1.29 Recover Binary Search Tree (hard->leetcode No.99)	35
1.30 Populating Next Right Pointers in Each Node II (hard->leetcode No.117)	36
1.31 Best Time to Buy and Sell Stock III (Hard->leetcode No.123)	37
1.32 Binary Tree Maximum Path Sum (Hard->leetcode No.124)	38
1.33 Longest Consecutive Sequence (hard->leetcode No.128)	39
1.34 Palindrome Partitioning II (hard-> leetcode No.132)	40
1.35 Candy (hard->leetcode No.135)	40
1.36 Copy List with Random Pointer (hard->leetcode No.138)	41
1.37 Word Break II (hard->leetcode No.140)	42
1.38 Binary Tree Postorder Traversal (hard->leetcode No.145)	43
1.39 LRU Cache (hard->leetcode No.146)	44
1.40 Max Points on a Line(hard->leetcode No.149)	46
1.41 Find Minimum in Rotated Sorted Array II (Hard -> leetcode No.154)	47
1.42 Maximum Gap (hard->leetcode No.164)	48
1.43 Dungeon Game (hard->leetcode No.174)	49
1.44 Best Time to Buy and Sell Stock IV (Hard->leetcode No.188)	50
1.45 Word Search II (hard->leetcode No.212)	51
1.46 The skyline Problem (hard->leetcode No.218)	53

1.47 Basic Calculator (hard->leetcode No.224)	55
1.48 Number of Digit One (hard->leetcode No.233)	57
1.49 Sliding Window Maximum (hard->leetcode No.239)	60
1.50 Paint House II (Hard->leetcode No.265)	61
1.51 Expression Add Operators (hard->leetcode No.282)	61
1.52 Find the Duplicate Number (hard->leetcode No. 287)	63
1.53 Find Median From data Stream (hard-> leetcode NO.295)	64
1.54 Serialize and Deserialize Binary Tree(hard->leetcode No.297)	66
1.55 Remove Invalid Parentheses (hard->leetcode No.301)	68
1.56 Burst Balloons (hard->leetcode No.312)	69
1.57 Count of Smaller Numbers after self(hard->leetcode No.315).....	70
1.58 Remove Duplicate Letters(hard->leetcode No.316)	71
1.59 Create maximum Number (hard->leetcode No.321)	72
1.60 Count of Range Sum (hard->leetcode No. 327).....	74
1.61 Longest Increasing Path in a Matrix(hard->leetcode No.329)	74
1.62 Self Crossing (hard -> leetcode No.335)	77
1.63 Data Stream as Disjoint intervals (hard->leetcode No.352)	79
1.64 Russian Doll Envelopes (hard->leetcode No.354)	80

1.1 Regular Expression Matching (Medium->Leetcode No.10)

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

```
'.' Matches any single character.
'*' Matches zero or more of the preceding element.

The matching should cover the entire input string (not partial).

The function prototype should be:
bool isMatch(const char *s, const char *p)

Some examples:
isMatch("aa","a") → false
isMatch("aa","aa") → true
isMatch("aaa","aa") → false
isMatch("aa", "a*") → true
isMatch("aa", ".*") → true
isMatch("ab", ".*") → true
isMatch("aab", "c*a*b") → true
```

Solution:

The problem should be simplified to handle 2 basic cases:

- » the second char of pattern is `"*"`
- » the second char of pattern is not `"*"`

For the 1st case, if the first char of pattern is not `"."`, the first char of pattern and string should be the same. Then continue to match the remaining part.

For the 2nd case, if the first char of pattern is `"."` or first char of pattern == the first i char of string, continue to match the remaining part.

```
public class Solution {
    public boolean isMatch(String s, String p) {
        // consider the base case when p.length()==0
        if(p.length()==0) return s.length()==0;
        // consider the special case when p.length()==1
        if(p.length()==1){
            if(s.length()<1) return false;
            else if(s.charAt(0)!=p.charAt(0) && p.charAt(0)!='.') return false;
            else return isMatch(s.substring(1), p.substring(1));
        }
        // case1: when the second digit of p is not *
        if(p.charAt(1)!='*'){
            if(s.length()<1) return false;
            if(s.charAt(0)!=p.charAt(0) && p.charAt(0)!='.') return false;
            else return isMatch(s.substring(1), p.substring(1));
        }
    }
}
```

```

    }else{
        // case 2: when the second digit of p is *, this is complex case
        if(isMatch(s, p.substring(2))) return true; /* represents the 0 element
        int i=0;
        while(i<s.length() && (s.charAt(i)==p.charAt(0) || p.charAt(0)=='.')){
            if(isMatch(s.substring(i+1), p.substring(2))) return true;
            i++;
        }
        return false;
    }
}
}
}

```

1.2 Substring With Concatenation for All Words (hard -> LeetCode No.30)

You are given a string, **s**, and a list of words, **words**, that are all of the same length. Find all starting indices of substring(s) in **s** that is a concatenation of each word in **words** exactly once and without any intervening characters.

For example, given:

s: "barfoothefoobarman"

words: ["foo", "bar"]

You should return the indices: [0,9] .

(order does not matter).

Solution:

```

public class Solution {
    public List<Integer> findSubstring(String s, String[] words) {
        // assertion
        List<Integer> result = new ArrayList<>();
        if(s==null || s.length()==0 || words==null || words.length==0)
            return result;
        Map<String, Integer> map = new HashMap<>();
        for(String word:words){
            if(map.containsKey(word)) map.put(word, map.get(word)+1);
            else map.put(word,1);
        }

        int len = words[0].length();

        for(int j = 0;j<len;j++){ // here we use just len loops because the word in words has len length
            Map<String, Integer> curr = new HashMap<>();
            int start = j; // the beginning index of start
            int count = 0; // count the number of qualified words so far
            for(int i=j;i<=s.length()-len;i=i+len){
                String tmp = s.substring(i, i+len);
                if(map.containsKey(tmp)){

```

```

        if(curr.containsKey(tmp)) curr.put(tmp, curr.get(tmp)+1);
        else curr.put(tmp, 1);

        count++;

        while(curr.get(tmp)>map.get(tmp)){
            String left = s.substring(start, start+len);
            curr.put(left, curr.get(left)-1);
            count--;
            start = start+len;
        }

        if(count==words.length){
            result.add(start);
            String left = s.substring(start, start+len);
            curr.put(left, curr.get(left)-1);
            count--;
            start = start+len;
        }
        else{
            curr.clear();
            start = i+len;
            count = 0;
        }
    }
}
return result;
}
}

```

1.3 Longest Valid Parentheses (hard -> LeetCode No.32)

Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed) parentheses substring.

For "()", the longest valid parentheses substring is "()", which has length = 2.

Another example is "()()()", where the longest valid parentheses substring is "()()()", which has length = 4.

```

public class Solution {
    public int longestValidParentheses(String s) {
        if(s==null || s.length()==0) return 0;
        int maxLen = 0;
        int accumulatedLen = 0;
        Stack<Integer> stack = new Stack<>();
        for(int i=0;i<s.length();i++){
            char c = s.charAt(i);
            if(c=='(') stack.push(i);
            else{

```

```

        if(stack.size()==0) accumulatedLen = 0;
        else{
            int matchedPos = stack.pop();
            int matchedLen = i-matchedPos + 1;
            if(stack.size()==0){
                accumulatedLen+=matchedLen;
                matchedLen = accumulatedLen;
            }else matchedLen = i-stack.peek();
            maxLen = Math.max(maxLen, matchedLen);
        }
    }
}
return maxLen;
}
}

```

1.4 Wildcard Matching (hard -> LeetCode No.44)

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

'?' Matches any single character.
 '*' Matches any sequence of characters (including the empty sequence).

The matching should cover the entire input string (not partial).

The function prototype should be:

```
bool isMatch(const char *s, const char *p)
```

Some examples:

```
isMatch("aa","a") → false
isMatch("aa","aa") → true
isMatch("aaa","aa") → false
isMatch("aa","*") → true
isMatch("aa","a*") → true
isMatch("ab","?*") → true
isMatch("aab","c*a*b") → false
```

```

public class Solution {
    public boolean isMatch(String s, String p) {
        int i = 0;
        int j = 0;
        int starIndex = -1; // map to the index of star in p
        int iIndex = -1; // the index of the element that maps to the star in p
        while(i<s.length()){
            if(j<p.length() && (s.charAt(i)==p.charAt(j) || p.charAt(j)=='?')){
                i++;
                j++;
            }else if(j<p.length() && p.charAt(j)=='*'){
                starIndex = j;
            }
        }
    }
}

```

```

        iIndex = i;
        j++;
    }else if(starIndex!=-1){
        j = starIndex+1;
        i = iIndex+1;
        iIndex++;
    }else return false;
    }
    while(j<p.length() && p.charAt(j)=='*') j++;
    return j==p.length();
}
}

```

1.5 Valid Number (hard -> LeetCode No.65)

Validate if a given string is numeric.

Some examples:

"0" => true

" 0.1 " => true


"abc" => false

"1 a" => false

"2e10" => true

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

Update (2015-02-10):

The signature of the `C++` function had been updated. If you still see your function signature accepts a `const char *` argument, please click the reload button  to reset your code definition.

```

public class Solution {
    public boolean isNumber(String s) {
        int len = s.length();
        int i = 0;
        int e = len-1;
        while(i<=e && Character.isWhitespace(s.charAt(i))) i++;
        if(i>len-1) return false;
        while(i<=e && Character.isWhitespace(s.charAt(e))) e--;
        if(s.charAt(i)=='+' || s.charAt(i)=='-') i++;
        boolean num = false;
        boolean dot = false;
        boolean exp = false;
        while(i<=e){
            char c = s.charAt(i);
            if(Character.isDigit(c)) num = true;

```

```

        else if(c=='.'){
            if(exp || dot) return false;
            dot = true;
        }else if(c=='e'){
            if(exp || num==false) return false;
            exp = true;
            num =false;
        }else if(c=='+' || c=='-'){
            if(s.charAt(i-1)!='e') return false;
        }else return false;
        i++;
    }
    return num;
}
}

```

1.6 Text Justification (hard -> LeetCode No.68)

Given an array of words and a length L , format the text such that each line has exactly L characters and is fully (left and right) justified.

You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces ' ' when necessary so that each line has exactly L characters.

Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line do not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right.

For the last line of text, it should be left justified and no extra space is inserted between words.

For example,

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

L: 16 .

Return the formatted lines as:

```

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

```

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

```

public class Solution {
    public List<String> fullJustify(String[] words, int maxWidth) {
        List<String> res = new ArrayList<>();
        if(words==null || words.length==0) return res;
        int count = 0;
    }
}

```



```

int lastIndex = 0;
for(int i = 0; i < words.length; i++){
    count += words[i].length();
    if(count + i - lastIndex > maxWidth){
        int wordLen = count - words[i].length();
        int spaceLen = maxWidth - wordLen;
        int eachLen = 1;
        int extraLen = 0;
        if(i - 1 - lastIndex > 0){
            eachLen = spaceLen / (i - 1 - lastIndex);
            extraLen = spaceLen % (i - 1 - lastIndex);
        }

        StringBuilder sb = new StringBuilder();
        for(int j = lastIndex; j < i - 1; j++){
            sb.append(words[j]);
            int tmp = 0;
            while(tmp < eachLen) {
                sb.append(" ");
                tmp++;
            }
            if(extraLen > 0){
                sb.append(" ");
                extraLen--;
            }
        }
        sb.append(words[i - 1]);
        while(sb.length() < maxWidth) sb.append(" ");
        res.add(sb.toString());
        count = words[i].length();
        lastIndex = i;
    }
}

StringBuilder sb = new StringBuilder();
for(int i = lastIndex; i < words.length - 1; i++) sb.append(words[i] + " ");
sb.append(words[words.length - 1]);
while(sb.length() < maxWidth){
    sb.append(" ");
}
res.add(sb.toString());
return res;
}
}

```

1.7 Edit Distance (hard -> LeetCode No.72)

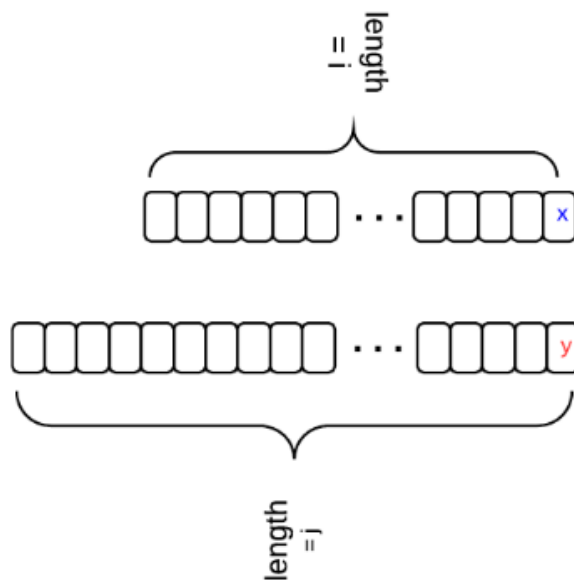
Given two words *word1* and *word2*, find the minimum number of steps required to convert *word1* to *word2*. (each operation is counted as 1 step.)

You have the following 3 operations permitted on a word:

- a) Insert a character
- b) Delete a character
- c) Replace a character

Let $dp[i][j]$ stands for the edit distance between two strings with length i and j , i.e., $word1[0, \dots, i-1]$ and $word2[0, \dots, j-1]$.

There is a relation between $dp[i][j]$ and $dp[i-1][j-1]$. Let's say we transform from one string to another. The first string has length i and its last character is "x"; the second string has length j and its last character is "y". The following diagram shows the relation.



1. if $x == y$, then $dp[i][j] == dp[i-1][j-1]$
2. if $x \neq y$, and we insert y for $word1$, then $dp[i][j] = dp[i][j-1] + 1$
3. if $x \neq y$, and we delete x for $word1$, then $dp[i][j] = dp[i-1][j] + 1$
4. if $x \neq y$, and we replace x with y for $word1$, then $dp[i][j] = dp[i-1][j-1] + 1$
5. When $x \neq y$, $dp[i][j]$ is the min of the three situations.

Initial condition:

$dp[i][0] = i$, $dp[0][j] = j$

```
public class Solution {
```

```

public int minDistance(String word1, String word2) {
    int len1 = word1.length();
    int len2 = word2.length();
    int[][] dp = new int[len1+1][len2+1];
    for(int i=0;i<=len1;i++) dp[i][0]=i;
    for(int i=0;i<=len2;i++) dp[0][i]=i;
    for(int i=0;i<len1;i++){
        char c1 = word1.charAt(i);
        for(int j = 0;j<len2;j++){
            char c2 = word2.charAt(j);
            if(c1==c2) dp[i+1][j+1]=dp[i][j];
            else{
                int delete = dp[i+1][j]+1;
                int insert = dp[i][j+1]+1;
                int replace = dp[i][j]+1;
                dp[i+1][j+1]=Math.min(delete, Math.min(insert, replace));
            }
        }
    }
    return dp[len1][len2];
}

```

1.8 Minimum Window Substring (hard ->Leetcode No.76)

Given a string *S* and a string *T*, find the minimum window in *S* which will contain all the characters in *T* in complexity $O(n)$.

For example,

S = "ADOBECODEBANC"

T = "ABC"

Minimum window is "BANC" .

Note:

If there is no such window in *S* that covers all characters in *T*, return the empty string "" .

If there are multiple such windows, you are guaranteed that there will always be only one unique minimum window in *S*.

```

public class Solution {
    public String minWindow(String s, String t) {
        int[] sarr = new int[256];
        int[] tarr = new int[256];
        for(int i=0;i<t.length();i++) tarr[t.charAt(i)]++;
        String res = "";
        int rPointer = 0;
    }
}

```

```

int min = Integer.MAX_VALUE;
for(int i=0;i<s.length();i++){
    char c = s.charAt(i);
    while(rPointer<s.length() && !sContaint(sarr, tarr)){
        sarr[s.charAt(rPointer)]++;
        rPointer++;
    }
    if(sContaint(sarr, tarr) && min>rPointer-i+1){
        min = rPointer-i+1;
        res = s.substring(i,rPointer);
    }
    sarr[c]--;
}
return res;
}

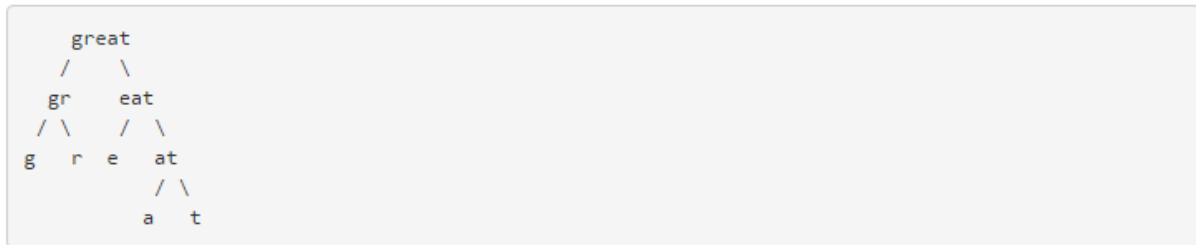
private boolean sContaint(int[] s, int[] t){
    for(int i=0;i<t.length;i++){
        if(t[i]>s[i]) return false;
    }
    return true;
}
}

```

1.9 Scramble String (hard -> Leetcode No.87)

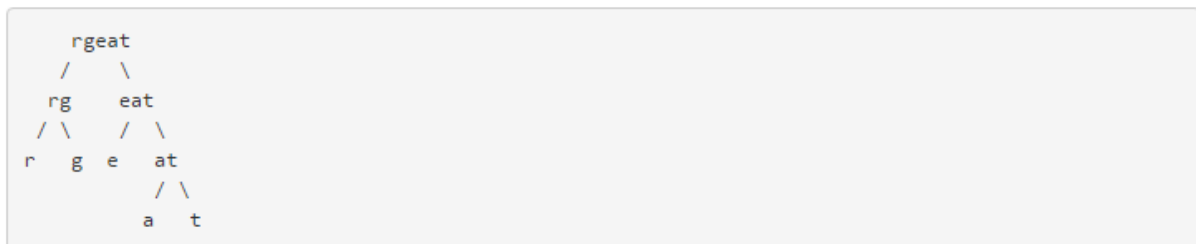
Given a string *s1*, we may represent it as a binary tree by partitioning it to two non-empty substrings recursively.

Below is one possible representation of *s1* = "great":



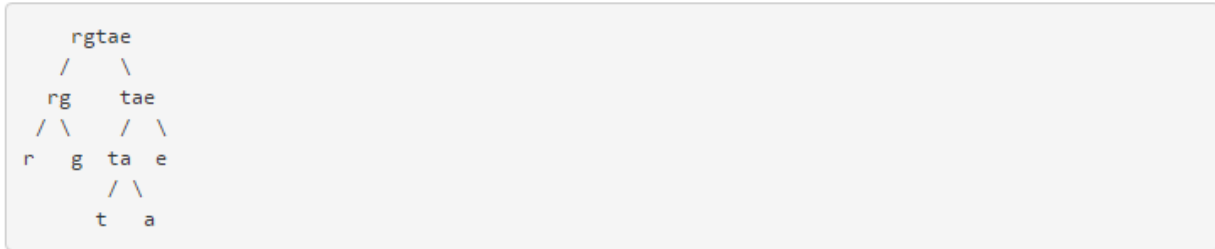
To scramble the string, we may choose any non-leaf node and swap its two children.

For example, if we choose the node "gr" and swap its two children, it produces a scrambled string "rgeat".



We say that "rgeat" is a scrambled string of "great".

Similarly, if we continue to swap the children of nodes "eat" and "at", it produces a scrambled string "rgtae".



We say that "rgtae" is a scrambled string of "great".

Given two strings *s1* and *s2* of the same length, determine if *s2* is a scrambled string of *s1*.

[Subscribe](#) to see which companies asked this question

```
public class Solution {
    public boolean isScramble(String s1, String s2) {
        if(s1.length()!=s2.length()) return false;
        if(s1.length()==0 || s1.equals(s2)) return true;
        char[] s1_arr = s1.toCharArray();
        char[] s2_arr = s2.toCharArray();
        Arrays.sort(s1_arr);
        Arrays.sort(s2_arr);
        if(!new String(s1_arr).equals(new String(s2_arr))) return false;
        for(int i=1;i<s1.length();i++){
            String s11 = s1.substring(0,i);
            String s12 = s1.substring(i, s1.length());
            String s21 = s2.substring(0,i);
            String s22 = s2.substring(i,s2.length());
            String s23 = s2.substring(0,s2.length()-i);
            String s24 = s2.substring(s2.length()-i,s2.length());
            if(isScramble(s11, s21) && isScramble(s12, s22)) return true;
            if(isScramble(s11,s24) && isScramble(s12,s23)) return true;
        }
        return false;
    }
}
```

1.10 Interleaving String (hard->Leetcode No.97)

Given s_1 , s_2 , s_3 , find whether s_3 is formed by the interleaving of s_1 and s_2 .

For example,

Given:

$s_1 = \text{"aabcc"}$,

$s_2 = \text{"dbbca"}$,

When $s_3 = \text{"aadbcbcbac"}$, return true.

When $s_3 = \text{"aadbbaaccc"}$, return false.

Solution 1: Recursive is TIME EXCEED LIMITED

```
public class Solution {
    public boolean isInterleave(String s1, String s2, String s3) {
        if(s1.length()+s2.length()!=s3.length()) return false;
        return rec(s1, 0, s2, 0, s3,0);
    }
    private boolean rec(String s1, int p1, String s2, int p2, String s3, int p3){
        if(p3==s3.length()) return true;
        if(p2==s2.length()) return s1.substring(p1).equals(s3.substring(p3));
        if(p1==s1.length()) return s2.substring(p2).equals(s3.substring(p3));
        if(s1.charAt(p1)==s3.charAt(p3) && s2.charAt(p2)==s3.charAt(p3))
            return rec(s1, p1+1,s2,p2,s3,p3+1) || rec(s1, p1,s2,p2+1,s3,p3+1);
        else if(s1.charAt(p1)==s3.charAt(p3)) return rec(s1, p1+1,s2,p2,s3,p3+1);
        else if (s2.charAt(p2)==s3.charAt(p3)) return rec(s1, p1,s2,p2+1,s3,p3+1);
        else return false;
    }
}
```


Solution 2: DP

$dp[i][j]$ 表示 s_1 取前 i 位, s_2 取前 j 位, 是否能组成 s_3 的前 $i+j$ 位

举个例子, 注意左上角那一对箭头指向的格子 $dp[1][1]$. 表示 s_1 取第1位a, s_2 取第1位d, 是否能组成 s_3 的前两位aa

从 $dp[0][1]$ 往下的箭头表示, s_1 目前取了0位, s_2 目前取了1位, 我们添加 s_1 的第1位, 看看它是不是等于 s_3 的第2位, ($i+j$ 位)

从 $dp[1][0]$ 往右的箭头表示, s_1 目前取了1位, s_2 目前取了0位, 我们添加 s_2 的第1位, 看看它是不是等于 s_3 的第2位, ($i+j$ 位)

 快速回复

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

```

        dp[i][0] = dp[i-1][0] && (s1.charAt(i-1)==s3.charAt(i-1));
    for(int i=1;i<=s2.length();i++)
        dp[0][i] = dp[0][i-1] && (s2.charAt(i-1)==s3.charAt(i-1));
    for(int i=1;i<=s1.length();i++){
        for(int j=1;j<=s2.length();j++){
            if (s1.charAt(i - 1) == s3.charAt(i + j - 1) && dp[i - 1][j]) {
                dp[i][j] = true;
            }
            if (s2.charAt(j - 1) == s3.charAt(i + j - 1) && dp[i][j - 1]) {
                dp[i][j] = true;
            }
        }
    }
    return dp[s1.length()][s2.length()];
}
}

```

1.11 Distinct Subsequences (hard->Leetcode No.115)

Given a string **S** and a string **T**, count the number of distinct subsequences of **T** in **S**.

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ACE" is a subsequence of "ABCDE" while "AEC" is not).

Here is an example:

S = "rabbbit", **T** = "rabbit"

Return 3.

When you see string problem that is about subsequence or matching, dynamic programming method should come to mind naturally. The key is to find the initial and changing condition.

1.12 Word Ladder II (hard -> Leetcode No.126)

Given two words (start and end), and a dictionary, find all shortest transformation sequence(s) from start to end, such that: 1) Only one letter can be changed at a time, 2) Each intermediate word must exist in the dictionary.

For example, given: start = "hit", end = "cog", and dict = ["hot","dot","dog","lot","log"], return:

```

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

```

```

class WordNode{
    String word;
    int numStep;
    WordNode pre;
    public WordNode(String word, int numStep, WordNode pre){
        this.word = word;
        this.numStep = numStep;
        this.pre = pre;
    }
}

public class Solution {
    public List<List<String>> findLadders(String beginWord, String endWord, Set<String> wordList) {
        List<List<String>> res = new ArrayList<List<String>>();
        LinkedList<WordNode> queue = new LinkedList<>();
        queue.add(new WordNode(beginWord, 1, null));
        wordList.add(endWord);

        Set<String> visited = new HashSet<>();
        Set<String> unvisited = new HashSet<>();
        unvisited.addAll(wordList);

        int preNumStep = 0;
        int minStep = 0;

        while(!queue.isEmpty()){
            WordNode top = queue.remove();
            String word = top.word;
            int curNumStep = top.numStep;
            if(word.equals(endWord)){
                if(minStep == 0){
                    minStep = top.numStep;
                }

                if(top.numStep == minStep && minStep != 0){
                    //nothing
                    ArrayList<String> t = new ArrayList<String>();
                    t.add(top.word);
                    while(top.pre != null){
                        t.add(0, top.pre.word);
                        top = top.pre;
                    }
                    res.add(t);
                    continue;
                }
            }
        }
    }
}

```



```

        if(preNumStep < curNumStep){
            unvisited.removeAll(visited);
        }

        preNumStep = curNumStep;

        char[] arr = word.toCharArray();
        for(int i=0;i<arr.length;i++){
            for(char c='a';c<='z';c++){
                char tmp = arr[i];
                if(arr[i]!=c) arr[i]=c;
                String newWord = new String(arr);
                if(unvisited.contains(newWord)){
                    queue.add(new WordNode(newWord, top.numStep+1, top));
                    visited.add(newWord);
                }
                arr[i] = tmp;
            }
        }
    }
    return res;
}
}

```

1.13 Shortest Palindrome (hard->Leetcode No.214)

Given a string S, you are allowed to convert it to a palindrome by adding characters in front of it. Find and return the shortest palindrome you can find by performing this transformation.

For example:

Given "aacecaaa", return "aaacecaaa".

Given "abcd", return "dcbabcd".

```

public class Solution {
    public String shortestPalindrome(String s) {
        int i=0;
        int j=s.length()-1;
        while(j>=0){
            if(s.charAt(i)==s.charAt(j))i++;
            j--;
        }
        if(i==s.length()) return s;
        String suffix = s.substring(i);
        String prefix = new StringBuilder(suffix).reverse().toString();
        String mid = shortestPalindrome(s.substring(0,i));
        return prefix+mid+suffix;
    }
}

```

1.14 Integer to English Words (hard->Leetcode No.273)

Convert a non-negative integer to its english words representation. Given input is guaranteed to be less than 2^{31} -

1.

For example,

```
123 -> "One Hundred Twenty Three"
12345 -> "Twelve Thousand Three Hundred Forty Five"
1234567 -> "One Million Two Hundred Thirty Four Thousand Five Hundred Sixty Seven"
```

```
public class Solution {
    private String[]
smalls={"Zero","One","Two","Three","Four","Five","Six","Seven","Eight","Nine","Ten","Eleven","Twelve",
,"Thirteen","Fourteen","Fifteen","Sixteen","Seventeen","Eighteen","Nineteen"};
    private String[] tens = {"","Twenty","Thirty","Forty","Fifty","Sixty","Seventy","Eighty","Ninety"};
    private String hundred = "Hundred";
    private String[] bigs = {"","Thousand","Million","Billion"};
    private String negative = "Negative";
    public String numberToWords(int num) {
        if(num==0) return smalls[0];
        else if(num<0) return negative+" "+numberToWords(-1*num);

        LinkedList<String> parts = new LinkedList<>();
        int chunkCount = 0;
        while(num>0){
            if(num%1000!=0){
                String chunk = convertChunk(num%1000)+" "+bigs[chunkCount];
                parts.addFirst(chunk);
            }
            num/=1000;
            chunkCount++;
        }
        return listToString(parts);
    }

    private String convertChunk(int num){
        LinkedList<String> parts = new LinkedList<>();
        if(num>=100){
            parts.addLast(smalls[num/100]);
            parts.addLast(hundred);
            num%=100;
        }
        if(num>=10 && num<=19) parts.addLast(smalls[num]);
        else if(num>=20) {
            parts.addLast(tens[num/10]);
            num%=10;
        }
        if(1<=num && num<=9) parts.addLast(smalls[num]);
    }
}
```

```

        return listToString(parts);
    }

    private String listToString(LinkedList<String> parts){
        StringBuilder sb = new StringBuilder();
        while(parts.size()>1){
            sb.append(parts.pop());
            sb.append(" ");
        }
        sb.append(parts.pop());
        return sb.toString().trim();
    }
}

```

1.15 Palindrome Pairs (hard->Leetcode No.336)

Given a list of unique words. Find all pairs of **distinct** indices (i, j) in the given list, so that the concatenation of the two words, i.e. `words[i] + words[j]` is a palindrome.

Example 1:

Given `words = ["bat", "tab", "cat"]`

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

The palindromes are `["battab", "tabbat"]`

Example 2:

Given `words = ["abcd", "dcba", "lls", "s", "sssll"]`

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

The palindromes are `["dcbaabcd", "abcddcba", "slls", "llssssll"]`

```

public class Solution {
    public List<List<Integer>> palindromePairs(String[] words) {
        List<List<Integer>> res = new ArrayList<List<Integer>>();
        if(words==null || words.length==0) return res;
        Map<String, Integer> map = new HashMap<>();
        for(int i=0;i<words.length;i++) map.put(words[i], i);
        for(int i=0;i<words.length;i++){
            String word = words[i];
            // if the word is the panlindrome, we need to check whether "" is in map
            if(isPanlindrome(word)){
                if(map.containsKey("")){
                    if(map.get("")!=i){
                        List<Integer> tmp = new ArrayList<>();
                        tmp.add(i);
                        tmp.add(map.get(""));
                        res.add(tmp);
                    }
                }
            }
        }
        return res;
    }
}

```

```

        tmp.add(i);
        res.add(tmp);
    }
}

// if the reverse is exist
String reverse = new StringBuilder(word).reverse().toString();
if(map.containsKey(reverse)){
    if(map.get(reverse)!=i){
        List<Integer> tmp = new ArrayList<>();
        tmp.add(i);
        tmp.add(map.get(reverse));
        res.add(tmp);
    }
}

//
for(int k=1;k<word.length();k++){
    String left = word.substring(0,k);
    String right = word.substring(k);
    if(isPanlindrome(left)){
        String reverseRight = new StringBuilder(right).reverse().toString();
        if(map.containsKey(reverseRight)){
            if(map.get(reverseRight)!=i){
                List<Integer> tmp = new ArrayList<>();
                tmp.add(map.get(reverseRight));
                tmp.add(i);
                res.add(tmp);
            }
        }
    }
    if(isPanlindrome(right)){
        String reverseLeft = new StringBuilder(left).reverse().toString();
        if(map.containsKey(reverseLeft)){
            if(map.get(reverseLeft)!=i){
                List<Integer> tmp = new ArrayList<>();
                tmp.add(i);
                tmp.add(map.get(reverseLeft));
                res.add(tmp);
            }
        }
    }
}
return res;
}
private boolean isPanlindrome(String s){

```

```

int left = 0;
int right = s.length()-1;
while(left<right){
    if(s.charAt(left)!=s.charAt(right)) return false;
    left++;
    right--;
}
return true;
}
}

```

1.16 Median of Two Sorted Arrays (hard->Leetcode No.4)

There are two sorted arrays **nums1** and **nums2** of size m and n respectively.

Find the median of the two sorted arrays. The overall run time complexity should be $O(\log(m+n))$.

Example 1:

```

nums1 = [1, 3]
nums2 = [2]

The median is 2.0

```

Example 2:

```

nums1 = [1, 2]
nums2 = [3, 4]

The median is (2 + 3)/2 = 2.5

```

```

public class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        int m = nums1.length;
        int n = nums2.length;
        if((m+n)%2==0){
            double num1 = (double)findMedian(nums1, 0, m,nums2,0,n,(m+n)/2);
            double num2 = (double)findMedian(nums1, 0, m,nums2,0,n,(m+n)/2+1);
            return (num1+num2)/2;
        }else return findMedian(nums1, 0, m,nums2,0,n,(m+n+1)/2);
    }

    public double findMedian(int[] nums1, int start1, int end1, int[] nums2, int start2, int end2, int k){
        int m = end1-start1;
        int n = end2-start2;
        if(m<=0) return nums2[start2+k-1];
    }
}

```

```

if(n<=0) return nums1[start1+k-1];
if(k==1) return nums1[start1]<nums2[start2]?nums1[start1]:nums2[start2];
int mid1 = start1+(end1-start1)/2;
int mid2 = start2+(end2-start2)/2;
if(nums1[mid1]<=nums2[mid2]){
    if(m/2+n/2+1>=k) return findMedian(nums1, start1, end1, nums2, start2, mid2,k);
    else return findMedian(nums1, mid1+1, end1, nums2, start2, end2,k-m/2-1);
}else{
    if (n / 2 + m / 2 + 1 >= k)
        return findMedian(nums1, start1, mid1, nums2, start2, end2, k);
    else
        return findMedian(nums1, start1, end1, nums2, mid2 + 1, end2, k-n/2-1);
    }
}
}
}

```

1.17 Reverse Nodes in K-Group (Hard->Leetcode No.25)

Given a linked list, reverse the nodes of a linked list k at a time and return its modified list.

If the number of nodes is not a multiple of k then left-out nodes in the end should remain as it is.

You may not alter the values in the nodes, only nodes itself may be changed.

Only constant memory is allowed.

For example,

Given this linked list: 1->2->3->4->5

For $k = 2$, you should return: 2->1->4->3->5

For $k = 3$, you should return: 3->2->1->4->5

```
/**
```

```
* Definition for singly-linked list.
```

```
* public class ListNode {
```

```
*     int val;
```

```
*     ListNode next;
```

```
*     ListNode(int x) { val = x; }
```

```
* }
```

```
*/
```

```
public class Solution {
```

```

public ListNode reverseKGroup(ListNode head, int k) {
    if(head==null || k==1) return head;
    ListNode dummy = new ListNode(0);
    dummy.next = head;
    ListNode pre = dummy;
    ListNode p = head;
    int i=0;
    while(p!=null){
        i++;
        if(i%k==0){
            pre = reverse(pre, p.next);
            p = pre.next;
        }else p = p.next;
    }
    return dummy.next;
}

private ListNode reverse(ListNode pre, ListNode next){
    ListNode last = pre.next;
    ListNode cur = last.next;
    while(cur!=next){
        last.next = cur.next;
        cur.next = pre.next;
        pre.next = cur;
        cur = last.next;
    }
    return last;
}
}

```

1.18 Search in Rotated Sorted Array (hard -> leetcode No.33)

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., `0 1 2 4 5 6 7` might become `4 5 6 7 0 1 2`).

You are given a target value to search. If found in the array return its index, otherwise return -1.

You may assume no duplicate exists in the array.

```

public class Solution {
    public int search(int[] nums, int target) {
        if(nums==null || nums.length==0) return -1;
        return binarySearch(nums, 0, nums.length-1,target);
    }

    private int binarySearch(int[] nums, int left, int right, int target){
        if(left>right) return -1;
        int mid = left+(right-left)/2;
        if(nums[mid]==target) return mid;
    }
}

```

```

    if(nums[left]<=nums[mid]){
        if(nums[left]<=target && target<nums[mid])
            return binarySearch(nums, left, mid-1, target);
        else return binarySearch(nums, mid+1, right, target);
    }else{
        if(nums[mid]<target && target<=nums[right])
            return binarySearch(nums, mid+1, right, target);
        else return binarySearch(nums, left, mid-1,target);
    }
}
}

```

1.19 Sudoku Solver (hard->leetcode No.37)

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

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

You may assume that there will be only one unique solution.

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

A sudoku puzzle...

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

...and its solution numbers marked in red.

```

public class Solution {
    public void solveSudoku(char[][] board) {
        solve(board);
    }
    private boolean solve(char[][] board){
        for(int i=0;i<9;i++){
            for(int j=0;j<9;j++){
                if(board[i][j]!='.') continue;
                for(int k=1;k<=9;k++){
                    board[i][j]=(char)(k+'0');
                    if(isValid(board, i, j) && solve(board)) return true;
                    board[i][j]='.';
                }
                return false;
            }
        }
        return true;
    }

    private boolean isValid(char[][] board, int i, int j){
        Set<Character> set = new HashSet<>();

```



```

// (i,j) the whole row
for(int k=0;k<9;k++){
    if(set.contains(board[i][k])) return false;
    if(board[i][k]!='.') set.add(board[i][k]);
}
set.clear();
// (i,j) the whole column
for(int k=0;k<9;k++){
    if(set.contains(board[k][j])) return false;
    if(board[k][j]!='.') set.add(board[k][j]);
}
set.clear();
// (i,j) the 9-cell block
for(int m=0;m<3;m++){
    for(int n=0;n<3;n++){
        int x = i/3*3+m;
        int y = j/3*3+n;
        if(set.contains(board[x][y])) return false;
        if(board[x][y]!='.') set.add(board[x][y]);
    }
}
return true;
}
}

```

1.20 First Missing positive (hard->leetcode No.41)

Given an unsorted integer array, find the first missing positive integer.

For example,

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

and `[3,4,-1,1]` return `2`.

Your algorithm should run in $O(n)$ time and uses constant space.

```

public class Solution {
    public int firstMissingPositive(int[] nums) {
        int n = nums.length;
        for(int i=0;i<n;i++){
            while(nums[i]!=i+1){
                if(nums[i]<=0 || nums[i]>=n) break;
                if(nums[i]==nums[nums[i]-1]) break;
                int tmp = nums[i];
                nums[i] = nums[tmp-1];
                nums[tmp-1] = tmp;
            }
        }
    }
}

```

```

    for(int i=0;i<n;i++){
        if(nums[i]!=i+1) return i+1;
    }
    return n+1;
}
}

```

1.21 Trapping Rain Water (hard->leetcode No.42)

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

For example,

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



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

Marcos for contributing this image!

```

public class Solution {
    public int trap(int[] height) {
        if(height==null || height.length==0) return 0;

        int[] left = new int[height.length];
        int[] right = new int[height.length];

        left[0] = height[0];
        int max = height[0];
        for(int i =1;i<height.length;i++){
            if(max>height[i]) left[i] = max;
            else{
                left[i] = height[i];
                max = height[i];
            }
        }

        right[height.length-1] = height[height.length-1];
        max = height[height.length-1];
        for(int i=height.length-2;i>=0;i--){
            if(max>height[i]) right[i] = max;
            else {
                max =height[i];
            }
        }
    }
}

```

```

        right[i] = height[i];
    }
}

int res = 0;
for(int i=0;i<height.length;i++){
    res+=Math.min(left[i],right[i])-height[i];
}
return res;
}
}

```

1.22 Jump Game II (hard->Leetcode No.45)

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Your goal is to reach the last index in the minimum number of jumps.

For example:

Given array A = [2,3,1,1,4]

The minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.)

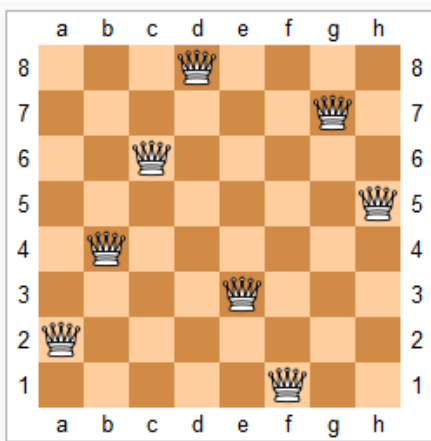
```

public class Solution {
    public int jump(int[] nums) {
        if(nums==null || nums.length==0) return 0;
        int reach = 0;
        int lastReach = 0;
        int step = 0;
        for(int i=0;i<=reach && i<nums.length;i++){
            if(i>lastReach){
                step++;
                lastReach = reach;
            }
            reach = Math.max(reach, i+nums[i]);
        }
        if(reach<nums.length-1) return 0;
        return step;
    }
}

```

1.23 N-Queens (hard->leetcode No.51)

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



One solution to the eight queens puzzle

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

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

Send F

For example,

There exist two distinct solutions to the 4-queens puzzle:

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

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

```
public class Solution {
    public List<List<String>> solveNQueens(int n) {
        List<List<String>> res = new ArrayList<List<String>>();
        if(n<=0) return res;
        search(n, res, new ArrayList<Integer>());
        return res;
    }

    private void search(int n, List<List<String>> res, ArrayList<Integer> cols){
```

```

        if(cols.size()==n){
            res.add(drawChessBoard(cols));
            return;
        }
        for(int col = 0;col<n;col++){
            if(!isValid(cols, col)) continue;
            cols.add(col);
            search(n, res, cols);
            cols.remove(cols.size()-1);
        }
    }

    private boolean isValid(List<Integer> cols, int col){
        int row = cols.size();
        for(int i=0;i<row;i++){
            if(cols.get(i)==col) return false;
            if(i-cols.get(i)==row-col) return false;
            if(i+cols.get(i)==row+col) return false;
        }
        return true;
    }

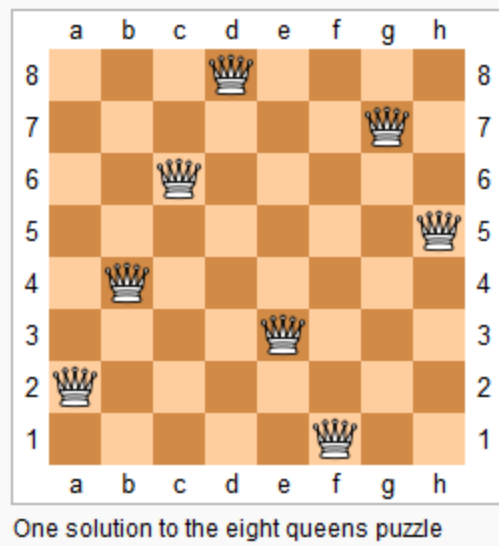
    private List<String> drawChessBoard(ArrayList<Integer> cols){
        List<String> chessBoard = new ArrayList<>();
        for(int i=0;i<cols.size();i++){
            String tmp = "";
            for(int j = 0;j<cols.size();j++){
                if(j==cols.get(i)) tmp+="Q";
                else tmp+=".";
            }
            chessBoard.add(tmp);
        }
        return chessBoard;
    }
}

```

1.24 N-Queens II(hard->leetcode No.52)

Follow up for N-Queens problem.

Now, instead outputting board configurations, return the total number of distinct solutions.



[Subscribe](#) to see which companies asked this question

```
public class Solution {
    private static int sum;
    public int totalNQueens(int n) {
        sum = 0;
        if(n<=0) return sum;
        int[] usedCols = new int[n];
        search(usedCols, 0);
        return sum;
    }

    private void search(int[] usedCols, int row){
        int n = usedCols.length;
        if(row==n){
            sum++;
            return;
        }

        for(int col=0;col<n;col++){
            if(isValid(usedCols, row, col)){
                usedCols[row] = col;
                search(usedCols, row+1);
            }
        }
    }

    private boolean isValid(int[] usedCols, int row, int col){
```

```

        for(int i=0;i<row;i++){
            if(usedCols[i]==col) return false;
            if(row-i==Math.abs(col-usedCols[i])) return false;
        }
        return true;
    }
}

```

1.25 Merge Intervals (hard->Leetcode No.56)

Given a collection of intervals, merge all overlapping intervals.

For example,

Given `[1,3],[2,6],[8,10],[15,18]`,

return `[1,6],[8,10],[15,18]`.

```

/**
 * 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 List<Interval> merge(List<Interval> intervals) {
        List<Interval> res = new ArrayList<>();
        if(intervals==null || intervals.size()==0) return res;
        Collections.sort(intervals, new Comparator<Interval>(){
            public int compare(Interval l1, Interval l2){
                if(l1.start!=l2.start) return l1.start-l2.start;
                else return l1.end-l2.end;
            }
        });
        Interval pre = intervals.get(0);
        for(int i=0;i<intervals.size();i++){
            Interval cur = intervals.get(i);
            if(cur.start>pre.end){
                res.add(pre);
                pre = cur;
            }else{
                Interval merge = new Interval(pre.start, Math.max(pre.end, cur.end));
                pre = merge;
            }
        }
        res.add(pre);
        return res;
    }
}

```

```
}
```

1.26 Insert Interval (hard->leetcode No.57)

Given a set of *non-overlapping* intervals, insert a new interval into the intervals (merge if necessary).

You may assume that the intervals were initially sorted according to their start times.

Example 1:

Given intervals `[1,3],[6,9]` , insert and merge `[2,5]` in as `[1,5],[6,9]` .

Example 2:

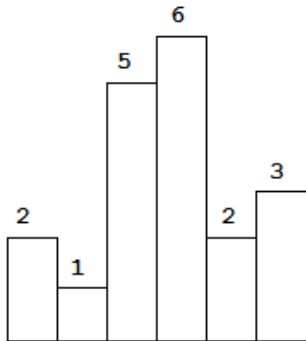
Given `[1,2],[3,5],[6,7],[8,10],[12,16]` , insert and merge `[4,9]` in as `[1,2],[3,10],[12,16]` .

This is because the new interval `[4,9]` overlaps with `[3,5],[6,7],[8,10]` .

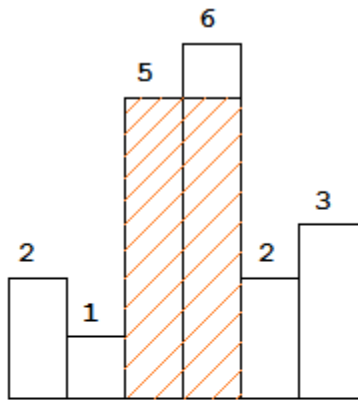
```
/**
 * Definition for an interval.
 * public class Interval {
 *     int start;
 *     int end;
 *     Interval() { start = 0; end = 0; }
 *     Interval(int s, int e) { start = s; end = e; }
 * }
 */
public class Solution {
    public List<Interval> insert(List<Interval> intervals, Interval newInterval) {
        List<Interval> res = new ArrayList<>();
        for(Interval interval:intervals){
            if(interval.end<newInterval.start) res.add(interval);
            else if(interval.start>newInterval.end){
                res.add(newInterval);
                newInterval = interval;
            }else if(interval.start<=newInterval.end || interval.end<=newInterval.end){
                newInterval = new Interval(Math.min(interval.start, newInterval.start),Math.max(interval.end,
newInterval.end));
            }
        }
        res.add(newInterval);
        return res;
    }
}
```

1.27 Largest Rectangle in Histogram (hard->leetcode No.84)

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



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



The largest rectangle is shown in the shaded area, which has area = `10` unit.

For example,

Given heights = `[2,1,5,6,2,3]`,

return `10`.

```
public class Solution {
    public int largestRectangleArea(int[] heights) {
        int max = 0;
        if(heights==null || heights.length==0) return 0;
        Stack<Integer> stack = new Stack<>();
        int i = 0;
        while(i<heights.length){
            if(stack.isEmpty() || heights[i]>=heights[stack.peek()]){
                stack.push(i);
                i++;
            }else{
                int index = stack.pop();
                int height = heights[index];
```

```

        int width = stack.isEmpty()?i:i-stack.peek()-1;
        max = Math.max(max, height*width);
    }
}
while(!stack.isEmpty()){
    int index = stack.pop();
    int height = heights[index];
    int width = stack.isEmpty()?i:i-stack.peek()-1;
    max = Math.max(max, height*width);
}
return max;
}
}

```

1.28 Maximal Rectangle (hard->leetcode NO.85)

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

For example, given the following matrix:

```

1 0 1 0 0
1 0 1 1 1
1 1 1 1 1
1 0 0 1 0

```

Return 6.

```

public class Solution {
    public int maximalRectangle(char[][] matrix) {
        if(matrix==null || matrix.length==0) return 0;
        int m = matrix.length;
        int n = m==0?0:matrix[0].length;
        int[][] height = new int[m][n+1];

        for(int i=0;i<m;i++){
            for(int j=0;j<n;j++){
                if(matrix[i][j]=='0') height[i][j]=0;
                else height[i][j]=i==0?1:height[i-1][j]+1;
            }
        }
        int max = 0;
        for(int i=0;i<m;i++){
            int area = maxAreaInhist(height[i]);
            if(area>max) max = area;
        }
        return max;
    }
    private int maxAreaInhist(int[] height){
        int max = 0;
        Stack<Integer> stack = new Stack<>();
        int i=0;

```

```

while(i<height.length){
    if(stack.isEmpty() || height[stack.peek()]<=height[i]) stack.push(i++);
    else{
        int index = stack.pop();
        int h = height[index];
        int width = stack.isEmpty()?i-i-stack.peek()-1;
        max = Math.max(max, h*width);
    }
}
return max;
}
}

```

1.29 Recover Binary Search Tree (hard->leetcode No.99)

Two elements of a binary search tree (BST) are swapped by mistake.

Recover the tree without changing its structure.

Note:

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

[Subscribe](#) to see which companies asked this question

```

/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    TreeNode pre;
    TreeNode first;
    TreeNode second;
    public void recoverTree(TreeNode root) {
        if(root==null) return;
        inorder(root);
        int tmp = first.val;
        first.val = second.val;
        second.val = tmp;
    }

    private void inorder(TreeNode root){
        if(root==null) return;
        inorder(root.left);
        if(pre==null) pre = root;
    }
}

```

```

    else{
        if(root.val<pre.val){
            if(first==null) first = pre;
            second = root;
        }
        pre = root;
    }
    inorder(root.right);
}
}

```

1.30 Populating Next Right Pointers in Each Node II (hard->leetcode No.117)

Follow up for problem "Populating Next Right Pointers in Each Node".

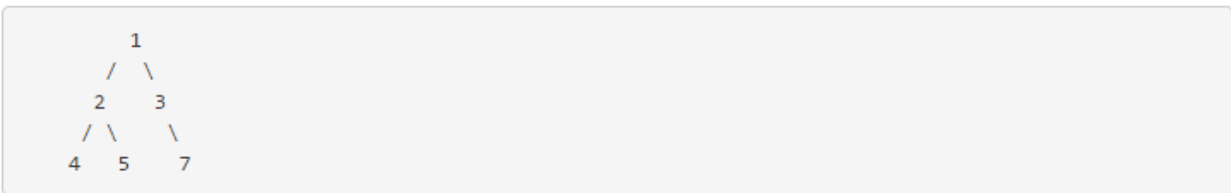
What if the given tree could be any binary tree? Would your previous solution still work?

Note:

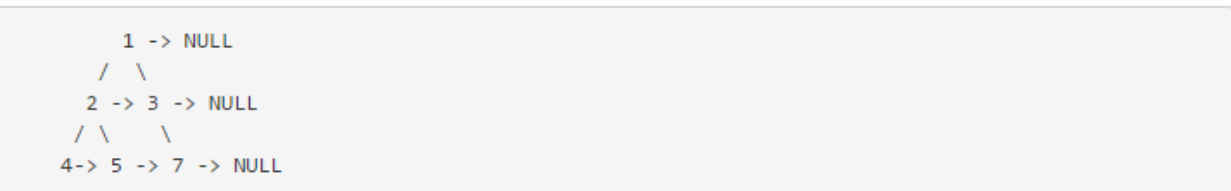
- You may only use constant extra space.

For example,

Given the following binary tree,



After calling your function, the tree should look like:



```

/**
 * Definition for binary tree with next pointer.
 * public class TreeLinkNode {
 *     int val;
 *     TreeLinkNode left, right, next;
 *     TreeLinkNode(int x) { val = x; }
 * }
 */
public class Solution {
    public void connect(TreeLinkNode root) {
        if(root==null) return;
        TreeLinkNode p = root.next;

```

```

while(p!=null){
    if(p.left!=null){
        p = p.left;
        break;
    }
    if(p.right!=null){
        p = p.right;
        break;
    }
    p = p.next;
}

if(root.right!=null)root.right.next = p;
if(root.left!=null){
    if(root.right!=null) root.left.next = root.right;
    else root.left.next = p;
}
connect(root.right);
connect(root.left);
}
}

```

1.31 Best Time to Buy and Sell Stock III (Hard->leetcode No.123)

Say you have an array for which the i^{th} element is the price of a given stock on day i .

Design an algorithm to find the maximum profit. You may complete at most two transactions.

Note:

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

Comparing to I and II, III limits the number of transactions to 2. This can be solve by "devide and conquer". We use left[i] to track the maximum profit for transactions before i, and use right[i] to track the maximum profit for transactions after i. You can use the following example to understand the Java solution:

```

Prices: 1 4 5 7 6 3 2 9
left = [0, 3, 4, 6, 6, 6, 6, 8]
right= [8, 7, 7, 7, 7, 7, 7, 0]

```

```

public class Solution {
    public int maxProfit(int[] prices) {
        if(prices==null || prices.length==0) return 0;
        int min = prices[0];
        int[] left = new int[prices.length];
        int[] right = new int[prices.length];
        for(int i=1;i<prices.length;i++){

```

```

        min = Math.min(min, prices[i]);
        left[i] = Math.max(left[i-1], prices[i]-min);
    }

    right[prices.length-1] = 0;
    int max = prices[prices.length-1];
    for(int i=prices.length-2;i>=0;i--){
        max = Math.max(max, prices[i]);
        right[i]=Math.max(right[i+1],max-prices[i]);
    }

    int profit = 0;
    for(int i=0;i<prices.length;i++){
        profit = Math.max(profit, left[i]+right[i]);
    }
    return profit;
}
}

```

1.32 Binary Tree Maximum Path Sum (Hard->leetcode No.124)

Given a binary tree, find the maximum path sum.

For this problem, a path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The path does not need to go through the root.

For example:

Given the below binary tree,



Return 6 .

```

/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    //private static int sum = 0;
    public int maxPathSum(TreeNode root) {
        if(root==null) return 0;
        int[] max = new int[1];
        max[0] = Integer.MIN_VALUE;
    }
}

```

```

        searchMax(root,max);
        return max[0];
    }
    private int searchMax(TreeNode root, int[] max){
        if(root==null) return 0;
        int left = searchMax(root.left,max);
        int right = searchMax(root.right,max);
        int curr = Math.max(root.val, Math.max(root.val+left, root.val+right));
        max[0] = Math.max(max[0], Math.max(curr, root.val+left+right));
        return curr;
    }
}

```

1.33 Longest Consecutive Sequence (hard->leetcode No.128)

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

For example,

Given `[100, 4, 200, 1, 3, 2]`,

The longest consecutive elements sequence is `[1, 2, 3, 4]`. Return its length: `4`.

Your algorithm should run in $O(n)$ complexity.

```

public class Solution {
    public int longestConsecutive(int[] nums) {
        if(nums==null || nums.length==0) return 0;
        Set<Integer> set = new HashSet<>();
        int max = 1;
        for(int num:nums) set.add(num);
        for(int num:nums){
            int left = num-1;
            int right = num+1;
            int count = 1;
            while(set.contains(left)){
                count++;
                set.remove(left);
                left--;
            }
            while(set.contains(right)){
                count++;
                set.remove(right);
                right++;
            }
            max = Math.max(max, count);
        }
        return max;
    }
}

```

1.34 Palindrome Partitioning II (hard-> leetcode No.132)

Given a string *s*, partition *s* such that every substring of the partition is a palindrome.

Return the minimum cuts needed for a palindrome partitioning of *s*.

For example, given *s* = "aab",

Return 1 since the palindrome partitioning ["aa", "b"] could be produced using 1 cut.

Analysis

This problem is similar to [Palindrome Partitioning](#). It can be efficiently solved by using dynamic programming. Unlike "Palindrome Partitioning", we need to maintain two cache arrays, one tracks the partition position and one tracks the number of minimum cut.

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

1.35 Candy (hard->leetcode No.135)

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

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

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

What is the minimum candies you must give?

This problem can be solved in $O(n)$ time.

We can always assign a neighbor with 1 more if the neighbor has higher a rating value. However, to get the minimum total number, we should always start adding 1s in the ascending order. We can solve this problem by scanning the array from both sides. First, scan the array from left to right, and assign values for all the ascending pairs. Then scan from right to left and assign values to descending pairs.

```
public class Solution {
    public int candy(int[] ratings) {
        if(ratings==null || ratings.length==0) return 0;
        int n = ratings.length;
        int[] candies = new int[n];
        candies[0] = 1;
        for(int i=1;i<n;i++){
            if(ratings[i]>ratings[i-1]) candies[i] = candies[i-1]+1;
            else candies[i] = 1;
        }

        int result = candies[n-1];
        for(int i=n-2;i>=0;i--){
            int cur = 1;
            if(ratings[i]>ratings[i+1]){
                cur = Math.max(cur, candies[i+1]+1);
            }
            result += Math.max(cur, candies[i]);
            candies[i] = Math.max(cur, candies[i]);
        }
        return result;
    }
}
```

1.36 Copy List with Random Pointer (hard->leetcode No.138)

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

Return a deep copy of the list.

```
/**
 * Definition for singly-linked list with a random pointer.
 * class RandomListNode {
 *   int label;
 *   RandomListNode next, random;
 *   RandomListNode(int x) { this.label = x; }
 * };
 */
public class Solution {
    public RandomListNode copyRandomList(RandomListNode head) {
        RandomListNode p = head;
        while(p!=null){
```

```

        RandomListNode next = p.next;
        RandomListNode copy = new RandomListNode(p.label);
        p.next = copy;
        copy.next = next;
        p = next;
    }
    // the second step, make copy's random to point to original's random
    p = head;
    while(p!=null){
        p.next.random = p.random!=null?p.random.next:null;
        p = p.next.next;
    }
    // the 3rd step, restore the original
    p = head;
    RandomListNode headcopy = p!=null?p.next:null;
    while(p!=null){
        RandomListNode copy = p.next;
        p.next = copy.next;
        p = p.next;
        copy.next = p!=null? p.next:null;
    }
    return headcopy;
}
}

```

1.37 Word Break II (hard->leetcode No.140)

Given a string *s* and a dictionary of words *dict*, add spaces in *s* to construct a sentence where each word is a valid dictionary word.

Return all such possible sentences.

For example, given

s = "catsanddog",

dict = ["cat", "cats", "and", "sand", "dog"] .

A solution is ["cats and dog", "cat sand dog"] .

```

public class Solution {
    public List<String> wordBreak(String s, Set<String> wordDict) {
        ArrayList<String>[] pos = new ArrayList[s.length()+1];
        pos[0] = new ArrayList<String>();
        for(int i=0;i<s.length();i++){
            if(pos[i]!=null){
                for(int j=i+1;j<=s.length();j++){
                    String sub = s.substring(i,j);
                    if(wordDict.contains(sub)){
                        if(pos[j]!=null) pos[j].add(sub);
                        else{
                            ArrayList<String> tmp = new ArrayList<>();

```

```

        tmp.add(sub);
        pos[j]=tmp;
    }
}
}
}

if(pos[s.length()]==null) return new ArrayList<String>();
else{
    List<String> result = new ArrayList<>();
    dfs(pos, result, "", s.length());
    return result;
}
}

private void dfs(ArrayList<String>[] pos, List<String> result, String cur, int index){
    if(index==0){
        result.add(cur.trim());
        return;
    }
    for(String sub:pos[index]){
        String combined = sub+" "+cur;
        dfs(pos, result, combined, index-sub.length());
    }
}
}
}

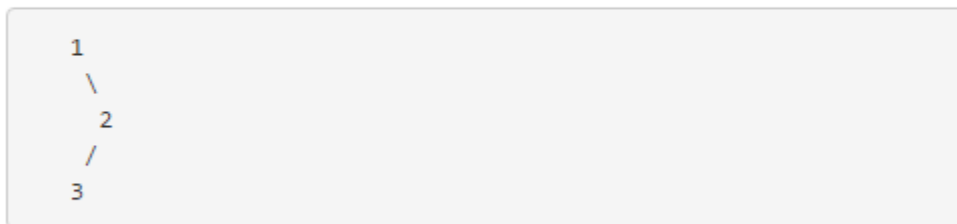
```

1.38 Binary Tree Postorder Traversal (hard->leetcode No.145)

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

For example:

Given binary tree {1,#,2,3} ,



return [3,2,1] .

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

/**

* Definition for a binary tree node.

```

* public class TreeNode {
*   int val;
*   TreeNode left;
*   TreeNode right;
*   TreeNode(int x) { val = x; }
* }
*/
public class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        if(root==null) return res;
        Stack<TreeNode> stack = new Stack<>();
        stack.push(root);
        while(!stack.isEmpty()){
            TreeNode node = stack.peek();
            if(node.left==null && node.right==null) res.add(stack.pop().val);
            else{
                if(node.right!=null){
                    stack.push(node.right);
                    node.right=null;
                }
                if(node.left!=null){
                    stack.push(node.left);
                    node.left=null;
                }
            }
        }
        return res;
    }
}

```

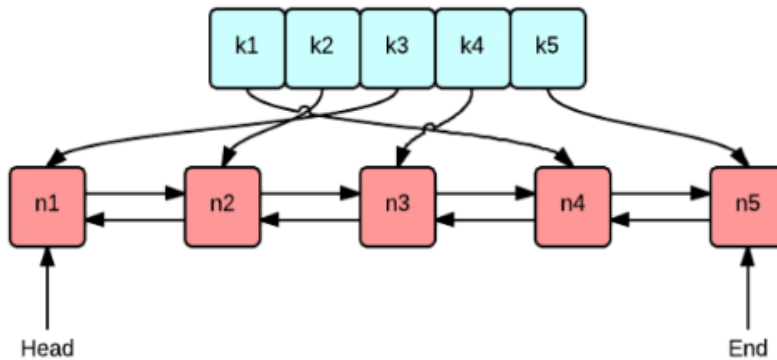
1.39 LRU Cache (hard->leetcode No.146)

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: `get` and `set`.

`get(key)` - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.

`set(key, value)` - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

The key to solve this problem is using a double linked list which enables us to quickly move nodes.



The LRU cache is a hash table of keys and double linked nodes. The hash table makes the time of `get()` to be $O(1)$. The list of double linked nodes make the nodes adding/removal operations $O(1)$.

```

class Node{
    Node pre;
    Node next;
    int key;
    int value;
    public Node(int key, int value){
        this.key = key;
        this.value = value;
    }
}

```

```

public class LRUCache {
    int capacity;
    Map<Integer, Node> map;
    Node head;
    Node end;
    public LRUCache(int capacity) {
        map = new HashMap<>();
        this.capacity = capacity;
    }
}

```

```

public int get(int key) {
    if(map.containsKey(key)){
        Node node = map.get(key);
        remove(node);
        setHead(node);
        return node.value;
    }
}

```

```

        return -1;
    }

    public void set(int key, int value) {
        if(map.containsKey(key)){
            Node node = map.get(key);
            node.value = value;
            remove(node);
            setHead(node);
        }else{
            Node node = new Node(key, value);
            if(map.size()>=capacity){
                map.remove(end.key);
                remove(end);
                setHead(node);
            }else setHead(node);
            map.put(key, node);
        }
    }

    private void setHead(Node node){
        node.next = head;
        node.pre = null;
        if(head!=null) head.pre = node;
        head = node;
        if(end==null) end = head;
    }

    private void remove(Node node){
        if(node.pre!=null) node.pre.next = node.next;
        else head = node.next;
        if(node.next!=null) node.next.pre = node.pre;
        else end = node.pre;
    }
}

```

1.40 Max Points on a Line(hard->leetcode No.149)

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

```

/**
 * Definition for a point.
 * class Point {
 *     int x;
 *     int y;
 *     Point() { x = 0; y = 0; }
 *     Point(int a, int b) { x = a; y = b; }
 * }
 */

```

```

public class Solution {
    public int maxPoints(Point[] points) {
        if(points==null || points.length==0) return 0;
        int max = 0;
        Map<Double, Integer> map = new HashMap<>();
        for(int i=0;i<points.length;i++){
            int duplicates=1;
            int verticals = 0;
            for(int j=i+1;j<points.length;j++){
                if(points[i].x==points[j].x){
                    if(points[i].y==points[j].y) duplicates++;
                    else verticals++;
                }else{
                    double slope = points[i].y==points[j].y?0.0:(1.0*(points[j].y-points[i].y)/(points[j].x-
points[i].x));
                    if(map.containsKey(slope)) map.put(slope, map.get(slope)+1);
                    else map.put(slope, 1);
                }
            }

            for(int result:map.values()){
                if(result+duplicates>max) max = result+duplicates;
            }
            max = Math.max(max, verticals+duplicates);
            map.clear();
        }
        return max;
    }
}

```

1.41 Find Minimum in Rotated Sorted Array II (Hard -> leetcode No.154)

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., `0 1 2 4 5 6 7` might become `4 5 6 7 0 1 2`).

Find the minimum element.

The array may contain duplicates.

```

public class Solution {
    public int findMin(int[] nums) {
        if(nums==null || nums.length==0) return 0;
        return BinaryFind(nums, 0, nums.length-1);
    }

    private int BinaryFind(int[] nums, int left, int right){
        if(left==right) return nums[left];
        if(left+1==right) return Math.min(nums[left],nums[right]);
    }
}

```

```

        int mid = left+(right-left)/2;
        if(nums[left]<nums[right]) return nums[left];
        else if(nums[left]==nums[right]) return BinaryFind(nums, left+1, right);
        else if(nums[left]<=nums[mid]) return BinaryFind(nums, mid, right);
        else return BinaryFind(nums, left, mid);
    }
}

```

1.42 Maximum Gap (hard->leetcode No.164)

Given an unsorted array, find the maximum difference between the successive elements in its sorted form.

Try to solve it in linear time/space.

Return 0 if the array contains less than 2 elements.

You may assume all elements in the array are non-negative integers and fit in the 32-bit signed integer range.

```

class Bucket{
    int low;
    int high;
    public Bucket(int low, int high){
        this.low = low;
        this.high = high;
    }
}

public class Solution {
    public int maximumGap(int[] nums) {
        if(nums==null || nums.length<2) return 0;
        int max = Integer.MIN_VALUE;
        int min = Integer.MAX_VALUE;
        for(int i=0;i<nums.length;i++){
            max = Math.max(max, nums[i]);
            min = Math.min(min, nums[i]);
        }
        Bucket[] buckets = new Bucket[nums.length+1];
        for(int i=0;i<buckets.length;i++) buckets[i] = new Bucket(-1,-1);
        double interval = (double)nums.length/(max-min);
        for(int i=0;i<nums.length;i++){
            int index = (int)(interval*(nums[i]-min));
            if(buckets[index].low==-1){
                buckets[index].low = nums[i];
                buckets[index].high = nums[i];
            }else{
                buckets[index].low = Math.min(buckets[index].low, nums[i]);
                buckets[index].high = Math.max(buckets[index].high, nums[i]);
            }
        }

        int result = 0;
    }
}

```



```

int pre = buckets[0].high;
for(int i=1;i<buckets.length;i++){
    if(buckets[i].low!=-1){
        result = Math.max(result, buckets[i].low-pre);
        pre = buckets[i].high;
    }
}
return result;
}
}

```

1.43 Dungeon Game (hard->leetcode No.174)

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

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

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

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

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

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

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

Notes:

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

✉ Se

```
public class Solution {
    public int calculateMinimumHP(int[][] dungeon) {
        if(dungeon==null || dungeon.length==0) return 0;
        int m = dungeon.length;
        int n = dungeon[0].length;
        int[][] dp = new int[m][n];

        dp[m-1][n-1] = Math.max(1-dungeon[m-1][n-1], 1);
        for(int i=n-2;i>=0;i--) dp[m-1][i] = Math.max(dp[m-1][i+1]-dungeon[m-1][i],1);
        for(int j = m-2;j>=0;j--) dp[j][n-1]=Math.max(dp[j+1][n-1]-dungeon[j][n-1],1);

        for(int i=m-2;i>=0;i--){
            for(int j=n-2;j>=0;j--){
                int right = Math.max(1, dp[i][j+1]-dungeon[i][j]);
                int down = Math.max(1, dp[i+1][j]-dungeon[i][j]);
                dp[i][j] = Math.min(right, down);
            }
        }
        return dp[0][0];
    }
}
```

1.44 Best Time to Buy and Sell Stock IV (Hard->leetcode No.188)

Say you have an array for which the i^{th} element is the price of a given stock on day i .

Design an algorithm to find the maximum profit. You may complete at most k transactions.

This is a generalized version of [Best Time to Buy and Sell Stock III](#). If we can solve this problem, we can also use $k=2$ to solve III.

The problem can be solve by using dynamic programming. The relation is:

```
local[i][j] = max(global[i-1][j-1] + max(diff,0), local[i-1][j]+diff)
global[i][j] = max(local[i][j], global[i-1][j])
```

We track two arrays - local and global. The local array tracks maximum profit of j transactions & the last transaction is on i th day. The global array tracks the maximum profit of j transactions until i th day.

```
public class Solution {
    public int maxProfit(int k, int[] prices) {
        if(prices==null || prices.length==0 || k<=0) return 0;
        int[] local = new int[k+1];
        int[] global = new int[k+1];
        for(int i=0;i<prices.length-1;i++){
            int diff = prices[i+1]-prices[i];
            for(int j = k;j>=1;j--){
                local[j] = Math.max(global[j-1]+Math.max(0,diff), local[j]+diff);
                global[i] = Math.max(global[i],local[i]);
            }
        }
        return global[k];
    }
}
```

1.45 Word Search II (hard->leetcode No.212)

Given a 2D board and a list of words from the dictionary, find all words in the board.

Each word must be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.

For example,

Given **words** = ["oath", "pea", "eat", "rain"] and **board** =

```
[
  ['o','a','a','n'],
  ['e','t','a','e'],
  ['i','h','k','r'],
  ['i','f','l','v']
]
```

Return ["eat", "oath"] .

Note:

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

```
class TrieNode{
    public String item;
    public TrieNode[] children;
    public TrieNode(){
        item="";
        children = new TrieNode[26];
    }
}

class Trie{
    private TrieNode root;
    public Trie(){root = new TrieNode();};
    public void insert(String word){
        TrieNode node = root;
        for(char c:word.toCharArray()){
            if(node.children[c-'a']==null) node.children[c-'a']=new TrieNode();
            node = node.children[c-'a'];
        }
        node.item = word;
    }
    public boolean search(String word){
        TrieNode node = root;
        for(char c:word.toCharArray()){
            if(node.children[c-'a']==null) return false;
            node = node.children[c-'a'];
        }
        if(node.item.equals(word)) return true;
        else return false;
    }
}
```

```

public boolean startWith(String pref){
    TrieNode node = root;
    for(char c:pref.toCharArray()){
        if(node.children[c-'a']==null) return false;
        node = node.children[c-'a'];
    }
    return true;
}

}

public class Solution {
    public Set<String> set = new HashSet<>();
    public List<String> findWords(char[][] board, String[] words) {
        List<String> res = new ArrayList<>();
        if(board==null || board.length==0 || board[0].length==0 || words==null || words.length==0)
            return res;
        int m = board.length;
        int n = board[0].length;

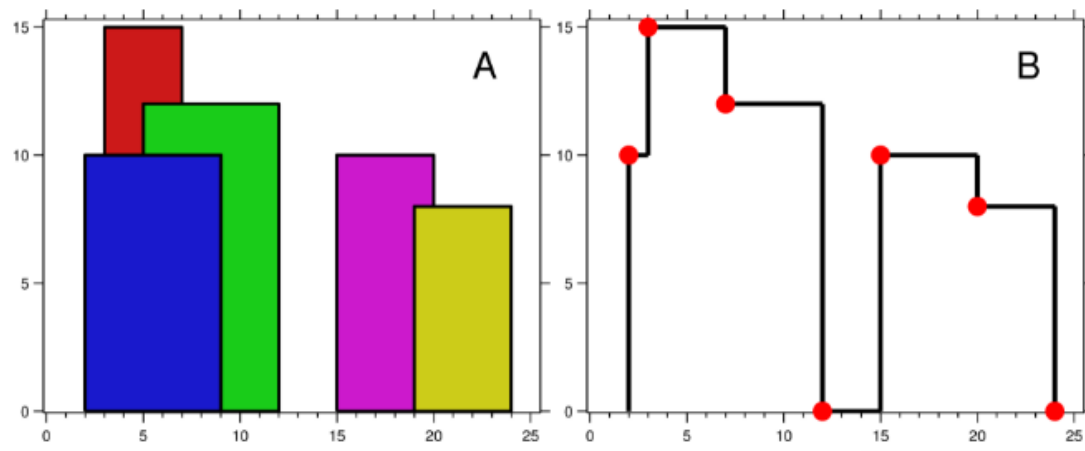
        Trie trie = new Trie();
        for(String word:words) trie.insert(word);
        boolean[][] visited = new boolean[m][n];
        for(int i=0;i<m;i++){
            for(int j=0;j<n;j++){
                dfs(board, "", visited, i, j, trie);
            }
        }
        return new ArrayList<String>(set);
    }

    public void dfs(char[][] board, String word, boolean[][] visited, int i, int j, Trie trie){
        int m = board.length;
        int n = board[0].length;
        if(i<0 || i>=m || j<0 || j>=n) return;
        if(visited[i][j]) return;
        word = word+board[i][j];
        if(!trie.startWith(word)) return;
        if(trie.search(word)) set.add(word);
        visited[i][j] = true;
        dfs(board, word, visited, i+1,j,trie);
        dfs(board, word, visited, i-1,j,trie);
        dfs(board, word, visited, i,j-1,trie);
        dfs(board, word, visited, i,j+1,trie);
        visited[i][j] = false;
    }
}

```

1.46 The skyline Problem (hard->leetcode No.218)

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



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

Send Feedback

For instance, the dimensions of all buildings in Figure A are recorded as: $[[2, 9, 10], [3, 7, 15], [5, 12, 12], [15, 20, 10], [19, 24, 8], [20, 24, 0]]$.

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

For instance, the skyline in Figure B should be represented as: $[[2, 0], [3, 15], [7, 12], [12, 0], [15, 10], [20, 8], [24, 0], [25, 0]]$.

Notes:

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

```

class Edge{
    int x;
    int height;
    boolean isStart;
    public Edge(int x, int height, boolean isStart){
        this.x = x;
        this.height = height;
        this.isStart = isStart;
    }
}

public class Solution {
    public List<int[]> getSkyline(int[][] buildings) {
        List<int[]> res = new ArrayList<>();
        if(buildings==null || buildings.length==0 || buildings[0].length==0) return res;
        List<Edge> edges = new ArrayList<>();
        for(int[] building:buildings){
            edges.add(new Edge(building[0],building[2], true));
            edges.add(new Edge(building[1],building[2],false));
        }
        Collections.sort(edges, new Comparator<Edge>(){
            public int compare(Edge e1, Edge e2){
                if(e1.x!=e2.x) return Integer.compare(e1.x,e2.x);
                if(e1.isStart && e2.isStart) return Integer.compare(e2.height, e1.height);
                if(!e1.isStart && !e2.isStart) return Integer.compare(e1.height, e2.height);
                return e1.isStart?-1:1;
            }
        });

        PriorityQueue<Integer> queue = new PriorityQueue<>(10, Collections.reverseOrder());
        for(Edge edge:edges){
            if(edge.isStart){
                if(queue.isEmpty() || edge.height>queue.peek())
                    res.add(new int[]{edge.x, edge.height});
                queue.add(edge.height);
            }else{
                queue.remove(edge.height);
                if(queue.isEmpty()) res.add(new int[]{edge.x, 0});
                else if(edge.height>queue.peek())
                    res.add(new int[]{edge.x, queue.peek()});
            }
        }
        return res;
    }
}

```

1.47 Basic Calculator (hard->leetcode No.224)

Implement a basic calculator to evaluate a simple expression string.

The expression string may contain open `(` and closing parentheses `)`, the plus `+` or minus sign `-`, **non-negative** integers and empty spaces .

You may assume that the given expression is always valid.

Some examples:

```
"1 + 1" = 2
" 2-1 + 2 " = 3
"(1+(4+5+2)-3)+(6+8)" = 23
```

This problem can be solved by using a stack. We keep pushing element to the stack, when `)` is met, calculate the expression up to the first `"(`.

```
public class Solution {
    public int calculate(String s) {
        if(s==null || s.length()==0) return 0;
        s = s.replaceAll(" ", "");
        Stack<String> stack = new Stack<>();
        StringBuilder sb = new StringBuilder();
        char[] arr = s.toCharArray();
        for(int i=0;i<arr.length;i++){
            if(arr[i]==' ') continue;
            if(arr[i]>='0' && arr[i]<='9'){
                sb.append(arr[i]);
                if(i==arr.length-1) stack.push(sb.toString());
            }else{
                if(sb.length()>0){
                    stack.push(sb.toString());
                    sb = new StringBuilder();
                }
                if(arr[i]!='+') stack.push(new String(new char[]{arr[i]}));
            }else{
                List<String> t = new ArrayList<>();
                while(!stack.isEmpty()){
                    String top = stack.pop();
                    if(top.equals("(")) break;
                    else t.add(0,top);
                }
                int tmp = calculate(t);
                stack.push(String.valueOf(tmp));
            }
        }
    }
}

List<String> t = init(stack);
int tmp = calculate(t);
```



```

        return tmp;
    }

    private List<String> init(Stack<String> stack){
        List<String> t = new ArrayList<>();
        while(!stack.isEmpty()){
            String top = stack.pop();
            t.add(0,top);
        }
        return t;
    }
    private int calculate(List<String> t){
        int tmp = 0;
        if(t.size()==1) tmp = Integer.valueOf(t.get(0));
        else{
            for(int j = t.size()-1;j>0;j=j-2){
                if(t.get(j-1).equals("-")) tmp+=0-Integer.valueOf(t.get(j));
                else tmp+=Integer.valueOf(t.get(j));
            }
            tmp+=Integer.valueOf(t.get(0));
        }
        return tmp;
    }
}

```

1.48 Number of Digit One (hard->leetcode No.233)

Given an integer n , count the total number of digit 1 appearing in all non-negative integers less than or equal to n .

For example:

Given $n = 13$,

Return 6, because digit 1 occurred in the following numbers: 1, 10, 11, 12, 13.

1的个数	含1的数字	数字范围
1	1	[1, 9]
11	10 11 12 13 14 15 16 17 18 19	[10, 19]
1	21	[20, 29]
1	31	[30, 39]
1	41	[40, 49]
1	51	[50, 59]
1	61	[60, 69]
1	71	[70, 79]
1	81	[80, 89]
1	91	[90, 99]
11	100 101 102 103 104 105 106 107 108 109	[100, 109]
21	110 111 112 113 114 115 116 117 118 119	[110, 119]
11	120 121 122 123 124 125 126 127 128 129	[120, 129]

通过上面的列举我们可以发现，100以内的数字，除了10-19之间有11个‘1’之外，其余都只有1个。如果我们不考虑[10, 19]区间上那多出来的10个‘1’的话，那么我们在对任意一个两位数，十位数上的数字(加1)就代表1出现的个数，这时候我们再把多出的10个加上即可。比如56就有(5+1)+10=16个。如何知道是否要加上多出的10个呢，我们就要看十位上的数字是否大于等于2，是的话就要加上多余的10个‘1’。那么我们就可以用(x+8)/10来判断一个数是否大于等于2。对于三位数也是一样，除了[110, 119]之间多出的10个数之外，其余的每10个数的区间都只有11个‘1’，那么还是可以用相同的方法来判断并累加1的个数，参见代码如下：

Solution Explanation.

Let's start by counting the ones for every 10 numbers...

0, 1, 2, 3 ... 9 (1)

10, 11, 12, 13 ... 19 (1) + 10

20, 21, 22, 23 ... 29 (1)

...

90, 91, 92, 93 ... 99 (1)

•

100, 101, 102, 103 ... 109 (10 + 1)

110, 111, 112, 113 ... 119 (10 + 1) + 10

120, 121, 122, 123 ... 129 (10 + 1)

...

190, 191, 192, 193 ... 199 (10 + 1)

1). If we don't look at those special rows (start with 10, 110 etc), we know that there's a one at ones' place in every 10 numbers, there are 10 ones at tens' place in every 100 numbers, and 100 ones at hundreds' place in every 1000 numbers, so on and so forth.

Ok, let's start with ones' place and count how many ones at this place, set $k = 1$, as mentioned above, there's a one at ones' place in every 10 numbers, so how many 10 numbers do we have?

The answer is $(n / k) / 10$.

Now let's count the ones in tens' place, set $k = 10$, as mentioned above, there are 10 ones at tens' place in every 100 numbers, so how many 100 numbers do we have?

The answer is $(n / k) / 10$, and the number of ones at ten's place is $(n / k) / 10 * k$.

Let $r = n / k$, now we have a formula to count the ones at k's place: **$r / 10 * k$**

•

2). So far, everything looks good, but we need to fix those special rows, how?

We can use the mod. Take 10, 11, and 12 for example, if n is 10, we get $(n / 1) / 10 * 1 = 1$ ones at ones's place, perfect, but for tens' place, we get $(n / 10) / 10 * 10 = 0$, that's not right, there should be a one at tens' place! Calm down, from 10 to 19, we always have a one at tens's place, let $m = n \% k$, the number of ones at this special place is $m + 1$, so let's fix the formula to be:

$r / 10 * k + (r \% 10 == 1 ? m + 1 : 0)$

•

```
public class Solution {
```

```

public int countDigitOne(int n) {
    int count = 0;
    for(long k=1;k<=n;k*=10){
        long r = n/k, m=n%k;
        count+=(r+8)/10*k+(r%10==1?m+1:0);
    }
    return count;
}
}

```

3). Wait, how about 20, 21 and 22?

Let's say 20, use the above formula we get 0 ones at tens' place, but it should be 10! How to fix it? We know that once the digit is larger than 2, we should add 10 more ones to the tens' place, a clever way to fix is to add 8 to r, so our final formula is:

$(r + 8) / 10 * k + (r \% 10 == 1 ? m + 1 : 0)$

As you can see, it's all about how we fix the formula. Really hope that makes sense to you.

1.49 Sliding Window Maximum (hard->leetcode No.239)

Given an array *nums*, there is a sliding window of size *k* which is moving from the very left of the array to the very right. You can only see the *k* numbers in the window. Each time the sliding window moves right by one position.

For example,

Given *nums* = [1,3,-1,-3,5,3,6,7], and *k* = 3.

Window position	Max
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5
1 3 -1 [-3 5 3] 6 7	5
1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7

Therefore, return the max sliding window as [3,3,5,5,6,7].

Note:

You may assume *k* is always valid, ie: $1 \leq k \leq$ input array's size for non-empty array.

Follow up:

Could you solve it in linear time?

 Send

```

public class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        if(nums==null || nums.length==0) return new int[0];
    }
}

```

```

PriorityQueue<Integer> queue = new PriorityQueue<>(new Comparator<Integer>(){
    public int compare(Integer l1, Integer l2){
        return l2-l1;
    }
});

List<Integer> res = new ArrayList<>();
for(int i=0;i<nums.length;i++){
    if(queue.size()>=k){
        res.add(queue.peek());
        queue.remove(nums[i-k]);
    }
    queue.add(nums[i]);
}
if(!queue.isEmpty() && queue.size()<=k) res.add(queue.poll());
int[] result = new int[res.size()];
for(int i=0;i<res.size();i++) result[i] = res.get(i);
return result;
}
}

```

1.50 Paint House II (Hard->leetcode No.265)

See Blocked II

1.51 Expression Add Operators (hard->leetcode No.282)

Given a string that contains only digits 0-9 and a target value, return all possibilities to add **binary** operators (not unary) **+**, **-**, or ***** between the digits so they evaluate to the target value.

Examples:

```

"123", 6 -> ["1+2+3", "1*2*3"]
"232", 8 -> ["2*3+2", "2+3*2"]
"105", 5 -> ["1*0+5", "10-5"]
"00", 0 -> ["0+0", "0-0", "0*0"]
"3456237490", 9191 -> []

```

因为要输出所有可能的情况，必定是用深度优先搜索。问题在于如何将问题拆分成多次搜索。加减法很好处理，每当我们截出一段数字时，将之前计算的结果加上或者减去这个数，就可以将剩余的字符串和新的计算结果代入下一次搜索中了，直到我们的计算结果和目标一样，就完成了搜索。然而，乘法如何处理呢？这里我们需要用一个变量记录乘法当前累乘的值，直到累乘完了，遇到下一个加号或减号再将其算入计算结果中。这里有两种情况：

1. 乘号之前是加号或减号，例如 $2+3*4$ ，我们在2那里算出来的结果，到3的时候会加上3，计算结果变为5。在到4的时候，因为4之前我们选择的是乘号，这里3就应该和4相乘，而不是和2相加，所以在计算结果时，要将5先减去刚才加的3得到2，然后再加上3乘以4，得到 $2+12=14$ ，这样14就是到4为止时的计算结果。
2. 另外一种情况是乘号之前也是乘号，如果 $2+3*4*5$ ，这里我们到4为止计算的结果是14了，然后我们到5的时候又是乘号，这时候我们要把刚才加的 $3*4$ 给去掉，然后再加上 $3*4*5$ ，也就是 $14-3*4+3*4*5=62$ 。这样5的计算结果就是62。

因为要解决上述几种情况，我们需要这么几个变量，一个是记录上次的计算结果 `currRes`，一个是记录上次被加或者被减的数 `prevNum`，一个是当前准备处理的数 `currNum`。当下一轮搜索是加减法时，`prevNum` 就是简单换成 `currNum`，当下一轮搜索是乘法时，`prevNum` 是 `prevNum` 乘以 `currNum`。

注意

- 第一次搜索不添加运算符，只添加数字，就不会出现 $+1+2$ 这种表达式了。
- 我们截出的数字不能包含0001这种前面有0的数字，但是一个0是可以的。这里一旦截出的数字前导为0，就可以return了，因为说明前面就截的不对，从这之后都是开始为0的，后面也不

```
public class Solution {
    List<String> res = new ArrayList<>();
    public List<String> addOperators(String num, int target) {
        dfs(num, target, "", 0, 0);
        return res;
    }
    private void dfs(String num, int target, String tmp, long curRes, long prevNum) {
        if (num.length() == 0 && curRes == target) {
            String result = new String(tmp);
            res.add(result);
            return;
        }

        for (int i = 1; i <= num.length(); i++) {
            String curStr = num.substring(0, i);
            if (curStr.length() > 1 && curStr.charAt(0) == '0') return;
            long curNum = Long.parseLong(curStr);
            String next = num.substring(i);
```

```

        if(tmp.length()!=0){
            dfs(next, target, tmp+"*"+curNum, (curRes-prevNum)+(prevNum*curNum), prevNum*curNum);
        }
        // process the multiplication
        dfs(next, target, tmp+"+"+curStr, curRes+curNum, curNum); // add
        dfs(next, target, tmp+"-"+curStr, curRes-curNum, -curNum); // subtract
    }else dfs(next, target, curStr, curNum, curNum);
}
}
}

```

1.52 Find the Duplicate Number (hard->leetcode No. 287)

Given an array *nums* containing $n + 1$ integers where each integer is between 1 and n (inclusive), prove that at least one duplicate number must exist. Assume that there is only one duplicate number, find the duplicate one.

Note:

1. You **must not** modify the array (assume the array is read only).
2. You must use only constant, $O(1)$ extra space.
3. Your runtime complexity should be less than $O(n^2)$.
4. There is only one duplicate number in the array, but it could be repeated more than once.

```

public class Solution {
    public int findDuplicate(int[] nums) {
        int min = 0;
        int max = nums.length-1;

        while(min<=max){
            int mid = min+(max-min)/2;
            int count = 0;
            for(int i=0;i<nums.length;i++){
                if(nums[i]<=mid) count++;
            }

            if(count>mid) max = mid-1;
            else min = mid+1;
        }
        return min;
    }
}

```

映射找环法

复杂度

时间 $O(N)$ 空间 $O(1)$

思路

假设数组中没有重复，那我们可以做到这么一点，就是将数组的下标和1到n每一个数一对一的映射起来。比如数组是 213，则映射关系为 $0 \rightarrow 2, 1 \rightarrow 1, 2 \rightarrow 3$ 。假设这个一对一映射关系是一个函数 $f(n)$ ，其中 n 是下标， $f(n)$ 是映射到的数。如果我们从下标为0出发，根据这个函数计算出一个值，以这个值为新的下标，再用这个函数计算，以此类推，直到下标超界。实际上可以产生一个类似链表一样的序列。比如在这个例子中有两个下标的序列， $0 \rightarrow 2 \rightarrow 3$ 。

但如果有重复的话，这中间就会产生多对一的映射，比如数组 2131，则映射关系为 $0 \rightarrow 2, \{1, 3\} \rightarrow 1, 2 \rightarrow 3$ 。这样，我们推演的序列就一定会有环路了，这里下标的序列是 $0 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 1 \rightarrow 1 \rightarrow 1 \rightarrow \dots$ ，而环的起点就是重复的数。

所以该题实际上就是找环路起点的题，和 [Linked List Cycle II](#) 一样。我们先用快慢两个下标都从0开始，快下标每轮映射两次，慢下标每轮映射一次，直到两个下标再次相同。这时候保持慢下标位置不变，再用一个新的下标从0开始，这两个下标都继续每轮映射一次，当这两个下标相遇时，就是环的起点，也就是重复的数。对这个找环起点算法不懂的，请参考 [Floyd's Algorithm](#)。

```
public class Solution {
    public int findDuplicate(int[] nums) {
        int slow = 0;
        int fast = 0;
        do{
            slow = nums[slow];
            fast = nums[nums[fast]];
        }while(slow!=fast);
        int find= 0;
        while(find!=slow){
            slow = nums[slow];
            find = nums[find];
        }
        return find;
    }
}
```

1.53 Find Median From data Stream (hard-> leetcode NO.295)

Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.

Examples:

[2,3,4] , the median is 3

[2,3] , the median is $(2 + 3) / 2 = 2.5$

Design a data structure that supports the following two operations:

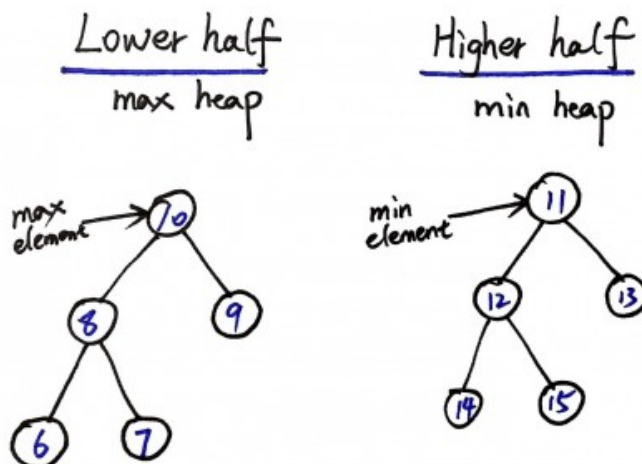
- void addNum(int num) - Add a integer number from the data stream to the data structure.
- double findMedian() - Return the median of all elements so far.

For example:

```
add(1)
add(2)
findMedian() -> 1.5
add(3)
findMedian() -> 2
```

First of all, it seems that the best time complexity we can get for this problem is $O(\log(n))$ of add() and $O(1)$ of getMedian(). This data structure seems highly likely to be a tree.

We can use heap to solve this problem. In Java, the `PriorityQueue` class is a priority heap. We can use two heaps to store the lower half and the higher half of the data stream. The size of the two heaps differs at most 1.



```
public class MedianFinder {
    PriorityQueue<Integer> maxHeap;
    PriorityQueue<Integer> minHeap;
    public MedianFinder(){
        maxHeap = new PriorityQueue<>(Collections.reverseOrder());
```

```

        minHeap = new PriorityQueue<>();
    }
    // Adds a number into the data structure.
    public void addNum(int num) {
        maxHeap.offer(num);
        minHeap.offer(maxHeap.poll());
        if(maxHeap.size()<minHeap.size()) maxHeap.offer(minHeap.poll());
    }

    // Returns the median of current data stream
    public double findMedian() {
        if(maxHeap.size()==minHeap.size())
            return (double)((maxHeap.peek()+minHeap.peek())/2);
        else return maxHeap.peek();
    }
};

```

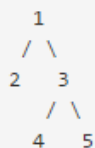
// Your MedianFinder object will be instantiated and called as such:
// MedianFinder mf = new MedianFinder();
// mf.addNum(1);
// mf.findMedian();

1.54 Serialize and Deserialize Binary Tree(hard->leetcode No.297)

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

For example, you may serialize the following tree



as "[1,2,3,null,null,4,5]", just the same as [how LeetCode OJ serializes a binary tree](#). You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.

Note: Do not use class member/global/static variables to store states. Your serialize and deserialize algorithms should be stateless.

```

/**
 * Definition for a binary tree node.
 * public class TreeNode {

```

```

*   int val;
*   TreeNode left;
*   TreeNode right;
*   TreeNode(int x) { val = x; }
* }
*/
public class Codec {

    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
        if(root==null) return "";
        LinkedList<TreeNode> queue = new LinkedList<>();
        StringBuilder sb = new StringBuilder();
        queue.add(root);
        while(!queue.isEmpty()){
            TreeNode node = queue.poll();
            if(node==null) sb.append("#,");
            else{
                sb.append(node.val+",");
                queue.add(node.left);
                queue.add(node.right);
            }
        }
        sb.deleteCharAt(sb.length()-1);
        return sb.toString();
    }

    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        if(data==null || data.length()==0) return null;
        String[] datas = data.split(",");
        TreeNode root = new TreeNode(Integer.parseInt(datas[0]));
        LinkedList<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        int i=1;
        while(!queue.isEmpty()){
            TreeNode node = queue.poll();
            if(node==null) continue;
            if(datas[i].equals("#")){
                node.left=null;
                queue.add(null);
            }else{
                node.left = new TreeNode(Integer.parseInt(datas[i]));
                queue.add(node.left);
            }
            i++;
            if(datas[i].equals("#")){
                node.right=null;
            }
        }
    }
}

```

```

        queue.add(null);
    }else{
        node.right = new TreeNode(Integer.parseInt(datas[i]));
        queue.add(node.right);
    }
    i++;
}
return root;
}
}

```

// Your Codec object will be instantiated and called as such:

// Codec codec = new Codec();

// codec.deserialize(codec.serialize(root));

1.55 Remove Invalid Parentheses (hard->leetcode No.301)

Remove the minimum number of invalid parentheses in order to make the input string valid. Return all possible results.

Note: The input string may contain letters other than the parentheses (and).

Examples:

```

"()())()" -> ["()()()", "(())()"]
"(a)())()" -> ["(a)()()", "(a())()"]
")(" -> [""]

```

```

public class Solution {
    private List<String> res = new ArrayList<>();
    private int max = 0;
    public List<String> removeInvalidParentheses(String s) {
        dfs(s, "", 0, 0);
        if(res.size()==0) res.add("");
        return res;
    }

    private void dfs(String s, String sub, int countLeft, int maxLeft){
        // countLeft is the # of '(' and the maxLeft is the # of pair of ()
        // base case
        if(s.length()==0){
            if(countLeft==0 && sub.length()!=0){
                if(maxLeft>max) max = maxLeft;
                if(maxLeft==max && !res.contains(sub)) res.add(sub);
            }
            return;
        }

        if(s.charAt(0)=='('){
            // two choices 1) keep and 2) miss

```

```

        dfs(s.substring(1), sub.concat("("), countLeft+1, maxLeft+1);
        dfs(s.substring(1), sub, countLeft, maxLeft);
    }else if(s.charAt(0)==''){
        if(countLeft>0) dfs(s.substring(1), sub.concat("("), countLeft-1, maxLeft);
        dfs(s.substring(1), sub, countLeft, maxLeft);
    }else{
        dfs(s.substring(1), sub.concat(String.valueOf(s.charAt(0))), countLeft, maxLeft);
    }
}
}

```

1.56 Burst Balloons (hard->leetcode No.312)

Given n balloons, indexed from 0 to $n-1$. Each balloon is painted with a number on it represented by array `nums`. You are asked to burst all the balloons. If the you burst balloon i you will get `nums[left] * nums[i] * nums[right]` coins. Here `left` and `right` are adjacent indices of i . After the burst, the `left` and `right` then becomes adjacent.

Find the maximum coins you can collect by bursting the balloons wisely.

Note:

(1) You may imagine `nums[-1] = nums[n] = 1`. They are not real therefore you can not burst them.

(2) $0 \leq n \leq 500$, $0 \leq \text{nums}[i] \leq 100$

Example:

Given `[3, 1, 5, 8]`

Return `167`

```

nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []
coins = 3*1*5 + 3*5*8 + 1*3*8 + 1*8*1 = 167

```

那么介于 i,j 之间的 x ，有： $dp[i][j] = \max(dp[i][j], dp[i][x-1] + \text{nums}[i-1] * \text{nums}[x] * \text{nums}[j+1] + dp[x+1][j])$;

这里，为了方便代码书写，我在首尾插入了两个1，所以答案是 `dp[1][n]` (n 为原来的长度)

可以用记忆化搜索也可以直接迭代DP,当然，记忆化搜索更好理解一点。

```

public class Solution {
    public int DP(int i, int j, int[] nums, int[][] dp) {
        if (dp[i][j] > 0) return dp[i][j];
        for (int x = i; x <= j; x++) {

```

```

        dp[i][j] = Math.max(dp[i][j], DP(i, x - 1, nums, dp) + nums[i - 1] * nums[x] * nums[j + 1] + DP(x + 1, j,
nums, dp));
    }
    return dp[i][j];
}

```

```

public int maxCoins(int[] iNums) {
    int n = iNums.length;
    int[] nums = new int[n + 2];
    for (int i = 0; i < n; i++) nums[i + 1] = iNums[i];
    nums[0] = nums[n + 1] = 1;
    int[][] dp = new int[n + 2][n + 2];
    return DP(1, n, nums, dp);
}
}

```

1.57 Count of Smaller Numbers after self(hard->leetcode No.315)

You are given an integer array *nums* and you have to return a new *counts* array. The *counts* array has the property where `counts[i]` is the number of smaller elements to the right of `nums[i]`.

Example:

Given *nums* = [5, 2, 6, 1]

To the right of 5 there are 2 smaller elements (2 and 1).
 To the right of 2 there is only 1 smaller element (1).
 To the right of 6 there is 1 smaller element (1).
 To the right of 1 there is 0 smaller element.

Return the array [2, 1, 1, 0].

```

class BSTNode{
    int val;
    int numOfLeftCnt;
    int dup = 1;
    BSTNode left;
    BSTNode right;
    public BSTNode(int val, int numOfLeftCnt){
        this.val = val;
        this.numOfLeftCnt = numOfLeftCnt;
    }
}

public class Solution {
    public List<Integer> countSmaller(int[] nums) {
        Integer[] res = new Integer[nums.length]; //res stores the # of leftnodes
        BSTNode root = null;
        for(int i=nums.length-1;i>=0;i--){
            /**
             res: store the # of left nodes
             i: the index of the nums[i]

```

```

        0: is the# of left nodes
        */
        root = insert(res, nums[i], i, root, 0);
    }
    return Arrays.asList(res);
}

private BSTNode insert(Integer[] res, int num, int i, BSTNode root, int preNum){
    if(root==null){
        root = new BSTNode(num, 0);
        res[i]=preNum;
        return root;
    }else if(root.val==num){
        root.dup++;
        res[i] = root.numOfLeftCnt+preNum;
    }else if(root.val>num){
        root.numOfLeftCnt++;
        root.left = insert(res, num, i, root.left, preNum);
    }else{
        root.right = insert(res, num, i, root.right, preNum+root.dup+root.numOfLeftCnt);
    }
    return root;
}
}

```

1.58 Remove Duplicate Letters(hard->leetcode No.316)

Given a string which contains only lowercase letters, remove duplicate letters so that every letter appear once and only once. You must make sure your result is the smallest in lexicographical order among all possible results.

Example:

Given "bcabc"

Return "abc"

Given "cbacdcbc"

Return "acdb"

```

public class Solution {
    public String removeDuplicateLetters(String s) {
        if(s==null || s.length()<=1) return s;
        Map<Character, Integer> map = new HashMap<>();
        for(int i=0;i<s.length();i++){
            char c = s.charAt(i);
            map.put(c, i);
        }

        int start = 0;
        int end = findMinIndex(map);
        StringBuilder sb = new StringBuilder();
    }
}

```

```

while(!map.isEmpty()){
    char bound='z'+1;
    int index = 0;
    for(int i=start;i<=end;i++){
        char c = s.charAt(i);
        if(c<bound && map.containsKey(c)){
            bound = c;
            index = i;
        }
    }
    sb.append(bound);
    map.remove(bound);
    start = index+1;
    end = findMinIndex(map);
}
return sb.toString();
}

private int findMinIndex(Map<Character, Integer> map){
    int min = Integer.MAX_VALUE;
    for(int res:map.values()){
        min = Math.min(min, res);
    }
    return min;
}
}

```

1.59 Create maximum Number (hard->leetcode No.321)

Given two arrays of length m and n with digits $0-9$ representing two numbers. Create the maximum number of length $k \leq m + n$ from digits of the two. The relative order of the digits from the same array must be preserved. Return an array of the k digits. You should try to optimize your time and space complexity.

Example 1:

nums1 = [3, 4, 6, 5]

nums2 = [9, 1, 2, 5, 8, 3]

k = 5

return [9, 8, 6, 5, 3]

Example 2:

nums1 = [6, 7]

nums2 = [6, 0, 4]

k = 5

return [6, 7, 6, 0, 4]

Example 3:

nums1 = [3, 9]

nums2 = [8, 9]

k = 3

return [9, 8, 9]

```
public class Solution {
    /**
     * @param nums1 an integer array of length m with digits 0-9
     * @param nums2 an integer array of length n with digits 0-9
     * @param k an integer and k <= m + n
     * @return an integer array
     */
    public int[] maxNumber(int[] nums1, int[] nums2, int k) {
        // Write your code here
        if (k == 0)
            return new int[0];

        int m = nums1.length, n = nums2.length;
        if (m + n < k) return null;
        if (m + n == k) {
            int[] results = merge(nums1, nums2, k);
            return results;
        } else {
            int max = m >= k ? k : m;
            int min = n >= k ? 0 : k - n;

            int[] results = new int[k];
            for(int i=0; i < k; ++i)
                results[i] = -0x7ffffff;
            for(int i = min; i <= max; ++i) {
                int[] temp = merge(getMax(nums1, i), getMax(nums2, k - i), k);
                results = isGreater(results, 0, temp, 0) ? results : temp;
            }
            return results;
        }
    }

    private int[] merge(int[] nums1, int[] nums2, int k) {
        int[] results = new int[k];
        if (k == 0) return results;
        int i = 0, j = 0;
        for(int l = 0; l < k; ++l) {
            results[l] = isGreater(nums1, i, nums2, j) ? nums1[i++] : nums2[j++];
        }
    }
}
```

```

        return results;
    }

    private boolean isGreater(int[] nums1, int i, int[] nums2, int j) {
        for(; i < nums1.length && j < nums2.length; ++i, ++j) {
            if (nums1[i] > nums2[j])
                return true;
            if (nums1[i] < nums2[j])
                return false;
        }
        return i != nums1.length;
    }

    private int[] getMax(int[] nums, int k) {
        if (k == 0)
            return new int[0];
        int[] results = new int[k];
        int i = 0;
        for(int j = 0; j < nums.length; j++) {
            while(nums.length - j + i > k && i > 0 && results[i-1] < nums[j])
                i--;
            if (i < k)
                results[i++] = nums[j];
        }
        return results;
    }
}

```

1.60 Count of Range Sum (hard->leetcode No. 327)

Given an integer array `nums`, return the number of range sums that lie in `[lower, upper]` inclusive.

Range sum `S(i, j)` is defined as the sum of the elements in `nums` between indices `i` and `j` ($i \leq j$), inclusive.

Note:

A naive algorithm of $O(n^2)$ is trivial. You MUST do better than that.

Example:

Given `nums = [-2, 5, -1]`, `lower = -2`, `upper = 2`,

Return `3`.

The three ranges are : `[0, 0]`, `[2, 2]`, `[0, 2]` and their respective sums are: `-2`, `-1`, `2`.

```

public class Solution {
    long[] count;
    int lower, upper;
    public int countRangeSum(int[] nums, int lower, int upper) {
        if(nums==null || nums.length==0) return 0;
    }
}

```

```

        count = new long[nums.length];
        count[0] = nums[0];
        this.lower = lower;
        this.upper = upper;
        for(int i=1;i<nums.length;i++) count[i] = count[i-1]+nums[i];
        return countNum(nums, 0, nums.length-1);
    }

    private int countNum(int[] nums, int left, int right){
        if(left==right){
            if(lower<=nums[left] && nums[right]<=upper) return 1;
            return 0;
        }
        int total = 0;
        int mid = left+(right-left)/2;
        for(int i=left;i<=mid;i++){
            for(int j=mid+1;j<=right;j++){
                long tmp = count[j]-count[i]+nums[i];
                if(lower<=tmp && tmp<=upper) total++;
            }
        }
        return total+countNum(nums, left, mid)+countNum(nums, mid+1,right);
    }
}

```

Solution 2: AC

```

public class Solution {
    public int countRangeSum(int[] nums, int lower, int upper) {
        List<Long> cand = new ArrayList<>();
        cand.add(Long.MIN_VALUE); // make sure no number gets a 0-index.
        cand.add(0L);
        long[] sum = new long[nums.length + 1];
        for (int i = 1; i < sum.length; i++) {
            sum[i] = sum[i - 1] + nums[i - 1];
            cand.add(sum[i]);
            cand.add(lower + sum[i - 1] - 1);
            cand.add(upper + sum[i - 1]);
        }
        Collections.sort(cand); // finish discretization

        int[] bit = new int[cand.size()];
        for (int i = 0; i < sum.length; i++) plus(bit, Collections.binarySearch(cand, sum[i]), 1);
        int ans = 0;
        for (int i = 1; i < sum.length; i++) {
            plus(bit, Collections.binarySearch(cand, sum[i - 1]), -1);
            ans += query(bit, Collections.binarySearch(cand, upper + sum[i - 1]));
            ans -= query(bit, Collections.binarySearch(cand, lower + sum[i - 1] - 1));
        }
    }
}

```

```

        return ans;
    }

    private void plus(int[] bit, int i, int delta) {
        for (; i < bit.length; i += i & -i) bit[i] += delta;
    }

    private int query(int[] bit, int i) {
        int sum = 0;
        for (; i > 0; i -= i & -i) sum += bit[i];
        return sum;
    }
}

```

1.61 Longest Increasing Path in a Matrix(hard->leetcode No.329)

Given an integer matrix, find the length of the longest increasing path.

From each cell, you can either move to four directions: left, right, up or down. You may NOT move diagonally or move outside of the boundary (i.e. wrap-around is not allowed).

Example 1:

```

nums = [
  [9,9,4],
  [6,6,8],
  [2,1,1]
]

```

Return 4

The longest increasing path is [1, 2, 6, 9] .

Example 2:

```

nums = [
  [3,4,5],
  [3,2,6],
  [2,2,1]
]

```

Return 4

The longest increasing path is [3, 4, 5, 6] . Moving diagonally is not allowed.

```

public class Solution {
    int[] dx = {0,0,1,-1};
    int[] dy = {1,-1, 0,0};
    public int longestIncreasingPath(int[][] matrix) {
        if(matrix==null || matrix.length==0 || matrix[0].length==0) return 0;
        int[][] memo = new int[matrix.length][matrix[0].length];
        int longest = 0;

```

 Se

```

    for(int i=0;i<matrix.length;i++){
        for(int j=0;j<matrix[0].length;j++){
            longest = Math.max(longest, dfs(matrix, i, j, memo));
        }
    }
    return longest;
}

private int dfs(int[][] matrix, int i, int j, int[][] memo){
    int m = matrix.length;
    int n = matrix[0].length;
    if(memo[i][j]!=0) return memo[i][j];
    for(int k=0;k<4;k++){
        int x=i+dx[k];
        int y = j+dy[k];
        if(x>=0 && x<m && y>=0 && y<n && matrix[x][y]>matrix[i][j])
            memo[i][j]=Math.max(memo[i][j], dfs(matrix, x, y, memo));
    }
    return ++memo[i][j]; //here should add 1 first
}
}

```

1.62 Self Crossing (hard -> leetcode No.335)

You are given an array x of n positive numbers. You start at point $(0,0)$ and moves $x[0]$ metres to the north, then $x[1]$ metres to the west, $x[2]$ metres to the south, $x[3]$ metres to the east and so on. In other words, after each move your direction changes counter-clockwise.

Write a one-pass algorithm with $O(1)$ extra space to determine, if your path crosses itself, or not.

Example 1:

Given $x = [2, 1, 1, 2]$,



Return true (self crossing)

Example 2:

Given $x = [1, 2, 3, 4]$,



Return **false** (not self crossing)

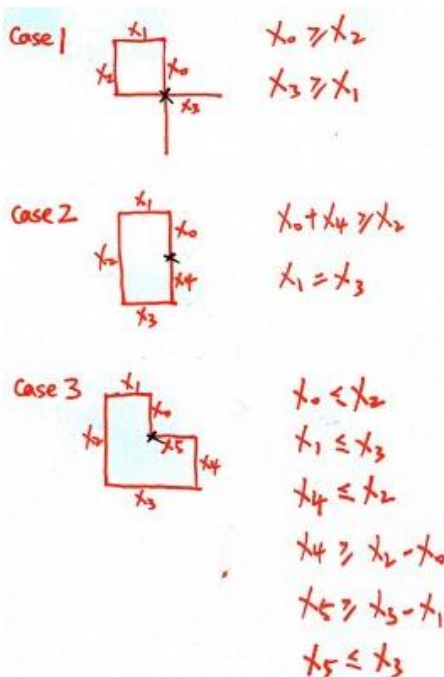
Example 3:

Given $x = [1, 1, 1, 1]$,



Return **true** (self crossing)

This problem can be easily solved if the three self crossing cases are summarized well. Here are the three self crossing cases. There are no other self crossing situations based on the restrictions of counter-clockwise.



```
public class Solution {  
    public boolean isSelfCrossing(int[] x) {  
        if(x==null || x.length<=3)  
            return false;  
    }  
}
```

```

for(int i=3; i<x.length; i++){
    if(x[i-3] >= x[i-1] && x[i-2]<=x[i]){
        return true;
    }

    if(i>=4 && x[i-4]+x[i]>=x[i-2] && x[i-3]==x[i-1]) {
        return true;
    }

    if(i>=5 && x[i-5]<=x[i-3] && x[i]<=x[i-2]&& x[i-1]<=x[i-3] && x[i-4]<=x[i-2] && x[i-1]>=x[i-3]-x[i-5] &&
x[i]>=x[i-2]-x[i-4]){
        return true;
    }
}

return false;
}
}

```

1.63 Data Stream as Disjoint intervals (hard->leetcode No.352)

Given a data stream input of non-negative integers $a_1, a_2, \dots, a_n, \dots$, summarize the numbers seen so far as a list of disjoint intervals.

For example, suppose the integers from the data stream are 1, 3, 7, 2, 6, ..., then the summary will be:

```

[1, 1]
[1, 1], [3, 3]
[1, 1], [3, 3], [7, 7]
[1, 3], [7, 7]
[1, 3], [6, 7]

```

Follow up:

What if there are lots of merges and the number of disjoint intervals are small compared to the data stream's size?

利用TreeSet数据结构，将不相交区间Interval存储在TreeSet中。

TreeSet底层使用红黑树实现，可以用log(n)的代价实现元素查找。

每次执行addNum操作时，利用TreeSet找出插入元素val的左近邻元素floor（start值不大于val的最大Interval），以及右近邻元素higher（start值严格大于val的最小Interval）

然后根据floor, val, higher之间的关系决定是否对三者进行合并。

```

/**
 * 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 SummaryRanges {
    TreeSet<Interval> set;
    /** Initialize your data structure here. */
    public SummaryRanges() {
        set = new TreeSet<>(new Comparator<Interval>(){
            public int compare(Interval a, Interval b){
                return a.start-b.start;
            }
        });
    }

    public void addNum(int val) {
        Interval valInterval = new Interval(val, val);
        Interval floor = set.floor(valInterval);
        if (floor != null) {
            if (floor.end >= val) return;
            else if (floor.end + 1 == val) {
                valInterval.start = floor.start;
                set.remove(floor);
            }
        }
        Interval higher = set.higher(valInterval);
        if (higher != null && higher.start == val + 1) {
            valInterval.end = higher.end;
            set.remove(higher);
        }
        set.add(valInterval);
    }

    public List<Interval> getIntervals() {
        return Arrays.asList(set.toArray(new Interval[0]));
    }
}

/**
 * Your SummaryRanges object will be instantiated and called as such:
 * SummaryRanges obj = new SummaryRanges();
 * obj.addNum(val);
 * List<Interval> param_2 = obj.getIntervals();
 */

```

1.64 Russian Doll Envelopes (hard->leetcode No.354)

You have a number of envelopes with widths and heights given as a pair of integers (w, h) . One envelope can fit into another if and only if both the width and height of one envelope is greater than the width and height of the other envelope.

What is the maximum number of envelopes can you Russian doll? (put one inside other)

Example:

Given envelopes = $[[5,4],[6,4],[6,7],[2,3]]$, the maximum number of envelopes you can Russian doll is 3
 $([2,3] \Rightarrow [5,4] \Rightarrow [6,7])$.

这道题给了我们一堆大小不一的信封，让我们像套俄罗斯娃娃那样把这些信封都给套起来，这道题实际上是之前那道Longest Increasing Subsequence的具体应用，而且难度增加了，从一维变成了二维，但是万变不离其宗，解法还是一样的，首先来看DP的解法，这是一种brute force的解法，首先要给所有的信封按从小到大排序，首先根据宽度从小到大排，如果宽度相同，那么高度小的再前面，这是STL里面sort的默认排序，所以我们不用写其他的comparator，直接排就可以了，然后我们开始遍历，对于每一个信封，我们都遍历其前面所有的信封，如果当前信封的长和宽都比前面那个信封的大，那么我们更新dp数组，通过 $dp[i] = \max(dp[i], dp[j] + 1)$ 。然后我们每遍历完一个信封，都更新一下结果res，参见代码如下；

```
public class Solution {
    public int maxEnvelopes(int[][] envelopes) {
        if(envelopes == null || envelopes.length == 0
            || envelopes[0] == null || envelopes[0].length != 2)
            return 0;
        Arrays.sort(envelopes, new Comparator<int[]>(){
            public int compare(int[] arr1, int[] arr2){
                if(arr1[0] == arr2[0])
                    return arr2[1] - arr1[1];
                else
                    return arr1[0] - arr2[0];
            }
        });
        int dp[] = new int[envelopes.length];
        int len = 0;
        for(int[] envelope : envelopes){
            int index = Arrays.binarySearch(dp, 0, len, envelope[1]);
            if(index < 0)
                index = -(index + 1);
            dp[index] = envelope[1];
            if(index == len)
                len++;
        }
        return len;
    }
}
```

