Contents

CHAPTER 1: String

1.1 Add Binary (Easy-> LeetCode No.67)

> Given a string, find the length of the **longest substring** without repeating characters.
>
> **Examples:**
>
> Given `"abcabcbb"`, the answer is `"abc"`, which the length is 3.
>
> Given `"bbbbb"`, the answer is `"b"`, with the length of 1.
>
> Given `"pwwkew"`, the answer is `"wke"`, with the length of 3. Note that the answer must be a **substring**, `"pwke"` is a *subsequence* and not a substring.
>
> Subscribe to see which companies asked this question
>
> Hide Tags | Hash Table | Two Pointers | String
>
> Hide Similar Problems | (H) Longest Substring with At Most Two Distinct Characters

Solution:

```
public class Solution {
   public int lengthOfLongestSubstring(String s) {
      if(s==null || s.length()==0) return 0;
      Map<Character, Integer> map = new HashMap<>();
      int max = 0;
      for(int i=0,j=0;i<s.length();i++){
         if(map.containsKey(s.charAt(i))){
            j=Math.max(j, map.get(s.charAt(i))+1);
         }
         map.put(s.charAt(i),i);
         max = Math.max(max, i-j+1);
      }
      return max;
   }
}
```

Remark: map always stores the recently inserted character and its index. i iterates normally the character index and j=Math.max(j, map.get(s.charAt(i)+1)) position.

1.2 Basic Calculator (Medium-> LeetCode No.227)

Implement a basic calculator to evaluate a simple expression string.

The expression string contains only **non-negative** integers, `+`, `-`, `*`, `/` operators and empty spaces . The integer division should truncate toward zero.

You may assume that the given expression is always valid.

Some examples:

```
"3+2*2" = 7
" 3/2 " = 1
" 3+5 / 2 " = 5
```

Note: Do not use the `eval` built-in library function.

Credits:

Special thanks to @ts for adding this problem and creating all test cases.

Subscribe to see which companies asked this question

Hide Tags String

Hide Similar Problems (H) Basic Calculator (H) Expression Add Operators

Have you met this question in a real interview? Yes No

Discuss Notes Pick One

Solution: this **solution is O(n) time and O(1) space**

To have O(1) space solution, we have to drop the stack. To see why we can drop it, we need to reexamine the main purpose of the stack: it is used to hold temporary results for partial expressions with lower precedence levels.
For problem 224. Basic Calculator, the depth of precedence levels is unknown, since you can have arbitrary levels of parentheses in the expression. Therefore we do need the stack in the solution. However for the current problem, we only have two precedence levels, lower level with '+' & '-' operations and higher level with '**' & '/' operations. So the stack can be replaced by two variables, one for the lower level and the other for the higher level. Note that when we are done with a partial expression involving '/' & '*' operations only, the result will fall back to the lower level.
Now let's look at each level separately.
First of course we will have a variable "num" to represent the current number involved in the operations.
For the lower level, we use a variable "pre" to denote the partial result. And as usual we will have a variable "sign" to indicate the sign of the incoming result.
For the higher level, we use a variable "curr" to represent the partial result, and another variable "op" to indicate what operation should be performed:

1. If op = 0, no '*' or '/' operation is needed and we simply assign num to curr;
2. if op = 1, we perform multiplication -- curr = curr * num;
3. if op = -1, we perform division -- curr = curr * num.

The key now is to figure out what to do depending on the scanned character from string s. There are three cases:

1. A digit is hit --> As usual we will update the variable "num". One more step here is that we need to determine if this is the last digit of the current number. If so, we need to perform the corresponding operation depending on the value of "op" and update the value of "curr" (It is assumed that we are at the higher precedence level by default);
2. A '*' or '/' is hit --> We need to update the value of "op" and reset "num" to 0;

3. A '+' or '-' is hit --> Current higher precedence level is over, so the partial result (which is denoted by "curr") will fall back to the lower level and can be incorporated into the lower level partial result "pre". And of course we need to update the "sign" as well as reset "op" and "num" to 0.

One last point is that the string will end with digit or space, so we need to add the result for the last partial higher level expression to "pre". Here is the Java program.

```
public class Solution {
   public int calculate(String s) {
      if(s==null || s.length()==0) return 0;
      int pre=0, curr=0, sign=1, op=0, num=0;
      for(int i=0;i<s.length();i++){
         if(Character.isDigit(s.charAt(i))){
            num = num*10+(s.charAt(i)-'0');
            if(i==s.length()-1 || !Character.isDigit(s.charAt(i+1))){
               curr=(op==0?num:(op==1?curr*num:curr/num));
            }
         }else if(s.charAt(i)=='+' || s.charAt(i)=='-'){
            pre+=sign*curr;
            sign = (s.charAt(i)=='+'?1:-1);
            op=0;
            num=0;
         }else if(s.charAt(i)=='*' || s.charAt(i)=='/'){
            op=(s.charAt(i)=='*'?1:-1);
            num = 0;
         }
      }
      return pre+sign*curr;
   }
}
```

1.3 Compare Version Numbers (Easy-> LeetCode No.165)

Compare two version numbers *version1* and *version2*.

If *version1* > *version2* return 1, if *version1* < *version2* return -1, otherwise return 0.

You may assume that the version strings are non-empty and contain only digits and the `.` character.

The `.` character does not represent a decimal point and is used to separate number sequences.

For instance, `2.5` is not "two and a half" or "half way to version three", it is the fifth second-level revision of the second first-level revision.

Here is an example of version numbers ordering:

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

Credits:

Special thanks to @ts for adding this problem and creating all test cases.

Subscribe to see which companies asked this question

Hide Tags String

Solution:

```
public class Solution {
```

```
    public int compareVersion(String version1, String version2) {
        String[] version_1 = version1.split("\\.");
        String[] version_2 = version2.split("\\.");
        int i=0;
        while(i<version_1.length || i<version_2.length){
            if(i<version_1.length && i<version_2.length){
                if(Integer.parseInt(version_1[i])<Integer.parseInt(version_2[i])) return -1;
                else if(Integer.parseInt(version_1[i])>Integer.parseInt(version_2[i])) return 1;
            }else if(i<version_1.length){
                if(Integer.parseInt(version_1[i])!=0) return 1;
            }else if(i<version_2.length){
                if(Integer.parseInt(version_2[i])!=0) return -1;
            }
            i++;
        }
        return 0;
    }
}
```

1.4 Count and Say (Easy-> LeetCode No.38)

The count-and-say sequence is the sequence of integers beginning as follows:

`1, 11, 21, 1211, 111221, ...`

`1` is read off as `"one 1"` or `11`.

`11` is read off as `"two 1s"` or `21`.

`21` is read off as `"one 2"`, then `one 1"` or `1211`.

Given an integer *n*, generate the n^{th} sequence.

Note: The sequence of integers will be represented as a string.

Subscribe to see which companies asked this question

Hide Tags String

Hide Similar Problems (M) Encode and Decode Strings

Solution:

```
public class Solution {
    public String countAndSay(int n) {
        if(n<=0) return null;
        int i=1;
        String result = "1";
        while(i<n){
```

```
            StringBuilder sb = new StringBuilder();
            int count = 1;
            for(int j=1;j<result.length();j++){
                if(result.charAt(j)==result.charAt(j-1)){
                    count++;
                }else{
                    sb.append(count);
                    sb.append(result.charAt(j-1));
                    count=1;
                }
            }
            sb.append(count);
            sb.append(result.charAt(result.length()-1));
            result = sb.toString();
            i++;
        }
        return result;
    }
}
```

1.5 Generate Parentheses (Medium-> LeetCode No.22)

Given *n* pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example, given *n* = 3, a solution set is:

```
"((()))", "(()())", "(())()", "()(())", "()()()"
```

Subscribe to see which companies asked this question

Hide Tags | Backtracking | String

Hide Similar Problems | (M) Letter Combinations of a Phone Number | (E) Valid Parentheses

Solution:

```
public class Solution {
    public List<String> generateParenthesis(int n) {
        List<String> result = new ArrayList<>();
        dfs(result, "", n,n);
        return result;
    }

    public void dfs(List<String> result, String s, int left, int right){
        if(left>right) return;
        if(left==0 && right==0) {
            result.add(s);
            return;
        }
```

```
        if(left>0) dfs(result, s+"(",left-1,right);
        if(right>0) dfs(result,s+")",left, right-1);
    }
}
```

1.6 Length of Last Word (Easy-> LeetCode No.58)

Given a string *s* consists of upper/lower-case alphabets and empty space characters `' '`, return the length of last word in the string.

If the last word does not exist, return 0.

Note: A word is defined as a character sequence consists of non-space characters only.

For example,

Given *s* = `"Hello World"`,

return `5`.

Subscribe to see which companies asked this question

Hide Tags String

Solution:

```
public class Solution {
    public int lengthOfLastWord(String s) {
        if(s==null || s.length()==0) return 0;
        int result = 0;
        int len = s.length();
        boolean flag = false;
        for(int i=len-1;i>=0;i--){
            char c = s.charAt(i);
            if((c>='a' && c<='z') || (c>='A' && c<='Z')){
                result++;
                flag = true;
            }else {
                if(flag) return result;
            }
        }
        return result;
    }
}
```

1.7 Letter Combinations of a Phone Number (Medium-> LeetCode No.17)

Given a digit string, return all possible letter combinations that the number could represent.

A mapping of digit to letters (just like on the telephone buttons) is given below.


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

Note:

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

Subscribe to see which companies asked this question

Hide Tags | Backtracking | String

Hide Similar Problems | (M) Generate Parentheses | (M) Combination Sum

Solution:

```
public class Solution {
    public List<String> letterCombinations(String digits) {
        HashMap<Integer, String> map = new HashMap<>();
        map.put(2,"abc");
        map.put(3,"def");
        map.put(4,"ghi");
        map.put(5,"jkl");
        map.put(6,"mno");
        map.put(7,"pqrs");
        map.put(8,"tuv");
        map.put(9,"wxyz");

        List<String> result = new ArrayList<>();
        if(digits==null || digits.length()==0) return result;
        List<Character> tmp = new ArrayList<>();
        dfs(result, tmp, digits, map);
        return result;
    }
```

```
    public void dfs(List<String> result, List<Character> tmp, String digits, HashMap<Integer, String> map){
        if(digits.length()==0){
            char[] carr = new char[tmp.size()];
            for(int i=0;i<tmp.size();i++) carr[i] = tmp.get(i);
            String s = String.valueOf(carr);
            result.add(s);
            return;
        }
        Integer curr = Integer.valueOf(digits.substring(0,1));
        String letters = map.get(curr);
        for(int i=0;i<letters.length();i++){
            tmp.add(letters.charAt(i));
            dfs(result, tmp, digits.substring(1),map);
            tmp.remove(tmp.size()-1);
        }
    }
}
```

1.8 Longest Common Prefix (Easy-> LeetCode No.14)

Solution:

```
public class Solution {
    public String longestCommonPrefix(String[] strs) {
        if(strs==null || strs.length==0) return "";
        if(strs.length==1) return strs[0];
        int shortest = 0;
        int len = strs[0].length();
        for(int i=1;i<strs.length;i++){
            String curr = strs[i];
            if(curr.length()<len){
                shortest = i;
                len = curr.length();
            }
        }

        // find the longest common prefix
        String sub = strs[shortest];
```

```
    for(int i=0;i<strs.length;i++){
      while(strs[i].indexOf(sub)!=0){
        sub = sub.substring(0,sub.length()-1);
      }
    }
    return sub;
  }
}
```

1.9 Longest Palindromic Substring (Medium-> LeetCode No.5)

Given a string *S*, find the longest palindromic substring in *S*. You may assume that the maximum length of *S* is 1000, and there exists one unique longest palindromic substring.

Subscribe to see which companies asked this question

Hide Tags String

Hide Similar Problems (H) Shortest Palindrome (E) Palindrome Permutation (H) Palindrome Pairs

Solution:

Solution

Approach #1 (Longest Common Substring) [Accepted]

Common mistake

Some people will be tempted to come up with a quick solution, which is unfortunately flawed (however can be corrected easily):

> Reverse S and become S'. Find the longest common substring between S and S', which must also be the longest palindromic substring.

This seemed to work, let's see some examples below.

For example, $S = \text{"caba"}$, $S' = \text{"abac"}$.

The longest common substring between S and S' is "aba", which is the answer.

Let's try another example: $S = \text{"abacdfgdcaba"}$, $S' = \text{"abacdgfdcaba"}$.

The longest common substring between S and S' is "abacd". Clearly, this is not a valid palindrome.

Algorithm

We could see that the longest common substring method fails when there exists a reversed copy of a non-palindromic substring in some other part of S. To rectify this, each time we find a longest common substring candidate, we check if the substring's indices are the same as the reversed substring's original indices. If it is, then we attempt to update the longest palindrome found so far; if not, we skip this and find the next candidate.

This gives us an $O(n^2)$ Dynamic Programming solution which uses $O(n^2)$ space (could be improved to use $O(n)$ space). Please read more about Longest Common Substring here.

Approach #3 (Dynamic Programming) [Accepted]

To improve over the brute force solution, we first observe how we can avoid unnecessary re-computation while validating palindromes. Consider the case **"ababa"**. If we already knew that **"bab"** is a palindrome, it is obvious that **"ababa"** must be a palindrome since the two left and right end letters are the same.

We define $P(i, j)$ as following:

$$P(i,j) = \begin{cases} \text{true}, & \text{if the substring } S_i \dots S_j \text{ is a palindrome} \\ \text{false}, & \text{otherwise.} \end{cases}$$

Therefore,

$$P(i,j) = (P(i+1, j-1) \text{ and } S_i == S_j)$$

The base cases are:

$$P(i,i) = true$$

$$P(i,i+1) = (S_i == S_{i+1})$$

This yields a straight forward DP solution, which we first initialize the one and two letters palindromes, and work our way up finding all three letters palindromes, and so on...

Complexity Analysis

- Time complexity : $O(n^2)$. This gives us a runtime complexity of $O(n^2)$.
- Space complexity : $O(n^2)$. It uses $O(n^2)$ space to store the table.

Additional Exercise

Could you improve the above space complexity further and how?

Approach #4 (Expand Around Center) [Accepted]

In fact, we could solve it in $O(n^2)$ time using only constant space.

We observe that a palindrome mirrors around its center. Therefore, a palindrome can be expanded from its center, and there are only $2n - 1$ such centers.

You might be asking why there are $2n - 1$ but not n centers? The reason is the center of a palindrome can be in between two letters. Such palindromes have even number of letters (such as "abba") and its center are between the two 'b's.

```
public String longestPalindrome(String s) {
    int start = 0, end = 0;
    for (int i = 0; i < s.length(); i++) {
        int len1 = expandAroundCenter(s, i, i);
        int len2 = expandAroundCenter(s, i, i + 1);
        int len = Math.max(len1, len2);
        if (len > end - start) {
            start = i - (len - 1) / 2;
            end = i + len / 2;
        }
    }
    return s.substring(start, end + 1);
}

private int expandAroundCenter(String s, int left, int right) {
    int L = left, R = right;
    while (L >= 0 && R < s.length() && s.charAt(L) == s.charAt(R)) {
        L--;
        R++;
    }
    return R - L - 1;
}
```

Complexity Analysis

- Time complexity : $O(n^2)$. Since expanding a palindrome around its center could take $O(n)$ time, the overall complexity is $O(n^2)$.
- Space complexity : $O(1)$.

1.10 ZigZag Conversion (Easy -> LeetCode No.6)

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

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

And then read line by line: `"PAHNAPLSIIGYIR"`

Write the code that will take a string and make this conversion given a number of rows:

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

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

Subscribe to see which companies asked this question

Hide Tags String

Solution:

1.11 String to Integer (Easy ->Leetcode No.8)
Solution: See CleanCodeHandbook-1 Question 8

1.12 Integer to Roman (Medium -> Leetcode No.12)
Solution: See CleanCodeHandbook-1 Question 36

1.13 Roman to Integer (Easy -> Leetcode No.13)
Solution: See CleanCodeHandbook-1 Question 37

1.14 Valid Parentheses (Easy -> Leetcode No.20)
Solution: See CleanCodeHandbook-1 Question 41

1.15 Implement strStr() (Easy -> Leetcode No.28)
Solution: See CleanCodeHandbook-1 Question 5

1.16 Multiply Strings (Medium -> Leetcode No.43)
Solution:

Given two numbers represented as strings, return multiplication of the numbers as a string.

Note:

- The numbers can be arbitrarily large and are non-negative.
- Converting the input string to integer is **NOT** allowed.
- You should **NOT** use internal library such as **BigInteger**.

Subscribe to see which companies asked this question

Hide Tags | Math | String

Hide Similar Problems | (M) Add Two Numbers | (E) Plus One | (E) Add Binary

```
public class Solution {
   public String multiply(String num1, String num2) {
      String n1 = new StringBuilder(num1).reverse().toString();
      String n2 = new StringBuilder(num2).reverse().toString();

      int[] result = new int[n1.length()+n2.length()];
      StringBuilder sb = new StringBuilder();
      for(int i=0;i<n1.length();i++){
         for(int j =0;j<n2.length();j++){
            result[i+j] += (n1.charAt(i)-'0')*(n2.charAt(j)-'0');
         }
      }
```

```
        for(int k=0;k<result.length;k++){
            int mod = result[k]%10;
            int carry = result[k]/10;
            if(k+1<result.length) result[k+1]+=carry;
            sb.insert(0,mod);
        }

        while(sb.charAt(0)=='0' && sb.length()>1) sb.deleteCharAt(0);
        return sb.toString();
    }
}
```

1.17 Simplify Path(Medium -> Leetcode No.71)

Given an absolute path for a file (Unix-style), simplify it.

For example,

path = `"/home/"`, => `"/home"`

path = `"/a/./b/../../c/"`, => `"/c"`

click to show corner cases.

Corner Cases:

- Did you consider the case where **path** = `"/../"`?

  In this case, you should return `"/"`.

- Another corner case is the path might contain multiple slashes `'/'` together, such as `"/home//foo/"`.

  In this case, you should ignore redundant slashes and return `"/home/foo"`.

Subscribe to see which companies asked this question

Hide Tags Stack String

Solution:

```
public class Solution {
    public String simplifyPath(String path) {
        Stack<String> stack = new Stack<>();
        while(path.length()>1 && path.charAt(path.length()-1)=='/'){
            path = path.substring(0,path.length()-1);
        }

        int start = 0;
```

```
        for(int i=1;i<path.length();i++){
            if(path.charAt(i)=='/'){
                stack.push(path.substring(start,i));
                start = i;
            }else if(i==path.length()-1){
                stack.push(path.substring(start));
            }
        }

        LinkedList<String> result = new LinkedList<>();
        int back = 0;
        while(!stack.isEmpty()){
            String top = stack.pop();
            if(top.equals("/.") || top.equals("/"));
            else if(top.equals("/..")) back++;
            else{
                if(back>0) back--;
                else result.push(top);
            }
        }

        if(result.isEmpty()) return "/";

        StringBuilder sb = new StringBuilder();
        while(!result.isEmpty()){
            String s  = result.pop();
            sb.append(s);
        }
        return sb.toString();
    }
}
```

1.18 Valid Palindrome (Medium -> Leetcode No.125)

Solution: See CleanCodeHandbook-1 Question 4

1.19 Reverse Words in String (Medium -> Leetcode No.151)

Solution: See CleanCodeHandbook-1 Question 6 and 7

1.20 Reverse String (Easy -> Leetcode No.344)

Write a function that takes a string as input and returns the string reversed.

Example:

Given s = "hello", return "olleh".

Subscribe to see which companies asked this question

Hide Tags | Two Pointers | String

Hide Similar Problems | (E) Reverse Vowels of a String

Solution:

```
public class Solution {
   public String reverseString(String s) {
      if(s==null || s.length()==0) return s;
      StringBuilder sb = new StringBuilder(s);
      return sb.reverse().toString();
   }
}
```

In summary, there are total 5 ways to reverse a string.

1)

1. The user will input the string to be reversed.
2. First we will convert String to character array by using the built in java String class method toCharArray().
3. Then , we will scan the string from end to start, and print the character one by one.

2)

1. In the second method , we will use the built in reverse() method of the StringBuilder class ,.

Note : String class does not have reverse() method . So we need to convert the input string to StringBuilder , which is achieved by using the append method of the StringBuilder.

2. After that print out the characters of the reversed string by scanning from the first till the last index.

3)

1. Convert the input string into character array by using the toCharArray() built in method of the String Class .
2. In this method we will scan the character array from both sides , that is from the start index (left) as well as from last index(right) simultaneously.
3. Set the left index equal to 0 and right index equal to the length of the string -1.
4. Swap the characters of the start index scanning with the last index scanning one by one .After that increase the left index by 1 (left++) and decrease the right by 1 i.e (right--) to move on to the next characters in the character array .
5. Continue till left is less than or equal to the right .

4)

1. Convert the input string into the character array by using toCharArray() built in method.
2. Then add the characters of the array into the LinkedList object . We used LinkedList because it maintains the insertion order of the input values.
3. Java also has built in reverse() method for the Collections class . Since Collections class reverse() method takes a list object , to reverse the list , we will pass the LinkedList object which is a type of list of

characters.
4. We will create the ListIterator object by using the listIterator() method on the LinkedList object. ListIterator object is used to iterate over the list.
5. ListIterator object will help us to iterate over the reversed list and print it one by one to the output screen.
5)
1. The last method is converting string into bytes . getBytes() method is used to convert the input string into bytes[].
2. Then we will create a temporary byte[] of length equal to the length of the input string.
3. We will store the bytes(which we get by using getBytes() method) in reverse order into the temporary byte[] .

1.21 Reverse Vowels of a String (Easy -> Leetcode No.345)

Write a function that takes a string as input and reverse only the vowels of a string.

Example 1:

Given s = "hello", return "holle".

Example 2:

Given s = "leetcode", return "leotcede".

Subscribe to see which companies asked this question

Hide Tags | Two Pointers | String

Hide Similar Problems | (E) Reverse String

Solution:

```
public class Solution {
   public String reverseVowels(String s) {
      ArrayList<Character> vowList = new ArrayList<>();
      vowList.add('a');
      vowList.add('e');
      vowList.add('i');
      vowList.add('o');
      vowList.add('u');
      vowList.add('A');
      vowList.add('E');
      vowList.add('I');
      vowList.add('O');
      vowList.add('U');

      char[] carr = s.toCharArray();
      int i=0;
      int j = carr.length-1;
      while(i<j){
```

```
            if(!vowList.contains(carr[i])){
                i++;
                continue;
            }
            if(!vowList.contains(carr[j])){
                j--;
                continue;
            }

            char tmp = carr[i];
            carr[i] = carr[j];
            carr[j] = tmp;

            i++;
            j--;
        }
        return new String(carr);
    }
}
```

CHAPTER 2: Linked List

2.1 Add Two Number (Medium->Leetcode No.2)

You are given two linked lists representing two non-negative numbers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)

Output: 7 -> 0 -> 8

Subscribe to see which companies asked this question

Hide Tags | Linked List | Math

Hide Similar Problems | (M) Multiply Strings | (E) Add Binary

Solution:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        int carry= 0;
        ListNode head = new ListNode(0);
        ListNode p1 = l1, p2 = l2, p = head;
        while(p1!=null || p2!=null){
            if(p1!=null){
                carry += p1.val;
                p1 = p1.next;
            }
            if(p2!=null){
                carry += p2.val;
                p2 = p2.next;
            }
            p.next = new ListNode(carry%10);
            p = p.next;
            carry = carry/10;
        }
        if(carry==1) p.next = new ListNode(carry);
        return head.next;
    }
}
```

2.2 Remove Nth Node From End of List (Easy->Leetcode No.19)

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

For example,

```
Given linked list: 1->2->3->4->5, and n = 2.

After removing the second node from the end, the linked list becomes 1->2->3->5.
```

Note:

Given *n* will always be valid.

Try to do this in one pass.

Subscribe to see which companies asked this question

Hide Tags | Linked List | Two Pointers

Solution:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        if(head==null) return null;
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode fast = dummy;
        ListNode slow = dummy;

        while(fast.next!=null){
            if(n<=0) slow = slow.next;
            fast = fast.next;
            n--;
        }
        if(slow.next!=null){
            slow.next = slow.next.next;
        }
        return dummy.next;
    }
}
```

2.3 Merge Two Sorted Lists (Easy->Leetcode No.21)

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

Subscribe to see which companies asked this question

Hide Tags | Linked List

Hide Similar Problems | (H) Merge k Sorted Lists | (E) Merge Sorted Array | (M) Sort List | (M) Shortest Word Distance II

Solution:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(0);
        ListNode head = dummy;
        while(l1!=null || l2!=null){
            if(l1!=null && l2!=null){
                if(l1.val<l2.val){
                    head.next = l1;
                    l1 = l1.next;
                }else{
                    head.next = l2;
                    l2 = l2.next;
                }
                head = head.next;
            }else if(l1==null){
                head.next = l2;
                break;
            }else if (l2==null){
                head.next = l1;
                break;
            }
        }
        return dummy.next;
    }
}
```

2.4 Merge K Sorted Lists (Hard -> Leetcode No.23)

Total Accepted: 86932 Total Submissions: 368212 Difficulty: Hard

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

Subscribe to see which companies asked this question

Show Tags

Show Similar Problems

Solution:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        if(lists==null || lists.length==0) return null;
        PriorityQueue<ListNode> queue = new PriorityQueue<>(new Comparator<ListNode>(){
            public int compare(ListNode l1, ListNode l2){
                return l1.val-l2.val;
            }
        });
        ListNode dummy = new ListNode(0);
        ListNode p = dummy;
        for(ListNode l:lists){
            if(l!=null) queue.offer(l);
        }
        while(!queue.isEmpty()){
            ListNode tmp = queue.poll();
            p.next = tmp;
            p = p.next;
            if(tmp.next!=null){
                queue.offer(tmp.next);
            }
        }
        return dummy.next;
    }
}
```

2.5 Swap Nodes in Pairs (Easy->Leetcode No.24)

Given a linked list, swap every two adjacent nodes and return its head.

For example,

Given `1->2->3->4`, you should return the list as `2->1->4->3`.

Your algorithm should use only constant space. You may **not** modify the values in the list, only nodes itself can be changed.

Subscribe to see which companies asked this question

Hide Tags Linked List

Hide Similar Problems (H) Reverse Nodes in k-Group

Solution:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode swapPairs(ListNode head) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode p = dummy;
        while(p.next!=null && p.next.next!=null){
            ListNode t1 = p;
            p = p.next;
            t1.next = p.next;

            ListNode t2 = p.next.next;
            p.next.next = p;
            p.next = t2;
        }
        return dummy.next;
    }
}
```

2.6 Rotate List (Medium->Leetcode No.61)

Given a list, rotate the list to the right by *k* places, where *k* is non-negative.

For example:

Given `1->2->3->4->5->NULL` and *k* = `2`,

return `4->5->1->2->3->NULL`.

Subscribe to see which companies asked this question

Hide Tags | Linked List | Two Pointers

Hide Similar Problems | (E) Rotate Array

Solution

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode rotateRight(ListNode head, int k) {
        if(head==null || k<=0) return head;
        ListNode p = head;
        int len = 1;
        while(p.next!=null){
            p = p.next;
            len++;
        }
        p.next = head;
        k%=len;
        for(int i=0;i<len-k;i++) p = p.next;
        head = p.next;
        p.next = null;
        return head;
    }
}
```

2.7 Delete Node in a Linked List (Easy->LeetCode No.237)

Write a function to delete a node (except the tail) in a singly linked list, given only access to that node.

Supposed the linked list is `1 -> 2 -> 3 -> 4` and you are given the third node with value `3`, the linked list should become `1 -> 2 -> 4` after calling your function.

Subscribe to see which companies asked this question

Hide Tags Linked List

Hide Similar Problems (E) Remove Linked List Elements

Solution

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public void deleteNode(ListNode node) {
        node.val = node.next.val;
        node.next = node.next.next;
    }
}
```

2.8 Insertion Sort List (Medium->LeetCode No.147)

Sort a linked list using insertion sort.

Subscribe to see which companies asked this question

Hide Tags Linked List Sort

Hide Similar Problems (M) Sort List

Solution

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
```

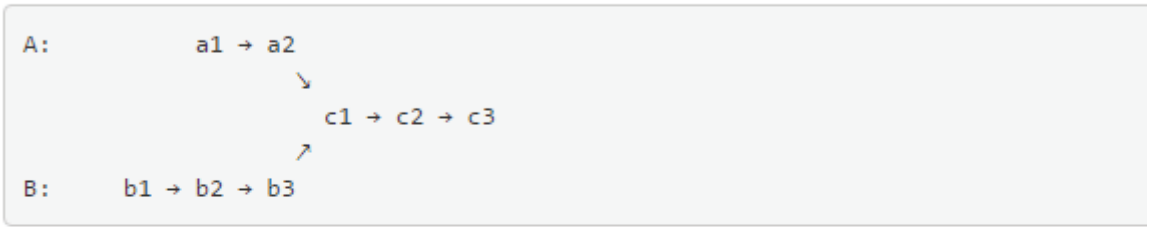
```
 */
public class Solution {
   public ListNode insertionSortList(ListNode head) {
   if (head == null || head.next == null)
   {
      return head;
   }

   ListNode sortedHead = head, sortedTail = head;
   head = head.next;
   sortedHead.next = null;

   while (head != null)
   {
      ListNode temp = head;
      head = head.next;
      temp.next = null;

      // new val is less than the head, just insert in the front
      if (temp.val <= sortedHead.val)
      {
         temp.next = sortedHead;
         sortedTail = sortedHead.next == null ? sortedHead : sortedTail;
         sortedHead = temp;
      }
      // new val is greater than the tail, just insert at the back
      else if (temp.val >= sortedTail.val)
      {
         sortedTail.next = temp;
         sortedTail = sortedTail.next;
      }
      // new val is somewhere in the middle, we will have to find its proper
      // location.
      else
      {
         ListNode current = sortedHead;
         while (current.next != null && current.next.val < temp.val)
         {
            current = current.next;
         }

         temp.next = current.next;
         current.next = temp;
      }
   }

   return sortedHead;
}
```

}

2.9 Intersection of Two Linked Lists (Easy->LeetCode No.160)

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

For example, the following two linked lists:



begin to intersect at node c1.

Notes:

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

Solution:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        if(headA==null || headB==null) return null;
        int lenA = getLength(headA);
        int lenB = getLength(headB);
        ListNode pA = headA;
        ListNode pB = headB;
```

```
        int diff = Math.abs(lenA-lenB);
        int i = 0;
        if(lenA>lenB){
            while(i<diff){
                pA = pA.next;
                i++;
            }
        }else {
            while(i<diff){
                pB = pB.next;
                i++;
            }
        }
        while(pA!=null && pB!=null){
            if(pA.val == pB.val) return pA;
            pA = pA.next;
            pB = pB.next;
        }
        return null;
    }
    public int getLength(ListNode head){
        ListNode p = head;
        int len = 1;
        while(p.next!=null){
            p = p.next;
            len++;
        }
        return len;
    }
}
```

2.10 Linked List Cycle I (Medium->LeetCode No. 141)

Given a linked list, determine if it has a cycle in it.

Follow up:

Can you solve it without using extra space?

Subscribe to see which companies asked this question

Hide Tags | Linked List | Two Pointers

Hide Similar Problems | (M) Linked List Cycle II

Solution:

```
/**
 * Definition for singly-linked list.
 * class ListNode {
```

```
*     int val;
*     ListNode next;
*     ListNode(int x) {
*         val = x;
*         next = null;
*     }
* }
*/
public class Solution {
    public boolean hasCycle(ListNode head) {
        ListNode slow = head;
        ListNode fast = head;
        while(fast!=null && fast.next!=null){
            slow = slow.next;
            fast = fast.next.next;
            if(slow==fast) return true;
        }
        return false;
    }
}
```

2.11 Linked List Cycle II (Medium-> LeetCode No.142)

Given a linked list, return the node where the cycle begins. If there is no cycle, return `null`.

Note: Do not modify the linked list.

Follow up:

Can you solve it without using extra space?

Subscribe to see which companies asked this question

Hide Tags | Linked List | Two Pointers

Hide Similar Problems | (E) Linked List Cycle | (H) Find the Duplicate Number

Solution

Define two pointers slow and fast, both start at head node, fast is twoce as fast as slow, if it reaches the end it means thre is no cycles, otherwise eventually it will eventually catch up to slow pointer somewhere in the cycle. Let the distance from the first node to the node where the cycle begins be A, and let say the slow pointer travels A+B. The fast pointer must travels 2A+2B to catch up. The cycle size is N. Full cycle is also how much more fast pointer has traverled than slow pointer at meeting point. Since A+B+N = 2A +2B, we have N=A+B.

From our calculatin, slow pointer traveled exactly full cycle when it meets fast pointer, and since originally it traveled A before starting on a cycle, it must travel A to reach the point where cycle begins! We can start another slow pointer at head node, and move both pointer until they meet at the beginning of a cycle.

```
/**
```

```
 * Definition for singly-linked list.
 * class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public ListNode detectCycle(ListNode head) {
        ListNode slow = head;
        ListNode fast = head;
        while(fast!=null && fast.next!=null){
            fast = fast.next.next;
            slow = slow.next;

            if(slow==fast){
                ListNode slowslow = head;
                while(slowslow!=slow){
                    slowslow = slowslow.next;
                    slow = slow.next;
                }
                return slow;
            }
        }
        return null;
    }
}
```

2.12 Remove Duplicate from Sorted List I (Medium->LeetCode No.83)

Given a sorted linked list, delete all duplicates such that each element appear only *once*.

For example,

Given `1->1->2`, return `1->2`.

Given `1->1->2->3->3`, return `1->2->3`.

Subscribe to see which companies asked this question

Hide Tags Linked List

Solution:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
```

```
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        if(head==null || head.next==null) return head;
        ListNode pre = head;
        ListNode p = head.next;
        while(p != null){
            if(pre.val==p.val){
                pre.next = p.next;
                p = p.next;
            }else{
                pre = p;
                p = p.next;
            }
        }
        return head;
    }
}
```

2.13 Remove Duplicate from Sorted List II (Medium-> LeetCode No.82)

Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only *distinct* numbers from the original list.

For example,

Given `1->2->3->3->4->4->5`, return `1->2->5`.

Given `1->1->1->2->3`, return `2->3`.

Solution

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        if(head==null || head.next==null) return head;
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode p = dummy;

        while(p.next!=null && p.next.next!=null){
            if(p.next.val==p.next.next.val){
                int dup = p.next.val;
```

```
            while(p.next!=null && p.next.val==dup) p.next = p.next.next;
        }else{
            p = p.next;
        }
    }
    return dummy.next;
  }
}
```

2.14 Partition List (Medium->Leetcode No.86)

Given a linked list and a value *x*, partition it such that all nodes less than *x* come before nodes greater than or equal to *x*.

You should preserve the original relative order of the nodes in each of the two partitions.

For example,

Given `1->4->3->2->5->2` and *x* = 3,

return `1->2->2->4->3->5`.

Subscribe to see which companies asked this question

Show Tags

Solution:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode partition(ListNode head, int x) {
        if(head==null) return null;
        ListNode dummy1 = new ListNode(0);
        ListNode dummy2 = new ListNode(0);
        dummy1.next = head;
        ListNode p = head;
        ListNode pre = dummy1;
        ListNode pp = dummy2;

        while(p!=null){
            if(p.val<x){
                p = p.next;
                pre = pre.next;
            }else{
                pp.next = p;
                pre.next = p.next;
                p = pre.next;
```

```
            pp = pp.next;
          }
        }
        pre.next = dummy2.next;
        pp.next = null;
        return dummy1.next;
    }
}
```

2.15 Reverse Linked List II (Medium->Leetcode No.92)

Reverse a linked list from position *m* to *n*. Do it in-place and in one-pass.

For example:

Given `1->2->3->4->5->NULL`, *m* = 2 and *n* = 4,

return `1->4->3->2->5->NULL`.

Note:

Given *m*, *n* satisfy the following condition:

1 ≤ *m* ≤ *n* ≤ length of list.

Solution:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode reverseBetween(ListNode head, int m, int n) {
        if(head==null || head.next==null || m==n) return head;
        ListNode dummy = new ListNode(0);
        ListNode pre = dummy;
        dummy.next = head;
        for(int i=0;i<m-1;i++){
            pre = pre.next;
        }
        ListNode cur = pre.next, p = pre.next, node = null;
        for(int i=0;i<=n-m;i++){
            ListNode next = cur.next;
            cur.next = node;
            node = cur;
            cur = next;
        }
```

```
        p.next = cur;
        pre.next = node;
        return dummy.next;
    }
}
```

2.16 Convert Sorted List to Binary Search Tree (Medium->Leetcode No.109)

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

Subscribe to see which companies asked this question

Hide Tags | Depth-first Search | Linked List

Hide Similar Problems | (M) Convert Sorted Array to Binary Search Tree

Solution:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    static ListNode h;
    public TreeNode sortedListToBST(ListNode head) {
        if(head==null) return null;
        int length = getLength(head);
        h=head;
        return sortedListToBST(0,length);
    }

    public TreeNode sortedListToBST(int start, int end){
        if(start>end) return null;
        int mid = start+(end-start)/2;
        TreeNode left =  sortedListToBST(start, mid-1);
        TreeNode root = new TreeNode(h.val);
        h = h.next;
```

```
        TreeNode right = sortedListToBST(mid+1,end);
        root.left = left;
        root.right = right;
        return root;
    }
    public int getLength(ListNode head){
        int len = 0;
        ListNode p = head;
        while(p.next!=null){
            len++;
            p = p.next;
        }
        return len;
    }
}
```

2.17 Reorder List (Medium->Leetcode No.143)

Given a singly linked list L: $L_0 \rightarrow L_1 \rightarrow \ldots \rightarrow L_{n-1} \rightarrow L_n$,

reorder it to: $L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \rightarrow \ldots$

You must do this in-place without altering the nodes' values.

For example,

Given `{1,2,3,4}`, reorder it to `{1,4,2,3}`.

Solution:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public void reorderList(ListNode head) {
        if(head==null) return;
        ListNode slow = head;
        ListNode fast = head;
        while(fast.next!=null && fast.next.next!=null){
            fast = fast.next.next;
            slow = slow.next;
        }
        fast = slow.next;
        slow.next = null;
        slow = head;
        // now slow is the head of the first part, and then the fast is the second part
```

```
        // reverse the second part
        ListNode prev = null;
        while(fast!=null){
            ListNode tmp = fast.next;
            fast.next = prev;
            prev = fast;
            fast = tmp;
        }
        // pre is the new head of the second part now
        // merge the two parts
        ListNode cur = slow;
        while(slow!=null && prev!=null){
            ListNode tmp1 = slow.next;
            ListNode tmp2 = prev.next;
            cur.next = prev;
            cur = cur.next;
            cur.next = tmp1;
            cur = cur.next;
            slow = tmp1;
            prev = tmp2;
        }
    }
}
```

2.18 Sort List(Medium->Leetcode No.148)

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

Subscribe to see which companies asked this question

Show Tags

Show Similar Problems

Solution

QuickSort:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode sortList(ListNode head) {
        if(head==null || head.next==null) return head;
        ListNode pivot = head;
        ListNode pivotHead = pivot;
```

```
        ListNode left = new ListNode(0);;
        ListNode leftHead = left;
        ListNode right = new ListNode(0);
        ListNode rightHead = right;
        ListNode crt = head.next;
        if(crt==null) return head;
        while(crt!=null){
           if(crt.val<pivot.val){
              left.next = crt;
              left = left.next;
           }else if(crt.val>pivot.val){
              right.next = crt;
              right = right.next;
           }else{
              pivot.next = crt;
              pivot = pivot.next;
           }
           crt = crt.next;
        }
        left.next = null;
        right.next = null;
        pivot.next = null;
        left = sortList(leftHead.next);
        right = sortList(rightHead.next);
        pivot.next = right;
        ListNode re = left;

        while(left!=null && left.next!=null) left = left.next;
        if(left==null) re = pivotHead;
        else left.next = pivotHead;
        return re;
    }
}
```

Merge Sort

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode sortList(ListNode head) {
        if(head==null || head.next==null) return head;
        ListNode prev = head;
        ListNode slow = head;
```

```
        ListNode fast = head;

        while(fast !=null && fast.next!=null){
            prev = slow;
            slow = slow.next;
            fast = fast.next.next;
        }
        prev.next = null;
        ListNode firstHalf = sortList(head);
        ListNode secondHalf = sortList(slow);
        return merge(firstHalf, secondHalf);
    }

    public ListNode merge(ListNode firstHalf, ListNode secondHalf){
        ListNode dummy = new ListNode(0);
        ListNode crt = dummy;
        while(firstHalf!=null && secondHalf!=null){
            if(firstHalf.val<secondHalf.val){
                crt.next = firstHalf;
                firstHalf = firstHalf.next;
            }else {
                crt.next = secondHalf;
                secondHalf = secondHalf.next;
            }
            crt = crt.next;
        }
        if(firstHalf!=null) crt.next = firstHalf;
        else crt.next = secondHalf;
        return dummy.next;
    }
}
```

2.19 Remove Linked List Elements (Easy->Leetcode No.203)

Remove all elements from a linked list of integers that have value ***val***.

Example

Given: 1 --> 2 --> 6 --> 3 --> 4 --> 5 --> 6, ***val*** = 6

Return: 1 --> 2 --> 3 --> 4 --> 5

Credits:

Special thanks to @mithmatt for adding this problem and creating all test cases.

Solution:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
```

```
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode removeElements(ListNode head, int val) {
        if(head==null) return head;
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode prev = dummy;
        ListNode curr = head;

        while(curr!=null){
            if(curr.val==val) prev.next = curr.next;
            else prev = curr;
            curr = curr.next;
        }
        return dummy.next;
    }
}
```

2.20 Reverse Linked List (Easy->leetcode No.206)

Reverse a singly linked list.

click to show more hints.

Subscribe to see which companies asked this question

Hide Tags | Linked List

Hide Similar Problems | (M) Reverse Linked List II | (M) Binary Tree Upside Down | (E) Palindrome Linked List

Solution:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode reverseList(ListNode head) {
        if(head==null) return head;
        ListNode prev = null;
        ListNode curr = head;
        while(curr!=null){
```

```
            ListNode next = curr.next;
            curr.next = prev;
            prev = curr;
            curr = next;
        }
        return prev;
    }
}
```

2.21 Palindrome Linked List <Easy->Leetcode NO.234>

Given a singly linked list, determine if it is a palindrome.

Follow up:

Could you do it in O(n) time and O(1) space?

Subscribe to see which companies asked this question

Hide Tags | Linked List | Two Pointers

Hide Similar Problems | (E) Palindrome Number | (E) Valid Palindrome | (E) Reverse Linked List

Solution:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public boolean isPalindrome(ListNode head) {
        if(head==null || head.next==null) return true;
        ListNode slow = head;
        ListNode fast = head;
        while(fast.next!=null && fast.next.next!=null){
            slow = slow.next;
            fast = fast.next.next;
        }
        ListNode secondHalf = slow.next;
        slow.next = null;

        // reverse the second half
        ListNode p1 = secondHalf;
        ListNode p2 = p1.next;
        while(p1!=null && p2!=null){
            ListNode tmp = p2.next;
```

```
            p2.next = p1;
            p1 = p2;
            p2 = tmp;
        }
        secondHalf.next = null;

        //compare the two sublists now
        ListNode p = p2==null?p1:p2;
        ListNode q = head;
        while(p!=null){
            if(p.val!=q.val) return false;
            p = p.next;
            q = q.next;
        }
        return true;
    }
}
```

2.22 Odd Even Linked List (Medium->Leetcode No.328)

Given a singly linked list, group all odd nodes together followed by the even nodes. Please note here we are talking about the node number and not the value in the nodes.

You should try to do it in place. The program should run in O(1) space complexity and O(nodes) time complexity.

Example:

Given `1->2->3->4->5->NULL`,

return `1->3->5->2->4->NULL`.

Note:

The relative order inside both the even and odd groups should remain as it was in the input.

The first node is considered odd, the second node even and so on ...

Solution:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode oddEvenList(ListNode head) {
        if(head==null) return head;
        ListNode p1 = head;
        ListNode p2 = head.next;
        ListNode connect = head.next;

        while(p1!=null && p2!=null){
            ListNode tmp = p2.next;
            if(tmp==null) break;
            p1.next = p2.next;
```

```
            p1 = p1.next;

            p2.next = p1.next;
            p2 = p2.next;
        }
        p1.next = connect;
        return head;
    }
}
```

CHAPTER 3: Hash Table

3.1. 4Sum (Medium->LeetCode No.18)

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

Note:

- Elements in a quadruplet (*a*,*b*,*c*,*d*) must be in non-descending order. (ie, *a* ≤ *b* ≤ *c* ≤ *d*)
- The solution set must not contain duplicate quadruplets.

```
For example, given array S = {1 0 -1 0 -2 2}, and target = 0.

A solution set is:
(-1,  0, 0, 1)
(-2, -1, 1, 2)
(-2,  0, 0, 2)
```

Solution:

```
public class Solution {
   public List<List<Integer>> fourSum(int[] nums, int target) {
      Arrays.sort(nums);
      HashSet<List<Integer>> set = new HashSet<>();
      List<List<Integer>> result = new ArrayList<List<Integer>>();
      for(int i=0;i<nums.length;i++){
         for(int j=i+1;j<nums.length;j++){
            int k=j+1;
            int p = nums.length-1;
            while(k<p){
               int sum = nums[i]+nums[j]+nums[k]+nums[p];
               if(sum<target) k++;
               else if(sum>target) p--;
               else{
                  List<Integer> tmp = new ArrayList<>();
                  tmp.add(nums[i]);
                  tmp.add(nums[j]);
                  tmp.add(nums[k]);
                  tmp.add(nums[p]);
                  if(!set.contains(tmp)){
                     set.add(tmp);
                     result.add(tmp);
                  }
                  k++;
                  p--;
               }
            }
         }
      }
      return result;
   }
}
```

3.2. Group Anagrams (Medium-> LeetCode No.49)

Given an array of strings, group anagrams together.

For example, given: `["eat", "tea", "tan", "ate", "nat", "bat"]`,

Return:

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

Note:

1. For the return value, each *inner* list's elements must follow the lexicographic order.
2. All inputs will be in lower-case.

Solution:

```
public class Solution {
   public List<List<String>> groupAnagrams(String[] strs) {
      List<List<String>> results = new ArrayList<List<String>>();
      if(strs==null || strs.length==0) return results;
      List<String> result = new ArrayList<>();
      HashMap<String, List<String>> map = new HashMap<>();
      for(String str:strs){
         char[] arr = str.toCharArray();
         Arrays.sort(arr);
         String tmp = new String(arr);
         if(map.containsKey(tmp)){
            map.get(tmp).add(str);
         }else{
            List<String> tmpList = new ArrayList<>();
            tmpList.add(str);
            map.put(tmp, tmpList);
         }
      }

      for(Map.Entry<String, List<String>> entry:map.entrySet()){
         Collections.sort(entry.getValue());
      }
      results.addAll(map.values());
      return results;
   }
}
```

3.3. Binary Tree Inorder Traversal (Medium->LeetCode No.94)

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

For example:

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

```
   1
    \
     2
    /
   3
```

return `[1,3,2]`.

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

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

Solution:

```
/**
 * 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> inorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        if(root==null) return result;
        dfs(result, root);
        return result;
    }

    public void dfs(List<Integer> result, TreeNode root){
        if(root.left!=null) dfs(result, root.left);
        result.add(root.val);
        if(root.right!=null) dfs(result, root.right);
    }
}
```

3.4. Bulls and Cows (Easy->LeetCode No.299)

You are playing the following Bulls and Cows game with your friend: You write down a number and ask your friend to guess what the number is. Each time your friend makes a guess, you provide a hint that indicates how many digits in said guess match your secret number exactly in both digit and position (called "bulls") and how many digits match the secret number but locate in the wrong position (called "cows"). Your friend will use successive guesses and hints to eventually derive the secret number.

For example:

```
Secret number:  "1807"
Friend's guess: "7810"
```

Hint: `1` bull and `3` cows. (The bull is `8`, the cows are `0`, `1` and `7`.)

Write a function to return a hint according to the secret number and friend's guess, use `A` to indicate the bulls and `B` to indicate the cows. In the above example, your function should return `"1A3B"`.

Please note that both secret number and friend's guess may contain duplicate digits, for example:

```
Secret number:  "1123"
Friend's guess: "0111"
```

In this case, the 1st `1` in friend's guess is a bull, the 2nd or 3rd `1` is a cow, and your function should return `"1A1B"`.

You may assume that the secret number and your friend's guess only contain digits, and their lengths are always equal.

Solution:

```
public class Solution {
   public String getHint(String secret, String guess) {
      int bull = 0;
      int cow = 0;
      int[] num = new int[10];
      for(int i=0;i<secret.length();i++){
         if(secret.charAt(i)==guess.charAt(i)) bull++;
         else{
            if(num[secret.charAt(i)-'0']++ < 0) cow++;
            if(num[guess.charAt(i)-'0']-- > 0) cow++;
         }
      }
      return bull+"A"+cow+"B";
   }
}
```

3.5. Contains Duplicate I (Easy->LeetCode No.217)

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

Subscribe to see which companies asked this question

Show Tags

Show Similar Problems

Solution:

```
public class Solution {
```

```
    public boolean containsDuplicate(int[] nums) {
        if(nums==null || nums.length==0) return false;
        HashSet<Integer> set = new HashSet<Integer>();
        for(int i: nums){
            if(!set.add(i)){
                return true;
            }
        }
    return false;
    }
}
```

3.6. Contains Duplicate II (Easy->LeetCode No.219)

Given an array of integers and an integer *k*, find out whether there are two distinct indices *i* and *j* in the array such that **nums[i] = nums[j]** and the difference between *i* and *j* is at most *k*.

Subscribe to see which companies asked this question

Show Tags

Show Similar Problems

Solution:

```
public class Solution {
    public boolean containsNearbyDuplicate(int[] nums, int k) {
        HashMap<Integer, Integer> map = new HashMap<>();
        for(int i=0;i<nums.length;i++){
            if(map.containsKey(nums[i])){
                int index = map.get(nums[i]);
                if(i-index<=k) return true;
            }
            map.put(nums[i],i);
        }
        return false;
    }
}
```

3.7. Count Prims (Easy-> LeetCode No.204)

Description:

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

Solution:

```
public class Solution {
    public int countPrimes(int n) {
        if(n<=2) return 0;
        boolean[] prime = new boolean[n];
        for(int i=2;i<n;i++) prime[i]=true;
        for(int i=2;i<=Math.sqrt(n);i++){
            if(prime[i]){
                for(int j=i+i;j<n;j+=i){
```

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

3.8. Fraction to Recurring Decimal (Medium-> LeetCode No.166)

Given two integers representing the numerator and denominator of a fraction, return the fraction in string format.

If the fractional part is repeating, enclose the repeating part in parentheses.

For example,

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

Hint:

1. No scary math, just apply elementary math knowledge. Still remember how to perform a *long division*?
2. Try a long division on 4/9, the repeating part is obvious. Now try 4/333. Do you see a pattern?
3. Be wary of edge cases! List out as many test cases as you can think of and test your code thoroughly.

Solution:

```
public class Solution {
    public String fractionToDecimal(int numerator, int denominator) {
        if(numerator==0) return "0";
        if(denominator==0) return "";
        String result = "";
        // decide the sign
        if((numerator<0)^(denominator<0)) result+="-";
        // convert int to long
        long num = numerator, den = denominator;
        num = Math.abs(num);
        den = Math.abs(den);

        // quotient
        long res = num/den;
        result +=String.valueOf(res);

        //if remainder equals to zero, return result
```

```
        long remainder = (num%den)*10;
        if(remainder==0) return result;

        // process the right side of the decimal point
        HashMap<Long, Integer> map = new HashMap<>();
        result+=".";
        while(remainder!=0){
            if(map.containsKey(remainder)){
                int beg = map.get(remainder);
                String part1 = result.substring(0,beg);
                String part2 = result.substring(beg);
                result = part1+"("+part2+")";
                return result;
            }

            map.put(remainder, result.length());
            res = remainder/den;
            result+=String.valueOf(res);
            remainder = (remainder%den)*10;
        }
        return result;
    }
}
```

3.9. Two Sum (Easy-> LeetCode No.1)

Given an array of integers, return **indices** of the two numbers such that they add up to a specific target.

You may assume that each input would have ***exactly*** one solution.

Example:

```
Given nums = [2, 7, 11, 15], target = 9,

Because nums[0] + nums[1] = 2 + 7 = 9,
return [0, 1].
```

UPDATE (2016/2/13):

The return format had been changed to **zero-based** indices. Please read the above updated description carefully.

Subscribe to see which companies asked this question

Solution:

```
public class Solution {
    public int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> map = new HashMap<>();
        int[] result = new int[2];
        for(int i=0;i<nums.length;i++){
            if(map.containsKey(nums[i])){
                int index = map.get(nums[i]);
                result[0] = index;
                result[1] = i;
```

```
        }else{
            map.put(target-nums[i],i);
        }
      }
      return result;
    }
}
```

3.10. Longest Substring Without Repeating Characters (Medium-> LeetCode No.3)

Given a string, find the length of the **longest substring** without repeating characters.

Examples:

Given `"abcabcbb"`, the answer is `"abc"`, which the length is 3.

Given `"bbbbb"`, the answer is `"b"`, with the length of 1.

Given `"pwwkew"`, the answer is `"wke"`, with the length of 3. Note that the answer must be a **substring**, `"pwke"` is a *subsequence* and not a substring.

Subscribe to see which companies asked this question

Hide Tags | Hash Table | Two Pointers | String

Hide Similar Problems | (H) Longest Substring with At Most Two Distinct Characters

Solution:

```
public class Solution {
    public int lengthOfLongestSubstring(String s) {
        if(s==null || s.length()==0) return 0;
        Map<Character, Integer> map = new HashMap<>();
        int max = 0;
        for(int i=0,j=0;i<s.length();i++){
            if(map.containsKey(s.charAt(i))){
                j=Math.max(j, map.get(s.charAt(i))+1);
            }
            map.put(s.charAt(i),i);
            max = Math.max(max, i-j+1);
        }
        return max;
    }
}
```

3.11. Valid Sudoku (Easy-> LeetCode No.36)

Determine if a Sudoku is valid, according to: Sudoku Puzzles - The Rules.

The Sudoku board could be partially filled, where empty cells are filled with the character `'.'`.

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

A partially filled sudoku which is valid.

Note:

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

Solution:

```
public class Solution {
   public boolean isValidSudoku(char[][] board) {
        if (board == null || board.length != 9 || board[0].length != 9)
            return false;
        // check each column
        for (int i = 0; i < 9; i++) {
            boolean[] m = new boolean[9];
            for (int j = 0; j < 9; j++) {
                if (board[i][j] != '.') {
                    if (m[(int) (board[i][j] - '1')]) {
                        return false;
                    }
                    m[(int) (board[i][j] - '1')] = true;
                }
            }
        }

        //check each row
        for (int j = 0; j < 9; j++) {
            boolean[] m = new boolean[9];
            for (int i = 0; i < 9; i++) {
                if (board[i][j] != '.') {
                    if (m[(int) (board[i][j] - '1')]) {
                        return false;
```

```
                                        }
                                        m[(int) (board[i][j] - '1')] = true;
                                }
                        }
                }

                //check each 3*3 matrix
                for (int block = 0; block < 9; block++) {
                        boolean[] m = new boolean[9];
                        for (int i = block / 3 * 3; i < block / 3 * 3 + 3; i++) {
                                for (int j = block % 3 * 3; j < block % 3 * 3 + 3; j++) {
                                        if (board[i][j] != '.') {
                                                if (m[(int) (board[i][j] - '1')]) {
                                                        return false;
                                                }
                                                m[(int) (board[i][j] - '1')] = true;
                                        }
                                }
                        }
                }

                return true;
        }
}
```

3.12. Single Number (Medium-> LeetCode No.136)

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

Note:

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

Subscribe to see which companies asked this question

Hide Tags | Hash Table | Bit Manipulation

Hide Similar Problems | (M) Single Number II | (M) Single Number III | (M) Missing Number | (H) Find the Duplicate Number

Solution:

```
public class Solution {
   public int singleNumber(int[] nums) {
      if(nums==null || nums.length==0) return 0;
      int sum = 0;
      for(int i=0;i<nums.length;i++) sum=sum^nums[i];
      return sum;
   }
}
```

3.13. Repeated DNA Sequences (Medium-> LeetCode No.187)

All DNA is composed of a series of nucleotides abbreviated as A, C, G, and T, for example: "ACGAATTCCG". When studying DNA, it is sometimes useful to identify repeated sequences within the DNA.

Write a function to find all the 10-letter-long sequences (substrings) that occur more than once in a DNA molecule.

For example,

```
Given s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT",

Return:
["AAAAACCCCC", "CCCCCAAAAA"].
```

Solution:

```
public class Solution {
   public List<String> findRepeatedDnaSequences(String s) {
      List<String> result = new ArrayList<>();
      if(s==null || s.length()<=10) return result;
      Set<String> resultset = new HashSet<>();
      Set<String> set = new HashSet<>();
      int len = s.length();
      for(int i=0;i<=len-10;i++){
         String tmp = s.substring(i,i+10);
         if(!set.add(tmp)) resultset.add(tmp);
      }
      result.addAll(resultset);
      return result;
   }
}
```

3.14. Happy Number (Easy-> LeetCode No.202)

Write an algorithm to determine if a number is "happy".

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

Example: 19 is a happy number

$1^2 + 9^2 = 82$

$8^2 + 2^2 = 68$

$6^2 + 8^2 = 100$

$1^2 + 0^2 + 0^2 = 1$

Solution:

```
public class Solution {
   public boolean isHappy(int n) {
      HashSet<Integer> set = new HashSet<>();
      while(!set.contains(n)){
         set.add(n);
         n = sum(getDigits(n));
         if(n==1) return true;
```

```
        }
        return false;
    }
    public int sum(int[] digits){
        int sum=0;
        for(int i:digits) sum+=i*i;
        return sum;
    }

    public int[] getDigits(int n){
        String s = String.valueOf(n);
        int[] digits = new int[s.length()];
        int i=0;
        while(n>0){
            int m = n%10;
            digits[i++]=m;
            n=n/10;
        }
        return digits;
    }
}
```

3.15. Isomorphic Strings (Easy-> LeetCode No.205)

Given two strings ***s*** and ***t***, determine if they are isomorphic.

Two strings are isomorphic if the characters in ***s*** can be replaced to get ***t***.

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character but a character may map to itself.

For example,

Given `"egg"`, `"add"`, return true.

Given `"foo"`, `"bar"`, return false.

Given `"paper"`, `"title"`, return true.

Note:

You may assume both ***s*** and ***t*** have the same length.

Subscribe to see which companies asked this question

Hide Tags | Hash Table

Hide Similar Problems | (E) Word Pattern

Solution:

```
public class Solution {
    public boolean isIsomorphic(String s, String t) {
        if(s==null || t==null) return false;
        if(s.length()!=t.length()) return false;
        if(s.length()==0 && t.length()==0) return true;
        HashMap<Character, Character> map = new HashMap<>();
```

```
        for(int i=0;i<s.length();i++){
            char c1 = s.charAt(i);
            char c2 = t.charAt(i);

            Character c = getKey(map,c2);
            if(c!=null && c!=c1) return false;
            else if(map.containsKey(c1)){
                if(map.get(c1)!=c2) return false;
            }else{
                map.put(c1,c2);
            }
        }
        return true;
    }
    public Character getKey(HashMap<Character, Character> map, Character c2){
        for(Map.Entry<Character, Character> entry:map.entrySet()){
            if(c2==entry.getValue()) return entry.getKey();
        }
        return null;
    }
}
```

3.16. Valid Anagram (Easy-> LeetCode No.242)

Given two strings *s* and *t*, write a function to determine if *t* is an anagram of *s*.

For example,

s = "anagram", *t* = "nagaram", return true.

s = "rat", *t* = "car", return false.

Note:

You may assume the string contains only lowercase alphabets.

Follow up:

What if the inputs contain unicode characters? How would you adapt your solution to such case?

Solution:

```
public class Solution {
    public boolean isAnagram(String s, String t) {
        if(s==null && t==null) return true;
        else if(s==null || t==null) return false;
        else if(s.length()!=t.length()) return false;

        char[] s_arr = s.toCharArray();
        char[] t_arr = t.toCharArray();
        Arrays.sort(s_arr);
```

```
        Arrays.sort(t_arr);
        String news = new String(s_arr);
        String newt = new String(t_arr);
        if(news.equals(newt)) return true;
        return false;
    }
}
```

3.17. H-index(Medium-> LeetCode No.274)

Given an array of citations (each citation is a non-negative integer) of a researcher, write a function to compute the researcher's h-index.

According to the definition of h-index on Wikipedia: "A scientist has index *h* if *h* of his/her *N* papers have **at least** *h* citations each, and the other *N* − *h* papers have **no more than** *h* citations each."

For example, given `citations = [3, 0, 6, 1, 5]`, which means the researcher has `5` papers in total and each of them had received `3, 0, 6, 1, 5` citations respectively. Since the researcher has `3` papers with **at least** `3` citations each and the remaining two with **no more than** `3` citations each, his h-index is `3`.

Note: If there are several possible values for `h`, the maximum one is taken as the h-index.

Hint:

1. An easy approach is to sort the array first.
2. What are the possible values of h-index?
3. A faster approach is to use extra space.

Solution:

```
public class Solution {
    public int hIndex(int[] citations) {
        int len = citations.length;
        int[] bucket = new int[len+1];
        for(int c:citations){
            if(c>=len) bucket[len]++;
            else bucket[c]++;
        }
        int count = 0;
        for(int i=len;i>=0;i--){
            count+=bucket[i];
            if(count>=i) return i;
        }
        return 0;
    }
}
```

3.18. Word Pattern (Easy-> LeetCode No.290)

Given a `pattern` and a string `str`, find if `str` follows the same pattern.

Here **follow** means a full match, such that there is a bijection between a letter in `pattern` and a **non-empty** word in `str`.

Examples:

1. pattern = `"abba"`, str = `"dog cat cat dog"` should return true.
2. pattern = `"abba"`, str = `"dog cat cat fish"` should return false.
3. pattern = `"aaaa"`, str = `"dog cat cat dog"` should return false.
4. pattern = `"abba"`, str = `"dog dog dog dog"` should return false.

Notes:

You may assume `pattern` contains only lowercase letters, and `str` contains lowercase letters separated by a single space.

Solution:

```
public class Solution {
   public boolean wordPattern(String pattern, String str) {
      if(pattern== null && str==null) return true;
      else if(pattern==null || str==null)return false;

      String[] strarr = str.split(" ");
      if(pattern.length()!=strarr.length) return false;
      HashMap<Character, String> map = new HashMap<>();
      for(int i=0;i<pattern.length();i++){
         if(!map.containsKey(pattern.charAt(i))){
            if(map.containsValue(strarr[i])) return false;
            map.put(pattern.charAt(i), strarr[i]);
         }else{
            String tmp = map.get(pattern.charAt(i));
            if(tmp.equals(strarr[i])) continue;
            else return false;
         }
      }
      return true;
   }
}
```

3.19. Top K Frequent Elements(Medium-> LeetCode No.347)

Given a non-empty array of integers, return the ***k*** most frequent elements.

For example,

Given `[1,1,1,2,2,3]` and k = 2, return `[1,2]`.

Note:

- You may assume *k* is always valid, 1 ≤ *k* ≤ number of unique elements.
- Your algorithm's time complexity **must be** better than O(*n* log *n*), where *n* is the array's size.

Subscribe to see which companies asked this question

Hide Tags | Hash Table | Heap

Hide Similar Problems | (M) Word Frequency | (M) Kth Largest Element in an Array

Solution:

```
public class Solution {
   public List<Integer> topKFrequent(int[] nums, int k) {
      Map<Integer, Integer> freq = new HashMap<>();

      for (int num : nums) {
         freq.put(num, freq.getOrDefault(num, 0) + 1);
      }

      PriorityQueue<Map.Entry<Integer, Integer>> pq = new PriorityQueue<>((o1, o2) -> o1.getValue() -
o2.getValue());
      for (Map.Entry<Integer, Integer> entry : freq.entrySet()) {
         if (pq.size() < k) {
            pq.offer(entry);
         } else if (entry.getValue() > pq.peek().getValue()) {
            pq.poll();
            pq.offer(entry);
         }
      }

      List<Integer> result = new ArrayList<>();
      for (Map.Entry<Integer, Integer> entry : pq) {
         result.add(entry.getKey());
      }

      return result;
   }
}
```

3.20 Intersection of Two Arrays (Easy->Leetcode No.349)

Given two arrays, write a function to compute their intersection.

Example:

Given *nums1* = `[1, 2, 2, 1]`, *nums2* = `[2, 2]`, return `[2]`.

Note:

- Each element in the result must be unique.
- The result can be in any order.

Subscribe to see which companies asked this question

Hide Tags | Binary Search | Hash Table | Two Pointers | Sort

Hide Similar Problems | (E) Intersection of Two Arrays II

Solution

```
public class Solution {
   public int[] intersection(int[] nums1, int[] nums2) {
      if(nums1==null || nums2==null || nums1.length==0 || nums2.length==0) return new int[0];
      Set<Integer> set1 = new HashSet<>();
      Set<Integer> set2 = new HashSet<>();
      Set<Integer> resultset = new HashSet<>();
      for(int i:nums1) set1.add(i);
      for(int j:nums2) set2.add(j);
      for(int i:set1) {
         if(!set2.add(i)) resultset.add(i);
      }
      int[] result = new int[resultset.size()];
      int i = 0;
      for(int k:resultset) result[i++]=k;
      return result;
   }
}
```

3.21 Intersection of Two Arrays (Easy->Leetcode No.350)

Given two arrays, write a function to compute their intersection.

Example:

Given *nums1* = `[1, 2, 2, 1]`, *nums2* = `[2, 2]`, return `[2, 2]`.

Note:

- Each element in the result should appear as many times as it shows in both arrays.
- The result can be in any order.

Follow up:

- What if the given array is already sorted? How would you optimize your algorithm?
- What if *nums1*'s size is small compared to *num2*'s size? Which algorithm is better?
- What if elements of *nums2* are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?

Solution

If the arrays are sorted, then we can use two pointers

```
public int[] intersect(int[] nums1, int[] nums2) {
    Arrays.sort(nums1);
    Arrays.sort(nums2);
    ArrayList<Integer> list = new ArrayList<Integer>();
    int p1=0, p2=0;
    while(p1<nums1.length && p2<nums2.length){
        if(nums1[p1]<nums2[p2]){
            p1++;
        }else if(nums1[p1]>nums2[p2]){
            p2++;
        }else{
            list.add(nums1[p1]);
            p1++;
            p2++;

        }
    }

    int[] result = new int[list.size()];
    int i=0;
    while(i<list.size()){
        result[i]=list.get(i);
        i++;
    }
    return result;
}
```

Normal solution using hashmap

```
public class Solution {
   public int[] intersect(int[] nums1, int[] nums2) {
   HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
   for(int i: nums1){
      if(map.containsKey(i)){
         map.put(i, map.get(i)+1);
      }else{
         map.put(i, 1);
```

```
        }
    }

    ArrayList<Integer> list = new ArrayList<Integer>();
    for(int i: nums2){
        if(map.containsKey(i)){
            if(map.get(i)>1){
                map.put(i, map.get(i)-1);
            }else{
                map.remove(i);
            }
            list.add(i);
        }
    }

    int[] result = new int[list.size()];
    int i =0;
    while(i<list.size()){
        result[i]=list.get(i);
        i++;
    }

    return result;
}
}
```

CHAPTER 4: Binary Search

4.1 Search For a Range (Medium-> Leetcode No.34)

Given a sorted array of integers, find the starting and ending position of a given target value.

Your algorithm's runtime complexity must be in the order of $O(\log n)$.

If the target is not found in the array, return `[-1, -1]`.

For example,

Given `[5, 7, 7, 8, 8, 10]` and target value 8,

return `[3, 4]`.

Solution

```
public class Solution {
    public int[] searchRange(int[] nums, int target) {
        if(nums==null || nums.length==0) return null;
        int[] result = new int[2];
        result[0]=-1;
        result[1]=-1;
        binarySearch(nums, target, 0, nums.length-1, result);
        return result;
    }

    public void binarySearch(int[] nums, int target, int left, int right, int[] result){
        if(right<left) return;
        if(nums[left]==nums[right] && nums[left]==target){
            result[0]= left;
            result[1]=right;
            return;
        }

        int mid = left+(right-left)/2;
        if(nums[mid]<target) binarySearch(nums, target, mid+1, right, result);
        else if(nums[mid]>target) binarySearch(nums, target, left, mid-1,result);
        else{
            result[0]=mid;
            result[1]=mid;

            // handle the duplicates from mid to left
            int t1= mid;
            while(t1>left&& nums[t1]==nums[t1-1]){
                t1--;
                result[0]=t1;
            }
            // handle the duplicates from mid to right
```

```
        int t2=mid;
        while(t2<right && nums[t2]==nums[t2+1]){
          t2++;
          result[1]=t2;
        }
        return;
      }
    }
}
```

4.2 Search Insert Position (Medium->Leetcode No.35).

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You may assume no duplicates in the array.

Here are few examples.

`[1,3,5,6]`, 5 → 2

`[1,3,5,6]`, 2 → 1

`[1,3,5,6]`, 7 → 4

`[1,3,5,6]`, 0 → 0

Solution

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

  public int binarySearch(int[] nums, int target, int left, int right){
    //if(left>right) return 0;
    int mid = left+(right-left)/2;
    if(nums[mid]==target) return mid;
    else if(target<nums[mid]) return left<mid?binarySearch(nums, target, left, mid-1):left;
    else return mid<right?binarySearch(nums, target, mid+1, right):(right+1);
  }
}
```

4.3 Pow(x, n) (Medium->Leetcode No.50)

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

Solution

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

```
    if(n<0) return 1/pow(x,-n);
    else return pow(x,n);
  }
  public double pow(double x, int n){
    if(n==0) return 1;
    if(x==0) return 0;
    double v = pow(x, n/2);
    if(n%2==0) return v*v;
    else return v*v*x;
  }
}
```

4.4 Sqrt(x) (Medium->Leetcode No.69)

Implement `int sqrt(int x)`.

Compute and return the square root of *x*.

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

4.5 Search a 2D Matrix (Medium->Leetcode No.74)

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

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

For example,

Consider the following matrix:

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

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

Solution

```
public class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        if(matrix==null || matrix.length==0 || matrix[0].length==0) return false;
        int m = matrix.length;
        int n = matrix[0].length;
        int start = 0;
        int end = m*n-1;
        while(start<=end){
            int mid = start+(end-start)/2;
            int midX = mid/n;
            int midY = mid%n;
            if(matrix[midX][midY]==target) return true;
            else if(matrix[midX][midY]<target) start = mid+1;
            else end = mid-1;
        }
        return false;
    }
}
```

4.6 Search in Rotated Sorted Array (Medium->Leetcode No.81)

Follow up for "Search in Rotated Sorted Array":

What if *duplicates* are allowed?

Would this affect the run-time complexity? How and why?

Write a function to determine if a given target is in the array.

Subscribe to see which companies asked this question

Hide Tags Array Binary Search

Hide Similar Problems (H) Search in Rotated Sorted Array

Solution

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

4.7 Minimum Size Subarray Sum (Medium->Leetcode No.209)

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

For example, given the array `[2,3,1,2,4,3]` and `s = 7`,

the subarray `[4,3]` has the minimal length under the problem constraint.

click to show more practice.

Credits:

Special thanks to @Freezen for adding this problem and creating all test cases.

Subscribe to see which companies asked this question

Hide Tags | Array | Two Pointers | Binary Search

Hide Similar Problems | (H) Minimum Window Substring | (E) Maximum Size Subarray Sum Equals k

Solution

```
public class Solution {
   public int minSubArrayLen(int s, int[] nums) {
      if(nums==null || nums.length==0) return 0;
      int start = 0;
      int sum = 0;
      int i = 0;
      int result = nums.length;
      boolean exists = false;
      while(i<=nums.length){
         if(sum>=s){
            exists = true;
            if(start==i-1) return 1;
            result = Math.min(result, i-start);
            sum = sum-nums[start];
            start++;
         }else{
            if(i == nums.length) break;
            sum = sum + nums[i];
            i++;
         }
      }
      if(exists) return result;
      else return 0;
   }
}
```

4.8 Kth Smallest Element in a BST (Medium->Leetcode No.230)

Given a binary search tree, write a function `kthSmallest` to find the **k**th smallest element in it.

Note:

You may assume k is always valid, 1 ≤ k ≤ BST's total elements.

Follow up:

What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the kthSmallest routine?

Hint:

1. Try to utilize the property of a BST.
2. What if you could modify the BST node's structure?
3. The optimal runtime complexity is O(height of BST).

Solution 1: inorder traverse the tree and get the kth smallest element. Time complexity is O(n).

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public int kthSmallest(TreeNode root, int k) {
        Stack<TreeNode> stack = new Stack<>();
        TreeNode p = root;
        int result = 0;
        while(!stack.isEmpty() || p!=null){
            if(p!=null){
                stack.push(p);
                p = p.left;
            }else{
                TreeNode tmp = stack.pop();
                k--;
                if(k==0) result = tmp.val;
                p = tmp.right;
            }
        }
        return result;
    }
```

```
}
```
Solution 2: using dfs and recursive
```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public int kthSmallest(TreeNode root, int k) {
        List<Integer> result = new ArrayList<>();
        dfs(root, result);
        return result.get(k-1);
    }

    public void dfs(TreeNode root, List<Integer> list){
        if(root==null) return;
        dfs(root.left, list);
        list.add(root.val);
        dfs(root.right, list);
    }
}
```
4.9 Searach a 2D Matrix II (Medium-> Leetcode No.240)

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

- Integers in each row are sorted in ascending from left to right.
- Integers in each column are sorted in ascending from top to bottom.

For example,

Consider the following matrix:

```
[
  [1,   4,  7, 11, 15],
  [2,   5,  8, 12, 19],
  [3,   6,  9, 16, 22],
  [10, 13, 14, 17, 24],
  [18, 21, 23, 26, 30]
]
```

Given **target** = `5`, return `true`.

Given **target** = `20`, return `false`.

Solution:

```
public class Solution {
   public boolean searchMatrix(int[][] matrix, int target) {
      if(matrix==null || matrix.length==0 || matrix[0].length==0) return false;
      int line = 0;
      int m = matrix.length;
      int n = matrix[0].length;
      while(line < m && n>0){
         int tmp = matrix[line][n-1];
         if(tmp>target) n--;
         if(tmp<target) line++;
         if(tmp==target) return true;
      }
      return false;
   }
}
```

4.10 Count Complete Tree Nodes (Medium-> LeetCode No.222)

Given a **complete** binary tree, count the number of nodes.

Definition of a complete binary tree from Wikipedia:

In a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and 2^h nodes inclusive at the last level h.

Subscribe to see which companies asked this question

Show Tags

Hide Similar Problems (E) Closest Binary Search Tree Value

Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public int countNodes(TreeNode root) {
        if(root==null) return 0;
        int leftHeight = getLeftOrRightHeight(root.left, true);
        int rightHeight = getLeftOrRightHeight(root.right, false);
        if(leftHeight==rightHeight) return (2<<(leftHeight))-1;
        else return countNodes(root.left)+countNodes(root.right)+1;
    }

    public int getLeftOrRightHeight(TreeNode node, boolean leftorright){
        if(node==null) return 0;
        int height = 0;
        while(node!=null){
            height++;
            node = leftorright==true?node.left:node.right;
        }
        return height;
    }
}
```

4.11 Divide Two Integers (Medium-> LeetCode No.29)

Divide two integers without using multiplication, division and mod operator.

If it is overflow, return MAX_INT.

Subscribe to see which companies asked this question

Show Tags

Analysis

This problem can be solved based on the fact that any number can be converted to the format of the following:

```
num=a_0*2^0+a_1*2^1+a_2*2^2+...+a_n*2^n
```

The time complexity is O(logn).

```
public class Solution {
    public int divide(int dividend, int divisor) {
        if(divisor==0 || (divisor==-1 && dividend==Integer.MIN_VALUE)) return Integer.MAX_VALUE;

        long pDivident = Math.abs((long)dividend);
        long pDivisor = Math.abs((long)divisor);

        int result = 0;
        while(pDivident>=pDivisor){
            int numShift = 0;
            while(pDivident>=(pDivisor<<numShift)) numShift++;
            result+=(1<<numShift-1);
            pDivident-=pDivisor<<(numShift-1);
        }
        if((dividend>0 && divisor>0) || (dividend<0 && divisor<0)) return result;
        else return -result;
    }
}
```

4.12 Find Peak Element (Medium-> LeetCode No.162)

A peak element is an element that is greater than its neighbors.

Given an input array where `num[i] ≠ num[i+1]`, find a peak element and return its index.

The array may contain multiple peaks, in that case return the index to any one of the peaks is fine.

You may imagine that `num[-1] = num[n] = -∞`.

For example, in array `[1, 2, 3, 1]`, 3 is a peak element and your function should return the index number 2.

click to show spoilers.

Note:

Your solution should be in logarithmic complexity.

Solution

```
public class Solution {
    public int findPeakElement(int[] nums) {
        if(nums.length==1) return 0;
        int start = 0;
        int end = nums.length-1;
        while(start+1<end){
            int mid = start + (end-start)/2;
            if(nums[mid]<nums[mid+1]) start=mid;
            else if (nums[mid]<nums[mid-1]) end = mid;
            else return mid;
        }
        return nums[start]>nums[end]?start:end;
    }
}
```

4.13 First Bad Version (Medium-> LeetCode No.278)

You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

Suppose you have `n` versions `[1, 2, ..., n]` and you want to find out the first bad one, which causes all the following ones to be bad.

You are given an API `bool isBadVersion(version)` which will return whether `version` is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

Solution

```
/* The isBadVersion API is defined in the parent class VersionControl.
      boolean isBadVersion(int version); */

public class Solution extends VersionControl {
    public int firstBadVersion(int n) {
        //if(n==0) return 0;
        //if(n==1) return 1;
```

```
        int start = 1;
        int end = n;
        while(start<end){
            int mid = start+(end-start)/2;
            if(isBadVersion(mid)) end = mid;
            else start = mid+1;
        }
        return start;
    }
}
```

4.14 H-Index II (Medium-> LeetCode No.275)

Follow up for H-Index: What if the `citations` array is sorted in ascending order? Could you optimize your algorithm?

Hint:

1. Expected runtime complexity is in $O(\log n)$ and the input is sorted.

Solution

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

CHAPTER 5: Dynamic Programming

5.1 Maximum Subarray (Medium-> LeetCode No.53)

Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

For example, given the array `[-2,1,-3,4,-1,2,1,-5,4]`,

the contiguous subarray `[4,-1,2,1]` has the largest sum = `6`.

Solution:

```
public class Solution {
   public int maxSubArray(int[] nums) {
      int max = nums[0];
      int[] dp = new int[nums.length];
      dp[0] = nums[0];
      for(int i=1;i<nums.length;i++){
         dp[i] = Math.max(nums[i], dp[i-1]+nums[i]);
         max = Math.max(max, dp[i]);
      }
      return max;
   }
}
```

5.2 Unique Paths (Medium-> LeetCode No.62)

A robot is located at the top-left corner of a *m* x *n* grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?

Above is a 3 x 7 grid. How many possible unique paths are there?

Note: *m* and *n* will be at most 100.

Solution:

```
public class Solution {
   public int uniquePaths(int m, int n) {
      int[][] dp = new int[m+1][n+1];
      dp[m][n-1] = 1;
      for(int i=m-1;i>=0;i--){
         for(int j=n-1;j>=0;j--){
            dp[i][j] = dp[i+1][j]+dp[i][j+1];
```

```
        }
      }
      return dp[0][0];
    }
}
```

5.3 Unique Paths II (Medium-> LeetCode No.63)

Follow up for "Unique Paths":

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space is marked as `1` and `0` respectively in the grid.

For example,

There is one obstacle in the middle of a 3x3 grid as illustrated below.

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

The total number of unique paths is `2`.

Note: *m* and *n* will be at most 100.

Solution:

```
public class Solution {
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {
        int m = obstacleGrid.length;
        int n = obstacleGrid[0].length;
        for(int i=0;i<m;i++){
            for(int j = 0;j<n;j++){
                if(obstacleGrid[i][j]==1) obstacleGrid[i][j]=0;
                else{
                    if(i==0 && j==0) obstacleGrid[i][j]=1;
                    else if(i==0 && j>0) obstacleGrid[i][j]=obstacleGrid[i][j-1];
                    else if(i>0 && j==0) obstacleGrid[i][j] = obstacleGrid[i-1][j];
                    else obstacleGrid[i][j] = obstacleGrid[i-1][j]+obstacleGrid[i][j-1];
                }
            }
        }
        return obstacleGrid[m-1][n-1];
    }
}
```

5.4 Minimum Path Sum (Medium-> LeetCode No.64)

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

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

Solution:

```
public class Solution {
   public int minPathSum(int[][] grid) {
      if(grid==null || grid.length==0) return 0;
      int m = grid.length;
      int n = grid[0].length;
      int[][] dp = new int[m][n];
      dp[0][0] = grid[0][0];
      for(int i=1; i<n;i++){
         dp[0][i] = dp[0][i-1]+grid[0][i];
      }
      for(int j = 1;j<m;j++){
         dp[j][0] = dp[j-1][0]+grid[j][0];
      }

      for(int i=1;i<m;i++){
         for(int j =1;j<n;j++){
            dp[i][j] = Math.min(dp[i-1][j], dp[i][j-1])+grid[i][j];
         }
      }
      return dp[m-1][n-1];
   }
}
```

5.5 Climbing Stairs (Medium-> LeetCode No.70)

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

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

Solution:

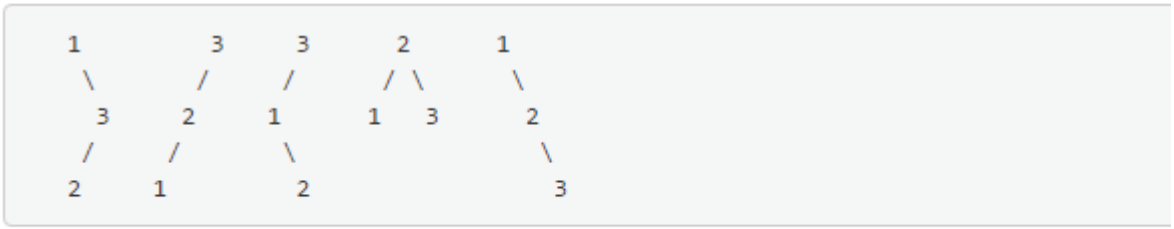
```
public class Solution {
   public int climbStairs(int n) {
      if(n==0 || n==1 || n==2) return n;
      int[] dp = new int[n+1];
      dp[0] = 0;
      dp[1] = 1;
      dp[2] = 2;
      for(int i=3;i<=n;i++){
         dp[i] = dp[i-1]+dp[i-2];
      }
      return dp[n];
   }
}
```

5.6 Unique Binary Search Trees II (Medium ->LeetCode No.95)

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

For example,

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



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

Solution:

```
/**
 * 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<TreeNode> generateTrees(int n) {
        List<TreeNode> result = new LinkedList<>();
        if(n<=0) return result;
        return generateTrees( 1, n);
    }

    public List<TreeNode> generateTrees( int start, int end){
        List<TreeNode> result = new LinkedList<>();
        if(start>end){
            result.add(null);
            return result;
        }
        for(int i=start;i<=end;i++){
            List<TreeNode> lefts = generateTrees(start, i-1);
            List<TreeNode> rights = generateTrees(i+1,end);
            for(TreeNode left:lefts){
                for(TreeNode right:rights){
                    TreeNode root = new TreeNode(i);
                    root.left = left;
                    root.right = right;
                    result.add(root);
```

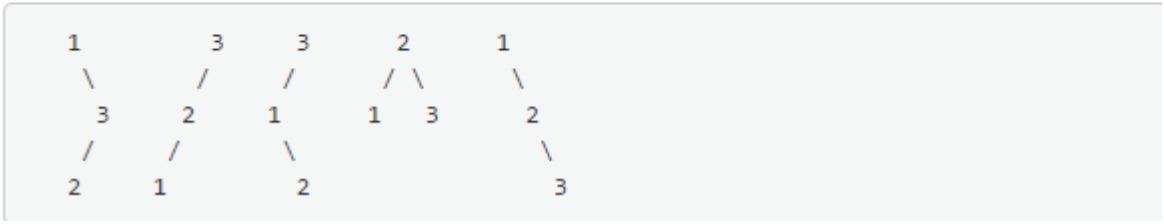
```
            }
        }
    }
    return result;
  }
}
```

5.7 Unique Binary Search Trees (Medium ->LeetCode No.96)

Given *n*, how many structurally unique **BST's** (binary search trees) that store values 1...*n*?

For example,

Given *n* = 3, there are a total of 5 unique BST's.



Solution:

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

5.8 Triangle (Medium->LeetCoede No.120)

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

For example, given the following triangle

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

The minimum path sum from top to bottom is `11` (i.e., 2 + 3 + 5 + 1 = 11).

Note:

Bonus point if you are able to do this using only *O*(*n*) extra space, where *n* is the total number of rows in the triangle.

Solution:

```
public class Solution {
   public int minimumTotal(List<List<Integer>> triangle) {
      if(triangle==null || triangle.size()==0) return 0;
      int len = triangle.size()-1;
      int[] total = new int[triangle.size()];

      for(int i=0;i<triangle.get(len).size();i++){
         total[i] = triangle.get(len).get(i);
      }

      for(int i = triangle.size()-2;i>=0;i--){
         for(int j=0;j<triangle.get(i).size();j++){
            total[j] = triangle.get(i).get(j)+Math.min(total[j],total[j+1]);
         }
      }
      return total[0];
   }
}
```

5.9. Best Time to Buy and Sell Stock (Medium-> LeetCode No.121)

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

If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

Subscribe to see which companies asked this question

Hide Tags | Array | Dynamic Programming

Hide Similar Problems | (M) Maximum Subarray | (M) Best Time to Buy and Sell Stock II | (H) Best Time to Buy and Sell Stock III | (H) Best Time to Buy and Sell Stock IV | (M) Best Time to Buy and Sell Stock with Cooldown

Solution:

```
public class Solution {
   public int maxProfit(int[] prices) {
```

```
        if(prices==null || prices.length==0) return 0;
        int profit = 0;
        int minElement = Integer.MAX_VALUE;
        for(int i=0;i<prices.length;i++){
            profit = Math.max(profit, prices[i]-minElement);
            minElement = Math.min(minElement, prices[i]);
        }
        return profit;
    }
}
```

5.10 Maximum Product Subarray (Medium -> LeetCode No.152)

Find the contiguous subarray within an array (containing at least one number) which has the largest product.

For example, given the array `[2,3,-2,4]`,

the contiguous subarray `[2,3]` has the largest product = `6`.

Solution:

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

5.11 Maximal Square (Medium->LeetCode No.221)

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

For example, given the following matrix:

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

Return 4.

Solution:

```
public class Solution {
    public int maximalSquare(char[][] matrix) {
```

```
        if(matrix==null || matrix.length==0 || matrix[0].length==0) return 0;
        int m = matrix.length;
        int n = matrix[0].length;

        int[][] dp = new int[m][n];
        for(int i=0;i<n;i++){
            dp[0][i]=Character.getNumericValue(matrix[0][i]);
        }
        for(int j=0;j<m;j++) dp[j][0] = Character.getNumericValue(matrix[j][0]);

        for(int i=1;i<m;i++){
            for(int j=1;j<n;j++){
                if(matrix[i][j]=='1'){
                    dp[i][j] = Math.min(Math.min(dp[i][j-1],dp[i-1][j]),dp[i-1][j-1])+1;
                }else{
                    dp[i][j]=0;
                }
            }
        }

        int max = 0;
        for(int i=0;i<m;i++){
            for(int j=0;j<n;j++){
                if(dp[i][j]>max) max=dp[i][j];
            }
        }
        return max*max;
    }
}
```

5.12 Ugly Number II (Medium->LeetCode No.264)

Write a program to find the `n`-th ugly number.

Ugly numbers are positive numbers whose prime factors only include `2, 3, 5`. For example, `1, 2, 3, 4, 5, 6, 8, 9, 10, 12` is the sequence of the first `10` ugly numbers.

Note that `1` is typically treated as an ugly number.

Hint:

1. The naive approach is to call `isUgly` for every number until you reach the n^{th} one. Most numbers are *not* ugly. Try to focus your effort on generating only the ugly ones.
2. An ugly number must be multiplied by either 2, 3, or 5 from a smaller ugly number.
3. The key is how to maintain the order of the ugly numbers. Try a similar approach of merging from three sorted lists: L_1, L_2, and L_3.
4. Assume you have U_k, the k^{th} ugly number. Then U_{k+1} must be Min($L_1 * 2$, $L_2 * 3$, $L_3 * 5$).

Solution:

```
public class Solution {
    public int nthUglyNumber(int n) {
        if(n==1) return 1;
```

```
        int[] dp = new int[n+1];
        int p2=1, p3=1,p5=1;
        dp[1] = 1;
        for(int i=2;i<=n;i++){
            dp[i]=Math.min(Math.min(2*dp[p2],3*dp[p3]),5*dp[p5]);
            if(dp[i]==2*dp[p2]) p2++;
            if(dp[i]==3*dp[p3]) p3++;
            if(dp[i]==5*dp[p5]) p5++;
        }
        return dp[n];
    }
}
```

5.13 Longest Increasing Subsequence (Medium ->LeetCode No.300)

Given an unsorted array of integers, find the length of longest increasing subsequence.

For example,

Given `[10, 9, 2, 5, 3, 7, 101, 18]`,

The longest increasing subsequence is `[2, 3, 7, 101]`, therefore the length is `4`. Note that there may be more than one LIS combination, it is only necessary for you to return the length.

Your algorithm should run in O(n^2) complexity.

Follow up: Could you improve it to O(n log n) time complexity?

O(n^2) Solution:

```
public class Solution {
    public int lengthOfLIS(int[] nums) {
        if(nums==null || nums.length==0) return 0;
        int max = 1;
        int[] dp = new int[nums.length];
        for(int i=0;i<nums.length;i++){
            dp[i] = 1;
            for(int j=0;j<i;j++){
                if(nums[j]<nums[i]) dp[i] = Math.max(dp[j]+1,dp[i]);
            }
            max = Math.max(max, dp[i]);
        }
        return max;
    }
}
```

O(nlogn) Solution:

```
public class Solution {
    public int lengthOfLIS(int[] nums) {
        if(nums==null || nums.length==0) return 0;
        ArrayList<Integer> list = new ArrayList<>();
        for(int n:nums){
            if(list.size()==0 || list.get(list.size()-1)<n){
                list.add(n);
            }else{
```

```
            updateLIS(list, n);
        }
    }
    return list.size();
}
public void updateLIS(ArrayList<Integer> list, int n){
    int low = 0;
    int high = list.size()-1;
    while(low<high){
        int mid = low+(high-low)/2;
        if(list.get(mid)<n) low = mid+1;
        else high = mid;
    }
    list.set(low, n);
}
}
```

5.14 Range Sum Query - Immutable (Easy -> LeetCode No.303)

Given an integer array *nums*, find the sum of the elements between indices *i* and *j* (*i* ≤ *j*), inclusive.

Example:

```
Given nums = [-2, 0, 3, -5, 2, -1]

sumRange(0, 2) -> 1
sumRange(2, 5) -> -1
sumRange(0, 5) -> -3
```

Note:

1. You may assume that the array does not change.
2. There are many calls to *sumRange* function.

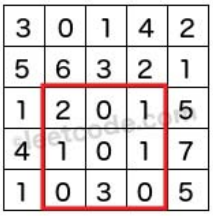
Solution:

```
public class NumArray {
    public int[] dp;
    public NumArray(int[] nums) {
        int sum = 0;
        dp = new int[nums.length+1];
        for(int i=0;i<nums.length;i++){
            sum += nums[i];
            dp[i+1] = sum;
        }
    }

    public int sumRange(int i, int j) {
        return dp[j+1]-dp[i];
```

```
    }
}
// Your NumArray object will be instantiated and called as such:
// NumArray numArray = new NumArray(nums);
// numArray.sumRange(0, 1);
// numArray.sumRange(1, 2);
```

5.15 Range Sum Query 2D - Immutable (Medium -> LeetCode No.304)

Given a 2D matrix *matrix*, find the sum of the elements inside the rectangle defined by its upper left corner (*row*1, *col*1) and lower right corner (*row*2, *col*2).



3	0	1	4	2
5	6	3	2	1
1	2	0	1	5
4	1	0	1	7
1	0	3	0	5

The above rectangle (with the red border) is defined by (row1, col1) = **(2, 1)** and (row2, col2) = **(4, 3)**, which contains sum = **8**.

Example:

```
Given matrix = [
  [3, 0, 1, 4, 2],
  [5, 6, 3, 2, 1],
  [1, 2, 0, 1, 5],
  [4, 1, 0, 1, 7],
  [1, 0, 3, 0, 5]
]

sumRegion(2, 1, 4, 3) -> 8
sumRegion(1, 1, 2, 2) -> 11
sumRegion(1, 2, 2, 4) -> 12
```

Note:

1. You may assume that the matrix does not change.
2. There are many calls to *sumRegion* function.
3. You may assume that *row*1 ≤ *row*2 and *col*1 ≤ *col*2.

Solution:

```
public class NumMatrix {
   private int[][] sumRegion;
   public NumMatrix(int[][] matrix) {
      if (matrix.length != 0)  sumRegion = new int[matrix.length + 1][matrix[0].length + 1];

      for (int i = 0; i < matrix.length; i++) {
         int sum = 0;
         for (int j = 0; j < matrix[0].length; j++) {
            sum += matrix[i][j];
            sumRegion[i + 1][j + 1] = sum + sumRegion[i][j + 1];
         }
      }
   }
```

```
    public int sumRegion(int row1, int col1, int row2, int col2) {
        return sumRegion[row2 + 1][col2 + 1] - sumRegion[row1][col2 + 1] - sumRegion[row2 + 1][col1] +
sumRegion[row1][col1];
    }
}
// Your NumMatrix object will be instantiated and called as such:
// NumMatrix numMatrix = new NumMatrix(matrix);
// numMatrix.sumRegion(0, 1, 2, 3);
// numMatrix.sumRegion(1, 2, 3, 4);
```

5.16 Integer Break (Medium -> LeetCode No.343)

Given a positive integer *n*, break it into the sum of **at least** two positive integers and maximize the product of those integers. Return the maximum product you can get.

For example, given *n* = 2, return 1 (2 = 1 + 1); given *n* = 10, return 36 (10 = 3 + 3 + 4).

Note: you may assume that *n* is not less than 2.

Hint:

1. There is a simple O(n) solution to this problem.
2. You may check the breaking results of *n* ranging from 7 to 10 to discover the regularities.

Solution 1:

If we see the breaking result for some numbers, we can see repeated pattern like the following:

```
2 -> 1*1
3 -> 1*2
4 -> 2*2
5 -> 3*2
6 -> 3*3
7 -> 3*4
8 -> 3*3*2
9 -> 3*3*3
10 -> 3*3*4
11 -> 3*3*3*2
```

we only need to find how many 3's we can get when n>4, if n%3==1, we donot want 1 to be one of the broken numbers, we want 4 instead.

```
public class Solution {
    public int integerBreak(int n) {
        if(n==2) return 1;
        if(n==3) return 2;
        if(n==4) return 4;
        int result = 1;
        if(n%3==0){
            int m = n/3;
```

```
            result = (int)Math.pow(3,m);
        }else if(n%3==2){
            int m = n/3;
            result = (int)Math.pow(3,m)*2;
        }else if(n%3==1){
            int m = (n-4)/3;
            result = (int)Math.pow(3,m)*4;
        }
        return result;
    }
}
```

Solution 2: dynamic programming

Let dp[i] be the max production value for breaking the number i, since dp[i+j] can be i*j, dp[i+j] = Math.max(Math.max(dp[i],i)*Math.max(dp[j],j), dp[i+j])

```
public class Solution {
    public int integerBreak(int n) {
        int[] dp = new int[n+1];
        for(int i =1;i<n;i++){
            for(int j=1;j<i+1;j++){
                if(i+j<=n) dp[i+j]=Math.max(Math.max(dp[i],i)*Math.max(dp[j],j),dp[i+j]);
            }
        }
        return dp[n];
    }
}
```

5.17 Best Time to Buy and Sell Stock with Cooldown (Medium-> LeetCode No.309)

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

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times) with the following restrictions:

- You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).
- After you sell your stock, you cannot buy stock on next day. (ie, cooldown 1 day)

Example:

```
prices = [1, 2, 3, 0, 2]
maxProfit = 3
transactions = [buy, sell, cooldown, buy, sell]
```

Solution

```
public class Solution {
    public int maxProfit(int[] prices) {
        if(prices == null || prices.length < 2){
            return 0;
        }
        int len = prices.length;
        int[] sell = new int[len]; //sell[i] means must sell at day i
```

```
        int[] cooldown = new int[len]; //cooldown[i] means day i is cooldown day
        sell[1] = prices[1] - prices[0];
        for(int i = 2; i < prices.length; ++i){
            cooldown[i] = Math.max(sell[i - 1], cooldown[i - 1]);
            sell[i] = prices[i] - prices[i - 1] + Math.max(sell[i - 1], cooldown[i - 2]);
        }
        return Math.max(sell[len - 1], cooldown[len - 1]);
    }
}
```

5.18 Coin Change (Medium-> LeetCode No.322)

You are given coins of different denominations and a total amount of money *amount*. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return `-1`.

Example 1:

coins = `[1, 2, 5]`, amount = `11`

return `3` (11 = 5 + 5 + 1)

Example 2:

coins = `[2]`, amount = `3`

return `-1`.

Note:

You may assume that you have an infinite number of each kind of coin.

Solution : using dynamic programmnig, we set dp[i] be the fewest number of coins that you need to make up that amount.

```
public class Solution {
    public int coinChange(int[] coins, int amount) {
        if(amount==0) return 0;
        int[] dp = new int[amount+1];
        Arrays.fill(dp, Integer.MAX_VALUE);
        dp[0] = 0;
        for(int i=0;i<=amount;i++){
            for(int coin:coins){
                if(i+coin<=amount){
                    if(dp[i]!=Integer.MAX_VALUE) dp[i+coin] = Math.min(dp[i+coin],dp[i]+1);
                }
            }
        }
        if(dp[amount]==Integer.MAX_VALUE) return -1;
        return dp[amount];
    }
}
```

5.19 Counting bits (Medium-> LeetCode No.338)

1. Naive Solution

We can simply count bits for each number like the following:

```
public int[] countBits(int num) {
    int[] result = new int[num+1];

    for(int i=0; i<=num; i++){
        result[i] = countEach(i);
    }

    return result;
}

public int countEach(int num){
    int result = 0;

    while(num!=0){
        if(num%2==1){
            result++;
        }
        num = num/2;
    }

    return result;
}
```

Given a non negative integer number **num**. For every numbers **i** in the range **0 ≤ i ≤ num** calculate the number of 1's in their binary representation and return them as an array.

Example:

For `num = 5` you should return `[0,1,1,2,1,2]`.

Follow up:

- It is very easy to come up with a solution with run time **O(n*sizeof(integer))**. But can you do it in linear time **O(n)** /possibly in a single pass?
- Space complexity should be **O(n)**.
- Can you do it like a boss? Do it without using any builtin function like **__builtin_popcount** in c++ or in any other language.

Hint:

1. You should make use of what you have produced already.
2. Divide the numbers in ranges like [2-3], [4-7], [8-15] and so on. And try to generate new range from previous.
3. Or does the odd/even status of the number help you in calculating the number of 1s?

Solution

```
public class Solution {
   public int[] countBits(int num) {
      int[] result = new int[num+1];
      int p = 1;
      int pow = 1;
```

```
        for(int i=1;i<=num;i++){
            if(i==pow){
                result[i] = 1;
                pow <<=1;
                p=1;
            }else{
                result[i] = result[p]+1;
                p++;
            }
        }
        return result;
    }
}
```

5.20 Decode Ways (Medium-> LeetCode No.91)

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

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

Given an encoded message containing digits, determine the total number of ways to decode it.

For example,

Given encoded message `"12"`, it could be decoded as `"AB"` (1 2) or `"L"` (12).

The number of ways decoding `"12"` is 2.

Solution

```
public class Solution {
    public int numDecodings(String s) {
        if(s==null || s.length()==0 || s.equals("0")) return 0;
        int[] dp = new int[s.length()+1];
        dp[0]=1;
        if(isValid(s.substring(0,1))) dp[1]=1;
        else dp[1] = 0;
        for(int i=2;i<=s.length();i++){
            if(isValid(s.substring(i-1,i))) dp[i]+=dp[i-1];
            if(isValid(s.substring(i-2,i))) dp[i]+=dp[i-2];
        }
        return dp[s.length()];
    }

    public boolean isValid(String s){
        if(s.charAt(0)=='0') return false;
        int tmp = Integer.parseInt(s);
```

```
        return tmp>=1&&tmp<=26;
    }
}
```

5.21 House Robber (Easy-> LeetCode No.198)

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

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

Credits:

Special thanks to @ifanchu for adding this problem and creating all test cases. Also thanks to @ts for adding additional test cases.

Subscribe to see which companies asked this question

Hide Tags | Dynamic Programming

Hide Similar Problems | (M) Maximum Product Subarray | (M) House Robber II | (M) Paint House | (E) Paint Fence | (M) House Robber III

Solution:

```
public class Solution {
    public int rob(int[] nums) {
        if(nums==null || nums.length==0) return 0;
        if(nums.length==1) return nums[0];
        int[] dp = new int[nums.length];
        dp[0] = nums[0];
        dp[1] = Math.max(nums[0],nums[1]);
        for(int i=2;i<nums.length;i++){
            dp[i] = Math.max(dp[i-2]+nums[i], dp[i-1]);
        }
        return dp[nums.length-1];
    }
}
```

5.22 House Robber II (Medium-> LeetCode No.213)

Note: This is an extension of House Robber.

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

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

Solution:

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

        // if 1st element is included, instead of the last one
        int[] dp1 = new int[len];
        dp1[0] = 0;
        dp1[1] = nums[0];
        for(int i=2;i<len;i++){
            dp1[i] = Math.max(dp1[i-2]+nums[i-1],dp1[i-1]);
        }

        // if the last element is included instead of the first one
        int[] dp2 = new int[len];
        dp2[0] = 0;
        dp2[1] = nums[1];
        for(int i=2;i<len;i++){
            dp2[i] = Math.max(dp2[i-2]+nums[i],dp2[i-1]);
        }
        return Math.max(dp1[len-1],dp2[len-1]);
    }
}
```

CHAPTER 6: Depth-First Search

6.1. Binary Tree Paths (Easy-> LeetCode No.257)

Given a binary tree, return all root-to-leaf paths.

For example, given the following binary tree:

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

All root-to-leaf paths are:

```
["1->2->5", "1->3"]
```

Solution:

```
/**
 * 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<String> binaryTreePaths(TreeNode root) {
        List<String> result = new ArrayList<>();
        if(root==null) return result;
        dfs(root,result, new StringBuilder());
        return result;
    }

    public void dfs(TreeNode root, List<String> result, StringBuilder sb){
        if(root.left==null && root.right==null){
            sb.append(root.val);
            result.add(sb.toString());
            return;
        }

        sb.append(root.val);
        sb.append("->");
        if(root.left!=null){
            dfs(root.left, result, new StringBuilder(sb));
```

```
        }
        if(root.right!=null){
            dfs(root.right, result, new StringBuilder(sb));
        }
    }
}
```

6.2. Binary Tree Right Side View (Medium-> LeetCode No.199)

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

For example:

Given the following binary tree,

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

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

Solution:

```
/**
 * 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> rightSideView(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        if(root==null) return result;
        LinkedList<TreeNode> queue = new LinkedList<>();

        queue.add(root);
        while(queue.size()>0){
            int size = queue.size();
            for(int i=0;i<size;i++){
                TreeNode node = queue.pop();
                if(i==0) result.add(node.val);
                if(node.right!=null) queue.add(node.right);
                if(node.left!=null) queue.add(node.left);
            }
```

```
    }
    return result;
  }
}
```

6.3. Construct Binary Tree from Inorder and Postorder Traversal (Medium-> LeetCode No.106)

Given inorder and postorder traversal of a tree, construct the binary tree.

Note:

You may assume that duplicates do not exist in the tree.

Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
  public TreeNode buildTree(int[] inorder, int[] postorder) {
    int inStart = 0;
    int inEnd = inorder.length-1;
    int pStart = 0;
    int pEnd = postorder.length-1;
    return buildTree(inorder, inStart, inEnd, postorder, pStart, pEnd);
  }

  public TreeNode buildTree(int[] inorder, int inStart, int inEnd, int[] postorder, int pStart, int pEnd){
    if(inStart>inEnd || pStart>pEnd) return null;
    int rootVal = postorder[pEnd];
    TreeNode root = new TreeNode(rootVal);

    int k=0;
    for(int i=0;i<=inEnd;i++){
      if(inorder[i]==rootVal){
        k = i;
        break;
      }
    }
    root.left = buildTree(inorder, inStart, k-1, postorder, pStart,pStart+k-(inStart+1));
    root.right = buildTree(inorder, k+1,inEnd, postorder, pStart+k-inStart, pEnd-1);
    return root;
  }
}
```

6.4. Construct Binary Tree from Preorder and Inordreorder Traversal (Medium-> LeetCode No.105)

Given preorder and inorder traversal of a tree, construct the binary tree.

Note:

You may assume that duplicates do not exist in the tree.

Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        int preStart = 0;
        int preEnd = preorder.length-1;
        int inStart = 0;
        int inEnd = inorder.length-1;
        return buildTree(preorder, preStart, preEnd, inorder, inStart, inEnd);
    }

    public TreeNode buildTree(int[] preorder, int preStart, int preEnd, int[] inorder, int inStart, int inEnd){
        if(preStart>preEnd || inStart>inEnd) return null;
        int rootVal = preorder[preStart];
        TreeNode root = new TreeNode(rootVal);
        int k = 0;
        for(int i=0;i<=inEnd;i++){
            if(inorder[i]==rootVal){
                k=i;
                break;
            }
        }

        root.left = buildTree(preorder, preStart+1, preStart+(k-inStart), inorder, inStart, k-1);
        root.right = buildTree(preorder, preStart+(k-inStart)+1, preEnd, inorder, k+1, inEnd);
        return root;
    }
}
```

6.5. Course Schedule (Medium-> LeetCode No.207)

There are a total of *n* courses you have to take, labeled from `0` to `n - 1`.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`

Given the total number of courses and a list of prerequisite **pairs**, is it possible for you to finish all courses?

For example:

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

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

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

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

Solution:

```
public class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        if(prerequisites==null) throw new IllegalArgumentException("illegal prerequisites array");
        int len = prerequisites.length;
        if(numCourses==0 || len==0) return true;

        int[] pCounter = new int[numCourses];
        for(int i=0;i<len;i++){
            pCounter[prerequisites[i][0]]++;
        }
        LinkedList<Integer> queue = new LinkedList<>();
        for(int i=0;i<numCourses;i++){
            if(pCounter[i]==0) queue.add(i);
        }

        int size = queue.size();
        while(!queue.isEmpty()){
            int top = queue.remove();
            for(int i =0;i<len;i++){
                if(prerequisites[i][1]==top){
                    pCounter[prerequisites[i][0]]--;
                    if(pCounter[prerequisites[i][0]]==0){
                        size++;
                        queue.add(prerequisites[i][0]);
                    }
```

```
            }
          }
        }
        return size==numCourses;
    }
}
```

6.6. Course Schedule II (Medium-> LeetCode No.201)

There are a total of n courses you have to take, labeled from `0` to `n - 1`.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`

Given the total number of courses and a list of prerequisite **pairs**, return the ordering of courses you should take to finish all courses.

There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.

For example:

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

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

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

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

【解析】

相比较于 【LeetCode】Course Schedule 解题报告，返回的结果是一组排序结果，而非是否能够排序。

因为只要返回任意一种合法的拓扑排序结果即可，所以只需在BFS过程中记录下来先后访问的节点即可。

Solution:

```
public int[] findOrder(int numCourses, int[][] prerequisites){
        if(prerequisites==null) throw new IllegalArgumentException("illegal prerequisites  array");
        int len = prerequisites.length;
        if(len==0){
                int[] result = new int[numCourses];
                for(int i=0;i<numCourses;i++) result[i]=i;
                return result;
```

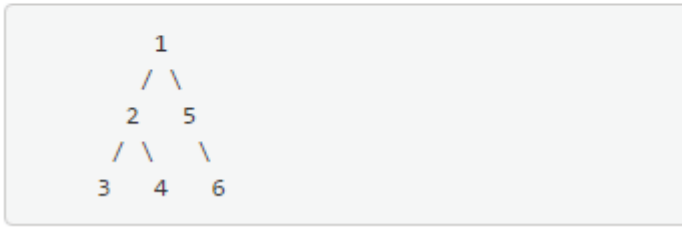
```
        }
        int[] pCounter = new int[numCourses];
        for(int i=0;i<len;i++){
                pCounter[prerequisites[i][0]]++;
        }

        ListedList<Integer> queue = new LinkedList<>();
        for(int i=0;i<numCourses;i++){
                if(pCounter[i]==0) queue.add(i);
        }

        int numNoPre = queue.size();
        int[] result = new int[numCourses];
        int j=0;
        while(!queue.isEmpty()){
                int c = queue.remove();
                result[j++]=c;
                for(int i=0;i<len;i++){
                        if(prerequisites[i][c]==c){
                                pCounter[prerequisites[i][0]]--;
                                if(pCounter[prerequisites[i][0]]==0){
                                        queue.add(prerequisites[i][0]);
                                        numNoPre++;
                                }
                        }
                }
        }
        if(numNoPre==numCourses) return result;
        else return new int[0];
}
```

6.7. Flatten Binary Tree To Linked List (Medium-> LeetCode No.114)

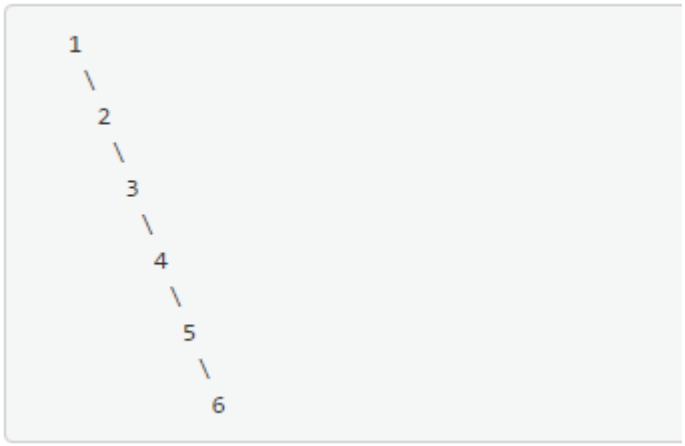
Given a binary tree, flatten it to a linked list in-place.

For example,

Given



The flattened tree should look like:



Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public void flatten(TreeNode root) {
        if(root==null) return;
        flatten(root.left);
        flatten(root.right);

        TreeNode temp = root.right;
        root.right = root.left;
        root.left = null;
```

```
        while(root.right!=null) root = root.right;
        root.right = temp;
    }
}
```

6.8. Validate Binary Search Tree (Medium-> LeetCode No.98)

Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

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

Example 1:

```
  2
 / \
1   3
```

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

Example 2:

```
  1
 / \
2   3
```

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

Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public boolean isValidBST(TreeNode root) {
        if(root==null) return true;
        return isValidBST(root, Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY);
```

```
    }

    public boolean isValidBST(TreeNode node, double min, double max){
        if(node==null) return true;
        if(node.val<=min || node.val>=max) return false;
        return isValidBST(node.left, min, node.val) && isValidBST(node.right, node.val, max);
    }
}
```

6.9. Same Tree (Easy-> LeetCode No.100)

Given two binary trees, write a function to check if they are equal or not.

Two binary trees are considered equal if they are structurally identical and the nodes have the same value.

Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {
        if(p==null && q==null) return true;
        else if(p!=null && q!=null){
            if(p.val==q.val) return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
        }else return false;
        return false;
    }
}
```

6.10. Maximum Depth of Binary Tree (Easy-> LeetCode No.104)

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
```

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

6.11. Convert Sorted Array to Binary Search Tree (Medium-> LeetCode No.108)

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

Subscribe to see which companies asked this question

Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
   public TreeNode sortedArrayToBST(int[] nums) {
      if(nums==null || nums.length==0) return null;
      return sortedArrayToBST(nums, 0,nums.length-1);
   }
   public TreeNode sortedArrayToBST(int[] nums, int start, int end){
      if(start>end) return null;
      int mid = start+(end-start)/2;
      TreeNode root = new TreeNode(nums[mid]);
      root.left = sortedArrayToBST(nums, start, mid-1);
      root.right = sortedArrayToBST(nums, mid+1,end);
      return root;
   }
}
```

6.12. Convert Sorted List to Binary Search Tree(Medium-> LeetCode No.109)

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

Solution:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
```

```
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
   static ListNode h;
   public TreeNode sortedListToBST(ListNode head) {
      if(head==null) return null;
      int length = getLength(head);
      h=head;
      return sortedListToBST(0,length);
   }

   public TreeNode sortedListToBST(int start, int end){
      if(start>end) return null;
      int mid = start+(end-start)/2;
      TreeNode left =  sortedListToBST(start, mid-1);
      TreeNode root = new TreeNode(h.val);
      h = h.next;
      TreeNode right = sortedListToBST(mid+1,end);
      root.left = left;
      root.right = right;
      return root;
   }
   public int getLength(ListNode head){
      int len = 0;
      ListNode p = head;
      while(p.next!=null){
         len++;
         p = p.next;
      }
      return len;
   }
}
```

6.13. Balanced Binary Tree (Easy-> LeetCode No.110)

Given a binary tree, determine if it is height-balanced.

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

Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
   public boolean isBalanced(TreeNode root) {
      if(root==null) return true;
      return Math.abs(maxHeight(root.left)-maxHeight(root.right))<=1 && isBalanced(root.left) &&
isBalanced(root.right);
   }
    public int maxHeight(TreeNode root){
       if(root==null) return 0;
       return Math.max(maxHeight(root.left),maxHeight(root.right))+1;
    }
}
```

6.14. Path Sum (Easy-> LeetCode No.112)

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

For example:

Given the below binary tree and `sum = 22`,

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

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

Solution:

```
/**
 * Definition for a binary tree node.
```

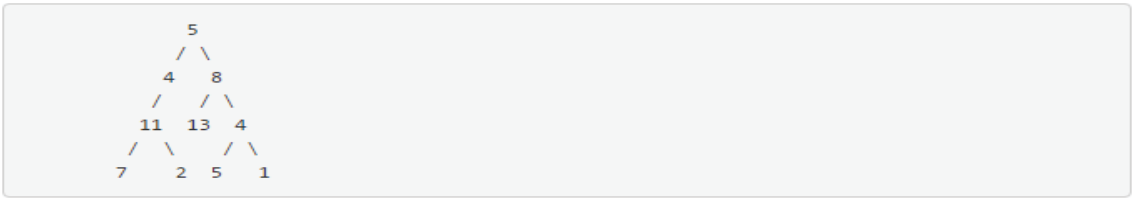
```
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public boolean hasPathSum(TreeNode root, int sum) {
        if(root==null) return false;
        if(root.val==sum && root.left==null && root.right==null) return true;
        return hasPathSum(root.left, sum-root.val) || hasPathSum(root.right, sum-root.val);
    }
}
```

6.15. Path Sum II (Medium-> LeetCode No.113)

Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.

For example:

Given the below binary tree and `sum = 22`,



return

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

Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<List<Integer>> pathSum(TreeNode root, int sum) {
        List<List<Integer>> results = new ArrayList<List<Integer>>();
        if(root==null) return results;
        List<Integer> result = new ArrayList<>();
        result.add(root.val);
        dfs(root, sum-root.val, result, results);
        return results;
```

```
    }
    public void dfs(TreeNode root, int sum, List<Integer> result, List<List<Integer>> results){
        if(sum==0 && root.left==null && root.right==null){
            List<Integer> tmp = new ArrayList<>(result);
            results.add(tmp);
            //return;
        }
        if(root.left!=null){
            result.add(root.left.val);
            dfs(root.left, sum-root.left.val, result, results);
            result.remove(result.size()-1);
        }
        if(root.right!=null){
            result.add(root.right.val);
            dfs(root.right, sum-root.right.val, result, results);
            result.remove(result.size()-1);
        }
    }
}
```

6.16. Populating Next Right Pointers in Each Node (Medium-> LeetCode No.116)

Given a binary tree

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

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to `NULL`.

Initially, all next pointers are set to `NULL`.

Note:

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

For example,

Given the following perfect binary tree,

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

After calling your function, the tree should look like:

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

Solution:

```
/**
 * 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;
      if(root!=null){
         if(root.right!=null){
            root.left.next = root.right;
            if(root.next!=null){
               root.right.next = root.next.left;
            }
         }
      }
      connect(root.left);
      connect(root.right);
   }
}
```

6.17.Sum Root to Leaf Numbers (Medium-> LeetCode No.129)

Given a binary tree containing digits from `0-9` only, each root-to-leaf path could represent a number.

An example is the root-to-leaf path `1->2->3` which represents the number `123`.

Find the total sum of all root-to-leaf numbers.

For example,

```
  1
 / \
2   3
```

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

The root-to-leaf path `1->3` represents the number `13`.

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

Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
   public int sumNumbers(TreeNode root) {
      if(root==null) return 0;
```

```
        int sum = 0;
        List<List<TreeNode>> results = new ArrayList<List<TreeNode>>();
        List<TreeNode> result = new ArrayList<>();
        result.add(root);
        dfs(root,result, results);
        for(List<TreeNode> list:results){
            StringBuilder sb = new StringBuilder();
            for(TreeNode node:list){
                sb.append(String.valueOf(node.val));
            }
            sum+=Integer.valueOf(sb.toString());
        }
        return sum;
    }
    public void dfs(TreeNode root, List<TreeNode> result, List<List<TreeNode>> results){
        if(root.left==null && root.right==null){
            List<TreeNode> tmp = new ArrayList<>(result);
            results.add(tmp);
            return;
        }

        if(root.left!=null){
            result.add(root.left);
            dfs(root.left, result, results);
            result.remove(result.size()-1);
        }
        if(root.right!=null){
            result.add(root.right);
            dfs(root.right, result, results);
            result.remove(result.size()-1);
        }
    }
}
```

6.18.Reconstruct Itinerary (Medium-> LeetCode No.332)

Given a list of airline tickets represented by pairs of departure and arrival airports `[from, to]`, reconstruct the itinerary in order. All of the tickets belong to a man who departs from `JFK`. Thus, the itinerary must begin with `JFK`.

Note:

1. If there are multiple valid itineraries, you should return the itinerary that has the smallest lexical order when read as a single string. For example, the itinerary `["JFK", "LGA"]` has a smaller lexical order than `["JFK", "LGB"]`.
2. All airports are represented by three capital letters (IATA code).
3. You may assume all tickets form at least one valid itinerary.

Example 1:

`tickets` = `[["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]`

Return `["JFK", "MUC", "LHR", "SFO", "SJC"]`.

Example 2:

`tickets` = `[["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],["ATL","SFO"]]`

Return `["JFK","ATL","JFK","SFO","ATL","SFO"]`.

Another possible reconstruction is `["JFK","SFO","ATL","JFK","ATL","SFO"]`. But it is larger in lexical order.

Solution:

```
public class Solution {
   Map<String, PriorityQueue<String>> map = new HashMap<>();
   LinkedList<String> queue = new LinkedList<>();
   public List<String> findItinerary(String[][] tickets) {
      for(String[] ticket:tickets){
         if(!map.containsKey(ticket[0])){
            PriorityQueue<String> q  = new PriorityQueue<>();
            map.put(ticket[0],q);
         }
         map.get(ticket[0]).add(ticket[1]);
      }
      dfs("JFK");
      return queue;
   }
   public void dfs(String s){
      PriorityQueue<String> q = map.get(s);
      while(q!=null && !q.isEmpty()){
         dfs(q.poll());
      }
      queue.addFirst(s);
```

```
    }
}
```

6.19. House Robber III (Medium-> LeetCode No.337)

The thief has found himself a new place for his thievery again. There is only one entrance to this area, called the "root." Besides the root, each house has one and only one parent house. After a tour, the smart thief realized that "all houses in this place forms a binary tree". It will automatically contact the police if two directly-linked houses were broken into on the same night.

Determine the maximum amount of money the thief can rob tonight without alerting the police.

Example 1:

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

Maximum amount of money the thief can rob = 3 + 3 + 1 = **7**.

Example 2:

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

Maximum amount of money the thief can rob = 4 + 5 = **9**.

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public int rob(TreeNode root) {
        if(root==null) return 0;
        int[] result = helper(root);
        return Math.max(result[0], result[1]);
    }
    public int[] helper(TreeNode root){
```

```
        int[] result = new int[2];
        if(root==null) return result;
        int[] left = helper(root.left);
        int[] right = helper(root.right);
        result[0] = root.val+left[1]+right[1];
        result[1] = Math.max(left[0],left[1])+Math.max(right[0],right[1]);
        return result;
    }
}
```

CHPATER 7: Breadth-First Search

7.1. Binary Tree Level Order Traversal (Easy-> LeetCode No.102)

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

For example:

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

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

return its level order traversal as:

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

Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> results = new ArrayList<List<Integer>>();
        if(root==null) return results;
        List<Integer> result = new ArrayList<>();

        LinkedList<TreeNode> current = new LinkedList<>();
        LinkedList<TreeNode> next = new LinkedList<>();
        current.add(root);
        while(!current.isEmpty()){
            TreeNode node = current.remove();
            if(node.left!=null) next.add(node.left);
            if(node.right!=null) next.add(node.right);
            result.add(node.val);
```

```
            if(current.isEmpty()){
                results.add(result);
                current = next;
                next = new LinkedList<>();
                result = new ArrayList<>();
            }
        }
        return results;
    }
}
```

7.2. Binary Tree Level Order Traversal (Easy-> LeetCode No.107)

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

For example:

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

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

return its bottom-up level order traversal as:

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

Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<List<Integer>> levelOrderBottom(TreeNode root) {
        List<List<Integer>> results = new ArrayList<List<Integer>>();
        if(root==null) return results;
        List<Integer> result = new ArrayList<>();
```

```
        LinkedList<TreeNode> current = new LinkedList<>();
        LinkedList<TreeNode> next = new LinkedList<>();
        current.add(root);

        Stack<List<Integer>> stack = new Stack<>();
        while(!current.isEmpty()){
            TreeNode node = current.remove();
            if(node.left!=null) next.add(node);
            if(node.right!=null) next.add(node);
            result.add(node.val);
            if(current.isEmpty()){
                current = next;
                next = new LinkedList<>();
                stack.push(result);
                result = new ArrayList<>();
            }
        }
        while(!stack.isEmpty()){
            results.add(stack.pop());
        }
        return results;
    }
}
```

7.3. Binary Tree Zigzag Level Order Traversal (Medium-> LeetCode No.103)

Given a binary tree, return the *zigzag level order* traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).

For example:

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

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

return its zigzag level order traversal as:

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

Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> results = new ArrayList<List<Integer>>();
        travel(results, root, 0);
        return results;
    }
    public void travel(List<List<Integer>> results, TreeNode root, int level){
        if(root==null) return;
        if(results.size()<=level) results.add(new ArrayList<>());
        if(level%2==0) results.get(level).add(root.val);
        else results.get(level).add(0,root.val);
        travel(results, root.left, level+1);
        travel(results, root.right,level+1);
    }
}
```

7.4. Minimum Height Trees (Medium-> LeetCode No.310)

For a undirected graph with tree characteristics, we can choose any node as the root. The result graph is then a rooted tree. Among all possible rooted trees, those with minimum height are called minimum height trees (MHTs). Given such a graph, write a function to find all the MHTs and return a list of their root labels.

Format

The graph contains `n` nodes which are labeled from `0` to `n - 1`. You will be given the number `n` and a list of undirected `edges` (each edge is a pair of labels).

You can assume that no duplicate edges will appear in `edges`. Since all edges are undirected, `[0, 1]` is the same as `[1, 0]` and thus will not appear together in `edges`.

Example 1:

Given `n = 4`, `edges = [[1, 0], [1, 2], [1, 3]]`

```
        0
        |
        1
       / \
      2   3
```

return `[1]`

Example 2:

Given `n = 6`, `edges = [[0, 3], [1, 3], [2, 3], [4, 3], [5, 4]]`

```
    0  1  2
     \ | /
       3
       |
       4
       |
       5
```

return `[3, 4]`

Hint:

1. How many MHTs can a graph have at most?

Solution:

```
public class Solution {
    public List<Integer> findMinHeightTrees(int n, int[][] edges) {
        List<List<Integer>> graph = new ArrayList<List<Integer>>();
```

```
        List<Integer> result = new ArrayList<>();
        if(n==1){
            result.add(0);
            return result;
        }
        for(int i=0;i<n;i++) graph.add(new ArrayList<>());
        int[] degree = new int[n]; // count the degree for each node
        // constrcut the graph and degree
        for(int i=0;i<edges.length;i++){
            graph.get(edges[i][0]).add(edges[i][1]);
            graph.get(edges[i][1]).add(edges[i][0]);
            degree[edges[i][0]]++;
            degree[edges[i][1]]++;
        }

        Queue<Integer> queue = new ArrayDeque<>();
        for(int i=0;i<n;i++){
            if(degree[i]==0) return result;  // Assume that it is a connected graph
            else if(degree[i]==1) queue.add(i);
        }
        while(!queue.isEmpty()){
            result = new ArrayList<>();
            int count = queue.size();
            for(int i=0;i<count;i++){
                int curr = queue.poll();
                result.add(curr);
                degree[curr]--;
                for(int j=0;j<graph.get(curr).size();j++){
                    int next = graph.get(curr).get(j);
                    if(degree[next]==0) continue;
                    if(degree[next]==2) queue.add(next);
                    degree[next]--;
                }
            }

        }
        return result;
    }
}
```

7.5. Symmetric Tree (Easy-> LeetCode No.101)

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

For example, this binary tree `[1,2,2,3,4,4,3]` is symmetric:

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

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

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

Note:

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

Subscribe to see which companies asked this question

Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public boolean isSymmetric(TreeNode root) {
        if(root==null) return true;
        return isSymmetric(root.left, root.right);
    }
    public boolean isSymmetric(TreeNode left, TreeNode right){
        if(left==null && right==null) return true;
        else if (left==null || right==null) return false;
        if(left.val!=right.val) return false;
        if(!isSymmetric(left.left, right.right)) return false;
        if(!isSymmetric(left.right,right.left)) return false;
```

```
        return true;
    }
}
```

7.6 Minimum Depth of Binary Tree (Easy->LeetCode No.111)

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public int minDepth(TreeNode root) {
        if(root==null) return 0;
        if(root.left==null) return minDepth(root.right)+1;
        else if(root.right==null) return minDepth(root.left)+1;
        else return Math.min(minDepth(root.right), minDepth(root.left))+1;
    }
}
```

7.7 Clone Graph (Medium->LeetCode No.133)

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

OJ's undirected graph serialization:

Nodes are labeled uniquely.

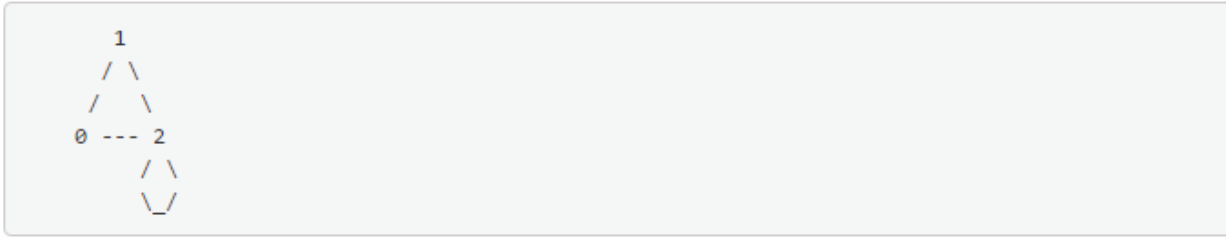
We use `#` as a separator for each node, and `,` as a separator for node label and each neighbor of the node.

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

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

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

Visually, the graph looks like the following:



Solution:

```
/**
 * Definition for undirected graph.
 * class UndirectedGraphNode {
 *     int label;
 *     List<UndirectedGraphNode> neighbors;
 *     UndirectedGraphNode(int x) { label = x; neighbors = new ArrayList<UndirectedGraphNode>(); }
 * };
 */
public class Solution {
    public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {
        if(node==null) return null;
        Map<UndirectedGraphNode,UndirectedGraphNode> map = new HashMap<>();
        LinkedList<UndirectedGraphNode> queue = new LinkedList<>();
        queue.add(node);
        UndirectedGraphNode nodecopy = new UndirectedGraphNode(node.label);
        map.put(node, nodecopy);
        while(!queue.isEmpty()){
```

```
        UndirectedGraphNode tmp = queue.remove();
        for(UndirectedGraphNode neighbor:tmp.neighbors){
          if(!map.containsKey(neighbor)){
            UndirectedGraphNode neighborcopy = new UndirectedGraphNode(neighbor.label);
            map.put(neighbor, neighborcopy);
            map.get(tmp).neighbors.add(neighborcopy);
            queue.add(neighbor);
          }else{
            map.get(tmp).neighbors.add(map.get(neighbor));
          }
        }
      }
      return nodecopy;
    }
}
```

7.8 Number of Islands (Medium->LeetCode No.200)

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

Example 1:

```
11110
11010
11000
00000
```

Answer: 1

Example 2:

```
11000
11000
00100
00011
```

Answer: 3

Solution:

```
public class Solution {
    public int numIslands(char[][] grid) {
        if(grid==null || grid.length==0 || grid[0].length==0) return 0;
        int m = grid.length;
        int n = grid[0].length;
        int count = 0;
```

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

7.9 Course Schedule II (Medium->LeetCode No.210)

There are a total of *n* courses you have to take, labeled from `0` to `n - 1`.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`

Given the total number of courses and a list of prerequisite **pairs**, return the ordering of courses you should take to finish all courses.

There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.

For example:

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

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

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

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

Note:

The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about how a graph is represented.

click to show more hints.

Hints:

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

Solution:

```
public class Solution {
   public int[] findOrder(int numCourses, int[][] prerequisites) {
      if(prerequisites==null) throw new IllegalArgumentException("unvalid argument");
```

```
        int len = prerequisites.length;
        if (len==0){
            int[] tmp = new int[numCourses];
            for(int i=0;i<numCourses;i++){
                tmp[i]=i;
            }
            return tmp;
        }
        int[] pCounter = new int[numCourses];
        for(int i=0;i<len;i++){
            pCounter[prerequisites[i][0]]++;
        }
        LinkedList<Integer> queue = new LinkedList<>();
        for(int i=0;i<numCourses;i++){
            if(pCounter[i]==0) queue.add(i);
        }

        int size = queue.size();
        int[] result = new int[numCourses];
        int index = 0;

        while(!queue.isEmpty()){
            int top = queue.remove();
            result[index++]=top;
            for(int i=0;i<len;i++){
                if(prerequisites[i][1]==top){
                    pCounter[prerequisites[i][0]]--;
                    if(pCounter[prerequisites[i][0]]==0){
                        size++;
                        queue.add(prerequisites[i][0]);
                    }
                }
            }
        }
        if(size==numCourses) return result;
        else return new int[0];
    }
}
```

7.10 Perfect Squares (Medium->LeetCode No.279)

Given a positive integer *n*, find the least number of perfect square numbers (for example, `1, 4, 9, 16, ...`) which sum to *n*.

For example, given *n* = `12`, return `3` because `12 = 4 + 4 + 4`; given *n* = `13`, return `2` because `13 = 4 + 9`.

Solution:

```
public class Solution {
   public int numSquares(int n) {
      if(n<=0) return 0;
      int[] record = new int[n+1];
      for(int i=1;i<=n;i++){
         record[i]=i;
         for(int j=(int)Math.sqrt(i);j*j*record[i]>i;j--){
            record[i]=Math.min(record[i-j*j]+1, record[i]);
         }
      }
      return record[n];
   }
}
```

7.11 Surrounded Regions (Medium->Leetcode No.130)

Given a 2D board containing `'X'` and `'O'`, capture all regions surrounded by `'X'`.

A region is captured by flipping all `'O'`s into `'X'`s in that surrounded region.

For example,

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

After running your function, the board should be:

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

Solution:

```
public class Solution {
        // use a queue to do BFS
        private Queue<Integer> queue = new LinkedList<Integer>();

        public void solve(char[][] board) {
                if (board == null || board.length == 0)
                        return;

                int m = board.length;
                int n = board[0].length;

                // merge O's on left & right boarder
                for (int i = 0; i < m; i++) {
```

```
                if (board[i][0] == 'O') {
                        bfs(board, i, 0);
                }

                if (board[i][n - 1] == 'O') {
                        bfs(board, i, n - 1);
                }
        }

        // merge O's on top & bottom boarder
        for (int j = 0; j < n; j++) {
                if (board[0][j] == 'O') {
                        bfs(board, 0, j);
                }

                if (board[m - 1][j] == 'O') {
                        bfs(board, m - 1, j);
                }
        }

        // process the board
        for (int i = 0; i < m; i++) {
                for (int j = 0; j < n; j++) {
                        if (board[i][j] == 'O') {
                                board[i][j] = 'X';
                        } else if (board[i][j] == '#') {
                                board[i][j] = 'O';
                        }
                }
        }
}

private void bfs(char[][] board, int i, int j) {
        int n = board[0].length;

        // fill current first and then its neighbors
        fillCell(board, i, j);

        while (!queue.isEmpty()) {
                int cur = queue.poll();
                int x = cur / n;
                int y = cur % n;

                fillCell(board, x - 1, y);
                fillCell(board, x + 1, y);
                fillCell(board, x, y - 1);
                fillCell(board, x, y + 1);
        }
```

```
        }

        private void fillCell(char[][] board, int i, int j) {
                int m = board.length;
                int n = board[0].length;
                if (i < 0 || i >= m || j < 0 || j >= n || board[i][j] != 'O')
                        return;

                // add current cell is queue & then process its neighbors in bfs
                queue.offer(i * n + j);
                board[i][j] = '#';
        }
}
```

CHAPTER 8: Binary Indexed Tree/Segment Tree

8.1 Range Sum Query-Mutable (Medium-> LeetCode No.307)

Given an integer array *nums*, find the sum of the elements between indices *i* and *j* (*i* ≤ *j*), inclusive.

The *update(i, val)* function modifies *nums* by updating the element at index *i* to *val*.

Example:

```
Given nums = [1, 3, 5]

sumRange(0, 2) -> 9
update(1, 2)
sumRange(0, 2) -> 8
```

Note:

1. The array is only modifiable by the *update* function.
2. You may assume the number of calls to *update* and *sumRange* function is distributed evenly.

Solution

```
class TreeNode{
   int start;
   int end;
   int sum;
   TreeNode left;
   TreeNode right;

   public TreeNode(int start, int end, int sum){
      this.start = start;
      this.end = end;
      this.sum = sum;
   }
   public TreeNode(int start, int end){
      this.start = start;
      this.end = end;
      this.sum = 0;
   }
}
public class NumArray {
   TreeNode root = null;

   public NumArray(int[] nums) {
      if(nums==null || nums.length==0) return;
      this.root = buildTree(nums,0,nums.length-1);
   }
```

```
    void update(int i, int val) {
        updateHelper(root, i, val);
    }

    public void updateHelper(TreeNode root, int i, int val){
        if(root==null) return;
        int mid = root.start+(root.end-root.start)/2;
        if(root.start==root.end && root.start==i){
            root.sum = val;
            return;
        }else if(i<=mid) updateHelper(root.left, i,val);
        else updateHelper(root.right, i,val);
        root.sum = root.left.sum+root.right.sum;
    }

    public int sumRange(int i, int j) {
        return sumRangeHelper(root, i,j);
    }

    public int sumRangeHelper(TreeNode root, int i, int j){
        if(root==null || j<root.start || i > root.end ||i>j )return 0;
        if(i<=root.start && j>=root.end) return root.sum;
        int mid = root.start + (root.end-root.start)/2;
        int result = sumRangeHelper(root.left, i, Math.min(mid, j))
                +sumRangeHelper(root.right, Math.max(mid+1, i), j);

        return result;
    }

    public TreeNode buildTree(int[] nums, int i, int j){
        if(nums==null || nums.length==0 || i> j) return null;
        if(i==j) return new TreeNode(i,j,nums[i]);
        int mid = i+(j-i)/2;
        TreeNode current = new TreeNode(i,j);
        current.left = buildTree(nums, i, mid);
        current.right = buildTree(nums, mid+1,j);
        current.sum = current.left.sum+current.right.sum;
        return current;
    }
}


// Your NumArray object will be instantiated and called as such:
// NumArray numArray = new NumArray(nums);
// numArray.sumRange(0, 1);
// numArray.update(1, 10);
// numArray.sumRange(1, 2);
```

CHAPTER 9: Brainteaser

9.1 Bulb Switcher (Medium-> LeetCode No.319)

There are *n* bulbs that are initially off. You first turn on all the bulbs. Then, you turn off every second bulb. On the third round, you toggle every third bulb (turning on if it's off or turning off if it's on). For the *i*th round, you toggle every *i* bulb. For the *n*th round, you only toggle the last bulb. Find how many bulbs are on after *n* rounds.

Example:

```
Given n = 3.

At first, the three bulbs are [off, off, off].
After first round, the three bulbs are [on, on, on].
After second round, the three bulbs are [on, off, on].
After third round, the three bulbs are [on, off, off].

So you should return 1, because there is only one bulb is on.
```

Solution

那么最后我们发现五次遍历后，只有1号和4号锁是亮的，而且很巧的是它们都是平方数，是巧合吗，还是其中有什么玄机。我们仔细想想，对于第n个灯泡，只有当次数是n的因子的之后，才能改变灯泡的状态，即n能被当前次数整除，比如当n为36时，它的因数有(1,36), (2,18), (3,12), (4,9), (6,6), 可以看到前四个括号里成对出现的因数各不相同，括号中前面的数改变了灯泡状态，后面的数又变回去了，等于锁的状态没有发生变化，只有最后那个(6,6)，在次数6的时候改变了一次状态，没有对应其它的状态能将其变回去了，所以锁就一直是打开状态的。所以所有平方数都有这么一个相等的因数对，即所有平方数的灯泡都将会是打开的状态。

```
public class Solution {
    public int bulbSwitch(int n) {
        return (int)Math.sqrt(n);
    }
}
```

9.2 Nim Game (Easy->Leetcode No.292)

You are playing the following Nim Game with your friend: There is a heap of stones on the table, each time one of you take turns to remove 1 to 3 stones. The one who removes the last stone will be the winner. You will take the first turn to remove the stones.

Both of you are very clever and have optimal strategies for the game. Write a function to determine whether you can win the game given the number of stones in the heap.

For example, if there are 4 stones in the heap, then you will never win the game: no matter 1, 2, or 3 stones you remove, the last stone will always be removed by your friend.

Hint:

1. If there are 5 stones in the heap, could you figure out a way to remove the stones such that you will always be the winner?

Solution and analysis: if we use the dynamic programming, obviously, we have the recursive formula for DP is dp[n] = !dp[n-1] || !dp[n-2] || !dp[n-3], where dp[n] is a boolean variable, indicating whether I can win or not, so that we can deduce the simple solution as follows:

```
public class Solution {
```

```
    public boolean canWinNim(int n) {
        return !(n%4==0);
    }
}
```

CHAPTER 10: Binary Search Tree

10.1 Contains Duplicates III (Medium-> LeetCode No.220)

Given an array of integers, find out whether there are two distinct indices *i* and *j* in the array such that the difference between **nums[i]** and **nums[j]** is at most *t* and the difference between *i* and *j* is at most *k*.

Solution

```
public class Solution {
    public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {
        if(nums==null || nums.length<2 || k<0 || t<0) return false;
        TreeSet<Long> set = new TreeSet<>();
        for(int i=0;i<nums.length;i++){
            long curr = (long)nums[i];
            long leftBoundary = (long)nums[i]-t;
            long rightBoundary = (long)nums[i]+t+1;
            SortedSet<Long> sub = set.subSet(leftBoundary,rightBoundary);
            if(sub.size()>0) return true;
            set.add(curr);
            if(i>=k) set.remove((long)nums[i-k]);
        }
        return false;
    }
}
```

CHAPTER 11: Trie

11.1 Implement Trie (Prefix Tree) (Medium-> LeetCode No.208)

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

Note:

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

Subscribe to see which companies asked this question

Solution: be careful about that the root does not store any element, which means the word should be stored from the second level.

```
class TrieNode {
    char c;
    Map<Character, TrieNode> children = new HashMap<>();
    boolean isLeaf;
    // Initialize your data structure here.
    public TrieNode() {}
    public TrieNode(char c) {this.c = c;}
```

```
}

public class Trie {
   private TrieNode root;
   public Trie() {
      root = new TrieNode();
   }
   // Inserts a word into the trie.
   public void insert(String word) {
      Map<Character, TrieNode> children = root.children;
      for(int i=0;i<word.length();i++){
         char c = word.charAt(i);
         TrieNode node;
         if(children.containsKey(c)) node = children.get(c);
         else{
            node = new TrieNode(c);
            children.put(c,node);
         }
         children = node.children;
         if(i==word.length()-1) node.isLeaf=true;
      }
   }

   // Returns if the word is in the trie.
   public boolean search(String word) {
      TrieNode node = searchNode(word);
      if(node!=null && node.isLeaf) return true;
      else return false;
   }

   public TrieNode searchNode(String word){
      Map<Character, TrieNode> children = root.children;
      TrieNode node=null;
      for(int i=0;i<word.length();i++){
         char c = word.charAt(i);
         if(children.containsKey(c)){
            node = children.get(c);
            children = node.children;
         }else return null;
      }
      return node;
   }

   // Returns if there is any word in the trie
   // that starts with the given prefix.
   public boolean startsWith(String prefix) {
      if(searchNode(prefix)==null) return false;
      else return true;
```

```
    }
}

// Your Trie object will be instantiated and called as such:
// Trie trie = new Trie();
// trie.insert("somestring");
// trie.search("key");
```

11.2 Add and Search Word -Data Structure Design (Medium-> Leetcode No.211)

Design a data structure that supports the following two operations:

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

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

For example:

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

Note:

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

Solution

```
class TrieNode{
    char c;
    Map<Character, TrieNode> children = new HashMap<>();
    boolean isLeaf;
    public TrieNode(){}
    public TrieNode(char c){this.c = c;}
}
public class WordDictionary {
    TrieNode root;
    public WordDictionary(){this.root = new TrieNode();}
    // Adds a word into the data structure.
    public void addWord(String word) {
        Map<Character, TrieNode> children = root.children;
        for(int i=0;i<word.length();i++){
            char c = word.charAt(i);
            TrieNode node = null;
            if(children.containsKey(c)) node = children.get(c);
            else{
                node = new TrieNode(c);
                children.put(c,node);
            }
```

```
            children = node.children;
            if(i==word.length()-1) node.isLeaf=true;
        }
    }

    // Returns if the word is in the data structure. A word could
    // contain the dot character '.' to represent any one letter.
    public boolean search(String word) {
        return dfs(root.children, word, 0);
    }
    public boolean dfs(Map<Character, TrieNode> children, String word, int start){
        if(start==word.length()){
            if(children.size()==0) return true;
            else return false;
        }
        char c = word.charAt(start);
        if(children.containsKey(c)){
            if(start==word.length() && children.get(c).isLeaf) return true;
            return dfs(children.get(c).children, word, start+1);
        }else if(c=='.'){
            boolean result = false;
            for(Map.Entry<Character, TrieNode> child:children.entrySet()){
                if(start==word.length()-1 && child.getValue().isLeaf) return true;
                if(dfs(child.getValue().children, word, start+1)) result = true;
            }
            return result;
        }else return false;
    }
}

// Your WordDictionary object will be instantiated and called as such:
// WordDictionary wordDictionary = new WordDictionary();
// wordDictionary.addWord("word");
// wordDictionary.search("pattern");
```

CHAPTER 12: Design

12.1 Min Stack (Easy->Leetcode No.155)

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

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

Example:

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

Solution:

```
public class MinStack {
    private Stack<Integer> stack = new Stack<>();
    private Stack<Integer> minStack = new Stack<>();
    /** initialize your data structure here. */
    public MinStack() {

    }

    public void push(int x) {
        if(minStack.isEmpty() || x<=minStack.peek()) minStack.push(x);
        stack.push(x);
    }

    public void pop() {
        if(stack.peek().equals(minStack.peek())) minStack.pop();
        stack.pop();
    }

    public int top() {
        return stack.peek();
    }
```

```
    public int getMin() {
        return minStack.peek();
    }
}

/**
 * Your MinStack object will be instantiated and called as such:
 * MinStack obj = new MinStack();
 * obj.push(x);
 * obj.pop();
 * int param_3 = obj.top();
 * int param_4 = obj.getMin();
 */
```

12.2 Binary Search Tree Iterator (Medium->Leetcode No.173)

Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST.

Calling `next()` will return the next smallest number in the BST.

Note: `next()` and `hasNext()` should run in average O(1) time and uses O(*h*) memory, where *h* is the height of the tree.

Solution

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */

public class BSTIterator {
    private Stack<TreeNode> stack = null;
    public BSTIterator(TreeNode root) {
        stack = new Stack<>();
        while(root!=null){
            stack.push(root);
            root = root.left;
        }
    }

    /** @return whether we have a next smallest number */
    public boolean hasNext() {
        return !stack.isEmpty();
    }

    /** @return the next smallest number */
    public int next() {
```

```
        TreeNode node = stack.pop();
        int result = node.val;
        if(node.right!=null){
            node = node.right;
            while(node!=null){
                stack.push(node);
                node = node.left;
            }
        }
        return result;
    }
}

/**
 * Your BSTIterator will be called like this:
 * BSTIterator i = new BSTIterator(root);
 * while (i.hasNext()) v[f()] = i.next();
 */
```

12.3 Implement Stack using Queues (Easy ->leetcode No.225)

Implement the following operations of a stack using queues.

- push(x) -- Push element x onto stack.
- pop() -- Removes the element on top of the stack.
- top() -- Get the top element.
- empty() -- Return whether the stack is empty.

Notes:

- You must use *only* standard operations of a queue -- which means only `push to back`, `peek/pop from front`, `size`, and `is empty` operations are valid.
- Depending on your language, queue may not be supported natively. You may simulate a queue by using a list or deque (double-ended queue), as long as you use only standard operations of a queue.
- You may assume that all operations are valid (for example, no pop or top operations will be called on an empty stack).

Update (2015-06-11):

The class name of the **Java** function had been updated to **MyStack** instead of Stack.

Solution

```
class MyStack {
    private Queue<Integer> queue1 = new LinkedList<>();
    private Queue<Integer> queue2 = new LinkedList<>();
    // Push element x onto stack.
    public void push(int x) {
        if(empty()) queue1.offer(x);
        else{
            if(queue1.size()>0){
                queue2.offer(x);
                int size = queue1.size();
                while(size>0){
```

```
            queue2.offer(queue1.poll());
            size--;
          }
        }else if(queue2.size()>0){
          queue1.offer(x);
          int size = queue2.size();
          while(size>0){
            queue1.offer(queue2.poll());
            size--;
          }
        }
      }

  // Removes the element on top of the stack.
  public void pop() {
    if(queue1.size()>0) queue1.poll();
    else if(queue2.size()>0) queue2.poll();
  }

  // Get the top element.
  public int top() {
    if(queue1.size()>0) return queue1.peek();
    else if(queue2.size()>0) return queue2.peek();
    return 0;
  }

  // Return whether the stack is empty.
  public boolean empty() {
    return queue1.size()==0 && queue2.size()==0;
  }
}
```

12.4 Implement Queue using Stacks (Easy->Leetcode No.232)

Implement the following operations of a queue using stacks.

- push(x) -- Push element x to the back of queue.
- pop() -- Removes the element from in front of queue.
- peek() -- Get the front element.
- empty() -- Return whether the queue is empty.

Notes:

- You must use *only* standard operations of a stack -- which means only `push to top`, `peek/pop from top`, `size`, and `is empty` operations are valid.
- Depending on your language, stack may not be supported natively. You may simulate a stack by using a list or deque (double-ended queue), as long as you use only standard operations of a stack.
- You may assume that all operations are valid (for example, no pop or peek operations will be called on an empty queue).

```
class MyQueue {
   private Stack<Integer> value = new Stack<>();
   private Stack<Integer> tmp = new Stack<>();
   // Push element x to the back of queue.
   public void push(int x) {
      if(value.isEmpty()) value.push(x);
      else{
         while(!value.isEmpty()) tmp.push(value.pop());
         value.push(x);
         while(!tmp.isEmpty()) value.push(tmp.pop());
      }
   }

   // Removes the element from in front of queue.
   public void pop() {
      value.pop();
   }

   // Get the front element.
   public int peek() {
      return value.peek();
   }

   // Return whether the queue is empty.
   public boolean empty() {
      return value.isEmpty();
   }
}
```

12.5 Peeking Iterator (Medium->Leetcode No.284)

Given an Iterator class interface with methods: `next()` and `hasNext()`, design and implement a PeekingIterator that support the `peek()` operation -- it essentially peek() at the element that will be returned by the next call to next().

Here is an example. Assume that the iterator is initialized to the beginning of the list: `[1, 2, 3]`.

Call `next()` gets you 1, the first element in the list.

Now you call `peek()` and it returns 2, the next element. Calling `next()` after that ***still*** return 2.

You call `next()` the final time and it returns 3, the last element. Calling `hasNext()` after that should return false.

Hint:

1. Think of "looking ahead". You want to cache the next element.
2. Is one variable sufficient? Why or why not?
3. Test your design with call order of `peek()` before `next()` vs `next()` before `peek()`.
4. For a clean implementation, check out Google's guava library source code.

Follow up: How would you extend your design to be generic and work with all types, not just integer?

Solution:

```
// Java Iterator interface reference:
// https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html
class PeekingIterator implements Iterator<Integer> {
   private Integer peek;
   private Iterator<Integer> it;
         public PeekingIterator(Iterator<Integer> iterator) {
            // initialize any member here.
            if(iterator==null) return;
            it = iterator;
            peek = it.hasNext()?it.next():null;
         }

   // Returns the next element in the iteration without advancing the iterator.
         public Integer peek() {
      return peek;
         }

         // hasNext() and next() should behave the same as in the Iterator interface.
         // Override them if needed.
         @Override
         public Integer next() {
            Integer tmp = peek;
            peek = it.hasNext()?it.next():null;
            return tmp;
         }

         @Override
         public boolean hasNext() {
            return peek!=null;
         }
}
```

12.6 Flatten Nested List Iterator (Medium->Leetcode No.341)

Given a nested list of integers, implement an iterator to flatten it.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

Example 1:

Given the list `[[1,1],2,[1,1]]`,

By calling *next* repeatedly until *hasNext* returns false, the order of elements returned by *next* should be: `[1,1,2,1,1]`.

Example 2:

Given the list `[1,[4,[6]]]`,

By calling *next* repeatedly until *hasNext* returns false, the order of elements returned by *next* should be: `[1,4,6]`.

Solution:

```
/**
 * // This is the interface that allows for creating nested lists.
 * // You should not implement it, or speculate about its implementation
 * public interface NestedInteger {
 *
 *     // @return true if this NestedInteger holds a single integer, rather than a nested list.
 *     public boolean isInteger();
 *
 *     // @return the single integer that this NestedInteger holds, if it holds a single integer
 *     // Return null if this NestedInteger holds a nested list
 *     public Integer getInteger();
 *
 *     // @return the nested list that this NestedInteger holds, if it holds a nested list
 *     // Return null if this NestedInteger holds a single integer
 *     public List<NestedInteger> getList();
 * }
 */
public class NestedIterator implements Iterator<Integer> {
    private Stack<NestedInteger> stack = new Stack<>();
    public NestedIterator(List<NestedInteger> nestedList) {
        if(nestedList==null) return;
        for(int i=nestedList.size()-1;i>=0;i--) stack.push(nestedList.get(i));
    }

    @Override
    public Integer next() {
        return stack.pop().getInteger();
```

```
    }

    @Override
    public boolean hasNext() {
        while(!stack.isEmpty()){
            NestedInteger tmp = stack.peek();
            if(tmp.isInteger()) return true;
            stack.pop();
            for(int i=tmp.getList().size()-1;i>=0;i--) stack.push(tmp.getList().get(i));
        }
        return false;
    }
}

/**
 * Your NestedIterator object will be instantiated and called as such:
 * NestedIterator i = new NestedIterator(nestedList);
 * while (i.hasNext()) v[f()] = i.next();
 */
```

CHAPTER 13: Backtracking

13.1 Combination Sum (Medium->LeetCode 39)

Given a set of candidate numbers (***C***) and a target number (***T***), find all unique combinations in ***C*** where the candidate numbers sums to ***T***.

The **same** repeated number may be chosen from ***C*** unlimited number of times.

Note:

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

For example, given candidate set `[2, 3, 6, 7]` and target `7`,

A solution set is:

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

Solution:

```
public class Solution {
    public List<List<Integer>> combinationSum(int[] candidates, int target) {
        List<List<Integer>> results = new ArrayList<List<Integer>>();
        if(candidates==null || candidates.length==0) return results;
        List<Integer> result = new ArrayList<>();
        Arrays.sort(candidates);
        dfs(candidates, target,0, result, results);

        HashSet<List<Integer>> set = new HashSet<>(results);
        results.clear();
        results.addAll(set);
        return results;
    }

    public void dfs(int[] candidates,  int target, int j, List<Integer> result, List<List<Integer>> results){
        if(target==0){
            List<Integer> tmp = new ArrayList<>(result);
            results.add(tmp);
            return;
        }
        for(int i=j;i<candidates.length;i++){
            if(target<candidates[i]) return;
            result.add(candidates[i]);
            dfs(candidates, target-candidates[i],i,result,results);
            result.remove(result.size()-1);
```

```
    }
  }
}
```

13.2 Combination Sum II (Medium->LeetCode 40)

Given a collection of candidate numbers (***C***) and a target number (***T***), find all unique combinations in ***C*** where the candidate numbers sums to ***T***.

Each number in ***C*** may only be used **once** in the combination.

Note:

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

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

A solution set is:

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

Solution:

```
public class Solution {
    public List<List<Integer>> combinationSum2(int[] candidates, int target) {
        List<List<Integer>> results = new ArrayList<List<Integer>>();
        if(candidates==null || candidates.length==0) return results;
        List<Integer> result = new ArrayList<>();
        Arrays.sort(candidates);
        dfs(candidates, target, 0, result, results);
        Set<List<Integer>> set = new HashSet<>(results);
        results.clear();
        results.addAll(set);
        return results;
    }
    public void dfs(int[] candidates, int target, int j, List<Integer> result, List<List<Integer>> results){
        if(target==0){
            List<Integer> tmp = new ArrayList<>(result);
            results.add(tmp);
            return;
        }

        for(int i=j;i<candidates.length;i++){
```

```
            if(target<candidates[i]) continue;
            result.add(candidates[i]);
            dfs(candidates, target-candidates[i], i+1, result, results);
            result.remove(result.size()-1);
        }
    }
}
```

13.3 Permutations (Medium->LeetCode 46)

Given a collection of **distinct** numbers, return all possible permutations.

For example,

`[1,2,3]` have the following permutations:

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

Solution:

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

    public void permutation(int[] nums, List<List<Integer>> results, int start){
        if(start==nums.length){
            List<Integer> tmp = convertArrayToList(nums);
            results.add(tmp);
            return;
        }

        for(int i=start; i<nums.length;i++){
            swap(nums, start, i);
            permutation(nums,results, start+1);
            swap(nums,start, i);
        }
    }
    public List<Integer> convertArrayToList(int[] nums){
        List<Integer> result = new ArrayList<>();
        for(int i:nums){
```

```
            result.add(i);
        }
        return result;
    }
    public void swap(int[] nums, int i, int j){
        int tmp = nums[i];
        nums[i] = nums[j];
        nums[j] = tmp;
    }
}
```

13.4 Permutations II (Medium->LeetCode 47)

Given a collection of numbers that might contain duplicates, return all possible unique permutations.

For example,

`[1,1,2]` have the following unique permutations:

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

Solution:

```
public class Solution {
    public List<List<Integer>> permuteUnique(int[] nums) {
        List<List<Integer>> results = new ArrayList<List<Integer>>();
        if(nums==null || nums.length==0) return results;
        permutation(nums, results, 0);
        Set<List<Integer>> set = new HashSet<>(results);
        results.clear();
        results.addAll(set);
        return results;
    }
    public void permutation(int[] nums, List<List<Integer>> results, int start){
        if(start==nums.length){
            List<Integer> tmp = convertArrayToList(nums);
            results.add(tmp);
            return;
        }

        for(int i=start;i<nums.length;i++){
            swap(nums, start, i);
            permutation(nums, results, start+1);
            swap(nums,start, i);
        }
    }
    public List<Integer> convertArrayToList(int[] nums){
```

```
        List<Integer> result = new ArrayList<>();
        for(int i:nums){
            result.add(i);
        }
        return result;
    }
    public void swap(int[] nums, int i, int j){
        int tmp = nums[i];
        nums[i] = nums[j];
        nums[j] = tmp;
    }
}
```

13.5 Permutation Sequence (Medium->LeetCode 60)

The set `[1,2,3,…,n]` contains a total of *n*! unique permutations.

By listing and labeling all of the permutations in order,

We get the following sequence (ie, for *n* = 3):

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

Given *n* and *k*, return the k^{th} permutation sequence.

Note: Given *n* will be between 1 and 9 inclusive.

Solution: 1) if we use the purely bruth force solution, the solution will exceed in time and 2) we have to find the mathematical law to handle it.

发现数学规律。

首先先捋捋这道题要干啥，给了我们n还有k，在数列 1，2，3，...，n构建的全排列中，返回第k个排列。

题目告诉我们：对于n个数可以有n!种排列；那么n-1个数就有(n-1)!种排列。

那么对于n位数来说，如果除去最高位不看，后面的n-1位就有 (n-1)!种排列。

所以，还是对于n位数来说，每一个不同的最高位数，后面可以拼接(n-1)!种排列。

所以你就可以看成是按照每组(n-1)!个这样分组。

利用 k/(n-1)! 可以取得最高位在数列中的index。这样第k个排列的最高位就能从数列中的index位取得，此时还要把这个数从数列中删除。

然后，新的k就可以有k%(n-1)!获得。循环n次即可。

同时，为了可以跟数组坐标对其，令k先--。

```
public class Solution {
   public String getPermutation(int n, int k) {
      StringBuilder sb = new StringBuilder();
      List<Integer> numbers = new ArrayList<>();
      int[] factorial = new int[n+1];

      int sum = 1;
      factorial[0] = 1;
      for(int i=1;i<=n;i++){
         numbers.add(i);
         sum*=i;
         factorial[i]=sum;
      }

      k--;
      for(int i=1;i<=n;i++){
         int index = k/factorial[n-i];
         sb.append(String.valueOf(numbers.get(index)));
         numbers.remove(index);
         k-=index*factorial[n-i];
      }
      return String.valueOf(sb);
   }
}
```

13.6 Combinations (Medium->LeetCode 77)

Given two integers *n* and *k*, return all possible combinations of *k* numbers out of 1 ... *n*.

For example,

If *n* = 4 and *k* = 2, a solution is:

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

Solution:

```
public class Solution {
    public List<List<Integer>> combine(int n, int k) {
        List<List<Integer>> results = new ArrayList<List<Integer>>();
        if(n<=0 || n<k) return results;
        List<Integer> result = new ArrayList<>();
        dfs(n,k,result,results,1);
        return results;
    }

    public void dfs(int n,int k, List<Integer> result, List<List<Integer>> results, int start){
        if(result.size()==k){
            List<Integer> tmp = new ArrayList<>(result);
            results.add(tmp);
            return;
        }

        for(int i=start;i<=n;i++){
            result.add(i);
            dfs(n, k, result,results, i+1);
            result.remove(result.size()-1);
        }
    }
}
```

13.7 Subsets (Medium->LeetCode 78)

Given a set of distinct integers, *nums*, return all possible subsets.

Note: The solution set must not contain duplicate subsets.

For example,

If ***nums*** = `[1,2,3]`, a solution is:

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

Solution:

```
public class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> results = new ArrayList<List<Integer>>();
        if(nums==null || nums.length==0) return results;
        List<Integer> result = new ArrayList<>();
        Arrays.sort(nums);
        results.add(new ArrayList<Integer>());
        dfs(nums, result, results, 0);
        return results;
    }

    public void dfs(int[] nums, List<Integer> result, List<List<Integer>> results, int start){
        for(int i=start;i<nums.length;i++){
            result.add(nums[i]);
            results.add(new ArrayList<>(result));
            dfs(nums, result, results, i+1);
            result.remove(result.size()-1);
        }
    }
}
```

13.8 Word Search (Medium->LeetCode 79)

Given a 2D board and a word, find if the word exists in the grid.

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

For example,

Given **board** =

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

word = `"ABCCED"`, -> returns `true`,

word = `"SEE"`, -> returns `true`,

word = `"ABCB"`, -> returns `false`.

Solution:

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

   public boolean dfs(char[][] board, String word, int i, int j, int index){
      if(i<0 || i>=board.length || j<0 || j>=board[0].length) return false;
      if(board[i][j]==word.charAt(index)){
         char tmp = board[i][j];
         board[i][j]='#';
         if(index==word.length()-1) return true;
         else if(dfs(board, word, i+1,j,index+1) ||dfs(board, word, i-1,j,index+1) || dfs(board, word,
i,j+1,index+1) || dfs(board, word, i,j-1,index+1)) return true;
         board[i][j] = tmp;
      }
      return false;
```

```
    }
}
```

13.9 Gray Code(Medium->LeetCode 89)

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

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

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

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

Note:

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

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

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

Solution: be careful about that two successive values differ in only one bit.

```
public class Solution {
    public List<Integer> grayCode(int n) {
        List<Integer> result = new ArrayList<>();
        if(n==0){
            result.add(0);
            return result;
        }
        result = grayCode(n-1);
        int tmp = 1<<(n-1);
        for(int i=result.size()-1;i>=0;i--) result.add(tmp+result.get(i));
        return result;
    }
}
```

13.10 Subsets II (Medium->LeetCode 90)

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

Note: The solution set must not contain duplicate subsets.

For example,

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

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

Solution:

```
public class Solution {
    public List<List<Integer>> subsetsWithDup(int[] nums) {
        List<List<Integer>> results = new ArrayList<List<Integer>>();
        if(nums==null || nums.length==0) return results;
        List<Integer> result = new ArrayList<>();
        Arrays.sort(nums);
        results.add(new ArrayList<Integer>());
        dfs(nums, result, results, 0);
        HashSet<List<Integer>> set = new HashSet<>(results);
        results.clear();
        results.addAll(set);
        return results;
    }
    public void dfs(int[] nums, List<Integer> result, List<List<Integer>> results, int start){
        for(int i=start;i<nums.length;i++){
            //if(i>start && nums[i]==nums[i-1]) continue;
            result.add(nums[i]);
            results.add(new ArrayList<>(result));
            dfs(nums, result, results, i+1);
            result.remove(result.size()-1);
        }
    }
}
```

13.11 Restore IP Address (Medium->LeetCode 93)

Given a string containing only digits, restore it by returning all possible valid IP address combinations.

For example:

Given `"25525511135"`,

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

Solution:

```
public class Solution {
    public List<String> restoreIpAddresses(String s) {
        List<List<String>> results = new ArrayList<List<String>>();
        List<String> result = new ArrayList<>();
        if(s==null || s.length()==0) return result;
        dfs(results,result,s, 0);

        List<String> finalret = new ArrayList<>();
        for(List<String> l: results){
            StringBuilder sb = new StringBuilder();
            for(String tmp:l){
                sb.append(tmp+".");
            }
            sb.setLength(sb.length()-1);
            finalret.add(sb.toString());
        }
        return finalret;
    }
    public void dfs(List<List<String>> results, List<String> result, String s, int start){
        // if already get 4 numbers, but s is not consumed, return
        if(result.size()>=4 && start!=s.length()) return;
        // make sure result's size+remaining string's length>=4
        if(result.size()+s.length()-start+1<4) return;
        if(result.size()==4 && start==s.length()){
            List<String> tmp = new ArrayList<>(result);
            results.add(tmp);
            return;
        }
        for(int i=1;i<=3;i++){
            if(start+i>s.length()) break;
            String sub = s.substring(start, start+i);
            // handle the case like 001, if length>1 and first char=0, ignore the case
            if(i>1 && s.charAt(start)=='0')  break;
            // make sure each number <=255
            if(Integer.valueOf(sub)>255) break;
            result.add(sub);
            dfs(results, result, s, start+i);
            result.remove(result.size()-1);
        }
```

```
    }
}
```

13.12 Palindrome Partitioning(Medium->LeetCode 131)

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

Return all possible palindrome partitioning of *s*.

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

Return

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

Subscribe to see which companies asked this question

Solution

```
public class Solution {
    public List<List<String>> partition(String s) {
        List<List<String>> results = new ArrayList<List<String>>();
        if(s==null || s.length()==0) return results;
        List<String> result = new ArrayList<>();
        dfs(results, result, s, 0);
        return results;
    }

    private void dfs(List<List<String>> results, List<String> result, String s, int start){
        if(s.length()==start){
            List<String> tmp = new ArrayList<>(result);
            results.add(tmp);
            return;
        }
        for(int i=start+1;i<=s.length();i++){
            String str = s.substring(start, i);
            if(isPalindrome(str)){
                result.add(str);
                dfs(results, result, s, i);
                result.remove(result.size()-1);
            }
        }
    }
    private boolean isPalindrome(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;
  }
}
```

13.13 Combinations III (Medium->LeetCode 216)

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

Example 1:

Input: ***k*** = 3, ***n*** = 7

Output:

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

Example 2:

Input: ***k*** = 3, ***n*** = 9

Output:

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


Solution:

```
public class Solution {
  public List<List<Integer>> combinationSum3(int k, int n) {
    List<List<Integer>> results = new ArrayList<List<Integer>>();
    if(k<=0) return results;
    List<Integer> result = new ArrayList<>();
    dfs(k, n , result, results, 1);
    return results;
  }
  public void dfs(int k, int n, List<Integer> result, List<List<Integer>> results, int j){
    if(result.size()==k && n==0){
      results.add(new ArrayList<Integer>(result));
      return;
    }
    for(int i=j;i<=9;i++){
      if(n<i) break;
      if(result.size()>k) break;
      result.add(i);
      dfs(k,n-i,result, results,i+1);
```

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

13.14 Count Numbers With Uniques Digits (Medium->LeetCode 357)

Given a **non-negative** integer n, count all numbers with unique digits, x, where $0 \le x < 10^n$.

Example:

Given n = 2, return 91. (The answer should be the total numbers in the range of $0 \le x < 100$, excluding `[11,22,33,44,55,66,77,88,99]`)

Hint:

1. A direct way is to use the backtracking approach.
2. Backtracking should contains three states which are (the current number, number of steps to get that number and a bitmask which represent which number is marked as visited so far in the current number). Start with state (0,0,0) and count all valid number till we reach number of steps equals to 10^n.
3. This problem can also be solved using a dynamic programming approach and some knowledge of combinatorics.
4. Let f(k) = count of numbers with unique digits with length equals k.
5. f(1) = 10, ..., f(k) = 9 * 9 * 8 * ... (9 - k + 2) [The first factor is 9 because a number cannot start with 0].

Solution

排列组合题。

设i为长度为i的各个位置上数字互不相同的数。

- i==1 : 1 0（0~9共10个数，均不重复）
- i==2: 9 * 9 （第一个位置上除0外有9种选择，第2个位置上除第一个已经选择的数，还包括数字0，也有9种选择）
- i ==3: 9* 9 * 8 （前面两个位置同i==2，第三个位置除前两个位置已经选择的数还有8个数可以用）
- .....
- i== n: 9 * 9 * 8 *..... (9-i+2)

需要注意的是，9- i + 2 >0 即 i < 11，也就是i最大为10，正好把每个数都用了一遍。

```
public class Solution {
    public int countNumbersWithUniqueDigits(int n) {
        n = Math.min(n,10);
        int[] dp = new int[n+1];
        dp[0]=1;
        for(int i=1;i<=n;i++){
            dp[i] = 9;
            for(int j = 9;j>=9-i+2;j--){
                dp[i] *=j;
            }
        }
        int ans = 0;
        for(int i=0;i<dp.length;i++) ans+=dp[i];
        return ans;
    }
}
```

CHAPTER 14: Heap

14.1 Kth Largest Element in an Array(Medium-> LeetCode No.215)

> Find the **k**th largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.
>
> For example,
>
> Given `[3,2,1,5,6,4]` and k = 2, return 5.
>
> **Note:**
>
> You may assume k is always valid, 1 ≤ k ≤ array's length.
>
> **Credits:**
>
> Special thanks to @mithmatt for adding this problem and creating all test cases.
>
> Subscribe to see which companies asked this question
>
> Hide Tags | Heap | Divide and Conquer
>
> Hide Similar Problems | (M) Wiggle Sort II | (M) Top K Frequent Elements

Solution:

```
public class Solution {
   public int findKthLargest(int[] nums, int k) {
      if(nums==null || nums.length==0 || k<=0) return 0;
      Arrays.sort(nums);
      return nums[nums.length-k];
   }
}
```

14.2 Super Ugly Number (Medium->Leetcode No.313)

> Write a program to find the n^{th} super ugly number.
>
> Super ugly numbers are positive numbers whose all prime factors are in the given prime list `primes` of size `k`. For example, `[1, 2, 4, 7, 8, 13, 14, 16, 19, 26, 28, 32]` is the sequence of the first 12 super ugly numbers given `primes` = `[2, 7, 13, 19]` of size 4.
>
> **Note:**
>
> (1) `1` is a super ugly number for any given `primes`.
>
> (2) The given numbers in `primes` are in ascending order.
>
> (3) 0 < `k` ≤ 100, 0 < `n` ≤ 10^6, 0 < `primes[i]` < 1000.

Solution:

```
public class Solution {
   public int nthSuperUglyNumber(int n, int[] primes) {
      int len = primes.length;
      int[] ugly = new int[n], index = new int[len], factor = new int[len], mul = new int[len];
      for(int i=0;i<len;i++){
         factor[i] = primes[i];
         mul[i] = primes[i];
      }

      ugly[0] = 1;
      for(int i=1;i<n;i++){
```

```
            int min = Integer.MAX_VALUE;
            for(int j=0;j<len;j++){
                if(min>factor[j]) min = factor[j];
            }
            ugly[i] = min;
            for(int k=0;k<len;k++){
                if(factor[k]==min) factor[k] = mul[k]*ugly[++index[k]];
            }
        }
        return ugly[n-1];
    }
}
```

CHAPTER 15: Array

15.1 Container With most water (Medium->Leetcode No.11)

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

Note: You may not slant the container.

Solution

```
public class Solution {
   public int maxArea(int[] height) {
      if(height==null || height.length<2) return 0;
      int max = 0;
      int left = 0;
      int right = height.length-1;
      while(left<right){
         max = Math.max(max, (right-left)*Math.min(height[left],height[right]));
         if(height[left]>height[right]) right--;
         else left++;
      }
      return max;
   }
}
```

15.2 3SUM (Medium->leetcode No.15)

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

Note: The solution set must not contain duplicate triplets.

```
For example, given array S = [-1, 0, 1, 2, -1, -4],

A solution set is:
[
  [-1, 0, 1],
  [-1, -1, 2]
]
```

Solution

```
public class Solution {
   public List<List<Integer>> threeSum(int[] nums) {
      List<List<Integer>> results = new ArrayList<List<Integer>>();
      if(nums==null || nums.length==0) return results;
      Arrays.sort(nums);
      for(int i=0;i<nums.length-2;i++){
         int j = i+1;
         int k = nums.length-1;
```

```
            while(j<k){
                int sum = nums[i]+nums[j]+nums[k];
                if(sum==0){
                    List<Integer> tmp = new ArrayList<>();
                    tmp.add(nums[i]);
                    tmp.add(nums[j]);
                    tmp.add(nums[k]);
                    results.add(tmp);
                    j++;
                    k--;
                    while(j<k && nums[j]==nums[j-1]) j++;
                    while(j<k && nums[k]==nums[k+1]) k--;
                }else if(sum<0) j++;
                else k--;
            }
        }
        Set<List<Integer>> set = new HashSet<>(results);
        results.clear();
        results.addAll(set);
        return results;
    }
}
```

15.3 3Sum Closet (Medium->Leetcode No.16)

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

```
    For example, given array S = {-1 2 1 -4}, and target = 1.

    The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).
```

Solution

```
public class Solution {
    public int threeSumClosest(int[] nums, int target) {
        int min = Integer.MAX_VALUE;
        int result = 0;
        Arrays.sort(nums);
        for(int i=0;i<nums.length-2;i++){
            int j = i+1;
            int k = nums.length-1;
            while(j<k){
                int sum = nums[i]+nums[j]+nums[k];
                int diff = Math.abs(sum-target);
                if(diff==0) return sum;
                if(diff<min){
                    min = diff;
```

```
                result = sum;
            }
            if(sum<=target) j++;
            else k--;
        }
    }
    return result;
  }
}
```

15.4 Remove Duplicates from Stored Array (Easy->leetcode NO.26)

Given a sorted array, remove the duplicates in place such that each element appear only *once* and return the new length.

Do not allocate extra space for another array, you must do this in place with constant memory.

For example,

Given input array *nums* = `[1,1,2]`,

Your function should return length = `2`, with the first two elements of *nums* being `1` and `2` respectively. It doesn't matter what you leave beyond the new length.

Solution

```
public class Solution {
  public int removeDuplicates(int[] nums) {
    if(nums==null || nums.length==0) return 0;
    Arrays.sort(nums);
    int count = 1;
    for(int i=1;i<nums.length;i++){
      if(nums[i]!=nums[i-1]){
        nums[count] = nums[i];
        count++;
      }
    }
    return count;
  }
}
```

15.5 Remove Element (Medium->leetcode NO.27)

Given an array and a value, remove all instances of that value in place and return the new length.

Do not allocate extra space for another array, you must do this in place with constant memory.

The order of elements can be changed. It doesn't matter what you leave beyond the new length.

Example:

Given input array *nums* = `[3,2,2,3]`, *val* = `3`

Your function should return length = 2, with the first two elements of *nums* being 2.

Hint:

1. Try two pointers.
2. Did you use the property of "the order of elements can be changed"?
3. What happens when the elements to remove are rare?

Solution

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

15.6 Next Permutation (Meidum->leetcode No.31)

Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).

The replacement must be in-place, do not allocate extra memory.

Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.

`1,2,3` → `1,3,2`

`3,2,1` → `1,2,3`

`1,1,5` → `1,5,1`

Solution

1) scan from right to left, find the first element that is less than its previous one.

```
4 5 6 3 2 1
  |
  p
```

2) scan from right to left, find the first element that is greater than p.

```
4 5 6 3 2 1
    |
    q
```

3) swap p and q

```
4 5 6 3 2 1
swap
4 6 5 3 2 1
```

4) reverse elements [p+1, nums.length]

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

```
public class Solution {
   public void nextPermutation(int[] nums) {
      if(nums==null || nums.length<2) return;
      // the first round to scan from right to left to find a number that is
      // less than its previous one
      int p = 0;
      for(int i=nums.length-2;i>=0;i--){
         if(nums[i]<nums[i+1]){
            p = i;
            break;
         }
      }

      // the second round to scan from right to left to find the first number that is greater than the one
above
      int q = 0;
```

```
        for(int i = nums.length-1;i>p;i--){
            if(nums[i]>nums[p]){
                q = i;
                break;
            }
        }
        if(p==0 && q==0) {
            reverse(nums, 0,nums.length-1);
            return;
        }
        // or the first step is to swap the numbers at p and q
        //  and the second step is to reverse the array from p
        int tmp = nums[p];
        nums[p] = nums[q];
        nums[q] = tmp;

        if(p<nums.length-1) reverse(nums, p+1, nums.length-1);
    }
    public void reverse(int[] nums, int left,int right){
        while(left<right){
            int tmp = nums[left];
            nums[left] = nums[right];
            nums[right] = tmp;
            left++;
            right--;
        }
    }
}
```

15.7 Rotate Image (Medium->leetcode No.48)

You are given an *n* x *n* 2D matrix representing an image.

Rotate the image by 90 degrees (clockwise).

Follow up:

Could you do this in-place?

Solution

```
public class Solution {
    public void rotate(int[][] matrix) {
        int n = matrix.length;
        for(int i=0;i<n/2;i++){
            for(int j = 0;j<Math.ceil((double)n/2.0);j++){
                int tmp = matrix[i][j];
                matrix[i][j] = matrix[n-1-j][i];
                matrix[n-1-j][i] = matrix[n-1-i][n-1-j];
                matrix[n-1-i][n-1-j] = matrix[j][n-1-i];
                matrix[j][n-1-i] = tmp;
```

```
        }
      }
    }
}
```

15.8 Spiral Matrix (Medium -> leetcode No.54)

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

For example,

Given the following matrix:

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

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

Solution

```
public class Solution {
   public List<Integer> spiralOrder(int[][] matrix) {
      List<Integer> result = new ArrayList<>();
      if(matrix==null || matrix.length==0) return result;
      int top = 0;
      int bottom = matrix.length-1;
      int left = 0;
      int right = matrix[0].length-1;
      while(true){
         for(int i = left; i<=right;i++) result.add(matrix[top][i]);
         top++;
         if(left>right || top>bottom) break;

         for(int i=top;i<=bottom;i++) result.add(matrix[i][right]);
         right--;
         if(left>right || top>bottom) break;

         for(int i = right;i>=left;i--) result.add(matrix[bottom][i]);
         bottom--;
         if(left>right || top>bottom) break;

         for(int i = bottom;i>=top;i--) result.add(matrix[i][left]);
         left++;
         if(left>right || top>bottom) break;
      }
      return result;
   }
```

}

15.9 Spiral Matrix II (Medium->leetcode No.59)

Given an integer n, generate a square matrix filled with elements from 1 to n^2 in spiral order.

For example,

Given n = `3`,

You should return the following matrix:

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

Solution

```
public class Solution {
   public int[][] generateMatrix(int n) {
      if(n==0) return new int[0][0];
      int[][] matrix = new int[n][n];
      int start = 0;
      int end = n;
      int counter = 1;
      while(start<end){
         for(int i=start;i<end;i++) matrix[start][i]=counter++;
         for(int i=start+1;i<end;i++) matrix[i][end-1]=counter++;
         for(int i=end-2;i>=start;i--) matrix[end-1][i]=counter++;
         for(int i=end-2;i>start;i--) matrix[i][start]=counter++;
         start++;
         end--;
      }
      return matrix;
   }
}
```

15.10 Plus one (Easy->leetcode NO.66)

Given a non-negative number represented as an array of digits, plus one to the number.

The digits are stored such that the most significant digit is at the head of the list.

Solution

```
public class Solution {
   public int[] plusOne(int[] digits) {
      if(digits==null || digits.length==0) return new int[0];
      int carry = 1;
```

```
    for(int i=digits.length-1;i>=0;i--){
      int sum = digits[i]+carry;
      if(sum>=10) carry = 1;
      else carry = 0;
      digits[i] = sum%10;
    }
    if(carry==1){
      int[] result = new int[digits.length+1];
      System.arraycopy(digits, 0, result, 1, digits.length);
      result[0] = 1;
      return result;
    }else return digits;
  }
}
```

15.11 Set Matrix ZEROES (Medium->leetcode No.73)

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

Solution

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

15.12 Sort Colors (Medimu->leetcode NO.75)

Given an array with *n* objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue.

Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

Note:

You are not suppose to use the library's sort function for this problem.

Solution:

```
public class Solution {
   public void sortColors(int[] nums) {
      if(nums==null || nums.length<2) return;
      int[] count = new int[3];
      for(int i=0;i<nums.length;i++) count[nums[i]]++;
      int j = 0, k = 0;
      while(j<=2){
         if(count[j]!=0){
            nums[k++] = j;
            count[j] = count[j]-1;
         }else j++;
      }
   }
}
```

15.13 Remove Duplicates from sorted array II (Medium -> leetcode No.80)

Follow up for "Remove Duplicates":

What if duplicates are allowed at most *twice*?

For example,

Given sorted array *nums* = `[1,1,1,2,2,3]`,

Your function should return length = `5`, with the first five elements of *nums* being `1`, `1`, `2`, `2` and `3`. It doesn't matter what you leave beyond the new length.

Soution

```
public class Solution {
   public int removeDuplicates(int[] nums) {
      if(nums==null ) return 0;
      if(nums.length<=2) return nums.length;
      int pre_index = 1;
      int cur_index = 2;
      while(cur_index<nums.length){
         if(nums[cur_index]==nums[pre_index] && nums[cur_index]==nums[pre_index-1]) cur_index++;
         else{
            pre_index++;
```

```
            nums[pre_index] = nums[cur_index];
            cur_index++;
        }
      }
      // pre_index+1 is because it begins from 1 instead of 0, so it needs tobe added one
      return pre_index+1;
    }
}
```

15.14 Merge Sorted Array (Easy->leetcode No.88)

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

Note:

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

Solution: the key to solve this problem is moving elements of A and B backwards. If B has some elements left after A is done, also need to handle that case.

```
public class Solution {
   public void merge(int[] nums1, int m, int[] nums2, int n) {
      while(m>0 && n>0){
         if(nums1[m-1]>nums2[n-1]){
            nums1[m+n-1] = nums1[m-1];
            m--;
         }else{
            nums1[m+n-1] = nums2[n-1];
            n--;
         }
      }
      while(n>0){
         nums1[m+n-1] = nums2[n-1];
         n--;
      }
   }
}
```

15.15 Pascal's Triangle II (Easy->leetcode No.119)

Given an index *k*, return the k^{th} row of the Pascal's triangle.

For example, given *k* = 3,

Return `[1,3,3,1]`.

Note:

Could you optimize your algorithm to use only *O*(*k*) extra space?

Solution

```
public class Solution {
   public List<Integer> getRow(int rowIndex) {
      List<Integer> result = new ArrayList<>();
      if(rowIndex<0) return result;
      result.add(1);
      for(int i=1;i<=rowIndex;i++){
         for(int j = result.size()-2;j>=0;j--){
            result.set(j+1, result.get(j)+result.get(j+1));
         }
         result.add(1);
      }
      return result;
   }
}
```

15.16 Best Time to Buy and Sell Stock II(Medium->leetcode NO.122)

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

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Solution:

```
public class Solution {
   public int maxProfit(int[] prices) {
      if(prices==null || prices.length<=1) return 0;
      int result = 0;
      for(int i=1;i<prices.length;i++){
         int diff = prices[i]-prices[i-1];
         if(diff>0) result += diff;
      }
      return result;
   }
}
```

15.17 Find Minimum in Rotated Sorted Array (Medium->leetcode NO.153)

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.

You may assume no duplicate exists in the array.

Solution

```
public class Solution {
```

```
    public int findMin(int[] nums) {
        if(nums==null || nums.length==0) return 0;
        if(nums.length==1) return nums[0];
        int left = 0;
        int right = nums.length-1;
        if(nums[left]<nums[right]) return nums[left];
        while(left<right){
            int mid = left+(right-left)/2;
            if(right-left==1) return nums[right];
            if(nums[right]<nums[mid]) left = mid;
            else right = mid;
        }
        return nums[left];
    }
}
```

15.18 majority Element (Medium->leetcode NO.169)

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

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

Solution

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

15.19 Rotate Array (Easy->leetcode No.189)

Rotate an array of *n* elements to the right by *k* steps.

For example, with *n* = 7 and *k* = 3, the array `[1,2,3,4,5,6,7]` is rotated to `[5,6,7,1,2,3,4]`.

Note:

Try to come up as many solutions as you can, there are at least 3 different ways to solve this problem.

[show hint]

Hint:

Could you do it in-place with O(1) extra space?

Related problem: Reverse Words in a String II

Solution: use the reversal method that takes the O(1) space and O(n) time. Suppose we are given

```
1. Divide the array two parts: 1,2,3,4 and 5, 6
2. Reverse first part: 4,3,2,1,5,6
3. Reverse second part: 4,3,2,1,6,5
4. Reverse the whole array: 5,6,1,2,3,4
```

```
public class Solution {
    public void rotate(int[] nums, int k) {
        if(nums==null || nums.length==0 || k<0){
            throw new IllegalArgumentException("Illegal argument");
        }
        if(k>nums.length) k = k%nums.length;
        int first = nums.length-k;
        reverse(nums, 0, first-1);
        reverse(nums, first, nums.length-1);
        reverse(nums, 0, nums.length-1);
    }
    public void reverse(int[] nums, int start, int end){
        if(nums==null || nums.length==1) return;
        while(start<end){
            int tmp = nums[start];
            nums[start] = nums[end];
            nums[end] = tmp;
            start++;
            end--;
        }
    }
}
```

15.20 Summary Ranges (Medium->leetcode No.228)

Given a sorted integer array without duplicates, return the summary of its ranges.

For example, given `[0,1,2,4,5,7]`, return `["0->2","4->5","7"].`

Solution

```
public class Solution {
    public List<String> summaryRanges(int[] nums) {
        List<String> result = new ArrayList<>();
        if(nums==null || nums.length==0) return result;
        if(nums.length==1) result.add(nums[0]+"");
        int pre = nums[0];
        int first = pre;
        for(int i=1;i<nums.length;i++){
            if(nums[i]==pre+1){
                if(i==nums.length-1) result.add(first+"->"+nums[i]);
            }else{
                if(first==pre) result.add(first+"");
```

```
            else result.add(first+"->"+pre);
            if(i==nums.length-1) result.add(nums[i]+"");
            first = nums[i];
          }
          pre = nums[i];
        }
        return result;
    }
}
```

15.21 Majority Element ii (Medium->leetcode No.229)

Given an integer array of size *n*, find all elements that appear more than `⌊ n/3 ⌋` times. The algorithm should run in linear time and in O(1) space.

Hint:

1. How many majority elements could it possibly have?
2. Do you have a better hint? Suggest it!

Solution: 1) the first solution is to use the HashMap to count the number of each number as number->count. And then it will iterate the map to find which number has occurred at least n/3 times. But it will use the O(n) time complexity and O(n) space complexity.
2) Time O(n) and space complexity is O(1). In this solution, we need to make full use of the condition that finding the elements that occur more than n/3 times, thus we can have at most two such elements, that is why we define num1 and num2.

```
public class Solution {
    public List<Integer> majorityElement(int[] nums) {
        List<Integer> result = new ArrayList<>();
        if(nums==null || nums.length==0) return result;
        int num1 = nums[0], num2 = nums[0], count1 = 0, count2 = 0;
        int len = nums.length;
        for(int i = 0;i<len;i++){
            if(num1==nums[i]) count1++;
            else if(num2 == nums[i]) count2++;
            else if (count1==0) {
                num1 = nums[i];
                count1 = 1;
            }else if(count2==0){
                num2 = nums[i];
                count2 = 1;
            }else {
                count1--;
                count2--;
            }
        }
        count1=0;
```

```
        count2=0;
        for(int i = 0;i<len;i++){
            if(nums[i]==num1) count1++;
            else if(nums[i]==num2) count2++;
        }
        if(count1>len/3) result.add(num1);
        if(count2>len/3) result.add(num2);
        return result;
    }
}
```

15.22 Product of Array Except Self (Medium->leetcode No.238)

Given an array of *n* integers where *n* > 1, `nums`, return an array `output` such that `output[i]` is equal to the product of all the elements of `nums` except `nums[i]`.

Solve it **without division** and in O(*n*).

For example, given `[1,2,3,4]`, return `[24,12,8,6]`.

Follow up:

Could you solve it with constant space complexity? (Note: The output array **does not** count as extra space for the purpose of space complexity analysis.)

Solution

```
public class Solution {
    public int[] productExceptSelf(int[] nums) {
        if(nums==null || nums.length==0) throw new IllegalArgumentException("");
        int[] output = new int[nums.length];
        output[nums.length-1] = 1;
        for(int i=nums.length-2;i>=0;i--) output[i] = output[i+1]*nums[i+1];
        int left = 1;
        for(int i=0;i<nums.length;i++){
            output[i] =output[i]*left;
            left = left*nums[i];
        }
        return output;
    }
}
```

15.23 Missing Number (Medium->leetcode NO.268)

Given an array containing *n* distinct numbers taken from `0, 1, 2, ..., n`, find the one that is missing from the array.

For example,

Given *nums* = `[0, 1, 3]` return `2`.

Note:

Your algorithm should run in linear runtime complexity. Could you implement it using only constant extra space complexity?

Solution

```
public class Solution {
   public int missingNumber(int[] nums) {
      if(nums==null || nums.length==0) return 0;
      int n = nums.length;
      int sum = n*(1+n)/2;
      int new_sum = 0;
      for(int i=0;i<n;i++) new_sum+=nums[i];
      return sum-new_sum;
   }
}
```

15.24 Move Zeroes (Easy->leetcode No.283)

Given an array `nums`, write a function to move all `0`'s to the end of it while maintaining the relative order of the non-zero elements.

For example, given `nums = [0, 1, 0, 3, 12]`, after calling your function, `nums` should be `[1, 3, 12, 0, 0]`.

Note:

1. You must do this **in-place** without making a copy of the array.
2. Minimize the total number of operations.

Solution

```
public class Solution {
   public void moveZeroes(int[] nums) {
      if(nums==null || nums.length==0) return;
      int i = 0;
      for(int d:nums){
         if(d!=0) nums[i++] = d;
      }
      Arrays.fill(nums, i, nums.length,0);
   }
}
```

15.25 Game of Life(Medium->leetcode No.289)

According to the Wikipedia's article: "The **Game of Life**, also known simply as **Life**, is a cellular automaton devised by the British mathematician John Horton Conway in 1970."

Given a *board* with *m* by *n* cells, each cell has an initial state *live* (1) or *dead* (0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules (taken from the above Wikipedia article):

1. Any live cell with fewer than two live neighbors dies, as if caused by under-population.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by over-population..
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

Write a function to compute the next state (after one update) of the board given its current state.

Follow up:

1. Could you solve it in-place? Remember that the board needs to be updated at the same time: You cannot update some cells first and then use their updated values to update other cells.
2. In this question, we represent the board using a 2D array. In principle, the board is infinite, which would cause problems when the active area encroaches the border of the array. How would you address these problems?

Solution

```
public class Solution {
   public void gameOfLife(int[][] board) {
   int m=board.length;
   int n=board[0].length;
   int[] dx = {-1, -1, 0, 1, 1, 1, 0, -1};
   int[] dy = {0, 1, 1, 1, 0, -1, -1, -1};

   for(int i=0; i<m; i++){
      for(int j=0; j<n; j++){
         int count=0;
         for(int k=0; k<8; k++){
            int x=i+dx[k];
            int y=j+dy[k];
            if(x>=0&&x<m&&y>=0&&y<n&&(board[x][y]&1)==1){
               count++;
            }
         }

         if(count==3 && (board[i][j]&1)==0){
```

```
            board[i][j]=2;
        }

        if(count<2 && (board[i][j]&1)==0){
            board[i][j]=0;
        }

        if(count>=2 && count<=3 && (board[i][j]&1)==1){
            board[i][j]=3;
        }

        if(count>3 && (board[i][j]&1)==1){
            board[i][j]=1;
        }
      }
    }

    for(int i=0; i<m; i++){
      for(int j=0; j<n; j++){
        board[i][j] >>=1;
      }
    }
}
}
```

CHAPTER 16: Math

16.1 Reverse Integer (Easy->Leetcode No.7)

Reverse digits of an integer.

Example1: x = 123, return 321

Example2: x = -123, return -321

Solution

```
public class Solution {
   public int reverse(int x) {
      int rev = 0;
      while(x!=0){
         rev = rev*10+x%10;
         x = x/10;
      }
      return rev;
   }
}
```

16.2 Palindrome Number (Easy->leetcode NO.9)

Determine whether an integer is a palindrome. Do this without extra space.

Solution

```
public class Solution {
   public boolean isPalindrome(int x) {
      if(x<0) return false;
      int ret = reverse(x);
      return ret==x;
   }
   public int reverse(int x){
      int res = 0;
      while(x!=0){
         res = res*10+x%10;
         x /= 10;
      }
      return res;
   }
}
```

16.3 Excel Sheet Column Title (Easy->leetcode No.168)

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

For example:

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

Solution

```
public class Solution {
   public String convertToTitle(int n) {
      if(n<0) throw new IllegalArgumentException("");
      StringBuilder sb = new StringBuilder();
      while(n>0){
         n--;
         char ch = (char)(n%26+'A');
         n/=26;
         sb.append(ch);
      }
      sb.reverse();
      return sb.toString();
   }
}
```

16.4 Excel Sheet Column Number (Easy->leetcode No.171)

Related to question Excel Sheet Column Title

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

For example:

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

Solution

```
public class Solution {
   public int titleToNumber(String s) {
      if(s==null || s.length==0) throw new IllegalArgumentException("");
      int result = 0;
      int i = s.length()-1;
      int t = 0;
      while(i>=0){
         char ch = s.charAt(i);
         result = result+(int)Math.pow(26,t)*(ch-'A'+1);
         t++;
         i--;
      }
      return result;
   }
}
```

16.5 Factorial Trailing Zeroes (Easy->leetcode No.172)

Given an integer *n*, return the number of trailing zeroes in *n*!.

Note: Your solution should be in logarithmic time complexity.

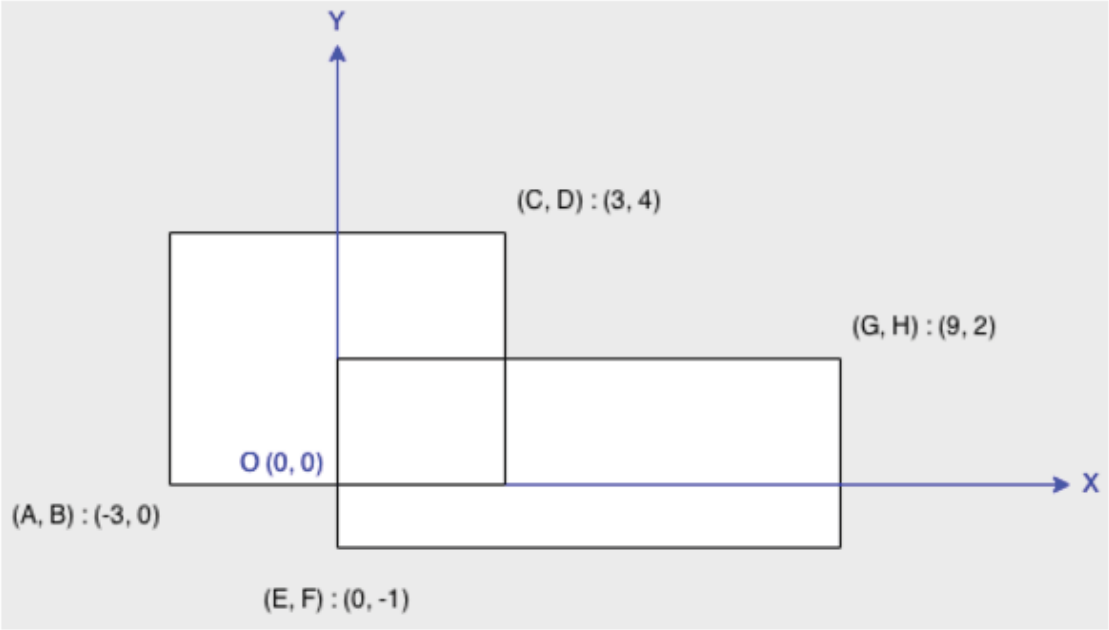
Solution: the idea is to count what is the amount of factor 5. Because all trailing 0 is from factors 5*2 and that one number may have several 5 factors, but factors 2 is always ample, so we just need to count how many 5 factors in all numbers from 1 to n.

```
public class Solution {
   public int trailingZeroes(int n) {
      if(n<0) return -1;
      return n==0?0:n/5+trailingZeroes(n/5);
   }
}
```

16.6 Rectanle Area (Easy->Leetcode NO.223)

Find the total area covered by two **rectilinear** rectangles in a **2D** plane.

Each rectangle is defined by its bottom left corner and top right corner as shown in the figure.



Assume that the total area is never beyond the maximum possible value of **int**.

Solution

```
public class Solution {
    public int computeArea(int A, int B, int C, int D, int E, int F, int G, int H) {
        if(C<E || G<A) return (G-E)*(H-F)+(C-A)*(D-B);
        if(D<F || H<B) return (G-E)*(H-F)+(C-A)*(D-B);
        int right = Math.min(C,G);
        int left = Math.max(A,E);
        int top = Math.min(H,D);
        int bottom = Math.max(F,B);
        return (G-E)*(H-F)+(C-A)*(D-B)-(right-left)*(top-bottom);
    }
}
```

16.7 Power of Two (Easy->leetcode No.231)

Given an integer, write a function to determine if it is a power of two.

Solution: the idea is that the number that is pow of two has the binary format 1000….

```
public class Solution {
    public boolean isPowerOfTwo(int n) {
        return n>0 && (n&n-1)==0;
    }
}
```

16.8 Add Digits (Easy->leetcode NO.258)

Given a non-negative integer `num`, repeatedly add all its digits until the result has only one digit.

For example:

Given `num = 38`, the process is like: `3 + 8 = 11`, `1 + 1 = 2`. Since `2` has only one digit, return it.

Follow up:

Could you do it without any loop/recursion in O(1) runtime?

Hint:

1. A naive implementation of the above process is trivial. Could you come up with other methods?
2. What are all the possible results?
3. How do they occur, periodically or randomly?
4. You may find this Wikipedia article useful.

Sen

Solution

```
public class Solution {
   public int addDigits(int num) {
      return num-9*((num-1)/9);
   }
}
```

16.9 Ugly Number (Easy -> leetcode No.263)

Write a program to check whether a given number is an ugly number.

Ugly numbers are positive numbers whose prime factors only include `2, 3, 5`. For example, `6, 8` are ugly while `14` is not ugly since it includes another prime factor `7`.

Note that `1` is typically treated as an ugly number.

Solution

```
public class Solution {
   public boolean isUgly(int num) {
      if(num<=0) return false;
      int[] div = {2,3,5};
      for(int i:div){
         while(num%i==0) num /= i;
      }
      return num==1;
   }
}
```

16.10 Power of Three (Easy -> leetcode NO.326)

Given an integer, write a function to determine if it is a power of three.

Follow up:

Could you do it without using any loop / recursion?

Solution 1: recursive solution

```
Return n>0 && (n==1 || (n%3==0 && isPowerOfThree(n/3)));
```

Solution 2: iterative solution

```
Public boolean isPowerOfThree (int n){
        If(n>1){
                While(n%3==0) n /= 3;
        }
        Return n==1;
}
```

Solution 3: Math

```
public class Solution {
   public boolean isPowerOfThree(int n) {
      return n>0 && (int)Math.pow(3,
(int)(Math.log(Integer.MAX_VALUE)/Math.log(3.0)))%n==0;
   }
}
```

New Method 4: the idea is to conver the original number to radix-3 (it makes 3 as the base, so the trinary format is 10*) format and check if it is of format 10* where 0* means k zeros with k>=0.

```
public class Solution {
   public boolean isPowerOfThree(int n) {
      return Integer.toString(n,3).matches("10*");
   }
}
```

CHAPTER 17: Bit Manipulation

17.1 Single Number II (Medium->leetcode No.137)

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

Note:

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

Solution

```
public class Solution {
   public int singleNumber(int[] nums) {
   int res = 0;
   for(int i = 0; i < 32; i++){
      int sum = 0;
      for(int n: nums)
         if((n >> i & 1) == 1)
            sum++;
      sum %= 3;
      res |= sum<<i;
   }
   return res;
}
}
```

17.2 Reverse Bits (Easy->leetcodeNo.190)

Reverse bits of a given 32 bits unsigned integer.

For example, given input 43261596 (represented in binary as **00000010100101000001111010011100**), return 964176192 (represented in binary as **00111001011110000010100101000000**).

Follow up:

If this function is called many times, how would you optimize it?

Related problem: Reverse Integer

Solution

```
public class Solution {
   // you need treat n as an unsigned value
   public int reverseBits(int n) {
      for(int i=0;i<16;i++){
         n = swapbits(n, i, 32-i-1);
      }
      return n;
   }
   public int swapbits(int n, int i, int j){
```

```
        int a = (n>>i)&1;
        int b = (n>>j)&1;
        if((a^b)!=0) return n^=(1<<i)|(1<<j);
        return n;
    }
}
```

17.3 Number of 1 Bits(Easy->leetcode NO.191)

Write a function that takes an unsigned integer and returns the number of '1' bits it has (also known as the Hamming weight).

For example, the 32-bit integer '11' has binary representation `00000000000000000000000000001011`, so the function should return 3.

Solution

```
public class Solution {
    // you need to treat n as an unsigned value
    public int hammingWeight(int n) {
        int sum = 0;
        for(int i=0;i<32;i++){
            if(((n>>i)&1)!=0) sum++;
        }
        return sum;
    }
}
```

17.4 Single Number III (Medium->leetcode No.260)

Given an array of numbers `nums`, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once.

For example:

Given `nums = [1, 2, 1, 3, 2, 5]`, return `[3, 5]`.

Note:

1. The order of the result is not important. So in the above example, `[5, 3]` is also correct.
2. Your algorithm should run in linear runtime complexity. Could you implement it using only constant space complexity?

Solution

```
public class Solution {
    public int[] singleNumber(int[] nums) {
        int sum = 0;
        for(int i = 0;i<nums.length;i++) sum^=nums[i];
        int[] result = new int[2];
```

```
// since sum is the result of the required two numbers that occures only once, we need to have bitflag to seperate them
      int bitflag = (sum & (~(sum-1)));
      for(int num:nums){
         if((bitflag&num)==0) result[0]^=num;
         else result[1]^=num;
      }
      return result;
   }
}
```

17.5 Maximum Product of Word Lengths (Medium->leetcode No.318)

Given a string array `words`, find the maximum value of `length(word[i]) * length(word[j])` where the two words do not share common letters. You may assume that each word will contain only lower case letters. If no such two words exist, return 0.

Example 1:

Given `["abcw", "baz", "foo", "bar", "xtfn", "abcdef"]`

Return `16`

The two words can be `"abcw", "xtfn"`.

Example 2:

Given `["a", "ab", "abc", "d", "cd", "bcd", "abcd"]`

Return `4`

The two words can be `"ab", "cd"`.

Example 3:

Given `["a", "aa", "aaa", "aaaa"]`

Return `0`

No such pair of words.

Solution

```
public class Solution {
   public int maxProduct(String[] words) {
      if(words==null || words.length==0) return 0;
      int[] arr = new int[words.length];
      for(int i = 0;i<words.length;i++){
         for(int j = 0;j<words[i].length();j++){
            char c = words[i].charAt(j);
            arr[i] |= (1<<(c-'a'));
         }
```

```
        }

        int result = 0;
        for(int i=0;i<words.length;i++){
            for(int j =i+1;j<words.length;j++){
                if((arr[i]&arr[j])==0) result = Math.max(result, words[i].length()*words[j].length());
            }
        }
        return result;
    }
}
```

17.6 Power of Four (Easy->leetcode No.342)

Given an integer (signed 32 bits), write a function to check whether it is a power of 4.

Example:

Given num = 16, return true. Given num = 5, return false.

Solution1:

```
public class Solution {
    public boolean isPowerOfFour(int num) {
        return num>0&&Integer.toString(num,4).matches("10*");
    }
}
```

Solution 2:

```
public class Solution {
    public boolean isPowerOfFour(int num) {
        int count1 = 0;
        int count0 = 0;
        while(num>0){
            if((num&1)==1) count1++;
            else count0++;
            num>>=1;
        }
        return count1==1 && count0%2==0;
    }
}
```

Java Solution 3 - Math Equation

We can use the following formula to solve this problem without using recursion/iteration.

$$\log_b(x) = \frac{\log_{10}(x)}{\log_{10}(b)} = \frac{\log_e(x)}{\log_e(b)}.$$

```java
public boolean isPowerOfFour(int num) {
    if(num==0) return false;

    int pow = (int) (Math.log(num) / Math.log(4));
    if(num==Math.pow(4, pow)){
        return true;
    }else{
        return false;
    }
}
```

CHAPTER 18: Others

18.1 Jump Game (Medium-> Leetcode No.55)

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

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

Determine if you are able to reach the last index.

For example:

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

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

Solution

```
public class Solution {
   public boolean canJump(int[] nums) {
      if(nums==null || nums.length<=1) return true;
      int max = nums[0];
      for(int i=0;i<nums.length;i++){
         // the steps represented by max has exhausted and nums[i]=0 can not
         // move further.
         if(max<=i && nums[i]==0) return false;
         if(i+nums[i]>max) max = i+nums[i];
         if(max>=nums.length-1) return true;
      }
      return false;
   }
}
```

18.2 Pascal's Triangle (Easy->leetcode No.118)

Given *numRows*, generate the first *numRows* of Pascal's triangle.

For example, given *numRows* = 5,

Return

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

Solution

```
public class Solution {
   public List<List<Integer>> generate(int numRows) {
      List<List<Integer>> results = new ArrayList<>();
      if(numRows<=0) return results;
      List<Integer> result = new ArrayList<>();
      result.add(1);
      results.add(new ArrayList<>(result));
      if(numRows==1) return results;
      for(int i=2;i<=numRows;i++){
         for(int j=result.size()-2;j>=0;j--){
            result.set(j+1, result.get(j)+result.get(j+1));
         }
         result.add(1);
         results.add(new ArrayList<>(result));
      }
      return results;
   }
}
```

18.3 Word Ladder (Medium->leetcode NO.127)

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

1. Only one letter can be changed at a time
2. Each intermediate word must exist in the word list

For example,

Given:

beginWord = `"hit"`

endWord = `"cog"`

wordList = `["hot","dot","dog","lot","log"]`

As one shortest transformation is `"hit" -> "hot" -> "dot" -> "dog" -> "cog"`, return its length `5`.

Note:

- Return 0 if there is no such transformation sequence.
- All words have the same length.
- All words contain only lowercase alphabetic characters.

Solution: this is the search problem and bfs guarantees the optimal solution

```
class WordNode{
   String word;
   int step;
   public WordNode(String word, int step){
      this.word = word;
      this.step = step;
   }
}
public class Solution {
   public int ladderLength(String beginWord, String endWord, Set<String> wordList) {
      LinkedList<WordNode> queue = new LinkedList<>();
      queue.add(new WordNode(beginWord,1));

      wordList.add(endWord);
      while(!queue.isEmpty()){
         WordNode top = queue.remove();
         String word = top.word;

         if(word.equals(endWord)) return top.step;
         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(wordList.contains(newword)){
                  queue.add(new WordNode(newword, top.step+1));
                  wordList.remove(newword);
               }
               arr[i] = tmp;
            }
         }
      }
      return 0;
   }
}
```

18.4 Gas Station (Medium->leetcode No.134)

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

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

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

Note:

The solution is guaranteed to be unique.

Solution

To solve this problem, we need to understand and use the following 2 facts:
1) if the sum of gas >= the sum of cost, then the circle can be completed.
2) if A can not reach C in a the sequence of A-->B-->C, then B can not make it either.

Proof of fact 2:

```
If gas[A] < cost[A], then A can not even reach B.
So to reach C from A, gas[A] must >= cost[A].
Given that A can not reach C, we have gas[A] + gas[B] < cost[A] + cost[B],
and gas[A] >= cost[A],
Therefore, gas[B] < cost[B], i.e., B can not reach C.
```

In the following solution, sumRemaining tracks the sum of remaining to the current index. If sumRemaining < 0, then every index between old start and current index is bad, and we need to update start to be the current index. You can use the following example to visualize the solution.

index	0	1	2	3	4
gas	1	2	3	4	5
cost	1	3	2	4	5

```
public class Solution {
   public int canCompleteCircuit(int[] gas, int[] cost) {
      int sumremaining = 0;
      int total = 0;
      int start = 0;
      for(int i=0;i<gas.length;i++){
         int remaining = gas[i]-cost[i];
         if(sumremaining>=0) sumremaining += remaining;
         else{
            start = i;
            sumremaining = remaining;
         }
         total+=remaining;
      }
      if(total>=0) return start;
      else return -1;
   }
}
```

18.5 Word Break (Medium->leetcode NO.139)

Given a string *s* and a dictionary of words *dict*, determine if *s* can be segmented into a space-separated sequence of one or more dictionary words.

For example, given

s = `"leetcode"`,

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

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

Solution

```
public class Solution {
   public boolean wordBreak(String s, Set<String> wordDict) {
      int[] pos = new int[s.length()+1];
      Arrays.fill(pos, -1);
      pos[0] = 0;
      for(int i=0;i<s.length();i++){
         if(pos[i]!=-1){
            for(int j =i+1;j<=s.length();j++){
               String sub = s.substring(i,j);
               if(wordDict.contains(sub)) pos[j]=i;
            }
         }
      }
      return pos[s.length()]!=-1;
   }
}
```

18.6 Reorder List (Medium->leetcode No.143)

Given a singly linked list L: $L_0 \rightarrow L_1 \rightarrow \ldots \rightarrow L_{n-1} \rightarrow L_n$,

reorder it to: $L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \rightarrow \ldots$

You must do this in-place without altering the nodes' values.

For example,

Given `{1,2,3,4}`, reorder it to `{1,4,2,3}`.

Solution

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
```

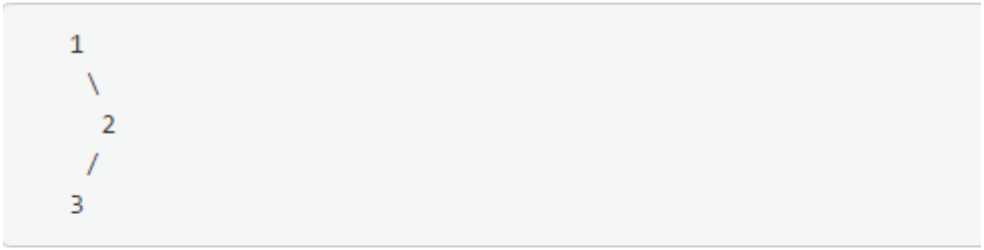
```
public class Solution {
   public void reorderList(ListNode head) {
      if(head==null || head.next==null) return;
      // find the middle
      ListNode slow = head;
      ListNode fast = head.next;
      while(fast!=null && fast.next!=null){
         slow = slow.next;
         fast = fast.next.next;
      }

      // reverse the second half
      ListNode cur = slow.next;
      slow.next = null;
      ListNode pre = null;
      while(cur!=null){
         ListNode next = cur.next;
         cur.next = pre;
         pre = cur;
         cur = next;
      }

      //merge two lists
      ListNode tail = pre;
      cur = head;
      while(tail!=null){
         ListNode next = cur.next;
         pre = tail.next;

         cur.next = tail;
         tail.next = next;

         cur = next;
         tail = pre;
      }

   }
}
```

18.7 Binary Tree Preorder Traversal (Medium->leetcode NO.144)

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

For example:

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



return `[1,2,3]`.

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

Solution

```
/**
 * 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> preorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        if(root==null) return result;
        Stack<TreeNode> stack = new Stack<>();
        stack.push(root);
        while(!stack.empty()){
            TreeNode top = stack.pop();
            result.add(top.val);

            if(top.right!=null) stack.push(top.right);
            if(top.left!=null) stack.push(top.left);
        }
        return result;
    }
}
```

18.8 Sort List (Medium->leetcode NO.148)

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

Solution 1: quick sort

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode sortList(ListNode head) {
        if (head == null || head.next == null)
            return head;
        ListNode pivot = head;
        ListNode left = new ListNode(-1);
        ListNode right = new ListNode(-1);
        ListNode leftHead = left;
        ListNode rightHead = right;
        ListNode pivotHead = pivot;
        ListNode crt = head.next;
        if (crt == null)
            return pivot;

        while(crt != null) {
            if (crt.val < pivot.val) {
                left.next = crt;
                left = left.next;
            } else if (crt.val > pivot.val){
                right.next = crt;
                right = right.next;
            } else {
                pivot.next = crt;
                pivot = pivot.next;
            }
            crt = crt.next;
        }

        pivot.next = null;
        left.next = null;
        right.next = null;
        left = sortList(leftHead.next);
        right = sortList(rightHead.next);
        pivot.next = right;
        ListNode re = left;
```

```
            while (left != null && left.next != null) {
                left = left.next;
            }

            if (left == null)
                re = pivotHead;
            else
                left.next = pivotHead;
            return re;
        }
}
```

Solution 2: merge sort

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode sortList(ListNode head) {
        if (head == null || head.next == null)
            return head;

        ListNode prev = head;
        ListNode slow = head;
        ListNode fast = head;
        while (fast != null && fast.next != null) {
            prev = slow;
            slow = slow.next;
            fast = fast.next.next;
        }

        prev.next = null;
        ListNode firstHalf = sortList(head);
        ListNode secondHalf = sortList(slow);
        return merge(firstHalf, secondHalf);
    }

    private ListNode merge(ListNode firstHalf, ListNode secondHalf) {
        ListNode dummy = new ListNode(-1);
        ListNode crt = dummy;
        while(firstHalf != null && secondHalf != null) {
            if (firstHalf.val < secondHalf.val) {
                crt.next = firstHalf;
                firstHalf = firstHalf.next;
            } else {
```

```
            crt.next = secondHalf;
            secondHalf = secondHalf.next;
        }

        crt = crt.next;
    }

    if (firstHalf != null)
        crt.next = firstHalf;
    else
        crt.next = secondHalf;

    return dummy.next;
  }
}
```

18.9 Evaluate Reverse Polish Notation (Medium->leetcode NO.150)

Evaluate the value of an arithmetic expression in Reverse Polish Notation.

Valid operators are `+`, `-`, `*`, `/`. Each operand may be an integer or another expression.

Some examples:

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

Solution

```
public class Solution {
  public int evalRPN(String[] tokens) {
    if(tokens==null || tokens.length==0) return 0;
    Stack<Integer> stack = new Stack<>();
    for(String s:tokens){
      if(s.equals("+") || s.equals("-") || s.equals("*") || s.equals("/")){
        int a = stack.pop();
        int b = stack.pop();
        switch(s){
          case "+":stack.push(a+b);break;
          case "-":stack.push(b-a);break;
          case "*":stack.push(a*b);break;
          case "/":stack.push(b/a);break;
        }
      }else{
        stack.push(Integer.valueOf(s));
      }
    }
    return stack.pop();
  }
```

}

18.10 Largest Number (Medium->leetcode No.179)

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

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

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

Solution

```
public class Solution {
    public String largestNumber(int[] nums) {
        String[] strs = new String[nums.length];
        for(int i=0;i<nums.length;i++){
            strs[i] = String.valueOf(nums[i]);
        }
        Arrays.sort(strs,(str1,str2)->(str2+str1).compareTo(str1+str2));
        StringBuilder sb = new StringBuilder();
        for(String s:strs) sb.append(s);
        while(sb.charAt(0)=='0' && sb.length()>1) sb.deleteCharAt(0);
        return sb.toString();
    }
}
```

18.11 Invert binary Tree (Easy->leetcode No.226)

Invert a binary tree.

```
     4
   /   \
  2     7
 / \   / \
1   3 6   9
```

to

```
     4
   /   \
  7     2
 / \   / \
9   6 3   1
```

Solution: Recusive

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

```
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
   public TreeNode invertTree(TreeNode root) {
      if(root!=null) helper(root);
      return root;
   }
   public TreeNode helper(TreeNode root){
      TreeNode tmp = root.left;
      root.left = root.right;
      root.right = tmp;
      if(root.left!=null) helper(root.left);
      if(root.right!=null) helper(root.right);
      return root;
   }
}
```

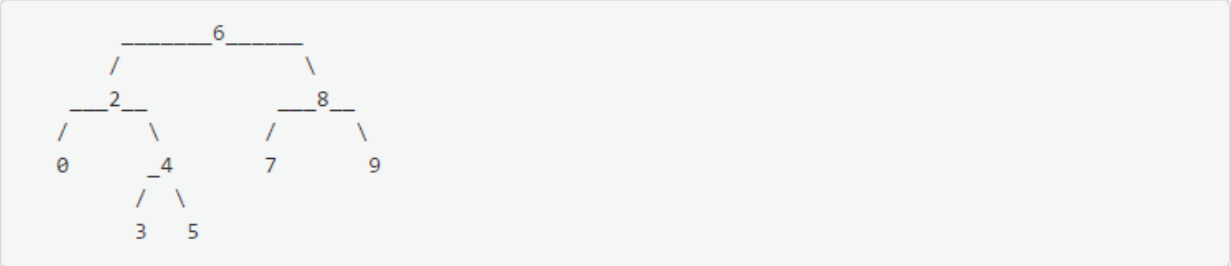
Solution2: iterative

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
   public TreeNode invertTree(TreeNode root) {
      LinkedList<TreeNode> queue = new LinkedList<>();
      if(root!=null) queue.add(root);
      while(!queue.isEmpty()){
         TreeNode top = queue.poll();
         if(top.left!=null) queue.add(top.left);
         if(top.right!=null) queue.add(top.right);

         TreeNode tmp = top.left;
         top.left = top.right;
         top.right = tmp;
      }
      return root;
   }
}
```

18.12 lowest Common Ancestor of a Binary Search Tree (Easy->leetcode NO.235)

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.

According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow **a node to be a descendant of itself**)."



For example, the lowest common ancestor (LCA) of nodes `2` and `8` is `6`. Another example is LCA of nodes `2` and `4` is `2`, since a node can be a descendant of itself according to the LCA definition.

Solution 1:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if(root==null) return null;
        if(root==p || root==q) return root;
        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);
        return left!=null && right!=null ? root:left==null?right:left;
    }
}
```

Solution 2: use the property of BST

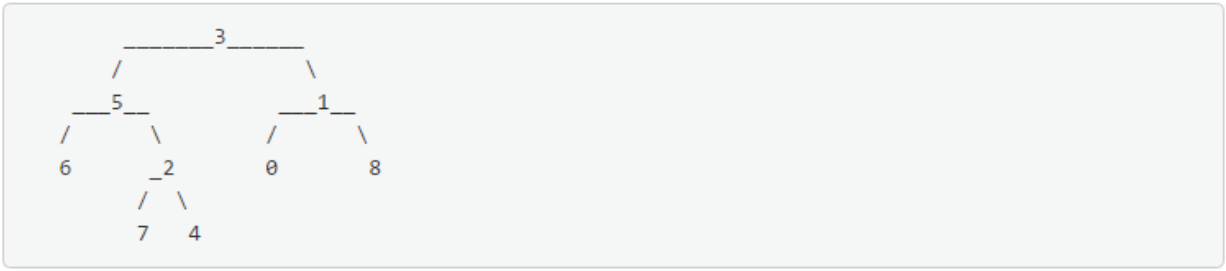
```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
```

```
  public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    if(root==null) return null;
    TreeNode tmp = root;
    if(tmp.val>p.val && tmp.val<q.val) return tmp;
    else if(tmp.val>p.val && tmp.val>q.val) return lowestCommonAncestor(tmp.left, p, q);
    else if (tmp.val<p.val && tmp.val<q.val) return lowestCommonAncestor(tmp.right, p, q);
    return root;
  }
}
```

18.13 Lowest Common Ancestor of A Binary Tree (Medium->leetcode No.236)

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

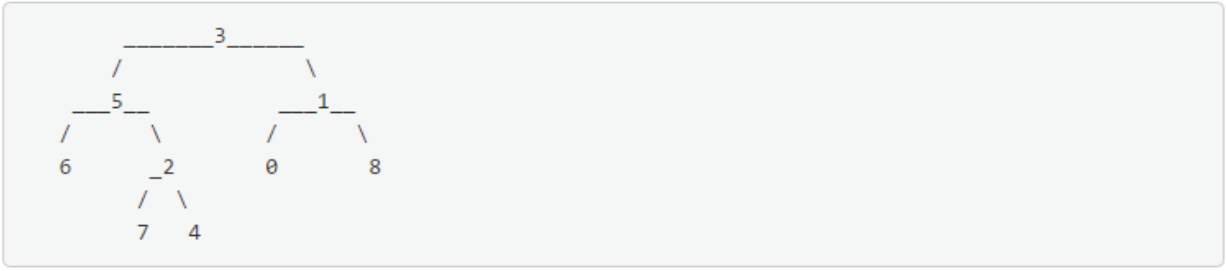
According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow **a node to be a descendant of itself**)."



For example, the lowest common ancestor (LCA) of nodes `5` and `1` is `3`. Another example is LCA of nodes `5` and `4` is `5`, since a node can be a descendant of itself according to the LCA definition.

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow **a node to be a descendant of itself**)."



For example, the lowest common ancestor (LCA) of nodes `5` and `1` is `3`. Another example is LCA of nodes `5` and `4` is `5`, since a node can be a descendant of itself according to the LCA definition.

Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if(root==null) return null;
        if(root==p || root==q) return root;
        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);
        return left!=null && right!=null ? root:left==null?right:left;
    }
}
```

18.14 Different Ways to Add Parentheses (Medium->leetcode NO.241)

Given a string of numbers and operators, return all possible results from computing all the different possible ways to group numbers and operators. The valid operators are `+`, `-` and `*`.

Example 1

Input: `"2-1-1"`.

```
((2-1)-1) = 0
(2-(1-1)) = 2
```

Output: `[0, 2]`

Example 2

Input: `"2*3-4*5"`

```
(2*(3-(4*5))) = -34
((2*3)-(4*5)) = -14
((2*(3-4))*5) = -10
(2*((3-4)*5)) = -10
(((2*3)-4)*5) = 10
```

Output: `[-34, -14, -10, -10, 10]`

Solution

```
public class Solution {
```

```
  private Map<String, List<Integer>> map = new HashMap<>();
  public List<Integer> diffWaysToCompute(String input) {
    if(map.containsKey(input)) return map.get(input);
    List<Integer> res = new ArrayList<>();
    for(int i=0;i<input.length();i++){
      char c = input.charAt(i);
      if(c=='+' || c=='-' || c=='*'){
        List<Integer> list1 = diffWaysToCompute(input.substring(0,i));
        List<Integer> list2 = diffWaysToCompute(input.substring(i+1));
        for(int v1:list1){
          for(int v2:list2){
            if(c=='+') res.add(v1+v2);
            if(c=='-') res.add(v1-v2);
            if(c=='*') res.add(v1*v2);
          }
        }
      }
    }
    if(res.isEmpty()) res.add(Integer.parseInt(input));
    map.put(input, res);
    return res;
  }
}
```

18.15 Additive Number (Medium->leetcode NO.306)

Additive number is a string whose digits can form additive sequence.

A valid additive sequence should contain **at least** three numbers. Except for the first two numbers, each subsequent number in the sequence must be the sum of the preceding two.

For example:

`"112358"` is an additive number because the digits can form an additive sequence: `1, 1, 2, 3, 5, 8`.

```
1 + 1 = 2, 1 + 2 = 3, 2 + 3 = 5, 3 + 5 = 8
```

`"199100199"` is also an additive number, the additive sequence is: `1, 99, 100, 199`.

```
1 + 99 = 100, 99 + 100 = 199
```

Note: Numbers in the additive sequence **cannot** have leading zeros, so sequence `1, 2, 03` or `1, 02, 3` is invalid.

Given a string containing only digits `'0'-'9'`, write a function to determine if it's an additive number.

Follow up:

How would you handle overflow for very large input integers?

Solution 1: Recursive version

```
public class Solution {
    public boolean isAdditiveNumber(String num) {
        if (num == null || num.length() <= 2) return false;

        //[0,i] is first number, [i+1,j] is second number,[j+1 any end is remaining]
        for (int i = 0; i < (num.length() - 1) / 2; i++) {
            for (int j = i + 1; num.length() - j - 1 >= Math.max(i + 1, j - i); j++) {
                if (isValid(num.substring(0, i + 1), num.substring(i + 1, j + 1), num.substring(j + 1)))
                    return true;
                }
            }
        return false;
    }

    public boolean isValid(String num1, String num2, String remain) {
        if (remain.isEmpty()) return true;
        if (num1.charAt(0) == '0' && num1.length() > 1) return false;
        if (num2.charAt(0) == '0' && num2.length() > 1) return false;
        String sum = String.valueOf(Long.parseLong(num1) + Long.parseLong(num2));
        if (!remain.startsWith(sum)) return false;
        return isValid(num2, sum, remain.substring(sum.length()));
    }
}
```

Solution 2: iterative version

```
public class Solution {
    public boolean isAdditiveNumber(String num) {
    if (num == null || num.length() <= 2) return false;

    //[0,i] is first number, [i+1,j] is second number,[j+1 any end is remaining]
    for (int i = 0; i < (num.length() - 1) / 2; i++) {
        for (int j = i + 1; num.length() - j - 1 >= Math.max(i + 1, j - i); j++) {
            int offset = j + 1;
            String num1 = num.substring(0, i + 1), num2 = num.substring(i + 1, j + 1);

            while (offset < num.length()) {
                if (num1.charAt(0) == '0' && num1.length() > 1) break;
                if (num2.charAt(0) == '0' && num2.length() > 1) break;
                String sum = String.valueOf(Long.parseLong(num1) + Long.parseLong(num2));
                if (!num.startsWith(sum, offset)) break;

                num1 = num2;
                num2 = sum;
                offset += sum.length();
            }
            if (offset == num.length()) return true;
        }
    }
```

```
    return false;
}
}
```

18.16 Wiggle Sort II (Medium->leetcode No.324)

Given an unsorted array `nums`, reorder it such that `nums[0] < nums[1] > nums[2] < nums[3]...`.

Example:

(1) Given `nums = [1, 5, 1, 1, 6, 4]`, one possible answer is `[1, 4, 1, 5, 1, 6]`.

(2) Given `nums = [1, 3, 2, 2, 3, 1]`, one possible answer is `[2, 3, 1, 3, 1, 2]`.

Note:

You may assume all input has valid answer.

Follow Up:

Can you do it in O(n) time and/or in-place with O(1) extra space?

Solution

```
import java.util.Random;
public class Solution {
int m = 0;
public void wiggleSort(int[] nums) {
    if (nums.length < 2) return;
    int lt = 0, gt = nums.length-1, i = 0;
    m = (gt - 1)/2; //used for virtual index mapping, the number of items that are larger than mid should
<= the number of items smaller than mid.
    int mid = quickSelect(nums, gt/2);
    while (i <= gt) { //put the larger items in the left to handle the problem of duplicate items with the
same value of mid, such as [4,5,5,6]
        if (nums[idx(i)] > mid) swap(nums, idx(i++), idx(lt++));
        else if (nums[idx(i)] < mid) swap(nums, idx(i), idx(gt--));
        else i++;
    }
}
private void swap(int[] nums, int x, int y) {
    int temp = nums[x];
    nums[x] = nums[y];
    nums[y] = temp;
}
private int idx(int i) { //idea from https://leetcode.com/discuss/77133/o-n-o-1-after-median-virtual-
indexing
    if (i <= m) return 2*i+1;
    else return 2*(i-m-1);
}
private int quickSelect(int[] a, int k) {
    randomShuffle(a);
```

```
    int lo = 0, hi = a.length-1;
    while (lo < hi) {
        int i = partition(a, lo, hi);
        if (i > k) hi = i - 1;
        else if (i < k) lo = i + 1;
        else return a[i];
    }
    return a[lo];
}
private int partition(int[] a, int lo, int hi) {
    int i = lo, j = hi + 1;
    int pivot = a[lo];
    while(true) {
        while(a[++i] < pivot)
            if (i == hi) break;
        while(a[--j] > pivot)
            if (j == lo) break;
        if (i >= j) break;
        swap(a, i, j);
    }
    swap(a, lo, j);
    return j;
}
private void randomShuffle(int[] nums) {
    Random rm = new Random();
    for(int i = 0; i < nums.length-1; i++) {
        int j = i + rm.nextInt(nums.length-i);
        swap(nums, i, j);
    }
}
}
```

18.17 Increasing Triplet Subsequence (Medium->leetcode NO.334)

Given an unsorted array return whether an increasing subsequence of length 3 exists or not in the array.

Formally the function should:

> Return true if there exists *i*, *j*, *k*
> such that *arr[i]* < *arr[j]* < *arr[k]* given 0 ≤ *i* < *j* < *k* ≤ *n*-1 else return false.

Your algorithm should run in O(*n*) time complexity and O(*1*) space complexity.

Examples:

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

return `true`.

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

return `false`.

Solution

```
public class Solution {
   public boolean increasingTriplet(int[] nums) {
      if(nums==null || nums.length<3) return false;
      int min1 = nums[0];
      int min2 = Integer.MAX_VALUE;
      for(int i=1;i<nums.length;i++){
         if(nums[i]>min2) return true;
         else if(nums[i]>min1 && nums[i]<min2) min2 = nums[i];
         else if(nums[i]<min1) min1 = nums[i];
      }
      return false;
   }
}
```

18.18 Design Twitter (Medium->leetcode No.355)

Design a simplified version of Twitter where users can post tweets, follow/unfollow another user and is able to see the 10 most recent tweets in the user's news feed. Your design should support the following methods:

1. **postTweet(userId, tweetId)**: Compose a new tweet.
2. **getNewsFeed(userId)**: Retrieve the 10 most recent tweet ids in the user's news feed. Each item in the news feed must be posted by users who the user followed or by the user herself. Tweets must be ordered from most recent to least recent.
3. **follow(followerId, followeeId)**: Follower follows a followee.
4. **unfollow(followerId, followeeId)**: Follower unfollows a followee.

Example:

```
Twitter twitter = new Twitter();

// User 1 posts a new tweet (id = 5).
twitter.postTweet(1, 5);

// User 1's news feed should return a list with 1 tweet id -> [5].
twitter.getNewsFeed(1);

// User 1 follows user 2.
twitter.follow(1, 2);

// User 2 posts a new tweet (id = 6).
twitter.postTweet(2, 6);

// User 1's news feed should return a list with 2 tweet ids -> [6, 5].
// Tweet id 6 should precede tweet id 5 because it is posted after tweet id 5.
twitter.getNewsFeed(1);

// User 1 unfollows user 2.
twitter.unfollow(1, 2);

// User 1's news feed should return a list with 1 tweet id -> [5],
// since user 1 is no longer following user 2.
twitter.getNewsFeed(1);
```

Solution 1:

```
public class Twitter {

    private int postCount;
    private Map<Integer, Set<Integer>> followeeMap = null;
    private Map<Integer, List<Integer>> tweetIdMap = null;
    private Map<Integer, Integer> tweetOwnerMap = null;
```

```
    private Map<Integer, Integer> tweetCountMap = null;
    /** Initialize your data structure here. */
    public Twitter() {
        followeeMap = new HashMap<>();
        tweetIdMap = new HashMap<>();
        tweetOwnerMap = new HashMap<>();
        tweetCountMap = new HashMap<>();
    }

    /** Compose a new tweet. */
    public void postTweet(int userId, int tweetId) {
        tweetCountMap.put(userId, ++postCount);
        tweetOwnerMap.put(tweetId, userId);
        getTweetList(userId).add(tweetId);
    }

    private List<Integer> getTweetList(Integer userId){
        List<Integer> tmp = tweetIdMap.get(userId);
        if(tmp==null){
            tmp = new ArrayList<>();
            tweetIdMap.put(userId, tmp);
        }
        return tmp;
    }

    /** Retrieve the 10 most recent tweet ids in the user's news feed. Each item in the news feed must be posted by users who the user followed or by the user herself. Tweets must be ordered from most recent to least recent. */
    public List<Integer> getNewsFeed(int userId) {
        Set<Integer> followeeSet = getFolloweeSet(userId);
        List<Integer> result = new ArrayList<>();
        Queue<Integer> pq = new PriorityQueue<Integer>(new Comparator<Integer>(){
            @Override
            public int compare(Integer a, Integer b){
                return tweetCountMap.get(b)-tweetCountMap.get(a);
            }
        });

        Map<Integer, Integer> idxMap = new HashMap<Integer, Integer>();
        for (Integer followeeId : followeeSet) {
            List<Integer> tweetIdList = getTweetList(followeeId);
            int idx = tweetIdList.size() - 1;
            if (idx >= 0) {
                idxMap.put(followeeId, idx - 1);
                pq.add(tweetIdList.get(idx));
            }
        }
        while (result.size() < 10 && !pq.isEmpty()) {
```

```
            Integer topTweetId = pq.poll();
            result.add(topTweetId);
            Integer ownerId = tweetOwnerMap.get(topTweetId);
            int idx = idxMap.get(ownerId);
            if (idx >= 0) {
                List<Integer> tweetIdList = getTweetList(ownerId);
                pq.add(tweetIdList.get(idx));
                idxMap.put(ownerId, idx - 1);
            }
        }
        return result;
    }
    private Set<Integer> getFolloweeSet(int userId){
        Set<Integer> set = followeeMap.get(userId);
        if(set==null){
            set = new HashSet<>();
            set.add(userId);
            followeeMap.put(userId, set);
        }
        return set;
    }

    /** Follower follows a followee. If the operation is invalid, it should be a no-op. */
    public void follow(int followerId, int followeeId) {
        getFolloweeSet(followerId).add(followeeId);
    }

    /** Follower unfollows a followee. If the operation is invalid, it should be a no-op. */
    public void unfollow(int followerId, int followeeId) {
        if(followerId!=followeeId) getFolloweeSet(followerId).remove(followeeId);
    }
}

/**
 * Your Twitter object will be instantiated and called as such:
 * Twitter obj = new Twitter();
 * obj.postTweet(userId,tweetId);
 * List<Integer> param_2 = obj.getNewsFeed(userId);
 * obj.follow(followerId,followeeId);
 * obj.unfollow(followerId,followeeId);
 */
```

Solution 2: Easy

```
public class Twitter {
    private static class Tweet{
        int tweetId;
        int userId;
        Tweet(int tweetId, int userId){
```

```
        this.tweetId = tweetId;
        this.userId = userId;
      }
    }

    private final static int NEWS_FEED_SIZE = 10;
    private Deque<Tweet> tweets = new ArrayDeque<>();
    private Map<Integer, Set<Integer>> followeeByFollower = new HashMap<>();
    /** Initialize your data structure here. */
    public Twitter() {

    }

    /** Compose a new tweet. */
    public void postTweet(int userId, int tweetId) {
      tweets.add(new Tweet(tweetId, userId));
    }

    /** Retrieve the 10 most recent tweet ids in the user's news feed. Each item in the news feed must be
posted by users who the user followed or by the user herself. Tweets must be ordered from most recent
to least recent. */
    public List<Integer> getNewsFeed(int userId) {
      List<Integer> result = new ArrayList<>(NEWS_FEED_SIZE);
      Set<Integer> followees = followeeByFollower.get(userId);
      for(Iterator<Tweet> it = tweets.descendingIterator(); it.hasNext() &&
result.size()<NEWS_FEED_SIZE;){
        Tweet tweet = it.next();
        if(tweet.userId==userId || (followees!=null && followees.contains(tweet.userId)))
result.add(tweet.tweetId);
      }
      return result;
    }

    /** Follower follows a followee. If the operation is invalid, it should be a no-op. */
    public void follow(int followerId, int followeeId) {
      if(followeeByFollower.containsKey(followerId)){
        followeeByFollower.get(followerId).add(followeeId);
      }else{
        Set<Integer> followees = new HashSet<>();
        followees.add(followeeId);
        followeeByFollower.put(followerId, followees);
      }
    }

    /** Follower unfollows a followee. If the operation is invalid, it should be a no-op. */
    public void unfollow(int followerId, int followeeId) {
      if(followeeByFollower.containsKey(followerId)){
        Set<Integer> set = followeeByFollower.get(followerId);
```

```
            if(set.contains(followeeId)) set.remove(followeeId);
        }
    }
}

/**
 * Your Twitter object will be instantiated and called as such:
 * Twitter obj = new Twitter();
 * obj.postTweet(userId,tweetId);
 * List<Integer> param_2 = obj.getNewsFeed(userId);
 * obj.follow(followerId,followeeId);
 * obj.unfollow(followerId,followeeId);
 */
```