Leetcode Offline
----------Single Element in a Sorted Array
----------Given a sorted array consisting of only integers where every element appears twice except for one element which appears once. Find this
single element that appears only once.
Example 1:
Input: [1,1,2,3,3,4,4,8,8]
Output: 2

Example 2:
Input: [3,3,7,7,10,11,11]
Output: 10

Note:
Your solution should run in O(log n) time and O(1) space.

```
class Solution {
    //O[n/2] time complexity
        public int singleNonDuplicate(int[] nums) {

                /*
                 * Condition below are
                 *
                 * if array length is one return first index
                 * if array length is 3 and if 1st and 2nd index value are equal return 3rd index value else return 1st index value
                 * if first two values are not same return 1st value
                 * if last two values are same return last value
                 */
                if(nums.length==1)
                        return nums[0];
                else if(nums.length==3){
                        if(nums[0]==nums[1])
                                return nums[2];
                        return nums[0];
                }else if(nums[0]!=nums[1])
                        return nums[0];
                else if(nums[nums.length-1] != nums[nums.length-2])
                        return nums[nums.length-1];

                int pointer = nums.length/2;
                boolean flag = true,reverse=false;

                /*
                 * validating forward or  reverse traversal with below condition
                 *
                 * if middle value and its next value are same and
                 *                 pointer is odd(means left array will be even) so traversal should be forward
                 * else middle value and its previous value are same and
                 *                 pointer is even (means left array will be odd) so traversal should be reverse
                 * else means middle pointer id single element
                 */
```

```
                    if(nums[pointer] == nums[pointer-1]) {
                            if(pointer%2==0) {
                                    pointer--;
                                    reverse=true;
                            }
                    }else if(nums[pointer] ==nums[pointer+1]) {
                            if(pointer%2==1) reverse=true;
                    }
                    else  return nums[pointer];


                    /*
                     *if the current and previous/next pointer is not equal flag is set to true, return
the index value if flag is already set as true
                     *else if equal set flag as false;
                     */
                    if(reverse) {
                            while(--pointer>1) {
                                    if(nums[pointer] != nums[pointer-1]) {
                                            if(flag) return nums[pointer];
                                            flag=true;
                                    }else flag = false;
                            }
                    }else {
                            while(++pointer<nums.length-2) {
                                    if(nums[pointer] != nums[pointer+1]) {
                                            if(flag) return nums[pointer];
                                            flag=true;
                                    }else flag = false;
                            }
                    }

                    //returning zero to avoid complier error
                    return 0;
            }
}

class Solution {
    public int singleNonDuplicate(int[] nums) {
        // initialize
        int low = 0,high = nums.length-1,mid = 0;

        // edge case check
        if(nums.length == 1)return nums[0];
        if(nums[low] != nums[low + 1])return nums[low];
        if(nums[high] != nums[high - 1])return nums[high];

        // Binary Search
        while(low <= high){
            mid = low + (high - low)/2;
            // found single element
            if(nums[mid] != nums[mid - 1] && nums[mid] != nums[mid + 1])return nums[mid];
            //left side check
            else if((nums[mid] == nums[mid + 1] && mid % 2 == 0) || (
```

```
            nums[mid] == nums[mid - 1] && mid % 2 != 0))low = mid + 1;
          //right side check
          else high = mid - 1;
      }

      return -1;
   }
}
```

----------Coin Change 2
----------You are given coins of different denominations and a total amount of money. Write a function to compute the number of combinations that
make up that amount. You may assume that you have infinite number of
each kind of coin.
Note:
You can assume that
0 <= amount <= 5000
1 <= coin <= 5000
the number of coins is less than 500
the answer is guaranteed to fit into signed 32-bit integer

Example 1:
Input: amount = 5, coins = [1, 2, 5]

Output: 4
Explanation: there are four ways to make up the amount:
5=5
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1
Example 2:
Input: amount = 3, coins = [2]
Output: 0
Explanation: the amount of 3 cannot be made up just with coins of 2.
Example 3:
Input: amount = 10, coins = [10]
Output: 1

```
class Solution {
    public int change(int amount, int[] coins) {
        // create an int array of length amount+1
                int[] dp = new int[amount+1];
       dp[0] = 1;

                // update the array with the minimum number of coins needed at each amount
                for(int coin: coins) {
                        for(int i = coin; i < dp.length; i++) {
                                dp[i] += dp[i-coin];
                        }
                }

                return dp[dp.length-1];
    }
}
```

----------Largest Palindrome Product
** DO IT LATER **
----------Find the largest palindrome made from the product of two n-digit numbers.
Since the result could be very large, you should return the largest
palindrome mod 1337.
Example:
Input: 2
Output: 987
Explanation: 99 x 91 = 9009, 9009 % 1337 = 987

Note:
The range of n is [1,8].
----------Find All Duplicates in an Array
----------Given an array of integers, 1 ≤ a[i] ≤ n (n = size of array), some elements appear twice and others appear once.
Find all the elements that appear twice in this array.
Could you do it without extra space and in O(n) runtime?
Example:

Input:
[4,3,2,7,8,2,3,1]
Output:
[2,3]

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

                    for(int i=0; i<nums.length; i++) {
                            int abs = Math.abs(nums[i]);
                            if(nums[abs-1]<0) {
                                    ans.add(abs);
                            }
                            else {
                                    nums[abs-1] = -1*nums[abs-1];
                            }
                    }

                    return ans;
    }
}
```

----------Two Sum
----------Given an array of integers, return indices of the two numbers such
that they add up to a specific target.
You may assume that each input would have exactly one solution, and
you may not use the same element twice.
Example:
Given nums = [2, 7, 11, 15], target = 9,
Because nums[0] + nums[1] = 2 + 7 = 9,
return [0, 1].

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> map = new HashMap<>();
        for(int i=0; i< nums.length; i++){
            int potentialMatch = target-nums[i];
            if(map.containsKey(potentialMatch))
                return new int [] {map.get(potentialMatch), i};
            map.put(nums[i], i);
        }
        return new int[] {0, 0};
    }
}
```

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

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

        int sum=0, carry=0;
        ListNode result=null, temp=null;

        while(l1!=null || l2!=null) {
            sum = 0;
            if(l1!=null) {
                sum += l1.val;
                l1 = l1.next;
            }
            if(l2!=null) {
                sum += l2.val;
                l2 = l2.next;
            }

            sum += carry;
            carry = sum/10;
            sum = sum%10;

            if(result==null) {
                result = new ListNode(sum);
                temp = result;
            } else {
                temp.next = new ListNode(sum);
                temp = temp.next;
            }
        }
        if(carry!=0) {
            temp.next = new ListNode(carry);
        }
        return result;
    }
}
```

----------Longest Substring Without Repeating Characters

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

Examples:

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

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

Given "pwwkew", the answer is "wke", with the length of 3. Note that

the answer must be a substring, "pwke" is a subsequence and not a substring.

```
class Solution {
    public int lengthOfLongestSubstring(String s) {
        int l = 0;
        int r = l;
        int res = 0;
        int[] set = new int[128];
        while(r<s.length()){
            char c = s.charAt(r);
            if(set[128-c]>0){
                while(l<r && set[128-c]>0){
                    set[128-s.charAt(l)]--;
                    l++;
                }
            }
            set[128-c]++;
            res = Math.max(res,r-l+1);
            r++;
        }
        return res;
    }
}
```

```
public int lengthOfLongestSubstring(String s) {
        Map<Character, Integer> lookup = new HashMap<>();
        int result = 0;
        int left = 0;
        for(int right = 0; right < s.length(); right++) {
            if(lookup.containsKey(s.charAt(right))) {
                left = Math.max(left, lookup.get(s.charAt(right)));
            }
            result = Math.max(result, right - left + 1);
            lookup.put(s.charAt(right), right + 1);
        }
        return result;
    }
```

----------Median of Two Sorted Arrays
----------There are two sorted arrays nums1 and nums2 of size m and n respectively.
Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).
Example 1:
nums1 = [1, 3]
nums2 = [2]
The median is 2.0

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

```
class Solution {
```

```
  public double findMedianSortedArrays(int[] a, int[] b) {

      int len = a.length + b.length;

      int aIndex = 0;  // Iterator for nums1 array
      int bIndex = 0;  // Iterator for nums2 array

      int previous = 0;           // for even case
      int current = 0;            // for odd and even case
      int resultIndex = 0;        // merged sorted array Index

      while(resultIndex++ <= (len / 2)) {     // LOOP UP TO (M + N) / 2

          previous = current;          // assign current value to previous

          if(aIndex >= a.length) {     // if aIndex >= a length then only use b;
              current = b[bIndex++];   // update current value and increment bIndex;
              continue;
          }

          if(bIndex >= b.length ) {    // similar as above,for bIndex
              current = a[aIndex++];
              continue;
          }

          if(a[aIndex] < b[bIndex]) {  // marge method;
              current = a[aIndex++];
          } else {
              current = b[bIndex++];
          }
      }

      if(len % 2 == 0) {
          return (double) (previous  + current)/2;   // if len is even
      } else {
          return (double) current;                   // if len is odd
      }
   }
}
```

----------Longest Palindromic Substring----------Given a string s, find the longest palindromic substring in s. You may assume that the maximum length of s is 1000.

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

Example:
Input: "cbbd"
Output: "bb"

```
class Solution {
    public String longestPalindrome(String S) {
```

```
        char[] A = new char[2 * S.length() + 3];
        A[0] = '@';
        A[1] = '#';
        A[A.length - 1] = '$';
        int t = 2;
        for (char c: S.toCharArray()) {
            A[t++] = c;
            A[t++] = '#';
        }

        int[] Z = new int[A.length];
        int center = 0, right = 0, maxZ = 0, maxCenter = 0;
        for (int i = 1; i < Z.length - 1; ++i) {
            if (i < right)
                Z[i] = Math.min(right - i, Z[2 * center - i]);
            while (A[i + Z[i] + 1] == A[i - Z[i] - 1])
                Z[i]++;
            if (i + Z[i] > right) {
                center = i;
                right = i + Z[i];
            }
            if(Z[i] > maxZ) {
                maxZ = Z[i];
                maxCenter = i;
            }
        }
        StringBuilder sb = new StringBuilder();
        for(int i = maxCenter - Z[maxCenter] + 1; i < maxCenter + Z[maxCenter] + 1; i+=2) {
            sb.append(A[i]);
        }
        return sb.toString();
    }
}
```

------------------------------ZigZag Conversion
------------------------------The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)
P A H N
APLSIIG
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".

```
class Solution {
    public String convert(String s, int numRows) {
        int slen = s.length();
        int period = 2 * numRows - 2;
        if(period == 0)return s;

        char res[] = new char[slen];
        int index = 0;

        //line 0
        for(int i=0; i < slen; i+=period){
            res[index++]= s.charAt(i);
        }
        //middle lines: line x
        for(int x=1; x <= numRows - 2; x++){
            for(int i=0; i<slen; i+=period){

                if(i+x >= slen)break;
                res[index++]=s.charAt(i + x);

                if(i+period-x >= slen)break;
                res[index++]=s.charAt(i+period-x);
            }
        }

        //last row
        for(int i=numRows-1; i < slen; i+=period){
            res[index++]=s.charAt(i);
        }

        return new String(res);
    }
}


/*

P    R
A   II
Y  H N
P S  G
```

AI
L

rows = 6; ladder chars = 6-2=4

|6 /4 |6 /4 |6 /4

pattern repeats after 10 chars
line 0 : 10n + 0
line 5 : 10n + 5

line 1 : 10n + 1, 10n + 9
line 2 : 10n + 2, 10n + 8
...
line 4 : 10n + 4, 10n + 6

period = 2*rows - 2
line 0   : period*n + 0
line row-1 : period*n + row-1

x <- 1 to row-2
    line x : period*n + x, period*n + period - x
*/

--Reverse Integer
----------------------------Reverse digits of an integer.

Given a signed 32-bit integer x, return x with its digits reversed.
If reversing x causes the value to go outside the signed 32-bit integer range [-231, 231 - 1],
then return 0.
Assume the environment does not allow you to store 64-bit integers (signed or unsigned).

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

Note:
The input is assumed to be a 32-bit signed integer. Your function
should return 0 when the reversed integer overflows.

```
class Solution {
    public int reverse(int x) {

        int reverseDigits = 0;

        while(x != 0 ) {
                            //check if reverseDigits will overflow
            if(reverseDigits > Integer.MAX_VALUE/10 || reverseDigits < Integer.MIN_VALUE/10)
                return 0;

                            //Update reverseDigits
            reverseDigits = (10*reverseDigits) + (x%10);
            x/=10;
        }

        return reverseDigits;

    }
}

class Solution {
    public int reverse(int x) { return (x == 0) ? 0 : process(x, 0); }

private int process(int x, int output) {
    if (output < Integer.MIN_VALUE/10 || output > Integer.MAX_VALUE/10) return 0;
    else if (x/10 == 0) return (output * 10) + (x % 10);
    else return process(x/10, (output * 10) + x % 10);
}
}
```

----------String to Integer (atoi)
----------Implement atoi to convert a string to an integer.

Implement the myAtoi(string s) function, which converts a string to a 32-bit signed integer
(similar to C/C++'s atoi function).

The algorithm for myAtoi(string s) is as follows:

Read in and ignore any leading whitespace.
Check if the next character (if not already at the end of the string) is '-' or '+'. Read this
character in if it is either.
This determines if the final result is negative or positive respectively. Assume the result is
positive if neither is present.
Read in next the characters until the next non-digit character or the end of the input is reached.
The rest of the string is ignored.
Convert these digits into an integer (i.e. "123" -> 123, "0032" -> 32). If no digits were read, then
the integer is 0.
Change the sign as necessary (from step 2).
If the integer is out of the 32-bit signed integer range [-231, 231 - 1], then clamp the integer so
that it remains in the range.

Specifically, integers less than -231 should be clamped to -231, and integers greater than 231 - 1 should be clamped to 231 - 1.
Return the integer as the final result.

Note:

Only the space character ' ' is considered a whitespace character.
Do not ignore any characters other than the leading whitespace or the rest of the string after the digits.

Example 1:

Input: s = "42"
Output: 42
Explanation: The underlined characters are what is read in, the caret is the current reader position.
Step 1: "42" (no characters read because there is no leading whitespace)
^
Step 2: "42" (no characters read because there is neither a '-' nor '+')
^
Step 3: "42" ("42" is read in)
^
The parsed integer is 42.
Since 42 is in the range [-231, 231 - 1], the final result is 42.
Example 2:

Input: s = "   -42"
Output: -42
Explanation:
Step 1: "   -42" (leading whitespace is read and ignored)
^
Step 2: "   -42" ('-' is read, so the result should be negative)
^
Step 3: "   -42" ("42" is read in)
^
The parsed integer is -42.
Since -42 is in the range [-231, 231 - 1], the final result is -42.
Example 3:

Input: s = "4193 with words"
Output: 4193
Explanation:
Step 1: "4193 with words" (no characters read because there is no leading whitespace)
^
Step 2: "4193 with words" (no characters read because there is neither a '-' nor '+')
^
Step 3: "4193 with words" ("4193" is read in; reading stops because the next character is a non-digit)
^
The parsed integer is 4193.
Since 4193 is in the range [-231, 231 - 1], the final result is 4193.
Example 4:

Input: s = "words and 987"
Output: 0
Explanation:
Step 1: "words and 987" (no characters read because there is no leading whitespace)
^
Step 2: "words and 987" (no characters read because there is neither a '-' nor '+')
^
Step 3: "words and 987" (reading stops immediately because there is a non-digit 'w')
^
The parsed integer is 0 because no digits were read.
Since 0 is in the range [-231, 231 - 1], the final result is 0.
Example 5:

Input: s = "-91283472332"
Output: -2147483648
Explanation:
Step 1: "-91283472332" (no characters read because there is no leading whitespace)
^
Step 2: "-91283472332" ('-' is read, so the result should be negative)
^
Step 3: "-91283472332" ("91283472332" is read in)
^
The parsed integer is -91283472332.
Since -91283472332 is less than the lower bound of the range [-231, 231 - 1], the final result is clamped to -231 = -2147483648.

Constraints:

0 <= s.length <= 200
s consists of English letters (lower-case and upper-case), digits (0-9), ' ', '+', '-', and '.'.

Hint: Carefully consider all possible input cases. If you want a challenge, please do not see below and ask yourself what are the possible input cases.
Notes:
It is intended for this problem to be specified vaguely (ie, no given input specs). You are responsible to gather all the input requirements up front.
Update (2015-02-10):
The signature of the C++ function had been updated. If you still see your function signature accepts a const char * argument, please click the reload button to reset your code definition.
spoilers alert... click to show requirements for atoi.
Requirements for atoi:
The function first discards as many whitespace characters as necessary until the first non-whitespace character is found. Then, starting from this character, takes an optional initial plus or minus sign followed by as many numerical digits as possible, and interprets them as a numerical value.
The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function.
If the first sequence of non-whitespace characters in str is not a

valid integral number, or if no such sequence exists because either
str is empty or it contains only whitespace characters, no
conversion is performed.
If no valid conversion could be performed, a zero value is returned.
If the correct value is out of the range of representable values,
INT_MAX (2147483647) or INT_MIN (-2147483648) is returned.

```
class Solution {

    // a function that makes sure that the character being analysed is a valid character or not
    public boolean isCharacterValid(char ch){
        if (ch == '-' || ch == '+')
            return true;
                    //we check if the character lies in the range of ascii values of 48 and 57 as ascii
of 0 is 48 and ascii of 9 is 57
        else if( ch>= 48 && ch < 58)
            return true;
        else
            return false;
    }

    public int myAtoi(String str) {

        boolean start = false;
        int sign = 1;
        long no = 0;

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

            char ch = str.charAt(i);

            //for detecting the first character
            if (start == false && isCharacterValid(ch)){
                start = true;
                if(ch == '-')
                    sign = -1;
                else if(ch == '+')
                    sign = 1;
                else
                    no = no*10+((int)ch-48);
            }

            //incase its a valid character and in sequence
            else if(start == true && isCharacterValid(ch)){

                if(Character.isDigit(ch))
                    no = no*10+((int)ch - 48);
                else if (ch == '-' || ch == '+')
                    break;

            }

            //incase its an invalid character, we will break only if its a non-whitespacce character
            else if(start == true && isCharacterValid(ch) == false)
```

```
            break;

        //incase its a whitespace character and we have not started counting continue
        else if(start == false && isCharacterValid(ch) == false && ch == ' ')
            continue;

        //for all other cases return 0
        else
            return 0;


        //during each loop keep checcking if the number formed is in the limit or not
        if(no>Integer.MAX_VALUE){

            if (sign == -1)
                return Integer.MIN_VALUE;
            else
                return Integer.MAX_VALUE;

        }

    }//end for

    //return the final formed number.
    return (int)(sign*no);
  }
}
```

----------Palindrome Number

Determine whether an integer is a palindrome. Do this without extra space.

Example 1:

Input: x = 121
Output: true

Example 2:

Input: x = -121
Output: false
Explanation: From left to right, it reads -121. From right to left, it becomes 121-. Therefore it is not a palindrome.

Example 3:

Input: x = 10
Output: false
Explanation: Reads 01 from right to left. Therefore it is not a palindrome.

Example 4:

Input: x = -101
Output: false

Constraints:

-231 <= x <= 231 - 1

click to show spoilers.
Some hints:
Could negative integers be palindromes? (ie, -1)
If you are thinking of converting the integer to string, note the
restriction of using extra space.
You could also try reversing an integer. However, if you have solved
the problem "Reverse Integer", you know that the reversed integer
might overflow. How would you handle such case?
There is a more generic way of solving this problem.

```
/**
 * Reverse Half & Compare
 *
 * Time Complexity: O((log10 N) / 2)
 *
 * Space Complexity: O(1)
 *
 * N = Number of digits in input number.
 */
class Solution {
    public boolean isPalindrome(int x) {
        if (x < 0 || (x != 0 && x % 10 == 0)) {
            return false;
        }
        if (x < 10) {
            return true;
        }

        int reverse = 0;
        while (reverse < x) {
            reverse = reverse * 10 + x % 10;
            x /= 10;
        }

       /**
         * If input number has even number of digits then check `x == reverse`.
         *
         * If input number has odd number of digits then check `x == reverse / 10`. This
         * is because, reverse will have one extra digit. Middle digit of original
         * number will be least significant digit of reverse.
         */
        return reverse == x || reverse / 10 == x;
    }
}
```

----------Regular Expression matching
----------Implement regular expression matching with support for '.' and '*'.

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

```
class Solution {
  public boolean isMatch(String s, String p) {

    if (s == null || p == null) {
        return false;
    }
    boolean[][] dp = new boolean[s.length()+1][p.length()+1];
    dp[0][0] = true;
    for (int i = 0; i < p.length(); i++) {
        if (p.charAt(i) == '*' && dp[0][i-1]) {
            dp[0][i+1] = true;
        }
    }
    for (int i = 0 ; i < s.length(); i++) {
        for (int j = 0; j < p.length(); j++) {
            if (p.charAt(j) == '.') {
                dp[i+1][j+1] = dp[i][j];
            }
            if (p.charAt(j) == s.charAt(i)) {
                dp[i+1][j+1] = dp[i][j];
            }
            if (p.charAt(j) == '*') {
                if (p.charAt(j-1) != s.charAt(i) && p.charAt(j-1) != '.') {
                    dp[i+1][j+1] = dp[i+1][j-1];
                } else {
                    dp[i+1][j+1] = (dp[i+1][j] || dp[i][j+1] || dp[i+1][j-1]);
                }
            }
        }
    }
    return dp[s.length()][p.length()];
}
}
```

----------Container With Most Water
----------Given n non-negative integers a1, a2, ..., an, where each represents a point at coordinate (i, ai). n vertical lines are drawn such that
the two endpoints of line i is at (i, ai) and (i, 0). Find two
lines, which together with x-axis forms a container, such that the

container contains the most water.

Diagram - https://leetcode.com/problems/container-with-most-water/

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

nput: height = [1,8,6,2,5,4,8,3,7]
Output: 49
Explanation: The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49.
Example 2:

Input: height = [1,1]
Output: 1
Example 3:

Input: height = [4,3,2,1,4]
Output: 16
Example 4:

Input: height = [1,2,1]
Output: 2

Constraints:

n == height.length
2 <= n <= 105
0 <= height[i] <= 104

```
class Solution {
    public int maxArea(int[] height)
    {
        int i = 0;
        int j = height.length - 1;
                    // Starting with the max-width container, and then moving inwards from the side which has a smaller height out of the two current pointers.

        int max = 0;
        while(i < j)
        {
            int minEle = Math.min(height[i], height[j]);
            int temp = minEle* Math.abs(i - j);
            max = Math.max(max, temp);
            if(minEle == height[i])
            {
                i++;
            }
            if(minEle == height[j])
            {
                j--;
            }
        }
        return max;
    }
}
```

----------Integer to Roman
----------Given an integer, convert it to a roman numeral.

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

Symbol	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

For example, 2 is written as II in Roman numeral, just two one's added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

I can be placed before V (5) and X (10) to make 4 and 9.
X can be placed before L (50) and C (100) to make 40 and 90.
C can be placed before D (500) and M (1000) to make 400 and 900.
Given an integer, convert it to a roman numeral.

Example 1:

Input: num = 3
Output: "III"
Example 2:

Input: num = 4
Output: "IV"
Example 3:

Input: num = 9
Output: "IX"
Example 4:

Input: num = 58
Output: "LVIII"
Explanation: L = 50, V = 5, III = 3.
Example 5:

Input: num = 1994
Output: "MCMXCIV"
Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.

Constraints:

1 <= num <= 3999

```
class Solution {
```

```
public String intToRoman(int num) {
        StringBuilder roman = new StringBuilder();
        while (num != 0) {
            if (num >= 1000) {
                roman.append("M");
                num = num - 1000;

            }
            else if (num >= 900 && num<1000) {
                roman.append("CM");
                num = num -900;
            }
            else if (num >= 500) {
                roman.append("D");
                num = num -500;

            }
            else if (num >= 400 && num<500) {
                roman.append("CD");
                num = num -400;
            }
            else if (num >=100) {
                roman.append("C");
                num = num -100;
            }
            else if (num >= 90 && num<100) {
                roman.append("XC");
                num = num -90;
            }
            else if (num >=50) {
                roman.append("L");
                num = num -50;
            }
            else if (num >= 40 && num<50) {
                roman.append("XL");
                num = num -40;
            }
            else if (num >=10) {
                roman.append("X");
                num = num - 10;
            }
            else if (num == 9){
                roman.append("IX");
                num = num -9;
            }
            else if (num>=5){
                roman.append("V");
                num = num-5;
            }
            else if (num == 4){
                roman.append("IV");
                num = num -4;
            }
            else {
```

```
            roman.append("I");
            num = num-1;
          }


       }return roman.toString();

   }

}
```

----------Roman to Integer
----------Given a roman numeral, convert it to an integer.
Input is guaranteed to be within the range from 1 to 3999.

Symbol	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

For example, 2 is written as II in Roman numeral, just two one's added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

I can be placed before V (5) and X (10) to make 4 and 9.
X can be placed before L (50) and C (100) to make 40 and 90.
C can be placed before D (500) and M (1000) to make 400 and 900.
Given a roman numeral, convert it to an integer.

Example 1:

Input: s = "III"
Output: 3
Example 2:

Input: s = "IV"
Output: 4
Example 3:

Input: s = "IX"
Output: 9
Example 4:

Input: s = "LVIII"
Output: 58

Explanation: L = 50, V= 5, III = 3.
Example 5:

Input: s = "MCMXCIV"
Output: 1994
Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.

Constraints:

1 <= s.length <= 15
s contains only the characters ('I', 'V', 'X', 'L', 'C', 'D', 'M').
It is guaranteed that s is a valid roman numeral in the range [1, 3999].

```
class Solution {

    private int romanDigitToInt(char digit) {
        switch (digit) {
            case 'I': return 1;
            case 'V': return 5;
            case 'X': return 10;
            case 'L': return 50;
            case 'C': return 100;
            case 'D': return 500;
            case 'M': return 1000;
            default: return 0;
        }
    }

    public int romanToInt(String s) {
        int previous = 0;
        int result = 0;
        for (char c : s.toCharArray()) {
            int current = romanDigitToInt(c);
            if (previous != 0 && previous < current) {
                result += current - (previous * 2);
                previous = 0;
                continue;
            }
            result += current;
            previous = current;
        }
        return result;
    }
}
```

----------Longest Common Prefix
----------Write a function to find the longest common prefix string amongst an array of strings.
If there is no common prefix, return an empty string "".

Example 1:

Input: strs = ["flower","flow","flight"]
Output: "fl"
Example 2:

Input: strs = ["dog","racecar","car"]
Output: ""
Explanation: There is no common prefix among the input strings.

Constraints:

1 <= strs.length <= 200
0 <= strs[i].length <= 200
strs[i] consists of only lower-case English letters.

```
class Solution {
    public String longestCommonPrefix(String[] strs) {
        //find the shortest String
        String pfx = null;
        for (String s : strs) {
            if (pfx == null || s.length() < pfx.length())
                pfx = s;
        }

        //strip a char from each pfx until you find it at the start of each string
        while(true) {
            boolean isPfx = true;
            for (String s : strs) {
                if (!s.startsWith(pfx)) {
                    pfx = pfx.substring(0, pfx.length()-1);
                    isPfx = false;
                    break;
                }
            }
            if (isPfx)
                break;
        }

        return pfx;
    }
}
```

----------3Sum
----------Given an array S of n integers, are there elements a, b, c in S such that a + b + c = 0? Find all unique triplets in the array which
gives the sum of zero.
Note: The solution set must not contain duplicate triplets.
For example, given array S = [-1, 0, 1, 2, -1, -4],
A solution set is:
[
[-1, 0, 1],
[-1, -1, 2]
]

Constraints:

0 <= nums.length <= 3000
-10^5 <= nums[i] <= 10^5

```
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        int maxVal = Integer.MIN_VALUE;
        int minVal = Integer.MAX_VALUE;
        int negNums = 0;
        int posNums = 0;
        List<List<Integer>> result = new LinkedList<>();
        int zeroNums = 0;

        // For all passed numbers in the nums[] array, find the min value,
        // max value, number of zeroes, number of positives, and number of
        // negatives.
        for (int num : nums) {
            if (num < minVal)  minVal = num;
            if (num > maxVal)  maxVal = num;
            if (num == 0)
                zeroNums++;
            else if (num > 0)
                posNums++;
            else
                negNums++;
        }

        // If there are at least three zeroes in the passed numbers, then
        // add the 3sum combination: 0+0+0
        if (zeroNums >= 3)  result.add(Arrays.asList(0, 0, 0));

        // If max < 0 or min > 0 then there cannot be any other valid 3sums.
        // Must have both positive and negative values so some values can
        // add to get zero.
        if (minVal >= 0 || maxVal <= 0)  return result;

        int[] negNumMap = new int[negNums];     // Array of all possible negative 3sum values.
        int[] posNumMap = new int[posNums];     // Array of all possible positive 3sum values.
        int posStart = 0;

        // If max or min values are too far from zero to use to make a 3sum,
        // then adjust max and/or min closer to zero.  This could eliminate
        // some of the outlying numbers that cannot be used to make a 3sum.
        // Those outlying numbers will be removed later when copying numbers
        // to the arrays of positives and negatives.
        if (maxVal + 2 * minVal > 0)  maxVal = -2 * minVal;
        if (minVal + 2 * maxVal < 0)  minVal = -2 * maxVal;

        // Scan through all of the numbers to build arrays of negative
        // numbers, positive numbers, and and array of counts of all
        // the numbers.
```

```
        byte[] numMap = new byte[maxVal - minVal + 1]; // Contains a count for each possible
number
                                        // between the min and max values.  To see if
                                        // a number n exists in the passed array, just
                                        // check if numMap[n-minVal] is non-zero.  The
                                        // value in numMap[n-minVal] is the number of
                                        // occurrences of n in the original array.
        negNums = 0;
        posNums = 0;
        for (int num : nums) {                  // Loop through all numbers in passed array.
            if (num >= minVal && num <= maxVal) {   // Skip numbers that cannot possibly make a
3sum because
                                        // they are too large (too positive) or too small
                                        // (too negative).
                if (numMap[num - minVal]++ != 0) {  // Count an occurrence if this number.  If already
                    numMap[num - minVal] = 2;       // seen this number, then set the count to 2,
because
                                        // anything greater than 2 isn't any different than a
                                        // count of 2, and this lets the counts fit into a byte.
                                        // Because this number already seen, skip adding this
                                        // number to the positive or negative maps, thereby NOT
                                        // allowing duplicate numbers in the positive or
                                        // negative maps.
                } else {                        // Else we haven't seen this number yet, so unless zero,
                                        // add the number to the positive or negative map.
                    if (num > 0) {
                        posNumMap[posNums++] = num; // Add unique positive numbers to positive
map.
                    } else if (num < 0) {
                        negNumMap[negNums++] = num; // Add unique negative numbers to negative
map.
                    }
                }
            }
        }

        // Sort the arrays of positive and negative numbers.  If arrays
        // are large, then .parallelSort() could be faster than .sort()
        Arrays.parallelSort(posNumMap, 0, posNums);
        Arrays.parallelSort(negNumMap, 0, negNums);

        // Loop through the negative numbers from highest negative number
        // (closest to zero) to lowest (farthest from zero; most negative).
        // By getting the negative numbers in this order (increasingly
        // negative), the 3sum will need to include a positive number with
        // the positive numbers being increasingly positive.
        for (int i = negNums - 1; i >= 0; i--) {
            int nv = negNumMap[i];              // Get next neg number to try as first num of a 3sum.
            int minpv = (-nv) / 2;              // Minimum positive value needed for the 3sum.  The
                                        // second 3sum value will be selected from the positive
                                        // numbers from half of the absolute value of the
                                        // negative number (first 3sum number), to a higher
                                        // positive number that would make the calculated third
                                        // 3sum number more negative than the first 3sum number.
```

```
                                    // This reduces the range of positive numbers to be tried
                                    // as the second 3sum number.

            while (posStart < posNums && posNumMap[posStart] < minpv)  posStart++;
                                    // Skip over any positive values that are below the
                                    // minimum positive value needed for the 3sum (minpv).
                                    // Since the negitive value nv will be increasingly
                                    // negative, the minimum positive value will be
                                    // increasingly positive, so start skipping positive
                                    // values that are below the minimum, starting at the
                                    // index (posStart) of the previous negative value
                                    // rather than scanning through all positive numbers
                                    // again.

            for (int j = posStart; j < posNums; j++) {  // Scan through possible pos values for this
3sum.
                int pv = posNumMap[j];              // Next possible highest positive value for this 3sum.
                                    // This could be the second value of the 3sum.
                int ln = 0 - nv - pv;               // Calculate the required third possible value for
                                    // this possible 3sum.

                if (ln >= nv && ln <= pv) {         // If the calculated third 3sum value is not between
                                    // the first and second 3sum values, then ignore this
                                    // 3sum combination.  This will eliminate duplicate 3sum
                                    // combinations, by having the first 3sum value being
                                    // the most negative number of the 3sum, and the second
                                    // 3sum value being the most positive number of the 3sum.
                    if (numMap[ln - minVal] == 0) { // If the calculated third value for the 3sum does not
                        continue;                   // exist in the passed array, this skip invalid 3sum.
                    } else if (ln == pv || ln == nv) {  // If the calculated third 3sum value is the same as
                                    // first or second 3sum value, then this is only allowed
                                    // if the calculated number occurs more than once in the
                                    // original passed array.
                        if (numMap[ln - minVal] > 1)  // If occurrence count of third number is more than
once.
                            result.add(Arrays.asList(nv, pv, ln));  // Valid 3sum with two numbers being the
same.
                    } else {
                        result.add(Arrays.asList(nv, pv, ln));  // Valid 3sum with all numbers being
different.
                    }
                } else if (ln < nv) {               // If third possible 3sum value is below negative first
                    break;                          // possible value for this 3sum, then we don't need to
                                    // test any higher positive values, because they would
                                    // only make the third value even more negative.  We don't
                                    // want the third value to be more negative than the first
                                    // 3sum value because this combination of 3sum numbers
could
                                    // be tested later as i gets closer to zero, and skipping
                                    // this more negative third 3sum value now, will prevent
                                    // duplicate 3sum combinations.
                }
            }
        }
```

```
        return result;
    }
}

class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        Arrays.sort(nums);
        for(int i=0; i<nums.length-2; i++) {
            // check-i, if i is repeated, then skip duplicate
            if(i>0 && nums[i] == nums[i-1]) continue;
            int j = i+1, k=nums.length-1;
            while(j<k) {
                int sum = nums[i]+nums[j]+nums[k];
                if(sum == 0) {
                    result.add(Arrays.asList(nums[i], nums[j], nums[k]));
                    //below two lines are for [-2,0,0,2,2] like scenario
                    while(j<k && nums[j] == nums[j+1]) j++; //check-j skip duplicates
                    while(j<k && nums[k] == nums[k-1]) k--; //check-k skip duplicates
                    j++; k--;
                } else if(sum<0) j++;
                else k--;
            }
        }
        return result;
    }
}
```

----------3Sum Closest

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

For example, given array S = {-1 2 1 -4}, and target = 1.

The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

Constraints:

3 <= nums.length <= 1000

-1000 <= nums[i] <= 1000

-10pow4 <= target <= 10pow4

```
class Solution {
    public int threeSumClosest(int[] a, int target) {
        Arrays.sort(a);
        int min_diff = Integer.MAX_VALUE;
        int result = 0;

        int start = 0; int end = a.length - 1;
        while(start < end) {
            int sum = a[start] + a[end];

            int index = binarysearch(a, 0, a.length - 1, target - sum);
            index = findItem(a, start, end, index);
```

```
            sum += a[index];

            if(sum == target) return target;
            if(min_diff > Math.abs(target - sum)) {
            min_diff = Math.abs(target - sum);
            result = sum;
            }

            if(target > sum) start++;
            else end--;
        }

        return result;
    }

    private int findItem(int[] a, int start, int end, int index) {
        if(index != start && index != end) return index;

        int min = Integer.MAX_VALUE;
        int pre = index;
        int next = index;

        while(pre == start || pre == end) pre--;
        if(pre >= 0 && min > Math.abs(a[index] - a[pre])) min = Math.abs(a[index] - a[pre]);

        while(next == start || next == end) next++;

        return next < a.length && min >= Math.abs(a[index] - a[next]) ? next : pre;
    }


    int binarysearch(int[] a, int start, int end, int key) {
        int s = start; int e = end;
        while(start <= end) {
            int mid = start + ((end - start) >> 1);
            if(key == a[mid]) return mid;
            if(key < a[mid]) end = mid - 1;
            else start = mid + 1;
        }

        if(end < s) return s;
        if(start > e) return e;
        return Math.abs(key - a[end]) <= Math.abs(key - a[start]) ? end : start;
    }
}
```

----------Letter Combinations of a Phone Number
----------Given a digit string, return all possible letter combinations that the number could represent.

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

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

Note:
Although the above answer is in lexicographical order, your answer
could be in any order you want.
----------4Sum
----------Given an array S of n integers, are there elements a, b, c, and d in
S such that a + b + c + d = target? Find all unique quadruplets in
the array which gives the sum of target.
Note: The solution set must not contain duplicate quadruplets.

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

set is:
0, 1],
1, 2],
0, 2]

```
class Solution {

 Map<Character,String> m ;
    List<String > ans ;
    public List<String> letterCombinations(String digits) {
        ans = new ArrayList<>();
        if(digits.length()==0)
            return ans;
        m = new HashMap<>();
        m.put('2',"abc");
        m.put('3',"def");
        m.put('4',"ghi");
        m.put('5',"jkl");
        m.put('6',"mno");
        m.put('7',"pqrs");
        m.put('8',"tuv");
        m.put('9',"wxyz");


        backtrack(digits,0,new StringBuilder());
        return ans;
    }
    public void backtrack(String d,int cur , StringBuilder path)
    {
        if(cur ==d.length()) {
            ans.add(path.toString());
            return ;
        }
        for( int i = 0 ; i < m.get(d.charAt(cur)).length();i++)
        {
            path.append(m.get(d.charAt(cur)).charAt(i));
```

```
            backtrack(d,cur+1,path);
            path.deleteCharAt(path.length()-1);
        }
    }
}
```

----------Remove Nth Node From End of List ----------Given a linked list, remove the nth 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.

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode fast = head;
        while (n-- > 0) {
            fast = fast.next;
        }

        ListNode sentinel = new ListNode(-1, head);
        ListNode slow = sentinel;
        while (fast != null) {
            fast = fast.next;
            slow = slow.next;
        }

        slow.next = slow.next.next;
        return sentinel.next;
    }
}
```

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

```
class Solution {

        /* use faster stack impl - 100% fast */
    public boolean isValid(String s) {
        int idx = 0;
        char[] stack = new char[s.length()];

        for(int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if(c == '(' || c == '{' || c == '[') {
                stack[idx++] = c;
```

```
            } else {
                if(idx == 0) {
                    return false;
                }
                char c1 = stack[idx - 1];
                if((c1 == '(' && c == ')')
                  || (c1 == '{' && c == '}')
                   || (c1 == '[' && c == ']')) {
                    idx--;
                } else {
                    return false;
                }
            }
        }

        if(idx != 0) {
            return false;
        }
        return true;
    }

    /* Most Common Solution
    public boolean isValid(String s) {
        if(s.length() % 2 != 0) return false;

        Stack<Character> stack = new Stack<>();

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

                char c1 = stack.pop();
                if( (c1 == '(' && c == ')')
                  || (c1 == '{' && c == '}')
                   || (c1 == '[' && c == ']')) {
                    continue;
                } else {
                    return false;
                }
            }
        }

        if(stack.isEmpty()) {
            return true;
        }
        return false;
    }
    */
}
```

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

Input: l1 = [1,2,4], l2 = [1,3,4]
Output: [1,1,2,3,4,4]
Example 2:

Input: l1 = [], l2 = []
Output: []
Example 3:

Input: l1 = [], l2 = [0]
Output: [0]

Constraints:

The number of nodes in both lists is in the range [0, 50].
-100 <= Node.val <= 100
Both l1 and l2 are sorted in non-decreasing order.

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

class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode head = new ListNode(-1);
        ListNode pointer = head;
        while(l1 != null || l2 != null)
        {
            if(l1 != null && l2 != null)
            {
                if(l1.val <= l2.val)
                {
                    pointer.next = l1;
                    l1 = l1.next;
                } else{
                    pointer.next = l2;
                    l2 = l2.next;
                }

            } else if(l1 != null)
            {
                pointer.next = l1;
```

```
                l1 = l1.next;
            } else if(l2 != null)
            {
                pointer.next = l2;
                l2 = l2.next;
            }
            pointer = pointer.next;
        }

        return head.next;
    }
}
```

----------Generate Parentheses

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

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

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

```
class Solution {

    // permutation
    // not - nclosing brakcets comes first
    // backtracking is about making decision

public List<String> generateParenthesis(int k) {
    // Write your solution here
    List<String> result = new ArrayList<String>();
    char[] cur = new char[k * 2];
    helper(cur, k, k, 0, result);
    return result;
  }

  private void helper(char[] cur, int left, int right, int index, List<String> result) {
    if (left == 0 && right == 0) {
      result.add(new String(cur));
      return;
    }
    if (left > 0) {
      cur[index] = '(';
      helper(cur, left - 1, right, index + 1, result);
    }
    if (right > left) {
      cur[index] = ')';
      helper(cur, left, right - 1, index + 1, result);
    }
  }
}
```

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

Example 1:

```
Input: lists = [[1,4,5],[1,3,4],[2,6]]
Output: [1,1,2,3,4,4,5,6]
Explanation: The linked-lists are:
[
  1->4->5,
  1->3->4,
  2->6
]
merging them into one sorted list:
1->1->2->3->4->4->5->6
```

Example 2:

```
Input: lists = []
Output: []
```

Example 3:

```
Input: lists = [[]]
Output: []
```

Constraints:

```
k == lists.length
0 <= k <= 10^4
0 <= lists[i].length <= 500
-10^4 <= lists[i][j] <= 10^4
lists[i] is sorted in ascending order.
The sum of lists[i].length won't exceed 10^4.
```

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

public ListNode mergeKLists(ListNode[] lists) {
    return partion(lists,0,lists.length-1);
}
public static ListNode partion(ListNode[] lists,int s,int e){
    if(s==e)  return lists[s];
    if(s<e){
        int q=(s+e)/2;
```

```
        ListNode l1=partion(lists,s,q);
        ListNode l2=partion(lists,q+1,e);
        return merge(l1,l2);
    }else
        return null;
}

//This function is from Merge Two Sorted Lists.
public static ListNode merge(ListNode l1,ListNode l2){
    if(l1==null||l2==null)
            return l1==null?l2:l1;
     ListNode head=new ListNode(0),temp=head;
        while(l1!=null&&l2!=null){
            if(l1.val<=l2.val){
                temp.next=l1;
                temp=temp.next;
                l1=l1.next;
            }
            else{
                temp.next=l2;
                temp=temp.next;
                l2=l2.next;
            }
        }
        while(l1!=null){
                temp.next=l1;
                temp=temp.next;
                l1=l1.next;
        }
        while(l2!=null){
                temp.next=l2;
                temp=temp.next;
                l2=l2.next;
        }
        return head.next;
}
}
```

----------Swap Nodes in Pairs
----------Given a linked list, swap every two adjacent nodes and return its head.

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

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

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

```
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {

    public ListNode swapPairs(ListNode head) {
        if(head == null || head.next == null)
            return head;
        ListNode save = head.next.next;
        ListNode ret = head.next;
        head.next.next = head;
        head.next = swapPairs(save);
        return ret;
    }
}
```

----------Reverse Nodes in k-Group

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

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

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

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

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
   public ListNode reverseKGroup(ListNode head, int k) {
    int count = 0;
    ListNode temp = head;
    while(temp!=null && count<k) {
        temp = temp.next;
        count++;
    }
    if(count<k) return head;

    int n = k;
    ListNode tail = head;
    ListNode front = reverse(head,n);
    tail.next = reverseKGroup(temp,n);
    return front;
}

private ListNode reverse(ListNode head,int k){
    ListNode prev = null;
    ListNode curr = head;
    ListNode next = head;
    while(k--!=0){
        next = curr.next;
        curr.next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}

}
```

----------Remove Duplicates from Sorted Array
----------Given a sorted array, remove the duplicates in place such that each element appear only once and return the new length.
Do not allocate extra space for another array, you must do this in place with constant memory.

For example,
Given input array nums = [1,1,2],
Your function should return length = 2, with the first two elements of nums being 1 and 2 respectively. It doesn't matter what you leave beyond the new length.

Given an integer array nums sorted in non-decreasing order, remove the duplicates in-place such that each unique element appears only once. The relative order of the elements should be kept the same.

Since it is impossible to change the length of the array in some languages, you must instead have the result be placed in the first part of the array nums. More formally, if there are k elements after removing the duplicates, then the first k elements of nums should hold the final result. It does not matter what you leave beyond the first k elements.

Return k after placing the final result in the first k slots of nums.

Do not allocate extra space for another array. You must do this by modifying the input array in-place with O(1) extra memory.

Custom Judge:

The judge will test your solution with the following code:

```
int[] nums = [...]; // Input array
int[] expectedNums = [...]; // The expected answer with correct length

int k = removeDuplicates(nums); // Calls your implementation

assert k == expectedNums.length;
for (int i = 0; i < k; i++) {
    assert nums[i] == expectedNums[i];
}
```

If all assertions pass, then your solution will be accepted.

Example 1:

```
Input: nums = [1,1,2]
Output: 2, nums = [1,2,_]
Explanation: Your function should return k = 2, with the first two elements of nums being 1 and 2 respectively.
It does not matter what you leave beyond the returned k (hence they are underscores).
```

Example 2:

```
Input: nums = [0,0,1,1,1,2,2,3,3,4]
Output: 5, nums = [0,1,2,3,4,_,_,_,_,_]
Explanation: Your function should return k = 5, with the first five elements of nums being 0, 1, 2, 3, and 4 respectively.
It does not matter what you leave beyond the returned k (hence they are underscores).
```

Constraints:

```
0 <= nums.length <= 3 * 104
-100 <= nums[i] <= 100
nums is sorted in non-decreasing order.
```

```
class Solution {
    public int removeDuplicates(int[] nums) {
            int pointer = 1;
        for(int k=1; k<nums.length; k++){
            if(nums[k]>nums[k-1]){
                nums[pointer++]=nums[k];
              }
        }
        return pointer;
    }
}
```

----------Remove Element----------Given an array and a value, remove all instances of that value in place and return the new length.
Do not allocate extra space for another array, you must do this in place with constant memory.
The order of elements can be changed. It doesn't matter what you leave beyond the new length.
Example:
Given input array nums = [3,2,2,3], val = 3
Your function should return length = 2, with the first two elements of nums being 2.
Try two pointers.
Did you use the property of "the order of elements can be changed"?
What happens when the elements to remove are rare?

```
class Solution {
    public int removeElement(int[] nums, int val) {
        int i=0,j=nums.length-1;
        int cnt=0;
        while(i<j){
            if(nums[i]==val&&nums[j]!=val){
                nums[i]=nums[j];
                nums[j]=val;
                j--;
                i++;
            }
            if(nums[j]==val){
                j--;
            }
            if(nums[i]!=val){
                i++;
            }
        }
        for(int no:nums){
            if(no!=val)
                cnt++;
            else
                break;
        }
        return cnt;
    }
}
```

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

Example 1:

Input: haystack = "hello", needle = "ll"
Output: 2
Example 2:

Input: haystack = "aaaaa", needle = "bba"
Output: -1
Example 3:

Input: haystack = "", needle = ""
Output: 0

```
class Solution {
public int strStr(String haystack, String needle) {
if (needle.isEmpty())
        return 0;

    int hlen = haystack.length();
    int nlen = needle.length();

    for (int i = 0; i < hlen - nlen + 1; i++) {
        if (haystack.substring(i, nlen + i).equals(needle))
            return i;
    }

    return -1;
}
}
```

----------Divide Two Integers
----------Given two integers dividend and divisor, divide two integers without using multiplication, division, and mod operator.
The integer division should truncate toward zero, which means losing its fractional part. For example, 8.345 would be truncated to 8, and -2.7335 would be truncated to -2.
Return the quotient after dividing dividend by divisor.

Note: Assume we are dealing with an environment that could only store integers within the 32-bit signed integer range: [–231, 231 − 1]. For this problem, if the quotient is strictly greater than 231 - 1, then return 231 - 1, and if the quotient is strictly less than -231, then return -231.

Example 1:

Input: dividend = 10, divisor = 3
Output: 3
Explanation: 10/3 = 3.33333.. which is truncated to 3.
Example 2:

Input: dividend = 7, divisor = -3
Output: -2

Explanation: 7/-3 = -2.33333.. which is truncated to -2.
Example 3:

Input: dividend = 0, divisor = 1
Output: 0
Example 4:

Input: dividend = 1, divisor = 1
Output: 1

Constraints:

-231 <= dividend, divisor <= 231 - 1
divisor != 0

```
class Solution {
    public int divide(int dividend, int divisor) {
        //if the outpur is greater then 2^31-1;
        if(dividend == -2147483648 && divisor == -1) return Integer.MAX_VALUE;

        //sign for the output
        boolean sign = (dividend>=0)==(divisor>=0)?true:false;

        // converting to positive integer
        dividend = Math.abs(dividend);
        divisor = Math.abs(divisor);
        int result = 0;
        int temp = 0;
        int prev = 0;
        //checking id dividend is smaller then divisor
        //we can divide 10 by 3 but not 1 by 3
        while(dividend-divisor>=0){
            int count = 0;
            temp = divisor;
            prev =0;
            // checking not for every multiple of divsor but 2^count multiple of divisor
            while(dividend-temp>=0){
                prev = temp;
                temp = temp<<1;
                count++;
            }
            result+=1<<count-1;
            dividend -=prev;
        }
        return sign?result:-result;
    }
}
```

----------Substring with Concatenation of All Words
----------You are given a string, s, and a list of words, words, that are all
of the same length. Find all starting indices of substring(s) in s
that is a concatenation of each word in words exactly once and

without any intervening characters.

For example, given:
s: "barfoothefoobarman"
words: ["foo", "bar"]

You should return the indices: [0,9].
(order does not matter).

Example 1:

Input: s = "barfoothefoobarman", words = ["foo","bar"]
Output: [0,9]
Explanation: Substrings starting at index 0 and 9 are "barfoo" and "foobar" respectively.
The output order does not matter, returning [9,0] is fine too.
Example 2:

Input: s = "wordgoodgoodgoodbestword", words = ["word","good","best","word"]
Output: []
Example 3:

Input: s = "barfoofoobarthefoobarman", words = ["bar","foo","the"]
Output: [6,9,12]

Constraints:

1 <= s.length <= 104
s consists of lower-case English letters.
1 <= words.length <= 5000
1 <= words[i].length <= 30
words[i] consists of lower-case English letters.

```
class Solution {
    public List<Integer> findSubstring(String s, String[] words) {
                    /**
                     * Let n=s.length, k=words[0].length traverse s with indices i, i+k,
                     * i+2k, ... for 0<=i<k, so that the time complexity is O(n).
                     */
                    List<Integer> res = new ArrayList<Integer>();
                    int n = s.length(), m = words.length, k;
                    if (n == 0 || m == 0 || (k = words[0].length()) == 0)
                            return res;

                    HashMap<String, Integer> wDict = new HashMap<String, Integer>();

                    for (String word : words) {
                            if (wDict.containsKey(word))
                                    wDict.put(word, wDict.get(word) + 1);
                            else
                                    wDict.put(word, 1);
                    }

                    int i, j, start, x, wordsLen = m * k;
```

```
		HashMap<String, Integer> curDict = new HashMap<String, Integer>();
		String test, temp;
		for (i = 0; i < k; i++) {
			curDict.clear();
			start = i;
			if (start + wordsLen > n)
				return res;
			for (j = i; j + k <= n; j += k) {
				test = s.substring(j, j + k);

				if (wDict.containsKey(test)) {
					if (!curDict.containsKey(test)) {
						curDict.put(test, 1);

						start = checkFound(res, start, wordsLen, j, k,
curDict, s);
						continue;
					}

					// curDict.containsKey(test)
					x = curDict.get(test);
					if (x < wDict.get(test)) {
						curDict.put(test, x + 1);

						start = checkFound(res, start, wordsLen, j, k,
curDict, s);
						continue;
					}

					// curDict.get(test)==wDict.get(test), slide start to
					// the next word of the first same word as test
					while (!(temp = s.substring(start, start + k)).equals(test)) {
						decreaseCount(curDict, temp);
						start += k;
					}
					start += k;
					if (start + wordsLen > n)
						break;
					continue;
				}

				// totally failed up to index j+k, slide start and reset all
				start = j + k;
				if (start + wordsLen > n)
					break;
				curDict.clear();
			}
		}
		return res;
	}

	public int checkFound(List<Integer> res, int start, int wordsLen, int j, int k,
			HashMap<String, Integer> curDict, String s) {
		if (start + wordsLen == j + k) {
```

```
                        res.add(start);
                        // slide start to the next word
                        decreaseCount(curDict, s.substring(start, start + k));
                        return start + k;
                }
                return start;
        }

        public void decreaseCount(HashMap<String, Integer> curDict, String key) {
                // remove key if curDict.get(key)==1, otherwise decrease it by 1
                int x = curDict.get(key);
                if (x == 1)
                        curDict.remove(key);
                else
                        curDict.put(key, x - 1);
        }
}
```

----------Next Permutation

----------Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers.
If such arrangement is not possible, it must rearrange it as the
lowest possible order (ie, sorted in ascending order).

The replacement must be in-place, do not allocate extra memory.
Here are some
corresponding
1,2,3 →
3,2,1 →
1,1,5 →

examples. Inputs are in the left-hand column and its
outputs are in the right-hand column.
1,3,2
1,2,3
1,5,1

```
class Solution {
    public void nextPermutation(int[] nums) {
        int end = nums.length-2;
        while (end >=0 && nums[end+1] <= nums[end])
            end--;

        if (end >=0){
            int start = nums.length - 1;
            while (start >=0 && nums[start] <= nums[end])
                start--;

                                swap(nums, start, end);
        }
        reverse(nums, end+1);
    }

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

    protected void reverse(int[] nums, int start){
        int end = nums.length-1;
        while (start < end){
            swap(nums, start, end);
            start++;
            end--;
        }
    }
}
```

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

```
class Solution {
```

```
    public int longestValidParentheses(String s) {
        int[] dp = new int[s.length()];

        int max = 0;
        for (int i = 1; i < s.length(); i++) {  // can skip first since cannot be valid
            char c = s.charAt(i);
            if (c == ')') {                     // only check if closed bracket
                if (s.charAt(i - 1) == '(')     // ...() so just add already longest valid + 2
                    dp[i] = i - 2 >= 0 ? dp[i - 2] + 2 : 2;
                else if (dp[i - 1] > 0 && i - 1 - dp[i - 1] >= 0 && s.charAt(i - 1 - dp[i - 1]) == '(') { // if
both closed bracket, check if previous is valid, then check if char before valid length within
range, and if '('
                    dp[i] = dp[i - 1] + 2;      // then add just previous dp + 2
                    dp[i] += i - dp[i] >= 0 ? dp[i - dp[i]] : 0;    // also add previous valid length since can
link together e.g. ()(())
                }
                max = Math.max(max, dp[i]);     // keep track of max
            }
        }

        return max;
    }
}
```

----------Search in Rotated Sorted Array

----------Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.
(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).
You are given a target value to search. If found in the array return
its index, otherwise return -1.
You may assume no duplicate exists in the array.

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

    public int partSearch(int[] nums, int target, int start, int end){

        if(start==end && nums[start]!=target) return -1;
        if(start==end && nums[start]==target) return start;
        if(nums[start]<nums[end] && (nums[start]>target || nums[end] < target)) return -1;

        int middle=start+(end-start)/2;
        int rightResult=-1;
        int leftResult=-1;

        if(nums[start]<nums[middle] && nums[middle]<target){
            rightResult=partSearch(nums, target, middle+1, end);
        }else if(nums[middle]<nums[end] && nums[middle]>target){
            leftResult=partSearch(nums, target, start, middle);
        }else{
                                //rotated part
            rightResult=partSearch(nums, target, middle+1, end);
```

```
            if(rightResult!=-1) return rightResult;
            leftResult=partSearch(nums, target, start, middle);
        }

        if(rightResult!=-1) return rightResult;
        if(leftResult!=-1) return leftResult;
        return -1;
    }
}
```

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

```
class Solution {
    public int[] searchRange(int[] nums, int target) {
        int start=0,end=nums.length-1,fi=-1;
        while(start<=end){
            int mid=start+(end-start)/2;
            if(nums[mid]==target){
                fi=mid;
                end=mid-1;
            }
            else if(nums[mid]>target){
                end=mid-1;
            }
            else{
                start=mid+1;
            }
        }
        start=0;end=nums.length-1;
        int ei=-1;
        while(start<=end){
            int mid=start+(end-start)/2;
           if(nums[mid]==target){
                ei=mid;
                start=mid+1;
            }
            else if(nums[mid]>target){
                end=mid-1;
            }
            else{
                start=mid+1;
            }
        }
        return new int[]{fi,ei};
    }
}
```

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

Example 1:

Input: nums = [1,3,5,6], target = 5
Output: 2
Example 2:

Input: nums = [1,3,5,6], target = 2
Output: 1
Example 3:

Input: nums = [1,3,5,6], target = 7
Output: 4
Example 4:

Input: nums = [1,3,5,6], target = 0
Output: 0
Example 5:

Input: nums = [1], target = 0
Output: 0

Constraints:

1 <= nums.length <= 104
-104 <= nums[i] <= 104
nums contains distinct values sorted in ascending order.
-104 <= target <= 104

```
class Solution {
    public int searchInsert(int[] nums, int target) {
        for(int i=0;i<nums.length;i++){
            if(nums[i]==target) return i;  // this means the value is present and we will returen the
index
            else if(nums[i]>target) return i;  // this mean the array value is more that target value and
                                    //we havn'e found any match yet this mean the value is not present so
we will put target
                                    //value in the index of its just greater value that is present at index i
        }
        return nums.length;// we have traversed the whole array and niether able to find the value,
                    //nor the value greater then it i.e. target will be added at the end end so return
nums.length
    }
```

```
}
class Solution {
  public int searchInsert(int[] A, int target) {
      int low = 0, high = A.length-1;
      while(low<=high){
          int mid = (low+high)/2;
          if(A[mid] == target) return mid;
          else if(A[mid] > target) high = mid-1;
          else low = mid+1;
      }
      return low;
  }
}
```

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

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.

Input: board =
[["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"]]
Output: true
Example 2:

Input: board =
[["8","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"]]
Output: false
Explanation: Same as Example 1, except with the 5 in the top left corner being modified to 8. Since there are two 8's in the top left 3x3 sub-box, it is invalid.

Constraints:

board.length == 9
board[i].length == 9
board[i][j] is a digit 1-9 or '.'.

```
class Solution {
  public boolean isValidSudoku(char[][] board) {
        boolean[][] rows = new boolean[9][9];
        boolean[][] columns = new boolean[9][9];
        boolean[][] boxes = new boolean[9][9];

        for(int i = 0; i < 9; i++){
            for(int j = 0; j < 9; j++){
                if(board[i][j] != '.'){
                    int n = Character.getNumericValue(board[i][j]);
                    int boxindex = (i / 3) * 3 + j / 3;
                    if(rows[i][n-1] ||
                       columns[j][n-1] ||
                       boxes[boxindex][n-1])
                        return false;
                    else{
                        rows[i][n-1] = true;
                        columns[j][n-1] = true;
                        boxes[boxindex][n-1] = true;
                    }
                }
            }
        }
        return true;
    }
}
```

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

A sudoku solution must satisfy all of the following rules:

Each of the digits 1-9 must occur exactly once in each row.
Each of the digits 1-9 must occur exactly once in each column.
Each of the digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxes of the grid.
The '.' character indicates empty cells.
Input: board = [["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"]]
Output: [["5","3","4","6","7","8","9","1","2"],["6","7","2","1","9","5","3","4","8"],
["1","9","8","3","4","2","5","6","7"],["8","5","9","7","6","1","4","2","3"],
["4","2","6","8","5","3","7","9","1"],["7","1","3","9","2","4","8","5","6"],
["9","6","1","5","3","7","2","8","4"],["2","8","7","4","1","9","6","3","5"],
["3","4","5","2","8","6","1","7","9"]]
Explanation: The input board is shown above and the only valid solution is shown below:

Constraints:

board.length == 9
board[i].length == 9
board[i][j] is a digit or '.'.
It is guaranteed that the input board has only one solution.

```
class Solution {

    /**
     * State of a bitset where all digits [1..9] are present
     */
    private static final int ALL_SET = 0b111_111_111_0;

    /**
     * Box indices by row and column
     */
    private static final int[][] BOX_INDICES = {
            {0, 0, 0, 1, 1, 1, 2, 2, 2},
            {0, 0, 0, 1, 1, 1, 2, 2, 2},
            {0, 0, 0, 1, 1, 1, 2, 2, 2},
            {3, 3, 3, 4, 4, 4, 5, 5, 5},
            {3, 3, 3, 4, 4, 4, 5, 5, 5},
            {3, 3, 3, 4, 4, 4, 5, 5, 5},
            {6, 6, 6, 7, 7, 7, 8, 8, 8},
            {6, 6, 6, 7, 7, 7, 8, 8, 8},
            {6, 6, 6, 7, 7, 7, 8, 8, 8}
    };

    /**
     * Bitmap of present digits by row
     */
    private int[] rows;

    /**
     * Bitmap of present digits by column
     */
    private int[] cols;

    /**
     * Bitmap of present digits by box
     */
    private int[] boxes;

    /**
     * Current Sudoku board
     */
    private char[][] board;

    public void solveSudoku(char[][] board) {
        this.board = board;
        rows = new int[9];
        cols = new int[9];
        boxes = new int[9];
        for (int row = 0; row < 9; row++) {
```

```
        for (int col = 0; col < 9; col++) {
            char c = board[row][col];
            if (c == '.') continue;
            int box = box(row, col);
            flip(row, col, box, c - '0');
        }
    }
    if (!solve(0, 0)) {
        throw new IllegalArgumentException("Unsolvable Sudoku!");
    }
}

private boolean solve(int row, int col) {
    for (int ri = row; ri < 9; ri++) {
        for (int ci = col; ci < 9; ci++) {
            if (board[ri][ci] != '.') continue;

            int bi = box(ri, ci);

            int set = rows[ri] | cols[ci] | boxes[bi];
            // if no unused digits are left, then solution is invalid
            if (set == ALL_SET) return false;

            for (int lo = 1, digit; lo <= 9; lo = digit + 1) {
                digit = nextDigit(set, lo);
                if (digit > 9) return false;

                board[ri][ci] = (char) ('0' + digit);
                flip(ri, ci, bi, digit);

                if (solve(ri, ci + 1)) return true;
                // if solution is not found, backtrack

                board[ri][ci] = '.';
                flip(ri, ci, bi, digit);
            }
            return false;
        }
        // next row will start from 0
        col = 0;
    }
    return true;
}

/**
 * Flips the bit that indicates the presence of a digit in the given row, column and box
 *
 * @param row row index
 * @param col column index
 * @param box box index
 * @param val digit to set/unset
 */
private void flip(int row, int col, int box, int val) {
    int bit = 1 << val;
```

```
            rows[row] ^= bit;
            cols[col] ^= bit;
            boxes[box] ^= bit;
        }

        /**
         * Calculates next digit absent from a set
         *
         * @param set    bitset of placed digits
         * @param lowest lowest digit to consider
         * @return index of the next unplaced digit
         */
        private static int nextDigit(int set, int lowest) {
            set >>>= lowest;
            return Integer.numberOfTrailingZeros(~set) + lowest;
        }

        /**
         * Calculates box index for given cell on a board
         *
         * @param row column index
         * @param col row index
         * @return box index
         */
        private static int box(int row, int col) {
            return BOX_INDICES[row][col];
        }

    }
    ----------------------------------------------------------------Count and Say
    -----------------------------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 nth sequence.

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

    Example 1:

    Input: n = 1
    Output: "1"
    Explanation: This is the base case.
    Example 2:

    Input: n = 4
    Output: "1211"
    Explanation:
    countAndSay(1) = "1"
```

countAndSay(2) = say "1" = one 1 = "11"
countAndSay(3) = say "11" = two 1's = "21"
countAndSay(4) = say "21" = one 2 + one 1 = "12" + "11" = "1211"

Constraints:

1 <= n <= 30

```
class Solution {
    public String countAndSay(int n) {
        if (n == 1) {
            return "1";
        }

        return analyze(countAndSay(n - 1));
    }

    private static String analyze(String res) {
        StringBuilder b = new StringBuilder();

        int count = 1;
        char curr = res.charAt(0);

        for (int i = 1; i < res.length(); i++) {
            char atI = res.charAt(i);
            if (atI != curr) {
                b.append(count).append(curr);
                curr = atI;
                count = 1;
            } else {
                count++;
            }
        }

        b.append(count).append(curr);
        return b.toString();
    }
}
```

----------Combination Sum----------

Given a set of candidate numbers (C) (without duplicates) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.

The same repeated number may be chosen from C unlimited number of times.

Note:

All numbers (including target) will be positive integers.

The solution set must not contain duplicate combinations.

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

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

]

```
class Solution {

    void findAllways(int[] candidates,List<List<Integer>> finalList,ArrayList<Integer> list,int i,int
target){
       if(target == 0)
          finalList.add(new ArrayList<Integer>(list));
       else{
          for(int j=i;j<candidates.length;j++)
          {
             if(candidates[j]<=target){
             list.add(candidates[j]);
             findAllways(candidates,finalList,list,j,target-candidates[j]);
             list.remove(list.size()-1);
             }
          }
       }
   }

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

       List<List<Integer>> finalList = new ArrayList<>();
       ArrayList<Integer> list = new ArrayList<>();
       findAllways(candidates,finalList,list,0,target);

       return finalList;
   }

}
```

----------Combination Sum II
----------Given a collection of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums
to T.
Each number in C may only be used once in the combination.
Note:
All numbers (including target) will be positive integers.
The solution set must not contain duplicate combinations.

For example, given candidate set [10, 1, 2, 7, 6, 1, 5] and target 8,

```
public class Solution {
  public List<List<Integer>> combinationSum2(int[] candidates, int target) {
     Arrays.sort(candidates);
       List<List<Integer>> results = new ArrayList<>();
       calcCombinationSum2(candidates, 0, new int[candidates.length], 0, target, results);
       return results;
   }

   private void calcCombinationSum2(int[] candidates, int cindex, int[] list, int lindex, int target,
List<List<Integer>> results) {
```

```
        if (target == 0) {
            List<Integer> result = new ArrayList<>();
            for (int i = 0; i < lindex; i++) {
                result.add(list[i]);
            }
            results.add(result);
            return;
        }

        int prev = 0;
        for (int i = cindex; i < candidates.length; i++) {
            if (candidates[i] != prev) {
                if (target - candidates[i] < 0) {
                    break;
                }

                list[lindex] = candidates[i];
                calcCombinationSum2(candidates, i + 1, list, lindex + 1, target - candidates[i],
results);
                prev = candidates[i];
            }
        }
    }
}
```

----------First Missing Positive
----------Given an unsorted integer array, find the first missing positive integer.

For example,
Given [1,2,0] return 3,
and [3,4,-1,1] return 2.

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

```
class Solution {
    public int firstMissingPositive(int[] nums) {
        int i = 0;
        while( i < nums.length){
            if(nums[i] > 0 && nums[i] <= nums.length && nums[i] != nums[nums[i] - 1]){
                swap(nums,i,nums[i] - 1);
            }else{
                i++;
            }
        }
        for(i = 0;i<nums.length;i++){
            if(nums[i] != i + 1){
                return i + 1;
            }
        }
        return nums.length + 1;
    }

    private void swap(int[] nums, int i, int j){
```

```
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
}
```

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

For example,
Given [0,1,0,2,1,0,1,3,2,1,2,1], return 6.

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

Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6
Explanation: The above elevation map (black section) is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.
Example 2:

Input: height = [4,2,0,3,2,5]
Output: 9

```
class Solution {
    public int trap(int[] height) {
        int max_height_index = 0;
        int max=0,n = height.length;
        if(n==0)return 0;
        for(int i=0;i<n;i++){
            if(height[i]>=max){
                max = height[i];
                max_height_index = i;
            }
        }

        int water=0;
        max=height[0];
        for(int i=1;i<=max_height_index;i++){
            if(height[i]<max){
                water+=max-height[i];
            }
            else{
                max = height[i];
            }
        }
        max = height[n-1];
        for(int i=n-2;i>=max_height_index;i--){
            if(height[i]<max){
                water+=max-height[i];
```

```
            }
            else{
                max = height[i];
            }
        }

        return water;
    }
}
```

----------Multiply Strings

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

Note:

The length of both num1 and num2 is < 110.

Both num1 and num2 contains only digits 0-9.

Both num1 and num2 does not contain any leading zero.

You must not use any built-in BigInteger library or convert the inputs to integer directly.

```
class Solution {
   public int [] getIntArray(String s){
    char [] chars = s.toCharArray();
    int [] arr = new int[chars.length];
    for(int i=0;i<chars.length;i++){
        arr[i] = chars[i] - '0';
    }

    return arr;
}

public String convertToStr(int [] res, int i){
    char [] chars = new char[res.length-i];
    int k = 0;

    for(;i<res.length;i++){
        chars[k] = (char) (res[i] + '0');
        k++;
    }

    return new String(chars);
}



public String multiply(String num1, String num2) {
    int [] arr1 = getIntArray(num1);
    int [] arr2 = getIntArray(num2);
    int [] res = new int[arr1.length + arr2.length];
    int index = arr1.length + arr2.length - 1;

    for(int i=arr2.length-1; i>=0;i--){
        int k = index--;
        for(int j = arr1.length-1; j>=0; j--){
            res[k] += arr2[i]*arr1[j];
```

```
            k--;
        }
    }
    // System.out.println(Arrays.toString(res));
    index = arr1.length + arr2.length - 1;
    int carry =0;

    for(int i=index;i>=0;i--){
        int temp = res[i]+carry;
        res[i] = temp%10;
        carry = temp/10;
    }

    // System.out.println(Arrays.toString(res));

    int i = 0;
    while(i<res.length && res[i]==0){
        i++;
    }

    if(i>index){
        return "0";
    }
    else{
        return convertToStr(res, i);
    }

}
}
```

----------Wildcard Matching

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

'?' Matches any single character.

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

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

The function prototype should be:

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

Some examples:

isMatch("aa","a") → false

isMatch("aa","aa") → true

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

```
class Solution {
    public boolean isMatch(String s, String p) {
        int i=0;
        int j=0;
        int starIdx=-1;
        int lastMatch=-1;

        while(i<s.length()){
            if(j<p.length() && (s.charAt(i)==p.charAt(j) ||
              p.charAt(j)=='?')){
                i++;
                j++;
            }else if(j<p.length() && p.charAt(j)=='*'){
                starIdx=j;
                lastMatch=i;
                j++;
            }else if(starIdx!=-1){
            //there is a no match and there was a previous star, we will reset the j to indx after
star_index
            //lastMatch will tell from which index we start comparing the string if we encounter * in
pattern
                j=starIdx+1;
                lastMatch++; // we are saying we included more characters in * so we incremented
the index
                i=lastMatch;

            }else{
                return false;
            }
        }

        while(j<p.length() && p.charAt(j)=='*') j++;

        if(i!=s.length() || j!=p.length()) return false;

        return true;

    }
}
```

----------Jump Game II
----------Given an array of non-negative integers, you are initially positioned at the first index of the array.
Each element in the array represents your maximum jump length at that position.
Your goal is to reach the last index in the minimum number of jumps.

Example 1:

Input: nums = [2,3,1,1,4]
Output: 2
Explanation: The minimum number of jumps to reach the last index is 2. Jump 1 step from index 0 to 1, then 3 steps to the last index.
Example 2:

Input: nums = [2,3,0,1,4]
Output: 2

Constraints:

1 <= nums.length <= 104
0 <= nums[i] <= 1000

```
class Solution {
   public int jump(int[] nums) {
    int lastValid = nums.length - 1;
    int jumps = 0;
    while (lastValid > 0) {
        for (int i = 0; i < lastValid; i++) {
            if (nums[i] + i >= lastValid) {
                lastValid = i;
                jumps++;
                break;
            }
        }
    }
    return jumps;
}
}
```

----------Jump Game III
----------Given an array of non-negative integers arr, you are initially positioned at start index of the array. When you are at index i, you can jump to i + arr[i] or i - arr[i], check if you can reach to any index with value 0.

Notice that you can not jump outside of the array at any time.

Example 1:

Input: arr = [4,2,3,0,3,1,2], start = 5
Output: true
Explanation:
All possible ways to reach at index 3 with value 0 are:
index 5 -> index 4 -> index 1 -> index 3
index 5 -> index 6 -> index 4 -> index 1 -> index 3
Example 2:

Input: arr = [4,2,3,0,3,1,2], start = 0
Output: true

Explanation:
One possible way to reach at index 3 with value 0 is:
index 0 -> index 4 -> index 1 -> index 3
Example 3:

Input: arr = [3,0,2,1,2], start = 2
Output: false
Explanation: There is no way to reach at index 1 with value 0.

Constraints:

1 <= arr.length <= 5 * 104
0 <= arr[i] < arr.length
0 <= start < arr.length

```
// dfs
class Solution {
    public boolean canReach(int[] arr, int start) {
        if(start < 0 || start >= arr.length || arr[start] < 0)
            return false;
        arr[start] *= -1;
        return arr[start] == 0 || canReach(arr, start + arr[start]) || canReach(arr, start - arr[start]);
    }
}

class Solution {
    public boolean canReach(int[] arr, int start) {
        return start >= 0 && start < arr.length && arr[start] >= 0 && ((arr[start] = -arr[start]) == 0 ||
canReach(arr, start + arr[start]) || canReach(arr, start - arr[start]));
    }
}


// bfs
class Solution {
    public boolean canReach(int[] arr, int start) {
        int N = arr.length;
        boolean[] visited = new boolean[N];
        Queue<Integer> Q = new LinkedList<>();
        Q.add(start);
        while (!Q.isEmpty()) {
            int top = Q.poll();
            if (arr[top] == 0)
                return true;
            visited[top] = true;
            int left = top - arr[top];
            int right = top + arr[top];
            if (left >= 0 && visited[left] == false)
                Q.add(left);
            if (right < N && visited[right] == false)
                Q.add(right);
        }
        return false;
```

```
    }
}
```

----------Jump Game IV
----------Given an array of integers arr, you are initially positioned at the first index of the array.

In one step you can jump from index i to index:

i + 1 where: i + 1 < arr.length.
i - 1 where: i - 1 >= 0.
j where: arr[i] == arr[j] and i != j.
Return the minimum number of steps to reach the last index of the array.

Notice that you can not jump outside of the array at any time.

Example 1:

Input: arr = [100,-23,-23,404,100,23,23,23,3,404]
Output: 3
Explanation: You need three jumps from index 0 --> 4 --> 3 --> 9. Note that index 9 is the last index of the array.
Example 2:

Input: arr = [7]
Output: 0
Explanation: Start index is the last index. You don't need to jump.
Example 3:

Input: arr = [7,6,9,6,9,6,9,7]
Output: 1
Explanation: You can jump directly from index 0 to index 7 which is last index of the array.
Example 4:

Input: arr = [6,1,9]
Output: 2
Example 5:

Input: arr = [11,22,7,7,7,7,7,7,7,22,13]
Output: 3

Constraints:

1 <= arr.length <= 5 * 104
-108 <= arr[i] <= 108

```
class Solution {
    public int minJumps(int[] arr) {
    int n = arr.length;
    if(n == 1) return 0;
    Map<Integer, List<Integer>> map = new HashMap<>();
```

```
    for(int i = 0; i < n; i++) {
        List<Integer> list = map.get(arr[i]);
        if(list == null) map.put(arr[i], list = new ArrayList<>());
        list.add(i);
    }
    int[] visited = new int[n];
    Deque<Integer> forward = new LinkedList<>(), backward = new LinkedList<>();
    visited[0] = 1;
    forward.add(0);
    visited[n - 1] = 2;
    backward.add(n - 1);
    for(int res = 1, dir = 1; ; res++) {
        if(forward.size() > backward.size()) {
            Deque<Integer> temp = forward; forward = backward; backward = temp;
            dir = 3 - dir;
        }
        for(int size = forward.size(); size-- > 0; ) {
            int i = forward.poll();
            List<Integer> list = map.get(arr[i]);
            if(i - 1 >= 0) list.add(i - 1);
            if(i + 1 < n) list.add(i + 1);
            for(int j: list) {
                if(visited[j] == 0) {
                    visited[j] = dir;
                    forward.add(j);
                } else if(visited[j] != dir) return res;
            }
            list.clear();
        }
    }
}
}
```

----------Jump Game V
----------Given an array of integers arr and an integer d. In one step you can jump from index i to index:

i + x where: i + x < arr.length and 0 < x <= d.
i - x where: i - x >= 0 and 0 < x <= d.
In addition, you can only jump from index i to index j if arr[i] > arr[j] and arr[i] > arr[k] for all indices k between i and j (More formally min(i, j) < k < max(i, j)).

You can choose any index of the array and start jumping. Return the maximum number of indices you can visit.

Notice that you can not jump outside of the array at any time.

Example 1:

Input: arr = [6,4,14,6,8,13,9,7,10,6,12], d = 2

Output: 4
Explanation: You can start at index 10. You can jump 10 --> 8 --> 6 --> 7 as shown.
Note that if you start at index 6 you can only jump to index 7. You cannot jump to index 5 because 13 > 9. You cannot jump to index 4 because index 5 is between index 4 and 6 and 13 > 9.
Similarly You cannot jump from index 3 to index 2 or index 1.
Example 2:

Input: arr = [3,3,3,3,3], d = 3
Output: 1
Explanation: You can start at any index. You always cannot jump to any index.
Example 3:

Input: arr = [7,6,5,4,3,2,1], d = 1
Output: 7
Explanation: Start at index 0. You can visit all the indicies.
Example 4:

Input: arr = [7,1,7,1,7,1], d = 2
Output: 2
Example 5:

Input: arr = [66], d = 1
Output: 1

Constraints:

1 <= arr.length <= 1000
1 <= arr[i] <= 10^5
1 <= d <= arr.length

```
class Solution {
    public int maxJumps(int[] arr, int d) {
    int n = arr.length, res = 0, top = 0;
    int[] dp = new int[n], stack = new int[n];
    for(int i = 0; i <= n; i++) {
        while(top > 0 && (i == n || arr[stack[top - 1]] < arr[i])) {
            int r = top - 1, l = r - 1;
            while(l >= 0 && arr[stack[l]] == arr[stack[r]]) l--;
            for(int j = l + 1; j <= r; j++) {
                if(l >= 0 && stack[j] - stack[l] <= d) dp[stack[l]] = Math.max(dp[stack[l]], 1 +
dp[stack[j]]);
                if(i < n && i - stack[j] <= d) dp[i] = Math.max(dp[i], 1 + dp[stack[j]]);
            }
            top -= r - l;
        }
        stack[top++] = i;
    }
    for(int i = 0; i < n; i++) res = Math.max(res, dp[i]);
    return res + 1;
}
}
```

----------Jump Game VI
----------You are given a 0-indexed integer array nums and an integer k.

You are initially standing at index 0. In one move, you can jump at most k steps forward without going outside the boundaries of the array. That is, you can jump from index i to any index in the range [i + 1, min(n - 1, i + k)] inclusive.

You want to reach the last index of the array (index n - 1). Your score is the sum of all nums[j] for each index j you visited in the array.

Return the maximum score you can get.

Example 1:

Input: nums = [1,-1,-2,4,-7,3], k = 2
Output: 7
Explanation: You can choose your jumps forming the subsequence [1,-1,4,3] (underlined above). The sum is 7.
Example 2:

Input: nums = [10,-5,-2,4,0,3], k = 3
Output: 17
Explanation: You can choose your jumps forming the subsequence [10,4,3] (underlined above). The sum is 17.
Example 3:

Input: nums = [1,-5,-20,4,-1,3,-6,-3], k = 2
Output: 0

Constraints:

1 <= nums.length, k <= 105
-104 <= nums[i] <= 104

```
class Solution {
    public int maxResult(int[] nums, int k) {
        int n = nums.length, a = 0, b = 0;
        int[] deq = new int[n];
        deq[0] = n - 1;
        for (int i = n - 2; i >= 0; i--) {
            if (deq[a] - i > k) a++;
            nums[i] += nums[deq[a]];
            while (b >= a && nums[deq[b]] <= nums[i]) b--;
            deq[++b] = i;
        }
        return nums[0];
    }
}
```

----------Jump Game VII

----------You are given a 0-indexed binary string s and two integers minJump and maxJump. In the beginning, you are standing at index 0, which is equal to '0'. You can move from index i to index j if the following conditions are fulfilled:

i + minJump <= j <= min(i + maxJump, s.length - 1), and
s[j] == '0'.
Return true if you can reach index s.length - 1 in s, or false otherwise.

Example 1:

Input: s = "011010", minJump = 2, maxJump = 3
Output: true
Explanation:
In the first step, move from index 0 to index 3.
In the second step, move from index 3 to index 5.
Example 2:

Input: s = "01101110", minJump = 2, maxJump = 3
Output: false

Constraints:

2 <= s.length <= 105
s[i] is either '0' or '1'.
s[0] == '0'
1 <= minJump <= maxJump < s.length

```
class Solution {
    public boolean canReach(String s, int minJump, int maxJump) {
        if(s.charAt(s.length() - 1) != '0' || minJump >= s.length()) return false;
        int farthestVisitedIndex = 0;
        Queue<Integer> q = new ArrayDeque<>();
        q.offer(0);
        while(!q.isEmpty()) {
            int index = q.poll();
            if(index == s.length() - 1) return true;
            int windowStart = Math.max(index + minJump, farthestVisitedIndex + 1);
            int windowEnd = Math.min(index + maxJump, s.length() - 1);
            farthestVisitedIndex = windowEnd;
            for(int i = windowStart; i <= windowEnd; i++) {
                if(s.charAt(i) == '0') q.offer(i);
            }
        }
        return false;
    }
}
```

----------Permutations
----------Given a collection of distinct numbers, return all possible permutations.

FExample 1:

Input: nums = [1,2,3]
Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
Example 2:

Input: nums = [0,1]
Output: [[0,1],[1,0]]
Example 3:

Input: nums = [1]
Output: [[1]]

Constraints:

1 <= nums.length <= 6
-10 <= nums[i] <= 10
All the integers of nums are unique.

```
class Solution {
    List<List<Integer>> ls=new ArrayList<>();
    public List<List<Integer>> permute(int[] nums) {
        helper(nums,0,nums.length-1);
        return ls;
    }
    public void helper(int[] a,int start,int end){
        if(start==end){
            List<Integer> out=new ArrayList<>();
           for(int i:a)
               out.add(i);
            ls.add(out);
        }
        else{
            for(int i=start;i<=end;i++){
                int t1=a[i];
                a[i]=a[start];
                a[start]=t1;
                helper(a,start+1,end);
                int t2=a[i];
                a[i]=a[start];
                a[start]=t2;
            }
        }

    }
}
```

----------Permutations II
----------Given a collection of numbers that might contain duplicates, return all possible unique permutations.

For example,
[1,1,2] have the following unique permutations:
[

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

```
class Solution {
    public static List<List<Integer>> permuteUnique(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        if (nums == null || nums.length == 0) return result;
        int n = nums.length;
        int[] visited = new int[n];
        Arrays.sort(nums);
        helper(nums, visited, new ArrayList<>(), result);
        return result;
    }

    public static void helper(int[] nums, int[] visited, List<Integer> curlist, List<List<Integer>>
result) {
        if (curlist.size() == nums.length) {
            result.add(new ArrayList<>(curlist));
        }
        for (int i = 0; i < nums.length; i++) {
            if (i > 0 && nums[i] == nums[i - 1] && visited[i-1]==0) continue;
            if (visited[i] == 0) {
                visited[i] = 1;
                curlist.add(nums[i]);
                helper(nums, visited, curlist, result);
                visited[i] = 0;
                curlist.remove(curlist.size() - 1);
            }
        }
    }
}
```

----------Rotate Image
----------You are given an n x n 2D matrix representing an image.
Rotate the image by 90 degrees (clockwise).
Follow up:
Could you do this in-place?
Example 1:

Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]
Output: [[7,4,1],[8,5,2],[9,6,3]]
Example 2:

Input: matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]
Output: [[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]
Example 3:

Input: matrix = [[1]]
Output: [[1]]
Example 4:

Input: matrix = [[1,2],[3,4]]
Output: [[3,1],[4,2]]

Constraints:

matrix.length == n
matrix[i].length == n
1 <= n <= 20
-1000 <= matrix[i][j] <= 1000

```
class Solution {
    public void rotate(int[][] matrix) {
        int midSize = matrix.length / 2;
        int n = matrix.length - 1;

        for(int i = 0; i < midSize; i++) {
            int len = n - (2 * i);
            for(int j = 0; j < len; j++) {
                int temp = matrix[i][i + j];
                matrix[i][i + j] = matrix[n - j - i][i];
                matrix[n - j - i][i] = matrix[n - i][n - i - j];
                matrix[n - i][n - i - j] = matrix[i + j][n - i];
                matrix[i + j][n - i] = temp;
            }
        }
    }
}
```

----------Group Anagrams
----------Given an array of strings strs, group the anagrams together. You can return the answer in any order.

An Anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Example 1:

Input: strs = ["eat","tea","tan","ate","nat","bat"]
Output: [["bat"],["nat","tan"],["ate","eat","tea"]]
Example 2:

Input: strs = [""]
Output: [[""]]
Example 3:

Input: strs = ["a"]
Output: [["a"]]

Constraints:

1 <= strs.length <= 104
0 <= strs[i].length <= 100
strs[i] consists of lowercase English letters.

```
class Solution {
 public List<List<String>> groupAnagrams(String[] strs) {
        if (strs == null || strs.length == 0) return new ArrayList<>();
        Map<String, List<String>> map = new HashMap<>();
        for (String s : strs) {
            char[] ca = s.toCharArray();
            Arrays.sort(ca);
            String keyStr = String.valueOf(ca);
            if (!map.containsKey(keyStr)) map.put(keyStr, new ArrayList<>());
            map.get(keyStr).add(s);
        }
        return new ArrayList<>(map.values());
    }
}
```

----------Pow(x, n)

----------Implement pow(x, n), which calculates x raised to the power n (i.e., xn).

Example 1:

Input: x = 2.00000, n = 10
Output: 1024.00000
Example 2:

Input: x = 2.10000, n = 3
Output: 9.26100
Example 3:

Input: x = 2.00000, n = -2
Output: 0.25000
Explanation: 2-2 = 1/22 = 1/4 = 0.25

Constraints:

-100.0 < x < 100.0
-231 <= n <= 231-1
-104 <= xn <= 104

```
class Solution {
    public double myPow(double x, int n) {
                    if(n==0) return 1;
        if(n<0)
        {
            n*=-1;
            double cal= x*myPow(x,n-1);
            return 1/cal;
        }
```

```
        double powvalue=myPow(x,n>>1);
        return (n&1)==0 ? powvalue*powvalue : x*powvalue*powvalue;
    }
}
```

----------Super Pow
----------Your task is to calculate ab mod 1337 where a is a positive integer and b is an extremely large positive integer given in the form of an array.

Example 1:

Input: a = 2, b = [3]
Output: 8
Example 2:

Input: a = 2, b = [1,0]
Output: 1024
Example 3:

Input: a = 1, b = [4,3,3,8,5,2]
Output: 1
Example 4:

Input: a = 2147483647, b = [2,0,0]
Output: 1198

Constraints:

1 <= a <= 231 - 1
1 <= b.length <= 2000
0 <= b[i] <= 9
b doesn't contain leading zeros.

```
class Solution {
    public int superPow(int a, int[] b) {
        int[] buf = new int[10];
        buf[0] = 1;
        a = a%1337;
        for(int i =1;i<10;i++){             //since b is large, and b[i] is between 0 and 9, which reuse several times, we build pattern for them
            buf[i] = (buf[i-1]*a)%1337;
        }
        int ans = buf[b[0]], temp;
        for(int i = 1;i<b.length;i++){
            ans = (ans*ans)%1337;       //calculate ans^10%1337 in 5 lines, since we may have ans^3>Integer.MAX_VALUE.
            temp = ans;
            ans = (ans*ans)%1337;
            ans = (ans*ans)%1337;
            ans = (ans*temp)%1337;
            ans = (ans*buf[b[i]])%1337;
```

```
        }
        return ans;
    }
}
```

----------Sqrt(x)
----------Given a non-negative integer x, compute and return the square root of x.

Since the return type is an integer, the decimal digits are truncated, and only the integer part of the result is returned.

Note: You are not allowed to use any built-in exponent function or operator, such as pow(x, 0.5) or x ** 0.5.

Example 1:

Input: x = 4
Output: 2
Example 2:

Input: x = 8
Output: 2
Explanation: The square root of 8 is 2.82842..., and since the decimal part is truncated, 2 is returned.

Constraints:

0 <= x <= 231 - 1

```
class Solution {
    public int mySqrt(int x) {

        if (x == 0 || x == 1) return x;

        // Binary Search
        int left = 0, right = x;
        while (left < right) {
            // mid = (left + right) / 2 can overflow if right > Integer.MAX_VALUE - left
            int mid = left + (right - left) / 2;

            // same thing here , mid * mid > x can overflow. replace by mid > x / mid
            if (mid > x / mid) {
                right = mid - 1;
            } else {
                left = mid + 1;
                // if mid * mid < x but (mid + 1) * (mid + 1) > x then mid was the right answer
                if (left > x / left) {
                    return mid;
                }
            }
```

```
        }

        return left;
    }
}
```

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

Given an integer n, return all distinct solutions to the n-queens puzzle. You may return the answer in any order.

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

Example 1:

Input: n = 4
Output: [[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]
Explanation: There exist two distinct solutions to the 4-queens puzzle as shown above
Example 2:

Input: n = 1
Output: [["Q"]]

Constraints:

1 <= n <= 9

```
// time: O(n!)
// space: O(n)
class Solution {
    private List<List<String>> res = new ArrayList<>();
    private int[] cols;
    private int[] rows;
    private int[] diagonalsA;
    private int[] diagonalsB;

    public List<List<String>> solveNQueens(int n) {
        cols = new int[n];
        diagonalsA = new int[n * 2 - 1];
        diagonalsB = new int[n * 2 - 1];

        char[][] buffer = new char[n][n];
        for (int i = 0; i < n; i++) Arrays.fill(buffer[i], '.');

        placeQueens(n, 0, buffer);

        return res;
```

```
    }

    private void placeQueens(int n, int row, char[][] buffer) {
        if (row == n) {
            // store
            List<String> sol = toListStr(buffer);
            res.add(sol);

            return;
        }

        for (int i = 0; i < n; i++) {

            if (cols[i] == 1) {
                continue;
            }

            // map.get(row - col)
            // + (n - 1) to offset the index and be able to use int[] arr
            if (diagonalsA[row - i + (n - 1)] == 1) {
                continue;
            }
            if (diagonalsB[row + i] == 1) {
                continue;
            }

            cols[i] = 1;
            diagonalsA[row - i + (n - 1)] = 1;
            diagonalsB[row + i] = 1;

            buffer[row][i] = 'Q';

            placeQueens(n, row + 1, buffer);

            buffer[row][i] = '.';

            cols[i] = 0;
            diagonalsA[row - i + (n - 1)] = 0;
            diagonalsB[row + i] = 0;
        }
    }

    private List<String> toListStr(char[][] buffer) {
        List<String> list = new ArrayList<>();
        for (int i = 0; i < buffer.length; i++) {
            list.add(new String(buffer[i]));
        }

        return list;
    }
}
```

----------N-Queens II

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

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

Example 1:

Input: n = 4
Output: 2
Explanation: There are two distinct solutions to the 4-queens puzzle as shown.
Example 2:

Input: n = 1
Output: 1

Constraints:

1 <= n <= 9

```
class Solution {
    public int totalNQueens(int n) {

        boolean[] ar = new boolean[n]; // to check if this row is attacked
        boolean[] ac = new boolean[n];

        boolean[] ad1 = new boolean[100]; // to check if primary diaglonal(r+c is constant) is attacked.
        boolean[] ad2 = new boolean[100]; // to check if 2ndary diaglonal(r-c is constant) is attacked.

        return backtrack(n, n, 0, ar,ac,ad1,ad2);

    }

    public int backtrack(int n, int tp, int r,  boolean[] ar, boolean[] ac, boolean[] ad1,  boolean[] ad2) {

        // tp is how many queens left to place.
        // r : present row

        if (tp == 0) return 1;
        if (r > n-1) return 0;
        if (r == n-1 && ar[r]) return 0;

        int ans = 0;

        if (ar[r]) {
            // if present row is attacked, visit the next row, no need of going over this row's cols trying to place
            ans += backtrack(n, tp, r+1, ar, ac, ad1, ad2);
```

```
            return ans;
        }

        for (int c = 0; c < n; c++) {

            if ( ac[c] || ad1[r-c+10] || ad2[r+c] ) continue;

            // mark this r,c attacked and the diagonals too; +10 as index goes -ve
            ac[c] = true; ad1[r-c+10] = true; ad2[r+c] = true;

            ans += backtrack(n, tp-1, r+1, ar,ac,ad1,ad2);

            ac[c] = false; ad1[r-c+10] = false; ad2[r+c] = false;

        }

        return ans;

    }
}
```

----------Minimum Operations to Reduce X to Zero----------You are given an integer array nums and an integer x. In one operation, you can either remove the leftmost or the rightmost element from the array nums and subtract its value from x. Note that this modifies the array for future operations.

Return the minimum number of operations to reduce x to exactly 0 if it is possible, otherwise, return -1.

Example 1:

Input: nums = [1,1,4,2,3], x = 5
Output: 2
Explanation: The optimal solution is to remove the last two elements to reduce x to zero.
Example 2:

Input: nums = [5,6,7,8,9], x = 4
Output: -1
Example 3:

Input: nums = [3,2,20,1,1,3], x = 10
Output: 5
Explanation: The optimal solution is to remove the last three elements and the first two elements (5 operations in total) to reduce x to zero.

Constraints:

1 <= nums.length <= 105
1 <= nums[i] <= 104
1 <= x <= 109

```
class Solution {
    // 3 ms. 100%
public int minOperations(int[] nums, int x) {
    int target = -x;
    for(int num: nums) target += num;
    if(target == 0) return nums.length;
    if(target < 0) return -1;
    int left = 0, sum = 0, L = 0;
    for(int right = 0; right < nums.length; right++) {
        sum += nums[right];
        while(sum > target) {
            sum -= nums[left++];
        }
        if(sum == target) {
            L = Math.max(L, right - left + 1);
        }
    }
    return L > 0 ? nums.length - L : -1;
}
}
```

----------X of a Kind in a Deck of Cards
----------In a deck of cards, each card has an integer written on it.

Return true if and only if you can choose X >= 2 such that it is possible to split the entire deck into 1 or more groups of cards, where:

Each group has exactly X cards.
All the cards in each group have the same integer.

Example 1:

Input: deck = [1,2,3,4,4,3,2,1]
Output: true
Explanation: Possible partition [1,1],[2,2],[3,3],[4,4].
Example 2:

Input: deck = [1,1,1,2,2,2,3,3]
Output: false
Explanation: No possible partition.
Example 3:

Input: deck = [1]
Output: false
Explanation: No possible partition.
Example 4:

Input: deck = [1,1]
Output: true
Explanation: Possible partition [1,1].
Example 5:

Input: deck = [1,1,2,2,2,2]

Output: true
Explanation: Possible partition [1,1],[2,2],[2,2].

Constraints:

1 <= deck.length <= 104
0 <= deck[i] < 104

```
class Solution {

    int gcd(int a, int b) {                /* to find gcd*/
        if(a<b) {a=a+b; b=a-b; a=a-b;}    /*if a<b swap */
        if(a%b==0) return b;
        else return gcd(b,a%b);

    }
    public boolean hasGroupsSizeX(int[] deck) {
        int len = deck.length;
        if(len == 1) return false;

        /*put each element of deck and their frequency in a hash map*/

        HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
        for(int i:deck) {
            if(map.containsKey(i)) map.put(i, map.get(i)+1);
            else map.put(i,1);
        }

        /*select a frequncy and repeat doing gcd for all the frequency values*/

        int min = map.get(deck[0]);
        for(int i:map.values()) min = gcd(min,i);

        /*if final gcd is 1 we cannot split the deck with >=2 cards*/

        return min!=1;

    }
}
```

----------N-th Tribonacci Number
----------The Tribonacci sequence Tn is defined as follows:

T0 = 0, T1 = 1, T2 = 1, and Tn+3 = Tn + Tn+1 + Tn+2 for n >= 0.

Given n, return the value of Tn.

Example 1:

Input: n = 4
Output: 4

Explanation:
T_3 = 0 + 1 + 1 = 2
T_4 = 1 + 1 + 2 = 4
Example 2:

Input: n = 25
Output: 1389537

Constraints:

0 <= n <= 37
The answer is guaranteed to fit within a 32-bit integer, ie. answer <= 2^31 - 1

```
class Solution {
    public int tribonacci(int n) {
        int T0=0,T1=1,T2=1,curr=0;
        if(n<=1)
            return n;
        if(n==2)
            return 1;
        for(int i=3;i<=n;i++){
            curr=T0+T1+T2;
            T0=T1;
            T1=T2;
            T2=curr;
        }
        return curr;
    }
}
```

----------Special Array With X Elements Greater Than or Equal X
----------You are given an array nums of non-negative integers. nums is considered special if there exists a number x such that there are exactly x numbers in nums that are greater than or equal to x.

Notice that x does not have to be an element in nums.

Return x if the array is special, otherwise, return -1. It can be proven that if nums is special, the value for x is unique.

Example 1:

Input: nums = [3,5]
Output: 2
Explanation: There are 2 values (3 and 5) that are greater than or equal to 2.
Example 2:

Input: nums = [0,0]
Output: -1
Explanation: No numbers fit the criteria for x.

If x = 0, there should be 0 numbers >= x, but there are 2.
If x = 1, there should be 1 number >= x, but there are 0.
If x = 2, there should be 2 numbers >= x, but there are 0.
x cannot be greater since there are only 2 numbers in nums.
Example 3:

Input: nums = [0,4,3,0,4]
Output: 3
Explanation: There are 3 values that are greater than or equal to 3.
Example 4:

Input: nums = [3,6,7,7,0]
Output: -1

Constraints:

1 <= nums.length <= 100
0 <= nums[i] <= 1000

```
class Solution {
    public int specialArray(int[] nums) {
        int x = nums.length;
        int[] counts = new int[x+1];

        for(int elem : nums)
            if(elem >= x)
                counts[x]++;
            else
                counts[elem]++;

        int res = 0;
        for(int i = counts.length-1; i > 0; i--) {
            res += counts[i];
            if(res == i)
                return i;
        }

        return -1;
    }
}
```

----------N-ary Tree Preorder Traversal
----------Given the root of an n-ary tree, return the preorder traversal of its nodes' values.

Nary-Tree input serialization is represented in their level order traversal. Each group of children is separated by the null value (See examples)

Example 1:

Input: root = [1,null,3,2,4,null,5,6]
Output: [1,3,5,6,2,4]
Example 2:

Input: root = [1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14]
Output: [1,2,3,6,7,11,14,4,8,12,5,9,13,10]

Constraints:

The number of nodes in the tree is in the range [0, 104].
0 <= Node.val <= 104
The height of the n-ary tree is less than or equal to 1000.

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

```
/*
// Definition for a Node.
class Node {
    public int val;
    public List<Node> children;

    public Node() {}

    public Node(int _val) {
        val = _val;
    }

    public Node(int _val, List<Node> _children) {
        val = _val;
        children = _children;
    }
};
*/

class Solution {
    public List<Integer> preorder(Node root) {
        List<Integer> res = new ArrayList();
        solve(root, res);
        return res;
    }

    void solve(Node root, List<Integer> res) {
        if(root == null) {
            return;
        }
        res.add(root.val);
        for(Node next : root.children) {
            solve(next, res);
        }
```

```
    }
}
```

----------N-ary Tree Postorder Traversal
----------Given the root of an n-ary tree, return the postorder traversal of its nodes' values.

Nary-Tree input serialization is represented in their level order traversal. Each group of children is separated by the null value (See examples)

Example 1:

Input: root = [1,null,3,2,4,null,5,6]
Output: [5,6,3,2,4,1]
Example 2:

Input: root = [1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14]
Output: [2,6,14,11,7,3,12,8,4,13,9,10,5,1]

Constraints:

The number of nodes in the tree is in the range [0, 104].
0 <= Node.val <= 104
The height of the n-ary tree is less than or equal to 1000

```
/*
// Definition for a Node.
class Node {
    public int val;
    public List<Node> children;

    public Node() {}

    public Node(int _val) {
        val = _val;
    }

    public Node(int _val, List<Node> _children) {
        val = _val;
        children = _children;
    }
};
*/

class Solution {
    List<Integer> result;
    public List<Integer> postorder(Node root) {
        result = new ArrayList<>();
        helper(root);
```

```
        return result;
    }

    private void helper(Node root){
        if (root == null)
            return;
        for (int i=0;i<root.children.size();i++){
            helper(root.children.get(i));
        }
        result.add(root.val);
    }
}
```

----------Maximum Subarray
----------Find the contiguous subarray within an array (containing at least one number) which has the largest sum.
Example 1:

Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: [4,-1,2,1] has the largest sum = 6.
Example 2:

Input: nums = [1]
Output: 1
Example 3:

Input: nums = [5,4,-1,7,8]
Output: 23

Constraints:

1 <= nums.length <= 105
-104 <= nums[i] <= 104

```
class Solution {
    public int maxSubArray(int[] nums) {
        int sum = 0;
        int max_i = nums[0];
        for(int i=0;i<nums.length;i++) {
        sum+=nums[i];
        if(sum>max_i) max_i = sum;
        if(sum < 0) sum = 0;
        }
        return max_i;
    }
}
```

----------Spiral Matrix
----------Given an m x n matrix, return all elements of the matrix in spiral order.

Example 1:

Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]
Output: [1,2,3,6,9,8,7,4,5]
Example 2:

Input: matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
Output: [1,2,3,4,8,12,11,10,9,5,6,7]

Constraints:

m == matrix.length
n == matrix[i].length
1 <= m, n <= 10
-100 <= matrix[i][j] <= 100

```
class Solution {
    public List<Integer> spiralOrder(int[][] matrix) {
        int m=matrix.length,n=matrix[0].length;
        List<Integer> out=new ArrayList<>();
        int minrow=0,mincol=0,maxrow=m-1,maxcol=n-1;
         while(minrow<=maxrow&&mincol<=maxcol){
                      //top-most row
            for(int j=mincol;j<=maxcol;j++)
                out.add(matrix[minrow][j]);
            minrow++;
        //right-most col
            for(int i=minrow;i<=maxrow;i++)
                out.add(matrix[i][maxcol]);
            maxcol--;
        //bottom-most row
                      if(minrow<=maxrow){
            for(int j=maxcol;j>=mincol;j--){
                out.add(matrix[maxrow][j]);
            }
                      }
            maxrow--;
        //left-most col
                      if(mincol<=maxcol){
            for(int i=maxrow;i>=minrow;i--){
                out.add(matrix[i][mincol]);
            }
                      }
                      mincol++;
        }
        return out;
    }
}
```

----------Jump Game

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

For example:
A = [2,3,1,1,4], return true.
A = [3,2,1,0,4], return false.

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

----------Merge Intervals
----------Given an array of intervals where intervals[i] = [starti, endi], merge all overlapping intervals, and return an array of the non-overlapping intervals that cover all the intervals in the input.

Example 1:

Input: intervals = [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: Since intervals [1,3] and [2,6] overlaps, merge them into [1,6].
Example 2:

Input: intervals = [[1,4],[4,5]]
Output: [[1,5]]
Explanation: Intervals [1,4] and [4,5] are considered overlapping.

Constraints:

1 <= intervals.length <= 104
intervals[i].length == 2
0 <= starti <= endi <= 104

```
class Solution {
    public int[][] merge(int[][] intervals) {
                    // max is being used to construct the tracking arrays
        int max = Integer.MIN_VALUE;
        List<int[]> mergedIntervals = new ArrayList<>();
        for (int[] i : intervals) {
            if (i[1] > max) {
                max = i[1];
```

```
            }
        }

                    // Edge case handling
        if (max == Integer.MIN_VALUE) {
            return intervals;
        }
        // Using two arrays for inputs of the form (x,x)
        int[] startIndexArray = new int[max + 1];
        int[] endIndexArray = new int[max + 1];
        for (int[] i : intervals) {
            int start = i[0];
            int end = i[1];
            startIndexArray[start]++;
            endIndexArray[end]--;
        }

        int sum = 0;
        int start = -1;
        int end = -1;
        for (int i = 0; i < max + 1; i++) {
            if (startIndexArray[i] > 0 && sum == 0) {
                start = i;
            }

            sum += startIndexArray[i];
            sum += endIndexArray[i];

            if (endIndexArray[i] < 0 && sum == 0) {
                end = i;
            }

            if (start != -1 && end != -1) {
                mergedIntervals.add(new int[] {start, end});
                start = -1;
                end = -1;
            }
        }

        return mergedIntervals.toArray(new int[mergedIntervals.size()][]);
    }
}
```

----------Insert Interval
----------You are given an array of non-overlapping intervals intervals where intervals[i] = [starti, endi] represent the start and the end of the ith interval and intervals is sorted in ascending order by starti. You are also given an interval newInterval = [start, end] that represents the start and end of another interval.

Insert newInterval into intervals such that intervals is still sorted in ascending order by starti and intervals still does not have any overlapping intervals (merge overlapping intervals if necessary).

Return intervals after the insertion.

Example 1:

Input: intervals = [[1,3],[6,9]], newInterval = [2,5]
Output: [[1,5],[6,9]]
Example 2:

Input: intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]], newInterval = [4,8]
Output: [[1,2],[3,10],[12,16]]
Explanation: Because the new interval [4,8] overlaps with [3,5],[6,7],[8,10].
Example 3:

Input: intervals = [], newInterval = [5,7]
Output: [[5,7]]
Example 4:

Input: intervals = [[1,5]], newInterval = [2,3]
Output: [[1,5]]
Example 5:

Input: intervals = [[1,5]], newInterval = [2,7]
Output: [[1,7]]

Constraints:

0 <= intervals.length <= 104
intervals[i].length == 2
0 <= starti <= endi <= 105
intervals is sorted by starti in ascending order.
newInterval.length == 2
0 <= start <= end <= 105

```
class Solution {
    public int[][] insert(int[][] intervals, int[] newInterval) {
        if (intervals.length == 0) {
            return new int[][]{newInterval};
        }

        int min = findMin(intervals, newInterval); // idx of the earlies interval which intersects with a
new one
        int max = findMax(intervals, newInterval); // idx of the latest interval which intersects with a
new one
        if (min == max) {
            intervals[min] = new int[]{Math.min(intervals[min][0], newInterval[0]),
Math.max(intervals[min][1], newInterval[1])};
            return intervals;
        }

        int[][] newIntervals = new int[intervals.length - (max - min)][2];

        // copy all intervals < newInterval
        for (int i = 0; i < min; i++) {
```

```
            newIntervals[i] = intervals[i];
        }

        // copy all intervals > newInterval
        for (int i = intervals.length - 1; i > max; i--) {
            newIntervals[i - (max - min)] = intervals[i];
        }

        // insert interval
        if (max < min) { // couldn't find intersecting intervals, just append
            newIntervals[min] = newInterval;
        } else {
            newIntervals[min] = new int[]{Math.min(intervals[min][0], newInterval[0]),
Math.max(intervals[max][1], newInterval[1])};
        }

        return newIntervals;
    }

    private static int findMin(int[][] intervals, int[] interval) {
        int idx = -1;
        int lo = 0;
        int hi = intervals.length - 1;
        while (lo <= hi) {
            int mid = lo + (hi - lo) / 2;
            int[] midInt = intervals[mid];
            if (intersects(midInt, interval)) {
                idx = mid;
                hi = mid - 1;
            } else if (interval[0] > midInt[1]) {
                lo = mid + 1;
            } else {
                hi = mid - 1;
            }
        }
        return idx == -1 ? lo : idx;
    }

    private static int findMax(int[][] intervals, int[] interval) {
        int idx = -1;
        int lo = 0;
        int hi = intervals.length - 1;
        while (lo <= hi) {
            int mid = lo + (hi - lo) / 2;
            int[] midInt = intervals[mid];
            if (intersects(midInt, interval)) {
                idx = mid;
                lo = mid + 1;
            } else if (interval[0] > midInt[1]) {
                lo = mid + 1;
            } else {
                hi = mid - 1;
            }
        }
```

```
        return idx == -1 ? hi : idx;
    }

    private static boolean intersects(int[] left, int[] right) {
        return left[0] <= right[1] && left[1] >= right[0];
    }
}
```

---Length of Last Word

---Given a string s consisting of some words separated by some number of spaces, return the length of the last word in the string.

A word is a maximal substring consisting of non-space characters only.

Example 1:

Input: s = "Hello World"
Output: 5
Explanation: The last word is "World" with length 5.
Example 2:

Input: s = "   fly me   to   the moon "
Output: 4
Explanation: The last word is "moon" with length 4.
Example 3:

Input: s = "luffy is still joyboy"
Output: 6
Explanation: The last word is "joyboy" with length 6.

Constraints:

1 <= s.length <= 104
s consists of only English letters and spaces ' '.
There will be at least one word in s.

```
class Solution {
    public int lengthOfLastWord(String s) {
        int sum = 0;
        int i=s.length()-1;

        while (i >= 0 && s.charAt(i) == ' ')
            i--;

        for (int j=i; j>=0; j--) {
            char c = s.charAt(j);
            if (c != ' ') {
                sum++;
            } else {
                break;
            }
        }
```

```
        return sum;
    }
}

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

----------Spiral Matrix II
----------Given a positive integer n, generate an n x n matrix filled with elements from 1 to n2 in spiral order.

Example 1:

Input: n = 3
Output: [[1,2,3],[8,9,4],[7,6,5]]
Example 2:

Input: n = 1
Output: [[1]]

Constraints:

1 <= n <= 20

```
class Solution {
    public int[][] generateMatrix(int n) {

        int minrow = 0, mincol = 0, maxrow = n-1, maxcol = n-1, count = 1;
        int out[][] = new int[n][n];

        while(count <= n*n){
            int i=minrow, j=mincol;
            while(j<=maxcol){
                out[i][j++] = count++;
            }j--; i++;
            while(i<=maxrow){
                out[i++][j] = count++;
            }i--;j--;
            while(j>=mincol){
                out[i][j--] = count++;
            }i--;j++;
            while(i>= minrow+1){
                out[i--][j] = count++;
            }
            minrow++; mincol++;
```

```
            maxrow--; maxcol--;
        }
        return out;
    }
}
```
----------Permutation Sequence
----------The set [1,2,3,…,n] contains a total of n! unique permutations.
By listing and labeling all of the permutations in order,
We get the following sequence (ie, for n = 3):
"123"
"132"
"213"
"231"
"312"
"321"

Given n and k, return the kth permutation sequence.
Note: Given n will be between 1 and 9 inclusive.

Constraints:

1 <= n <= 9
1 <= k <= n!

```
public class Solution {
    public String getPermutation(int n, int k) {
        int fact = 1;
        char[] seq = new char[n];
        for (int i = 0; i < n; i++) {
            if (i != n - 1) fact *= i + 1;
            seq[i] = Character.forDigit(i + 1, 10);
        }
        int index = 0;
        while (index < n - 1) {
            int m = index + (k - 1) / fact;
            char lead = seq[m];
            for (int i = m; i >= index + 1; i--) {
                seq[i] = seq[i - 1];
            }
            seq[index] = lead;
            k = (k - 1) % fact + 1;
            fact /= n - index - 1;
            index++;
        }
        return new String(seq);
    }
}
```
----------Rotate List
----------Given the head of a linked list, rotate the list to the right by k places.

Example 1:

Input: head = [1,2,3,4,5], k = 2

Output: [4,5,1,2,3]
Example 2:

Input: head = [0,1,2], k = 4
Output: [2,0,1]

Constraints:

The number of nodes in the list is in the range [0, 500].
-100 <= Node.val <= 100
0 <= k <= 2 * 109

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode rotateRight(ListNode head, int k) {
        if(head == null || head.next == null || k == 0) return head;

        int length=0;
        int actualRotations = 0;
        ListNode temp = head;
        ListNode newHead = null;

        while(temp != null){
            length++;
                            //make a circular linked list
            if(temp.next == null && k%length != 0){
                temp.next = head;
                break;
            }
            temp = temp.next;
        }

        k=k%length;

        if(k == 0) return head;


        actualRotations = length - k;

        while(--actualRotations != 0){
            head=head.next;
        }
        newHead = head.next;
        head.next = null;
```

```
        return newHead;


    }
}
```

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

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

How many possible unique paths are there?

Example 1:

Input: m = 3, n = 7
Output: 28
Example 2:

Input: m = 3, n = 2
Output: 3
Explanation:
From the top-left corner, there are a total of 3 ways to reach the bottom-right corner:
1. Right -> Down -> Down
2. Down -> Down -> Right
3. Down -> Right -> Down
Example 3:

Input: m = 7, n = 3
Output: 28
Example 4:

Input: m = 3, n = 3
Output: 6

Constraints:

1 <= m, n <= 100
It's guaranteed that the answer will be less than or equal to 2 * 109.

```
class Solution {
    public int uniquePaths(int m, int n) {
    int smaller = m>n ? n-1 : m-1;
    int totalsteps = m+n-2;
    long result = 1;
    for (int counter = 1; counter<=smaller; counter++){
```

```
        result *= totalsteps--;
        result /= counter;
    }
    return (int)result;
}
}
```
----------Unique Paths II
----------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).

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

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

Example 1:

Input: obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]
Output: 2
Explanation: There is one obstacle in the middle of the 3x3 grid above.
There are two ways to reach the bottom-right corner:
1. Right -> Right -> Down -> Down
2. Down -> Down -> Right -> Right
Example 2:

Input: obstacleGrid = [[0,1],[0,0]]
Output: 1

Constraints:

m == obstacleGrid.length
n == obstacleGrid[i].length
1 <= m, n <= 100
obstacleGrid[i][j] is 0 or 1.

```
class Solution
{
    public int uniquePathsWithObstacles(int[][] obstacleGrid)
    {
        int m = obstacleGrid.length, n = obstacleGrid[0].length;

        if(obstacleGrid[0][0]==1 || obstacleGrid[m-1][n-1]==1)
            return 0;

        int[][] mat = new int[m][n];
```

```
        boolean obs = false;
        for(int i = 0; i < m; i++)
        {
            if(obstacleGrid[i][0]==1)
                obs = true;
            if(obs)
                mat[i][0] = 0;
            else
                mat[i][0] = 1;
        }

        obs = false;
        for(int j = 0; j < n; j++)
        {
            if(obstacleGrid[0][j]==1)
                obs = true;
            if(obs)
                mat[0][j] = 0;
            else
                mat[0][j] = 1;
        }

        for(int r = 1; r < m; r++)
        {
            for(int c = 1; c < n; c++)
            {
                if(obstacleGrid[r][c]==1)
                    mat[r][c] = 0;
                else
                    mat[r][c] = mat[r-1][c] + mat[r][c-1];
            }
        }

        return mat[m-1][n-1];
    }
}
```

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

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

Example 1:

Input: grid = [[1,3,1],[1,5,1],[4,2,1]]
Output: 7
Explanation: Because the path 1 → 3 → 1 → 1 → 1 minimizes the sum.
Example 2:

Input: grid = [[1,2,3],[4,5,6]]

Output: 12

Constraints:

m == grid.length
n == grid[i].length
1 <= m, n <= 200
0 <= grid[i][j] <= 100

```
class Solution {
    public int minPathSum(int[][] grid) {
        return backtrack(grid, 0 , 0);
    }

    int backtrack(int[][] grid, int r, int c){
        if(r>= grid.length || c>=grid[0].length) return Integer.MAX_VALUE;
        if(r == grid.length - 1 && c == grid[0].length -1) return grid[r][c];
        if(grid[r][c] < 0) return -1*grid[r][c];
        int temp =  grid[r][c] + Math.min(backtrack(grid, r+1, c) ,  backtrack(grid, r, c+1));
        grid[r][c] = -1*temp;
        return temp;
    }
}
```

----------Valid Number
----------A valid number can be split up into these components (in order):

A decimal number or an integer.
(Optional) An 'e' or 'E', followed by an integer.
A decimal number can be split up into these components (in order):

(Optional) A sign character (either '+' or '-').
One of the following formats:
One or more digits, followed by a dot '.'.
One or more digits, followed by a dot '.', followed by one or more digits.
A dot '.', followed by one or more digits.
An integer can be split up into these components (in order):

(Optional) A sign character (either '+' or '-').
One or more digits.
For example, all the following are valid numbers: ["2", "0089", "-0.1", "+3.14", "4.", "-.9", "2e10", "-90E3", "3e+7", "+6e-1", "53.5e93", "-123.456e789"], while the following are not valid numbers: ["abc", "1a", "1e", "e3", "99e2.5", "--6", "-+3", "95a54e53"].

Given a string s, return true if s is a valid number.

Example 1:

Input: s = "0"
Output: true
Example 2:

Input: s = "e"
Output: false
Example 3:

Input: s = "."
Output: false
Example 4:

Input: s = ".1"
Output: true

Constraints:

1 <= s.length <= 20
s consists of only English letters (both uppercase and lowercase), digits (0-9), plus '+', minus '-', or dot

```
class Solution {
    public boolean isNumber(String s) {

                if (s.length() < 1)
                        return false;

                boolean hasDigit = false, hasDot = false, hasE = false;
                int countSign = 0;

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

        char ch = s.charAt(i);

                        // see if it is valid character or not
                        if (isValidCharacter(ch) == false)
                            return false;

                        // check digit
                        if (ch >= '0' && ch <= '9')
                                hasDigit = true;

                        // check sign
                        else if (ch == '+' || ch == '-') {
                                // sign can be at max two
                                if (countSign == 2)
                                        return false;

                                // sign in between first and last character
                                if (i > 0 && !(s.charAt(i - 1) == 'e' || s.charAt(i - 1) == 'E')) {
                                    return false;
            }
            // sign can't be at last of String
                                if (i == s.length() - 1)
                                        return false;
```

```
                countSign++;
                        }

                        // check dot
                         else if (ch == '.') {
                                // if dot and E/e was already there, can have only one dot
                                if (hasDot || hasE)
                                        return false;

                                // if dot is at last, and there was no digit before it
                                if (i == s.length() - 1 && !hasDigit)
                                        return false;

                                hasDot = true;
                        }

                        // check e/E
                        else if (ch == 'e' || ch == 'E' ) {
                // can't have more than 1 e/E, must have digit before e/E
                                if (hasE || !hasDigit)
                                        return false;

                                // e/E can't be at last or at first
                                if (i == s.length() - 1 || i == 0)
                                        return false;

                                hasE = true;
                        }


                }
                return true;
    }

        private  boolean isValidCharacter(char ch) {
                return ch >= 48 && ch <= 57 || ch == 'e' || ch == 'E' || ch == '.' || ch == '+' || ch ==
'-';
        }

}
```

----------Plus One----------

You are given a large integer represented as an integer array digits, where each digits[i] is the ith digit of the integer. The digits are ordered from most significant to least significant in left-to-right order. The large integer does not contain any leading 0's.

Increment the large integer by one and return the resulting array of digits.

Example 1:

Input: digits = [1,2,3]
Output: [1,2,4]
Explanation: The array represents the integer 123.

Incrementing by one gives 123 + 1 = 124.
Thus, the result should be [1,2,4].
Example 2:

Input: digits = [4,3,2,1]
Output: [4,3,2,2]
Explanation: The array represents the integer 4321.
Incrementing by one gives 4321 + 1 = 4322.
Thus, the result should be [4,3,2,2].
Example 3:

Input: digits = [0]
Output: [1]
Explanation: The array represents the integer 0.
Incrementing by one gives 0 + 1 = 1.
Thus, the result should be [1].
Example 4:

Input: digits = [9]
Output: [1,0]
Explanation: The array represents the integer 9.
Incrementing by one gives 9 + 1 = 10.
Thus, the result should be [1,0].

Constraints:

1 <= digits.length <= 100
0 <= digits[i] <= 9
digits does not contain any leading 0's.

```
class Solution {
    public int[] plusOne(int[] digits) {
                    // adding 1 to last digit, so for-loop can catch it if last digit == 10
        digits[digits.length-1] += 1;
        for (int i = digits.length - 1; i > -1; i--) {
                            // if current value in array is == 10, increment value on the left by 1 and
set current value to 0
            if (digits[i] == 10 && i > 0) {
                digits[i - 1] += 1;
                digits[i] = 0;

            // since only scenario where leftmost integer is 10 is when all other values are 0
                            // hence, create new array of size + 1, set first integer to 1
            } else if (digits[0] == 10) {
                int[] digitsNew = new int[digits.length + 1];
                digitsNew[0] = 1;
                return digitsNew;

                            // if current value < 10, no longer need to carry value
            } else {
                break;
            }
        }
        return digits;
```

```
    }
}
```
----------Add Binary
----------Given two binary strings a and b, return their sum as a binary string.

Example 1:

Input: a = "11", b = "1"
Output: "100"
Example 2:

Input: a = "1010", b = "1011"
Output: "10101"

Constraints:

1 <= a.length, b.length <= 104
a and b consist only of '0' or '1' characters.
Each string does not contain leading zeros except for the zero itself.

```
class Solution {
    public String addBinary(String a, String b) {
        var result = new StringBuilder();

        for (int i = a.length() - 1, j = b.length() - 1, carry = 0; i >= 0 || j >= 0 || carry != 0; i--, j--) {
            var sum = carry;
            if (i >= 0)
                sum += Character.digit(a.charAt(i), 2);
            if (j >= 0)
                sum += Character.digit(b.charAt(j), 2);
            carry = sum / 2;
            sum %= 2;
            result.append(sum);
        }

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

----------Text Justification
----------Given an array of strings words and a width maxWidth, format the text such that each line has exactly maxWidth characters and is fully (left and right) justified.

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

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

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

Note:

A word is defined as a character sequence consisting of non-space characters only.
Each word's length is guaranteed to be greater than 0 and not exceed maxWidth.
The input array words contains at least one word.

Example 1:

Input: words = ["This", "is", "an", "example", "of", "text", "justification."], maxWidth = 16
Output:
[
  "This is an",
  "example of text",
  "justification. "
]
Example 2:

Input: words = ["What","must","be","acknowledgment","shall","be"], maxWidth = 16
Output:
[
  "What must be",
  "acknowledgment ",
  "shall be   "
]
Explanation: Note that the last line is "shall be   " instead of "shall   be", because the last line must be left-justified instead of fully-justified.
Note that the second line is also left-justified becase it contains only one word.
Example 3:

Input: words =
["Science","is","what","we","understand","well","enough","to","explain","to","a","computer.","Art","is","everything","else","we","do"], maxWidth = 20
Output:
[
  "Science is what we",
  "understand   well",
  "enough to explain to",
  "a computer. Art is",
  "everything else we",
  "do   "
]

Constraints:

1 <= words.length <= 300
1 <= words[i].length <= 20
words[i] consists of only English letters and symbols.
1 <= maxWidth <= 100
words[i].length <= maxWidth

```
class Solution {
    public List<String> fullJustify(String[] words, int maxWidth) {
        List<String> res = new ArrayList<>();
        if (words == null) return res;
        // l, r are the boundary of the window that constitutes one line
                    int l = 0, r = 0;
        while (l < words.length) {
            int len = 0;
            while (r < words.length && len + words[r].length() + 1 <= maxWidth + 1) len += words[r+
+].length() + 1;
            int space = 1, extra = 0;
            if (r != words.length && r != l + 1) {
                space = (maxWidth - len + 1) / (r - l - 1) + 1;
                extra = (maxWidth - len + 1) % (r - l - 1);
            }
            StringBuilder sb = new StringBuilder(words[l++]);
            while (l < r) {
                for (int i = 0; i < space; i++) sb.append(' ');
                if (extra-- > 0) sb.append(' ');
                sb.append(words[l++]);
            }
            int remaining = maxWidth - sb.length();
            while (remaining-- > 0) sb.append(' ');
            res.add(sb.toString());
        }
        return res;
    }
}
```

----------Climbing Stairs
----------You are climbing a staircase. It takes n steps to reach the top.

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

Example 1:

Input: n = 2
Output: 2
Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps
Example 2:

Input: n = 3
Output: 3
Explanation: There are three ways to climb to the top.
1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step

Constraints:

1 <= n <= 45

```
class Solution {
    public int climbStairs(int n) {
        int prev1 = 0, prev2 = 1, result = 0;
        for (int i = 1; i <= n; i++) {
            result = prev1 + prev2;
            prev1 = prev2;
            prev2 = result;
        }
        return result;
    }
}
```

----------Simplify Path
----------Given a string path, which is an absolute path (starting with a slash '/') to a file or directory in a Unix-style file system, convert it to the simplified canonical path.

In a Unix-style file system, a period '.' refers to the current directory, a double period '..' refers to the directory up a level, and any multiple consecutive slashes (i.e. '//') are treated as a single slash '/'. For this problem, any other format of periods such as '...' are treated as file/directory names.

The canonical path should have the following format:

The path starts with a single slash '/'.
Any two directories are separated by a single slash '/'.
The path does not end with a trailing '/'.
The path only contains the directories on the path from the root directory to the target file or directory (i.e., no period '.' or double period '..')
Return the simplified canonical path.

Example 1:

Input: path = "/home/"
Output: "/home"
Explanation: Note that there is no trailing slash after the last directory name.
Example 2:

Input: path = "/../"
Output: "/"
Explanation: Going one level up from the root directory is a no-op, as the root level is the highest level you can go.
Example 3:

Input: path = "/home//foo/"
Output: "/home/foo"
Explanation: In the canonical path, multiple consecutive slashes are replaced by a single one.
Example 4:

Input: path = "/a/./b/../../c/"
Output: "/c"

Constraints:

1 <= path.length <= 3000
path consists of English letters, digits, period '.', slash '/' or '_'.
path is a valid absolute Unix path.

```
class Solution {
    public String simplifyPath(String path) {
        Stack<String> stk = new Stack();
        int start = 0;
        while (start < path.length()) {
            while (start < path.length() && path.charAt(start) == '/') start++;
            int end = start;
            while (end < path.length() && path.charAt(end) != '/') end ++;
            String s = path.substring(start, end);
            if (s.equals("..")) {
                if (! stk.empty()) stk.pop();
            }
            else if (! s.equals(".") && ! s.equals("")) {
                stk.push(s);
            }
            start = end + 1;
        }
        StringBuilder ans = new StringBuilder();
        while (! stk.empty()) {
            ans.insert(0, stk.pop());
            ans.insert(0, "/");
        }
        return (ans.length() > 0) ? ans.toString(): "/";
    }
}
```

----------Edit Distance
----------Given two strings word1 and word2, return the minimum number of operations required to convert word1 to word2.

You have the following three operations permitted on a word:

Insert a character
Delete a character
Replace a character

Example 1:

Input: word1 = "horse", word2 = "ros"
Output: 3
Explanation:
horse -> rorse (replace 'h' with 'r')

rorse -> rose (remove 'r')
rose -> ros (remove 'e')
Example 2:

Input: word1 = "intention", word2 = "execution"
Output: 5
Explanation:
intention -> inention (remove 't')
inention -> enention (replace 'i' with 'e')
enention -> exention (replace 'n' with 'x')
exention -> exection (replace 'n' with 'c')
exection -> execution (insert 'u')

Constraints:

0 <= word1.length, word2.length <= 500
word1 and word2 consist of lowercase English letters.

```
class Solution {
 public int minDistance(String word1, String word2) {
        int m = word1.length();
        int n = word2.length();
        if (m == 0) {
            return n;
        }
        if (n == 0) {
            return m;
        }
        int[][] dp = new int[2][n + 1];
        for (int i = 1; i <= n; i++) {
            dp[0][i] = i;
        }
        for (int i = 1; i <= m; i++) {
            dp[1][0] = i;
            for (int j = 1; j <= n; j++) {
                if (word1.charAt(i - 1) == word2.charAt(j - 1)) {
                    dp[1][j] = dp[0][j - 1];
                } else {
                    //dp[i][j] = Min(delete,add,change) + 1
                    dp[1][j] = Math.min(dp[0][j], Math.min(dp[0][j - 1], dp[1][j - 1])) + 1;
                }
            }
            System.arraycopy(dp[1],0,dp[0],0,n+1);
        }
        return dp[1][n];
    }
}
```

----------Set Matrix Zeroes
----------Given an m x n integer matrix matrix, if an element is 0, set its entire row and column to 0's, and return the matrix.

You must do it in place.

Example 1:

Input: matrix = [[1,1,1],[1,0,1],[1,1,1]]
Output: [[1,0,1],[0,0,0],[1,0,1]]
Example 2:

Input: matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]
Output: [[0,0,0,0],[0,4,5,0],[0,3,1,0]]

Constraints:

m == matrix.length
n == matrix[0].length
1 <= m, n <= 200
-231 <= matrix[i][j] <= 231 - 1

```
class Solution {
	public void setZeroes(int[][] matrix) {

		for(int i=0; i<matrix.length; i++){
			for(int j=0; j<matrix[i].length; j++){
				if(matrix[i][j]==0)
					inorderReplace(matrix, i, j);
			}
		}

		for(int i=0; i<matrix.length; i++){
			for(int j=0; j<matrix[i].length; j++){
				if(matrix[i][j]==-99)
					matrix[i][j]=0;
			}
		}
	}

	public void inorderReplace(int[][] matrix, int row, int col){
		for(int i=0; i<matrix.length; i++)
			matrix[i][col] = matrix[i][col]==0? 0 : -99;
		for(int j=0; j<matrix[0].length; j++)
			matrix[row][j] = matrix[row][j]==0? 0 : -99;
	}
}
```

----------Search a 2D Matrix
----------Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:

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

Example 1:

Input: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3
Output: true
Example 2:

Input: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 13
Output: false

Constraints:

m == matrix.length
n == matrix[i].length
1 <= m, n <= 100
-104 <= matrix[i][j], target <= 104

```
class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {


        int idx=0;
        for(int i=0;i<matrix.length;i++){
            if(target<=matrix[i][matrix[0].length-1]){
                idx=i;
                break;
            }
        }

        for(int i=matrix[0].length-1;i>=0;i--){
            if(target==matrix[idx][i]){
                return true;
            }
        }
        return false;
    }
}
```

----------Sort Colors
----------Given an array nums with n objects colored red, white, or blue, sort them in-place so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectively.

You must solve this problem without using the library's sort function.

Example 1:

Input: nums = [2,0,2,1,1,0]
Output: [0,0,1,1,2,2]
Example 2:

Input: nums = [2,0,1]
Output: [0,1,2]
Example 3:

Input: nums = [0]
Output: [0]
Example 4:

Input: nums = [1]
Output: [1]

Constraints:

n == nums.length
1 <= n <= 300
nums[i] is 0, 1, or 2

```
class Solution {
    public void sortColors(int[] a) {
        int first=0,second=0,third=0;
      for(int i:a){
          if(i==0)
          first++;
          if(i==1)
          second++;
          if(i==2)
          third++;
      }
       for(int i=0;i<first;i++)
           a[i]=0;
        for(int i=first;i<(first+second);i++)
           a[i]=1;
        for(int i=(first+second);i<a.length;i++)
           a[i]=2;
    }
}
```

----------Minimum Window Substring
----------Given two strings s and t of lengths m and n respectively, return the minimum window substring of s such that every character in t (including duplicates) is included in the window. If there is no such substring, return the empty string "".

The testcases will be generated such that the answer is unique.

A substring is a contiguous sequence of characters within the string.

Example 1:

Input: s = "ADOBECODEBANC", t = "ABC"
Output: "BANC"
Explanation: The minimum window substring "BANC" includes 'A', 'B', and 'C' from string t.
Example 2:

Input: s = "a", t = "a"
Output: "a"
Explanation: The entire string s is the minimum window.
Example 3:

Input: s = "a", t = "aa"
Output: ""
Explanation: Both 'a's from t must be included in the window.
Since the largest window of s only has one 'a', return empty string.

Constraints:

m == s.length
n == t.length
1 <= m, n <= 105
s and t consist of uppercase and lowercase English letters.

```
class Solution {
    public String minWindow(String s, String t) {

        int sLen = s.length(), tLen = t.length();

        if (tLen > sLen) return "";

        if (s.equals(t)) return t;

        int[] Tcount = new int[256];
        int[] Scount = new int[256];
        for (int i = 0; i < t.length(); i++) {

            ++Tcount[t.charAt(i)];
        }

        int windowStart = 0;

        int min = Integer.MAX_VALUE, tCount = 0;
        int start = -1, end = -1;

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

            char ch = s.charAt(windowEnd);

            ++Scount[ch];

            if (Scount[ch] <= Tcount[ch] && Scount[ch] > 0) {

                ++tCount;
            }
```

```
            if (tCount == tLen) {
                --tCount;

                while ((windowEnd-windowStart+1) >= tLen) {

                    char stCh = s.charAt(windowStart);

                    if (min > (windowEnd-windowStart+1)) {

                        min = windowEnd-windowStart+1;
                        start = windowStart;
                        end = windowEnd;
                    }

                    ++windowStart;

                    if (Tcount[stCh] > 0 &&
                        (--Scount[stCh] == 0 || Scount[stCh] < Tcount[stCh])) {

                        break;
                    }
                }
            }
        }

        return start == -1 ? "" : s.substring(start, end+1);
    }
}
```

----------Combinations

----------Given two integers n and k, return all possible combinations of k numbers out of the range [1, n].

You may return the answer in any order.

Example 1:

```
Input: n = 4, k = 2
Output:
[
  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
```

Example 2:

```
Input: n = 1, k = 1
Output: [[1]]
```

Constraints:

1 <= n <= 20
1 <= k <= n

```
public class Solution {
    // II). Recursion -- C(n,k)=C(n-1,k-1)+C(n-1,k)
    public List<List<Integer>> combine(int n, int k) {
   List<List<Integer>> result = new ArrayList<>();
                    if(n < 1 || k < 1 || k > n) {
                            return result;
                    }
                    List<Integer> curPath = new ArrayList<>();
                    dfs(n, k, curPath, 1, result);
                    return result;
            }

            public static void dfs(int n, int k, List<Integer> curPath, int start,
List<List<Integer>> result) {
                    if(k == curPath.size()) {
                            result.add(new ArrayList<Integer>(curPath));
                            return;
                    }
                    if(n < 0) {
                            return;
                    }

                    for(int i = start; i <= n; ++i) {
                            curPath.add(i);
                            dfs(n, k, curPath, i+1, result);
                            curPath.remove(curPath.size() - 1);
                    }
}
}
```

----------Subsets
----------Given an integer array nums of unique elements, return all possible subsets (the power set).

The solution set must not contain duplicate subsets. Return the solution in any order.

Example 1:

Input: nums = [1,2,3]
Output: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]
Example 2:

Input: nums = [0]
Output: [[],[0]]

Constraints:

1 <= nums.length <= 10
-10 <= nums[i] <= 10
All the numbers of nums are unique.

```
class Solution {
    List<List<Integer>> lists;
    public List<List<Integer>> subsets(int[] nums) {
        lists = new ArrayList<>();
        List<Integer> list = new ArrayList<>();
        solve(list,nums,0);

        return lists;
    }

    void solve(List<Integer> list,int[] nums,int curr)
    {

        if(curr == nums.length)
        {

            lists.add(new ArrayList(list));
            //System.out.println(true + " --> "+ lists.toString());
            return;
        }

        List<Integer> list2 = new ArrayList(list);
        list.add(nums[curr]);
        //System.out.println("nums--" +nums[curr] + " list 1--> " + list.toString() + " //list 2--> " +
list2.toString() +" main --> " + lists.toString());
        solve(list2,nums,curr+1);
        solve(list,nums,curr+1);

    }

}
```

----------Word Search
----------Given an m x n grid of characters board and a string word, return true if word exists in the grid.

The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

Example 1:

Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"
Output: true
Example 2:

Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "SEE"
Output: true
Example 3:

Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCB"
Output: false

Constraints:

m == board.length
n = board[i].length
1 <= m, n <= 6
1 <= word.length <= 15
board and word consists of only lowercase and uppercase English letters.

```
class Solution {
    public boolean exist(char[][] board, String word) {
        if(board == null || board.length == 0 || board[0] == null || board[0].length == 0 || word == null
|| word.length() == 0)
            return false;

        boolean[][] isVisited = new boolean[board.length][board[0].length];
        for(int i = 0; i < board.length; i++){
            for(int j = 0; j < board[0].length; j++){
                if(dfs(board, word, i, j, 0, isVisited))
                    return true;
            }
        }
        return false;
    }
    private boolean dfs(char[][] board, String word, int i, int j, int index, boolean[][] isVisited){
        int row = board.length;
        int col = board[0].length;

        // return case
        if(word.length() == index) return true;
        // fail case
        if(i < 0 || i >= row || j < 0 || j >= col || board[i][j] != word.charAt(index) || isVisited[i][j])
            return false;
        // do dfs
        isVisited[i][j] = true;
        boolean res = dfs(board, word, i+1, j, index+1, isVisited) || dfs(board, word, i-1, j, index+1,
isVisited) || dfs(board, word, i, j+1, index+1, isVisited) || dfs(board, word, i, j-1, index+1,
isVisited);
        isVisited[i][j] = false;

        return res;
    }
}
```

----------Remove Duplicates from Sorted Array II

----------Given an integer array nums sorted in non-decreasing order, remove some duplicates in-place such that each unique element appears at most twice. The relative order of the elements should be kept the same.

Since it is impossible to change the length of the array in some languages, you must instead have the result be placed in the first part of the array nums. More formally, if there are k elements after removing the duplicates, then the first k elements of nums should hold the final result. It does not matter what you leave beyond the first k elements.

Return k after placing the final result in the first k slots of nums.

Do not allocate extra space for another array. You must do this by modifying the input array in-place with O(1) extra memory.

Custom Judge:

The judge will test your solution with the following code:

```
int[] nums = [...]; // Input array
int[] expectedNums = [...]; // The expected answer with correct length

int k = removeDuplicates(nums); // Calls your implementation

assert k == expectedNums.length;
for (int i = 0; i < k; i++) {
    assert nums[i] == expectedNums[i];
}
```

If all assertions pass, then your solution will be accepted.

Example 1:

```
Input: nums = [1,1,1,2,2,3]
Output: 5, nums = [1,1,2,2,3,_]
Explanation: Your function should return k = 5, with the first five elements of nums being 1, 1, 2, 2 and 3 respectively.
It does not matter what you leave beyond the returned k (hence they are underscores).
```

Example 2:

```
Input: nums = [0,0,1,1,1,1,2,3,3]
Output: 7, nums = [0,0,1,1,2,3,3,_,_]
Explanation: Your function should return k = 7, with the first seven elements of nums being 0, 0, 1, 1, 2, 3 and 3 respectively.
It does not matter what you leave beyond the returned k (hence they are underscores).
```

Constraints:

```
1 <= nums.length <= 3 * 104
-104 <= nums[i] <= 104
nums is sorted in non-decreasing order.
```

```
class Solution {
```

```
    public int removeDuplicates(int[] nums) {
        if (nums.length < 3)
            return nums.length;

        int count = 1;
        boolean isSame = (nums[count] == nums[count-1]);

        for(int i = 2; i < nums.length; i++){
                                // Reason below
            if (!isSame || nums[i] != nums[count]) {
                nums[++count] = nums[i];
                isSame = nums[count] == nums[count-1];
            }
        }
        return ++count;
    }
}
```

----------Search in Rotated Sorted Array II
----------There is an integer array nums sorted in non-decreasing order (not necessarily with distinct values).

Before being passed to your function, nums is rotated at an unknown pivot index k (0 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]] (0-indexed). For example, [0,1,2,4,4,4,5,6,6,7] might be rotated at pivot index 5 and become [4,5,6,6,7,0,1,2,4,4].

Given the array nums after the rotation and an integer target, return true if target is in nums, or false if it is not in nums.

You must decrease the overall operation steps as much as possible.

Example 1:

Input: nums = [2,5,6,0,0,1,2], target = 0
Output: true
Example 2:

Input: nums = [2,5,6,0,0,1,2], target = 3
Output: false

Constraints:

1 <= nums.length <= 5000
-104 <= nums[i] <= 104
nums is guaranteed to be rotated at some pivot.
-104 <= target <= 104

Follow up: This problem is similar to Search in Rotated Sorted Array, but nums may contain duplicates.
Would this affect the runtime complexity? How and why?

```
class Solution {
    public boolean search(int[] nums, int target) {

                //if array length is greater than zero will enter into the loop
                if(nums.length != 0){

                        //if target is lesser than first value will traverse from front
                        //else traverse from reverse
                        if(nums[0] <= target){

                                //loop from 0 to length of the array
                                for(int i =0; i< nums.length; i++ ){

                                        //if target matches return true,
                                        //else if current value is greater than target break the loop
                                        if(nums[i] ==  target)
                                                return true;
                                        else if(nums[i] > target)
                                                break;
                                }
                        }else{

                                //loop from length of the array to 0
                                for(int i = nums.length-1; i >= 0 ; i--){

                                        //if target matches return true,
                                        //else if current value is lesser than target break the loop
                                        if(nums[i] ==  target)
                                                return true;
                                        else if(nums[i] < target)
                                                break;
                                }
                        }
                }
                return false;
        }
}
```

----------Remove Duplicates from Sorted List II ----------Given the head of a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list. Return the linked list sorted as well.

Example 1:

Input: head = [1,2,3,3,4,4,5]
Output: [1,2,5]
Example 2:

Input: head = [1,1,1,2,3]

Output: [2,3]

Constraints:

The number of nodes in the list is in the range [0, 300].
-100 <= Node.val <= 100
The list is guaranteed to be sorted in ascending order.

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        ListNode sentinel = new ListNode(Integer.MAX_VALUE, head);
        ListNode current = sentinel;
        while (current != null) {
            ListNode next = current.next;
            if (next == null || next.next == null || next.val != next.next.val) {
                current = current.next;
            } else {
                int val = next.val;
                while (current.next != null && current.next.val == val) {
                    current.next = current.next.next;
                }
            }
        }
        return sentinel.next;
    }
}
```

----------Remove Duplicates from Sorted List
----------Given the head of a sorted linked list, delete all duplicates such that each element appears only once. Return the linked list sorted as well.

Example 1:

Input: head = [1,1,2]
Output: [1,2]

Input: head = [1,1,2,3,3]
Output: [1,2,3]

Constraints:

The number of nodes in the list is in the range [0, 300].

-100 <= Node.val <= 100
The list is guaranteed to be sorted in ascending order.

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
    ListNode curr = head;
    if(head == null) return null;

    // if the head.next == null it means that the list contains only one node (head)
    //so there is no duplicates  we will simply return the head of the list

    if(head.next == null) return head;
    else{
        while(curr.next != null){
        //we compare the data of the current node with the next node if they are equal
        //we remove the next node
            if(curr.val == curr.next.val)
                curr.next = curr.next.next;
            else
            //if they are not equal we simply move forward
                curr=curr.next;
        }
    }
    return head;
}
}
```

----------Largest Rectangle in Histogram
----------Given an array of integers heights representing the histogram's bar height where the width of each bar is 1, return the area of the largest rectangle in the histogram.

Example 1:

Input: heights = [2,1,5,6,2,3]
Output: 10
Explanation: The above is a histogram where width of each bar is 1.
The largest rectangle is shown in the red area, which has an area = 10 units.
Example 2:

Input: heights = [2,4]
Output: 4

Constraints:

1 <= heights.length <= 105
0 <= heights[i] <= 104

```
public class Solution {
    public int largestRectangleArea(int[] heights) {
        if (heights == null || heights.length == 0) return 0;
        return getMax(heights, 0, heights.length);
    }
    int getMax(int[] heights, int s, int e) {
        if (s + 1 >= e) return heights[s];
        int min = s;
        boolean sorted = true;
        for (int i = s; i < e; i++) {
            if (i > s && heights[i] < heights[i - 1]) sorted = false;
            if (heights[min] > heights[i]) min = i;
        }
        if (sorted) {
            int max = 0;
            for (int i = s; i < e; i++) {
                max = Math.max(max, heights[i] * (e - i));
            }
            return max;
        }
        int left = (min > s) ? getMax(heights, s, min) : 0;
        int right = (min < e - 1) ? getMax(heights, min + 1, e) : 0;
        return Math.max(Math.max(left, right), (e - s) * heights[min]);
    }
}
```

----------Maximal Rectangle
----------Given a rows x cols binary matrix filled with 0's and 1's, find the largest rectangle containing only 1's and return its area.

Example 1:

Input: matrix = [["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],
["1","0","0","1","0"]]
Output: 6
Explanation: The maximal rectangle is shown in the above picture.
Example 2:

Input: matrix = []
Output: 0
Example 3:

Input: matrix = [["0"]]
Output: 0
Example 4:

Input: matrix = [["1"]]
Output: 1
Example 5:

Input: matrix = [["0","0"]]
Output: 0

Constraints:

rows == matrix.length
cols == matrix[i].length
0 <= row, cols <= 200
matrix[i][j] is '0' or '1'.


```
class Solution {
public int maximalRectangle(char[][] matrix) {
        /**
         * idea: using [LC84 Largest Rectangle in Histogram]. For each row
         * of the matrix, construct the histogram based on the current row
         * and the previous histogram (up to the previous row), then compute
         * the largest rectangle area using LC84.
         */
        int m = matrix.length, n;
        if (m == 0 || (n = matrix[0].length) == 0)
            return 0;

        int i, j, res = 0;
        int[] heights = new int[n];
        for (i = 0; i < m; i++) {
            for (j = 0; j < n; j++) {
                if (matrix[i][j] == '0')
                    heights[j] = 0;
                else
                    heights[j] += 1;
            }
            res = Math.max(res, largestRectangleArea(heights));
        }

        return res;
    }

    public int largestRectangleArea(int[] heights) {
        /**
         * idea: scan and store if a[i-1]<=a[i] (increasing), then as long
         * as a[i]<a[i-1], then we can compute the largest rectangle area
         * with base a[j], for j<=i-1, and a[j]>a[i], which is a[j]*(i-j).
         * And meanwhile, all these bars (a[j]'s) are already done, and thus
         * are throwable (using pop() with a stack).
```

```
         *
         * We can use an array nLeftGeq[] of size n to simulate a stack.
         * nLeftGeq[i] = the number of elements to the left of [i] having
         * value greater than or equal to a[i] (including a[i] itself). It
         * is also the index difference between [i] and the next index on
         * the top of the stack.
         */
        int n = heights.length;
        if (n == 0)
            return 0;

        int[] nLeftGeq = new int[n]; // the number of elements to the left
                                     // of [i] with value >= heights[i]
        nLeftGeq[0] = 1;

        // preIdx=the index of stack.peek(), res=max area so far
        int preIdx = 0, res = 0;

        for (int i = 1; i < n; i++) {
            nLeftGeq[i] = 1;

            // notice that preIdx = i - 1 = peek()
            while (preIdx >= 0 && heights[i] < heights[preIdx]) {
                res = Math.max(res, heights[preIdx] * (nLeftGeq[preIdx] + i - preIdx - 1));
                nLeftGeq[i] += nLeftGeq[preIdx]; // pop()

                preIdx = preIdx - nLeftGeq[preIdx]; // peek() current top
            }

            if (preIdx >= 0 && heights[i] == heights[preIdx])
                nLeftGeq[i] += nLeftGeq[preIdx]; // pop()
            // otherwise nothing to do

            preIdx = i;
        }

        // compute the rest largest rectangle areas with (indices of) bases
        // on stack
        while (preIdx >= 0 && 0 < heights[preIdx]) {
            res = Math.max(res, heights[preIdx] * (nLeftGeq[preIdx] + n - preIdx - 1));
            preIdx = preIdx - nLeftGeq[preIdx]; // peek() current top
        }

        return res;
    }
}
```

----------Partition List
----------Given the head of 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.

Example 1:

Input: head = [1,4,3,2,5,2], x = 3
Output: [1,2,2,4,3,5]
Example 2:

Input: head = [2,1], x = 2
Output: [1,2]

Constraints:

The number of nodes in the list is in the range [0, 200].
-100 <= Node.val <= 100
-200 <= x <= 200

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

        ListNode headF = new ListNode(-1);
        ListNode headS = new ListNode(-1);

        ListNode dummyF = headF, dummyS = headS;

        while( head != null ) {

            if( head.val < x) {

                headF.next = head;
                headF = headF.next;
                head = head.next;
                headF.next = null;
            }
            else{
                headS.next = head;
                headS = headS.next;
                head = head.next;
                headS.next = null;
            }


        }
```

```
        headF.next = dummyS.next;

        return dummyF.next;

    }
}
```
----------Scramble String
----------We can scramble a string s to get a string t using the following algorithm:

If the length of the string is 1, stop.
If the length of the string is > 1, do the following:
Split the string into two non-empty substrings at a random index, i.e., if the string is s, divide it to x and y where s = x + y.
Randomly decide to swap the two substrings or to keep them in the same order. i.e., after this step, s may become s = x + y or s = y + x.
Apply step 1 recursively on each of the two substrings x and y.
Given two strings s1 and s2 of the same length, return true if s2 is a scrambled string of s1, otherwise, return false.

Example 1:

Input: s1 = "great", s2 = "rgeat"
Output: true
Explanation: One possible scenario applied on s1 is:
"great" --> "gr/eat" // divide at random index.
"gr/eat" --> "gr/eat" // random decision is not to swap the two substrings and keep them in order.
"gr/eat" --> "g/r / e/at" // apply the same algorithm recursively on both substrings. divide at ranom index each of them.
"g/r / e/at" --> "r/g / e/at" // random decision was to swap the first substring and to keep the second substring in the same order.
"r/g / e/at" --> "r/g / e/ a/t" // again apply the algorithm recursively, divide "at" to "a/t".
"r/g / e/ a/t" --> "r/g / e/ a/t" // random decision is to keep both substrings in the same order.
The algorithm stops now and the result string is "rgeat" which is s2.
As there is one possible scenario that led s1 to be scrambled to s2, we return true.
Example 2:

Input: s1 = "abcde", s2 = "caebd"
Output: false
Example 3:

Input: s1 = "a", s2 = "a"
Output: true

Constraints:

s1.length == s2.length
1 <= s1.length <= 30
s1 and s2 consist of lower-case English letters.

----------Merge Sorted Array
----------Given two sorted integer arrays nums1 and nums2, merge nums2 into nums1 as one sorted array.
Note:
You may assume that nums1 has enough space (size that is greater or equal to m + n) to hold additional elements from nums2. The number of elements initialized in nums1 and nums2 are m and n respectively.
----------Gray Code
----------The gray code is a binary numeral system where two successive values
differ in only one bit.
Given a non-negative integer n representing the total number of bits in the code, print the sequence of gray code. A gray code sequence must begin with 0.
For example, given n = 2, return [0,1,3,2]. Its gray code sequence is:
00
01
11
10

-

0
1
3
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.

```
class Solution {
    Map<String, Boolean> mem = new HashMap();
    StringBuilder sb = new StringBuilder();
    public boolean isScramble(String s1, String s2) {
        int m = s1.length();
        int n = s2.length();
        if (m != n) return false;
        String key = getKey(s1, s2);
        if (mem.containsKey(key)) return mem.get(key);
        if (m == 1 && s1.equals(s2))
            return true;
        char[] arr1 = s1.toCharArray();
        char[] arr2 = s2.toCharArray();
        Arrays.sort(arr1);
        Arrays.sort(arr2);
        if (!String.valueOf(arr1).equals(String.valueOf(arr2))) {
            mem.put(key, false);
            return false;
        }
```

```
        for (int i = 0; i < n; i++) {
            String left1 = s1.substring(0, i);
            String right1 = s1.substring(i);
            String left2 = s2.substring(0, i);
            String right2 = s2.substring(i);
            if (isScramble(left1, left2) && isScramble(right1, right2)) {
                mem.put(key, true);
                return true;
            }
            if (isScramble(left1, s2.substring(n-i)) && isScramble(right1, s2.substring(0, n-i))) {
                mem.put(key, true);
                return true;
            }
        }
        mem.put(key, false);
        return false;
    }

    private String getKey(String s1, String s2) {
        sb.setLength(0);
        sb.append(s1);
        sb.append(s2);
        return sb.toString();
    }
}
```

----------Subsets II

----------Given an integer array nums that may contain duplicates, return all possible subsets (the power set).

The solution set must not contain duplicate subsets. Return the solution in any order.

Example 1:

```
Input: nums = [1,2,2]
Output: [[],[1],[1,2],[1,2,2],[2],[2,2]]
```

Example 2:

```
Input: nums = [0]
Output: [[],[0]]
```

Constraints:

```
1 <= nums.length <= 10
-10 <= nums[i] <= 10
```

```
class Solution {
    public List<List<Integer>> subsetsWithDup(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        if(nums==null || nums.length == 0) return result;
```

```
        Arrays.sort(nums);

        solve(nums, result, new ArrayList<>(), 0);
        return result;
    }

    void solve(int[] nums, List<List<Integer>> result, List<Integer> current, int index) {
        result.add(new ArrayList<>(current));
        for(int i = index; i<nums.length; i++) {
            if(i>index && nums[i]==nums[i-1]) continue;
        current.add(nums[i]);
        solve(nums, result, current, i+1);
        current.remove(current.size()-1);
        }
    }
}
```

----------Decode Ways

----------A message containing letters from A-Z can be encoded into numbers using the following mapping:

'A' -> "1"
'B' -> "2"
...
'Z' -> "26"
To decode an encoded message, all the digits must be grouped then mapped back into letters using the reverse of the mapping above (there may be multiple ways). For example, "11106" can be mapped into:

"AAJF" with the grouping (1 1 10 6)
"KJF" with the grouping (11 10 6)
Note that the grouping (1 11 06) is invalid because "06" cannot be mapped into 'F' since "6" is different from "06".

Given a string s containing only digits, return the number of ways to decode it.

The answer is guaranteed to fit in a 32-bit integer.

Example 1:

Input: s = "12"
Output: 2
Explanation: "12" could be decoded as "AB" (1 2) or "L" (12).
Example 2:

Input: s = "226"
Output: 3
Explanation: "226" could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2 6).
Example 3:

Input: s = "0"
Output: 0

Explanation: There is no character that is mapped to a number starting with 0.
The only valid mappings with 0 are 'J' -> "10" and 'T' -> "20", neither of which start with 0.
Hence, there are no valid ways to decode this since all digits need to be mapped.
Example 4:

Input: s = "06"
Output: 0
Explanation: "06" cannot be mapped to "F" because of the leading zero ("6" is different from "06").

Constraints:

1 <= s.length <= 100
s contains only digits and may contain leading zero(s).

```
class Solution {
    public int numDecodings(String s) {
        int len=s.length();
        if(len==1){
            //Handling string = '0' case
            if(s.charAt(0)=='0') return 0;
            return 1;
        }
        //before we begin the for loop, we have to hardcode the last 2 values
        int[] arr = new int[len];
        arr[len-1]= (s.charAt(len-1)!='0'?1:0);
        //if the last char is 0, then it's only possible to make a valid string if the last-1 number is 1
or 2. Any other case, the string is invalid and we return 0;
        if(arr[len-1]==0)
        {
            if(s.charAt(len-2)-'0'==0 || s.charAt(len-2)-'0'>=3) return 0;
            arr[len-2] = 1;
        }else{
            //If the last-1 char is 0, then we have to check the last-2 char to see if this 0 can be
used to make a valid string. Eg. 10 or 20 are the only valid possibilities
            if(s.charAt(len-2)-'0'==0) arr[len-2]=0;
            else{
                int val = (s.charAt(len-2)-'0')*10+s.charAt(len-1)-'0';
                if(val<=26) arr[len-2]=2;
                else arr[len-2] = 1;
            }
        }
        for(int i=len-3;i>=0;i--){
            //for 0, it's only possible to create a valid string if the preceeding char is 1 or 2.
            if(s.charAt(i)=='0') {
                arr[i] = 0;
                continue;
            }
            //Everytime valid string can be formed either considering ith char as a single char or by
creating a char from ith and (i+1)th char
            arr[i]=arr[i+1];
            int val = (s.charAt(i)-'0')*10+s.charAt(i+1)-'0';
            if(val<=26) arr[i] += arr[i+2];
```

```
        }
        return arr[0];
    }
}
```

----------Reverse Linked List II----------

Given the head of a singly linked list and two integers left and right where left <= right, reverse the nodes of the list from position left to position right, and return the reversed list.

Example 1:

Input: head = [1,2,3,4,5], left = 2, right = 4
Output: [1,4,3,2,5]

Example 2:

Input: head = [5], left = 1, right = 1
Output: [5]

Constraints:

The number of nodes in the list is n.
1 <= n <= 500
-500 <= Node.val <= 500
1 <= left <= right <= n

Follow up: Could you do it in one pass?

```
class Solution {
public ListNode reverseBetween(ListNode head, int left, int right) {
    ListNode prev=null,temp=head;

    int i=1;
    for(i=1;i<left;i++)
    {
        prev=temp;
        temp=temp.next;
    }

    ListNode after=temp,before=prev;

    while(i<=right && temp!=null)
    {

        ListNode check=temp.next;
        temp.next=prev;
        prev=temp;
        temp=check;
        i++;
```

```
    }

    if(before!=null)
        before.next=prev;

    else
        head=prev;

    after.next=temp;

    return head;
}
}
```

----------Restore IP Addresses
----------A valid IP address consists of exactly four integers separated by single dots. Each integer is between 0 and 255 (inclusive) and cannot have leading zeros.

For example, "0.1.2.201" and "192.168.1.1" are valid IP addresses, but "0.011.255.245", "192.168.1.312" and "192.168@1.1" are invalid IP addresses.
Given a string s containing only digits, return all possible valid IP addresses that can be formed by inserting dots into s. You are not allowed to reorder or remove any digits in s. You may return the valid IP addresses in any order.

Example 1:

Input: s = "25525511135"
Output: ["255.255.11.135","255.255.111.35"]
Example 2:

Input: s = "0000"
Output: ["0.0.0.0"]
Example 3:

Input: s = "1111"
Output: ["1.1.1.1"]
Example 4:

Input: s = "010010"
Output: ["0.10.0.10","0.100.1.0"]
Example 5:

Input: s = "101023"
Output: ["1.0.10.23","1.0.102.3","10.1.0.23","10.10.2.3","101.0.2.3"]

Constraints:

0 <= s.length <= 20
s consists of digits only.

```
class Solution {
    void recurse(List<String> res, String s, StringBuilder buffer, int pos, int value, int dotsSoFar) {
        if (pos == s.length()) {
            //String length should be exactly s.length() + 3 (for each dot).
            if (buffer.length() == s.length() + 3)
                res.add(buffer.toString());
            return;
        }

        char c = s.charAt(pos);
        int num = c - '0';

        //Try to place a dot, then write the current digit.
        /*
            Explanation of the conditions:
            dotsSoFar < 3: since there cannot be more than 3 dots in any valid string.
            buffer.length() > 0: to prevent a dot from being placed at beginning of the string.
            buffer.charAt(buffer.length() - 1) != '.': to prevent consecutive dots.
        */
        if (dotsSoFar < 3 && buffer.length() > 0 && buffer.charAt(buffer.length() - 1) != '.') {
            buffer.append('.');
            buffer.append(c);
            recurse(res, s, buffer, pos + 1, num, dotsSoFar + 1);
            buffer.delete(buffer.length() - 2, buffer.length());
        }

        //Try to write the current digit, without a dot.

        /*
            Explanation of the conditions:
            10*value + num < 256: if the octet so far (including this digit) is less than 256.
            (buffer.length() == 0 || value > 0 || buffer.charAt(buffer.length() - 1) != '0'): this part
basically checks if we are placing a 0 next to a leading 0.

        */
        if (10*value + num < 256 && (buffer.length() == 0 || value > 0 || buffer.charAt(buffer.length() -
1) != '0')) {
            buffer.append(c);
            recurse(res, s, buffer, pos + 1, 10*value + num, dotsSoFar);
            buffer.deleteCharAt(buffer.length() - 1);
        }
    }

    public List<String> restoreIpAddresses(String s) {
        List<String> res = new ArrayList<>();
        recurse(res, s, new StringBuilder(s.length() + 3), 0, 0, 0);
        return res;
    }
}

------------------------------------------------------------Binary Tree Inorder Traversal
----------------------------Given the root of a binary tree, return the inorder traversal of its
nodes' values.
```

Example 1:

Input: root = [1,null,2,3]
Output: [1,3,2]
Example 2:

Input: root = []
Output: []
Example 3:

Input: root = [1]
Output: [1]
Example 4:

Input: root = [1,2]
Output: [2,1]
Example 5:

Input: root = [1,null,2]
Output: [1,2]

Constraints:

The number of nodes in the tree is in the range [0, 100].
-100 <= Node.val <= 100

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        helper(root, res);
        return res;
    }
```

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

----------Unique Binary Search Trees II

----------Given an integer n, return all the structurally unique BST's (binary search trees), which has exactly n nodes of unique values from 1 to n. Return the answer in any order.

Example 1:

Input: n = 3
Output: [[1,null,2,null,3],[1,null,3,2],[2,1,3],[3,1,null,null,2],[3,2,null,1]]
Example 2:

Input: n = 1
Output: [[1]]

Constraints:

1 <= n <= 8

----------Unique Binary Search Trees

https://leetcode.com/problems/unique-binary-search-trees-ii/

----------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.
Input: n = 3
Output: [[1,null,2,null,3],[1,null,3,2],[2,1,3],[3,1,null,null,2],[3,2,null,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[] recursion(int s, int e, int[] dp){
        TreeNode[] roots = null;
        int curlen = 0;
        if(s > e){
            roots = new TreeNode[1];
            roots[0] = null;
            return roots;
        }
        roots = new TreeNode[dp[e - s + 1]];
        for(int i = s; i <= e; i++){
            TreeNode[] lefts = recursion(s, i - 1, dp);
            TreeNode[] rights = recursion(i + 1, e, dp);
            for(TreeNode left : lefts){
                for(TreeNode right : rights){
                    TreeNode root = new TreeNode(i);
                    root.left = left;
                    root.right = right;
                    roots[curlen++] = root;
                }
            }
        }
        return roots;
    }
    public List<TreeNode> generateTrees(int n) {
        if(n == 0)
            return new ArrayList<TreeNode>();
        else{
            int[] dp = new int[n + 1];
            dp[0] = 1;
            dp[1] = 1;
            for(int i = 2; i <= n; i++)
                for(int j = 1; j <= i; j++)
                    dp[i] += dp[j - 1] * dp[i - j];
            TreeNode[] resarr =  recursion(1, n, dp);
            List<TreeNode> res = new ArrayList<>();
            for(TreeNode node : resarr){
                res.add(node);
            }
            return res;
        }
    }
}
```

----------Interleaving String
----------Given strings s1, s2, and s3, find whether s3 is formed by an interleaving of s1 and s2.

An interleaving of two strings s and t is a configuration where they are divided into non-empty substrings such that:

s = s1 + s2 + ... + sn
t = t1 + t2 + ... + tm
|n - m| <= 1
The interleaving is s1 + t1 + s2 + t2 + s3 + t3 + ... or t1 + s1 + t2 + s2 + t3 + s3 + ...

Note: a + b is the concatenation of strings a and b.

Example 1:

Input: s1 = "aabcc", s2 = "dbbca", s3 = "aadbbcbcac"
Output: true
Example 2:

Input: s1 = "aabcc", s2 = "dbbca", s3 = "aadbbbaccc"
Output: false
Example 3:

Input: s1 = "", s2 = "", s3 = ""
Output: true

Constraints:

0 <= s1.length, s2.length <= 100
0 <= s3.length <= 200
s1, s2, and s3 consist of lowercase English letters.

```
class Solution {

    private boolean dfs(String s1, String s2, String s3, int i, int j, int k, boolean[][] visited){
        if(i == s1.length() && j == s2.length()) return true;
        if(i > s1.length() || j > s2.length()) return false;
        if(visited[i][j]) return false;
        visited[i][j] = true;
        boolean a = false, b = false;
        if(i < s1.length() && s1.charAt(i) == s3.charAt(k)){
            a = dfs(s1, s2, s3, i+1, j, k+1, visited);
        }
        if(j < s2.length() && s2.charAt(j) == s3.charAt(k)){
            b = dfs(s1, s2, s3, i, j+1, k+1, visited);
        }
        return a || b;
    }

    public boolean isInterleave(String s1, String s2, String s3) {
        if(s3.length() < (s1.length() + s2.length()) || s3.length() > (s1.length() + s2.length())) return
false;
        boolean[][] visited = new boolean[s1.length()+1][s2.length()+1];
        return dfs(s1, s2, s3, 0, 0, 0, visited);
    }
}
```

----------Validate Binary Search Tree

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

Assume a BST is defined as follows:
The left subtree of a node contains only nodes with keys less than the node's key.
The right subtree of a node contains only nodes with keys greater than the node's key.
Both the left and right subtrees must also be binary search trees.

Example 1:
2
/ \
1

3

Binary tree [2,1,3], return true.
Example 2:
1
/ \
2

3

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

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    TreeNode prev;
    public boolean isValidBST(TreeNode root) {
        if (root == null)
            return true;
        if(!isValidBST(root.left))
            return false;
        if (prev != null && prev.val >= root.val)
            return false;
        prev = root;
        if (!isValidBST(root.right))
            return false;
```

```
        return true;
    }
}
```
----------Recover Binary Search Tree
----------You are given the root of a binary search tree (BST), where the values of exactly two nodes of the tree were swapped by mistake. Recover the tree without changing its structure.

Example 1:

Input: root = [1,3,null,null,2]
Output: [3,1,null,null,2]
Explanation: 3 cannot be a left child of 1 because 3 > 1. Swapping 1 and 3 makes the BST valid.
Example 2:

Input: root = [3,1,4,null,null,2]
Output: [2,1,4,null,null,3]
Explanation: 2 cannot be in the right subtree of 3 because 2 < 3. Swapping 2 and 3 makes the BST valid.

Constraints:

The number of nodes in the tree is in the range [2, 1000].
-231 <= Node.val <= 231 - 1

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public void recoverTree(TreeNode root) {
        TreeNode x = null, y = null, pred = null, predcessor = null;
        while (root != null){
            if (root.left == null){
                if (pred != null && root.val < pred.val){
                    y = root;
                    if (x == null)
```

```
                    x = pred;
                }
                pred = root;
                root = root.right;
            } else {
                predcessor = root.left;
                while (predcessor.right != null && predcessor.right != root){
                    predcessor = predcessor.right;
                }
                if (predcessor.right == null){
                    predcessor.right = root;
                    root = root.left;
                } else {
                    if (pred != null && root.val < pred.val){
                        y = root;
                        if (x == null)
                            x = pred;
                    }
                    pred = root;
                    root = root.right;
                    predcessor.right = null;
                }
            }
        }
        swap(x, y);
    }

    private void swap(TreeNode node1, TreeNode node2){
        int temp = node1.val;
        node1.val = node2.val;
        node2.val = temp;
    }
}
```

----------Same Tree
----------Given the roots of two binary trees p and q, write a function to check if they are the same or not.

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

Example 1:

Input: p = [1,2,3], q = [1,2,3]
Output: true
Example 2:

Input: p = [1,2], q = [1,null,2]
Output: false
Example 3:

Input: p = [1,2,1], q = [1,1,2]
Output: false

Constraints:

The number of nodes in both trees is in the range [0, 100].
-104 <= Node.val <= 104

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
  public boolean check(TreeNode p, TreeNode q) {
    // p and q are null
    if (p == null && q == null) return true;
    // one of p and q is null
    if (q == null || p == null) return false;
    if (p.val != q.val) return false;
    return true;
  }

  public boolean isSameTree(TreeNode p, TreeNode q) {
    if (p == null && q == null) return true;
    if (!check(p, q)) return false;

    // init deques
    ArrayDeque<TreeNode> deqP = new ArrayDeque<TreeNode>();
    ArrayDeque<TreeNode> deqQ = new ArrayDeque<TreeNode>();
    deqP.addLast(p);
    deqQ.addLast(q);

    while (!deqP.isEmpty()) {
      p = deqP.removeFirst();
      q = deqQ.removeFirst();

      if (!check(p, q)) return false;
      if (p != null) {
        // in Java nulls are not allowed in Deque
        if (!check(p.left, q.left)) return false;
        if (p.left != null) {
```

```
          deqP.addLast(p.left);
          deqQ.addLast(q.left);
        }
        if (!check(p.right, q.right)) return false;
        if (p.right != null) {
          deqP.addLast(p.right);
          deqQ.addLast(q.right);
        }
      }
    }
    return true;
  }
}
```

----------Symmetric Tree
----------Given the root of a binary tree, check whether it is a mirror of itself (i.e., symmetric around its center).

Example 1:

Input: root = [1,2,2,3,4,4,3]
Output: true
Example 2:

Input: root = [1,2,2,null,3,null,3]
Output: false

Constraints:

The number of nodes in the tree is in the range [1, 1000].
-100 <= Node.val <= 100

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

```
class Solution {
    public boolean isSymmetric(TreeNode root) {
        if(root == null)
            return true;
        // traversing left and right subtree
        return multiDfs(root.left, root.right);
    }

    boolean multiDfs(TreeNode left, TreeNode right) {
        if(left == null && right == null){
            return true;
        }
        if(right == null || left == null){
            return false;
        }
        if(left.val != right.val){
            return false;
        }
        return multiDfs(left.left, right.right) &&
        multiDfs(left.right, right.left);
    }
}
```

----------Binary Tree Level Order Traversal----------Given the root of a binary tree, return the level order traversal of its nodes' values. (i.e., from left to right, level by level).

Example 1:

Input: root = [3,9,20,null,null,15,7]
Output: [[3],[9,20],[15,7]]
Example 2:

Input: root = [1]
Output: [[1]]
Example 3:

Input: root = []
Output: []

Constraints:

The number of nodes in the tree is in the range [0, 2000].
-1000 <= Node.val <= 1000

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
```

```
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
   public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> res = new ArrayList();
        solve(root, res, 0);
        return res;
    }

    void solve(TreeNode root, List<List<Integer>> res, int level) {
        if(root == null) return;
        if(level == res.size()) {
            res.add(new ArrayList());
        }
        res.get(level).add(root.val);
        solve(root.left, res, level+1);
        solve(root.right, res, level+1);
    }
}
```

----------Binary Tree Zigzag Level Order Traversal----------Given a binary tree, return the zigzag level order traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).

For example:

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

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

return its zigzag level order traversal as:

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

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

```
*     TreeNode(int val, TreeNode left, TreeNode right) {
*         this.val = val;
*         this.left = left;
*         this.right = right;
*     }
* }
*/
class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        if (root == null)
            return new LinkedList<>();
        List<List<Integer>> res = new ArrayList<>();
        Deque<TreeNode> q = new LinkedList<>();
        q.add(root);
        int level = 0;
        while (!q.isEmpty()){
            int size = q.size();
            LinkedList<Integer> temp = new LinkedList<>();
            for (int i=0; i<size; ++i){
                TreeNode node = q.poll();
                if (level%2 == 0)
                    temp.add(node.val);
                else
                    temp.addFirst(node.val);
                if (node.left != null)
                    q.add(node.left);
                if (node.right != null)
                    q.add(node.right);
            }
            res.add(temp);
            level++;
        }
        return res;
    }
}
```

----------Maximum Depth of Binary Tree
----------Given the root of a binary tree, return its maximum depth.

A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Example 1:

Input: root = [3,9,20,null,null,15,7]
Output: 3
Example 2:

Input: root = [1,null,2]
Output: 2
Example 3:

Input: root = []
Output: 0
Example 4:

Input: root = [0]
Output: 1

Constraints:

The number of nodes in the tree is in the range [0, 104].
-100 <= Node.val <= 100

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

    public int heigth(TreeNode root){
        if(root == null)
            return 0;
        int lp = heigth(root.left);
        int rp = heigth(root.right);

        return 1 + Math.max(lp , rp);
    }



    public int maxDepth(TreeNode root) {
 if(root == null)
     return 0 ;

        return Math.max( heigth(root) ,Math.max(heigth(root.left) ,heigth(root.right)));

    }
}
```

----------Construct Binary Tree from Preorder and Inorder Traversal
----------Given preorder and inorder traversal of a tree, construct the binary

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

Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
Output: [3,9,20,null,null,15,7]
Example 2:

Input: preorder = [-1], inorder = [-1]
Output: [-1]

Constraints:

1 <= preorder.length <= 3000
inorder.length == preorder.length
-3000 <= preorder[i], inorder[i] <= 3000
preorder and inorder consist of unique values.
Each value of inorder also appears in preorder.
preorder is guaranteed to be the preorder traversal of the tree.
inorder is guaranteed to be the inorder traversal of the tree.

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    int preorderIndex;
    Map<Integer, Integer> inorderIndexMap;
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        preorderIndex = 0;
        // build a hashmap to store value -> its index relations
        inorderIndexMap = new HashMap<>();
        for (int i = 0; i < inorder.length; i++) {
            inorderIndexMap.put(inorder[i], i);
        }

        return arrayToTree(preorder, 0, preorder.length - 1);
    }

    private TreeNode arrayToTree(int[] preorder, int left, int right) {
        // if there are no elements to construct the tree
        if (left > right) return null;
```

```
        // select the preorder_index element as the root and increment it
        int rootValue = preorder[preorderIndex++];
        TreeNode root = new TreeNode(rootValue);

        // build left and right subtree
        // excluding inorderIndexMap[rootValue] element because it's the root
        root.left = arrayToTree(preorder, left, inorderIndexMap.get(rootValue) - 1);
        root.right = arrayToTree(preorder, inorderIndexMap.get(rootValue) + 1, right);
        return root;
    }
}
```

----------Construct Binary Tree from Inorder and Postorder Traversal
----------Given inorder and postorder traversal of a tree, construct the binary tree.
Note:
You may assume that duplicates do not exist in the tree.
Input: inorder = [9,3,15,20,7], postorder = [9,15,7,20,3]
Output: [3,9,20,null,null,15,7]
Example 2:

Input: inorder = [-1], postorder = [-1]
Output: [-1]

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public TreeNode buildTree(int[] inorder, int[] postorder) {
        if (inorder.length == 0) return null;
        TreeNode root = null;
        // index of first root in every level in inorder
        int i =  0;
        // index of the right node of the first root
        int j = 0;
        while (i < inorder.length) {
            final int nextRootVal = inorder[i];
            // Figure out the end of right tree node in postorder
            int postorderEndForJ = j;
```

```
            // This will always be valid index cuz there is root ahead
            while (postorder[postorderEndForJ] != nextRootVal) postorderEndForJ++;
            final int[] postOrderForJ = Arrays.copyOfRange(postorder, j, postorderEndForJ);
            // Find inorder end for right tree;
            int inorderEndForJ = i + 1;
            // Same here cuz we must have all the rights
            for (int k = j; k < postorderEndForJ; k++) inorderEndForJ++;
            final int[] inOrderForJ = Arrays.copyOfRange(inorder, i + 1, inorderEndForJ);
            root = new TreeNode(inorder[i], root, buildTree(inOrderForJ, postOrderForJ));

            i = inorderEndForJ;
            j = postorderEndForJ + 1;
        }
        return root;
    }
}
```

----------Binary Tree Level Order Traversal II
----------Given a binary tree, return the bottom-up level order traversal of its nodes' values. (ie, from left to right, level by level from leaf to root).
For example:
Given binary tree [3,9,20,null,null,15,7],
3
/ \
9 20
/ \
15
7

return its bottom-up level order traversal as:
[
[15,7],
[9,20],
[3]
]

Input: root = [3,9,20,null,null,15,7]
Output: [[15,7],[9,20],[3]]
Example 2:

Input: root = [1]
Output: [[1]]
Example 3:

Input: root = []
Output: []

Constraints:

The number of nodes in the tree is in the range [0, 2000].
-1000 <= Node.val <= 1000

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

public List<List<Integer>> levelOrderBottom(TreeNode root) {
        List<List<Integer>> result = new ArrayList<>();
        solve(root, 0, result);
        return result;

    }
    void solve(TreeNode root, int level, List<List<Integer>> result) {
        if(root == null) {
            return;
        }
        if(level == result.size()) {
            result.add(0, new ArrayList());
        }
        result.get(result.size() - level - 1).add(root.val);

        solve(root.left, level + 1, result);
        solve(root.right, level + 1, result);
    }
}
```

----------Convert Sorted Array to Binary Search Tree----------Given an integer array nums where the elements are sorted in ascending order, convert it to a height-balanced binary search tree.

A height-balanced binary tree is a binary tree in which the depth of the two subtrees of every node never differs by more than one.

Example 1:

Input: nums = [-10,-3,0,5,9]
Output: [0,-3,9,-10,null,5]
Explanation: [0,-10,5,null,-3,null,9] is also accepted:

Example 2:

Input: nums = [1,3]
Output: [3,1]
Explanation: [1,3] and [3,1] are both a height-balanced BSTs.

Constraints:

1 <= nums.length <= 104
-104 <= nums[i] <= 104
nums is sorted in a strictly increasing order.

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
   public TreeNode sortedArrayToBST(int[] nums) {
if(nums.length == 0){
return null;
}
return helper(nums,0,nums.length - 1);
}

public TreeNode  helper(int num[],int min,int max){
    if(min > max){
        return null;
    }
    int mid = (min+max+1)/2;
    TreeNode node = new TreeNode (num[mid]);
    node.left = helper(num,min,mid-1);
    node.right = helper(num,mid+1,max);
    return node;
}
}
```

----------Convert Sorted List to Binary Search Tree----------Given the head of a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

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.

Example 1:

Input: head = [-10,-3,0,5,9]
Output: [0,-3,9,-10,null,5]
Explanation: One possible answer is [0,-3,9,-10,null,5], which represents the shown height balanced BST.
Example 2:

Input: head = []
Output: []
Example 3:

Input: head = [0]
Output: [0]
Example 4:

Input: head = [1,3]
Output: [3,1]

Constraints:

The number of nodes in head is in the range [0, 2 * 104].
-105 <= Node.val <= 105

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
```

```
    public TreeNode sortedListToBST(ListNode head) {
        if(head==null)
            return null;
        return bstTree(head,null);
    }

            public TreeNode bstTree(ListNode head,ListNode tail)
            {
                ListNode fast=head;
                ListNode slow=head;

                if(head==tail)
                    return null;
                while(fast!=tail && fast.next!=tail)
                {
                    fast=fast.next.next;
                    slow=slow.next;
                }
                TreeNode thead=new TreeNode(slow.val);
                thead.left=bstTree(head,slow);
                thead.right=bstTree(slow.next,tail);

        return thead;
            }


}
```

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

For this problem, a height-balanced binary tree is defined as:

a binary tree in which the left and right subtrees of every node differ in height by no more than 1.

Example 1:

Input: root = [3,9,20,null,null,15,7]
Output: true
Example 2:

Input: root = [1,2,2,3,3,null,null,4,4]
Output: false
Example 3:

Input: root = []
Output: true

Constraints:

The number of nodes in the tree is in the range [0, 5000].
-104 <= Node.val <= 104

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
public boolean isBalanced(TreeNode root) {
int a= isBalance(root);
return (a!=-1);
}

int isBalance(TreeNode node){
if(node==null)
    return 0;
    if(node.left==null && node.right==null)
        return 1;

    int l= isBalance(node.left);
    int r= isBalance(node.right);

    if(l==-1 || r==-1 || Math.abs(r-l)>1)
        return -1;

    return Math.max(l,r )+1;

    }
}
```

----------Minimum Depth of Binary Tree
----------Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

Note: A leaf is a node with no children.

Example 1:

Input: root = [3,9,20,null,null,15,7]
Output: 2
Example 2:

Input: root = [2,null,3,null,4,null,5,null,6]
Output: 5

Constraints:

The number of nodes in the tree is in the range [0, 105].
-1000 <= Node.val <= 1000

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public int minDepth(TreeNode root) {
        if(root == null) return 0;
        Queue<TreeNode> q = new LinkedList<TreeNode>();
        q.add(root);
        int min = 0;
        int temp = Integer.MAX_VALUE;
        while(!q.isEmpty()) {
            min++;
            int size = q.size();
            while(size-- > 0) {
                TreeNode cur = q.poll();
                if(cur.left == null && cur.right == null) {
                    temp = Math.min(min, temp);
                }
                if(cur.left != null) {
                    q.add(cur.left);
                }
                if(cur.right != null) {
                    q.add(cur.right);
                }

            }
            if(temp < min) return temp;
        }
        return temp;
```

```
    }
}
```

---Path Sum

---Given the root of a binary tree and an integer targetSum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals targetSum.

A leaf is a node with no children.

Example 1:

Input: root = [5,4,8,11,null,13,4,7,2,null,null,null,1], targetSum = 22
Output: true
Explanation: The root-to-leaf path with the target sum is shown.
Example 2:

Input: root = [1,2,3], targetSum = 5
Output: false
Explanation: There two root-to-leaf paths in the tree:
(1 --> 2): The sum is 3.
(1 --> 3): The sum is 4.
There is no root-to-leaf path with sum = 5.
Example 3:

Input: root = [], targetSum = 0
Output: false
Explanation: Since the tree is empty, there are no root-to-leaf paths.

Constraints:

The number of nodes in the tree is in the range [0, 5000].
-1000 <= Node.val <= 1000
-1000 <= targetSum <= 1000

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

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public boolean hasPathSum(TreeNode root, int sum) {
        if(root == null)
            return false;
        else
            return (root.left == null && root.right == null && root.val == sum)
            || hasPathSum(root.left, sum - root.val) || hasPathSum(root.right, sum - root.val);
    }
}
```

----------Path Sum II

----------Given the root of a binary tree and an integer targetSum, return all root-to-leaf paths where the sum of the node values in the path equals targetSum. Each path should be returned as a list of the node values, not node references.

A root-to-leaf path is a path starting from the root and ending at any leaf node. A leaf is a node with no children.

Example 1:

Input: root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22
Output: [[5,4,11,2],[5,8,4,5]]
Explanation: There are two paths whose sum equals targetSum:
5 + 4 + 11 + 2 = 22
5 + 8 + 4 + 5 = 22
Example 2:

Input: root = [1,2,3], targetSum = 5
Output: []
Example 3:

Input: root = [1,2], targetSum = 0
Output: []

Constraints:

The number of nodes in the tree is in the range [0, 5000].
-1000 <= Node.val <= 1000
-1000 <= targetSum <= 1000

```
/**
```

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

  public List<List<Integer>> pathSum(TreeNode root, int sum) {
       List<List<Integer>> result = new ArrayList<>();
       pathSum(root, sum, result, new ArrayList<>());
       return result;
    }
    private void pathSum(TreeNode root, int sum, List<List<Integer>> list, ArrayList<Integer>
path) {

        if (root == null) return;
        path.add(root.val);
        if (root.left == null && root.right == null && sum == root.val) {
            List<Integer> current = new ArrayList<>(path);
            list.add(current);
        }
        pathSum(root.left, sum - root.val, list, path);
        pathSum(root.right, sum - root.val, list, path);
        path.remove(path.size() - 1);

    }

}
```

Flatten Binary Tree to Linked List

Given the root of a binary tree, flatten the tree into a "linked list":

The "linked list" should use the same TreeNode class where the right child pointer points to the next node in the list and the left child pointer is always null.
The "linked list" should be in the same order as a pre-order traversal of the binary tree.

Example 1:

Input: root = [1,2,5,3,4,null,6]
Output: [1,null,2,null,3,null,4,null,5,null,6]
Example 2:

Input: root = []
Output: []

Example 3:

Input: root = [0]
Output: [0]

Constraints:

The number of nodes in the tree is in the range [0, 2000].
-100 <= Node.val <= 100

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public void flatten(TreeNode root) {
    helper(root);
}

public static TreeNode helper(TreeNode root){
    if(root==null){
        return null;
    }


    TreeNode lf=helper(root.left);
    TreeNode rt=helper(root.right);



    if(lf==null && rt==null){
        root.left=null;
        root.right=null;
        return root;
    }
    else if(lf!=null && rt==null){
        root.left=null;
        root.right=lf;
        return root;
    }

    else if (lf==null && rt!=null){
        root.left=null;
```

```
        root.right=rt;
        return root;
    }
    else{
    root.left=null;
    TreeNode a =lf;
    while(a.right!=null){
        a=a.right;
    }
    a.right=rt;

    root.right=lf;
       TreeNode t=root;
        // while(t!=null){-->for debugging
        //     System.out.println(t.val);
        //     t=t.right;
        // }



          return root;
    }


}
}
```

----------Distinct Subsequences
----------Given two strings s and t, return the number of distinct subsequences of s which equals t.

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

The test cases are generated so that the answer fits on a 32-bit signed integer.

Example 1:

Input: s = "rabbbit", t = "rabbit"
Output: 3
Explanation:
As shown below, there are 3 ways you can generate "rabbit" from S.
rabbbit
rabbbit
rabbbit
Example 2:

Input: s = "babgbag", t = "bag"
Output: 5
Explanation:
As shown below, there are 5 ways you can generate "bag" from S.
babgbag

babgbag
babgbag
babgbag
babgbag

Constraints:

1 <= s.length, t.length <= 1000
s and t consist of English letters.

```
class Solution {
    public int numDistinct(String s, String t) {
        int n = s.length();
        int m = t.length();              // t = "abcdefghijklmnopqrstuvwxyz";m=26;

        long mod = 1000000000+7;          // modulo for large numbers
        long dp[][] = new long[2][1+m];   //(dp,0,sizeof(dp));

        for(int i = 0 ; i<=n ; i++)
            dp[i%2][0] = 1;

        for(int i = 1 ; i<n+1 ;i++)
        {
            for(int j = 1 ; j<m+1 ;j++)
            {
                if(s.charAt(i-1)==t.charAt(j-1))
                    dp[i%2][j] = (  dp[(i-1)%2][j]+dp[(i-1)%2][j-1]   ) % mod ;      //not match+match
                else
                    dp[i%2][j] = ( dp[(i-1)%2][j] ) % mod;                          //not match

            }

        }

        return (int)dp[n%2][m] % mod ;
    }
}
```

```
class Solution {
public int numDistinct(String s, String t) {
        int dp[][] = new int[s.length()+1][t.length()+1];
        //fill every rows 1st column with 1.  Because every rows first colums denote solution for
searching "" in a string.
        for(int i=0;i<dp.length;i++){
            dp[i][0]=1;
        }
        for(int i=1;i<dp.length;i++){
            for(int j=1;j<dp[0].length;j++){
                if(s.charAt(i-1)==t.charAt(j-1)){
                    dp[i][j] = dp[i-1][j] + dp[i-1][j-1];
                }
                else {
```

```
                dp[i][j] = dp[i-1][j];
            }
        }
    }

    return dp[s.length()][t.length()];

  }

}
```

----------Populating Next Right Pointers in Each Node

----------You are given a perfect binary tree where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *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.

Example 1:

Input: root = [1,2,3,4,5,6,7]
Output: [1,#,2,3,#,4,5,6,7,#]
Explanation: Given the above perfect binary tree (Figure A), your function should populate each next pointer to point to its next right node, just like in Figure B. The serialized output is in level order as connected by the next pointers, with '#' signifying the end of each level.
Example 2:

Input: root = []
Output: []

Constraints:

The number of nodes in the tree is in the range [0, 212 - 1].
-1000 <= Node.val <= 1000

Follow-up:

You may only use constant extra space.
The recursive approach is fine. You may assume implicit stack space does not count as extra space for this problem

----------Populating Next Right Pointers in Each Node II

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

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

Note:

You may only use constant extra space.

For example,

Given the following binary tree,

1

/

\

2

3

/ \

4

\

5

7

After calling your function, the tree should look like:

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

/*
// Definition for a Node.
class Node {
    public int val;
    public Node left;
    public Node right;
    public Node next;

    public Node() {}

    public Node(int _val) {
        val = _val;
    }

    public Node(int _val, Node _left, Node _right, Node _next) {
        val = _val;
        left = _left;
        right = _right;
        next = _next;
    }
};
*/

class Solution {
    public Node connect(Node root) {
        if(root == null) return null;
        if(root.left != null) root.left.next = root.right;
        if(root.right != null && root.next != null) root.right.next = root.next.left;
        connect(root.left);
        connect(root.right);
        return root;
    }
}

------------------------------------------------------------Pascal's Triangle
-----------------------------Given numRows, generate the first numRows of Pascal's triangle.
For example, given numRows = 5,
Return
[
[1],
[1,1],
[1,2,1],
[1,3,3,1],
[1,4,6,4,1]
]
```

Example 1:

Input: numRows = 5
Output: [[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]
Example 2:

Input: numRows = 1
Output: [[1]]

Constraints:

1 <= numRows <= 30

```
class Solution {
    public List<List<Integer>> generate(int numRows) {
        List<List<Integer>>res = new ArrayList<>();
        for (int i=0;i<numRows;i++){
            List<Integer>temp = new ArrayList<>();
            temp.add(1);
            if (i>0){
                List<Integer>prelast = res.get(i-1);
                for (int j=0;j<prelast.size()-1;j++){
                    temp.add(prelast.get(j)+prelast.get(j+1));
                }
                temp.add(1);
            }
            res.add(temp);
        }
        return res;
    }
}
```

----------Pascal's Triangle II
----------Given an index k, return the kth row of the Pascal's triangle.
For example, given k = 3,
Return [1,3,3,1].

Note:
Could you optimize your algorithm to use only O(k) extra space?

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

----------Triangle
----------Input: triangle = [[2],[3,4],[6,5,7],[4,1,8,3]]
Output: 11
Explanation: The triangle looks like:

```
  2
 3 4
6 5 7
4 1 8 3
```
The minimum path sum from top to bottom is 2 + 3 + 5 + 1 = 11 (underlined above).
Example 2:

Input: triangle = [[-10]]
Output: -10

Constraints:

1 <= triangle.length <= 200
triangle[0].length == 1
triangle[i].length == triangle[i - 1].length + 1
-104 <= triangle[i][j] <= 104

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

----------Best Time to Buy and Sell Stock----------Say you have an array for which the ith element is the price of a given stock on day i.
If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.
Example 1:
Input: [7, 1, 5, 3, 6, 4]
Output: 5
max. difference = 6-1 = 5 (not 7-1 = 6, as selling price needs to be larger than buying price)

Example 2:
Input: [7, 6, 4, 3, 1]
Output: 0
In this case, no transaction is done, i.e. max profit = 0.

```
class Solution {
  public int maxProfit(int[] prices) {
          int ans=0;
          if(prices.length==0)
          {
              return ans;
          }
          int bought=prices[0];
          for(int i=1;i<prices.length;i++)
```

```
        {
            if(prices[i]>bought)
            {
                if(ans<(prices[i]-bought))
                {
                    ans=prices[i]-bought;
                }
            }
            else
            {
                bought=prices[i];
            }
        }
    return ans;
}
}
```

----------Best Time to Buy and Sell Stock II----------You are given an integer array prices where prices[i] is the price of a given stock on the ith day.

On each day, you may decide to buy and/or sell the stock. You can only hold at most one share of the stock at any time. However, you can buy it then immediately sell it on the same day.

Find and return the maximum profit you can achieve.

Example 1:

Input: prices = [7,1,5,3,6,4]
Output: 7
Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1 = 4.
Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6-3 = 3.
Total profit is 4 + 3 = 7.
Example 2:

Input: prices = [1,2,3,4,5]
Output: 4
Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.
Total profit is 4.
Example 3:

Input: prices = [7,6,4,3,1]
Output: 0
Explanation: There is no way to make a positive profit, so we never buy the stock to achieve the maximum profit of 0.

Constraints:

1 <= prices.length <= 3 * 104
0 <= prices[i] <= 104

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

```
        if (p.length < 2) return 0;

        var min = 0;
        var cursor = 0;
        var limit = p.length - 1;
        var total = 0;

        while (cursor < limit) {
            while (cursor < limit && p[cursor] >= p[cursor + 1]) {
                cursor++;
            }
            min = p[cursor];
            while (cursor < limit && p[cursor] <= p[cursor + 1]) {
                cursor++;
            }
            total += p[cursor] - min;
        }

        return total;
    }
}
```

----------Best Time to Buy and Sell Stock III

----------Example 1:

Input: prices = [3,3,5,0,0,3,1,4]
Output: 6
Explanation: Buy on day 4 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.
Then buy on day 7 (price = 1) and sell on day 8 (price = 4), profit = 4-1 = 3.
Example 2:

Input: prices = [1,2,3,4,5]
Output: 4
Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.
Note that you cannot buy on day 1, buy on day 2 and sell them later, as you are engaging multiple transactions at the same time. You must sell before buying again.
Example 3:

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

Input: prices = [1]
Output: 0

Constraints:

1 <= prices.length <= 105
0 <= prices[i] <= 105

```
class Solution {
    public int maxProfit(int[] prices) {
        if(prices == null || prices.length < 1) return 0;
```

```
        int buy1 = -prices[0], sell1 = 0, buy2 = -prices[0], sell2 = 0;
        for(int i = 1; i < prices.length; i++) {
            buy1 = Math.max(buy1, -prices[i]);
            sell1 = Math.max(sell1, buy1 + prices[i]);
            buy2 = Math.max(buy2, sell1 - prices[i]);
            sell2 = Math.max(sell2, buy2 + prices[i]);
        }
        return sell2;
    }
}
```

----------Binary Tree Maximum Path Sum----------

A path in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence at most once. Note that the path does not need to pass through the root.

The path sum of a path is the sum of the node's values in the path.

Given the root of a binary tree, return the maximum path sum of any non-empty path.

Example 1:

Input: root = [1,2,3]
Output: 6
Explanation: The optimal path is 2 -> 1 -> 3 with a path sum of 2 + 1 + 3 = 6.
Example 2:

Input: root = [-10,9,20,null,null,15,7]
Output: 42
Explanation: The optimal path is 15 -> 20 -> 7 with a path sum of 15 + 20 + 7 = 42.

Constraints:

The number of nodes in the tree is in the range [1, 3 * 104].
-1000 <= Node.val <= 1000

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

```
 */
class Solution {
    public int findPathSum(TreeNode root){
        if(root == null){
            return 0;
        }
        else{
            int left = findPathSum(root.left);
            int right = findPathSum(root.right);
            int currentPathSum = Math.max(left+root.val, Math.max(right+root.val, root.val));
            maxPathSum = Math.max(maxPathSum, (left+right+root.val));
            maxPathSum = Math.max(maxPathSum, currentPathSum);
            return currentPathSum;
        }
    }

    int maxPathSum;
    public int maxPathSum(TreeNode root) {
        maxPathSum = Integer.MIN_VALUE;
        return Math.max(findPathSum(root), maxPathSum);
    }
}
```

----------Valid Palindrome
----------A phrase is a palindrome if, after converting all uppercase letters into lowercase letters and removing all non-alphanumeric characters, it reads the same forward and backward. Alphanumeric characters include letters and numbers.

Given a string s, return true if it is a palindrome, or false otherwise.

Example 1:

Input: s = "A man, a plan, a canal: Panama"
Output: true
Explanation: "amanaplanacanalpanama" is a palindrome.
Example 2:

Input: s = "race a car"
Output: false
Explanation: "raceacar" is not a palindrome.
Example 3:

Input: s = " "
Output: true
Explanation: s is an empty string "" after removing non-alphanumeric characters.
Since an empty string reads the same forward and backward, it is a palindrome.

Constraints:

1 <= s.length <= 2 * 105

s consists only of printable ASCII characters.

```
class Solution {
    public boolean isPalindrome(String s) {
        int start=0,end=s.length()-1;
        while(start<end){
            while(!((s.charAt(start)>='a'&&s.charAt(start)<='z')||
                    (s.charAt(start)>='A'&&s.charAt(start)<='Z')||
                    (s.charAt(start)>='0'&&s.charAt(start)<='9'))&&start<end)
                      start++;
            while(!((s.charAt(end)>='a'&&s.charAt(end)<='z')||
                    (s.charAt(end)>='A'&&s.charAt(end)<='Z')||
                    (s.charAt(end)>='0'&&s.charAt(end)<='9'))&&start<end)
                       end--;
        char left = s.charAt(start);
        char right = s.charAt(end);
        //convert uppercase to lowercase
        if(left>='A' && left<='Z') left = (char)(left - 'A' + 'a');
        if(right>='A' && right<='Z') right = (char)(right - 'A' + 'a');
            if(left!=right)
                        return false;
            start++;
            end--;
        }
       return true;
    }
}
```

----------Valid Palindrome II
----------Given a string s, return true if the s can be palindrome after deleting at most one character from it.

Example 1:

Input: s = "aba"
Output: true
Example 2:

Input: s = "abca"
Output: true
Explanation: You could delete the character 'c'.
Example 3:

Input: s = "abc"
Output: false

Constraints:

1 <= s.length <= 105
s consists of lowercase English letters.

```
class Solution {

    public boolean validPalindrome(String s) {
        return isPalindrome(s, 0, s.length() - 1, false);
    }

    public boolean isPalindrome(final String s, int leftIndex, int rightIndex, final boolean
isCharacterDeleted){

        while(leftIndex < rightIndex){

            if(s.charAt(leftIndex) != s.charAt(rightIndex)){

                if(isCharacterDeleted){
                    return false;
                }

                // isPalindrome(s, leftIndex + 1, rightIndex, true) for cases like "ececabbacec"
                // isPalindrome(s, leftIndex, rightIndex - 1, true) for cases like "abccbab"
                return isPalindrome(s, leftIndex + 1, rightIndex, true) || isPalindrome(s, leftIndex,
rightIndex - 1, true);
            }

            ++leftIndex;
            --rightIndex;

        }

        return true;
    }
}
```

----------Word Ladder II
----------A transformation sequence from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s1 -> s2 -> ... -> sk such that:

Every adjacent pair of words differs by a single letter.
Every si for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.
sk == endWord
Given two words, beginWord and endWord, and a dictionary wordList, return all the shortest transformation sequences from beginWord to endWord, or an empty list if no such sequence exists. Each sequence should be returned as a list of the words [beginWord, s1, s2, ..., sk].

Example 1:

Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]
Output: [["hit","hot","dot","dog","cog"],["hit","hot","lot","log","cog"]]
Explanation: There are 2 shortest transformation sequences:
"hit" -> "hot" -> "dot" -> "dog" -> "cog"
"hit" -> "hot" -> "lot" -> "log" -> "cog"

Example 2:

Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]
Output: []
Explanation: The endWord "cog" is not in wordList, therefore there is no valid transformation sequence.

Constraints:

1 <= beginWord.length <= 5
endWord.length == beginWord.length
1 <= wordList.length <= 1000
wordList[i].length == beginWord.length
beginWord, endWord, and wordList[i] consist of lowercase English letters.
beginWord != endWord
All the words in wordList are unique.

```
class Solution {

    Set<String> wordSet = new HashSet<> ();
    List<String> currPath = new ArrayList<> ();
    List<List<String>> shortestPath = new ArrayList<> ();
    Map<String, List<String>> adjacencyList = new HashMap<> ();

    public List<String> findNeighbors (String currWord) {

        List<String> neighborsList = new ArrayList<> ();
        char[] letters = currWord.toCharArray ();

        for (int i = 0; i < letters.length; i++) {
            char letter = letters[i];

            for (char c = 'a'; c <= 'z'; c++) {
                letters[i] = c;
                String word = new String (letters);
                if (letter != c && wordSet.contains (word)) {
                    neighborsList.add (word);
                }
            }

            letters[i] = letter;
        }

        return neighborsList;
    }

    public void bfs (String beginWord) {

        Queue<String> queue = new LinkedList<> ();
        queue.offer (beginWord);
        wordSet.remove (beginWord);

        while (!queue.isEmpty ()) {
```

```
            int size = queue.size ();
            Set<String> visitedSet = new HashSet<> ();

            while (size-- != 0) {
                String currWord = queue.poll ();
                List<String> neighborsList = findNeighbors (currWord);

                for (String word : neighborsList) {
                    if (!adjacencyList.containsKey (currWord)) {
                        adjacencyList.put (currWord, new ArrayList<> ());
                    }

                    adjacencyList.get (currWord).add (word);
                    visitedSet.add (word);
                }
            }

            for (String word : visitedSet) {
                queue.offer (word);
                wordSet.remove (word);
            }
        }
    }

    public void backtrack (String beginWord, String endWord) {

        if (beginWord.equals (endWord)) {
            shortestPath.add (new ArrayList<> (currPath));
            return;
        }
        else if (!adjacencyList.containsKey (beginWord)) {
            return;
        }

        for (String word : adjacencyList.get (beginWord)) {
            currPath.add (word);
            backtrack (word, endWord);
            currPath.remove (word);
        }
    }

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

        for (String word : wordList) {
            wordSet.add (word);
        }

        if (!wordSet.contains (endWord)) {
            return shortestPath;
        }

        bfs (beginWord);
        currPath.add (beginWord);
```

```
        backtrack (beginWord, endWord);

        return shortestPath;
    }
}
```
----------Word Ladder
----------A transformation sequence from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s1 -> s2 -> ... -> sk such that:

Every adjacent pair of words differs by a single letter.
Every si for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.
sk == endWord
Given two words, beginWord and endWord, and a dictionary wordList, return the number of words in the shortest transformation sequence from beginWord to endWord, or 0 if no such sequence exists.

Example 1:

Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]
Output: 5
Explanation: One shortest transformation sequence is "hit" -> "hot" -> "dot" -> "dog" -> cog", which is 5 words long.
Example 2:

Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]
Output: 0
Explanation: The endWord "cog" is not in wordList, therefore there is no valid transformation sequence.

Constraints:

1 <= beginWord.length <= 10
endWord.length == beginWord.length
1 <= wordList.length <= 5000
wordList[i].length == beginWord.length
beginWord, endWord, and wordList[i] consist of lowercase English letters.
beginWord != endWord
All the words in wordList are unique.

```
class Solution {
  public int ladderLength(String beginWord, String endWord, List<String> wordList) {
    Set<String> wordSet = new HashSet<String>(wordList);
    if( !wordSet.contains(endWord) )
        return 0;
    // 3. Use set instead of queue during bfs
    Set<String> forwardSet = new HashSet<String>();
    Set<String> backwardSet = new HashSet<String>();
    forwardSet.add(beginWord);
    backwardSet.add(endWord);
    wordSet.remove(endWord);
```

```
    wordSet.remove(beginWord);
    // 1. Search from entry and exit simultaneously
    return transform(forwardSet, backwardSet, wordSet);
}

public int transform(Set<String> forwardSet, Set<String> backwardSet, Set<String> wordSet) {
    Set<String> newSet = new HashSet<String>();
    for(String fs : forwardSet) {
        char wordArray[] = fs.toCharArray();
        for(int i = 0; i < wordArray.length; i++) {
            for(int c = 'a'; c <= 'z'; c++) {
                char origin = wordArray[i];
                wordArray[i] = (char) c;
                String target = String.valueOf(wordArray);
                if( backwardSet.contains(target) )
                    return 2; // stop bfs when entry and exits meet
                else if( wordSet.contains(target) && !forwardSet.contains(target) ) {
                    wordSet.remove(target); // 4. Remove visited word from wordList to decrease the
search time
                    newSet.add(target);
                }
                wordArray[i] = origin;
            }
        }
    }
    if( newSet.size() == 0 )
        return 0;
    forwardSet = newSet;
    // 2. Pick the queue with less elements to bfs
    int result = forwardSet.size() > backwardSet.size() ?
        transform(backwardSet, forwardSet, wordSet) : transform(forwardSet, backwardSet,
wordSet);
    return result == 0 ? 0 : result + 1;
}
}
```

Longest Consecutive Sequence

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

You must write an algorithm that runs in O(n) time.

Example 1:

Input: nums = [100,4,200,1,3,2]
Output: 4
Explanation: The longest consecutive elements sequence is [1, 2, 3, 4]. Therefore its length is 4.

Example 2:

Input: nums = [0,3,7,2,5,8,4,6,0,1]
Output: 9

Constraints:

0 <= nums.length <= 105
-109 <= nums[i] <= 109

```
//HashSet solution
class Solution {
    public int longestConsecutive(int[] nums) {
        HashSet<Integer> hash = new HashSet();
        for(int x : nums){
            hash.add(x);
        }
        int maxx = 0;
        for(int x : hash){
            if(!hash.contains(x-1)){
                int currentNum = x;
                int currentMax = 1;
                while(hash.contains(currentNum+1)){
                    currentMax += 1;
                    currentNum += 1;
                }
                maxx = Math.max(maxx, currentMax);
            }
        }
        return maxx;
    }
}
```

----------Sum Root to Leaf Numbers
----------Given a binary tree containing digits from 0-9 only, each root-toleaf 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.

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

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

----------Surrounded Regions
----------Given an m x n matrix board containing 'X' and 'O', capture all regions that are 4-directionally surrounded by 'X'.

A region is captured by flipping all 'O's into 'X's in that surrounded region.

Example 1:

Input: board = [["X","X","X","X"],["X","O","O","X"],["X","X","O","X"],["X","O","X","X"]]
Output: [["X","X","X","X"],["X","X","X","X"],["X","X","X","X"],["X","O","X","X"]]
Explanation: Surrounded regions should not be on the border, which means that any 'O' on the border of the board are not flipped to 'X'. Any 'O' that is not on the border and it is not connected to an 'O' on the border will be flipped to 'X'. Two cells are connected if they are adjacent cells connected horizontally or vertically.
Example 2:

Input: board = [["X"]]
Output: [["X"]]

Constraints:

m == board.length
n == board[i].length
1 <= m, n <= 200

board[i][j] is 'X' or 'O'.

```
class Solution {
int array[][] = new int[][]{{0, 1}, {0, -1}, {-1, 0}, {1, 0}};

public void solve(char[][] board) {
    if( board.length < 1 || board[0].length < 1 )
        return;
    boolean isBorder[][] = new boolean[board.length][board[0].length]; // record the result of dfs

    // 1. dfs 'O' on the border
    for(int i = 0; i < board.length; i++) {
        if( board[i][0] == 'O' && !isBorder[i][0] )
            traverse(board, isBorder, i, 0);
        if( board[i][board[0].length - 1] == 'O' && !isBorder[i][board[0].length - 1] )
            traverse(board, isBorder, i, board[0].length - 1);
    }
    for(int i = 0; i < board[0].length; i++) {
        if( board[0][i] == 'O' && !isBorder[0][i] )
            traverse(board, isBorder, 0, i);
        if( board[board.length - 1][i] == 'O' && !isBorder[board.length - 1][i] )
            traverse(board, isBorder, board.length - 1, i);
    }

    // 2. Flip regions
    for(int i = 0; i < board.length; i++)
        for(int j = 0; j < board[0].length; j++)
            if( !isBorder[i][j] )
                board[i][j] = 'X';
}

public void traverse(char[][] board, boolean[][] isBorder, int r, int c) {
    isBorder[r][c] = true;
    for(int i = 0; i < 4; i++) {
        int rr = r + array[i][0];
        int cc = c + array[i][1];
        if( rr >= 0 && cc >= 0 && rr < board.length && cc < board[0].length
           && board[rr][cc] == 'O' && !isBorder[rr][cc] )
            traverse(board, isBorder, rr, cc);
    }
}
}
```

----------Palindrome Partitioning
----------Given a string s, partition s such that every substring of the partition is a palindrome. Return all possible palindrome partitioning of s.

A palindrome string is a string that reads the same backward as forward.

Example 1:

Input: s = "aab"
Output: [["a","a","b"],["aa","b"]]
Example 2:

Input: s = "a"
Output: [["a"]]

Constraints:

1 <= s.length <= 16
s contains only lowercase English letters.

```
class Solution {
    public List<List<String>> partition(String s) {
        List<List<String>> result = new ArrayList<List<String>>();
        dfs(0, result, new ArrayList<String>(), s);
        return result;
    }

    void dfs(int start, List<List<String>> result, List<String> currentList, String s) {
        if (start >= s.length()) result.add(new ArrayList<String>(currentList));
        for (int end = start; end < s.length(); end++) {
            if (isPalindrome(s, start, end)) {
                // add current substring in the currentList
                currentList.add(s.substring(start, end + 1));
                dfs(end + 1, result, currentList, s);
                // backtrack and remove the current substring from currentList
                currentList.remove(currentList.size() - 1);
            }
        }
    }

    boolean isPalindrome(String s, int low, int high) {
        while (low < high) {
            if (s.charAt(low++) != s.charAt(high--)) return false;
        }
        return true;
    }
}
```

----------Palindrome Partitioning II
----------Given a string s, partition s such that every substring of the partition is a palindrome.

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

Example 1:

Input: s = "aab"
Output: 1
Explanation: The palindrome partitioning ["aa","b"] could be produced using 1 cut.
Example 2:

Input: s = "a"
Output: 0
Example 3:

Input: s = "ab"
Output: 1

Constraints:

1 <= s.length <= 2000
s consists of lower-case English letters only.

```
class Solution {
  public void extendPalindrome (char[] c, int start, int end, int[] minimumCuts) {

        while (start >= 0 && end < c.length && c[start] == c[end]) {
            minimumCuts[end + 1] = Math.min (minimumCuts[end + 1], minimumCuts[start] + 1);
            start--;
            end++;
        }
    }

    public int minCut(String s) {

        int length = s.length ();
        char[] c = s.toCharArray ();
        int[] minimumCuts = new int[length + 1];

        for (int i = 0; i <= length; i++) {
            minimumCuts[i] = i - 1;
        }

        for (int i = 0; i < length; i++) {
            extendPalindrome (c, i, i, minimumCuts);
            extendPalindrome (c, i, i + 1, minimumCuts);
        }

        return minimumCuts[length];
    }
}
```

----------Clone Graph
----------Given a reference of a node in a connected undirected graph.

Return a deep copy (clone) of the graph.

Each node in the graph contains a value (int) and a list (List[Node]) of its neighbors.

```
class Node {
    public int val;
    public List<Node> neighbors;
}
```

Test case format:

For simplicity, each node's value is the same as the node's index (1-indexed). For example, the first node with val == 1, the second node with val == 2, and so on. The graph is represented in the test case using an adjacency list.

An adjacency list is a collection of unordered lists used to represent a finite graph. Each list describes the set of neighbors of a node in the graph.

The given node will always be the first node with val = 1. You must return the copy of the given node as a reference to the cloned graph.

Example 1:

Input: adjList = [[2,4],[1,3],[2,4],[1,3]]
Output: [[2,4],[1,3],[2,4],[1,3]]
Explanation: There are 4 nodes in the graph.
1st node (val = 1)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).
2nd node (val = 2)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).
3rd node (val = 3)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).
4th node (val = 4)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).
Example 2:

Input: adjList = [[]]
Output: [[]]
Explanation: Note that the input contains one empty list. The graph consists of only one node with val = 1 and it does not have any neighbors.
Example 3:

Input: adjList = []
Output: []
Explanation: This an empty graph, it does not have any nodes.
Example 4:

Input: adjList = [[2],[1]]
Output: [[2],[1]]

Constraints:

The number of nodes in the graph is in the range [0, 100].
1 <= Node.val <= 100
Node.val is unique for each node.
There are no repeated edges and no self-loops in the graph.
The Graph is connected and all nodes can be visited starting from the given node.

/*
// Definition for a Node.

```
class Node {
    public int val;
    public List<Node> neighbors;
    public Node() {
        val = 0;
        neighbors = new ArrayList<Node>();
    }
    public Node(int _val) {
        val = _val;
        neighbors = new ArrayList<Node>();
    }
    public Node(int _val, ArrayList<Node> _neighbors) {
        val = _val;
        neighbors = _neighbors;
    }
}
*/

class Solution {
    Map<Integer, Node> map = new HashMap<Integer, Node>();
    public Node cloneGraph(Node node) {
    if(node == null) return null;
    if(map.containsKey(node.val)) {
        return map.get(node.val);
    }

    Node cloned = new Node(node.val);
    map.put(node.val, cloned);
    for(Node nb : node.neighbors) {
        cloned.neighbors.add(cloneGraph(nb));
    }
    return cloned;
}
}
```

----------Gas Station
----------There are n gas stations along a circular route, where the amount of gas at the ith station is gas[i].

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

Given two integer arrays gas and cost, return the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return -1. If there exists a solution, it is guaranteed to be unique

Example 1:

Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
Output: 3
Explanation:
Start at station 3 (index 3) and fill up with 4 unit of gas. Your tank = 0 + 4 = 4
Travel to station 4. Your tank = 4 - 1 + 5 = 8
Travel to station 0. Your tank = 8 - 2 + 1 = 7
Travel to station 1. Your tank = 7 - 3 + 2 = 6
Travel to station 2. Your tank = 6 - 4 + 3 = 5
Travel to station 3. The cost is 5. Your gas is just enough to travel back to station 3.
Therefore, return 3 as the starting index.
Example 2:

Input: gas = [2,3,4], cost = [3,4,3]
Output: -1
Explanation:
You can't start at station 0 or 1, as there is not enough gas to travel to the next station.
Let's start at station 2 and fill up with 4 unit of gas. Your tank = 0 + 4 = 4
Travel to station 0. Your tank = 4 - 3 + 2 = 3
Travel to station 1. Your tank = 3 - 3 + 3 = 3
You cannot travel back to station 2, as it requires 4 unit of gas but you only have 3.
Therefore, you can't travel around the circuit once no matter where you start.

Constraints:

gas.length == n
cost.length == n
1 <= n <= 105
0 <= gas[i], cost[i] <= 104

```
class Solution {
    public int canCompleteCircuit(int[] gas, int[] cost) {

        int minFuel = Integer.MAX_VALUE, minIdx = 0, currFuel = 0;

        for(int i=0; i<gas.length; i++){
            currFuel += gas[i]-cost[i];
            if(currFuel < minFuel){
```

```
                minFuel =  currFuel;
                minIdx = i;
            }
        }

        // if after going through the loop, the 'currFuel' is -ve, it means not enough fuel to circle;
        // if 'currFuel' >= 0, it means sufficent fuel there to circle
        return currFuel >=0? (minIdx+1)%gas.length : -1;
    }
}
```

There are n children standing in a line. Each child is assigned a rating value given in the integer array ratings.

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

Each child must have at least one candy.
Children with a higher rating get more candies than their neighbors.
Return the minimum number of candies you need to have to distribute the candies to the children.

Example 1:

Input: ratings = [1,0,2]
Output: 5
Explanation: You can allocate to the first, second and third child with 2, 1, 2 candies respectively.
Example 2:

Input: ratings = [1,2,2]
Output: 4
Explanation: You can allocate to the first, second and third child with 1, 2, 1 candies respectively.
The third child gets 1 candy because it satisfies the above two conditions.

Constraints:

n == ratings.length
1 <= n <= 2 * 104
0 <= ratings[i] <= 2 * 104

```
class Solution {
    public int candy(int[] ratings) {
        int n=ratings.length;
        int[] test1=new int[n];
        int[] test2=new int[n];
        //initialization both arrays to 1
        for(int i=0;i<n;i++)test2[i]=test1[i]=1;
        //calculating the values from left to right
        for(int i=0;i<n;i++){
            if(i>0 && ratings[i]>ratings[i-1]){
```

```
                test1[i]=Math.max(test1[i-1]+1,test1[i]);
            }
            if(i+1<n && ratings[i]>ratings[i+1]){
                test1[i]=Math.max(test1[i+1]+1,test1[i]);
            }
        }
        //calculating the values from right to left
        for(int i=n-1;i>=0;i--){
            if(i>0 && ratings[i]>ratings[i-1]){
                test2[i]=Math.max(test2[i-1]+1,test2[i]);
            }
            if(i+1<n && ratings[i]>ratings[i+1]){
                test2[i]=Math.max(test2[i+1]+1,test2[i]);
            }
        }
        //summing up the max value from both the routes
        int sum=0;
        for(int i=0;i<n;i++){
            sum+=Math.max(test1[i],test2[i]);
        }
        //returning the result
        return sum;
    }
}
```

----------Single Number

----------Given a non-empty array of integers nums, every element appears twice except for one. Find that single one.

You must implement a solution with a linear runtime complexity and use only constant extra space.

Example 1:

Input: nums = [2,2,1]
Output: 1
Example 2:

Input: nums = [4,1,2,1,2]
Output: 4
Example 3:

Input: nums = [1]
Output: 1

Constraints:

1 <= nums.length <= 3 * 104
-3 * 104 <= nums[i] <= 3 * 104
Each element in the array appears twice except for one element which appears only once.

```
class Solution {
```

```
    public int singleNumber(int[] nums) {
        int res = nums[0];
        for(int i=1; i<nums.length; i++){
            res = res^nums[i];
        }
        return res;
    }
}
```

----------Single Number II

----------Given an integer array nums where every element appears three times except for one, which appears exactly once. Find the single element and return it.

You must implement a solution with a linear runtime complexity and use only constant extra space.

Example 1:

Input: nums = [2,2,3,2]
Output: 3
Example 2:

Input: nums = [0,1,0,1,0,1,99]
Output: 99

Constraints:

1 <= nums.length <= 3 * 104
-231 <= nums[i] <= 231 - 1
Each element in nums appears exactly three times except for one element which appears once.

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

----------Copy List with Random Pointer

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

Construct a deep copy of the list. The deep copy should consist of exactly n brand new nodes, where each new node has its value set to the value of its corresponding original node. Both the next and random pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. None of the pointers in the new list should point to nodes in the original list.

For example, if there are two nodes X and Y in the original list, where X.random --> Y, then for the corresponding two nodes x and y in the copied list, x.random --> y.

Return the head of the copied linked list.

The linked list is represented in the input/output as a list of n nodes. Each node is represented as a pair of [val, random_index] where:

val: an integer representing Node.val
random_index: the index of the node (range from 0 to n-1) that the random pointer points to, or null if it does not point to any node.
Your code will only be given the head of the original linked list.

Example 1:

Input: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
Output: [[7,null],[13,0],[11,4],[10,2],[1,0]]
Example 2:

Input: head = [[1,1],[2,1]]
Output: [[1,1],[2,1]]
Example 3:

Input: head = [[3,null],[3,0],[3,null]]
Output: [[3,null],[3,0],[3,null]]
Example 4:

Input: head = []
Output: []
Explanation: The given linked list is empty (null pointer), so return null.

Constraints:

0 <= n <= 1000
-10000 <= Node.val <= 10000
Node.random is null or is pointing to some node in the linked list.

```
/*
// Definition for a Node.
class Node {
    int val;
    Node next;
    Node random;

    public Node(int val) {
        this.val = val;
```

```
        this.next = null;
        this.random = null;
    }
}
*/

class Solution {
    public Node copyRandomList(Node head) {
        if (head == null) {
            return null;
        }
        // O(n) solution using map
        // key: random pointer
        // value: original pointer
        Map<Node, Node> pseudoRandomMap = new HashMap<Node, Node>(); // used to deep
copy the random nodes

        // first, deep copy the next pointers
        Node dummy = new Node(0);
        Node curr = head;
        Node currPrev = new Node(0);
        Node prev = dummy;
        while (curr != null) {
            prev.next = new Node(curr.val);
            prev.next.random = curr;

            // we temporarily set the random pointers of the cloned nodes
            // to the original node that is in the same relative position
            pseudoRandomMap.put(curr, prev.next);

            prev = prev.next;
            curr = curr.next;
        }

        // second, deep copy the randomPointers
        // curr and prev are the in the same relative position, so
        // curr.random and prev.random should be in the same relative position
        curr = head;
        prev = dummy.next;
        while (curr != null) {
            Node rand = curr.random;
            if (rand == null) {
                prev.random = null;
            } else {
                prev.random = pseudoRandomMap.get(rand);
            }
            curr = curr.next;
            prev = prev.next;
        }

        return dummy.next;
    }
}
```

Word Break

------------------------------Given a string s and a dictionary of strings wordDict, return true if s can be segmented into a space-separated sequence of one or more dictionary words.

Note that the same word in the dictionary may be reused multiple times in the segmentation.

Example 1:

Input: s = "leetcode", wordDict = ["leet","code"]
Output: true
Explanation: Return true because "leetcode" can be segmented as "leet code".
Example 2:

Input: s = "applepenapple", wordDict = ["apple","pen"]
Output: true
Explanation: Return true because "applepenapple" can be segmented as "apple pen apple".
Note that you are allowed to reuse a dictionary word.
Example 3:

Input: s = "catsandog", wordDict = ["cats","dog","sand","and","cat"]
Output: false

Constraints:

1 <= s.length <= 300
1 <= wordDict.length <= 1000
1 <= wordDict[i].length <= 20
s and wordDict[i] consist of only lowercase English letters.
All the strings of wordDict are unique.

```
// for each word in dict:
//     if current index of s starts with word:
//         increment index to word length and recurse until end of word, returning true
// else nothing found to work so return false

class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {
        byte[] dp = new byte[s.length()];    // dp[i] = if possible to use words from wordDict to make
s up to index i
        Arrays.fill(dp, (byte) -1);          // initialize all to -1 to indicate not calculated yet

        return wordBreak(s, 0, wordDict, dp);
    }

    private boolean wordBreak(String s, int index, List<String> wordDict, byte[] dp) {
        if (index >= s.length())                // index past entire string already, means we found a
solution
            return true;

        if (dp[index] != -1)
            return dp[index] == 1;              // decision here was using 1 to represent true, and 0 to
represent false
```

```
        byte match = 0;
        for (int i = 0; i < wordDict.size(); i++) {
            String word = wordDict.get(i);
            if (s.startsWith(word, index) && wordBreak(s, index + word.length(), wordDict, dp)) {    //
if the word starts at index, we recursively look for solution down this path
                match = 1;
                break;    // found a solution already so can break and skip work
            }
        }

        dp[index] = match;    // save result in dp for future use

        return match == 1;
    }
}
```

----------Word Break II
----------Given a string s and a dictionary of strings wordDict, add spaces in s to construct a sentence where each word is a valid dictionary word. Return all such possible sentences in any order.

Note that the same word in the dictionary may be reused multiple times in the segmentation.

Example 1:

Input: s = "catsanddog", wordDict = ["cat","cats","and","sand","dog"]
Output: ["cats and dog","cat sand dog"]
Example 2:

Input: s = "pineapplepenapple", wordDict = ["apple","pen","applepen","pine","pineapple"]
Output: ["pine apple pen apple","pineapple pen apple","pine applepen apple"]
Explanation: Note that you are allowed to reuse a dictionary word.
Example 3:

Input: s = "catsandog", wordDict = ["cats","dog","sand","and","cat"]
Output: []

Constraints:

1 <= s.length <= 20
1 <= wordDict.length <= 1000
1 <= wordDict[i].length <= 10
s and wordDict[i] consist of only lowercase English letters.
All the strings of wordDict are unique.

```
class Solution {
    public List<String> wordBreak(String s, List<String> wordDict) {
        Set<String> s1 = new HashSet<>(wordDict);
        List<String>r  = new ArrayList<String>();
        search(s, 0, s1, new StringBuilder(), r);
        return r;
    }

    public void search(String s, int i,  Set<String> d, StringBuilder sb, List<String> result) {
        if(i == s.length()) {
            result.add(sb.toString().trim());
            return;
        }
        int len = sb.length();
        for(int j = i+1; j<= s.length();j++) {
            String t1 = s.substring(i, j);
            if(d.contains(t1)) {
                search(s, j, d, sb.append(t1).append(" "),result);
            }
            sb.setLength(len);
        }
    }
```

}
----------Linked List Cycle
----------Given head, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. Note that pos is not passed as a parameter.

Return true if there is a cycle in the linked list. Otherwise, return false.

Example 1:

Input: head = [3,2,0,-4], pos = 1
Output: true
Explanation: There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).
Example 2:

Input: head = [1,2], pos = 0
Output: true
Explanation: There is a cycle in the linked list, where the tail connects to the 0th node.
Example 3:

Input: head = [1], pos = -1
Output: false
Explanation: There is no cycle in the linked list.

Constraints:

The number of the nodes in the list is in the range [0, 104].
-105 <= Node.val <= 105
pos is -1 or a valid index in the linked-list.

```
/**
 * 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, fast = head;

  while (fast != null && fast.next != null) {
    slow = slow.next;
    fast = fast.next.next;

    if (slow == fast)
        return true;
  }

  return false;
}
}
```
----------Linked List Cycle II
----------Given the head of a linked list, return the node where the cycle begins. If there is no cycle, return null.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to (0-indexed). It is -1 if there is no cycle. Note that pos is not passed as a parameter.

Do not modify the linked list.

Example 1:

Input: head = [3,2,0,-4], pos = 1
Output: tail connects to node index 1
Explanation: There is a cycle in the linked list, where tail connects to the second node.
Example 2:

Input: head = [1,2], pos = 0
Output: tail connects to node index 0
Explanation: There is a cycle in the linked list, where tail connects to the first node.
Example 3:

Input: head = [1], pos = -1
Output: no cycle
Explanation: There is no cycle in the linked list.

Constraints:

The number of the nodes in the list is in the range [0, 104].
-105 <= Node.val <= 105
pos is -1 or a valid index in the linked-list.

```
public class Solution {
    public ListNode detectCycle(ListNode head) {
```

```
    var slow = head;
    var fast = head;

    // If there's a cycle, let the pointers enter it and meet
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
        if (slow == fast)
            break;
    }

    // check if there's no cycle
    if (fast == null || fast.next == null)
        return null;

    // find the entry point of the cycle
    slow = head;
    while (slow != fast) {
        slow = slow.next;
        fast = fast.next;
    }
    return slow;
}
}
----------------------------------------------------------------Reorder List
------------------------------Given a singly linked list L: L0→L1→…→Ln-1→Ln,
reorder it to: L0→Ln→L1→Ln-1→L2→Ln-2→…
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}.
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public void reorderList(ListNode head) {
        if(head == null || head.next == null) return;
```

```
        ListNode slow = head, fast = head;

        // 1. Divide the list into two halves
        ListNode tmp  = null;
        while(fast != null && fast.next != null){
            tmp = slow;
            slow = slow.next;
            fast = fast.next.next;
        }
        tmp.next = null; // to end the first half

        // 2. Reverse the second half
        ListNode prev = null;
        ListNode curr = slow;  // Slow is the head for second half.
        ListNode next = null;
        while(curr != null){
            next = curr.next; // temporarily store next node
            curr.next = prev; // have prev appended at curr's next
            prev = curr; // make curr the prev for next iteration, at last when next would be null
            //prev will be the start node
            curr = next; // take curr to next using the temporary storage
        }
        ListNode revHead = prev;

        // 3. Merge the two LLs
        curr = head; // For first half
        tmp = null;
        while(curr != null && revHead != null){
            tmp = curr.next; // temporarily hold the remaining first half except current element

            curr.next = revHead; // make the current element point to head of reversed list
            curr = curr.next; // move curr forward

            revHead = revHead.next; // move revHead forward

            if(tmp == null && revHead != null) curr.next = revHead; // If size of main list is odd
            // first half would have n / 2 - 1 elements and get iterated first. In this case tmp would
become
            // null due to assignment on line 43 and reversed list would still have one item
            // so we assign it to curr.next
            else curr.next = tmp;
            curr = tmp;
        }
    }
}
----------------------------------------------------------Binary Tree Preorder Traversal
-----------------------------Given a binary tree, return the preorder traversal of its nodes'
values.
For example:
Given binary tree {1,#,2,3},
1
\
2
```

/
3

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

// using stack

```
class Solution
{
    public List<Integer> preorderTraversal(TreeNode root)
    {
        List<Integer> list=new ArrayList<>();//for storing the element inorder

        if(root == null)//base case when the tree is empty
            return list;

        Stack<TreeNode> stack= new Stack<>();

        stack.push(root);//pushing the root node

        while(!stack.isEmpty())//terminating condition
        {
            TreeNode temp=stack.pop();//popping the top element

            list.add(temp.val);//adding the value to the ArrayList //Root

            if(temp.right != null)//first pushing the right because to access the left first and then the
right //Right
                stack.push(temp.right);//pushing the right node

            if(temp.left != null)//pushing the left node if present after right because to access the left
first //Left
                stack.push(temp.left);
        }//the main purpose of pushing this way is to achive Root Left Right pattern of Inorder
Traversal
        return list;//returning the List of integer that are stored in Inorder fashion
    }
}
```

// recursion

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

```
 *    }
 * }
 */
class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> pre = new LinkedList<Integer>();
        preHelper(root,pre);
        return pre;
    }
    public void preHelper(TreeNode root, List<Integer> pre) {
        if(root==null) return;
        pre.add(root.val);
        preHelper(root.left,pre);
        preHelper(root.right,pre);
    }
}
```

Binary Tree Postorder Traversal

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

Example 1:

Input: root = [1,null,2,3]
Output: [3,2,1]

Example 2:

Input: root = []
Output: []

Example 3:

Input: root = [1]
Output: [1]

Example 4:

Input: root = [1,2]
Output: [2,1]

Example 5:

Input: root = [1,null,2]
Output: [2,1]

Constraints:

The number of the nodes in the tree is in the range [0, 100].
-100 <= Node.val <= 100

----------LRU Cache
----------Design a data structure that follows the constraints of a Least Recently Used (LRU) cache.

Implement the LRUCache class:

LRUCache(int capacity) Initialize the LRU cache with positive size capacity.
int get(int key) Return the value of the key if the key exists, otherwise return -1.
void put(int key, int value) Update the value of the key if the key exists. Otherwise, add the key-value pair to the cache. If the number of keys exceeds the capacity from this operation, evict the least recently used key.
The functions get and put must each run in O(1) average time complexity.

Example 1:

Input
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
Output
[null, null, null, 1, null, -1, null, -1, 3, 4]

Explanation
LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // cache is {1=1}
lRUCache.put(2, 2); // cache is {1=1, 2=2}
lRUCache.get(1);   // return 1
lRUCache.put(3, 3); // LRU key was 2, evicts key 2, cache is {1=1, 3=3}
lRUCache.get(2);   // returns -1 (not found)
lRUCache.put(4, 4); // LRU key was 1, evicts key 1, cache is {4=4, 3=3}
lRUCache.get(1);   // return -1 (not found)
lRUCache.get(3);   // return 3
lRUCache.get(4);   // return 4

Constraints:

1 <= capacity <= 3000
0 <= key <= 104
0 <= value <= 105
At most 2 * 105 calls will be made to get and put.

```
class LRUCache extends LinkedHashMap<Integer,Integer>{
    private int cap;
    public LRUCache(int capacity) {
        super(capacity,0.75f,true);
        //true for access-order
        cap=capacity;
    }

    public int get(int key){
        if(super.get(key)==null){
            return -1;
```

```
        }else{
            return super.get(key);
        }
        //or simply use this:
        //return getOrDefault(key, -1);
    }
    protected boolean removeEldestEntry(Map.Entry eldest) {
        return size() > cap;
     }
}
```

----------Insertion Sort List

----------Given the head of a singly linked list, sort the list using insertion sort, and return the sorted list's head.

The steps of the insertion sort algorithm:

Insertion sort iterates, consuming one input element each repetition and growing a sorted output list.
At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list and inserts it there.
It repeats until no input elements remain.
The following is a graphical example of the insertion sort algorithm. The partially sorted list (black) initially contains only the first element in the list. One element (red) is removed from the input data and inserted in-place into the sorted list with each iteration.

Example 1:

Input: head = [4,2,1,3]
Output: [1,2,3,4]
Example 2:

Input: head = [-1,5,3,4,0]
Output: [-1,0,3,4,5]

Constraints:

The number of nodes in the list is in the range [1, 5000].
-5000 <= Node.val <= 5000

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

```
 */
class Solution {
    public ListNode insertionSortList(ListNode head) {
        if (head == null) return head;
        ListNode pivot = new ListNode(Integer.MIN_VALUE), toInsert, preInsert;
        pivot.next = head;
        ListNode p = head;
        while (p.next != null) {
            if (p.val <= p.next.val) p = p.next;
            else {
                preInsert = pivot;
                toInsert = p.next;
                p.next = toInsert.next;
                while (preInsert.next.val <= toInsert.val) {
                    preInsert = preInsert.next;
                }
                toInsert.next = preInsert.next;
                preInsert.next = toInsert;
            }
        }
        return pivot.next;
    }
}
```

----------Sort List

----------Given the head of a linked list, return the list after sorting it in ascending order.

Example 1:

Input: head = [4,2,1,3]
Output: [1,2,3,4]
Example 2:

Input: head = [-1,5,3,4,0]
Output: [-1,0,3,4,5]
Example 3:

Input: head = []
Output: []

Constraints:

The number of nodes in the list is in the range [0, 5 * 104].
-105 <= Node.val <= 105

Follow up: Can you sort the linked list in O(n logn) time and O(1) memory (i.e. constant space)?

```
public class Solution {
```

```
  public ListNode sortList(ListNode head) {
    if (head == null || head.next == null)
      return head;

    // step 1. cut the list to two halves
    ListNode prev = null, slow = head, fast = head;

    while (fast != null && fast.next != null) {
      prev = slow;
      slow = slow.next;
      fast = fast.next.next;
    }

    prev.next = null;

    // step 2. sort each half
    ListNode l1 = sortList(head);
    ListNode l2 = sortList(slow);

    // step 3. merge l1 and l2
    return merge(l1, l2);
  }

  ListNode merge(ListNode l1, ListNode l2) {
    ListNode l = new ListNode(0), p = l;

    while (l1 != null && l2 != null) {
      if (l1.val < l2.val) {
        p.next = l1;
        l1 = l1.next;
      } else {
        p.next = l2;
        l2 = l2.next;
      }
      p = p.next;
    }

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

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

    return l.next;
  }

}
```

----------Max Points on a Line

----------Given an array of points where points[i] = [xi, yi] represents a point on the X-Y plane, return the maximum number of points that lie on the same straight line.

Example 1:

Input: points = [[1,1],[2,2],[3,3]]
Output: 3
Example 2:

Input: points = [[1,1],[3,2],[5,3],[4,1],[2,3],[1,4]]
Output: 4

Constraints:

1 <= points.length <= 300
points[i].length == 2
-104 <= xi, yi <= 104
All the points are unique.

```
class Solution {

    final private Map<Double, Integer> counts = new HashMap<>();
    final private static BiFunction<Integer, Integer, Integer> merger = (o, n) -> o + 1;

    public int maxPoints(int[][] points) {
        if (points.length == 1)
            return 1;
        int i, j, max = 0;
        double slope;
        for (i = 0; i < points.length - 1; i++) {
            counts.clear();
            for (j = i + 1; j < points.length; j++) {
                slope = getSlope(points[i], points[j]);
                max = Math.max(max, counts.merge(slope, 2, merger));
            }
        }
        return max;
    }

    private double getSlope(int[] p1, int[] p2) {
        double ret = ((p2[1] - p1[1]) * 1.0) / (p2[0] - p1[0]);
        if (ret == -0.0)
            return 0.0;
        if (ret == Double.NEGATIVE_INFINITY)
            return Double.POSITIVE_INFINITY;
        return ret;
    }
}
```

----------Evaluate Reverse Polish Notation
----------Evaluate the value of an arithmetic expression in Reverse Polish Notation.

Valid operators are +, -, *, and /. Each operand may be an integer or another expression.

Note that division between two integers should truncate toward zero.

It is guaranteed that the given RPN expression is always valid. That means the expression would always evaluate to a result, and there will not be any division by zero operation.

Example 1:

Input: tokens = ["2","1","+","3","*"]
Output: 9
Explanation: ((2 + 1) * 3) = 9
Example 2:

Input: tokens = ["4","13","5","/","+"]
Output: 6
Explanation: (4 + (13 / 5)) = 6
Example 3:

Input: tokens = ["10","6","9","3","+","-11","*","/","*","17","+","5","+"]
Output: 22
Explanation: ((10 * (6 / ((9 + 3) * -11))) + 17) + 5
= ((10 * (6 / (12 * -11))) + 17) + 5
= ((10 * (6 / -132)) + 17) + 5
= ((10 * 0) + 17) + 5
= (0 + 17) + 5
= 17 + 5
= 22

Constraints:

1 <= tokens.length <= 104
tokens[i] is either an operator: "+", "-", "*", or "/", or an integer in the range [-200, 200].

```
class Solution {
    public int evalRPN(String[] tokens) {
        Stack<Integer> stack= new Stack<Integer>();
        for(String s:tokens){
                switch(s){
                    case "+":
                        stack.push(stack.pop()+stack.pop());
                        break;
                    case "-":
                        stack.push(-stack.pop()+stack.pop());
                        break;
                    case "*":
                        stack.push(stack.pop()*stack.pop());
                        break;
                    case "/":
                        stack.push((int)((1.0/stack.pop())*stack.pop()));
                        break;
                    default:
                        stack.push(Integer.parseInt(s));
```

```
            }
        }
        return stack.pop();
    }
}
```

----------Reverse Words in a String
----------Given an input string s, reverse the order of the words.

A word is defined as a sequence of non-space characters. The words in s will be separated by at least one space.

Return a string of the words in reverse order concatenated by a single space.

Note that s may contain leading or trailing spaces or multiple spaces between two words. The returned string should only have a single space separating the words. Do not include any extra spaces.

Example 1:

Input: s = "the sky is blue"
Output: "blue is sky the"
Example 2:

Input: s = " hello world "
Output: "world hello"
Explanation: Your reversed string should not contain leading or trailing spaces.
Example 3:

Input: s = "a good   example"
Output: "example good a"
Explanation: You need to reduce multiple spaces between two words to a single space in the reversed string.
Example 4:

Input: s = " Bob   Loves Alice   "
Output: "Alice Loves Bob"
Example 5:

Input: s = "Alice does not even like bob"
Output: "bob like even not does Alice"

Constraints:

1 <= s.length <= 104
s contains English letters (upper-case and lower-case), digits, and spaces ' '.
There is at least one word in s.

```
class Solution {
    public String reverseWords(String s) {
      if(s == null || s.length() <2) return s;
```

```
    int l = 0, r = s.length() -1;
    char [] charArr = s.toCharArray();
    reverseWord(charArr, l, r);


    while(l <= r){
        int start = l;
        while(start <= r && charArr[start] == ' ') start++;
        int end = start;
        while(end <= r && charArr[end] != ' ') end ++;
        //System.out.println(start +" "+ end);
        if(start == end) break;

        reverseWord(charArr, start, end-1);

        l = end;
    }


    int i =0, j =0;
    while(j <= r){
        while(j<=r && charArr[j] == ' ') j++;
        while(j<=r && charArr[j] != ' '){
            charArr[i++] =charArr[j++];
        }
        if(i<r){
            charArr[i++] = ' ';
        }
    }

    return (new String(charArr, 0, i)).trim();
}

public void reverseWord(char[] charArr, int l, int r){
    if(r - l < 1) return;

    while(l<r){
        char temp = charArr[l];
        charArr[l] = charArr[r];
        charArr[r] = temp;
        l++;
        r--;
    }
    }
}
```

----------Maximum Product Subarray
----------Given an integer array nums, find a contiguous non-empty subarray within the array that has the largest product, and return the product.

It is guaranteed that the answer will fit in a 32-bit integer.

A subarray is a contiguous subsequence of the array.

Example 1:

Input: nums = [2,3,-2,4]
Output: 6
Explanation: [2,3] has the largest product 6.
Example 2:

Input: nums = [-2,0,-1]
Output: 0
Explanation: The result cannot be 2, because [-2,-1] is not a subarray.

Constraints:

1 <= nums.length <= 2 * 104
-10 <= nums[i] <= 10
The product of any prefix or suffix of nums is guaranteed to fit in a 32-bit integer.

```
/*
idea is to keep 3 variables
1. max -> maximum product ending at a[i]
2. min -> minimum product ending at a[i]
3. ans -> maximum product subarray

Ex.
num = [3, 2, -1,   5, -2]
min = [3, 2, -6, -30, -2]
max = [3, 6, -1,   5, 60]
ans = [3, 6,  6,   6, 60]
*/
class Solution {
    public int maxProduct(int[] nums) {
        if(nums==null||nums.length==0) return 0;
        int max = nums[0];
        int min = nums[0];
        int ans = nums[0];
        int temp;
        for(int i=1;i<nums.length;i++) {
            if(nums[i]<0) {
                temp = max;
                max = min;
                min = temp;
            }
            max = Integer.max(nums[i],nums[i]*max);
            min = Integer.min(nums[i],nums[i]*min);
            ans = Integer.max(ans,max);
        }
        return ans;
    }
}
```

----------Find Minimum in Rotated Sorted Array

------------------------------Suppose an array of length n sorted in ascending order is rotated between 1 and n times. For example, the array nums = [0,1,2,4,5,6,7] might become:

[4,5,6,7,0,1,2] if it was rotated 4 times.
[0,1,2,4,5,6,7] if it was rotated 7 times.
Notice that rotating an array [a[0], a[1], a[2], ..., a[n-1]] 1 time results in the array [a[n-1], a[0], a[1], a[2], ..., a[n-2]].

Given the sorted rotated array nums of unique elements, return the minimum element of this array.

You must write an algorithm that runs in O(log n) time.

Example 1:

Input: nums = [3,4,5,1,2]
Output: 1
Explanation: The original array was [1,2,3,4,5] rotated 3 times.
Example 2:

Input: nums = [4,5,6,7,0,1,2]
Output: 0
Explanation: The original array was [0,1,2,4,5,6,7] and it was rotated 4 times.
Example 3:

Input: nums = [11,13,15,17]
Output: 11
Explanation: The original array was [11,13,15,17] and it was rotated 4 times.

Constraints:

n == nums.length
1 <= n <= 5000
-5000 <= nums[i] <= 5000
All the integers of nums are unique.
nums is sorted and rotated between 1 and n times.

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

----------Find Minimum in Rotated Sorted Array II
----------Suppose an array of length n sorted in ascending order is rotated between 1 and n times. For example, the array nums = [0,1,4,4,5,6,7] might become:

[4,5,6,7,0,1,4] if it was rotated 4 times.
[0,1,4,4,5,6,7] if it was rotated 7 times.
Notice that rotating an array [a[0], a[1], a[2], ..., a[n-1]] 1 time results in the array [a[n-1], a[0], a[1], a[2], ..., a[n-2]].

Given the sorted rotated array nums that may contain duplicates, return the minimum element of this array.

You must decrease the overall operation steps as much as possible.

Example 1:

Input: nums = [1,3,5]
Output: 1
Example 2:

Input: nums = [2,2,2,0,1]
Output: 0

Constraints:

n == nums.length
1 <= n <= 5000
-5000 <= nums[i] <= 5000
nums is sorted and rotated between 1 and n times.

```
class Solution {
    public int findMin(int[] nums) {
        int n=nums.length;
        int i=0,j=n-1;
        while(i<j){

            while(i+1<n && nums[i]==nums[i+1]) i++;
            while(j-1>=0 && nums[j-1]==nums[j]) j--;

            int mid=i+(j-i)/2;
            if(nums[j]<nums[mid]) i=mid+1;
            else j=mid;
        }
        return nums[i];
    }
}
```

----------Min Stack

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

Implement the MinStack class:

MinStack() initializes the stack object.
void push(int val) pushes the element val onto the stack.
void pop() removes the element on the top of the stack.
int top() gets the top element of the stack.
int getMin() retrieves the minimum element in the stack.

Example 1:

Input
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]

Output
[null,null,null,null,-3,null,0,-2]

Explanation
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // return -3
minStack.pop();
minStack.top();   // return 0
minStack.getMin(); // return -2

Constraints:

-231 <= val <= 231 - 1
Methods pop, top and getMin operations will always be called on non-empty stacks.
At most 3 * 104 calls will be made to push, pop, top, and getMin.

----------Intersection of Two Linked Lists
----------Write a program to find the node at which the intersection of two singly linked lists begins.
For example, the following two linked lists:
A:

a1 → a2
↘
c1 → c2 → c3
↗

B:

b1 → b2 → b3

begin to intersect at node c1.
Notes:
If the two linked lists have no intersection at all, return null.
The linked lists must retain their original structure after the function returns.
You may assume there are no cycles anywhere in the entire linked structure.
Your code should preferably run in O(n) time and use only O(1) memory.

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

```
class MinStack {

    /** initialize your data structure here. */
    Stack<Integer> stack;
    int min = Integer.MAX_VALUE;
    public MinStack() {
        stack = new Stack<>();
    }

    public void push(int x) {
        stack.push(x);
        if(min > x){
            min = x;
        }
    }

    public void pop() {
        int x = stack.pop();
        if(x == min){
            min = Integer.MAX_VALUE;
            for(int i=0; i <stack.size();i++){
                int temp = stack.get(i);
                 if(min > temp){
                     min = temp;
                 }
            }
        }

    }

    public int top() {
        return stack.peek();
    }

    public int getMin() {
        return min;
    }
}


/**
```

```
 * 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();
 */
```

----------Find Peak Element----------A peak element is an element that is strictly greater than its neighbors.

Given an integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to any of the peaks.

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

You must write an algorithm that runs in O(log n) time.

Example 1:

Input: nums = [1,2,3,1]
Output: 2
Explanation: 3 is a peak element and your function should return the index number 2.
Example 2:

Input: nums = [1,2,1,3,5,6,4]
Output: 5
Explanation: Your function can return either index number 1 where the peak element is 2, or index number 5 where the peak element is 6.

Constraints:

1 <= nums.length <= 1000
-231 <= nums[i] <= 231 - 1
nums[i] != nums[i + 1] for all valid i

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

```
    }
}
```
----------Maximum Gap----------Given an integer array nums, return the maximum difference between two successive elements in its sorted form. If the array contains less than two elements, return 0.

You must write an algorithm that runs in linear time and uses linear extra space.

Example 1:

Input: nums = [3,6,9,1]
Output: 3
Explanation: The sorted form of the array is [1,3,6,9], either (3,6) or (6,9) has the maximum difference 3.
Example 2:

Input: nums = [10]
Output: 0
Explanation: The array contains less than 2 elements, therefore return 0.

Constraints:

1 <= nums.length <= 105
0 <= nums[i] <= 109

//bucket sort

```
class Solution {
    public int maximumGap(int[] nums) {
        if (nums.length==1) return 0;
        int min=Integer.MAX_VALUE, max=Integer.MIN_VALUE;
        int n=nums.length;
        for (int num: nums) {
            if (min>num) min=num;
            if (max<num) max=num;
        }
        if (min==max) return 0;
        int bucketSize = Math.max(1, (max-min)/(n-1));
        int bucketNum = (max - min) / bucketSize + 1;
        int[] mins = new int[bucketNum];
        int[] maxs = new int[bucketNum];
        Arrays.fill(mins, Integer.MAX_VALUE);
        Arrays.fill(maxs, Integer.MIN_VALUE);
        for (int num: nums) {
            int bucketIndex = (num-min)/bucketSize;
            mins[bucketIndex]=Math.min(mins[bucketIndex], num);
            maxs[bucketIndex]=Math.max(maxs[bucketIndex], num);
        }
        int maxDiff=0;
        int prev = min;
```

```
        for (int i=0; i<bucketNum; i++) {
            if (mins[i]==Integer.MAX_VALUE) continue;
            maxDiff = Math.max(maxDiff, mins[i]-prev);
            maxDiff = Math.max(maxDiff, maxs[i]-mins[i]);
            prev=maxs[i];
        }
        return maxDiff;
    }
}
```

----------Compare Version Numbers

----------Given two version numbers, version1 and version2, compare them.

Version numbers consist of one or more revisions joined by a dot '.'. Each revision consists of digits and may contain leading zeros. Every revision contains at least one character. Revisions are 0-indexed from left to right, with the leftmost revision being revision 0, the next revision being revision 1, and so on. For example 2.5.33 and 0.1 are valid version numbers.

To compare version numbers, compare their revisions in left-to-right order. Revisions are compared using their integer value ignoring any leading zeros. This means that revisions 1 and 001 are considered equal. If a version number does not specify a revision at an index, then treat the revision as 0. For example, version 1.0 is less than version 1.1 because their revision 0s are the same, but their revision 1s are 0 and 1 respectively, and 0 < 1.

Return the following:

If version1 < version2, return -1.
If version1 > version2, return 1.
Otherwise, return 0.

Example 1:

Input: version1 = "1.01", version2 = "1.001"
Output: 0
Explanation: Ignoring leading zeroes, both "01" and "001" represent the same integer "1".
Example 2:

Input: version1 = "1.0", version2 = "1.0.0"
Output: 0
Explanation: version1 does not specify revision 2, which means it is treated as "0".
Example 3:

Input: version1 = "0.1", version2 = "1.1"
Output: -1
Explanation: version1's revision 0 is "0", while version2's revision 0 is "1". 0 < 1, so version1 < version2.
Example 4:

Input: version1 = "1.0.1", version2 = "1"
Output: 1
Example 5:

Input: version1 = "7.5.2.4", version2 = "7.5.3"
Output: -1

Constraints:

1 <= version1.length, version2.length <= 500
version1 and version2 only contain digits and '.'.
version1 and version2 are valid version numbers.
All the given revisions in version1 and version2 can be stored in a 32-bit integer.

```
class Solution {
    public int compareVersion(String v1, String v2) {
        for(int i = 0, j = 0; i < v1.length() || j < v2.length(); i++,j++) {
            int num1 = 0, num2 = 0;
            while(i < v1.length() && v1.charAt(i) != '.') {
                num1 = (num1 * 10) + v1.charAt(i) - '0';
                i++;
            }
            while(j < v2.length() && v2.charAt(j) != '.') {
                num2 = (num2 * 10) + v2.charAt(j) - '0';
                j++;
            }
            if(num1 < num2) return -1;
            else if(num1 > num2) return 1;
        }
        return 0;
    }
}
```

----------Fraction to Recurring Decimal
----------Given two integers representing the numerator and denominator of a fraction, return the fraction in string format.

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

If multiple answers are possible, return any of them.

It is guaranteed that the length of the answer string is less than 104 for all the given inputs.

Example 1:

Input: numerator = 1, denominator = 2
Output: "0.5"
Example 2:

Input: numerator = 2, denominator = 1
Output: "2"
Example 3:

Input: numerator = 2, denominator = 3
Output: "0.(6)"
Example 4:

Input: numerator = 4, denominator = 333

Output: "0.(012)"
Example 5:

Input: numerator = 1, denominator = 5
Output: "0.2"

Constraints:

-231 <= numerator, denominator <= 231 - 1
denominator != 0

```
public class Solution {
       public String fractionToDecimal(int numerator, int denominator) {
          //to avoid overflow
          long numeratorl=numerator, denominatorl=denominator;
          StringBuffer buffer=new StringBuffer();
          //handle negetives
          if(numeratorl<0&&denominatorl>0)buffer.append('-');
          else if(numeratorl>0&&denominatorl<0)buffer.append('-');
          numeratorl=Math.abs(numeratorl);denominatorl=Math.abs(denominatorl);
          //map,key:  numerator ,because denominator never changes;value:position of
numerator/denominator
          HashMap<Long, Integer>map=new HashMap<>();
          //handle integer part
          long res=numeratorl/denominatorl;
          numeratorl=(numeratorl%denominatorl)*10;
          buffer.append(res);
          if(numeratorl!=0)buffer.append('.');
          //handle flaot part
          while(numeratorl != 0){
             res=numeratorl/denominatorl;
             if(map.get(numeratorl)!=null)
             {
                //handle repaeting part
                buffer.insert(map.get(numeratorl).intValue(), '(');
                buffer.append(')');
                break;
             }
             map.put(numeratorl, buffer.length());
             numeratorl=(numeratorl%denominatorl)*10;
             buffer.append(res);
          }
          return buffer.toString();
       }
   }
```

----------Two Sum II - Input array is sorted----------Given a 1-indexed array of integers numbers that is already sorted in non-decreasing order, find two numbers such that they add up to a specific target number. Let these two numbers be numbers[index1] and numbers[index2] where 1 <= index1 < index2 <= numbers.length.

Return the indices of the two numbers, index1 and index2, added by one as an integer array [index1, index2] of length 2.

The tests are generated such that there is exactly one solution. You may not use the same element twice.

Example 1:

Input: numbers = [2,7,11,15], target = 9
Output: [1,2]
Explanation: The sum of 2 and 7 is 9. Therefore, index1 = 1, index2 = 2. We return [1, 2].
Example 2:

Input: numbers = [2,3,4], target = 6
Output: [1,3]
Explanation: The sum of 2 and 4 is 6. Therefore index1 = 1, index2 = 3. We return [1, 3].
Example 3:

Input: numbers = [-1,0], target = -1
Output: [1,2]
Explanation: The sum of -1 and 0 is -1. Therefore index1 = 1, index2 = 2. We return [1, 2].

Constraints:

2 <= numbers.length <= 3 * 104
-1000 <= numbers[i] <= 1000
numbers is sorted in non-decreasing order.
-1000 <= target <= 1000
The tests are generated such that there is exactly one solution.

```
class Solution {
    public int[] twoSum(int[] numbers, int target) {
    int start=0,end=numbers.length-1;
        while(start<end){
            if(numbers[start]+numbers[end]==target)
                return new int[]{start+1,end+1};
            if(numbers[start]+numbers[end]>target)
                end--;
            else
                start++;
        }
        return null;
    }
}
```

----------Excel Sheet Column Title
----------Given an integer columnNumber, return its corresponding column title as it appears in an Excel sheet.

For example:

A -> 1
B -> 2
C -> 3

...
Z -> 26
AA -> 27
AB -> 28
...

Example 1:

Input: columnNumber = 1
Output: "A"
Example 2:

Input: columnNumber = 28
Output: "AB"
Example 3:

Input: columnNumber = 701
Output: "ZY"
Example 4:

Input: columnNumber = 2147483647
Output: "FXSHRXW"

Constraints:

1 <= columnNumber <= 231 - 1

```
class Solution {
    public String convertToTitle(int columnNumber) {
        StringBuilder sb=new StringBuilder();
       while(columnNumber>0) {
           int rem=columnNumber%26;
           if(rem==0){
               sb.append('Z');
               columnNumber=(columnNumber/26)-1;
           }
           else
           {
              sb.append((char)('A'+rem-1));
              columnNumber/=26;
           }
       }
        return sb.reverse().toString();
    }
}
```

----------Majority Element
----------Given an array nums of size n, return the majority element.

The majority element is the element that appears more than ⌊n / 2⌋ times. You may assume that the majority element always exists in the array.

Example 1:

Input: nums = [3,2,3]
Output: 3
Example 2:

Input: nums = [2,2,1,1,1,2,2]
Output: 2

Constraints:

n == nums.length
1 <= n <= 5 * 104
-231 <= nums[i] <= 231 - 1

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

----------Excel Sheet Column Number----------Given a string columnTitle that represents the 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
...

Example 1:

Input: columnTitle = "A"
Output: 1

Example 2:

Input: columnTitle = "AB"
Output: 28
Example 3:

Input: columnTitle = "ZY"
Output: 701
Example 4:

Input: columnTitle = "FXSHRXW"
Output: 2147483647

Constraints:

1 <= columnTitle.length <= 7
columnTitle consists only of uppercase English letters.
columnTitle is in the range ["A", "FXSHRXW"].

```
class Solution {
    public int titleToNumber(String s) {

        int ans = 0;

        for(char c: s.toCharArray()) {

            ans *= 26;

            ans += (c - 'A' + 1);
        }
        return ans;
    }
}
```

----------Factorial Trailing Zeroes
----------Given an integer n, return the number of trailing zeroes in n!.

Note that n! = n * (n - 1) * (n - 2) * ... * 3 * 2 * 1.

Example 1:

Input: n = 3
Output: 0
Explanation: 3! = 6, no trailing zero.
Example 2:

Input: n = 5
Output: 1
Explanation: 5! = 120, one trailing zero.
Example 3:

Input: n = 0

Output: 0

Constraints:

0 <= n <= 104

Follow up: Could you write a solution that works in logarithmic time complexity?

```
public class Solution {
    public int trailingZeroes(int n) {
        int r = 0;
        while (n > 0) {
            n /= 5;
            r += n;
        }
        return r;
    }
}
```

----------Binary Search Tree Iterator----------Implement the BSTIterator class that represents an iterator over the in-order traversal of a binary search tree (BST):

BSTIterator(TreeNode root) Initializes an object of the BSTIterator class. The root of the BST is given as part of the constructor. The pointer should be initialized to a non-existent number smaller than any element in the BST.
boolean hasNext() Returns true if there exists a number in the traversal to the right of the pointer, otherwise returns false.
int next() Moves the pointer to the right, then returns the number at the pointer.
Notice that by initializing the pointer to a non-existent smallest number, the first call to next() will return the smallest element in the BST.

You may assume that next() calls will always be valid. That is, there will be at least a next number in the in-order traversal when next() is called.

Example 1:

Input
["BSTIterator", "next", "next", "hasNext", "next", "hasNext", "next", "hasNext", "next", "hasNext"]
[[[7, 3, 15, null, null, 9, 20]], [], [], [], [], [], [], [], [], []]
Output
[null, 3, 7, true, 9, true, 15, true, 20, false]

Explanation
BSTIterator bSTIterator = new BSTIterator([7, 3, 15, null, null, 9, 20]);
bSTIterator.next();   // return 3
bSTIterator.next();   // return 7
bSTIterator.hasNext(); // return True
bSTIterator.next();   // return 9

```
bSTIterator.hasNext(); // return True
bSTIterator.next();    // return 15
bSTIterator.hasNext(); // return True
bSTIterator.next();    // return 20
bSTIterator.hasNext(); // return False
```

Constraints:

The number of nodes in the tree is in the range [1, 105].
0 <= Node.val <= 106
At most 105 calls will be made to hasNext, and next.

Follow up:

Could you implement next() and hasNext() to run in average O(1) time and use O(h) memory, where h is the height of the tree?

```
public class BSTIterator {

    private TreeNode root;

    public BSTIterator(TreeNode root) {
        this.root = root;
    }

    /** @return the next smallest number */
    public int next() {
        return nextHelper(root, root);
    }

    private int nextHelper(TreeNode parent, TreeNode node) {
        // a node without a left child is the smallest one -> O(h)
        while (node.left != null) {
            parent = node;
            node = node.left;
        }

        // extract the value
        final int res = node.val;

        // remove the smallest node
        if (node.right != null) {
            node.val = node.right.val;
            node.left = node.right.left;
            node.right = node.right.right;
        }
        else {
            // no parent or a right child => the last node
            if (parent == node) root = null;
            // the parent exist => node must be the left child
            else parent.left = null;
        }
```

```
        return res;
    }

    /** @return whether we have a next smallest number */
    public boolean hasNext() {
        return root!=null;
    }
}
```

----------Dungeon Game

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

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

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

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

Return the knight's minimum initial health so that he can rescue the princess.

Note that any room can contain threats or power-ups, even the first room the knight enters and the bottom-right room where the princess is imprisoned.

Example 1:

Input: dungeon = [[-2,-3,3],[-5,-10,1],[10,30,-5]]
Output: 7
Explanation: The initial health of the knight must be at least 7 if he follows the optimal path: RIGHT-> RIGHT -> DOWN -> DOWN.
Example 2:

Input: dungeon = [[0]]
Output: 1

Constraints:

m == dungeon.length
n == dungeon[i].length
1 <= m, n <= 200
-1000 <= dungeon[i][j] <= 1000

```
public class DungeonGame {
    public int calculateMinimumHP(int[][] dungeon) {
        if(dungeon == null || dungeon.length == 0 || dungeon[0].length == 0){//[],[[]]
```

```
            return 0;
        }
        int[][] flag = new int[dungeon.length][dungeon[0].length];
        int min = dfs(dungeon, flag, 0, 0);
        return min;
    }

    private int dfs(int[][] dungeon, int[][] flag, int x, int y){
        if(flag[x][y] != 0){
            return flag[x][y];
        }
        if(x == dungeon.length - 1 && y == dungeon[0].length - 1){//The down-right corner
            flag[x][y] = dungeon[x][y] < 0 ? -dungeon[x][y] + 1 : 1; //The minimum is 1
            return flag[x][y];
        }
        int min = Integer.MAX_VALUE;
        //go down
        if(x < dungeon.length - 1){
            int down = dfs(dungeon, flag, x + 1, y);
            min = min < down ? min : down;
        }
        //go right
        if(y < dungeon[0].length - 1){
            int right = dfs(dungeon, flag, x, y + 1);
            min = min < right ? min : right;
        }
        if(dungeon[x][y] >= min){//If min is 6, dungeon[x][y] if 10, then min should be updated to 1
            min = 1;
        }else{//If min is 6, dungeon[x][y] is 3 or -3, then min should be updated to 3 or 9
            min = min - dungeon[x][y];
        }
        flag[x][y] = min;
        return min;
    }
}
```

----------Largest Number

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

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

Example 1:

Input: nums = [10,2]
Output: "210"
Example 2:

Input: nums = [3,30,34,5,9]
Output: "9534330"
Example 3:

Input: nums = [1]

Output: "1"
Example 4:

Input: nums = [10]
Output: "10"

Constraints:

1 <= nums.length <= 100
0 <= nums[i] <= 109

```
// comparator
class Solution {
    public String largestNumber(int[] nums) {
        if(nums.length == 0)
            return "";
        List<String> strs = new ArrayList();
        for(int n : nums){
            strs.add(Integer.toString(n));
        }
        Collections.sort(strs,new Comparator<String>(){
            public int compare(String a,String b){
                String x = a+b;
                String y = b+a;
                return y.compareTo(x);
            }
        });
        if(strs.get(0).equals("0"))
            return "0";
        StringBuffer sb = new StringBuffer();
        for(String str : strs){
            sb.append(str);
        }
        return new String(sb);
    }
}

// 2nd solution - best
// Custom sort numbers: Ot(n * log2(n)), Os(n)
class Solution {
public static String largestNumber(int[] nums) {
    assert nums != null && nums.length > 0;

    String[] strs = new String[nums.length];
    for (int i = 0; i < nums.length; i++) {
        strs[i] = String.valueOf(nums[i]);
    }

    Arrays.sort(strs, new NumberStringComparator1());   // With recursive comparison
    // Arrays.sort(strs, new NumberStringComparator2());    // With concatenated comparison

    // Skip leading zeroes
    if (strs[0].equals("0")) {
```

```
        return "0";
    }

    // Concatenate number strings
    StringBuilder sb = new StringBuilder();
    for (String s : strs) {
        sb.append(s);
    }

    return sb.toString();
}

static class NumberStringComparator1 implements Comparator<String> {

    @Override
    public int compare(String s1, String s2) {
        int index1 = 0, index2 = 0;
        while (index1 < s1.length() && index2 < s2.length()) {
            int digit1 = s1.charAt(index1) - '0', digit2 = s2.charAt(index2) - '0';
            if (digit1 != digit2) {
                return Integer.compare(digit2, digit1);
            }

            index1++;
            index2++;
        }

        // KEY: Recursive comparison, if prefixes are identical
        if (index1 < s1.length()) {
            return compare(s1.substring(index1), s2);
        } else if (index2 < s2.length()) {
            return compare(s1, s2.substring(index2));
        } else {
            return 0;
        }
    }
}

static class NumberStringComparator2 implements Comparator<String> {

    @Override
    public int compare(String s1, String s2) {
        return (s2 + s1).compareTo(s1 + s2);
    }
}
}
```

----------Repeated DNA Sequences
----------The DNA sequence is composed of a series of nucleotides abbreviated as 'A', 'C', 'G', and 'T'.

For example, "ACGAATTCCG" is a DNA sequence.
When studying DNA, it is useful to identify repeated sequences within the DNA.

Given a string s that represents a DNA sequence, return all the 10-letter-long sequences (substrings) that occur more than once in a DNA molecule. You may return the answer in any order.

Example 1:

Input: s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT"
Output: ["AAAAACCCCC","CCCCCAAAAA"]
Example 2:

Input: s = "AAAAAAAAAAAAA"
Output: ["AAAAAAAAAA"]

Constraints:

1 <= s.length <= 105
s[i] is either 'A', 'C', 'G', or 'T'.

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

----------Best Time to Buy and Sell Stock IV----------You are given an integer array prices where prices[i] is the price of a given stock on the ith day, and an integer k.

Find the maximum profit you can achieve. You may complete at most k transactions.

Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

Example 1:

Input: k = 2, prices = [2,4,1]
Output: 2
Explanation: Buy on day 1 (price = 2) and sell on day 2 (price = 4), profit = 4-2 = 2.
Example 2:

Input: k = 2, prices = [3,2,6,5,0,3]
Output: 7
Explanation: Buy on day 2 (price = 2) and sell on day 3 (price = 6), profit = 6-2 = 4. Then buy on day 5 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.

Constraints:

0 <= k <= 100
0 <= prices.length <= 1000
0 <= prices[i] <= 1000

```
class Solution {
/**
 * dp[i, j] represents the max profit up until prices[j] using at most i transactions.
 * dp[i, j] = max(dp[i, j-1], prices[j] - prices[jj] + dp[i-1, jj]) { jj in range of [0, j-1] }
 *          = max(dp[i, j-1], prices[j] + max(dp[i-1, jj] - prices[jj]))
 * dp[0, j] = 0; 0 transactions makes 0 profit
 * dp[i, 0] = 0; if there is only one price data point you can't make any transaction.
 */

public int maxProfit(int k, int[] prices) {
    int n = prices.length;
    if (n <= 1)
        return 0;

    //if k >= n/2, then you can make maximum number of transactions.
    if (k >=  n/2) {
        int maxPro = 0;
        for (int i = 1; i < n; i++) {
            if (prices[i] > prices[i-1])
                maxPro += prices[i] - prices[i-1];
        }
        return maxPro;
    }

    int[][] dp = new int[k+1][n];
    for (int i = 1; i <= k; i++) {
        int localMax = dp[i-1][0] - prices[0];
        for (int j = 1; j < n; j++) {
            dp[i][j] = Math.max(dp[i][j-1],  prices[j] + localMax);
            localMax = Math.max(localMax, dp[i-1][j] - prices[j]);
        }
    }
    return dp[k][n-1];
}
}
```

----------Rotate Array
----------Given an array, rotate the array to the right by k steps, where k is non-negative.

Example 1:

Input: nums = [1,2,3,4,5,6,7], k = 3
Output: [5,6,7,1,2,3,4]
Explanation:
rotate 1 steps to the right: [7,1,2,3,4,5,6]

rotate 2 steps to the right: [6,7,1,2,3,4,5]
rotate 3 steps to the right: [5,6,7,1,2,3,4]
Example 2:

Input: nums = [-1,-100,3,99], k = 2
Output: [3,99,-1,-100]
Explanation:
rotate 1 steps to the right: [99,-1,-100,3]
rotate 2 steps to the right: [3,99,-1,-100]

Constraints:

1 <= nums.length <= 105
-231 <= nums[i] <= 231 - 1
0 <= k <= 105

Follow up:

Try to come up with as many solutions as you can. There are at least three different ways to solve this problem.
Could you do it in-place with O(1) extra space?

```
/*
      Array Rotation Using Reversal Algorithm
      Step 1: reverse the Array from indices 0 to length-1
      Step 2: reverse the Array from indices 0 to k-1
      Step 3: reverse the Array from indices 0 to length-1

      Example:
        let  arr is --> [ 1, 2, 3, 4, 5 ]      k = 2

        index -->     0   k-1  k  3  length-1
                    [ 1,  2,  3,  4,  5 ]

         Reversing from 0 to length -1
                We get     [5,4,3,2,1]
         Reversing from 0 to k-1
                We get     [4, 5, 3, 2, 1]
         Reversing from k to length-1
                We get     [4, 5, 1, 2, 3]   --> RESULTANT Rotated Array
      */
class Solution {
   public void rotate(int[] nums, int k) {
      k = k % nums.length;
      reverse(nums, 0, nums.length-1);
      reverse(nums, 0, k-1);
      reverse(nums, k, nums.length-1);
   }

   static void reverse(int [] nums, int start, int end){
      // reversing the Array Using two pointer Method
      while(start < end){
```

```
            int temp = nums[start];
            nums[start] = nums[end];
            nums[end] = temp;
            start++;
            end--;
        }
    }
}
```

----------Reverse Bits

----------Reverse bits of a given 32 bits unsigned integer.

Note:

Note that in some languages, such as Java, there is no unsigned integer type. In this case, both input and output will be given as a signed integer type. They should not affect your implementation, as the integer's internal binary representation is the same, whether it is signed or unsigned.
In Java, the compiler represents the signed integers using 2's complement notation. Therefore, in Example 2 above, the input represents the signed integer -3 and the output represents the signed integer -1073741825.

Example 1:

Input: n = 00000010100101000001111010011100
Output: 964176192 (00111001011110000010100101000000)
Explanation: The input binary string 00000010100101000001111010011100 represents the unsigned integer 43261596, so return 964176192 which its binary representation is 00111001011110000010100101000000.
Example 2:

Input: n = 11111111111111111111111111111101
Output: 3221225471 (10111111111111111111111111111111)
Explanation: The input binary string 11111111111111111111111111111101 represents the unsigned integer 4294967293, so return 3221225471 which its binary representation is 10111111111111111111111111111111.

Constraints:

The input must be a binary string of length 32

```
public class Solution {
    // you need treat n as an unsigned value
    public int reverseBits(int n) {
        int ret=0;
        // because there are 32 bits in total
        for (int i = 0; i <32;i++){
            ret = ret<<1;
            // If the bit is 1 we OR it with 1, ie add 1
            if((n & 1) > 0){
                ret = ret | 1;
            }
            n=n>>>1;
```

```
        }
        return ret;
    }
}
```

----------Number of 1 Bits----------

Write a function that takes an unsigned integer and returns the number of '1' bits it has (also known as the Hamming weight).

Note:

Note that in some languages, such as Java, there is no unsigned integer type. In this case, the input will be given as a signed integer type. It should not affect your implementation, as the integer's internal binary representation is the same, whether it is signed or unsigned.
In Java, the compiler represents the signed integers using 2's complement notation. Therefore, in Example 3, the input represents the signed integer. -3.

Example 1:

Input: n = 00000000000000000000000000001011
Output: 3
Explanation: The input binary string 00000000000000000000000000001011 has a total of three '1' bits.
Example 2:

Input: n = 00000000000000000000000010000000
Output: 1
Explanation: The input binary string 00000000000000000000000010000000 has a total of one '1' bit.
Example 3:

Input: n = 11111111111111111111111111111101
Output: 31
Explanation: The input binary string 11111111111111111111111111111101 has a total of thirty one '1' bits.

Constraints:

The input must be a binary string of length 32.

```
public class Solution {
    // you need to treat n as an unsigned value
    public int hammingWeight(int n) {
        //System.out.println(n);
        int count = 0;
        while (n != 0) {
            if ((n | 1) == n) {
                count++;
            }
            n = n >>> 1;
            //System.out.println(n);
        }
        return count;
```

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

Given an integer array nums representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

Example 1:

Input: nums = [1,2,3,1]
Output: 4
Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).
Total amount you can rob = 1 + 3 = 4.
Example 2:

Input: nums = [2,7,9,3,1]
Output: 12
Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1).
Total amount you can rob = 2 + 9 + 1 = 12.

Constraints:

1 <= nums.length <= 100
0 <= nums[i] <= 400

```
class Solution {
    public int rob(int[] nums) {
        int n = nums.length;

        if (n == 0) return 0;
        else if (n == 1) return nums[0];

        int[] sol = new int[n];
        sol[0] = nums[0];
        sol[1] = Math.max(nums[0], nums[1]);

        for (int i = 2; i < n; i++) {
            sol[i] = Math.max(sol[i-2] + nums[i], sol[i-1]);
        }
        return sol[n-1];
    }
}
```
----------Binary Tree Right Side View
----------Given the root of 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.

Example 1:

Input: root = [1,2,3,null,5,null,4]
Output: [1,3,4]
Example 2:

Input: root = [1,null,3]
Output: [1,3]
Example 3:

Input: root = []
Output: []

Constraints:

The number of nodes in the tree is in the range [0, 100].
-100 <= Node.val <= 100

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
public List<Integer> rightSideView(TreeNode root) {
    List<Integer> ans = new ArrayList<>();

    addRightmost(root, ans, 0);

    return ans;
}

private void addRightmost(TreeNode root, List<Integer> ans, int level) {
    if (root == null)
        return;

    if (ans.size() == level)    // since we're adding 1 node per level, and List is 0 indexed,
        ans.add(root.val);      // we start at level 0 and add node val when equal

    addRightmost(root.right, ans, level + 1);   // go right side first
    addRightmost(root.left, ans, level + 1);
```

}
}
----------Number of Islands
----------Given an m x n 2D binary grid grid which represents a map of '1's (land) and '0's (water), return 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:

Input: grid = [
  ["1","1","1","1","0"],
  ["1","1","0","1","0"],
  ["1","1","0","0","0"],
  ["0","0","0","0","0"]
]
Output: 1
Example 2:

Input: grid = [
  ["1","1","0","0","0"],
  ["1","1","0","0","0"],
  ["0","0","1","0","0"],
  ["0","0","0","1","1"]
]
Output: 3

Constraints:

m == grid.length
n == grid[i].length
1 <= m, n <= 300
grid[i][j] is '0' or '1'

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


      return ans;

    }
```

```
    void BFS(int i, int j, char[][] grid) {
        if(i<0 || i>=grid.length || j<0 || j>=grid[i].length || grid[i][j]=='0') return;

        grid[i][j] = '0';
        BFS(i+1,j, grid);
        BFS(i-1,j, grid);
        BFS(i,j+1, grid);
        BFS(i,j-1, grid);


    }
}
```

----------Bitwise AND of Numbers Range
----------Given two integers left and right that represent the range [left, right], return the bitwise AND of all numbers in this range, inclusive.

Example 1:

Input: left = 5, right = 7
Output: 4
Example 2:

Input: left = 0, right = 0
Output: 0
Example 3:

Input: left = 1, right = 2147483647
Output: 0

Constraints:

0 <= left <= right <= 231 - 1

Goal is to get the MSB of both m and n, we keep shifting m and n to the RIGHT until they are same and keep a track how many times we shifted the integers. Then shift either m or n to the LEFT by counter times.

m = 18 (10010) base 2
n = 20 (10100) base 2

Background Info:
10010 AND 10100 = 10000

The answer of the AND operation also seems to be our MSB.

Control Flow of the Logic:

m >> 1 = 01001
n >> 1 = 01010

count++

m >> 1 = 00100
n >> 1 = 00101
count++

m >> 1 = 00100
n >> 1 = 00010
count++

answer = m << count = 10010 << 3 = 10000

```
class Solution {
    public int rangeBitwiseAnd(int m, int n) {
        int count = 0;
        while (m != n) {
            m = m >> 1;
            n = n >> 1;
            count++;
        }
        return m << count;
    }
}
```

----------Happy Number
----------Write an algorithm to determine if a number n 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.
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.
Return true if n is a happy number, and false if not.

Example 1:

Input: n = 19
Output: true
Explanation:
12 + 92 = 82
82 + 22 = 68
62 + 82 = 100
12 + 02 + 02 = 1
Example 2:

Input: n = 2
Output: false

Constraints:

1 <= n <= 231 - 1

```
class Solution {
    public boolean isHappy(int n) {
        while(true)
        {
            int sum=0;
            while(n!=0)
            {
                int lastDigit=n%10;             //here, we have to extract the digits from the given
number starting from the last one
                sum+=lastDigit*lastDigit;       //add that digit to the sum, by squaring it
                n/=10;
            }
            if(sum/10==0)                       //if it is a single digit number, check if it 1 or 7
            {
                if(sum==1 || sum==7)            //if yes, then this a happy number
                    return true;
                else                            // else you want to terminate the loop as it isn't a happy
number
                    return false;
            }
            n=sum;
        }
    }
}
```

----------Remove Linked List Elements----------Given the head of a linked list and an integer val, remove all the nodes of the linked list that has Node.val == val, and return the new head.

Example 1:

```
Input: head = [1,2,6,3,4,5,6], val = 6
Output: [1,2,3,4,5]
```

Example 2:

```
Input: head = [], val = 1
Output: []
```

Example 3:

```
Input: head = [7,7,7,7], val = 7
Output: []
```

Constraints:

The number of nodes in the list is in the range [0, 104].
1 <= Node.val <= 50
0 <= val <= 50

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

```
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode removeElements(ListNode head, int val) {

        ListNode t = new ListNode(); // Create a temporary copy of head - will be used to create
new linkedlist
        ListNode newHead = t; // Create a temporary copy of above copy - will be used to return
new linkedlist head node
        while(head != null) {
            if(head.val != val) { // only add those nodes where head.val != val
                t.next = head;
                t = t.next;
            }
            head = head.next;
        }

        t.next = null;// to pass edge case if node to be deleted is the last node

        return newHead.next;
    }
}
```

Count Primes

Given an integer n, return the number of prime numbers that are strictly less than n.

Example 1:

Input: n = 10
Output: 4
Explanation: There are 4 prime numbers less than 10, they are 2, 3, 5, 7.
Example 2:

Input: n = 0
Output: 0
Example 3:

Input: n = 1
Output: 0

Constraints:

0 <= n <= 5 * 106

```
class Solution {
public int countPrimes(int n) {
    if (n < 3)
        return 0;
```

```
    boolean[] f = new boolean[n];
    //Arrays.fill(f, true); boolean[] are initialed as false by default
    int count = n / 2;
    for (int i = 3; i * i < n; i += 2) {
        if (f[i])
            continue;

        for (int j = i * i; j < n; j += 2 * i) {
            if (!f[j]) {
                --count;
                f[j] = true;
            }
        }
    }
    return count;
}
}
```

----------Isomorphic Strings
----------Given two strings s and t, determine if they are isomorphic.

Two strings s and t 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.

Example 1:

Input: s = "egg", t = "add"
Output: true
Example 2:

Input: s = "foo", t = "bar"
Output: false
Example 3:

Input: s = "paper", t = "title"
Output: true

Constraints:

1 <= s.length <= 5 * 104
t.length == s.length
s and t consist of any valid ascii character

```
public class Solution {
    public boolean isIsomorphic(String s, String t) {
        if(s == null || t == null){
            return false;
```

```
        }
        if(s.length() == 0 || t.length() == 0){
            return true;
        }

        Map<Character,Character> corr = new HashMap<Character,Character>();
        for(int i = 0; i < s.length(); i ++){
            char sc = s.charAt(i);
            char tc = t.charAt(i);
            if(corr.containsKey(sc)){
                if(corr.get(sc) != tc){
                    return false;
                }
            }
            else{
                if(corr.containsValue(tc)){
                    return false;
                }
                else{
                    corr.put(sc,tc);
                }
            }
        }
        return true;
    }
}
```

----------Reverse Linked List

----------Given the head of a singly linked list, reverse the list, and return the reversed list.

Example 1:

Input: head = [1,2,3,4,5]
Output: [5,4,3,2,1]
Example 2:

Input: head = [1,2]
Output: [2,1]
Example 3:

Input: head = []
Output: []

Constraints:

The number of nodes in the list is the range [0, 5000].
-5000 <= Node.val <= 5000

Follow up: A linked list can be reversed either iteratively or recursively. Could you implement both?

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

        while(curr != null){
            next = curr.next;
            curr.next = prev;
            prev = curr;
            curr = next;
        }

        return prev;
    }
}
```

----------Course Schedule----------There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you must take course bi first if you want to take course ai.

For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.
Return true if you can finish all courses. Otherwise, return false.

Example 1:

Input: numCourses = 2, prerequisites = [[1,0]]
Output: true
Explanation: There are a total of 2 courses to take.
To take course 1 you should have finished course 0. So it is possible.
Example 2:

Input: numCourses = 2, prerequisites = [[1,0],[0,1]]
Output: false
Explanation: 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.

Constraints:

1 <= numCourses <= 105
0 <= prerequisites.length <= 5000
prerequisites[i].length == 2
0 <= ai, bi < numCourses
All the pairs prerequisites[i] are unique.

```
class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        List<Integer>[] graph = new ArrayList[numCourses];
        int[] preCount = new int[numCourses];
        for(int i = 0; i<graph.length;i++)graph[i] = new ArrayList<Integer>();
        for(int[] p : prerequisites){
            graph[p[1]].add(p[0]);
            preCount[p[0]]++;
        }
        Queue<Integer> q = new LinkedList<Integer>();
        for(int i = 0;i<preCount.length;i++){
            if(preCount[i]==0)q.offer(i);
        }
        while(!q.isEmpty()){
            int node = q.poll();
            for(int n : graph[node]){
               preCount[n]--;
               if(preCount[n]==0)q.offer(n);
            }
        }
        for(int num : preCount)if(num>0)return false;
        return true;
    }
}
```

----------Implement Trie (Prefix Tree)
----------A trie (pronounced as "try") or prefix tree is a tree data structure used to efficiently store and retrieve keys in a dataset of strings. There are various applications of this data structure, such as autocomplete and spellchecker.

Implement the Trie class:

Trie() Initializes the trie object.
void insert(String word) Inserts the string word into the trie.
boolean search(String word) Returns true if the string word is in the trie (i.e., was inserted before), and false otherwise.
boolean startsWith(String prefix) Returns true if there is a previously inserted string word that has the prefix prefix, and false otherwise.

Example 1:

Input
["Trie", "insert", "search", "search", "startsWith", "insert", "search"]
[[], ["apple"], ["apple"], ["app"], ["app"], ["app"], ["app"]]
Output
[null, null, true, false, true, null, true]

Explanation
Trie trie = new Trie();
trie.insert("apple");
trie.search("apple");   // return True
trie.search("app");   // return False
trie.startsWith("app"); // return True
trie.insert("app");
trie.search("app");   // return True

Constraints:

1 <= word.length, prefix.length <= 2000
word and prefix consist only of lowercase English letters.
At most 3 * 104 calls in total will be made to insert, search, and startsWith.

```
class Trie {
    Node root;

    /** Initialize your data structure here. */
    public Trie() {
        root = new Node();
    }

    /** Inserts a word into the trie. */
    public void insert(String word) {
        char[] chars = word.toCharArray();
        Node current = root;

        for(int i=0; i<word.length(); i++){
            current = current.addChar(chars[i]);
        }

        current.ended = true;
    }

    /** Returns if the word is in the trie. */
    public boolean search(String word) {
        Node current = root;
        char[] chars = word.toCharArray();

        for(int i=0; i<chars.length; i++){
            if(current.nodes[chars[i]-'a'] == null) return false;
            current = current.nodes[chars[i]-'a'];
        }

        if(current.ended) return true;
        return false;

    }

    /** Returns if there is any word in the trie that starts with the given prefix. */
    public boolean startsWith(String prefix) {
```

```
        Node current = root;
        char[] chars = prefix.toCharArray();

        for(int i=0; i< chars.length; i++){
            if(current.nodes[chars[i]-'a'] == null) return false;
            current = current.nodes[chars[i]-'a'];
        }

        return true;
    }

    class Node {
        Node[] nodes = new Node['z'-'a'+1];
        boolean ended = false;

        public Node addChar(char c) {
            if(nodes[c-'a'] == null) {
                nodes[c-'a'] = new Node();
            }

            return nodes[c-'a'];
        }
    }
}
```

----------Minimum Size Subarray Sum
----------Given an array of positive integers nums and a positive integer target, return the minimal length of a contiguous subarray [numsl, numsl+1, ..., numsr-1, numsr] of which the sum is greater than or equal to target. If there is no such subarray, return 0 instead.

Example 1:

Input: target = 7, nums = [2,3,1,2,4,3]
Output: 2
Explanation: The subarray [4,3] has the minimal length under the problem constraint.
Example 2:

Input: target = 4, nums = [1,4,4]
Output: 1
Example 3:

Input: target = 11, nums = [1,1,1,1,1,1,1,1]
Output: 0

Constraints:

1 <= target <= 109
1 <= nums.length <= 105
1 <= nums[i] <= 105

```
class Solution {
```

```
  public int minSubArrayLen(int target, int[] nums) {
   var minWindowSize = Integer.MAX_VALUE;

   for (int right = 0, left = 0, windowSum = 0; right < nums.length; right++) {
      windowSum += nums[right]; // expand window to the right

      while (windowSum >= target) {
         minWindowSize = Math.min(minWindowSize, right - left + 1); // record min window size
         windowSum -= nums[left++]; // shrink window from the left
      }
   }
   return minWindowSize == Integer.MAX_VALUE ? 0 : minWindowSize;
}
}
```

----------Course Schedule II
----------There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you must take course bi first if you want to take course ai.

For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.
Return the ordering of courses you should take to finish all courses. If there are many valid answers, return any of them. If it is impossible to finish all courses, return an empty array.

Example 1:

Input: numCourses = 2, prerequisites = [[1,0]]
Output: [0,1]
Explanation: 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].
Example 2:

Input: numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]
Output: [0,2,1,3]
Explanation: 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].
Example 3:

Input: numCourses = 1, prerequisites = []
Output: [0]

Constraints:

1 <= numCourses <= 2000
0 <= prerequisites.length <= numCourses * (numCourses - 1)
prerequisites[i].length == 2
0 <= ai, bi < numCourses
ai != bi
All the pairs [ai, bi] are distinct.

// Topological kahn's algo - best

```
class Solution {
    public int[] findOrder(int numCourses, int[][] prerequisites) {
        int []indeg = new int[numCourses]; // array to maintain indegee count
        List<List<Integer>> adj = new ArrayList<>();
        computeIndegreeAndAjacencyList(indeg, prerequisites, adj); // compute indegree and
adjaceny matrix for graphical representation
        return bfsScheduler(adj, indeg); // generate order  based on topological sort(BFS, Kahns
algo for DAG topological sort)
    }

    private void computeIndegreeAndAjacencyList(int indeg[], int [][]g, List<List<Integer>> adj) {
        int n = indeg.length;

        for (int i = 0; i < n; i++) {
            adj.add(new ArrayList<>());
        }

        for (int[] edge : g) {
            indeg[edge[0]]++;
            adj.get(edge[1]).add(edge[0]);
        }
    }


    private int[] bfsScheduler(List<List<Integer>> adj, int[] indeg) {
        int[] res = new int[indeg.length];
        Queue<Integer> q = new LinkedList<>();
        for (int i = 0; i < indeg.length; i++) {
            if (indeg[i] == 0) { // add only edges which have zero indegree i.e. traverse first without
any node coming to them but might be going from them
                q.offer(i);
            }
        }
        int index = 0;
        int count = 0;
        while (!q.isEmpty()) {
            int u = q.poll();
            res[index++] = u;
            for (int i : adj.get(u)) {
                if (--indeg[i] == 0) {
                    q.offer(i);
                }
            }
            count++;
        }

        return count == indeg.length ? res : new int[0];
    }
}

// DFS
class Solution {
    private int seq = 0;
```

```
    public int[] findOrder(int numCourses, int[][] prerequisites) {
        int[] sequence = new int[numCourses];
        List<Integer> courses[] = new ArrayList[numCourses];
        for(int i = 0; i < numCourses; i++) {
            courses[i] = new ArrayList<>();
        }
        for(int[] pair : prerequisites) {
            courses[pair[0]].add(pair[1]);
        }
        boolean[] taking = new boolean[numCourses];
        boolean[] taken = new boolean[numCourses];
        for(int i = 0; i < numCourses; i++) {
            if(!taken[i] && !complete(i, courses, taken, taking, sequence)) {
                return new int[]{};
            }
        }
        return sequence;
    }

    private boolean complete(int cur, List<Integer>[] courses, boolean[] taken,
                             boolean[] taking, int[] sequence) {
        taking[cur] = true;
        for(Integer pre : courses[cur]) {
            if(taking[pre] || !taken[pre] && !complete(pre, courses, taken, taking, sequence)) {
                return false;
            }
        }
        taking[cur] = false;
        taken[cur] = true;
        if(taken[cur]) {
            sequence[seq++] = cur;
        }
        return true;
    }
}

// cycle detection

class Solution {
    private int pos = 0;
    public int[] findOrder(int numCourses, int[][] prerequisites) {
        List<Integer> schedule[] = new ArrayList[numCourses];
        int[] res = new int[numCourses];
        boolean[] normal = new boolean[numCourses];
        boolean[] loop = new boolean[numCourses];
        for(int i = 0; i < numCourses; i++) {
            schedule[i] = new ArrayList<>();
        }
        for(int[] seq : prerequisites) {
            schedule[seq[0]].add(seq[1]);
        }

        for(int i = 0; i < numCourses; i++) {
            if(detectCycle(i, normal, loop, schedule, res)) {
```

```
                return new int[]{};
            }
        }
        return res;
    }


    private boolean detectCycle(int cur, boolean[] normal, boolean[] loop,
                            List<Integer>[] schedule, int[] res) {
        if(loop[cur]) return true;
        if(normal[cur]) return false;
        loop[cur] = true;
        normal[cur] = true;
        for(Integer pre : schedule[cur]) {
            if(detectCycle(pre, normal, loop, schedule, res)) {
                return true;
            }
        }
        loop[cur] = false;
        res[pos++] = cur;
        return false;
    }
}
```

----------Add and Search Word - Data structure design----------Design a data structure that supports adding new words and finding if a string matches any previously added string.

Implement the WordDictionary class:

WordDictionary() Initializes the object.
void addWord(word) Adds word to the data structure, it can be matched later.
bool search(word) Returns true if there is any string in the data structure that matches word or false otherwise. word may contain dots '.' where dots can be matched with any letter.

Example:

Input
["WordDictionary","addWord","addWord","addWord","search","search","search","search"]
[[],["bad"],["dad"],["mad"],["pad"],["bad"],[".ad"],["b.."]]
Output
[null,null,null,null,false,true,true,true]

Explanation
WordDictionary wordDictionary = new WordDictionary();
wordDictionary.addWord("bad");
wordDictionary.addWord("dad");
wordDictionary.addWord("mad");
wordDictionary.search("pad"); // return False
wordDictionary.search("bad"); // return True
wordDictionary.search(".ad"); // return True
wordDictionary.search("b.."); // return True

Constraints:

1 <= word.length <= 500
word in addWord consists lower-case English letters.
word in search consist of '.' or lower-case English letters.
At most 50000 calls will be made to addWord and search.

```
// using hashmap - not optimised
public class WordDictionary {
private HashMap<Integer, HashSet<String>> map = new HashMap<>();

public void addWord(String word) {
    int L = word.length();
    if(map.containsKey(L)) {
        map.get(L).add(word);
    }
    else {
        HashSet<String> set = new HashSet<>();
        set.add(word);
        map.put(L, set);
    }
}

public boolean search(String word) {
    int L = word.length();
    if(!map.containsKey(L)) return false;
    HashSet<String> set = map.get(L);
    for(String str : set) {
        int i = 0, LL = word.length();
        while(i<LL) {
            if(word.charAt(i)!='.' && str.charAt(i)!=word.charAt(i)) break;
            ++i;
        }
        //System.out.println(i +"\t"+ LL);
        if(i==LL && (word.charAt(i-1)=='.' || str.charAt(i-1)==word.charAt(i-1))) return true;
    }
    return false;
}
}

// TRIE
class WordDictionary {

    private static final int ALPHABETS = 26;

    /** TrieNode class contains child TrieNode and end to denote end of particular word  */
    private class TrieNode {
        private TrieNode[] child = new TrieNode[ALPHABETS];
        private boolean end;
    }

    private TrieNode root;
```

```
    /** Initialize your data structure here. */
    public WordDictionary() {
        this.root = new TrieNode();
    }

    /** Adds a word into the data structure. */
    public void addWord(String word) {
        TrieNode trieNode = root;
        for(Character c : word.toCharArray()){
            int index = c - 'a';
            if(trieNode.child[index]== null){
                trieNode.child[index] = new TrieNode();
            }
            trieNode = trieNode.child[index];
        }
        trieNode.end = 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 match(word.toCharArray(), root, 0);
    }

    private boolean match(char[] word,TrieNode node, int index) {

        /** word is not present in trie, return false */
        if(node == null) return false;

        /** word of exact length found in trie, return if it is a valid word (TrieNode.end) */
        if(index == word.length) return node.end;

        char ch = word[index];

        /* for ".", run matching condition for every alphabet */
        if(ch == '.') {

            for(int i = 0; i< ALPHABETS ; i++) {
                if (node.child[i] != null && match(word, node.child[i], index+1)) {
                    return true;
                }
            }

        } else {
            /*  for normal character, check if child node is present & matches */
            return node.child[ch - 'a'] != null && match(word, node.child[ch - 'a'], index+1);
        }

        /* return false default  */
        return false;

    }
}
```

```
/**
 * Your WordDictionary object will be instantiated and called as such:
 * WordDictionary obj = new WordDictionary();
 * obj.addWord(word);
 * boolean param_2 = obj.search(word);
 */
```

----------Word Search II

----------Given an m x n board of characters and a list of strings words, return all words on the board.

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

Example 1:

Input: board = [["o","a","a","n"],["e","t","a","e"],["i","h","k","r"],["i","f","l","v"]], words = ["oath","pea","eat","rain"]
Output: ["eat","oath"]
Example 2:

Input: board = [["a","b"],["c","d"]], words = ["abcb"]
Output: []

Constraints:

m == board.length
n == board[i].length
1 <= m, n <= 12
board[i][j] is a lowercase English letter.
1 <= words.length <= 3 * 104
1 <= words[i].length <= 10
words[i] consists of lowercase English letters.
All the strings of words are unique.

```
class Solution {
    class Node {
        String word;
        char ch;
        Node left, mid, right;
        public Node(char ch) {
            this.ch = ch;
            word = null;
            left = null;
            mid = null;
            right = null;
        }
    }
```

```
private Node addWord(Node node, String word, int index) {
    if (index == word.length())
        return node;

    char ch = word.charAt(index);
    if (node == null)
        node = new Node(ch);

    if (node.ch == ch) {
        node.mid = addWord(node.mid, word, index + 1);
        if (index == word.length() - 1)
            node.word = word;
    } else if (node.ch > ch)
        node.left = addWord(node.left, word, index);
    else
        node.right = addWord(node.right, word, index);

    return node;
}

public List<String> findWords(char[][] board, String[] words) {
    // Create the Trie with all the words
    Node root = null;
    for(String word : words)
        root = addWord(root, word, 0);

    // start backtracking from all the possible positions
    List<String> answer = new ArrayList<>();
    for (int i = 0; i < board.length; i++) {
        for (int j = 0; j < board[0].length; j++) {
            findWords(board, i, j, root, answer);
        }

        // If we found all the words passed, no need to run any more backtracking
        if (answer.size() == words.length)
            break;
    }

    return answer;
}

int[] directions = new int[]{-1, 0, 1, 0, -1};
private Node findWords(char[][] board, int row, int col, Node node, List<String> answer) {
    if (node == null)
        return null;

    char ch = board[row][col];

    if (ch < node.ch)
        node.left = findWords(board, row, col, node.left, answer);
    else if (ch > node.ch)
        node.right = findWords(board, row, col, node.right, answer);
    else {
```

```
            // If a word is ending on this node, add it to list and remove the word from this node to
avoid possible duplication in answer ArrayList
            if (node.word != null) {
                answer.add(node.word);
                node.word = null;
            }

            // Run backtraking in all possible 4 directions from this node
            board[row][col] = '*';
            for(int i = 0; i < 4; i++) {
                int newR = row + directions[i];
                int newC = col + directions[i + 1];
                if (validIndex(board, newR, newC)) {
                    node.mid = findWords(board, newR, newC, node.mid, answer);
                }
            }
            board[row][col] = ch;
        }

        // prune this node out of the Trie since it's a leaf node and any word ending on it is already
added to the answer list
        // Pruning is important. Without pruning i.e this if condition, program takes 61ms instead
of 0ms
        if (node.left == null && node.mid == null && node.right == null && node.word == null)
            return null;

        return node;
    }

    private boolean validIndex(char[][] board, int row, int col) {
        return (
            row >= 0 && col >= 0 &&
            row < board.length && col < board[0].length &&
            board[row][col] != '*'
        );
    }
}
```

----------House Robber II----------

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses at this place are arranged in a circle. That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have a security system connected, and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given an integer array nums representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

Example 1:

Input: nums = [2,3,2]
Output: 3

Explanation: You cannot rob house 1 (money = 2) and then rob house 3 (money = 2), because they are adjacent houses.
Example 2:

Input: nums = [1,2,3,1]
Output: 4
Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).
Total amount you can rob = 1 + 3 = 4.
Example 3:

Input: nums = [1,2,3]
Output: 3

Constraints:

1 <= nums.length <= 100
0 <= nums[i] <= 1000

```
class Solution {

    public int rob(int[] nums) {
        if(nums.length==1){
            return nums[0];
        }
        return Math.max(robHouses(nums,0,nums.length-1), robHouses(nums,1,nums.length));
    }

    public int robHouses(int [] nums, int low, int high){
        int Include = nums[low];
        int exclude = 0;
        for(int i=low+1;i<high;i++){
            int newInclude = exclude + nums[i];
            int newExclude = Math.max(Include,exclude);
            Include=newInclude;
            exclude=newExclude;
        }
        return Math.max(Include,exclude);
    }
}
```

----------Shortest Palindrome
----------You are given a string s. You can convert s to a palindrome by adding characters in front of it.

Return the shortest palindrome you can find by performing this transformation.

Example 1:

Input: s = "aacecaaa"
Output: "aaacecaaa"
Example 2:

Input: s = "abcd"
Output: "dcbabcd"

Constraints:

0 <= s.length <= 5 * 104
s consists of lowercase English letters only.

```
class Solution {
   public String shortestPalindrome(String s) {
    int n = s.length(), pos = -1;
    long B = 29, MOD = 1000000007, POW = 1, hash1 = 0, hash2 = 0;
    for (int i = 0; i < n; i++, POW = POW * B % MOD) {
        hash1 = (hash1 * B + s.charAt(i) - 'a' + 1) % MOD;
        hash2 = (hash2 + (s.charAt(i) - 'a' + 1) * POW) % MOD;
        if (hash1 == hash2) pos = i;
    }
    return new StringBuilder().append(s.substring(pos + 1, n)).reverse().append(s).toString();
}


}
```

----------Kth Largest Element in an Array
----------Given an integer array nums and an integer k, return the kth largest element in the array.

Note that it is the kth largest element in the sorted order, not the kth distinct element.

Example 1:

Input: nums = [3,2,1,5,6,4], k = 2
Output: 5
Example 2:

Input: nums = [3,2,3,1,2,4,5,5,6], k = 4
Output: 4

Constraints:

1 <= k <= nums.length <= 104
-104 <= nums[i] <= 104

```
class Solution {
    public static void minHeapify(int[] arr, int s, int n) {
        int sm = s, l = s * 2 + 1, r = 2 * s + 2;
        if (l < n && arr[l] < arr[sm])
            sm = l;
        if (r < n && arr[r] < arr[sm])
            sm = r;
```

```
        if (sm != s) {
            int t = arr[sm];
            arr[sm] = arr[s];
            arr[s] = t;
            minHeapify(arr, sm, n);
        }
    }
    public static void maxKEle(int[] arr, int k) {
        for (int i = (k / 2) - 1; i >= 0; i--)
            minHeapify(arr, i, k);
    }
    public int findKthLargest(int[] nums, int k) {
    maxKEle(nums, k);
        for (int i = k; i < nums.length; i++) {
            if (nums[i] > nums[0]) {
                nums[0] = nums[i];
                minHeapify(nums, 0, k);
            }
        }
        return nums[0];
    }
}
```

----------Combination Sum III

----------Find all possible combinations of k numbers that add up to a number n, given that only numbers from 1 to 9 can be used and each combination should be a unique set of numbers.

Example 1:
Input: k = 3,
Output:

n = 7

[[1,2,4]]
Example 2:
Input: k = 3,
Output:

n = 9

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

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

```
class Solution {
  public List<List<Integer>> combinationSum3(int k, int n) {
    List<List<Integer>> res = new ArrayList<>();
     solve(k, n, new ArrayList<>(), res, 0, 1);
      return res;
  }
    void solve (int k, int target, List<Integer>temp, List<List<Integer>> res, int sum, int start) {
    if(sum==target && temp.size()==k) {
    res.add(new ArrayList<>(temp));
```

```
        return;
    }
        if(temp.size()>=k || sum>target) return;


        for(int i = start; i<=9; i++) {
        temp.add(i);
        solve(k, target, temp, res, sum+i, i+1);
            temp.remove(temp.size()-1);
        }
    }


}
```

----------Contains Duplicate
----------Given an integer array nums, return true if any value appears at least twice in the array, and return false if every element is distinct.

Example 1:

Input: nums = [1,2,3,1]
Output: true
Example 2:

Input: nums = [1,2,3,4]
Output: false
Example 3:

Input: nums = [1,1,1,3,3,4,3,2,4,2]
Output: true

Constraints:

1 <= nums.length <= 105
-109 <= nums[i] <= 109

```
class Solution {
    public boolean containsDuplicate(int[] nums) {
        Arrays.sort(nums);
        for(int i=0;i<nums.length-1;i++){
            if(nums[i]==nums[i+1])return true;
        }

        return false;
    }
}
```

----------The Skyline Problem
----------A city's skyline is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Given the locations and heights of all the buildings, return the skyline formed by these buildings collectively.

The geometric information of each building is given in the array buildings where buildings[i] = [lefti, righti, heighti]:

lefti is the x coordinate of the left edge of the ith building.
righti is the x coordinate of the right edge of the ith building.
heighti is the height of the ith building.
You may assume all buildings are perfect rectangles grounded on an absolutely flat surface at height 0.

The skyline should be represented as a list of "key points" sorted by their x-coordinate in the form [[x1,y1],[x2,y2],...]. Each key point is the left endpoint of some horizontal segment in the skyline except the last point in the list, which always has a y-coordinate 0 and is used to mark the skyline's termination where the rightmost building ends. Any ground between the leftmost and rightmost buildings should be part of the skyline's contour.

Note: There must be no consecutive horizontal lines of equal height in the output skyline. For instance, [...,[2 3],[4 5],[7 5],[11 5],[12 7],...] is not acceptable; the three lines of height 5 should be merged into one in the final output as such: [...,[2 3],[4 5],[12 7],...]

Example 1:

Input: buildings = [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]
Output: [[2,10],[3,15],[7,12],[12,0],[15,10],[20,8],[24,0]]
Explanation:
Figure A shows the buildings of the input.
Figure B shows the skyline formed by those buildings. The red points in figure B represent the key points in the output list.
Example 2:

Input: buildings = [[0,2,3],[2,5,3]]
Output: [[0,3],[5,0]]

Constraints:

1 <= buildings.length <= 104
0 <= lefti < righti <= 231 - 1
1 <= heighti <= 231 - 1
buildings is sorted by lefti in non-decreasing order.

```
// Sort + TreeMap Solution
// 1. Build sorted lines by x then by y
//      1) Split each building into left lines (negative y) and right lines (positive y).
//         In this way we can have have (l2, (l1, r1), r2) to remove the l1, r2 if x is same
//      2) Sort the lines by x from left to right. If same x, sort y from low to high.
// 2. TreeMap to process overlapped buildings by store <height, count> pairs
//      1) If left lines, add to TreeMap. Otherwise, remove 1 for this height.
//      2) Get the max height in current TreeMap.
//         If current max height is different with previous max height,
//         add to the result list and update the previous max height.
```

```
// Time complexity: O(NlogN)
// Space complexity: O(N)
class Solution {
    public List<List<Integer>> getSkyline(int[][] buildings) {
        if (buildings == null || buildings.length == 0) return new ArrayList<>();
        List<List<Integer>> res = new ArrayList<>();
        // 1. Build sorted lines by x first then by y
        List<Line> lines = buildSortedLines(buildings);

        // 2. TreeMap to process overlapped buildings by store <height, count> pairs
        TreeMap<Integer, Integer> map = new TreeMap<>();
        map.put(0, 1);
        int curHeight = 0, prevHeight = 0;
        for (Line line : lines) {
            // Update TreeMap
            if (line.y < 0) { // If left line, add the height to TreeMap.
                int count = map.getOrDefault(-line.y, 0) + 1;
                map.put(-line.y, count);
            } else {  // If right line, check whether height exists.
                int count = map.get(line.y) - 1;
                if (count > 0) {
                    map.put(line.y, count);
                } else {
                    // If count == 0, then the line is the last coordinate left line, remove it.
                    map.remove(line.y);
                }
            }
            // Update skyline if curHeight != prevHeight
            curHeight = map.lastKey();
            if (curHeight != prevHeight) {
                res.add(Arrays.asList(line.x, curHeight));
            }
            prevHeight = curHeight;
        }
        return res;
    }

    private List<Line> buildSortedLines(int[][] buildings) {
        List<Line> lines = new ArrayList<>(buildings.length * 2);
        for (int[] building : buildings) {
            lines.add(new Line(building[0], -building[2])); // Add left line x and -height
            lines.add(new Line(building[1], building[2]));  // Add right line x and height
        }
        Collections.sort(lines, (l1, l2) -> {
            int comp = Integer.compare(l1.x, l2.x); // Sort by x from left to right first
            if (comp != 0) {
                return comp;
            } else {
                return Integer.compare(l1.y, l2.y); // Sort by y from low to high if x same
            }
        });
        return lines;
    }
```

```
    class Line {
        int x, y;
        Line(int x, int y) {
            this.x = x;
            this.y = y;
        }
    }
}
```

----------Contains Duplicate II

----------Given an integer array nums and an integer k, return true if there are two distinct indices i and j in the array such that nums[i] == nums[j] and abs(i - j) <= k.

Example 1:

Input: nums = [1,2,3,1], k = 3
Output: true
Example 2:

Input: nums = [1,0,1,1], k = 1
Output: true
Example 3:

Input: nums = [1,2,3,1,2,3], k = 2
Output: false

Constraints:

1 <= nums.length <= 105
-109 <= nums[i] <= 109
0 <= k <= 105

```
class Solution {
public boolean containsNearbyDuplicate(int[] nums, int k) {
        HashMap<Integer,Integer> hm = new HashMap<>();
        int check = 0;
        for(int i = 0; i < nums.length; i++){

            if(hm.containsKey(nums[i])){
                check = (int)hm.get(nums[i]);

                if(check != i && Math.abs(check - i) <= k)
                    return true;
                else
                    hm.put(nums[i],i);
            }else
                hm.put(nums[i],i);
        }
        return false;
    }
}
```

----------Contains Duplicate III

------------------------------Given an integer array nums and two integers k and t, return true if there are two distinct indices i and j in the array such that abs(nums[i] - nums[j]) <= t and abs(i - j) <= k.

Example 1:

Input: nums = [1,2,3,1], k = 3, t = 0
Output: true
Example 2:

Input: nums = [1,0,1,1], k = 1, t = 2
Output: true
Example 3:

Input: nums = [1,5,9,1,5,9], k = 2, t = 3
Output: false

Constraints:

1 <= nums.length <= 2 * 104
-231 <= nums[i] <= 231 - 1
0 <= k <= 104
0 <= t <= 231 - 1

```
/**
 * Sliding Window solution using Buckets
 *
 * Time Complexity: O(N)
 *
 * Space Complexity: O(min(N, K+1))
 *
 * N = Length of input array. K = Input difference between indexes.
 */
class Solution {
    public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {
        if (nums == null || nums.length < 2 || k < 1 || t < 0) {
            return false;
        }

        HashMap<Long, Long> buckets = new HashMap<>();
        // The bucket size is t+1 as the ranges are from 0..t, t+1..2t+1, ..
        long bucketSize = (long) t + 1;

        for (int i = 0; i < nums.length; i++) {
            // Making sure only K buckets exists in map.
            if (i > k) {
                long lastBucket = ((long) nums[i - k - 1] - Integer.MIN_VALUE) / bucketSize;
                buckets.remove(lastBucket);
            }

            long remappedNum = (long) nums[i] - Integer.MIN_VALUE;
```

```
            long bucket = remappedNum / bucketSize;

            // If 2 numbers belong to same bucket
            if (buckets.containsKey(bucket)) {
                return true;
            }

            // If numbers are in adjacent buckets and the difference between them is at most
            // t.
            if (buckets.containsKey(bucket - 1) && remappedNum - buckets.get(bucket - 1) <= t) {
                return true;
            }
            if (buckets.containsKey(bucket + 1) && buckets.get(bucket + 1) - remappedNum <= t) {
                return true;
            }

            buckets.put(bucket, remappedNum);
        }

        return false;
    }
}
```

----------Maximal Square

----------Given an m x n binary matrix filled with 0's and 1's, find the largest square containing only 1's and return its area.

Example 1:

Input: matrix = [["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],
["1","0","0","1","0"]]
Output: 4
Example 2:

Input: matrix = [["0","1"],["1","0"]]
Output: 1
Example 3:

Input: matrix = [["0"]]
Output: 0

Constraints:

m == matrix.length
n == matrix[i].length
1 <= m, n <= 300
matrix[i][j] is '0' or '1'.

```
/**
 * Space optimized Dynamic Programming solution
 *
```

```
 * DP[i][j] = Maximal size (square = size*size) of the square that can be formed ending at point
(i,j).
 *
 * Time Complexity: O(M * N)
 *
 * Space Complexity: O(min(M, N))
 *
 * M = Number of rows. N = Number of columns.
 */
class Solution {
    public int maximalSquare(char[][] matrix) {
        if (matrix == null) {
            throw new IllegalArgumentException("Input is null");
        }
        if (matrix.length == 0 || matrix[0].length == 0) {
            return 0;
        }

        int rows = matrix.length;
        int cols = matrix[0].length;

        if (rows < cols) {
            return maximalSquareHelper(matrix, cols, rows, false);
        } else {
            return maximalSquareHelper(matrix, rows, cols, true);
        }
    }

    private int maximalSquareHelper(char[][] matrix, int big, int small, boolean isColsSmall) {
        int[] dp = new int[small + 1];
        int maxSide = 0;
        for (int j = 1; j <= big; j++) {
            int prev = dp[0]; // Since we have added a padding in-front, dp[0] will always be zero
            for (int i = 1; i <= small; i++) {
                int temp = dp[i];
                if ((isColsSmall && matrix[j - 1][i - 1] == '0') || (!isColsSmall && matrix[i - 1][j - 1] == '0'))
{
                    dp[i] = 0;
                } else {
                    dp[i] = Math.min(prev, Math.min(dp[i], dp[i - 1])) + 1;
                    maxSide = Math.max(maxSide, dp[i]);
                }
                prev = temp;
            }
        }
        return maxSide * maxSide;
    }
}
----------------------------------------------------------------Count Complete Tree Nodes
-----------------------------Given the root of a complete binary tree, return the number of the
nodes in the tree.
```

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

Design an algorithm that runs in less than O(n) time complexity.

Example 1:

Input: root = [1,2,3,4,5,6]
Output: 6
Example 2:

Input: root = []
Output: 0
Example 3:

Input: root = [1]
Output: 1

Constraints:

The number of nodes in the tree is in the range [0, 5 * 104].
0 <= Node.val <= 5 * 104
The tree is guaranteed to be complete.

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

        rec(root);

        return count;
    }
```

```
    private void rec(TreeNode root) {
        if (root == null)
            return;

        count++;

        rec(root.left);
        rec(root.right);
    }
}
```

----------Rectangle Area

----------Given the coordinates of two rectilinear rectangles in a 2D plane, return the total area covered by the two rectangles.

The first rectangle is defined by its bottom-left corner (ax1, ay1) and its top-right corner (ax2, ay2).

The second rectangle is defined by its bottom-left corner (bx1, by1) and its top-right corner (bx2, by2).

Example 1:

Rectangle Area
Input: ax1 = -3, ay1 = 0, ax2 = 3, ay2 = 4, bx1 = 0, by1 = -1, bx2 = 9, by2 = 2
Output: 45
Example 2:

Input: ax1 = -2, ay1 = -2, ax2 = 2, ay2 = 2, bx1 = -2, by1 = -2, bx2 = 2, by2 = 2
Output: 16

Constraints:

-104 <= ax1, ay1, ax2, ay2, bx1, by1, bx2, by2 <= 104

```
class Solution {
    public int computeArea(int A, int B, int C, int D, int E, int F, int G, int H) {
        //find area of the 1st rectangle (length*breath)
        long area1 = (1L*D-1L*B)*(1L*C-1L*A);
        //finding area of the 2nd rectangle
        long area2 = (1L*H-1L*F)*(1L*G-1L*E);
        //finding overlaped region
        long overlap = Math.max(1L*Math.min(C,G)-1L*Math.max(A,E),0)*
                    Math.max(1L*Math.min(H,D)-1L*Math.max(B,F),0);
                    //Math.max(somevalue,0) -> 0 is for if there is no overlap it will took 0

        // now to area consists of 2 times overlap of rectangle since we adding area of both
        //rectangle. so subtracting 1 overlap region
        return (int)(area1+area2-overlap);

    }
```

}
----------Basic Calculator
----------Given a string s representing a valid expression, implement a basic calculator to evaluate it, and return the result of the evaluation.

Note: You are not allowed to use any built-in function which evaluates strings as mathematical expressions, such as eval().

Example 1:

Input: s = "1 + 1"
Output: 2
Example 2:

Input: s = " 2-1 + 2 "
Output: 3
Example 3:

Input: s = "(1+(4+5+2)-3)+(6+8)"
Output: 23

Constraints:

1 <= s.length <= 3 * 105
s consists of digits, '+', '-', '(', ')', and ' '.
s represents a valid expression.
'+' is not used as a unary operation (i.e., "+1" and "+(2 + 3)" is invalid).
'-' could be used as a unary operation (i.e., "-1" and "-(2 + 3)" is valid).
There will be no two consecutive operators in the input.
Every number and running calculation will fit in a signed 32-bit integer.

```
class Solution {
    int i =0;
    public int calculate(String s) {
        i=0;
        s = "(" + s + ")";
        return helper(s);
    }
    int helper(String s) {
        int num =0;
        int val = 0;
        int presign = 1;
        while(i < s.length()) {
            char c = s.charAt(i);
            if(c==' ') {
                i++;
            }else if(c=='+') {
                val = val + presign * num;
                num=0;
                presign = 1;
```

```
                i++;
            }else if(c=='-') {
                val = val + presign * num;
                num=0;
                presign = -1;
                i++;
            }else if(c=='(') {
                i++;
                val = val + presign * helper(s);
            }else if(c==')') {
                val = val + presign * num;
                i++;
                return val;
            }else {
                int n = c-'0';
                num = num * 10 + n;
                i++;
            }
        }
        return val;
    }
}
```

----------Implement Stack using Queues
----------Implement a last-in-first-out (LIFO) stack using only two queues. The implemented stack should support all the functions of a normal stack (push, top, pop, and empty).

Implement the MyStack class:

void push(int x) Pushes element x to the top of the stack.
int pop() Removes the element on the top of the stack and returns it.
int top() Returns the element on the top of the stack.
boolean empty() Returns true if the stack is empty, false otherwise.
Notes:

You must use only standard operations of a queue, which means that only push to back, peek/ pop from front, size and is empty operations are valid.
Depending on your language, the queue may not be supported natively. You may simulate a queue using a list or deque (double-ended queue) as long as you use only a queue's standard operations.

Example 1:

Input
["MyStack", "push", "push", "top", "pop", "empty"]
[[], [1], [2], [], [], []]
Output
[null, null, null, 2, 2, false]

Explanation
MyStack myStack = new MyStack();
myStack.push(1);
myStack.push(2);

myStack.top(); // return 2
myStack.pop(); // return 2
myStack.empty(); // return False

Constraints:

1 <= x <= 9
At most 100 calls will be made to push, pop, top, and empty.
All the calls to pop and top are valid.

Follow-up: Can you implement the stack using only one queue?

```
class MyStack {

    Queue<Integer> queue;

    public MyStack() {
        queue = new LinkedList<>();
    }

    public void push(int x) {

        Queue<Integer> temp = new LinkedList<>();
        temp.offer(x);

        while (!queue.isEmpty()) {
            temp.offer(queue.poll());
        }
        while (!temp.isEmpty()) {
            queue.offer(temp.poll());
        }
    }

    public int pop() {
        return queue.poll();
    }

    public int top() {
        return queue.peek();
    }

    public boolean empty() {
        return queue.isEmpty();
    }
}
```

----------Invert Binary Tree
----------Given the root of a binary tree, invert the tree, and return its root.

Example 1:

Input: root = [4,2,7,1,3,6,9]
Output: [4,7,2,9,6,3,1]
Example 2:

Input: root = [2,1,3]
Output: [2,3,1]
Example 3:

Input: root = []
Output: []

Constraints:

The number of nodes in the tree is in the range [0, 100].
-100 <= Node.val <= 100

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

        // Get the left sub tree
        TreeNode left = invertTree(root.left);
        // Get the right sub tree
        TreeNode right = invertTree(root.right);

        // Swap them -- make left to right and right to left
        root.left = right;
        root.right = left;

        // Return root node
        return root;
    }
}
```

Basic Calculator II

------------------------------Given a string s which represents an expression, evaluate this expression and return its value.

The integer division should truncate toward zero.

You may assume that the given expression is always valid. All intermediate results will be in the range of [-231, 231 - 1].

Note: You are not allowed to use any built-in function which evaluates strings as mathematical expressions, such as eval().

Example 1:

Input: s = "3+2*2"
Output: 7
Example 2:

Input: s = " 3/2 "
Output: 1
Example 3:

Input: s = " 3+5 / 2 "
Output: 5

Constraints:

1 <= s.length <= 3 * 105
s consists of integers and operators ('+', '-', '*', '/') separated by some number of spaces.
s represents a valid expression.
All the integers in the expression are non-negative integers in the range [0, 231 - 1].
The answer is guaranteed to fit in a 32-bit integer.

```
class Solution {
    public int calculate(String s) {
        Stack<Integer> st  = new Stack<>();
        int n = s.length();
        char sign = '+';
        for(int i = 0; i < n ; i++)
        {
            char ch = s.charAt(i);

            if(Character.isDigit(ch))
            {
                int val = 0;
                while(i < n && Character.isDigit(s.charAt(i))){
                    val = val*10 + (s.charAt(i) - '0');
                    i++;
                }
                i--;
                if(sign == '+'){
                    st.push(val);
```

```
            }
            else if(sign == '-'){
                st.push(-val);
            }
            else if(sign == '*'){
                int a = st.pop();
                int ans = a*val;
                st.push(ans);
            }
            else if(sign == '/'){
                int a = st.pop();
                int ans = a/val;
                st.push(ans);
            }
        }else if(ch != ' '){
            sign = ch;
        }
    }
    int sum = 0;
    while(st.size() > 0)
        sum += st.pop();
    return sum;

  }
}
```

----------Summary Ranges
----------You are given a sorted unique integer array nums.

Return the smallest sorted list of ranges that cover all the numbers in the array exactly. That is, each element of nums is covered by exactly one of the ranges, and there is no integer x such that x is in one of the ranges but not in nums.

Each range [a,b] in the list should be output as:

"a->b" if a != b
"a" if a == b

Example 1:

Input: nums = [0,1,2,4,5,7]
Output: ["0->2","4->5","7"]
Explanation: The ranges are:
[0,2] --> "0->2"
[4,5] --> "4->5"
[7,7] --> "7"
Example 2:

Input: nums = [0,2,3,4,6,8,9]
Output: ["0","2->4","6","8->9"]
Explanation: The ranges are:
[0,0] --> "0"
[2,4] --> "2->4"
[6,6] --> "6"

[8,9] --> "8->9"
Example 3:

Input: nums = []
Output: []
Example 4:

Input: nums = [-1]
Output: ["-1"]
Example 5:

Input: nums = [0]
Output: ["0"]

Constraints:

0 <= nums.length <= 20
-231 <= nums[i] <= 231 - 1
All the values of nums are unique.
nums is sorted in ascending order.

```
class Solution {

    public List<String> summaryRanges(int[] nums) {
        List<String> result = new ArrayList<>();
        if(nums.length==0) return result;
        int pos = 0;
        for(int i=1; i< nums.length; i++){
            if(nums[i]-nums[i-1] !=1){
                StringBuilder sb = new StringBuilder();
                sb.append(nums[pos]);
                if(i-pos-1==0)
                    result.add(sb.toString());
                else result.add(sb.append("->").append(nums[i-1]).toString());
                pos = i;
            }
        }
        StringBuilder sb = new StringBuilder();
        sb.append(nums[pos]);
        if(nums.length-pos-1==0)
            result.add(sb.toString());
        else  result.add(sb.append("->").append(nums[nums.length-1]).toString());
        return result;
    }
}
```

----------Majority Element II
----------Given an integer array of size n, find all elements that appear more than ⌊ n/3 ⌋ times.

Example 1:

Input: nums = [3,2,3]
Output: [3]
Example 2:

Input: nums = [1]
Output: [1]
Example 3:

Input: nums = [1,2]
Output: [1,2]

Constraints:

1 <= nums.length <= 5 * 104
-109 <= nums[i] <= 109

```
 /**
    Using modified Boyer-moore Algorithm
    For n/k, array can have atmost k-1 majority elements.  So using map instead of two
variables to generalize solution.
    There are only 2 possibilities:
    case 1: if new element is already in candidates then increase the count of the candidate.
    case 2: if not then
        2a: if any candidate has 0 votes then remove that candidate and make curr a new
candidate.
        2b: Otherwise reduce the votes of all candidates by 1;
    **/
class Solution {
    public List<Integer> majorityElement(int[] nums)
    {
        int val1 = nums[0];
        int count1 = 1;
        int val2 = val1;
        int count2 = 0;
        int i = 1;
        for(i = 1 ; i < nums.length ; i++)
        {
            if(val1 != nums[i])
            {
                val2 = nums[i];
                count2 = 1;
                i++;
                break;
            }

            count1++;
        }

        while(i < nums.length)
        {
            if(nums[i] == val1)
            {
```

```
                count1++;
            }
            else if(nums[i] == val2)
            {
                count2++;
            }
            else if(count1 == 0)
            {
                val1 = nums[i];
                count1 = 1;
            }
            else if(count2 == 0)
            {
                val2 = nums[i];
                count2 = 1;
            }
            else
            {
                count1--;
                count2--;
            }

            i++;
        }

        count1 = 0;
        count2 = 0;

        for(i = 0 ; i < nums.length ; i++)
        {
            if(nums[i] == val1)
            {
                count1++;
            }
            else if(nums[i] == val2)
            {
                count2++;
            }
        }

        List<Integer> ans = new ArrayList<>();

        if(count1 > nums.length / 3)
        {
            ans.add(val1);
        }
        if(count2 > nums.length / 3)
        {
            ans.add(val2);
        }

        return ans;
    }
}
```

----------Kth Smallest Element in a BST
----------Given the root of a binary search tree, and an integer k, return the kth smallest value (1-indexed) of all the values of the nodes in the tree.

Example 1:

Input: root = [3,1,4,null,2], k = 1
Output: 1
Example 2:

Input: root = [5,3,6,2,4,null,null,1], k = 3
Output: 3

Constraints:

The number of nodes in the tree is n.
1 <= k <= n <= 104
0 <= Node.val <= 104

Follow up: If the BST is modified often (i.e., we can do insert and delete operations) and you need to find the kth smallest frequently, how would you optimize?

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    int k;
    int ans;
    public int kthSmallest(TreeNode root, int k) {
        this.k=k;
        helper(root);
        return ans;
    }
    public void helper(TreeNode root){
        if(root==null){
            return;
```

```
        }
        helper(root.left);
        this.k--;
        if(k==0){
            ans=root.val;
            return;
        }
        helper(root.right);
    }
}
```

----------Power of Two
----------Given an integer n, return true if it is a power of two. Otherwise, return false.

An integer n is a power of two, if there exists an integer x such that n == 2x.

Example 1:

Input: n = 1
Output: true
Explanation: 20 = 1
Example 2:

Input: n = 16
Output: true
Explanation: 24 = 16
Example 3:

Input: n = 3
Output: false
Example 4:

Input: n = 4
Output: true
Example 5:

Input: n = 5
Output: false

Constraints:

-231 <= n <= 231 - 1

```
class Solution {
    public boolean isPowerOfTwo(int n) {

        if(n<=0)
        {
            return false;
        }
```

```
        while(n>1)
        {
            if(n%2==0)
            {
                n=n/2;
            }
            else
            {
                return false;
            }
        }

    return true;
    }
}
```

----------Implement Queue using Stacks----------IImplement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (push, peek, pop, and empty).

Implement the MyQueue class:

void push(int x) Pushes element x to the back of the queue.
int pop() Removes the element from the front of the queue and returns it.
int peek() Returns the element at the front of the queue.
boolean empty() Returns true if the queue is empty, false otherwise.
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, the stack may not be supported natively. You may simulate a stack using a list or deque (double-ended queue) as long as you use only a stack's standard operations.

Example 1:

Input
["MyQueue", "push", "push", "peek", "pop", "empty"]
[[], [1], [2], [], [], []]
Output
[null, null, null, 1, 1, false]

Explanation
MyQueue myQueue = new MyQueue();
myQueue.push(1); // queue is: [1]
myQueue.push(2); // queue is: [1, 2] (leftmost is front of the queue)
myQueue.peek(); // return 1
myQueue.pop(); // return 1, queue is [2]
myQueue.empty(); // return false

Constraints:

1 <= x <= 9
At most 100 calls will be made to push, pop, peek, and empty.
All the calls to pop and peek are valid.

Follow-up: Can you implement the queue such that each operation is amortized O(1) time complexity? In other words, performing n operations will take overall O(n) time even if one of those operations may take longer.

```
class MyQueue {

   Stack<Integer> s1 = new Stack<>();
    Stack<Integer> s2 = new Stack<>();
    /** Initialize your data structure here. */
    public MyQueue() {

    }

    /** Push element x to the back of queue. */
    public void push(int x) {
        s1.push(x);
    }

    /** Removes the element from in front of queue and returns that element. */
    public int pop() {
        while(s1.size() > 1){
            s2.push(s1.pop());
        }
        int ans = s1.pop();
        while(!s2.isEmpty()) {
            s1.push(s2.pop());
        }
        return ans;
    }

    /** Get the front element. */
    public int peek() {
        while(s1.size() > 1){
            s2.push(s1.pop());
        }
        int ans = s1.peek();
        while(!s2.isEmpty()) {
            s1.push(s2.pop());
        }
        return ans;
    }

    /** Returns whether the queue is empty. */
    public boolean empty() {
        return s1.isEmpty();
    }
}
```

----------Number of Digit One

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

Example 1:

Input: n = 13
Output: 6
Example 2:

Input: n = 0
Output: 0

Constraints:

0 <= n <= 109

```
class Solution {
    public int countDigitOne(int n) {
     int countr = 0;
    for (long i = 1; i <= n; i *= 10) {
        long  divider = i * 10;
        countr += (n / divider) * i + Math.min(Math.max(n % divider - i + 1, 0L), i);
    }
    return countr;
    }
}
```

----------Palindrome Linked List
----------Given the head of a singly linked list, return true if it is a palindrome.

Example 1:

Input: head = [1,2,2,1]
Output: true
Example 2:

Input: head = [1,2]
Output: false

Constraints:

The number of nodes in the list is in the range [1, 105].
0 <= Node.val <= 9

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

```
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
/**
 * Time Complexity: O(n)
 * Space Complexity: O(1)
 */
class Solution {
  public boolean isPalindrome(ListNode head) {
    if (head == null) return false;
    if (head.next == null) return true;
    ListNode pre = null;
    ListNode cur = head;
    ListNode next = null;
    ListNode run = head;
    /**
     * Time Complexity: O(n)
     *   - O(n/2), only run half link, find the middle position.
     */
    while (run != null) {
      if (run.next == null) {
        // If odd nodes, e.g. [1 -> 2 -> 1]
        next = cur.next;
        cur = pre;
        break;
      } else if (run.next.next != null) {
        // Progress
        run = run.next.next;
        next = cur.next;
        cur.next = pre;
        pre = cur;
        cur = next;
      } else {
        // If even nodes, e.g. [1 -> 2 -> 2 -> 1]
        next = cur.next;
        cur.next = pre;
        break;
      }
    }

    /**
     * Time Complexity: O(n)
     *   - O(n/2), go through left and right from middle position.
     */
    while (next != null) {
      if (cur.val != next.val) {
        return false;
      } else {
        cur = cur.next;
        next = next.next;
      }
    }
```

```
    return cur != null ? false : true;
  }
}
```
----------Lowest Common Ancestor of a Binary Search Tree
----------Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.

According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself).”

Example 1:

Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
Output: 6
Explanation: The LCA of nodes 2 and 8 is 6.
Example 2:

Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
Output: 2
Explanation: The LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition.
Example 3:

Input: root = [2,1], p = 2, q = 1
Output: 2

Constraints:

The number of nodes in the tree is in the range [2, 105].
-109 <= Node.val <= 109
All Node.val are unique.
p != q
p and q will exist in the BST.

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
//Algo:
    // if p and q is less than root, then lca should be in left subtree
    // if p and q is greater than root, then lca should be in right subtree
    // otherwise the root is the lca
```

```
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

        if(root==null){
            return null;
        }
        int curr = root.val;

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

        return root;

    }
}
```

----------Lowest Common Ancestor of a Binary Tree----------Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself).”

Example 1:

Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
Output: 3
Explanation: The LCA of nodes 5 and 1 is 3.
Example 2:

Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
Output: 5
Explanation: The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition.
Example 3:

Input: root = [1,2], p = 1, q = 2
Output: 1

Constraints:

The number of nodes in the tree is in the range [2, 105].
-109 <= Node.val <= 109
All Node.val are unique.
p != q
p and q will exist in the tree

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

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

    if (left == null) return right;
    else if (right == null) return left;
    else return root;
}
}
```

----------Delete Node in a Linked List----------Write a function to delete a node in a singly-linked list. You will not be given access to the head of the list, instead you will be given access to the node to be deleted directly.

It is guaranteed that the node to be deleted is not a tail node in the list.

Example 1:

Input: head = [4,5,1,9], node = 5
Output: [4,1,9]
Explanation: You are given the second node with value 5, the linked list should become 4 -> 1 -> 9 after calling your function.
Example 2:

Input: head = [4,5,1,9], node = 1
Output: [4,5,9]
Explanation: You are given the third node with value 1, the linked list should become 4 -> 5 -> 9 after calling your function.
Example 3:

Input: head = [1,2,3,4], node = 3
Output: [1,2,4]
Example 4:

Input: head = [0,1], node = 0
Output: [1]
Example 5:

Input: head = [-3,5,-99], node = -3
Output: [5,-99]

Constraints:

The number of the nodes in the given list is in the range [2, 1000].
-1000 <= Node.val <= 1000
The value of each node in the list is unique.
The node to be deleted is in the list and is not a tail node

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

}
}
```

----------Product of Array Except Self
----------Given an integer array nums, return an array answer such that answer[i] is equal to the product of all the elements of nums except nums[i].

The product of any prefix or suffix of nums is guaranteed to fit in a 32-bit integer.

You must write an algorithm that runs in O(n) time and without using the division operation.

Example 1:

Input: nums = [1,2,3,4]
Output: [24,12,8,6]
Example 2:

Input: nums = [-1,1,0,-3,3]
Output: [0,0,9,0,0]

Constraints:

2 <= nums.length <= 105
-30 <= nums[i] <= 30
The product of any prefix or suffix of nums is guaranteed to fit in a 32-bit integer.

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

```
        int[] res = new int[nums.length];
        Arrays.fill(res, 1);
        int prefix = 1;
        for(int i = 0; i < nums.length; i++) {
            res[i] = prefix;
            prefix *= nums[i];
        }
        int postfix = 1;
        for(int i = nums.length - 1; i >=0; i--) {
            res[i] *= postfix;
            postfix *= nums[i];
        }
        return res;
    }
}
```

----------Sliding Window Maximum
----------You are given an array of integers nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position.

Return the max sliding window.

Example 1:

Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
Output: [3,3,5,5,6,7]
Explanation:

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

Example 2:

Input: nums = [1], k = 1
Output: [1]
Example 3:

Input: nums = [1,-1], k = 1
Output: [1,-1]
Example 4:

Input: nums = [9,11], k = 2
Output: [11]
Example 5:

Input: nums = [4,-2], k = 2

Output: [4]

Constraints:

1 <= nums.length <= 105
-104 <= nums[i] <= 104
1 <= k <= nums.length

```
class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        int n = nums.length;
        int prefixmax[] = new int[n];
        int suffixmax[] = new int[n];
        prefixmax[0] = nums[0];

        for(int i = 1 ; i < n ; i++)
            prefixmax[i] = (i%k == 0) ?nums[i] : Math.max(prefixmax[i-1] , nums[i]);

        suffixmax[n-1] = nums[n-1];
        for(int i = n-2 ; i >= 0 ; i--)
            suffixmax[i] = ((i+1)%k == 0) ? nums[i] : Math.max(suffixmax[i+1] , nums[i]);



        int ans[] = new int[n-k+1];
        for(int i = 0 ; i < ans.length ; i++)
            ans[i] = Math.max(suffixmax[i] , prefixmax[i+k-1]);

        return ans;
    }
}
```

----------Search a 2D Matrix II
----------Write an efficient algorithm that searches for a target value in an m x n integer matrix. The 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.

Example 1:

Input: 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]], target = 5
Output: true
Example 2:

Input: 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]], target = 20
Output: false

Constraints:

m == matrix.length
n == matrix[i].length
1 <= n, m <= 300
-109 <= matrix[i][j] <= 109
All the integers in each row are sorted in ascending order.
All the integers in each column are sorted in ascending order.
-109 <= target <= 109

```
class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        if(matrix[0].length == 1)       //if matrix has only one column
        {
            for(int i = 0; i < matrix.length; ++i)
                if(matrix[i][0] == target)
                    return true;
            return false;
        }
        else if(matrix.length == 1)     //if matrix has only one row
        {
            if(Arrays.binarySearch(matrix[0], target) >= 0)
                return true;
            else
                return false;
        }
        for(int i = 0; i < matrix.length; ++i)
            if(matrix[i][0] <= target && matrix[i][matrix[i].length-1] >= target)
                if(Arrays.binarySearch(matrix[i], target) >= 0)
                    return true;
        return false;
    }
}
```

----------Different Ways to Add Parentheses----------Given a string expression of numbers and operators, return all possible results from computing all the different possible ways to group numbers and operators. You may return the answer in any order.

Example 1:

Input: expression = "2-1-1"
Output: [0,2]
Explanation:
((2-1)-1) = 0
(2-(1-1)) = 2
Example 2:

Input: expression = "2*3-4*5"
Output: [-34,-14,-10,-10,10]
Explanation:

(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

Constraints:

1 <= expression.length <= 20
expression consists of digits and the operator '+', '-', and '*'.
All the integer values in the input expression are in the range [0, 99].

```
class Solution {
public List<Integer> diffWaysToCompute(String expression) {
    return diffWaysToCompute(expression, new HashMap<>());
}

private List<Integer> diffWaysToCompute(String expression, Map<String, List<Integer>> map) {
    if (map.containsKey(expression))
        return map.get(expression);

    var values = new ArrayList<Integer>();
    if (!hasOperator(expression)) {
        // base case
        values.add(Integer.parseInt(expression));
    } else {
        // Recursive case. DFS
        for (var i = 0; i < expression.length(); i++) {
            var symbol = expression.charAt(i);

            if (!Character.isDigit(symbol)) {
                var left = diffWaysToCompute(expression.substring(0, i), map);
                var right = diffWaysToCompute(expression.substring(i + 1), map);
                for (var l : left) {
                    for (var r : right) {
                        switch (symbol) {
                            case '+' -> values.add(l + r);
                            case '-' -> values.add(l - r);
                            case '*' -> values.add(l * r);
                        }
                    }
                }
            }
        }
    }

    map.put(expression, values);
    return values;
}

private boolean hasOperator(String expression) {
    for (var i = 0; i < expression.length(); i++)
```

```
        switch (expression.charAt(i)) {
            case '+', '-', '*' -> {
                return true;
            }
        }
    return false;
}
}
```

----------Valid Anagram
----------Given two strings s and t, return true if t is an anagram of s, and false otherwise.

Example 1:

Input: s = "anagram", t = "nagaram"
Output: true
Example 2:

Input: s = "rat", t = "car"
Output: false

Constraints:

1 <= s.length, t.length <= 5 * 104
s and t consist of lowercase English letters.

```
class Solution {
public boolean isAnagram(String s, String t) {

    if(s.length()!=t.length())
        return false;

    int[] arr = new int[26];

    for(int i = 0; i < s.length(); i++){
        arr[s.charAt(i)-'a']++;
        arr[t.charAt(i)-'a']--;
    }


    for(int i = 0;i<s.length();i++)
        if(arr[(s.charAt(i)-'a')]!=0)
            return false;

    return true;

}
}
```

----------Binary Tree Paths

------------------------------Given the root of a binary tree, return all root-to-leaf paths in any order.

A leaf is a node with no children.

Example 1:

Input: root = [1,2,3,null,5]
Output: ["1->2->5","1->3"]
Example 2:

Input: root = [1]
Output: ["1"]

Constraints:

The number of nodes in the tree is in the range [1, 100].
-100 <= Node.val <= 100

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

class Solution {
    List list = new ArrayList<>();
    public List binaryTreePaths(TreeNode root) {
        if(root==null)
            return list;
        String result="";
        findPath(root, result);
        return list;
    }

    public void findPath(TreeNode root, String result) {
        result+=root.val;
        if(root.left==null && root.right==null) {
            list.add(result);
        }
```

```
        result+="->";
        if(root.left!=null) {
            findPath(root.left, result);
        }
        if(root.right!=null) {
            findPath(root.right, result);
        }
    }
}
```

--Add Digits
----------------------------Given an integer num, repeatedly add all its digits until the result has only one digit, and return it.

Example 1:

Input: num = 38
Output: 2
Explanation: The process is
38 --> 3 + 8 --> 11
11 --> 1 + 1 --> 2
Since 2 has only one digit, return it.
Example 2:

Input: num = 0
Output: 0

Constraints:

0 <= num <= 231 - 1

```
class Solution {
    public int addDigits(int num) {
        if(num<10)
            return num;
        int res=num;
        while(res>=10){
            res=addsingleDigit(res);
        }
        return res;
    }
    public int addsingleDigit(int no){
        int out=0;
        while(no!=0){
            out+=no%10;
            no/=10;
        }
        return out;
    }
}
```

----------Single Number III
----------Given an integer array 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. You can return the answer in any order.

You must write an algorithm that runs in linear runtime complexity and uses only constant extra space.

Example 1:

Input: nums = [1,2,1,3,2,5]
Output: [3,5]
Explanation: [5, 3] is also a valid answer.
Example 2:

Input: nums = [-1,0]
Output: [-1,0]
Example 3:

Input: nums = [0,1]
Output: [1,0]

Constraints:

2 <= nums.length <= 3 * 104
-231 <= nums[i] <= 231 - 1
Each integer in nums will appear twice, only two integers will appear once.

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

        int xor = 0, res1 = 0, res2 = 0;
        for (int i : nums) xor ^= i;

        int sn = xor & ~(xor - 1);

        for (int i = 0; i < nums.length; i++)
            if ((nums[i] & sn) != 0) res1 ^= nums[i];
            else res2 ^= nums[i];

        return new int[]{res1, res2};

    }
}
```

----------Ugly Number
----------An ugly number is a positive integer whose prime factors are limited to 2, 3, and 5.

Given an integer n, return true if n is an ugly number.

Example 1:

Input: n = 6
Output: true
Explanation: 6 = 2 × 3
Example 2:

Input: n = 8
Output: true
Explanation: 8 = 2 × 2 × 2
Example 3:

Input: n = 14
Output: false
Explanation: 14 is not ugly since it includes the prime factor 7.
Example 4:

Input: n = 1
Output: true
Explanation: 1 has no prime factors, therefore all of its prime factors are limited to 2, 3, and 5.

Constraints:

-231 <= n <= 231 - 1

```
class Solution {
public boolean isUgly(int n) {
if(n==0)
{
return false;
}
if(n==1)
{
return true;
}
int value=n;
int a[]={2,3,5};
for(int i=0;i<a.length;i++)
{
while(value%a[i]==0)
{
value=value/a[i];
}
}
return value==1;
}
}
```

----------Ugly Number II
----------An ugly number is a positive integer whose prime factors are limited to 2, 3, and 5.

Given an integer n, return the nth ugly number.

Example 1:

Input: n = 10
Output: 12
Explanation: [1, 2, 3, 4, 5, 6, 8, 9, 10, 12] is the sequence of the first 10 ugly numbers.
Example 2:

Input: n = 1
Output: 1
Explanation: 1 has no prime factors, therefore all of its prime factors are limited to 2, 3, and 5.

Constraints:

1 <= n <= 1690

```
class Solution {
    public int nthUglyNumber(int n) {
        int[] ugly=new int[n];
        int i2=0,i3=0,i5=0,ugly2=2,ugly3=3,ugly5=5,nextugly=1;
         ugly[0]=1;
         for(int i=1;i<n;i++){
             nextugly=Math.min(Math.min(ugly2,ugly3),ugly5);
             ugly[i]=nextugly;
             if(nextugly==ugly2){
                 i2++;
                 ugly2=ugly[i2]*2;
             }
             if(nextugly==ugly3){
                 i3++;
                 ugly3=ugly[i3]*3;
             }
             if(nextugly==ugly5){
                 i5++;
                 ugly5=ugly[i5]*5;
             }
         }
         return ugly[n-1];
    }
}
```

----------Missing Number
----------Given an array nums containing n distinct numbers in the range [0, n], return the only number in the range that is missing from the array.

Example 1:

Input: nums = [3,0,1]

Output: 2
Explanation: n = 3 since there are 3 numbers, so all numbers are in the range [0,3]. 2 is the missing number in the range since it does not appear in nums.
Example 2:

Input: nums = [0,1]
Output: 2
Explanation: n = 2 since there are 2 numbers, so all numbers are in the range [0,2]. 2 is the missing number in the range since it does not appear in nums.
Example 3:

Input: nums = [9,6,4,2,3,5,7,0,1]
Output: 8
Explanation: n = 9 since there are 9 numbers, so all numbers are in the range [0,9]. 8 is the missing number in the range since it does not appear in nums.
Example 4:

Input: nums = [0]
Output: 1
Explanation: n = 1 since there is 1 number, so all numbers are in the range [0,1]. 1 is the missing number in the range since it does not appear in nums.

Constraints:

n == nums.length
1 <= n <= 104
0 <= nums[i] <= n
All the numbers of nums are unique.

```
class Solution {
    public int missingNumber(int[] nums) {
        int sum = (nums.length*(nums.length+1))/2 , sumSF = 0;
        for(int i : nums) sumSF += i;
        return sum - sumSF;
    }
}
```

----------Integer to English Words
----------Convert a non-negative integer num to its English words representation.

Example 1:

Input: num = 123
Output: "One Hundred Twenty Three"
Example 2:

Input: num = 12345
Output: "Twelve Thousand Three Hundred Forty Five"
Example 3:

Input: num = 1234567
Output: "One Million Two Hundred Thirty Four Thousand Five Hundred Sixty Seven"
Example 4:

Input: num = 1234567891
Output: "One Billion Two Hundred Thirty Four Million Five Hundred Sixty Seven Thousand Eight Hundred Ninety One"

Constraints:

0 <= num <= 231 - 1

```
class Solution {
    private static final String[] ones = new String[]
{"One","Two","Three","Four","Five","Six","Seven","Eight","Nine"};
    private static final String[] tens = new String[]
{"Ten","Twenty","Thirty","Forty","Fifty","Sixty","Seventy","Eighty","Ninety"};
    private static final String[] special = new String[]
{"Eleven","Twelve","Thirteen","Fourteen","Fifteen","Sixteen","Seventeen","Eighteen","Nineteen
"
    };

    private static final int billion = 1000000000;
    private static final int million = 1000000;
    private static final int thousand = 1000;

    public String numberToWords(int num) {
        if (num == 0) {
            return "Zero";
        }

        StringBuilder sb = new StringBuilder();
        boolean flag = false;
        if (num >= billion) {
            int b = num / billion;
            getSegment(b, sb);
            sb.append(" Billion");
            num = num % billion;
            flag = true;
        }
        if (num >= million) {
            if (flag) sb.append(' ');
            int m = num / million;
            getSegment(m, sb);
            sb.append(" Million");
            num = num % million;
            flag = true;
        }
        if (num >= thousand) {
            if (flag) sb.append(' ');
            int  t = num / thousand;
            getSegment(t, sb);
```

```
            sb.append(" Thousand");
            num = num % thousand;
            flag = true;
        }

        if (flag && num > 0) sb.append(' ');
        getSegment(num, sb);
        return sb.toString();
    }

    private void getSegment(int val, StringBuilder sb) {
        boolean flag = false;

        if (val >= 100) {
            int hunds = val / 100;
            sb.append(ones[hunds-1]);
            sb.append(' ');
            sb.append("Hundred");
            val = val % 100;
            flag = true;
        }
        if (val >= 10) {
            if (flag) sb.append(' ');
            if (val >= 11 && val <= 19) {
                int ind = val - 10;
                sb.append(special[ind-1]);
                val = 0;
            } else {
                int t = val / 10;
                sb.append(tens[t-1]);
                val = val % 10;
            }
            flag = true;
        }
        if (val >= 1) {
            if (flag) sb.append(' ');
            sb.append(ones[val-1]);
        }

    }
}
```

----------H-Index----------Given an array of integers citations where citations[i] is the number of citations a researcher received for their ith paper, return compute the researcher's h-index.

According to the definition of h-index on Wikipedia: A scientist has an index h if h of their n papers have at least h citations each, and the other n − h papers have no more than h citations each.

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

Example 1:

Input: citations = [3,0,6,1,5]
Output: 3
Explanation: [3,0,6,1,5] 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, their h-index is 3.
Example 2:

Input: citations = [1,3,1]
Output: 1

Constraints:

n == citations.length
1 <= n <= 5000
0 <= citations[i] <= 1000

```
class Solution {
    public int hIndex(int[] citations) {

        int[] frequencies = new int[citations.length + 1];
        int current;
        int solution = 0;

        for (int i = 0; i < citations.length; i++) {
            current = citations[i];
            if (current > citations.length)
                current = citations.length;
            frequencies[current]++;
        }

        int total = 0;
        for (int i = frequencies.length - 1; i > 0; i--) {
            total += frequencies[i];
            if (total >= i)
                return Math.min(total, i);
        }
        return total;
    }
}
```

----------H-Index II
----------Given an array of integers citations where citations[i] is the number of citations a researcher received for their ith paper and citations is sorted in an ascending order, return compute the researcher's h-index.

According to the definition of h-index on Wikipedia: A scientist has an index h if h of their n papers have at least h citations each, and the other n − h papers have no more than h citations each.

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

You must write an algorithm that runs in logarithmic time.

Example 1:

Input: citations = [0,1,3,5,6]
Output: 3
Explanation: [0,1,3,5,6] means the researcher has 5 papers in total and each of them had received 0, 1, 3, 5, 6 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, their h-index is 3.
Example 2:

Input: citations = [1,2,100]
Output: 2

Constraints:

n == citations.length
1 <= n <= 105
0 <= citations[i] <= 1000
citations is sorted in ascending order.

```
class Solution {
    public int hIndex(int[] citations) {
        int h_index = 0;
        for (int i = 0, j = citations.length - 1; i < citations.length; i ++, j --){
            if (citations[j] >= i + 1){
                h_index ++;
            }
            else{
                break;
            }
        }
        return h_index;
    }
}
```

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

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

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

Example 1:

Input: n = 5, bad = 4
Output: 4
Explanation:
call isBadVersion(3) -> false
call isBadVersion(5) -> true
call isBadVersion(4) -> true
Then 4 is the first bad version.
Example 2:

Input: n = 1, bad = 1
Output: 1

Constraints:

1 <= bad <= n <= 231 - 1

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

public class Solution extends VersionControl {
    public int firstBadVersion(int n) {
        int low  = 1,high = n;
        while(low < high){
            int mid = low + (high - low)/2;
            if(isBadVersion(mid))high = mid;
            else low = mid + 1;
        }
        return high;
    }
}
```

----------Perfect Squares
----------Given an integer n, return the least number of perfect square numbers that sum to n.

A perfect square is an integer that is the square of an integer; in other words, it is the product of some integer with itself. For example, 1, 4, 9, and 16 are perfect squares while 3 and 11 are not.

Example 1:

Input: n = 12
Output: 3
Explanation: 12 = 4 + 4 + 4.
Example 2:

Input: n = 13
Output: 2
Explanation: 13 = 4 + 9.

Constraints:

1 <= n <= 104

```
class Solution {
public int numSquares(int n) {
        Queue<Integer> queue = new ArrayDeque<>();

        // For memorization
        boolean[] visited = new boolean[n + 1];
        visited[n] = true;

        int steps = 0;
        queue.add(n);

        while(!queue.isEmpty()) {

            steps += 1;

            int size = queue.size();
            for(int i = 0; i < size; ++i) {

                int val = queue.poll();

                /* 24ms Runtime
                if(val == 0) {
                    return --steps;
                }
                */
                for(int j = 1; j * j <= val; ++j) {
                    int res = val - j * j;

                    if(res == 0) { // 8ms Runtime
                        return steps;
                    }

                    if(!visited[res]) {
                        queue.add(res);
                        visited[res] = true;
                    }
                }

            }
        }

        return -1;
    }
}
```

----------Expression Add Operators
----------Given a string num that contains only digits and an integer target, return all possibilities to insert the binary operators '+', '-', and/or '*' between the digits of num so that the resultant expression evaluates to the target value.

Note that operands in the returned expressions should not contain leading zeros.

Example 1:

Input: num = "123", target = 6
Output: ["1*2*3","1+2+3"]
Explanation: Both "1*2*3" and "1+2+3" evaluate to 6.
Example 2:

Input: num = "232", target = 8
Output: ["2*3+2","2+3*2"]
Explanation: Both "2*3+2" and "2+3*2" evaluate to 8.
Example 3:

Input: num = "105", target = 5
Output: ["1*0+5","10-5"]
Explanation: Both "1*0+5" and "10-5" evaluate to 5.
Note that "1-05" is not a valid expression because the 5 has a leading zero.
Example 4:

Input: num = "00", target = 0
Output: ["0*0","0+0","0-0"]
Explanation: "0*0", "0+0", and "0-0" all evaluate to 0.
Note that "00" is not a valid expression because the 0 has a leading zero.
Example 5:

Input: num = "3456237490", target = 9191
Output: []
Explanation: There are no expressions that can be created from "3456237490" to evaluate to 9191.

Constraints:

1 <= num.length <= 10
num consists of only digits.
-231 <= target <= 231 - 1

```
class Solution {

  public ArrayList<String> answer;
  public String digits;
  public long target;

  public void recurse(
      int index, long previousOperand, long currentOperand, long value, ArrayList<String> ops) {
    String nums = this.digits;

    // Done processing all the digits in num
    if (index == nums.length()) {

      // If the final value == target expected AND
      // no operand is left unprocessed
      if (value == this.target && currentOperand == 0) {
        StringBuilder sb = new StringBuilder();
```

```
      ops.subList(1, ops.size()).forEach(v -> sb.append(v));
      this.answer.add(sb.toString());
    }
    return;
  }

  // Extending the current operand by one digit
  currentOperand = currentOperand * 10 + Character.getNumericValue(nums.charAt(index));
  String current_val_rep = Long.toString(currentOperand);
  int length = nums.length();

  // To avoid cases where we have 1 + 05 or 1 * 05 since 05 won't be a
  // valid operand. Hence this check
  if (currentOperand > 0) {

    // NO OP recursion
    recurse(index + 1, previousOperand, currentOperand, value, ops);
  }

  // ADDITION
  ops.add("+");
  ops.add(current_val_rep);
  recurse(index + 1, currentOperand, 0, value + currentOperand, ops);
  ops.remove(ops.size() - 1);
  ops.remove(ops.size() - 1);

  if (ops.size() > 0) {

    // SUBTRACTION
    ops.add("-");
    ops.add(current_val_rep);
    recurse(index + 1, -currentOperand, 0, value - currentOperand, ops);
    ops.remove(ops.size() - 1);
    ops.remove(ops.size() - 1);

    // MULTIPLICATION
    ops.add("*");
    ops.add(current_val_rep);
    recurse(
        index + 1,
        currentOperand * previousOperand,
        0,
        value - previousOperand + (currentOperand * previousOperand),
        ops);
    ops.remove(ops.size() - 1);
    ops.remove(ops.size() - 1);
  }
}

public List<String> addOperators(String num, int target) {

  if (num.length() == 0) {
    return new ArrayList<String>();
  }
```

```
    this.target = target;
    this.digits = num;
    this.answer = new ArrayList<String>();
    this.recurse(0, 0, 0, 0, new ArrayList<String>());
    return this.answer;
  }
}
```
----------Move Zeroes
----------Given an integer array nums, move all 0's to the end of it while maintaining the relative order of the non-zero elements.

Note that you must do this in-place without making a copy of the array.

Example 1:

Input: nums = [0,1,0,3,12]
Output: [1,3,12,0,0]
Example 2:

Input: nums = [0]
Output: [0]

Constraints:

1 <= nums.length <= 104
-231 <= nums[i] <= 231 - 1

```
class Solution {
    public void moveZeroes(int[] nums) {
        int j = 0;
        int size = nums.length;
        for (int i = 0; i < size; i++) {
            if (nums[i] != 0) {
                nums[j] = nums[i];
                j++;
            }
        }
        while (j < size) {
            nums[j] = 0;
            j++;
        }
    }
}
```
----------Peeking Iterator
----------Design an iterator that supports the peek operation on an existing iterator in addition to the hasNext and the next operations.

Implement the PeekingIterator class:

PeekingIterator(Iterator<int> nums) Initializes the object with the given integer iterator iterator.

int next() Returns the next element in the array and moves the pointer to the next element.
boolean hasNext() Returns true if there are still elements in the array.
int peek() Returns the next element in the array without moving the pointer.
Note: Each language may have a different implementation of the constructor and Iterator, but they all support the int next() and boolean hasNext() functions.

Example 1:

Input
["PeekingIterator", "next", "peek", "next", "next", "hasNext"]
[[[1, 2, 3]], [], [], [], [], []]
Output
[null, 1, 2, 2, 3, false]

Explanation
PeekingIterator peekingIterator = new PeekingIterator([1, 2, 3]); // [1,2,3]
peekingIterator.next(); // return 1, the pointer moves to the next element [1,2,3].
peekingIterator.peek(); // return 2, the pointer does not move [1,2,3].
peekingIterator.next(); // return 2, the pointer moves to the next element [1,2,3]
peekingIterator.next(); // return 3, the pointer moves to the next element [1,2,3]
peekingIterator.hasNext(); // return False

Constraints:

1 <= nums.length <= 1000
1 <= nums[i] <= 1000
All the calls to next and peek are valid.
At most 1000 calls will be made to next, hasNext, and peek.

```
import java.util.NoSuchElementException;
class PeekingIterator implements Iterator<Integer> {
    Integer next;
    Iterator<Integer> iter;
    boolean noSuchElement;

    public PeekingIterator(Iterator<Integer> iterator) {
    // initialize any member here.
    iter = iterator;
        advanceIter();
    }

    // Returns the next element in the iteration without advancing the iterator.
    public Integer peek() {
        // you should confirm with interviewer what to return/throw
        // if there are no more values
        return next;
    }

    // hasNext() and next() should behave the same as in the Iterator interface.
    // Override them if needed.
```

```
    @Override
    public Integer next() {
        if (noSuchElement)
            throw new NoSuchElementException();
        Integer res = next;
        advanceIter();
        return res;
    }

    @Override
    public boolean hasNext() {
        return !noSuchElement;
    }

    private void advanceIter() {
        if (iter.hasNext()) {
            next = iter.next();
        } else {
            noSuchElement = true;
        }
    }
}
```

----------Find the Duplicate Number
----------Given an array of integers nums containing n + 1 integers where each integer is in the range [1, n] inclusive.

There is only one repeated number in nums, return this repeated number.

You must solve the problem without modifying the array nums and uses only constant extra space.

Example 1:

Input: nums = [1,3,4,2,2]
Output: 2
Example 2:

Input: nums = [3,1,3,4,2]
Output: 3
Example 3:

Input: nums = [1,1]
Output: 1
Example 4:

Input: nums = [1,1,2]
Output: 1

Constraints:

1 <= n <= 105

nums.length == n + 1
1 <= nums[i] <= n
All the integers in nums appear only once except for precisely one integer which appears two or more times.

Follow up:

How can we prove that at least one duplicate number must exist in nums?
Can you solve the problem in linear runtime complexity?

```
class Solution {
    public int findDuplicate(int[] nums) {
        int duplicate = -1;
        for (int i = 0; i < nums.length; i++) {
            int cur = Math.abs(nums[i]);
            if (nums[cur] < 0) {
                duplicate = cur;
                break;
            }
            nums[cur] *= -1;
        }

        // Restore numbers
        for (int i = 0; i < nums.length; i++)
            nums[i] = Math.abs(nums[i]);

        return duplicate;
    }
}
```

Game of Life

According to 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."

The board is made up of an m x n grid of cells, where each cell has an initial state: live (represented by a 1) or dead (represented by a 0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules (taken from the above Wikipedia article):

Any live cell with fewer than two live neighbors dies as if caused by under-population.
Any live cell with two or three live neighbors lives on to the next generation.
Any live cell with more than three live neighbors dies, as if by over-population.
Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.
The next state is created by applying the above rules simultaneously to every cell in the current state, where births and deaths occur simultaneously. Given the current state of the m x n grid board, return the next state.

Example 1:

Input: board = [[0,1,0],[0,0,1],[1,1,1],[0,0,0]]

Output: [[0,0,0],[1,0,1],[0,1,1],[0,1,0]]
Example 2:

Input: board = [[1,1],[1,0]]
Output: [[1,1],[1,1]]

Constraints:

m == board.length
n == board[i].length
1 <= m, n <= 25
board[i][j] is 0 or 1.

```
class Solution {
    public void gameOfLife(int[][] board) {
        int[][] nextState = new int[board.length][board[0].length];
        for(int i=0;i<board.length;++i){
            for(int j=0;j<board[i].length;++j){
                // getting state of the current cell based on four cases and updating it in the next
state grid
                nextState[i][j] = getCellState(i,j,board);
            }
        }

        // Once the result is calculated, we update the cells of the grid with our resultant grid cells
        for(int i=0;i<board.length;++i){
            for(int j=0;j<board[i].length;++j){
                board[i][j] = nextState[i][j];
            }
        }
    }

    public int getCellState(int row,int col, int[][] board){
        // checking for neighbours in all the 8 directions of the grid
        int neighbourCount = 0;
        int rowLength = board.length-1;
        int colLength = board[0].length-1;
        if(row > 0  && col > 0)
            neighbourCount += board[row-1][col-1];
        if(row > 0){
            neighbourCount += board[row-1][col];
            if(col < colLength){
                neighbourCount += board[row-1][col+1];
            }
        }
        if(col < colLength){
            neighbourCount += board[row][col+1];
        }
        if(col > 0){
            neighbourCount += board[row][col-1];
            if(row < rowLength){
                neighbourCount += board[row+1][col-1];
```

```
            }
        }
        if(row < rowLength){
            neighbourCount += board[row+1][col];
        }
        if(row < rowLength  && col < colLength)
            neighbourCount += board[row+1][col+1];

        // Case 4 : Any dead cell with exactly three live neighbors becomes a live cell, as if by
reproduction.
        // So we return 1
        if(board[row][col] == 0 && neighbourCount == 3)
            return 1;

        // Case 2 : Any live cell with two or three live neighbors lives on to the next generation.
        // So we return 1
        else if(board[row][col] == 1 && (neighbourCount==2 || neighbourCount==3))
            return 1;

        // For Case 1 and Case 3
        else
            return 0;
    }

}
```

Word Pattern

Given a pattern and a string s, find if s 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 s.

Example 1:

Input: pattern = "abba", s = "dog cat cat dog"
Output: true
Example 2:

Input: pattern = "abba", s = "dog cat cat fish"
Output: false
Example 3:

Input: pattern = "aaaa", s = "dog cat cat dog"
Output: false
Example 4:

Input: pattern = "abba", s = "dog dog dog dog"
Output: false

Constraints:

1 <= pattern.length <= 300

pattern contains only lower-case English letters.
1 <= s.length <= 3000
s contains only lower-case English letters and spaces '

```
class Solution {
    public boolean wordPattern(String pattern, String str) {
        String[] words = str.split(" ");
        if(pattern.length() != words.length){
            return false;
        }
        HashMap<Character, String> map = new HashMap();
        for(int i = 0; i < pattern.length();i++){
            char seen = pattern.charAt(i);
            if(map.containsKey(seen)){
                if(!map.get(seen).equals(words[i])){
                    return false;
                }
            }
            else{
                if(map.containsValue(words[i])){
                    return false;
                }
                map.put(seen, words[i]);
            }
        }
        return true;
    }
}
```

----------Nim Game
----------You are playing the following Nim Game with your friend:

Initially, there is a heap of stones on the table.
You and your friend will alternate taking turns, and you go first.
On each turn, the person whose turn it is will remove 1 to 3 stones from the heap.
The one who removes the last stone is the winner.
Given n, the number of stones in the heap, return true if you can win the game assuming both you and your friend play optimally, otherwise return false.

Example 1:

Input: n = 4
Output: false
Explanation: These are the possible outcomes:
1. You remove 1 stone. Your friend removes 3 stones, including the last stone. Your friend wins.
2. You remove 2 stones. Your friend removes 2 stones, including the last stone. Your friend wins.
3. You remove 3 stones. Your friend removes the last stone. Your friend wins.
In all outcomes, your friend wins.
Example 2:

Input: n = 1
Output: true

Example 3:

Input: n = 2
Output: true

Constraints:

1 <= n <= 231 - 1

```
class Solution {
    public boolean canWinNim(int n) {
        return (n % 4 != 0);
    }
}
```

----------Find Median from Data Stream----------The median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value and the median is the mean of the two middle values.

For example, for arr = [2,3,4], the median is 3.
For example, for arr = [2,3], the median is (2 + 3) / 2 = 2.5.
Implement the MedianFinder class:

MedianFinder() initializes the MedianFinder object.
void addNum(int num) adds the integer num from the data stream to the data structure.
double findMedian() returns the median of all elements so far. Answers within 10-5 of the actual answer will be accepted.

Example 1:

Input
["MedianFinder", "addNum", "addNum", "findMedian", "addNum", "findMedian"]
[[], [1], [2], [], [3], []]
Output
[null, null, null, 1.5, null, 2.0]

Explanation
MedianFinder medianFinder = new MedianFinder();
medianFinder.addNum(1);   // arr = [1]
medianFinder.addNum(2);   // arr = [1, 2]
medianFinder.findMedian(); // return 1.5 (i.e., (1 + 2) / 2)
medianFinder.addNum(3);   // arr[1, 2, 3]
medianFinder.findMedian(); // return 2.0

Constraints:

-105 <= num <= 105
There will be at least one element in the data structure before calling findMedian.
At most 5 * 104 calls will be made to addNum and findMedian.

Follow up:

If all integer numbers from the stream are in the range [0, 100], how would you optimize your solution?
If 99% of all integer numbers from the stream are in the range [0, 100], how would you optimize your solution?

```
import java.util.Comparator;
import java.util.PriorityQueue;

/**
 * Return the median of the data stream.
 *
 * The idea is to maintain two ordered list holding respectively the
 *  two halves of a data stream.
 *
 * The insertion/update of the data would then have the complexity of O(nlgn)
 *  while we would have a constant O(1) complexity for the median calculation.
 *
 * Definition of Median:
 *
 * mid = size / 2
 * If size(data_stream) % 2 == 1 then
 *     median = data_stream[mid]
 * If size(data_stream) % 2 == 0 then
 *     median = data_stream[mid-1] + data_stream[mid]
 *
 */
class MedianFinder {

    /**
     * A comparator to have a descending order.
     */
    class Descend implements Comparator<Integer>
    {
        public int compare(Integer A, Integer B) {
            return B - A;
        }
    }

    private static final Integer INIT_CAPACITY = 1000;

    //! Maintain the first half list in the descending order, so that
    //   we could benefit from the O(nlgn) operators peek() and poll().
    private PriorityQueue<Integer> firstHalf =
            new PriorityQueue<Integer>(INIT_CAPACITY, new Descend());

    //! Use the default (ascending) order for the second half list.
    private PriorityQueue<Integer> secondHalf =
            new PriorityQueue<Integer>(INIT_CAPACITY);

    private double median = Double.MAX_VALUE;

    /**
     *  Attribute the incoming data and update the median.
```

```
     * @param num
     */
    public void addNum(int num) {
        if(num < median) {
            firstHalf.add(num);
        } else {
            secondHalf.add(num);
        }

        updateMedian();
    }

    /**
     *  Balance the two halves to keep the constrain that
     *    | size(firstHalf) - size(secondHalf) | <= 1
     */
    private void updateMedian() {
        int firstHalfSize = firstHalf.size();
        int secondHalfSize = secondHalf.size();

        // The lists are balanced.
        if(firstHalfSize == secondHalfSize) {
            median = (firstHalf.peek() + secondHalf.peek())/2.0;

        } else if(firstHalfSize > secondHalfSize) {
            // The lists are still balanced.
            if(firstHalfSize - secondHalfSize == 1) {
                median = firstHalf.peek();
            } else {
                // re-balancing by moving last element from the first half to second.
                int last_firstHalf = firstHalf.poll();
                secondHalf.offer(last_firstHalf);
                // recursively rebalancing the lists.
                // The recursion would not exceeds twice though.
                updateMedian();
            }
        } else {
            // firstHalfSize < secondHalfSize
            if(secondHalfSize - firstHalfSize == 1) {
                median = secondHalf.peek();
            } else {
                int first_secondHalf = secondHalf.poll();
                firstHalf.offer(first_secondHalf);
                updateMedian();
            }
        }
    }

    public double findMedian() {
        return median;
    }

};
----------------------------------------------Serialize and Deserialize Binary Tree
```

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

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

Clarification: The input/output format is the same as how LeetCode serializes a binary tree. You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.

Example 1:

Input: root = [1,2,3,null,null,4,5]
Output: [1,2,3,null,null,4,5]
Example 2:

Input: root = []
Output: []
Example 3:

Input: root = [1]
Output: [1]
Example 4:

Input: root = [1,2]
Output: [1,2]

Constraints:

The number of nodes in the tree is in the range [0, 104].
-1000 <= Node.val <= 1000

```
public class Codec {
    // Cannot be 0x00 or 0xff because this value will be used against a loaded
    // flag later.
    private static final char NULL_SYMBOL = 0x01;

    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
        StringBuilder buffer = new StringBuilder();
        visit(root, buffer);
        return buffer.toString();
    }

    // Since we only have to account for the range [-1000, 1000], we really only
    // need two bytes to represent each number. We will use a variable length
```

```
// encoding for the values, in the scheme:
// [flag, (value)] - where flag is required and value is optional. For a null
// flag (i.e.: when it is set to {@code NULL_SYMBOL}), no value will follow.
// For values, the flag will be overloaded by the upper byte. This is necessary
// to encode the signess of the value. The endianess will be swapped, purely
// to further optimize the amount of logic needed. For example:
//     15 => [0x00, 0x0f00]
//     -1 => [0xff, 0x1000]
private static void visit(TreeNode node, StringBuilder output) {
    if (node == null) {
        output.append(NULL_SYMBOL);
        return;
    }
    if (node != null) {
        int v = node.val;
        output.append((char) ((v >> 16) & 0xff));
        for (int i = 0; i < 2; ++i) {
            output.append((char) (v & 0xff));
            v >>= 8;
        }
        visit(node.left, output);
        visit(node.right, output);
    }
}


// Decodes your encoded data to tree.
public TreeNode deserialize(String data) {
    char[] c = data.toCharArray();
    return deserialize(new InputQueue(data.toCharArray()));
}

private static TreeNode deserialize(InputQueue q) {
    TreeNode node = produceNode(q);

    if (node == null) {
        return null;
    }

    node.left = deserialize(q);
    node.right = deserialize(q);

    return node;
}

// nullable
private static TreeNode produceNode(InputQueue q) {
    char flag = q.getNext();
    if (flag == NULL_SYMBOL) {
        return null;
    }

    // Use the flag to set the signess of the value,
    // we need to replicate the signess in the upper
```

```
        // two bytes.
        int v = 0;
        v |= flag;
        v <<= 8;
        v |= flag;
        v <<= 16;
        for (int i = 0; i < 2; ++i) {
            int component = q.getNext();
            v |= (component << (i * 8));
        }
        return new TreeNode(v);
    }

    private static class InputQueue {
        private final char[] input;
        private int counter;

        private InputQueue(char[] input) {
            this.input = input;
            counter = 0;
        }

        private char getNext() {
            return input[counter++];
        }
    }
}
```

Bulls and Cows

You are playing the Bulls and Cows game with your friend.

You write down a secret number and ask your friend to guess what the number is. When your friend makes a guess, you provide a hint with the following info:

The number of "bulls", which are digits in the guess that are in the correct position.
The number of "cows", which are digits in the guess that are in your secret number but are located in the wrong position. Specifically, the non-bull digits in the guess that could be rearranged such that they become bulls.
Given the secret number secret and your friend's guess guess, return the hint for your friend's guess.

The hint should be formatted as "xAyB", where x is the number of bulls and y is the number of cows. Note that both secret and guess may contain duplicate digits.

Example 1:

Input: secret = "1807", guess = "7810"
Output: "1A3B"
Explanation: Bulls are connected with a '|' and cows are underlined:
"1807"
  |
"7810"
Example 2:

Input: secret = "1123", guess = "0111"
Output: "1A1B"
Explanation: Bulls are connected with a '|' and cows are underlined:
"1123" "1123"
  | or |
"0111" "0111"
Note that only one of the two unmatched 1s is counted as a cow since the non-bull digits can only be rearranged to allow one 1 to be a bull.
Example 3:

Input: secret = "1", guess = "0"
Output: "0A0B"
Example 4:

Input: secret = "1", guess = "1"
Output: "1A0B"

Constraints:

1 <= secret.length, guess.length <= 1000
secret.length == guess.length
secret and guess consist of digits only.

```
class Solution {
    public String getHint(String secret, String guess) {
        char [] s = secret.toCharArray();
        char [] g = guess.toCharArray();

        int [] count = new int[10];
        int a = 0;
        int b = 0;

        StringBuffer sb = new StringBuffer();

        for(int i=0; i<s.length; i++){
            if(s[i] == g[i]){
                g[i] = '*';
                a++;
            }else{
                count[s[i]-'0']++;
            }
        }

        for(int i=0; i<g.length; i++){
            if(g[i] != '*' && count[g[i]-'0'] > 0){
                count[g[i] - '0']--;
                b++;
            }
        }

        return sb.append(a).append("A").append(b).append("B").toString();
```

```
    }
}
```
----------Longest Increasing Subsequence
----------Given an integer array nums, return the length of the longest strictly increasing subsequence.

A subsequence is a sequence that can be derived from an array by deleting some or no elements without changing the order of the remaining elements. For example, [3,6,2,7] is a subsequence of the array [0,3,1,6,2,2,7].

Example 1:

Input: nums = [10,9,2,5,3,7,101,18]
Output: 4
Explanation: The longest increasing subsequence is [2,3,7,101], therefore the length is 4.
Example 2:

Input: nums = [0,1,0,3,2,3]
Output: 4
Example 3:

Input: nums = [7,7,7,7,7,7,7]
Output: 1

Constraints:

1 <= nums.length <= 2500
-104 <= nums[i] <= 104

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

        /*
        Approach: Suppose we will be storing all the subsequences with distinct lengths we've found so far.
                Go through each item in the input array.
                If the item is smaller than any of the found items so far, go to the subsequences, find the
                first (smallest) subseq with last item larger than current item,
                e.g. subseqs: [[2],[2,4,7],[2,4,7,11]] and current item is 5, so we will target subseq [2,4,7].
                Replace the last item of that subseq with the current item, i.e. [2,4,7] -> [2,4,5].
                This will potentially allow us to add more items to that subseq later.
                Otherwise, if the item is larger than all the items found so far, add that to the largest subseq.

                **** Detailed approach ****
                Firstly, since we are only using the last items if the stored subsequences, instead of storing the
                whole subsequences, it's fine to just store the last indexes of the subsequences.
```

```
        Maintain an array (subseqLenLastItemMap) where each index represent length of a
subsequence -1
        and the content in that index represents the last (so largest) item in the
subsequence.
        The array will by default always be sorted, as longer subsequence will surely have
larger last item.

        Have a variable maxSubseqLen, which will store the length of the longest
subsequence found yet.
        As a consequence, it will also indicate the next index till which the
subseqLenLastItemMap is filled.

        For each item in the input array, Arrays.binarySearch() for that item in the
subseqLenLastItemMap.
        Only search in already added subsequence lengths (index 0 to maxSubseqLen-1).

        [Case 1]. If the result of the search is non-negative, it indicates that the item is
found,
        and the item is not the last item of any subsequence. Since in this problem
        we are not considering duplicate items (as we want strictly increasing subseq),
ignore.

        [Case 2]. If the result of the search is negative, it indicates that the item is not found.
        In that case, the search result returns (-(insertion point) - 1). The insertion point is
the point at
        which the key would be inserted into the array.
        We extract the insertion point by doing (-(returnValue) + 1).

        [Case 2a]. If the item is less than the current found last item of max subseq,
        e.g. searched for item: 5 where subseqLenLastItemMap is [2,6,7,0,0,0], the insertion
point will be 1.
        So we will store the item in that index, making subseqLenLastItemMap [2,5,7,0,0,0].
        Functionally, decreasing the last item of the subsequence, as this will potentially
allow us to
        add more items to that subseq later.

        [Case 2b].  If the item is greater than the last item of max subseq,
        e.g. searched for item: 11 where subseqLenLastItemMap is [2,6,7,0,0,0], the
insertion point
        will be 3 (which will be equal to maxSubseqLen).
        So, this item is larger than all items in each subsequences. So we will add it to the
max subseq.
        So we will store the item in that index, making subseqLenLastItemMap
[2,6,7,11,0,0].
        Functionally, we just found a larger subsequence! So we will increment
maxSubseqLen.

    Complexity analysis: Time: O(n log(n)) as doing binary search O(log(n) for each element
O(n),
                    Space: O(n) for subseqLenLastItemMap
    */


    int[] subseqLenLastItemMap = new int[nums.length];
```

```
        int maxSubseqLen = 0;                       //Length of the longest subsequence

        for(int num : nums){                        //For each item

            int i = Arrays.binarySearch(            //Search for insertion index in
subseqLenLastItemMap
                subseqLenLastItemMap, 0, maxSubseqLen,
                num);


            if(i<0){                                //Item not found (Case 2 (2a or 2b))
              i = -(i+1);                           //Extract insertion index

              subseqLenLastItemMap[i] = num;        //Replace last item of existing subseq, or
add new longer subseq

                if(!(i < maxSubseqLen))             //Case 2b: added new longer subseq
                    maxSubseqLen++;                 //Increment max subseq len
            }
        }

        return maxSubseqLen;
    }
}
```

----------Remove Invalid Parentheses----------Given a string s that contains parentheses and letters, remove the minimum number of invalid parentheses to make the input string valid.

Return all the possible results. You may return the answer in any order.

Example 1:

Input: s = "()())()"
Output: ["(())()","()()()"]
Example 2:

Input: s = "(a)())()"
Output: ["(a())()","(a)()()"]
Example 3:

Input: s = ")("
Output: [""]

Constraints:

1 <= s.length <= 25
s consists of lowercase English letters and parentheses '(' and ')'.
There will be at most 20 parentheses in s.

```
class Solution {
```

```
public List<String> removeInvalidParentheses(String s) {
    List<String> ans = new ArrayList<>();
    removeLeft(s, ans, new char[s.length()], 0, 0, 0);
    return ans;
}

private static void removeLeft(String s, List<String> ans, char[] buf, int pos, int start, int last) {
    int p = start, b = 0;
    for (int i = start; i < s.length(); ++i) {
        if (s.charAt(i) == '(') {
            ++b;
        } else if (s.charAt(i) == ')') {
            --b;
        }
        if (b <= 0) {
            p = i + 1;
        }
        if (b < 0) {
            for (int j = last; j <= i; ++j) {
                if (s.charAt(j) == ')' && (j == last || s.charAt(j - 1) != ')')) {
                    s.getChars(last, j, buf, pos);
                    removeLeft(s, ans, buf, pos + j - last, i + 1, j + 1);
                }
            }
            return;
        }
    }
    s.getChars(last, p, buf, pos);
    int rem = b + (last - pos); // total number of parentheses to remove, including already
removed
    removeRight(s, ans, buf, buf.length - rem, buf.length - rem, s.length() - 1, s.length() - 1, p);
}

private static void removeRight(String s, List<String> ans, char[] buf, int pos, int len,
        int start, int last, int p) {
    int b = 0;
    for (int i = start; i >= p; --i) {
        if (s.charAt(i) == ')') {
            ++b;
        } else if (s.charAt(i) == '(') {
            --b;
        }
        if (b < 0) {
            for (int j = last; j >= i; --j) {
                if (s.charAt(j) == '(' && (j == last || s.charAt(j + 1) != '(')) {
                    s.getChars(j + 1, last + 1, buf, pos - (last - j));
                    removeRight(s, ans, buf, pos - (last - j), len, i - 1, j - 1, p);
                }
            }
            return;
        }
    }
    s.getChars(p, last + 1, buf, pos - (last + 1 - p));
```

```
    ans.add(new String(buf, 0, len));
}
}
```
----------Range Sum Query - Immutable----------Given an integer array nums, handle multiple queries of the following type:

Calculate the sum of the elements of nums between indices left and right inclusive where left <= right.
Implement the NumArray class:

NumArray(int[] nums) Initializes the object with the integer array nums.
int sumRange(int left, int right) Returns the sum of the elements of nums between indices left and right inclusive (i.e. nums[left] + nums[left + 1] + ... + nums[right]).

Example 1:

Input
["NumArray", "sumRange", "sumRange", "sumRange"]
[[[-2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]
Output
[null, 1, -1, -3]

Explanation
NumArray numArray = new NumArray([-2, 0, 3, -5, 2, -1]);
numArray.sumRange(0, 2); // return (-2) + 0 + 3 = 1
numArray.sumRange(2, 5); // return 3 + (-5) + 2 + (-1) = -1
numArray.sumRange(0, 5); // return (-2) + 0 + 3 + (-5) + 2 + (-1) = -3

Constraints:

1 <= nums.length <= 104
-105 <= nums[i] <= 105
0 <= left <= right < nums.length
At most 104 calls will be made to sumRange.

```
class NumArray {

    int[] leftSum;

    public NumArray(int[] nums) {
        leftSum = new int[nums.length];

        int total = 0;
        for(int i = 0; i < nums.length; i++){
            total = total + nums[i];
            leftSum[i] = total;
        }
    }

    public int sumRange(int left, int right) {
```

```
        int prefixLeft = ((left - 1) >= 0) ? leftSum[left - 1] : 0;
        int prefixRight = leftSum[right];
        return prefixRight - prefixLeft;
    }
}
```

----------Range Sum Query 2D - Immutable----------Given a 2D matrix matrix, handle multiple queries of the following type:

Calculate the sum of the elements of matrix inside the rectangle defined by its upper left corner (row1, col1) and lower right corner (row2, col2).
Implement the NumMatrix class:

NumMatrix(int[][] matrix) Initializes the object with the integer matrix matrix.
int sumRegion(int row1, int col1, int row2, int col2) Returns the sum of the elements of matrix inside the rectangle defined by its upper left corner (row1, col1) and lower right corner (row2, col2).

Example 1:

Input
["NumMatrix", "sumRegion", "sumRegion", "sumRegion"]
[[[[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]]], [2, 1, 4, 3], [1, 1, 2, 2], [1, 2, 2, 4]]
Output
[null, 8, 11, 12]

Explanation
NumMatrix numMatrix = new NumMatrix([[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]]);
numMatrix.sumRegion(2, 1, 4, 3); // return 8 (i.e sum of the red rectangle)
numMatrix.sumRegion(1, 1, 2, 2); // return 11 (i.e sum of the green rectangle)
numMatrix.sumRegion(1, 2, 2, 4); // return 12 (i.e sum of the blue rectangle)

Constraints:

m == matrix.length
n == matrix[i].length
1 <= m, n <= 200
-105 <= matrix[i][j] <= 105
0 <= row1 <= row2 < m
0 <= col1 <= col2 < n
At most 104 calls will be made to sumRegion.

```
class NumMatrix {
    int[][] sumFromOrigin;

    public NumMatrix(int[][] matrix) {
        if(matrix.length==0) return;

        sumFromOrigin=new int[matrix.length][matrix[0].length];
```

```
        sumFromOrigin[0][0]=matrix[0][0];

        for(int c=1;c<matrix[0].length;c++){
            sumFromOrigin[0][c]=sumFromOrigin[0][c -1] + matrix[0][c];
        }

        for(int r=1;r<matrix.length;r++){
           sumFromOrigin[r][0]=sumFromOrigin[r-1][0] + matrix[r][0];
        }

        for(int r=1;r<matrix.length;r++){
          for(int c=1;c<matrix[0].length;c++){
              sumFromOrigin[r][c]=matrix[r][c] + sumFromOrigin[r][c-1] + sumFromOrigin[r-1][c] -
sumFromOrigin[r-1][c-1];
          }
        }

    }

    public int sumRegion(int row1, int col1, int row2, int col2) {
        int result=sumFromOrigin[row2][col2];

        if(row1>0) result-=sumFromOrigin[row1-1][col2];
        if(col1>0) result-=sumFromOrigin[row2][col1-1];
        if(row1>0 && col1>0) result+=sumFromOrigin[row1-1][col1-1];

        return result;
    }
}
```

----------Additive Number----------

Additive number is a string whose digits can form additive sequence.

A valid additive sequence should contain at least three numbers. Except for the first two numbers, each subsequent number in the sequence must be the sum of the preceding two.

Given a string containing only digits '0'-'9', write a function to determine if it's an additive number.

Note: Numbers in the additive sequence cannot have leading zeros, so sequence 1, 2, 03 or 1, 02, 3 is invalid.

Example 1:

Input: "112358"
Output: true
Explanation: 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
Example 2:

Input: "199100199"
Output: true
Explanation: The additive sequence is: 1, 99, 100, 199.

1 + 99 = 100, 99 + 100 = 199

Constraints:

num consists only of digits '0'-'9'.
1 <= num.length <= 35

```
class Solution {
    public boolean isAdditiveNumber(String num) {
        return findIsAdditive(num,0,-1,-1,0);
    }

    private boolean findIsAdditive(String s, int index, long first, long second, int size) {
        //If we reach the end, check if we have more that 3 elements
        if(index>=s.length()) return size>=3;
        //Handling the case where starting index is 0
        if(s.charAt(index)=='0'){
            if(size<2 || first+second==0) return findIsAdditive(s,index+1,second,0,size+1);
            return false;
        }
        long num=0;
        //For each new num, check if it is the sum of first and second.
        for(int i=index;i<s.length();i++){
            char c = s.charAt(i);
            num = num*10+(c-'0');
            //num < 0 implies that the data has overflown
            if(num<0) break;
            if(size<2 || first+second==num){
                boolean possible = findIsAdditive(s,i+1,second,num,size+1);
                if(possible) return true;
            }
        }
        return false;
    }
}
```

----------Range Sum Query - Mutable----------Given an integer array nums, handle multiple queries of the following types:

Update the value of an element in nums.
Calculate the sum of the elements of nums between indices left and right inclusive where left <= right.
Implement the NumArray class:

NumArray(int[] nums) Initializes the object with the integer array nums.
void update(int index, int val) Updates the value of nums[index] to be val.
int sumRange(int left, int right) Returns the sum of the elements of nums between indices left and right inclusive (i.e. nums[left] + nums[left + 1] + ... + nums[right]).

Example 1:

Input

["NumArray", "sumRange", "update", "sumRange"]
[[[1, 3, 5]], [0, 2], [1, 2], [0, 2]]
Output
[null, 9, null, 8]

Explanation
NumArray numArray = new NumArray([1, 3, 5]);
numArray.sumRange(0, 2); // return 1 + 3 + 5 = 9
numArray.update(1, 2);   // nums = [1, 2, 5]
numArray.sumRange(0, 2); // return 1 + 2 + 5 = 8

Constraints:

1 <= nums.length <= 3 * 104
-100 <= nums[i] <= 100
0 <= index < nums.length
-100 <= val <= 100
0 <= left <= right < nums.length
At most 3 * 104 calls will be made to update and sumRange.

```
// sqyared tree
class NumArray {

    private int[] b;
private int len;
private int[] nums;

public NumArray(int[] nums) {
    this.nums = nums;
    double l = Math.sqrt(nums.length);
    len = (int) Math.ceil(nums.length/l);
    b = new int [len];
    for (int i = 0; i < nums.length; i++)
        b[i / len] += nums[i];
}

public int sumRange(int i, int j) {
    int sum = 0;
    int startBlock = i / len;
    int endBlock = j / len;
    if (startBlock == endBlock) {
        for (int k = i; k <= j; k++)
            sum += nums[k];
    } else {
        for (int k = i; k <= (startBlock + 1) * len - 1; k++)
            sum += nums[k];
        for (int k = startBlock + 1; k <= endBlock - 1; k++)
            sum += b[k];
        for (int k = endBlock * len; k <= j; k++)
            sum += nums[k];
    }
    return sum;
}
```

```
public void update(int i, int val) {
    int b_l = i / len;
    b[b_l] = b[b_l] - nums[i] + val;
    nums[i] = val;
}
// Accepted
}

// segment tree -- best

/**
 * Your NumArray object will be instantiated and called as such:
 * NumArray obj = new NumArray(nums);
 * obj.update(index,val);
 * int param_2 = obj.sumRange(left,right);
 */

/*
Classic segment tree or fenwick tree (AKA Binary Index Tree) problem.
*/
class NumArray {
    Node root;

    public NumArray(int[] nums) {
        this.root = constructTree(nums, 0, nums.length - 1);
    }

    public void update(int index, int val) {
        updateTree(index, val, root);
    }

    private Node constructTree(int[] nums, int from, int to) {
        Node node = new Node(from, to);
        if (from == to) {
            node.sum = nums[from];
        } else {
            int mid = (from + to) / 2;
            node.left = constructTree(nums, from, mid);
            node.right = constructTree(nums, mid + 1, to);
            node.sum = node.left.sum + node.right.sum;
        }
        return node;
    }

    private void updateTree(int index, int value, Node node) {
        if (node.from == node.to && node.from == index) {
            node.sum = value;
            return;
        }
        int mid = (node.from + node.to) / 2;
        if (index <= mid) {
            updateTree(index, value, node.left);
        } else {
```

```
            updateTree(index, value, node.right);
        }
        node.sum = node.left.sum + node.right.sum;
    }

    public int sumRange(int left, int right) {
        return getTreeSum(right, root) - getTreeSum(left - 1, root);
    }

    private int getTreeSum(int index, Node node) {
        if (index >= node.to) {
            return node.sum;
        }
        if (index < node.from) {
            return 0;
        }
        return getTreeSum(index, node.left) + getTreeSum(index, node.right);
    }

    static class Node {
        int from;
        int to;
        int sum = 0;
        Node left = null;
        Node right = null;

        Node (int from, int to) {
            this.from = from;
            this.to = to;
        }
    }
}
```

----------Minimum Height Trees----------A tree is an undirected graph in which any two vertices are connected by exactly one path. In other words, any connected graph without simple cycles is a tree.

Given a tree of n nodes labelled from 0 to n - 1, and an array of n - 1 edges where edges[i] = [ai, bi] indicates that there is an undirected edge between the two nodes ai and bi in the tree, you can choose any node of the tree as the root. When you select a node x as the root, the result tree has height h. Among all possible rooted trees, those with minimum height (i.e. min(h)) are called minimum height trees (MHTs).

Return a list of all MHTs' root labels. You can return the answer in any order.

The height of a rooted tree is the number of edges on the longest downward path between the root and a leaf.

Example 1:

Input: n = 4, edges = [[1,0],[1,2],[1,3]]

Output: [1]
Explanation: As shown, the height of the tree is 1 when the root is the node with label 1 which is the only MHT.
Example 2:

Input: n = 6, edges = [[3,0],[3,1],[3,2],[3,4],[5,4]]
Output: [3,4]
Example 3:

Input: n = 1, edges = []
Output: [0]
Example 4:

Input: n = 2, edges = [[0,1]]
Output: [0,1]

Constraints:

1 <= n <= 2 * 104
edges.length == n - 1
0 <= ai, bi < n
ai != bi
All the pairs (ai, bi) are distinct.
The given input is guaranteed to be a tree and there will be no repeated edges.

```
class Solution {
    public List<Integer> findMinHeightTrees(int n, int[][] edges) {
        if(n==1){  // BASE CASE
            List<Integer> a = new ArrayList<>();
            a.add(0);
            return a;
        }
        int[] degree = new int[n]; // FOR STROING DEGREE OF EACH NODE IN TREE
        ArrayList<Integer>[] graph = new ArrayList[n]; // DECLARING ADJACENCY LIST
        for (int i = 0; i < n; i++) {
            graph[i] = new ArrayList<Integer>();
        }

        for(int[] edge : edges){
            degree[edge[0]]++; degree[edge[1]]++;
            graph[edge[0]].add(edge[1]);
            graph[edge[1]].add(edge[0]);
        }
        Queue<Integer> q = new LinkedList<>();
        List<Integer> leafs = new ArrayList();
        for(int i=0;i<n;i++){
            if(degree[i] ==1)q.offer(i); // ONLY LEAFS ARE ALLOWED TO ENTER IN QUEUE
        }
        while(n>2){ //  2 IS THE MAXIMUM NUMBER OF ROOTS WITH MINIMUM HEIGHT TREE
            n-=q.size();
            int size = q.size();
            while(size-->0){
```

```
                int node = q.poll(); //DELEATING THE LEAFS NODE FROM QUEUE
                for(int i : graph[node]){
                    degree[i]--; // MAKING NEW LEAFS NODE BY DECREASING DEGREE
                    if(degree[i]== 1)q.offer(i); // AND ADDING IF NODE POSSIBLE TO BE LEAFS
                }
            }
        }
        leafs.addAll(q);//REMAINING NODE THAT IS ROOT OF MHT IN Q
        return leafs;
    }
}
```

----------Burst Balloons----------You are given n balloons, indexed from 0 to n - 1. Each balloon is painted with a number on it represented by an array nums. You are asked to burst all the balloons.

If you burst the ith balloon, you will get nums[i - 1] * nums[i] * nums[i + 1] coins. If i - 1 or i + 1 goes out of bounds of the array, then treat it as if there is a balloon with a 1 painted on it.

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

Example 1:

Input: nums = [3,1,5,8]
Output: 167
Explanation:
nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []
coins = 3*1*5 + 3*5*8 + 1*3*8 + 1*8*1 = 167
Example 2:

Input: nums = [1,5]
Output: 10

Constraints:

n == nums.length
1 <= n <= 500
0 <= nums[i] <= 100

```
class Solution {
    public int maxCoins(int[] iNums) {
    int[] nums = new int[iNums.length + 2];
    int n = 1;
    for (int x : iNums) if (x > 0) nums[n++] = x;
    nums[0] = nums[n++] = 1;


    int[][] dp = new int[n][n];
    for (int k = 2; k < n; ++k)
        for (int left = 0; left < n - k; ++left) {
            int right = left + k;
```

```
            for (int i = left + 1; i < right; ++i)
                dp[left][right] = Math.max(dp[left][right],
                nums[left] * nums[i] * nums[right] + dp[left][i] + dp[i][right]);
        }

    return dp[0][n - 1];
}
}
```

----------Super Ugly Number

----------A super ugly number is a positive integer whose prime factors are in the array primes.

Given an integer n and an array of integers primes, return the nth super ugly number.

The nth super ugly number is guaranteed to fit in a 32-bit signed integer.

Example 1:

Input: n = 12, primes = [2,7,13,19]
Output: 32
Explanation: [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].
Example 2:

Input: n = 1, primes = [2,3,5]
Output: 1
Explanation: 1 has no prime factors, therefore all of its prime factors are in the array primes = [2,3,5].

Constraints:

1 <= n <= 106
1 <= primes.length <= 100
2 <= primes[i] <= 1000
primes[i] is guaranteed to be a prime number.
All the values of primes are unique and sorted in ascending order.

```
public class Solution {
    public int nthSuperUglyNumber(int n, int[] primes) {

        int[] indexs = new int[primes.length];
        int[] uglyNums = new int[n];
        uglyNums[0] = 1;

        for(int i = 1; i < n; ++i){
            int min = uglyNums[indexs[0]]*primes[0];

            for(int j = 1; j < primes.length; ++j){
                int temp = uglyNums[indexs[j]]*primes[j];

                if(min > temp & temp > 0)
```

```
                min = temp;
            }
            uglyNums[i] = min;
            for(int j = 0; j < primes.length; ++j){
                if(uglyNums[indexs[j]]*primes[j] == min)
                    ++indexs[j];
            }
        }
        return uglyNums[n-1];
    }
}
```

----------Count of Smaller Numbers After Self----------You are given an integer array nums and you have to return a new counts array. The counts array has the property where counts[i] is the number of smaller elements to the right of nums[i].

Example 1:

Input: nums = [5,2,6,1]
Output: [2,1,1,0]
Explanation:
To the right of 5 there are 2 smaller elements (2 and 1).
To the right of 2 there is only 1 smaller element (1).
To the right of 6 there is 1 smaller element (1).
To the right of 1 there is 0 smaller element.
Example 2:

Input: nums = [-1]
Output: [0]
Example 3:

Input: nums = [-1,-1]
Output: [0,0]

Constraints:

1 <= nums.length <= 105
-104 <= nums[i] <= 104

```
class Solution {
public List<Integer> countSmaller(int[] nums) {
        List<Integer> count = new ArrayList<>();
        int max = 20_001, offset = 10_000;
        int[] arr = new int[2*max];

        for(int j=nums.length-1; j>=0; j--) {
            int index = nums[j] + max + offset;
            count.add(getSum(arr, max, index));

            arr[index] += 1;
```

```
            for(index/=2 ;index>1; index/=2) {
                arr[index] = arr[index * 2] + arr[index * 2 + 1];
            }
        }

        Collections.reverse(count);
        return count;
    }

    private int getSum(int[] arr, int l, int r) {
        int sum = 0;
        while(l < r) {
            if(l%2 == 1)
                sum += arr[l++];
            if(r%2 == 1)
                sum += arr[--r];

            l >>= 1;
            r >>= 1;
        }

        return sum;
    }
}
```

----------Remove Duplicate Letters----------Given a string s, remove duplicate letters so that every letter appears once and only once. You must make sure your result is the smallest in lexicographical order among all possible results.

Example 1:

Input: s = "bcabc"
Output: "abc"
Example 2:

Input: s = "cbacdcbc"
Output: "acdb"

Constraints:

1 <= s.length <= 104
s consists of lowercase English letters.

// solution is pretty simple and straight forward
// first we count the frequency of each char in the string
// then we put each char of the string in the string res(the string that we'll be returning)
// if we find a char c (in text) that is smaller than the last char in our res string
// then we remove the char from the end of the res string
// then we will add the char c at the end of our res string
// this will ensure that the res will be lexicographically smallest subsequence of text

```
class Solution {
    public String removeDuplicateLetters(String text) {
        int[] cnt = new int[26];
        boolean[] added = new boolean[26];

        for(char c: text.toCharArray()) cnt[c - 'a']++;

        StringBuilder res = new StringBuilder();
        res.append('0');

        for(char c: text.toCharArray()) {
            cnt[c-'a']--;
            if(!added[c - 'a']) {
                char x = res.charAt(res.length() - 1);
                while(x > c && cnt[x - 'a'] > 0) {
                    res.deleteCharAt(res.length() - 1);
                    added[x - 'a'] = false;
                    x = res.charAt(res.length() - 1);
                }
                res.append(c);
                added[c-'a'] = true;
            }
        }

        return res.substring(1).toString();
    }
}
```

----------Maximum Product of Word Lengths
----------Given a string array words, return the maximum value of length(word[i]) * length(word[j]) where the two words do not share common letters. If no such two words exist, return 0.

Example 1:

Input: words = ["abcw","baz","foo","bar","xtfn","abcdef"]
Output: 16
Explanation: The two words can be "abcw", "xtfn".
Example 2:

Input: words = ["a","ab","abc","d","cd","bcd","abcd"]
Output: 4
Explanation: The two words can be "ab", "cd".
Example 3:

Input: words = ["a","aa","aaa","aaaa"]
Output: 0
Explanation: No such pair of words.

Constraints:

2 <= words.length <= 1000

1 <= words[i].length <= 1000
words[i] consists only of lowercase English letters.

```
// for every word:
//     check against all other words for chars
//     use bitset to set 1 for a char
//     if a & b > 0 -> overlapped
class Solution {
    public int maxProduct(String[] words) {

        int maximumProduct = 0;
        int[] bits = new int[words.length];

        for (int i = 0; i < words.length; i++) {
            for (char c : words[i].toCharArray ()) {
                bits[i] |= (1 << (c - 'a'));
            }
        }

        for (int i = 0; i < bits.length; i++) {
            for (int j = i + 1; j < bits.length; j++) {
                if ((bits[i] & bits[j]) == 0) {
                    maximumProduct = Math.max (maximumProduct, words[i].length () *
words[j].length ());
                }
            }
        }

        return maximumProduct;
    }
}
```

----------Bulb Switcher
----------There are n bulbs that are initially off. You first turn on all the bulbs, then you turn off every second bulb.

On the third round, you toggle every third bulb (turning on if it's off or turning off if it's on). For the ith round, you toggle every i bulb. For the nth round, you only toggle the last bulb.

Return the number of bulbs that are on after n rounds.

Example 1:

Input: n = 3
Output: 1
Explanation: At first, the three bulbs are [off, off, off].
After the first round, the three bulbs are [on, on, on].
After the second round, the three bulbs are [on, off, on].
After the third round, the three bulbs are [on, off, off].
So you should return 1 because there is only one bulb is on.
Example 2:

Input: n = 0
Output: 0
Example 3:

Input: n = 1
Output: 1

Constraints:

0 <= n <= 109

```
class Solution {
public int bulbSwitch(int n) {
int i=1;
while(i<=n/i)
i++;
return i-1;
}
}
```

----------Create Maximum Number----------

You are given two integer arrays nums1 and nums2 of lengths m and n respectively. nums1 and nums2 represent the digits of two numbers. You are also given an integer k.

Create the maximum number of length k <= m + n from digits of the two numbers. The relative order of the digits from the same array must be preserved.

Return an array of the k digits representing the answer.

Example 1:

Input: nums1 = [3,4,6,5], nums2 = [9,1,2,5,8,3], k = 5
Output: [9,8,6,5,3]
Example 2:

Input: nums1 = [6,7], nums2 = [6,0,4], k = 5
Output: [6,7,6,0,4]
Example 3:

Input: nums1 = [3,9], nums2 = [8,9], k = 3
Output: [9,8,9]

Constraints:

m == nums1.length
n == nums2.length
1 <= m, n <= 500
0 <= nums1[i], nums2[i] <= 9
1 <= k <= m + n

```
public class Solution {
  public int[] maxNumber(int[] nums1, int[] nums2, int k) {
    int n = nums1.length;
    int m = nums2.length;
    int[] ans = new int[k];
    for (int i = Math.max(0, k - m); i <= k && i <= n; ++i) {
        int[] candidate = merge(maxArray(nums1, i), maxArray(nums2, k - i), k);
        if (greater(candidate, 0, ans, 0)) ans = candidate;
    }
    return ans;
}
private int[] merge(int[] nums1, int[] nums2, int k) {
    int[] ans = new int[k];
    for (int i = 0, j = 0, r = 0; r < k; ++r)
        ans[r] = greater(nums1, i, nums2, j) ? nums1[i++] : nums2[j++];
    return ans;
}
public boolean greater(int[] nums1, int i, int[] nums2, int j) {
    while (i < nums1.length && j < nums2.length && nums1[i] == nums2[j]) {
        i++;
        j++;
    }
    return j == nums2.length || (i < nums1.length && nums1[i] > nums2[j]);
}
public int[] maxArray(int[] nums, int k) {
    int n = nums.length;
    int[] ans = new int[k];
    for (int i = 0, j = 0; i < n; ++i) {
        while (n - i + j > k && j > 0 && ans[j - 1] < nums[i]) j--;
        if (j < k) ans[j++] = nums[i];
    }
    return ans;
}


}
```

----------Power of Three----------Given an integer n, return true if it is a power of three. Otherwise, return false.

An integer n is a power of three, if there exists an integer x such that n == 3x.

Example 1:

Input: n = 27
Output: true
Example 2:

Input: n = 0
Output: false
Example 3:

Input: n = 9
Output: true
Example 4:

Input: n = 45
Output: false

Constraints:

-231 <= n <= 231 - 1

```
public class Solution {
    public boolean isPowerOfThree(int n) {
        return (Math.log10(n) / Math.log10(3)) % 1 == 0;
    }
}

public class Solution {
    public boolean isPowerOfThree(int n) {
        return n > 0 && 1162261467 % n == 0;
    }
}
```

----------Count of Range Sum
----------Given an integer array nums and two integers lower and upper, return the number of range sums that lie in [lower, upper] inclusive.

Range sum S(i, j) is defined as the sum of the elements in nums between indices i and j inclusive, where i <= j.

Example 1:

Input: nums = [-2,5,-1], lower = -2, upper = 2
Output: 3
Explanation: The three ranges are: [0,0], [2,2], and [0,2] and their respective sums are: -2, -1, 2.
Example 2:

Input: nums = [0], lower = 0, upper = 0
Output: 1

Constraints:

1 <= nums.length <= 105
-231 <= nums[i] <= 231 - 1
-105 <= lower <= upper <= 105
The answer is guaranteed to fit in a 32-bit integer.

```
// Solution for countRangeSum
class Solution {
    private int lower;
```

```
private int upper;
private int count = 0;
private long[] pfxSum;

public int countRangeSum(int[] nums, int lower, int upper) {
    int n = nums.length;
    this.lower = lower;
    this.upper = upper;

    this.pfxSum = new long[n+1];
    for (int i = 0; i < n; i++) {
        pfxSum[i+1] = pfxSum[i] + nums[i];
    }

    mergeSort(0, n);
    return count;
}

private void mergeSort(int low, int high){
    if (low >= high) return;
    int mid = low + (high - low) / 2;

    mergeSort(low, mid);
    mergeSort(mid+1, high);

    int i = mid + 1, j = mid + 1;
    for (int k = low; k <= mid; k++) {
        while (i <= high && pfxSum[i] - pfxSum[k] < lower) i++;
        while (j <= high && pfxSum[j] - pfxSum[k] <= upper) j++;

        count += j - i;
    }

    merge(low, mid, high);
}

private void merge(int low, int mid, int high) {
    long[] helper = new long[high - low + 1];
    for (int i = low; i <= high; i++) {
        helper[i - low] = pfxSum[i];
    }

    int i = low, j = mid+1;
    int idx = low;

    while (i <= mid && j <= high) {
        if (helper[i - low] < helper[j - low]) {
            pfxSum[idx++] = helper[i++ - low];
        } else {
            pfxSum[idx++] = helper[j++ - low];
        }
    }

    while (i <= mid) {
```

```
            pfxSum[idx++] = helper[i++ - low];
        }
    }
}
```

----------Odd Even Linked List
----------Given the head of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return the reordered list.

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

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

You must solve the problem in O(1) extra space complexity and O(n) time complexity.

Example 1:

Input: head = [1,2,3,4,5]
Output: [1,3,5,2,4]
Example 2:

Input: head = [2,1,3,5,6,4,7]
Output: [2,3,6,7,1,5,4]

Constraints:

n == number of nodes in the linked list
0 <= n <= 104
-106 <= Node.val <= 106

```
class Solution {
    public ListNode oddEvenList(ListNode head) {
        if (head == null || head.next == null)
        return head;
    ListNode odd_index = head, even_index = head.next, pre = even_index;
    while (even_index != null && even_index.next != null) {
        odd_index.next = even_index.next;
        odd_index = odd_index.next;
        even_index.next = odd_index.next;
        even_index = even_index.next;
    }
    odd_index.next = pre;
    return head;
    }
}
```

----------Longest Increasing Path in a Matrix
----------Given an m x n integers matrix, return the length of the longest increasing path in matrix.

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

Example 1:

Input: matrix = [[9,9,4],[6,6,8],[2,1,1]]
Output: 4
Explanation: The longest increasing path is [1, 2, 6, 9].
Example 2:

Input: matrix = [[3,4,5],[3,2,6],[2,2,1]]
Output: 4
Explanation: The longest increasing path is [3, 4, 5, 6]. Moving diagonally is not allowed.
Example 3:

Input: matrix = [[1]]
Output: 1

Constraints:

m == matrix.length
n == matrix[i].length
1 <= m, n <= 200
0 <= matrix[i][j] <= 231 - 1

```
class Solution {
    public int longestIncreasingPath(int[][] M) {
        int ylen = M.length, xlen = M[0].length, ans = 0;
        int[][] memo = new int[ylen][xlen];
        for (int i = 0; i < ylen; i++)
            for (int j = 0; j < xlen; j++)
                ans = Math.max(ans, dfs(i,j,M,memo));
        return ans;
    }
    public int dfs(int y, int x, int[][] M, int[][] memo) {
        if (memo[y][x] > 0) return memo[y][x];
        int val = M[y][x];
        memo[y][x] = 1 + Math.max(
            Math.max(y < M.length - 1 && M[y+1][x] < val ? dfs(y+1,x,M,memo) : 0,
                    y > 0 && M[y-1][x] < val ? dfs(y-1,x,M,memo) : 0),
            Math.max(x < M[0].length - 1 && M[y][x+1] < val ? dfs(y,x+1,M,memo) : 0,
                    x > 0 && M[y][x-1] < val ? dfs(y,x-1,M,memo) : 0));
        return memo[y][x];
    }
}
```

----------Patching Array

------------------------------Given a sorted integer array nums and an integer n, add/patch elements to the array such that any number in the range [1, n] inclusive can be formed by the sum of some elements in the array.

Return the minimum number of patches required.

Example 1:

Input: nums = [1,3], n = 6
Output: 1
Explanation:
Combinations of nums are [1], [3], [1,3], which form possible sums of: 1, 3, 4.
Now if we add/patch 2 to nums, the combinations are: [1], [2], [3], [1,3], [2,3], [1,2,3].
Possible sums are 1, 2, 3, 4, 5, 6, which now covers the range [1, 6].
So we only need 1 patch.
Example 2:

Input: nums = [1,5,10], n = 20
Output: 2
Explanation: The two patches can be [2, 4].
Example 3:

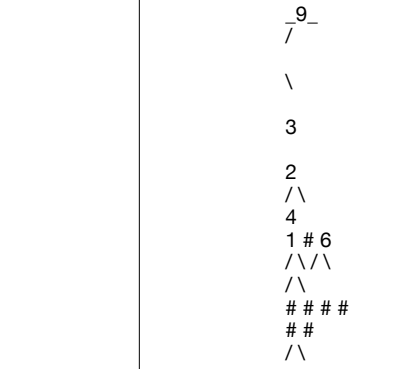
Input: nums = [1,2,2], n = 5
Output: 0

Constraints:

1 <= nums.length <= 1000
1 <= nums[i] <= 104
nums is sorted in ascending order.
1 <= n <= 231 - 1

```
public class Solution {
    public int minPatches(int[] nums, int n) {
        long sum = 0;
        int count = 0;
        for (int x : nums) {
            if (sum >= n) break;
            while (sum+1 < x && sum < n) {
                ++count;
                sum += sum+1;
            }
            sum += x;
        }
        while (sum < n) {
            sum += sum+1;
            ++count;
        }
        return count;
    }
}
```

----------Verify Preorder Serialization of a Binary Tree start here
----------One way to serialize a binary tree is to use pre-order traversal.

When we encounter a non-null node, we record the node's value. If it is a null node, we record using a sentinel value such as #.


For example, the above binary tree can be serialized to the string "9,3,4,#,#,1,#,#,2,#,6,#,#", where # represents a null node.
Given a string of comma separated values, verify whether it is a correct preorder traversal serialization of a binary tree. Find an algorithm without reconstructing the tree.
Each comma separated value in the string must be either an integer or a character '#' representing null pointer.
You may assume that the input format is always valid, for example it could never contain two consecutive commas such as "1,,3".
Example 1:
"9,3,4,#,#,1,#,#,2,#,6,#,#"
Return true
Example 2:
"1,#"
Return false
Example 3:
"9,#,#,1"
Return false
Credits:Special thanks to @dietpepsi for adding this problem and creating all test cases.
----------Reconstruct Itinerary
----------Given a list of airline tickets represented by pairs of departure and arrival airports [from, to], reconstruct the itinerary in order.
All of the tickets belong to a man who departs from JFK. Thus, the itinerary must begin with JFK.
Note:
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

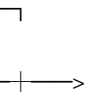
lexical order than ["JFK", "LGB"].
All airports are represented by three capital letters (IATA code).
You may assume all tickets form at least one valid itinerary.

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.
Credits:Special thanks to @dietpepsi for adding this problem and
creating all test cases.
----------Increasing Triplet Subsequence
----------Given an unsorted array return whether an increasing subsequence of length 3 exists or not in the array.
Formally the function should:
Return true if there exists i, j, k
such that arr[i] < arr[j] < arr[k] given 0 ≤ 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.

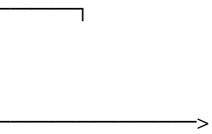
Credits:Special thanks to @DjangoUnchained for adding this problem
and creating all test cases.
----------Self Crossing
----------You are given an array x of n positive numbers. You start at
point (0,0) and moves x[0] metres to the north, then x[1] metres to
the west,
x[2] metres to the south,
x[3] metres to the east and so on. In other words, after each
move your direction changes
counter-clockwise.
Write a one-pass algorithm with O(1) extra space to determine,
if your path crosses itself, or not.

Example 1:
Given x = [2, 1, 1, 2],



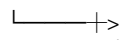
Return true (self crossing)

Example 2:
Given x = [1, 2, 3, 4],



Return false (not self crossing)

Example 3:
Given x = [1, 1, 1, 1],



Return true (self crossing)

Credits:Special thanks to @dietpepsi for adding this problem and
creating all test cases.
----------Palindrome Pairs
----------Given a list of unique words, find all pairs of distinct indices
(i, j) in the given list, so that the concatenation of the two
words, i.e. words[i] + words[j] is a palindrome.

Example 1:
Given words = ["bat", "tab", "cat"]
Return [[0, 1], [1, 0]]
The palindromes are ["battab", "tabbat"]
Example 2:
Given words = ["abcd", "dcba", "lls", "s", "sssll"]
Return [[0, 1], [1, 0], [3, 2], [2, 4]]
The palindromes are ["dcbaabcd", "abcddcba", "slls", "llssssll"]
Credits:Special thanks to @dietpepsi for adding this problem and
creating all test cases.
----------House Robber III
----------The thief has found himself a new place for his thievery again.
There is only one entrance to this area, called the "root." Besides
the root, each house has one and only one parent house. After a
tour, the smart thief realized that "all houses in this place forms
a binary tree". It will automatically contact the police if two
directly-linked houses were broken into on the same night.

Determine the maximum amount of money the thief can rob tonight
without alerting the police.
Example 1:
3

/ \
2

3
\

\
3

1

Maximum amount of money the thief can rob = 3 + 3 + 1 = 7.
Example 2:
3
/ \
4
/ \
1

5
\
3

1

Maximum amount of money the thief can rob = 4 + 5 = 9.
Credits:Special thanks to @dietpepsi for adding this problem and
creating all test cases.
----------Counting Bits
----------Given a non negative integer number num. For every numbers i in the
range 0 ≤ i ≤ num calculate the number of 1's in their binary
representation and return them as an array.
Example:
For num = 5 you should return [0,1,1,2,1,2].
Follow up:
It is very easy to come up with a solution with run time
O(n*sizeof(integer)). But can you do it in linear time O(n) /
possibly in a single pass?
Space complexity should be O(n).
Can you do it like a boss? Do it without using any builtin function
like __builtin_popcount in c++ or in any other language.

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

for adding this problem and

creating all test cases.

-------Flatten Nested List Iterator
----------Given a nested list of integers, implement an iterator to flatten
it.
Each element is either an integer, or a list -- whose elements may
also be integers or other lists.
Example 1:
Given the list [[1,1],2,[1,1]],
By calling next repeatedly until hasNext returns false, the order of
elements returned by next should be: [1,1,2,1,1].

Example 2:
Given the list [1,[4,[6]]],
By calling next repeatedly until hasNext returns false, the order of
elements returned by next should be: [1,4,6].
----------Power of Four
----------Given an integer (signed 32 bits), write a function to check whether
it is a power of 4.
Example:
Given num = 16, return true.
Given num = 5, return false.
Follow up: Could you solve it without loops/recursion?
Credits:Special thanks to @yukuairoy
creating all test cases.
----------Integer Break
---

for adding this problem and

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

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

Note: You may assume that n is not less than 2 and not larger than 58.

There is a simple O(n) solution to this problem.
You may check the breaking results of n ranging from 7 to 10 to discover the regularities.
Credits:Special thanks to @jianchao.li.fighter for adding this problem and creating all test cases.
---Reverse String
---Write a function that takes a string as input and returns the string reversed.
Example:
Given s = "hello", return "olleh".
---Reverse Vowels of a String
---Write a function that takes a string as input and reverse only the vowels of a string.
Example 1:
Given s = "hello", return "holle".

Example 2:
Given s = "leetcode", return "leotcede".

Note:
The vowels does not include the letter "y".

Top K Frequent Elements
----------Given a non-empty array of integers, return the k most frequent elements.
For example,
Given [1,1,1,2,2,3] and k = 2, return [1,2].
Note:
You may assume k is always valid, 1 ≤ k ≤ number of unique elements.
Your algorithm's time complexity must be better than O(n log n),
where n is the array's size.
----------Intersection of Two Arrays
----------Given two arrays, write a function to compute their intersection.
Example:
Given nums1 = [1, 2, 2, 1], nums2 = [2, 2], return [2].
Note:
Each element in the result must be unique.
The result can be in any order.
----------Intersection of Two Arrays II
----------Given two arrays, write a function to compute their intersection.
Example:
Given nums1 = [1, 2, 2, 1], nums2 = [2, 2], return [2, 2].
Note:
Each element in the result should appear as many times as it shows in both arrays.
The result can be in any order.

Follow up:
What if the given array is already sorted? How would you optimize your algorithm?
What if nums1's size is small compared to nums2's size? Which algorithm is better?
What if elements of nums2 are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?
----------Data Stream as Disjoint Intervals
----------Given a data stream input of non-negative integers a1, a2, ..., an, ..., summarize the numbers seen so far as a list of disjoint intervals.
For example, suppose the integers from the data stream are 1, 3, 7, 2, 6, ..., then the summary will be:
[1,
[1,
[1,
[1,
[1,

1]
1],
1],
3],
3],

[3,
[3,
[7,
[6,

3]
3], [7, 7]
7]
7]

Follow up:
What if there are lots of merges and the number of disjoint intervals are small compared to the data stream's size?
Credits:Special thanks to @yunhong for adding this problem and creating most of the test cases.
----------Russian Doll Envelopes
----------You have a number of envelopes with widths and heights given as a pair of integers (w, h). One envelope can fit into another if and only if both the width and height of one envelope is greater than the width and height of the other envelope.
What is the maximum number of envelopes can you Russian doll? (put one inside other)
Example:

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

Design Twitter

Design a simplified version of Twitter where users can post tweets, follow/unfollow another user and is able to see the 10 most recent tweets in the user's news feed. Your design should support the following methods:

postTweet(userId, tweetId): Compose a new tweet.
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.
follow(followerId, followeeId): Follower follows a followee.
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);
```


----------Max Sum of Rectangle No Larger Than K
----------Given a non-empty 2D matrix matrix and an integer k, find the max sum of a rectangle in the matrix such that its sum is no larger than k.
Example:
Given matrix = [
[1, 0, 1],
[0, -2, 3]
]
k = 2

The answer is 2. Because the sum of rectangle [[0, 1], [-2, 3]] is 2 and 2 is the max number no larger than k (k = 2).
Note:
The rectangle inside the matrix must have an area > 0.
What if the number of rows is much larger than the number of columns?

Credits:Special thanks to @fujiaozhu for adding this problem and creating all test cases.
----------Water and Jug Problem
----------You are given two jugs with capacities x and y litres. There is an infinite amount of water supply available.
You need to determine whether it is possible to measure exactly z litres using these two jugs.
If z liters of water is measurable, you must have z liters of water contained within one or both buckets by the end.
Operations allowed:
Fill any of the jugs completely with water.
Empty any of the jugs.

Pour water from one jug into another till the other jug is
completely full or the first jug itself is empty.

Example 1: (From the famous "Die Hard" example)
Input: x = 3, y = 5, z = 4
Output: True

Example 2:
Input: x = 2, y = 6, z = 5
Output: False

Credits:Special thanks to @vinod23 for adding this problem and
creating all test cases.
----------Valid Perfect
Square
----------Given a positive integer num, write a function which returns True if
num is a perfect square else False.
Note: Do not use any built-in library function such as sqrt.
Example 1:
Input: 16
Returns: True

Example 2:
Input: 14
Returns: False

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

Given a set of distinct positive integers, find the largest subset
such that every pair (Si, Sj) of elements in this subset satisfies:
Si % Sj = 0 or Sj % Si = 0.
If there are multiple solutions, return any subset is fine.
Example 1:
nums: [1,2,3]
Result: [1,2] (of course, [1,3] will also be ok)

Example 2:
nums: [1,2,4,8]
Result: [1,2,4,8]

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

-------Sum of Two Integers
----------Calculate the sum of two integers a and b, but you are not allowed
to use the operator + and -.
Example:
Given a = 1 and b = 2, return 3.
Credits:Special thanks to @fujiaozhu for adding this problem and
creating all test cases.
----------Super Pow
----------Your task is to calculate ab mod 1337 where a is a positive integer
and b is an extremely large positive integer given in the form of an
array.
Example1:
a = 2

b = [3]
Result: 8

Example2:
a = 2
b = [1,0]
Result: 1024

Credits:Special thanks to @Stomach_ache for adding this problem and
creating all test cases.
----------Find K Pairs with Smallest Sums
----------You are given two integer arrays nums1 and nums2 sorted in
ascending
order and an integer k.
Define a pair (u,v) which consists of one element from the first
array and one element from the second array.
Find the k pairs (u1,v1),(u2,v2) ...(uk,vk) with the smallest sums.
Example 1:
Given nums1 = [1,7,11], nums2 = [2,4,6],

k = 3

Return: [1,2],[1,4],[1,6]
The first 3 pairs are returned from the sequence:
[1,2],[1,4],[1,6],[7,2],[7,4],[11,2],[7,6],[11,4],[11,6]

Example 2:
Given nums1 = [1,1,2], nums2 = [1,2,3],

k = 2

Return: [1,1],[1,1]
The first 2 pairs are returned from the sequence:
[1,1],[1,1],[1,2],[2,1],[1,2],[2,2],[1,3],[1,3],[2,3]

Example 3:
Given nums1 = [1,2], nums2 = [3],

k = 3

Return: [1,3],[2,3]
All possible pairs are returned from the sequence:
[1,3],[2,3]

Credits:Special thanks to @elmirap and @StefanPochmann for adding
this problem and creating all test cases.
----------Guess Number Higher or Lower
----------We are playing the Guess Game. The game is as follows:
I pick a number from 1 to n. You have to guess which number I
picked.
Every time you guess wrong, I'll tell you whether the number is
higher or lower.
You call a pre-defined API guess(int num) which returns 3 possible
results (-1, 1, or 0):
-1 : My number is lower
1 : My number is higher
0 : Congrats! You got it!
Example:
n = 10, I pick 6.
Return 6.
----------Guess Number Higher or Lower II
----------We are playing the Guess Game. The game is as follows:
I pick a number from 1 to n. You have to guess which number I
picked.
Every time you guess wrong, I'll tell you whether the number I
picked is higher or lower.
However, when you guess a particular number x, and you guess wrong,
you pay $x. You win the game when you guess the number I picked.

Example:
n = 10, I pick 8.
First round: You guess 5, I tell you that it's higher. You pay $5.
Second round: You guess 7, I tell you that it's higher. You pay $7.
Third round: You guess 9, I tell you that it's lower. You pay $9.
Game over. 8 is the number I picked.
You end up paying $5 + $7 + $9 = $21.

Given a particular n ≥ 1, find out how much money you need to have to guarantee a win.
The best strategy to play the game is to minimize the maximum loss you could possibly face. Another strategy is to minimize the expected loss. Here, we are interested in the first scenario.
Take a small example (n = 3). What do you end up paying in the worst case?
Check out this article if you're still stuck.
The purely recursive implementation of minimax would be worthless for even a small n. You MUST use dynamic programming.
As a follow-up, how would you modify your code to solve the problem of minimizing the expected loss, instead of the worst-case loss?
Credits:Special thanks to @agave and @StefanPochmann for adding this problem and creating all test cases.
----------Wiggle Subsequence
----------A sequence of numbers is called a wiggle sequence if the differences between successive numbers strictly alternate between positive and negative. The first difference (if one exists) may be either positive or negative. A sequence with fewer than two elements is trivially a wiggle sequence.
For example, [1,7,4,9,2,5] is a wiggle sequence because the differences (6,-3,5,-7,3) are alternately positive and negative. In contrast, [1,4,7,2,5] and [1,7,4,5,5] are not wiggle sequences, the first because its first two differences are positive and the second because its last difference is zero.
Given a sequence of integers, return the length of the longest subsequence that is a wiggle sequence. A subsequence is obtained by deleting some number of elements (eventually, also zero) from the original sequence, leaving the remaining elements in their original

order.
Examples:
Input: [1,7,4,9,2,5]
Output: 6
The entire sequence is a wiggle sequence.
Input: [1,17,5,10,13,15,10,5,16,8]
Output: 7
There are several subsequences that achieve this length. One is
[1,17,10,13,10,16,8].
Input: [1,2,3,4,5,6,7,8,9]
Output: 2

Follow up:
Can you do it in O(n) time?
Credits:Special thanks to @agave and @StefanPochmann for adding this
problem and creating all test cases.
---Combination Sum IV
-----------------------------Given an integer array with all positive numbers and no duplicates,
find the number of possible combinations that add up to a positive
integer target.
Input: nums = [1,2,3], target = 4
Output: 7
Explanation:
The possible combination ways are:
(1, 1, 1, 1)
(1, 1, 2)
(1, 2, 1)
(1, 3)
(2, 1, 1)
(2, 2)
(3, 1)
Note that different sequences are counted as different combinations.
Example 2:

Input: nums = [9], target = 3
Output: 0

Constraints:

1 <= nums.length <= 200
1 <= nums[i] <= 1000
All the elements of nums are unique.
1 <= target <= 1000

Note that different sequences are counted as different combinations.
Therefore the output is 7.

Follow up:
What if negative numbers are allowed in the given array?

How does it change the problem?
What limitation we need to add to the question to allow negative numbers?

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

    private int combinationSumDp(int[] nums, int target, int[] dp) {
                    //// Once target reaches to 0 -> contributes to a way we can create a combination sum
        if(target == 0) return 1;
                    //// dp has updated value, if it is not equal to -1
        if(dp[target] != -1) return dp[target];

        int ways = 0;
        /// Try all elements one by one present in nums arrays, if we can reduce our target using them
                    /// Index starts from `0`, since we want to count instances like `(2, 1)` and `(1, 2)` separately
        for(int i=0;i<nums.length;i++) {
            if(nums[i] <= target) { //// Try this nums[i] only if target remains greater than or equal to 0 after using this nums[i]
                ways += combinationSumDp(nums, target - nums[i], dp);
            }
        }

                    /// Ultimately update the DP
        dp[target] = ways;
        return ways;
    }
}
```

----------Kth Smallest Element in a Sorted Matrix----------Given a n x n matrix where each of the rows and columns are sorted in ascending order, find the kth smallest element in the matrix.
Note that it is the kth smallest element in the sorted order, not the kth distinct element.
Example:
matrix = [
[1, 5, 9],
[10, 11, 13],
[12, 13, 15]
],
k = 8,
return 13.

Note:
You may assume k is always valid, 1 ≤ k ≤ n2.

----------Insert Delete GetRandom O(1)
----------Design a data structure that supports all following operations in average O(1) time.

insert(val): Inserts an item val to the set if not already present.
remove(val): Removes an item val from the set if present.
getRandom: Returns a random element from current set of elements.
Each element must have the same probability of being returned.

Example:
// Init an empty set.
RandomizedSet randomSet = new RandomizedSet();

// Inserts 1 to the set. Returns true as 1 was inserted
successfully.
randomSet.insert(1);
// Returns false as 2 does not exist in the set.
randomSet.remove(2);
// Inserts 2 to the set, returns true. Set now contains [1,2].
randomSet.insert(2);
// getRandom should return either 1 or 2 randomly.
randomSet.getRandom();
// Removes 1 from the set, returns true. Set now contains [2].
randomSet.remove(1);
// 2 was already in the set, so return false.
randomSet.insert(2);
// Since 2 is the only number in the set, getRandom always return 2.
randomSet.getRandom();
----------Insert Delete GetRandom O(1) - Duplicates
allowed
----------Design a data structure that supports all following operations in
average O(1) time.
Note: Duplicate elements are allowed.
insert(val): Inserts an item val to the collection.
remove(val): Removes an item val from the collection if present.
getRandom: Returns a random element from current collection of
elements. The probability of each element being returned is linearly
related to the number of same value the collection contains.

Example:
// Init an empty collection.
RandomizedCollection collection = new RandomizedCollection();
// Inserts 1 to the collection. Returns true as the collection did
not contain 1.
collection.insert(1);
// Inserts another 1 to the collection. Returns false as the
collection contained 1. Collection now contains [1,1].
collection.insert(1);

// Inserts 2 to the collection, returns true. Collection now
contains [1,1,2].
collection.insert(2);
// getRandom should return 1 with the probability 2/3, and returns 2
with the probability 1/3.
collection.getRandom();
// Removes 1 from the collection, returns true. Collection now
contains [1,2].
collection.remove(1);
// getRandom should return 1 and 2 both equally likely.
collection.getRandom();
----------Linked List Random Node
----------Given a singly linked list, return a random node's value from the
linked list. Each node must have the same probability of being
chosen.
Follow up:
What if the linked list is extremely large and its length is unknown
to you? Could you solve this efficiently without using extra space?
Example:
// Init a singly linked list [1,2,3].
ListNode head = new ListNode(1);
head.next = new ListNode(2);
head.next.next = new ListNode(3);
Solution solution = new Solution(head);
// getRandom() should return either 1, 2, or 3 randomly. Each
element should have equal probability of returning.
solution.getRandom();
----------Ransom Note
----------Given an arbitrary ransom note string and another string containing
letters from all the magazines, write a function that will return
true if the ransom
note can be constructed from the magazines ; otherwise, it will
return false.

Each letter in the magazine string can only be used once in your
ransom note.
Note:
You may assume that both strings contain only lowercase letters.

canConstruct("a", "b") -> false
canConstruct("aa", "ab") -> false
canConstruct("aa", "aab") -> true
----------Shuffle an Array
----------Shuffle a set of numbers without duplicates.
Example:
// Init an array with set 1, 2, and 3.
int[] nums = {1,2,3};
Solution solution = new Solution(nums);
// Shuffle the array [1,2,3] and return its result. Any permutation
of [1,2,3] must equally likely to be returned.
solution.shuffle();
// Resets the array back to its original configuration [1,2,3].
solution.reset();
// Returns the random shuffling of array [1,2,3].
solution.shuffle();
----------Mini Parser
----------Given a nested list of integers represented as a string, implement a
parser to deserialize it.
Each element is either an integer, or a list -- whose elements may
also be integers or other lists.
Note:
You may assume that the string is well-formed:
String is non-empty.
String does not contain white spaces.
String contains only digits 0-9, [, - ,,].

Example 1:
Given s = "324",
You should return a NestedInteger object which contains a single integer 324.

Example 2:
Given s = "[123,[456,[789]]]",
Return a NestedInteger object containing a nested list with 2 elements:
1. An integer containing value 123.
2. A nested list containing two elements:
i. An integer containing value 456.
ii. A nested list with one element:
a. An integer containing value 789.
--Lexicographical Numbers
----------------------------Given an integer n, return 1 - n in lexicographical order.

For example, given 13, return: [1,10,11,12,13,2,3,4,5,6,7,8,9].

Please optimize your algorithm to use less time and space. The input size may be as large as 5,000,000.
--First Unique Character in a String
----------------------------Given a string, find the first non-repeating character in it and return it's index. If it doesn't exist, return -1.
Examples:
s = "leetcode"
return 0.

s = "loveleetcode",
return 2.

Note: You may assume the string contain only lowercase letters.
----------Longest Absolute File Path
----------Suppose we abstract our file system by a string in the following manner:
The string "dir\n\tsubdir1\n\tsubdir2\n\t\tfile.ext" represents:
dir
subdir1
subdir2
file.ext
The directory dir contains an empty sub-directory subdir1 and a subdirectory subdir2 containing a file file.ext.
The string
"dir\n\tsubdir1\n\t\tfile1.ext\n\t\tsubsubdir1\n\tsubdir2\n\t\tsubsu
bdir2\n\t\t\tfile2.ext" represents:
dir
subdir1
file1.ext
subsubdir1
subdir2
subsubdir2
file2.ext
The directory dir contains two sub-directories subdir1 and subdir2.
subdir1 contains a file file1.ext and an empty second-level subdirectory subsubdir1. subdir2 contains a second-level sub-directory
subsubdir2 containing a file file2.ext.
We are interested in finding the longest (number of characters) absolute path to a file within our file system. For example, in the second example above, the longest absolute path is "dir/subdir2/subsubdir2/file2.ext", and its length is 32 (not including the double quotes).
Given a string representing the file system in the above format, return the length of the longest absolute path to file in the abstracted file system. If there is no file in the system, return 0.

Note:
The name of a file contains at least a . and an extension.
The name of a directory or sub-directory will not contain a ..

Time complexity required: O(n) where n is the size of the input
string.
Notice that a/aa/aaa/file1.txt is not the longest file path, if
there is another path aaaaaaaaaaaaaaaaaaaaa/sth.png.
----------Find the Difference
----------Given two strings s and t which consist of only lowercase letters.
String t is generated by random shuffling string s and then add one
more letter at a random position.
Find the letter that was added in t.
Example:
Input:
s = "abcd"
t = "abcde"
Output:
e
Explanation:
'e' is the letter that was added.
----------Elimination Game
----------There is a list of sorted integers from 1 to n. Starting from left
to right, remove the first number and every other number afterward
until you reach the end of the list.
Repeat the previous step again, but this time from right to left,
remove the right most number and every other number from the
remaining numbers.
We keep repeating the steps again, alternating left to right and
right to left, until a single number remains.
Find the last number that remains starting with a list of length n.

Example:
Input:
n = 9,
1 2 3 4 5 6 7 8 9
2 4 6 8
2 6
6
Output:
6
----------Perfect Rectangle
----------Given N axis-aligned rectangles where N > 0, determine if they all together form an exact cover of a rectangular region.

Each rectangle is represented as a bottom-left point and a top-right point. For example, a unit square is represented as [1,1,2,2]. (coordinate of bottom-left point is (1, 1) and top-right point is (2, 2)).

Example 1:
rectangles = [
[1,1,3,3],
[3,1,4,2],
[3,2,4,4],
[1,3,2,4],
[2,3,3,4]
]
Return true. All 5 rectangles together form an exact cover of a rectangular region.

Example 2:
rectangles = [
[1,1,2,3],
[1,3,2,4],

[3,1,4,2],
[3,2,4,4]
]
Return false. Because there is a gap between the two rectangular
regions.

Example 3:
rectangles = [
[1,1,3,3],
[3,1,4,2],
[1,3,2,4],
[3,2,4,4]
]
Return false. Because there is a gap in the top center.

Example 4:
rectangles = [
[1,1,3,3],
[3,1,4,2],
[1,3,2,4],
[2,2,4,4]
]
Return false. Because two of the rectangles overlap with each other.

----------Is Subsequence
----------Given a string s and a string t, check if s is subsequence of t.

You may assume that there is only lower case English letters in both
s and t. t is potentially a very long (length ~= 500,000) string,
and s is a short string (<=100).

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ace" is a subsequence of "abcde" while "aec" is not).
Example 1:
s = "abc", t = "ahbgdc"
Return true.
Example 2:
s = "axc", t = "ahbgdc"
Return false.
Follow up:
If there are lots of incoming S, say S1, S2, ... , Sk where k >= 1B, and you want to check one by one to see if T has its subsequence. In this scenario, how would you change your code?
Credits:Special thanks to @pbrother for adding this problem and creating all test cases.
----------UTF-8 Validation
----------A character in UTF8 can be from 1 to 4 bytes long, subjected to the following rules:
For 1-byte character, the first bit is a 0, followed by its unicode code.
For n-bytes character, the first n-bits are all one's, the n+1 bit is 0, followed by n-1 bytes with most significant 2 bits being 10.
This is how the UTF-8 encoding would work:
Char. number range |
UTF-8 octet sequence
(hexadecimal)
|
(binary)
----------+----------0000 0000-0000 007F | 0xxxxxxx
0000 0080-0000 07FF | 110xxxxx 10xxxxxx
0000 0800-0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx
0001 0000-0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Given an array of integers representing the data, return whether it is a valid utf-8 encoding.
Note:
The input is an array of integers. Only the least significant 8 bits of each integer is used to store the data. This means each integer represents only 1 byte of data.

Example 1:
data = [197, 130, 1], which represents the octet sequence: 11000101 10000010 00000001.
Return true.
It is a valid utf-8 encoding for a 2-bytes character followed by a 1-byte character.

Example 2:
data = [235, 140, 4], which represented the octet sequence: 11101011 10001100 00000100.
Return false.
The first 3 bits are all one's and the 4th bit is 0 means it is a 3bytes character.
The next byte is a continuation byte which starts with 10 and that's correct.
But the second continuation byte does not start with 10, so it is invalid.
----------Decode String
----------Given an encoded string, return it's decoded string.
The encoding rule is: k[encoded_string], where the encoded_string inside the square brackets is being repeated exactly k times. Note that k is guaranteed to be a positive integer.
You may assume that the input string is always valid; No extra white spaces, square brackets are well-formed, etc.

Furthermore, you may assume that the original data does not contain
any digits and that digits are only for those repeat numbers, k. For
example, there won't be input like 3a or 2[4].
Examples:
s = "3[a]2[bc]", return "aaabcbc".
s = "3[a2[c]]", return "accaccacc".
s = "2[abc]3[cd]ef", return "abcabccdcdcdef".
----------Longest Substring with At Least K
Repeating Characters
----------Find the length of the longest substring T of a given string
(consists of lowercase letters only) such that every character in T
appears no less than k times.
Example 1:
Input:
s = "aaabb", k = 3
Output:
3
The longest substring is "aaa", as 'a' is repeated 3 times.

Example 2:
Input:
s = "ababbc", k = 2
Output:
5
The longest substring is "ababb", as 'a' is repeated 2 times and 'b'
is repeated 3 times.
----------Rotate Function
----------Given an array of integers A and let n to be its length.

Assume Bk to be an array obtained by rotating the array A k positions clock-wise, we define a "rotation function" F on A as follow:

F(k) = 0 * Bk[0] + 1 * Bk[1] + ... + (n-1) * Bk[n-1].
Calculate the maximum value of F(0), F(1), ..., F(n-1).
Note:
n is guaranteed to be less than 105.
Example:
A = [4, 3, 2, 6]
F(0)
F(1)
F(2)
F(3)

=
=
=
=

(0
(0
(0
(0

*
*
*
*

4)
6)
2)
3)

+
+
+
+

(1
(1
(1
(1

*
*
*
*

3)
4)
6)
2)

+
+
+
+

(2
(2
(2
(2

*
*
*
*

2)
3)
4)
6)

+
+
+
+

(3
(3
(3
(3

*
*
*
*

6)
2)
3)
4)

=
=
=
=

0
0
0
0

+
+
+

+

3
4
6
2

+
+
+
+

4 + 18 = 25
6 + 6 = 16
8 + 9 = 23
12 + 12 = 26

So the maximum value of F(0), F(1), F(2), F(3) is F(3) = 26.
----------Integer Replacement
----------Given a positive integer n and you can do operations as follow:

If n is even, replace n with n/2.
If n is odd, you can replace n with either n + 1 or n - 1.

What is the minimum number of replacements needed for n to become 1?

Example 1:
Input:
8

Output:
3
Explanation:
8 -> 4 -> 2 -> 1

Example 2:
Input:
7
Output:
4
Explanation:
7 -> 8 -> 4 -> 2 -> 1
or
7 -> 6 -> 3 -> 2 -> 1
----------Random Pick Index
----------Given an array of integers with possible duplicates, randomly output the index of a given target number. You can assume that the given target number must exist in the array.

Note:
The array size can be very large. Solution that uses too much extra space will not pass the judge.
Example:
int[] nums = new int[] {1,2,3,3,3};
Solution solution = new Solution(nums);
// pick(3) should return either index 2, 3, or 4 randomly. Each index should have equal probability of returning.
solution.pick(3);
// pick(1) should return 0. Since in the array only nums[0] is equal to 1.
solution.pick(1);

Evaluate Division
----------Equations are given in the format A / B = k, where A and B are variables represented as strings, and k is a real number (floating point number). Given some queries, return the answers. If the answer does not exist, return -1.0.
Example:
Given a / b = 2.0, b / c = 3.0. queries are: a / c = ?, b / a = ?, a / e = ?, a / a = ?, x / x = ? . return [6.0, 0.5, -1.0, 1.0, -1.0].
The input is: vector<pair<string, string>> equations, vector<double>& values, vector<pair<string, string>> queries , where equations.size() == values.size(), and the values are positive. This represents the equations. Return vector<double>.
According to the example above:
equations = [["a", "b"], ["b", "c"]],
values = [2.0, 3.0],
queries = [["a", "c"], ["b", "a"], ["a", "e"], ["a", "a"], ["x", "x"]].

The input is always valid. You may assume that evaluating the queries will result in no division by zero and there is no contradiction.
----------Nth Digit
----------Find the nth digit of the infinite integer sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...
Note:
n is positive and will fit within the range of a 32-bit signed integer (n < 231).
Example 1:
Input:
3
Output:
3

Example 2:
Input:
11
Output:
0
Explanation:
The 11th digit of the sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, ... is a 0, which is part of the number 10.
----------Binary Watch
----------A binary watch has 4 LEDs on the top which represent the hours
(0-11), and the 6 LEDs on the bottom represent the minutes (0-59).
Each LED represents a zero or one, with the least significant bit on
the right.
For example, the above binary watch reads "3:25".
Given a non-negative integer n which represents the number of LEDs
that are currently on, return all possible times the watch could
represent.
Example:
Input: n = 1Return: ["1:00", "2:00", "4:00", "8:00", "0:01", "0:02",
"0:04", "0:08", "0:16", "0:32"]
Note:
The order of output does not matter.
The hour must not contain a leading zero, for example "01:00" is not
valid, it should be "1:00".
The minute must be consist of two digits and may contain a leading
zero, for example "10:2" is not valid, it should be "10:02".
----------Remove K Digits
----------Given a non-negative integer num represented as a string, remove k
digits from the number so that the new number is the smallest
possible.
Note:

The length of num is less than 10002 and will be ≥ k.
The given num does not contain any leading zero.

Example 1:
Input: num = "1432219", k = 3
Output: "1219"
Explanation: Remove the three digits 4, 3, and 2 to form the new number 1219 which is the smallest.

Example 2:
Input: num = "10200", k = 1
Output: "200"
Explanation: Remove the leading 1 and the number is 200. Note that the output must not contain leading zeroes.

Example 3:
Input: num = "10", k = 2
Output: "0"
Explanation: Remove all the digits from the number and it is left with nothing which is 0.
----------Frog Jump
----------A frog is crossing a river. The river is divided into x units and at each unit there may or may not exist a stone. The frog can jump on a stone, but it must not jump into the water.
Given a list of stones' positions (in units) in sorted ascending order, determine if the frog is able to cross the river by landing on the last stone. Initially, the frog is on the first stone and assume the first jump must be 1 unit.
If the frog's last jump was k units, then its next jump must be either k - 1, k, or k + 1 units. Note that the frog can only jump in the forward direction.
Note:
The number of stones is ≥ 2 and is < 1,100.
Each stone's position will be a non-negative integer < 231.

The first stone's position is always 0.

Example 1:
[0,1,3,5,6,8,12,17]
There are a total of 8 stones.
The first stone at the 0th unit, second stone at the 1st unit,
third stone at the 3rd unit, and so on...
The last stone at the 17th unit.
Return true. The frog can jump to the last stone by jumping
1 unit to the 2nd stone, then 2 units to the 3rd stone, then
2 units to the 4th stone, then 3 units to the 6th stone,
4 units to the 7th stone, and 5 units to the 8th stone.

Example 2:
[0,1,2,3,4,8,9,11]
Return false. There is no way to jump to the last stone as
the gap between the 5th and 6th stone is too large.
----------Sum of Left Leaves
----------Find the sum of all left leaves in a given binary tree.
Example:
3
9

/ \
20
/ \
15
7

There are two left leaves in the binary tree, with values 9 and 15
respectively. Return 24.
----------Convert a Number to Hexadecimal
----------Given an integer, write an algorithm to convert it to hexadecimal.
For negative integer, two’s complement method is used.

Note:
All letters in hexadecimal (a-f) must be in lowercase.
The hexadecimal string must not contain extra leading 0s. If the number is zero, it is represented by a single zero character '0'; otherwise, the first character in the hexadecimal string will not be the zero character.
The given number is guaranteed to fit within the range of a 32-bit signed integer.
You must not use any method provided by the library which converts/formats the number to hex directly.

Example 1:
Input:
26
Output:
"1a"

Example 2:
Input:
-1
Output:
"ffffffff"
----------Queue Reconstruction by Height
----------Suppose you have a random list of people standing in a queue. Each person is described by a pair of integers (h, k), where h is the height of the person and k is the number of people in front of this person who have a height greater than or equal to h. Write an algorithm to reconstruct the queue.
Note:
The number of people is less than 1,100.
Example
Input:
[[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]

Output:
[[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]
----------Trapping Rain Water II
----------Given an m x n matrix of positive integers representing the height of each unit cell in a 2D elevation map, compute the volume of water it is able to trap after raining.
Note:
Both m and n are less than 110. The height of each unit cell is greater than 0 and is less than 20,000.
Example:
Given the following 3x6 height map:
[
[1,4,3,1,3,2],
[3,2,1,3,2,4],
[2,3,3,2,3,1]
]
Return 4.

The above image represents the elevation map [[1,4,3,1,3,2], [3,2,1,3,2,4],[2,3,3,2,3,1]] before the rain.

After the rain, water are trapped between the blocks. The total volume of water trapped is 4.
----------Longest Palindrome
----------Given a string which consists of lowercase or uppercase letters, find the length of the longest palindromes that can be built with those letters.
This is case sensitive, for example "Aa" is not considered a palindrome here.

Note:
Assume the length of given string will not exceed 1,010.
Example:
Input:
"abccccdd"
Output:
7
Explanation:
One longest palindrome that can be built is "dccaccd", whose length
is 7.
----------Split Array Largest Sum
----------Given an array which consists of non-negative integers and an
integer m, you can split the array into m non-empty continuous
subarrays. Write an algorithm to minimize the largest sum among
these m subarrays.
Note:
If n is the length of array, assume the following constraints are
satisfied:
1 ≤ n ≤ 1000
1 ≤ m ≤ min(50, n)

Examples:
Input:
nums = [7,2,5,10,8]
m = 2
Output:
18
Explanation:
There are four ways to split nums into two subarrays.
The best way is to split it into [7,2,5] and [10,8],
where the largest sum among the two subarrays is only 18.

Fizz Buzz
----------Write a program that outputs the string representation of numbers from 1 to n.
But for multiples of three it should output “Fizz” instead of the number and for the multiples of five output “Buzz”. For numbers which are multiples of both three and five output “FizzBuzz”.
Example:
n = 15,
Return:
[
"1",
"2",
"Fizz",
"4",
"Buzz",
"Fizz",
"7",
"8",
"Fizz",
"Buzz",
"11",
"Fizz",
"13",
"14",
"FizzBuzz"
]
----------Arithmetic Slices
----------A sequence of number is called arithmetic if it consists of at least three elements and if the difference between any two consecutive elements is the same.
For example, these are arithmetic sequence:
1, 3, 5, 7, 9
7, 7, 7, 7
3, -1, -5, -9
The following sequence is not arithmetic. 1, 1, 2, 5, 7
A zero-indexed array A consisting of N numbers is given. A slice of that array is any pair of integers (P, Q) such that 0 <= P < Q < N.
A slice (P, Q) of array A is called arithmetic if the sequence:
A[P], A[p + 1], ..., A[Q - 1], A[Q] is arithmetic. In

particular, this means that P + 1 < Q.
The function should return the number of arithmetic slices in the
array A.
Example:
A = [1, 2, 3, 4]
return: 3, for 3 arithmetic slices in A: [1, 2, 3], [2, 3, 4] and
[1, 2, 3, 4] itself.
----------Third Maximum Number
----------Given a non-empty array of integers, return the third maximum
number
in this array. If it does not exist, return the maximum number. The
time complexity must be in O(n).
Example 1:
Input: [3, 2, 1]
Output: 1
Explanation: The third maximum is 1.

Example 2:
Input: [1, 2]
Output: 2
Explanation: The third maximum does not exist, so the maximum (2) is
returned instead.

Example 3:
Input: [2, 2, 3, 1]
Output: 1
Explanation: Note that the third maximum here means the third
maximum distinct number.
Both numbers with value 2 are both considered as second maximum.

----------Add Strings
----------Given two non-negative integers num1 and num2 represented as string,
return the sum of num1 and num2.
Note:
The length of both num1 and num2 is < 5100.
Both num1 and num2 contains only digits 0-9.
Both num1 and num2 does not contain any leading zero.
You must not use any built-in BigInteger library or convert the
inputs to integer directly.
----------Partition Equal Subset Sum
----------Given a non-empty array containing only positive integers, find if the array can be partitioned into two subsets such that the sum of
elements in both subsets is equal.
Note:
Each of the array element will not exceed 100.
The array size will not exceed 200.

Example 1:
Input: [1, 5, 11, 5]
Output: true
Explanation: The array can be partitioned as [1, 5, 5] and [11].

Example 2:
Input: [1, 2, 3, 5]
Output: false
Explanation: The array cannot be partitioned into equal sum subsets.
----------Pacific Atlantic Water Flow

Given an m x n matrix of non-negative integers representing the
height of each unit cell in a continent, the "Pacific ocean" touches
the left and top edges of the matrix and the "Atlantic ocean"
touches the right and bottom edges.
Water can only flow in four directions (up, down, left, or right)
from a cell to another one with height equal or lower.
Find the list of grid coordinates where water can flow to both the
Pacific and Atlantic ocean.
Note:
The order of returned grid coordinates does not matter.
Both m and n are less than 150.
Example:
Given the following 5x5 matrix:
Pacific ~
~
~
~
~
~ 1
2
2
3 (5) *
~ 3
2
3 (4) (4) *
~ 2
4 (5) 3
1 *
~ (6) (7) 1
4
5 *
~ (5) 1
1
2
4 *
*
*
*
*
* Atlantic
Return:
[[0, 4], [1, 3], [1, 4], [2, 2], [3, 0], [3, 1], [4, 0]] (positions
with parentheses in above matrix).
----------Battleships in a
Board
----------Given an 2D board, count how many battleships are in it. The
battleships are represented with 'X's, empty slots are represented
with '.'s. You may assume the following rules:
You receive a valid board, made of only battleships or empty slots.
Battleships can only be placed horizontally or vertically. In other
words, they can only be made of the shape 1xN (1 row, N columns) or
Nx1 (N rows, 1 column), where N can be of any size.
At least one horizontal or vertical cell separates between two
battleships - there are no adjacent battleships.

Example:

X..X
...X
...X
In the above board there are 2 battleships.
Invalid Example:
...X
XXXX
...X
This is an invalid board that you will not receive - as battleships
will always have a cell separating between them.
Follow up:Could you do it in one-pass, using only O(1) extra memory
and without modifying the value of the board?
----------Strong Password Checker
----------A password is considered strong if below conditions are all met:
It has at least 6 characters and at most 20 characters.
It must contain at least one lowercase letter, at least one
uppercase letter, and at least one digit.
It must NOT contain three repeating characters in a row
("...aaa..." is weak, but "...aa...a..." is strong, assuming other
conditions are met).
Write a function strongPasswordChecker(s), that takes a string s as
input, and return the MINIMUM change required to make s a strong
password. If s is already strong, return 0.
Insertion, deletion or replace of any one character are all
considered as one change.
----------Maximum XOR of Two Numbers in an Array
----------Given a non-empty array of numbers, a0, a1, a2, … , an-1, where 0
≤ ai < 231.
Find the maximum result of ai XOR aj, where 0 ≤ i, j < n.
Could you do this in O(n) runtime?
Example:
Input: [3, 10, 5, 25, 2, 8]
Output: 28

Explanation: The maximum result is 5 ^ 25 = 28.
----------Reconstruct Original Digits from English
----------Given a non-empty string containing an out-of-order English representation of digits 0-9, output the digits in ascending order.
Note:
Input contains only lowercase English letters.
Input is guaranteed to be valid and can be transformed to its original digits. That means invalid inputs such as "abc" or "zerone" are not permitted.
Input length is less than 50,000.

Example 1:
Input: "owoztneoer"
Output: "012"

Example 2:
Input: "fviefuro"
Output: "45"
----------Longest Repeating Character Replacement start here
----------Given a string that consists of only uppercase English letters, you can replace any letter in the string with another letter at most k times. Find the length of a longest substring containing all repeating letters you can get after performing the above operations.
Note:
Both the string's length and k will not exceed 104.

Example 1:
Input:
s = "ABAB", k = 2

Output:
4
Explanation:
Replace the two 'A's with two 'B's or vice versa.

Example 2:
Input:
s = "AABABBA", k = 1
Output:
4
Explanation:
Replace the one 'A' in the middle with 'B' and form "AABBBBA".
The substring "BBBB" has the longest repeating letters, which is 4.
----------All O`one Data Structure
----------Design a data structure to store the strings' count with the ability to return the strings with minimum and maximum counts.

Implement the AllOne class:

AllOne() Initializes the object of the data structure.
inc(String key) Increments the count of the string key by 1. If key does not exist in the data structure, insert it with count 1.
dec(String key) Decrements the count of the string key by 1. If the count of key is 0 after the decrement, remove it from the data structure. It is guaranteed that key exists in the data structure before the decrement.
getMaxKey() Returns one of the keys with the maximal count. If no element exists, return an empty string "".
getMinKey() Returns one of the keys with the minimum count. If no element exists, return an empty string "".

Example 1:

Input
["AllOne", "inc", "inc", "getMaxKey", "getMinKey", "inc", "getMaxKey", "getMinKey"]
[[], ["hello"], ["hello"], [], [], ["leet"], [], []]
Output
[null, null, null, "hello", "hello", null, "hello", "leet"]

Explanation
AllOne allOne = new AllOne();
allOne.inc("hello");
allOne.inc("hello");
allOne.getMaxKey(); // return "hello"
allOne.getMinKey(); // return "hello"
allOne.inc("leet");
allOne.getMaxKey(); // return "hello"
allOne.getMinKey(); // return "leet"

Constraints:

1 <= key.length <= 10

key consists of lowercase English letters.
It is guaranteed that for each call to dec, key is existing in the data structure.
At most 5 * 104 calls will be made to inc, dec, getMaxKey, and getMinKey.

```
class AllOne {
    class DLNode{
        DLNode prev;
        DLNode next;
        Set<String> keys;
        int count;
        DLNode(){}
        DLNode(int val){
            this.count = val;
            this.keys = new HashSet();
        }
        DLNode(int val, String key){
            this.count = val;
            this.keys = new HashSet();
            keys.add(key);
        }
    }

    Map<String, DLNode> keyNodeMap;
    DLNode head;
    DLNode tail;
    public AllOne() {
        this.keyNodeMap = new HashMap();
        this.head = new DLNode();
        this.tail = new DLNode();
        head.next = tail;
        tail.prev = head;
    }

    public void removeNode(DLNode node){
        node.prev.next = node.next;
        node.next.prev = node.prev;
    }

    public DLNode removeKey(String key){
        DLNode oldNode = keyNodeMap.get(key);
        oldNode.keys.remove(key);
        if(oldNode.keys.isEmpty()) removeNode(oldNode);
        keyNodeMap.remove(key);
        return oldNode;
    }

    public void addKey(String key, int count, DLNode prev, DLNode next){
        if(prev.count == count){
            prev.keys.add(key);
            keyNodeMap.put(key, prev);
        }else if(next.count == count){
            next.keys.add(key);
            keyNodeMap.put(key, next);
        }else{
```

```
            DLNode newNode = new DLNode(count, key);
            newNode.next = next;
            newNode.prev = prev;
            prev.next = newNode;
            next.prev = newNode;
            keyNodeMap.put(key, newNode);
        }
    }

    public void inc(String key) {
        if(keyNodeMap.containsKey(key)){
            DLNode oldNode = removeKey(key);
            if(oldNode.keys.isEmpty()){
                addKey(key, oldNode.count + 1, oldNode.prev, oldNode.next);
            }else{
                addKey(key, oldNode.count + 1, oldNode.prev, oldNode);
            }
        }else{
            addKey(key, 1, tail.prev, tail);
        }
    }

    public void dec(String key) {
        DLNode oldNode = removeKey(key);
        if(oldNode.count != 1){
            if(oldNode.keys.isEmpty()){
                addKey(key, oldNode.count - 1, oldNode.prev, oldNode.next);
            }else{
                addKey(key, oldNode.count - 1, oldNode, oldNode.next);
            }
        }
    }

    public String getMaxKey() {
        return head.next == tail? "" : head.next.keys.iterator().next();
    }

    public String getMinKey() {
        return tail.prev == head? "" : tail.prev.keys.iterator().next();
    }
}

/**
 * Your AllOne object will be instantiated and called as such:
 * AllOne obj = new AllOne();
 * obj.inc(key);
 * obj.dec(key);
 * String param_3 = obj.getMaxKey();
 * String param_4 = obj.getMinKey();
 */
```

----------Number of Segments in a String

----------You are given a string s, return the number of segments in the string.

A segment is defined to be a contiguous sequence of non-space characters.

Example 1:

Input: s = "Hello, my name is John"
Output: 5
Explanation: The five segments are ["Hello,", "my", "name", "is", "John"]
Example 2:

Input: s = "Hello"
Output: 1
Example 3:

Input: s = "love live! mu'sic forever"
Output: 4
Example 4:

Input: s = ""
Output: 0

Constraints:

0 <= s.length <= 300
s consists of lower-case and upper-case English letters, digits or one of the following characters "!@#$%^&*()_+-=',.:".
The only space character in s is ' '.

```
class Solution {
    public int countSegments(String s) {
        int segmentCount = 0;

        for (int i = 0; i < s.length(); i++) {
            if ((i == 0 || s.charAt(i-1) == ' ') && s.charAt(i) != ' ') {
                segmentCount++;
            }
        }

        return segmentCount;
    }
}
```

----------Non-overlapping Intervals
----------Given an array of intervals intervals where intervals[i] = [starti, endi], return the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.

Example 1:

Input: intervals = [[1,2],[2,3],[3,4],[1,3]]
Output: 1
Explanation: [1,3] can be removed and the rest of the intervals are non-overlapping.
Example 2:

Input: intervals = [[1,2],[1,2],[1,2]]
Output: 2
Explanation: You need to remove two [1,2] to make the rest of the intervals non-overlapping.
Example 3:

Input: intervals = [[1,2],[2,3]]
Output: 0
Explanation: You don't need to remove any of the intervals since they're already non-overlapping.

Constraints:

1 <= intervals.length <= 105
intervals[i].length == 2
-5 * 104 <= starti < endi <= 5 * 104

```
class Solution {
    public int eraseOverlapIntervals(int[][] intervals) {
        if(intervals == null || intervals.length <= 1) return 0;
        Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));
        int count = 0;


       for(int i =0, j = i+1; i < intervals.length && j < intervals.length; i++, j++){

            int [] a = intervals[i];
            int [] b = intervals[j];

           //if start of b is less than end of a (overlap)
           //overwrite the value of b with the max end and min start since thats the smaller interval
            if(b[0] < a[1]) {
                count++;
                b[0] = Math.max(a[0], a[1]);
                b[1] = Math.min(a[1], b[1]);

            }

        }


        return count;
    }
}
```

----------Find Right Interval
----------You are given an array of intervals, where intervals[i] = [starti, endi] and each starti is unique.

The right interval for an interval i is an interval j such that startj >= endi and startj is minimized.

Return an array of right interval indices for each interval i. If no right interval exists for interval i, then put -1 at index i.

Example 1:

Input: intervals = [[1,2]]
Output: [-1]
Explanation: There is only one interval in the collection, so it outputs -1.
Example 2:

Input: intervals = [[3,4],[2,3],[1,2]]
Output: [-1,0,1]
Explanation: There is no right interval for [3,4].
The right interval for [2,3] is [3,4] since start0 = 3 is the smallest start that is >= end1 = 3.
The right interval for [1,2] is [2,3] since start1 = 2 is the smallest start that is >= end2 = 2.
Example 3:

Input: intervals = [[1,4],[2,3],[3,4]]
Output: [-1,2,-1]
Explanation: There is no right interval for [1,4] and [3,4].
The right interval for [2,3] is [3,4] since start2 = 3 is the smallest start that is >= end1 = 3.

Constraints:

1 <= intervals.length <= 2 * 104
intervals[i].length == 2
-106 <= starti <= endi <= 106
The start point of each interval is unique.

```
// HashMap + Binary search
// 1. Use hashmap to store the <start, index> pair;
// 2. Use starts array to store the start time;
// 3. Sort the array;
// 4. Binary search each interval.end to get the position or insertion position.
//    Then get the coordinated index by the start.
// Time complexity: O(NlogN)
// Space complexity: O(N)
class Solution {
    public int[] findRightInterval(int[][] intervals) {
        if (intervals == null || intervals.length == 0) return new int[0];
        final int N = intervals.length;
        // Store <interval start, index> pair, start is unique.
        Map<Integer, Integer> indexMap = new HashMap<>();
        int[] starts = new int[N];
        for (int i = 0; i < N; i++) {
            starts[i] = intervals[i][0];
            indexMap.put(intervals[i][0], i);
        }
        Arrays.sort(starts);  // sort starts in ascending order
        int[] res = new int[N];
        for (int i = 0; i < N; i++) {
            int idx = Arrays.binarySearch(starts, intervals[i][1]);
            if (idx < 0) idx = -idx - 1;
            if (idx < N) {
                res[i] = indexMap.get(starts[idx]);
```

```
            } else {
                res[i] = -1;
            }
        }
        return res;
    }
}
```

----------Path Sum III

----------Given the root of a binary tree and an integer targetSum, return the number of paths where the sum of the values along the path equals targetSum.

The path does not need to start or end at the root or a leaf, but it must go downwards (i.e., traveling only from parent nodes to child nodes).

Example 1:

Input: root = [10,5,-3,3,2,null,11,3,-2,null,1], targetSum = 8
Output: 3
Explanation: The paths that sum to 8 are shown.
Example 2:

Input: root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22
Output: 3

Constraints:

The number of nodes in the tree is in the range [0, 1000].
-109 <= Node.val <= 109
-1000 <= targetSum <= 1000

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    private Map<Integer,Integer> hmp = new HashMap<>();
    int ans = 0,sum=0;
    private void solve(TreeNode root,int target) {
        if(root == null) return;
```

```
        sum+=root.val;
        if(hmp.containsKey(sum-target)) ans+=hmp.get(sum-target);
        int cnt = hmp.getOrDefault(sum,0);
        hmp.put(sum,cnt+1);
        solve(root.left,target);
        solve(root.right,target);
        if(hmp.get(sum) == 1) {
            hmp.remove(sum);
        } else hmp.put(sum,hmp.get(sum)-1);
        sum-=root.val;
    }
    public int pathSum(TreeNode root, int targetSum) {
        if(root == null) return 0;
        hmp.put(0,1);
        solve(root,targetSum);
        return ans;
    }
}
```

Find All Anagrams in a String

Given two strings s and p, return an array of all the start indices of p's anagrams in s. You may return the answer in any order.

An Anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Example 1:

Input: s = "cbaebabacd", p = "abc"
Output: [0,6]
Explanation:
The substring with start index = 0 is "cba", which is an anagram of "abc".
The substring with start index = 6 is "bac", which is an anagram of "abc".
Example 2:

Input: s = "abab", p = "ab"
Output: [0,1,2]
Explanation:
The substring with start index = 0 is "ab", which is an anagram of "ab".
The substring with start index = 1 is "ba", which is an anagram of "ab".
The substring with start index = 2 is "ab", which is an anagram of "ab".

Constraints:

1 <= s.length, p.length <= 3 * 104
s and p consist of lowercase English letters.

```
class Solution {
public List findAnagrams(String s, String p) {

   int freq1[] = new int[26];
   int freq2[] = new int[26];
```

```
    List<Integer> list = new ArrayList<>();

    if(s.length()<p.length())
        return list;

    for(int i=0; i<p.length(); i++){
        freq1[s.charAt(i)-'a']++;
        freq2[p.charAt(i)-'a']++;
    }

    int start=0;
    int end=p.length();

    if(Arrays.equals(freq1,freq2))
        list.add(start);

    while(end<s.length()){

        freq1[s.charAt(start)-'a']--;
        freq1[s.charAt(end)-'a']++;

        if(Arrays.equals(freq1,freq2))
        list.add(start+1);

        start++;
        end++;
    }
  return list;
}
}
```

----------K-th Smallest in Lexicographical Order

----------Given two integers n and k, return the kth lexicographically smallest integer in the range [1, n].

Example 1:

Input: n = 13, k = 2
Output: 10
Explanation: The lexicographical order is [1, 10, 11, 12, 13, 2, 3, 4, 5, 6, 7, 8, 9], so the second smallest number is 10.
Example 2:

Input: n = 1, k = 1
Output: 1

Constraints:

1 <= k <= n <= 109

```
class Solution {
    public int findKthNumber(int n, int k) {
        long cur = 1;
```

```
        while(k > 1) {
            long gap = findGap(cur, cur + 1, n);
            if(gap <= k - 1) {
                k -= gap;
                cur = cur + 1;
            }
            else {
                cur = cur * 10;
                k -= 1;
            }
        }

        return (int)cur;
    }

    private long findGap(long a, long b, int n) {
        long gap = 0;
        while(a <= n) {
            gap += Math.min(n + 1, b) - a;
            a = a * 10;
            b = b * 10;
        }
        return gap;
    }
}
```

----------Arranging Coins
----------You have n coins and you want to build a staircase with these coins. The staircase consists of k rows where the ith row has exactly i coins. The last row of the staircase may be incomplete.

Given the integer n, return the number of complete rows of the staircase you will build.

Example 1:

Input: n = 5
Output: 2
Explanation: Because the 3rd row is incomplete, we return 2.
Example 2:

Input: n = 8
Output: 3
Explanation: Because the 4th row is incomplete, we return 3.

Constraints:

1 <= n <= 231 - 1

```
class Solution {
  public int arrangeCoins(int n) {
```

```
    return (int)(Math.sqrt(2 * (long)n + 0.25) - 0.5);
  }
}

class Solution {
  public int arrangeCoins(int n) {
    long left = 0, right = n;
    long k, curr;
    while (left <= right) {
      k = left + (right - left) / 2;
      curr = k * (k + 1) / 2;

      if (curr == n) return (int)k;

      if (n < curr) {
        right = k - 1;
      } else {
        left = k + 1;
      }
    }
    return (int)right;
  }
}
```

----------Find All Duplicates in an Array
----------Given an integer array nums of length n where all the integers of nums are in the range [1, n] and each integer appears once or twice, return an array of all the integers that appears twice.

You must write an algorithm that runs in O(n) time and uses only constant extra space.

Example 1:

Input: nums = [4,3,2,7,8,2,3,1]
Output: [2,3]
Example 2:

Input: nums = [1,1,2]
Output: [1]
Example 3:

Input: nums = [1]
Output: []

Constraints:

n == nums.length
1 <= n <= 105
1 <= nums[i] <= n
Each element in nums appears once or twice.

```
class Solution {
```

```
    public List<Integer> findDuplicates(int[] arr) {
        int i=0;
        while(i< arr.length)
        {
            int correct = arr[i]-1;
            if(arr[i] == arr[correct])
            {
                i++;
            }
            else{
                swap(arr, correct, i);
            }
        }

        ArrayList<Integer> ans = new ArrayList<Integer>();
        for(int j=0;j<arr.length;j++)
        {
            if(arr[j] != j+1)
            {
                ans.add(arr[j]);
            }
        }
        return ans;
    }

    public void swap(int[] arr, int first, int second)
    {
        int temp = arr[first];
        arr[first] = arr[second];
        arr[second] = temp;
    }
}
```

----------Add Two Numbers II----------

You are given two non-empty linked lists representing two nonnegative integers. The most significant digit comes first and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.
You may assume the two numbers do not contain any leading zero, except the number 0 itself.
Follow up:
What if you cannot modify the input lists? In other words, reversing the lists is not allowed.

Example:
Input: (7 -> 2 -> 4 -> 3) + (5 -> 6 -> 4)
Output: 7 -> 8 -> 0 -> 7

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

```
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        l1 = reverseList(l1);
        l2 = reverseList(l2);
        if(len(l2) > len(l1)){
            ListNode temp = l1;
            l1 = l2;
            l2 = temp;
        }
        ListNode main = l1;
        int carry = 0;
        ListNode prev = l1;
        while(l1 != null){
            if(l2 != null){
                l1.val += l2.val;
                l2 = l2.next;
            }
            l1.val += carry;
            carry = l1.val / 10;
            l1.val %= 10;
            prev = l1;
            l1 = l1.next;
        }
        while(carry > 0){
            prev.next = new ListNode(carry);
            prev = prev.next;
            carry = prev.val / 10;
            prev.val %= 10;
        }
        return reverseList(main);
    }
    private int len(ListNode n){
        int l = 0;
        while(n!=null){
            n = n.next; l++;
        }
        return l;
    }
    public ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode temp = head;
        while(temp != null){
            ListNode currnext = temp.next;
            temp.next = prev;
            prev = temp;
            temp = currnext;
        }
        return prev;
    }
}
```

----------Arithmetic Slices II - Subsequence----------A sequence of numbers is called arithmetic if it consists of at least three elements and if the difference between any two
consecutive elements is the same.
For example, these are arithmetic sequences:
1, 3, 5, 7, 9
7, 7, 7, 7
3, -1, -5, -9
The following sequence is not arithmetic. 1, 1, 2, 5, 7

A zero-indexed array A consisting of N numbers is given. A subsequence slice of that array is any sequence of integers (P0, P1, ..., Pk) such that 0 ≤ P0 < P1 < ... < Pk < N.
A subsequence slice (P0, P1, ..., Pk) of array A is called arithmetic if the sequence A[P0], A[P1], ..., A[Pk-1], A[Pk] is arithmetic. In particular, this means that k ≥ 2.
The function should return the number of arithmetic subsequence slices in the array A.
The input contains N integers. Every integer is in the range of -231 and 231-1 and 0 ≤ N ≤ 1000. The output is guaranteed to be less than 231-1.
Example:
Input: [2, 4, 6, 8, 10]
Output: 7
Explanation:
All arithmetic subsequence slices are:
[2,4,6]
[4,6,8]
[6,8,10]
[2,4,6,8]
[4,6,8,10]
[2,4,6,8,10]
[2,6,10]
----------Number of Boomerangs
----------Given n points in the plane that are all pairwise distinct, a "boomerang" is a tuple of points (i, j, k) such that the distance between i and j equals the distance between i and k (the order of the tuple matters).
Find the number of boomerangs. You may assume that n will be at most 500 and coordinates of points are all in the range [-10000, 10000] (inclusive).
Example:
Input:
[[0,0],[1,0],[2,0]]
Output:
2

Explanation:
The two boomerangs are [[1,0],[0,0],[2,0]] and [[1,0],[2,0],[0,0]]
----------Find All Numbers Disappeared in an Array
----------Given an array of integers where 1 ≤ a[i] ≤ n (n = size of array),
some elements appear twice and others appear once.
Find all the elements of [1, n] inclusive that do not appear in this
array.
Could you do it without extra space and in O(n) runtime? You may
assume the returned list does not count as extra space.
Example:
Input:
[4,3,2,7,8,2,3,1]
Output:
[5,6]
----------Serialize and Deserialize BST
----------Serialization is the process of converting a data structure or
object into a sequence of bits so that it can be stored in a file or
memory buffer, or transmitted across a network connection link to be
reconstructed later in the same or another computer environment.
Design an algorithm to serialize and deserialize a binary search
tree. There is no restriction on how your serialization/
deserialization algorithm should work. You just need to ensure that
a binary search tree can be serialized to a string and this string
can be deserialized to the original tree structure.
The encoded string should be as compact as possible.

Note: Do not use class member/global/static variables to store
states. Your serialize and deserialize algorithms should be
stateless.
----------Delete Node in a BST

Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return the root node reference (possibly updated) of the BST.
Basically, the deletion can be divided into two stages:
Search for a node to remove.
If the node is found, delete the node.

Note: Time complexity should be O(height of tree).
Example:
root = [5,3,6,2,4,null,7]
key = 3
5
/ \
3
/ \
2

6
\
4

7

Given key to delete is 3. So we find the node with value 3 and delete it.
One valid answer is [5,4,6,2,null,null,7], shown in the following BST.
5
/ \
4

6

/

\

2

7

Another valid answer is [5,2,6,null,4,null,7].
5
/ \
2

6
\

\
4

7

----------Sort Characters By Frequency
----------Given a string, sort it in decreasing order based on the frequency of characters.
Example 1:

Input:
"tree"
Output:
"eert"
Explanation:
'e' appears twice while 'r' and 't' both appear once.
So 'e' must appear before both 'r' and 't'. Therefore "eetr" is also
a valid answer.

Example 2:
Input:
"cccaaa"
Output:
"cccaaa"
Explanation:
Both 'c' and 'a' appear three times, so "aaaccc" is also a valid
answer.
Note that "cacaca" is incorrect, as the same characters must be
together.

Example 3:
Input:
"Aabb"
Output:
"bbAa"
Explanation:
"bbaA" is also a valid answer, but "Aabb" is incorrect.
Note that 'A' and 'a' are treated as two different characters.
----------Minimum Moves to Equal Array Elements
----------Given a non-empty integer array of size n, find the minimum number
of moves required to make all array elements equal, where a move is
incrementing n - 1 elements by 1.
Example:
Input:
[1,2,3]

Output:
3
Explanation:
Only three moves are needed (remember each move increments two
elements):
[1,2,3]

=>

[2,3,3]

=>

[3,4,3]

=>

[4,4,4]

----------4Sum II
----------Given four lists A, B, C, D of integer values, compute how many
tuples (i, j, k, l) there are such that A[i] + B[j] + C[k] + D[l] is
zero.
To make problem a bit easier, all A, B, C, D have same length of N
where 0 ≤ N ≤ 500. All integers are in the range of -228 to
228 - 1 and the result is guaranteed to be at most 231 - 1.
Example:
Input:
A = [1, 2]
B = [-2,-1]
C = [-1, 2]
D = [0, 2]
Output:
2
Explanation:
The two tuples are:
1. (0, 0, 0, 1) -> A[0] + B[0] + C[0] + D[1] = 1 + (-2) + (-1) + 2 =
0
2. (1, 1, 0, 0) -> A[1] + B[1] + C[0] + D[0] = 2 + (-1) + (-1) + 0 =
0
----------Assign Cookies
----------Assume you are an awesome parent and want to give your children
some
cookies. But, you should give each child at most one cookie. Each
child i has a greed factor gi, which is the minimum size of a cookie
that the child will be content with; and each cookie j has a size
sj. If sj >= gi, we can assign the cookie j to the child i, and the
child i will be content. Your goal is to maximize the number of your
content children and output the maximum number.

Note:
You may assume the greed factor is always positive.
You cannot assign more than one cookie to one child.
Example 1:
Input: [1,2,3], [1,1]
Output: 1
Explanation: You have 3 children and 2 cookies. The greed factors of
3 children are 1, 2, 3.
And even though you have 2 cookies, since their size is both 1, you
could only make the child whose greed factor is 1 content.
You need to output 1.

Example 2:
Input: [1,2], [1,2,3]
Output: 2
Explanation: You have 2 children and 3 cookies. The greed factors of
2 children are 1, 2.
You have 3 cookies and their sizes are big enough to gratify all of
the children,
You need to output 2.
----------132 Pattern
----------Given a sequence of n integers a1, a2, ..., an, a 132 pattern is a
subsequence ai, aj, ak such
that i < j < k and ai < ak < aj. Design an algorithm that takes a
list of n numbers as input and checks whether there is a 132 pattern
in the list.
Note: n will be less than 15,000.
Example 1:
Input: [1, 2, 3, 4]
Output: False
Explanation: There is no 132 pattern in the sequence.

Example 2:
Input: [3, 1, 4, 2]
Output: True
Explanation: There is a 132 pattern in the sequence: [1, 4, 2].

Example 3:
Input: [-1, 3, 2, 0]
Output: True
Explanation: There are three 132 patterns in the sequence: [-1, 3,
2], [-1, 3, 0] and [-1, 2, 0].
--Repeated Substring Pattern
----------------------------Given a non-empty string check if it can be constructed by taking a
substring of it and appending multiple copies of the substring
together. You may assume the given string consists of lowercase
English letters only and its length will not exceed 10000.
Example 1:
Input: "abab"
Output: True
Explanation: It's the substring "ab" twice.

Example 2:
Input: "aba"
Output: False

Example 3:
Input: "abcabcabcabc"

Output: True
Explanation: It's the substring "abc" four times. (And the substring "abcabc" twice.)
----------LFU Cache
----------Design and implement a data structure for Least Frequently Used (LFU) cache. It should support the following operations: get and put.

get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.
put(key, value) - Set or insert the value if the key is not already present. When the cache reaches its capacity, it should invalidate the least frequently used item before inserting a new item. For the purpose of this problem, when there is a tie (i.e., two or more keys that have the same frequency), the least recently used key would be evicted.
Follow up:
Could you do both operations in O(1) time complexity?
Example:
LFUCache cache = new LFUCache(2 /* capacity */);
cache.put(1, 1);
cache.put(2, 2);
cache.get(1);
cache.put(3, 3);
cache.get(2);
cache.get(3);
cache.put(4, 4);
cache.get(1);
cache.get(3);
cache.get(4);

//
//
//
//
//
//
//
//

returns 1
evicts key
returns -1
returns 3.
evicts key
returns -1
returns 3
returns 4

2
(not found)
1.
(not found)

----------Hamming Distance
----------The Hamming distance between two integers is the number of positions
at which the corresponding bits are different.
Given two integers x and y, calculate the Hamming distance.

Note:
0 ≤ x, y < 231.
Example:
Input: x = 1, y = 4
Output: 2
Explanation:
1
(0 0 0 1)
4
(0 1 0 0)
↑
↑
The above arrows point to positions where the corresponding bits are
different.
----------Minimum Moves to Equal Array Elements II
----------Given a non-empty integer array, find the minimum number of moves
required to make all array elements equal, where a move is
incrementing a selected element by 1 or decrementing a selected
element by 1.
You may assume the array's length is at most 10,000.
Example:
Input:
[1,2,3]
Output:
2
Explanation:
Only two moves are needed (remember each move increments or
decrements one element):
[1,2,3]

=>

[2,2,3]

=>

[2,2,2]

----------Island Perimeter
----------You are given a map in form of a two-dimensional integer grid where
1 represents land and 0 represents water. Grid cells are connected
horizontally/vertically (not diagonally). The grid is completely
surrounded by water, and there is exactly one island (i.e., one or
more connected land cells). The island doesn't have "lakes" (water

inside that isn't connected to the water around the island). One cell is a square with side length 1. The grid is rectangular, width and height don't exceed 100. Determine the perimeter of the island.
Example:
[[0,1,0,0],
[1,1,1,0],
[0,1,0,0],
[1,1,0,0]]
Answer: 16
Explanation: The perimeter is the 16 yellow stripes in the image below:

----------Can I Win
----------In the "100 game," two players take turns adding, to a running total, any integer from 1..10. The player who first causes the running total to reach or exceed 100 wins.
What if we change the game so that players cannot re-use integers?
For example, two players might take turns drawing from a common pool of numbers of 1..15 without replacement until they reach a total >= 100.
Given an integer maxChoosableInteger and another integer desiredTotal, determine if the first player to move can force a win, assuming both players play optimally.
You can always assume that maxChoosableInteger will not be larger than 20 and desiredTotal will not be larger than 300.
Example
Input:
maxChoosableInteger = 10
desiredTotal = 11
Output:
false
Explanation:
No matter which integer the first player choose, the first player will lose.
The first player can choose an integer from 1 up to 10.
If the first player choose 1, the second player can only choose integers from 2 up to 10.

The second player will win by choosing 10 and get a total = 11,
which is >= desiredTotal.
Same with other integers chosen by the first player, the second
player will always win.
----------Count The
Repetitions
----------Define S = [s,n] as the string S which consists of n connected
strings s. For example, ["abc", 3] ="abcabcabc".
On the other hand, we define that string s1 can be obtained from
string s2 if we can remove some characters from s2 such that it
becomes s1. For example, “abc” can be obtained from “abdbec” based
on our definition, but it can not be obtained from “acbbe”.
You are given two non-empty strings s1 and s2 (each at most 100
characters long) and two integers 0 ≤ n1 ≤ 106 and 1 ≤ n2
≤ 106. Now consider the strings S1 and S2, where S1=[s1,n1] and
S2=[s2,n2]. Find the maximum integer M such that [S2,M] can be
obtained from S1.
Example:
Input:
s1="acb", n1=4
s2="ab", n2=2
Return:
2
----------Unique Substrings in Wraparound String
----------Consider the string s to be the infinite wraparound string of
"abcdefghijklmnopqrstuvwxyz", so s will look like this:
"...zabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcd....".
Now we have another string p. Your job is to find out how many
unique non-empty substrings of p are present in s. In particular,
your input is the string p and you need to output the number of
different non-empty substrings of p in the string s.
Note: p consists of only lowercase English letters and the size of p
might be over 10000.
Example 1:
Input: "a"
Output: 1

Explanation: Only the substring "a" of string "a" is in the string s.

Example 2:
Input: "cac"
Output: 2
Explanation: There are two substrings "a", "c" of string "cac" in the string s.

Example 3:
Input: "zab"
Output: 6
Explanation: There are six substrings "z", "a", "b", "za", "ab", "zab" of string "zab" in the string s.
----------Validate IP Address----------Write a function to check whether an input string is a valid IPv4 address or IPv6 address or neither.

IPv4 addresses are canonically represented in dot-decimal notation, which consists of four decimal numbers, each ranging from 0 to 255, separated by dots ("."), e.g.,172.16.254.1;

Besides, leading zeros in the IPv4 is invalid. For example, the address 172.16.254.01 is invalid.

IPv6 addresses are represented as eight groups of four hexadecimal digits, each group representing 16 bits. The groups are separated by colons (":"). For example, the address 2001:0db8:85a3:0000:0000:8a2e:0370:7334 is a valid one. Also, we could omit some leading zeros among four hexadecimal digits and some low-case characters in the address to upper-case ones, so 2001:db8:85a3:0:0:8A2E:0370:7334 is also a valid IPv6 address(Omit leading zeros and using upper cases).

However, we don't replace a consecutive group of zero value with a single empty group using two consecutive colons (::) to pursue simplicity. For example, 2001:0db8:85a3::8A2E:0370:7334 is an invalid IPv6 address.

Besides, extra leading zeros in the IPv6 is also invalid. For example, the address 02001:0db8:85a3:0000:0000:8a2e:0370:7334 is invalid.

Note:
You may assume there is no extra space or special characters in the input string.
Example 1:
Input: "172.16.254.1"
Output: "IPv4"
Explanation: This is a valid IPv4 address, return "IPv4".

Example 2:
Input: "2001:0db8:85a3:0:0:8A2E:0370:7334"
Output: "IPv6"
Explanation: This is a valid IPv6 address, return "IPv6".

Example 3:
Input: "256.256.256.256"
Output: "Neither"
Explanation: This is neither a IPv4 address nor a IPv6 address.

-------Concatenated Words

----------Given a list of words (without duplicates), please write a program that returns all concatenated words in the given list of words.
A concatenated word is defined as a string that is comprised entirely of at least two shorter words in the given array.
Example:
Input:
["cat","cats","catsdogcats","dog","dogcatsdog","hippopotamuses","rat","ratcatdogcat"]
Output: ["catsdogcats","dogcatsdog","ratcatdogcat"]
Explanation: "catsdogcats" can be concatenated by "cats", "dog" and "cats"; "dogcatsdog" can be concatenated by "dog", "cats" and "dog"; "ratcatdogcat" can be concatenated by "rat", "cat", "dog" and "cat".

Note:
The number of elements of the given array will not exceed 10,000
The length sum of elements in the given array will not exceed 600,000.
All the input string will only include lower case letters.
The returned elements order does not matter.
----------Matchsticks to Square
----------Remember the story of Little Match Girl? By now, you know exactly what matchsticks the little match girl has, please find out a way you can make one square by using up all those matchsticks. You should not break any stick, but you can link them up, and each matchstick must be used exactly one time.
Your input will be several matchsticks the girl has, represented with their stick length. Your output will either be true or false, to represent whether you could make one square using all the matchsticks the little match girl has.
Example 1:
Input: [1,1,2,2,2]
Output: true
Explanation: You can form a square with length 2, one side of the square came two sticks with length 1.

Example 2:
Input: [3,3,3,3,4]
Output: false
Explanation: You cannot find a way to form a square with all the matchsticks.

Note:
The length sum of the given matchsticks is in the range of 0 to 10^9.
The length of the given matchstick array will not exceed 15.
----------Ones and Zeroes
----------In the computer world, use restricted resource you have to generate maximum benefit is what we always want to pursue.
For now, suppose you are a dominator of m 0s and n 1s respectively. On the other hand, there is an array with strings consisting of only 0s and 1s.
Now your task is to find the maximum number of strings that you can form with given m 0s and n 1s. Each 0 and 1 can be used at most once.

Note:
The given numbers of 0s and 1s will both not exceed 100
The size of given string array won't exceed 600.

Example 1:
Input: Array = {"10", "0001", "111001", "1", "0"}, m = 5, n = 3
Output: 4
Explanation: This are totally 4 strings can be formed by the using of 5 0s and 3 1s, which are “10,”0001”,”1”,”0”

Example 2:

Input: Array = {"10", "0", "1"}, m = 1, n = 1
Output: 2
Explanation: You could form "10", but then you'd have nothing left.
Better form "0" and "1".
----------Heaters
----------Winter is coming! Your first job during the contest is to design a
standard heater with fixed warm radius to warm all the houses.
Now, you are given positions of houses and heaters on a horizontal
line, find out minimum radius of heaters so that all houses could be
covered by those heaters.
So, your input will be the positions of houses and heaters
seperately, and your expected output will be the minimum radius
standard of heaters.
Note:
Numbers of houses and heaters you are given are non-negative and
will not exceed 25000.
Positions of houses and heaters you are given are non-negative and
will not exceed 10^9.
As long as a house is in the heaters' warm radius range, it can be
warmed.
All the heaters follow your radius standard and the warm radius will
the same.

Example 1:
Input: [1,2,3],[2]
Output: 1
Explanation: The only heater was placed in the position 2, and if we
use the radius 1 standard, then all the houses can be warmed.

Example 2:
Input: [1,2,3,4],[1,4]
Output: 1
Explanation: The two heater was placed in the position 1 and 4. We
need to use radius 1 standard, then all the houses can be warmed.

Number Complement
----------Given a positive integer, output its complement number. The complement strategy is to flip the bits of its binary representation.
Note:
The given integer is guaranteed to fit within the range of a 32-bit signed integer.
You could assume no leading zero bit in the integer's binary representation.

Example 1:
Input: 5
Output: 2
Explanation: The binary representation of 5 is 101 (no leading zero bits), and its complement is 010. So you need to output 2.

Example 2:
Input: 1
Output: 0
Explanation: The binary representation of 1 is 1 (no leading zero bits), and its complement is 0. So you need to output 0.
----------Total Hamming Distance
----------The Hamming distance between two integers is the number of positions at which the corresponding bits are different.
Now your job is to find the total Hamming distance between all pairs of the given numbers.
Example:
Input: 4, 14, 2
Output: 6
Explanation: In binary representation, the 4 is 0100, 14 is 1110, and 2 is 0010 (just showing the four bits relevant in this case). So the answer will be: HammingDistance(4, 14) + HammingDistance(4, 2) + HammingDistance(14, 2) = 2 + 2 + 2 = 6.

Note:
Elements of the given array are in the range of 0
Length of the array will not exceed 10^4.

to 10^9

----------Sliding Window Median
----------Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.
Examples:
[2,3,4] , the median is 3
[2,3], the median is (2 + 3) / 2 = 2.5
Given an array nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position. Your job is to output the median array for each window in the original array.
For example,
Given nums = [1,3,-1,-3,5,3,6,7], and k = 3.
Window position
----------[1 3 -1] -3 5 3
1 [3 -1 -3] 5 3
1 3 [-1 -3 5] 3
1 3 -1 [-3 5 3]
1 3 -1 -3 [5 3
1 3 -1 -3 5 [3

6
6
6
6
6]
6

7
7
7
7
7
7]

Median
----1
-1
-1
3
5
6

Therefore, return the median sliding window as [1,-1,-1,3,5,6].
Note:
You may assume k is always valid, ie: 1 ≤ k ≤ input array's size for non-empty array.

----------Magical String
----------A magical string S consists of only '1' and '2' and obeys the following rules:

The string S is magical because concatenating the number of contiguous occurrences of characters '1' and '2' generates the string S itself.

The first few elements of string S is the following:
S = "1221121221221121122……"

If we group the consecutive '1's and '2's in S, it will be:
1

22

11

2

1

22

1

22

11

2

11

22 .....

and the occurrences of '1's or '2's in each group are:
1

2

2

1

1

2

1

2

2

1

2

2 .....

You can see that the occurrence sequence above is the S itself.

Given an integer N as input, return the number of '1's in the first
N number in the magical string S.
Note:
N will not exceed 100,000.

Example 1:
Input: 6
Output: 3
Explanation: The first 6 elements of magical string S is "12211" and
it contains three 1's, so return 3.
----------License Key Formatting
----------Now you are given a string S, which represents a software license
key which we would like to format. The string S is composed of
alphanumerical characters and dashes. The dashes split the
alphanumerical characters within the string into groups. (i.e. if
there are M dashes, the string is split into M+1 groups). The dashes

in the given string are possibly misplaced.
We want each group of characters to be of length K (except for possibly the first group, which could be shorter, but still must contain at least one character). To satisfy this requirement, we will reinsert dashes. Additionally, all the lower case letters in the string must be converted to upper case.
So, you are given a non-empty string S, representing a license key to format, and an integer K. And you need to return the license key formatted according to the description above.
Example 1:
Input: S = "2-4A0r7-4k", K = 4
Output: "24A0-R74K"
Explanation: The string S has been split into two parts, each part has 4 characters.

Example 2:
Input: S = "2-4A0r7-4k", K = 3
Output: "24-A0R-74K"
Explanation: The string S has been split into three parts, each part has 3 characters except the first part as it could be shorter as said above.

Note:
The length of string S will not exceed 12,000, and K is a positive integer.
String S consists only of alphanumerical characters (a-z and/or A-Z and/or 0-9) and dashes(-).
String S is non-empty.
----------Smallest Good Base
----------For an integer n, we call k>=2 a good base of n, if all digits of n base k are 1.
Now given a string representing n, you should return the smallest good base of n in string format.

Example 1:
Input: "13"
Output: "3"
Explanation: 13 base 3 is 111.

Example 2:
Input: "4681"
Output: "8"
Explanation: 4681 base 8 is 11111.

Example 3:
Input: "1000000000000000000"
Output: "999999999999999999"
Explanation: 1000000000000000000 base 999999999999999999 is 11.

Note:
The range of n is [3, 10^18].
The string representing n is always valid and will not have leading
zeros.
----------Max Consecutive
Ones
----------Given a binary array, find the maximum number of consecutive 1s in
this array.
Example 1:
Input: [1,1,0,1,1,1]
Output: 3
Explanation: The first two digits or the last three digits are
consecutive 1s.
The maximum number of consecutive 1s is 3.

Note:
The input array will only contain 0 and 1.
The length of input array is a positive integer and will not exceed

10,000
----------Predict the Winner
----------Given an array of scores that are non-negative integers. Player 1 picks one of the numbers from either end of the array followed by the player 2 and then player 1 and so on. Each time a player picks a number, that number will not be available for the next player. This continues until all the scores have been chosen. The player with the maximum score wins.
Given an array of scores, predict whether player 1 is the winner. You can assume each player plays to maximize his score.
Example 1:
Input: [1, 5, 2]
Output: False
Explanation: Initially, player 1 can choose between 1 and 2. If he chooses 2 (or 1), then player 2 can choose from 1 (or 2) and 5. If player 2 chooses 5, then player 1 will be left with 1 (or 2). So, final score of player 1 is 1 + 2 = 3, and player 2 is 5. Hence, player 1 will never be the winner and you need to return False.

Example 2:
Input: [1, 5, 233, 7]
Output: True
Explanation: Player 1 first chooses 1. Then player 2 have to choose between 5 and 7. No matter which number player 2 choose, player 1 can choose 233.Finally, player 1 has more score (234) than player 2 (12), so you need to return True representing player1 can win.

Note:
1 <= length of the array <= 20.
Any scores in the given array are non-negative integers and will not exceed 10,000,000.
If the scores of both players are equal, then player 1 is still the winner.
----------Zuma Game

Think about Zuma Game. You have a row of balls on the table, colored red(R), yellow(Y), blue(B), green(G), and white(W). You also have several balls in your hand.
Each time, you may choose a ball in your hand, and insert it into the row (including the leftmost place and rightmost place). Then, if there is a group of 3 or more balls in the same color touching, remove these balls. Keep doing this until no more balls can be removed.
Find the minimal balls you have to insert to remove all the balls on the table. If you cannot remove all the balls, output -1.
Examples:
Input: "WRRBBW", "RB"
Output: -1
Explanation: WRRBBW -> WRR[R]BBW -> WBBW -> WBB[B]W -> WW
Input: "WWRRBBWW", "WRBRW"
Output: 2
Explanation: WWRRBBWW -> WWRR[R]BBWW -> WWBBWW -> WWBB[B]WW -> WWWW -> empty
Input:"G", "GGGGG"
Output: 2
Explanation: G -> G[G] -> GG[G] -> empty
Input: "RBYYBBRRB", "YRBGB"
Output: 3
Explanation: RBYYBBRRB -> RBYY[Y]BBRRB -> RBBBRRB -> RRRB -> B -> B[B] -> BB[B] -> empty

Note:
You may assume that the initial row of balls on the table won’t have any 3 or more consecutive balls with the same color.
The number of balls on the table won't exceed 20, and the string represents these balls is called "board" in the input.
The number of balls in your hand won't exceed 5, and the string represents these balls is called "hand" in the input.
Both input strings will be non-empty and only contain characters 'R','Y','B','G','W'.
----------Increasing Subsequences
----------Given an integer array, your task is to find all the different

possible increasing subsequences of the given array, and the length of an increasing subsequence should be at least 2 .
Example:
Input: [4, 6, 7, 7]
Output: [[4, 6], [4, 7], [4, 6, 7], [4, 6, 7, 7], [6, 7], [6, 7, 7], [7,7], [4,7,7]]

Note:
The length of the given array will not exceed 15.
The range of integer in the given array is [-100,100].
The given array may contain duplicates, and two equal integers should also be considered as a special case of increasing sequence.
----------Construct the Rectangle
----------For a web developer, it is very important to know how to design a web page's size. So, given a specific rectangular web page's area, your job by now is to design a rectangular web page, whose length L and width W satisfy the following requirements:
1. The area of the rectangular web page you designed must equal to the given target area.
2. The width W should not be larger than the length L, which means L >= W.
3. The difference between length L and width W should be as small as possible.
You need to output the length L and the width W of the web page you designed in sequence.

Example:
Input: 4
Output: [2, 2]
Explanation: The target area is 4, and all the possible ways to construct it are [1,4], [2,2], [4,1].
But according to requirement 2, [1,4] is illegal; according to requirement 3, [4,1] is not optimal compared to [2,2]. So the length L is 2, and the width W is 2.

Note:

The given area won't exceed 10,000,000 and is a positive integer
The web page's width and length you designed must be positive integers.
----------Reverse Pairs
----------Given an array nums, we call (i, j) an important reverse pair if i < j and nums[i] > 2*nums[j].
You need to return the number of important reverse pairs in the given array.
Example1:
Input: [1,3,2,3,1]
Output: 2
Example2:
Input: [2,4,3,5,1]
Output: 3
Note:
The length of the given array will not exceed 50,000.
All the numbers in the input array are in the range of 32-bit integer.
----------Target Sum
----------You are given a list of non-negative integers, a1, a2, ..., an, and a target, S. Now you have 2 symbols + and -. For each integer, you should choose one from + and - as its new symbol.
Find out how many ways to assign symbols to make sum of integers equal to target S.
Example 1:
Input: nums is [1, 1, 1, 1, 1], S is 3.
Output: 5
Explanation:

-1+1+1+1+1
+1-1+1+1+1
+1+1-1+1+1
+1+1+1-1+1
+1+1+1+1-1

=
=
=
=
=

3
3
3
3
3

There are 5 ways to assign symbols to make the sum of nums be target 3.

Note:
The length of the given array is positive and will not exceed 20.
The sum of elements in the given array will not exceed 1000.
Your output answer is guaranteed to be fitted in a 32-bit integer.
----------Teemo Attacking
----------In LLP world, there is a hero called Teemo and his attacking can make his enemy Ashe be in poisoned condition. Now, given the Teemo's attacking ascending time series towards Ashe and the poisoning time duration per Teemo's attacking, you need to output the total time that Ashe is in poisoned condition.
You may assume that Teemo attacks at the very beginning of a specific time point, and makes Ashe be in poisoned condition immediately.
Example 1:
Input: [1,4], 2
Output: 4
Explanation: At time point 1, Teemo starts attacking Ashe and makes Ashe be poisoned immediately. This poisoned status will last 2 seconds until the end of time point 2. And at time point 4, Teemo attacks Ashe again, and causes Ashe to be in poisoned status for another 2 seconds. So you finally need to output 4.

Example 2:
Input: [1,2], 2
Output: 3
Explanation: At time point 1, Teemo starts attacking Ashe and makes Ashe be poisoned. This poisoned status will last 2 seconds until the

end of time point 2. However, at the beginning of time point 2,
Teemo attacks Ashe again who is already in poisoned status. Since
the poisoned status won't add up together, though the second
poisoning attack will still work at time point 2, it will stop at
the end of time point 3. So you finally need to output 3.

Note:
You may assume the
10000.
You may assume the
his poisoning time
which won't exceed

length of given time series array won't exceed
numbers in the Teemo's attacking time series and
duration per attacking are non-negative integers,
10,000,000.

----------Next Greater Element I
----------You are given two arrays (without duplicates) nums1 and nums2
where
nums1’s elements are subset of nums2. Find all the next greater
numbers for nums1's elements in the corresponding places of nums2.

The Next Greater Number of a number x in nums1 is the first greater
number to its right in nums2. If it does not exist, output -1 for
this number.
Example 1:
Input: nums1 = [4,1,2], nums2 = [1,3,4,2].
Output: [-1,3,-1]
Explanation:
For number 4 in the first array, you cannot find the next
greater number for it in the second array, so output -1.
For number 1 in the first array, the next greater number for it
in the second array is 3.
For number 2 in the first array, there is no next greater number
for it in the second array, so output -1.

Example 2:
Input: nums1 = [2,4], nums2 = [1,2,3,4].
Output: [3,-1]
Explanation:
For number 2 in the first array, the next greater number for it

in the second array is 3.
For number 4 in the first array, there is no next greater number
for it in the second array, so output -1.

Note:
All elements in nums1 and nums2 are unique.
The length of both nums1 and nums2 would not exceed 1000.
--Diagonal Traverse
----------------------------Given a matrix of M x N elements (M rows, N columns), return all
elements of the matrix in diagonal order as shown in the below
image.
Example:
Input:
[
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]
]
Output: [1,2,4,7,5,3,6,8,9]
Explanation:

Note:
The total number of elements of the given matrix will not exceed
10,000.
---Keyboard Row
----------------------------Given a List of words, return the words that can be typed using
letters of alphabet on only one row's of American keyboard like the
image below.

Example 1:
Input: ["Hello", "Alaska", "Dad", "Peace"]
Output: ["Alaska", "Dad"]

Note:
You may use one character in the keyboard more than once.
You may assume the input string will only contain letters of
alphabet.
----------Find Mode in Binary Search Tree
----------Given a binary search tree (BST) with duplicates, find all the
mode(s) (the most frequently occurred element) in the given BST.
Assume a BST is defined as follows:
The left subtree of a node contains only nodes with keys less than
or equal to the node's key.
The right subtree of a node contains only nodes with keys greater
than or equal to the node's key.
Both the left and right subtrees must also be binary search trees.

For example:
Given BST [1,null,2,2],
1
\
2
/
2

return [2].
Note:
If a tree has more than one mode, you can return them in any order.

Follow up:
Could you do that without using any extra space? (Assume that the
implicit stack space incurred due to recursion does not count).
--IPO
-----------------------------Suppose LeetCode will start its IPO soon. In order to sell a good
price of its shares to Venture Capital, LeetCode would like to work
on some projects to increase its capital before the IPO. Since it
has limited resources, it can only finish at most k distinct
projects before the IPO. Help LeetCode design the best way to
maximize its total capital after finishing at most k distinct
projects.

You are given several projects. For each project i, it has a pure
profit Pi and a minimum capital of Ci is needed to start the
corresponding project. Initially, you have W capital. When you
finish a project, you will obtain its pure profit and the profit
will be added to your total capital.

To sum up, pick a list of at most k distinct projects from given
projects to maximize your final capital, and output your final
maximized capital.
Example 1:
Input: k=2, W=0, Profits=[1,2,3], Capital=[0,1,1].
Output: 4
Explanation: Since your initial capital is 0, you can only start the
project indexed 0.
After finishing it you will obtain profit 1 and your
capital becomes 1.
With capital 1, you can either start the project
indexed 1 or the project indexed 2.
Since you can choose at most 2 projects, you need to
finish the project indexed 2 to get the maximum capital.
Therefore, output the final maximized capital, which is
0 + 1 + 3 = 4.

Note:

You may assume all numbers in the input are non-negative integers.
The length of Profits array and Capital array will not exceed
50,000.
The answer is guaranteed to fit in a 32-bit signed integer.
----------Next Greater Element II
----------Given a circular array (the next element of the last element is the first element of the array), print the Next Greater Number for every element. The Next Greater Number of a number x is the first greater number to its traversing-order next in the array, which means you could search circularly to find its next greater number. If it doesn't exist, output -1 for this number.
Example 1:
Input: [1,2,1]
Output: [2,-1,2]
Explanation: The first 1's next greater number is 2; The number 2 can't find next greater number; The second 1's next greater number needs to search circularly, which is also 2.

Note:
The length of given array won't exceed 10000.
----------Base 7
----------Given an integer, return its base 7 string representation.
Example 1:
Input: 100
Output: "202"

Example 2:
Input: -7
Output: "-10"

Note:
The input will be in range of [-1e7, 1e7].

----------Relative Ranks
----------Given scores of N athletes, find their relative ranks and the people with the top three highest scores, who will be awarded medals: "Gold Medal", "Silver Medal" and "Bronze Medal".
Example 1:
Input: [5, 4, 3, 2, 1]
Output: ["Gold Medal", "Silver Medal", "Bronze Medal", "4", "5"]
Explanation: The first three athletes got the top three highest scores, so they got "Gold Medal", "Silver Medal" and "Bronze Medal".
For the left two athletes, you just need to output their relative ranks according to their scores.

Note:
N is a positive integer and won't exceed 10,000.
All the scores of athletes are guaranteed to be unique.
----------Perfect Number
----------We define the Perfect Number is a positive integer that is equal to the sum of all its positive divisors except itself.
Now, given an integer n, write a function that returns true when it is a perfect number and false when it is not.
Example:
Input: 28
Output: True
Explanation: 28 = 1 + 2 + 4 + 7 + 14

Note:
The input number n will not exceed 100,000,000. (1e8)
----------Most Frequent Subtree Sum

Given the root of a tree, you are asked to find the most frequent subtree sum. The subtree sum of a node is defined as the sum of all the node values formed by the subtree rooted at that node (including the node itself). So what is the most frequent subtree sum value? If there is a tie, return all the values with the highest frequency in any order.
Examples 1
Input:
5
/

\
-3

2

return [2, -3, 4], since all the values happen only once, return all of them in any order.
Examples 2
Input:
5
/

\
-5

2

return [2], since 2 happens twice, however -5 only occur once.
Note:
You may assume the sum of values in any subtree is in the range of 32-bit signed integer.
----------Find Bottom Left Tree Value
----------Given a binary tree, find the leftmost value in the last row of the tree.
Example 1:
Input:
2
/ \
1

3

Output:
1

Example 2:
Input:
1
/ \
2

3

/

/ \

4

5

6

/
7
Output:
7

Note:
You may assume the tree (i.e., the given root node) is not NULL.
----------Freedom Trail
----------In the video game Fallout 4, the quest "Road to Freedom" requires players to reach a metal dial called the "Freedom Trail Ring", and use the dial to spell a specific keyword in order to open the door.

Given a string ring, which represents the code engraved on the outer ring and another string key, which represents the keyword needs to be spelled. You need to find the minimum number of steps in order to spell all the characters in the keyword.
Initially, the first character of the ring is aligned at 12:00 direction. You need to spell all the characters in the string key one by one by rotating the ring clockwise or anticlockwise to make each character of the string key aligned at 12:00 direction and then by pressing the center button.
At the stage of rotating the ring to spell the key character key[i]:
You can rotate the ring clockwise or anticlockwise one place, which counts as 1 step. The final purpose of the rotation is to align one of the string ring's characters at the 12:00 direction, where this character must equal to the character key[i].
If the character key[i] has been aligned at the 12:00 direction, you need to press the center button to spell, which also counts as 1

step. After the pressing, you could begin to spell the next character in the key (next stage), otherwise, you've finished all the spelling.

Example:

Input: ring = "godding", key = "gd"
Output: 4
Explanation: For the first key character 'g', since it is already in place, we just need 1 step to spell this character. For the second key character 'd', we need to rotate the ring "godding" anticlockwise by two steps to make it become "ddinggo". Also, we need 1 more step for spelling. So the final output is 4.

Note:
Length of both ring and key will be in range 1 to 100.
There are only lowercase letters in both strings and might be some duplcate characters in both strings.
It's guaranteed that string key could always be spelled by rotating the string ring.
----------Find Largest Value in Each Tree Row
----------You need to find the largest value in each row of a binary tree.
Example:
Input:
1
/ \
3

2

/ \
5

\
3

9

Output: [1, 3, 9]
----------Longest Palindromic Subsequence

----------Given a string s, find the longest palindromic subsequence's length in s. You may assume that the maximum length of s is 1000.
Example 1:
Input:
"bbbab"
Output:
4
One possible longest palindromic subsequence is "bbbb".
Example 2:
Input:
"cbbd"
Output:
2
One possible longest palindromic subsequence is "bb".
----------Super Washing Machines
----------You have n super washing machines on a line. Initially, each washing machine has some dresses or is empty.
For each move, you could choose any m (1 ≤ m ≤ n) washing machines, and pass one dress of each washing machine to one of its adjacent washing machines at the same time .
Given an integer array representing the number of dresses in each washing machine from left to right on the line, you should find the minimum number of moves to make all the washing machines have the same number of dresses. If it is not possible to do it, return -1.
Example1
Input: [1,0,5]
Output: 3
Explanation:
1st move:
1

0 <-- 5

=>

1

1

4

2nd move:
3rd move:

1 <-- 1 <-- 4
2
1 <-- 3

=>
=>

2
2

1
2

3
2

=>
=>

1
1

2
1

0
1

Example2
Input: [0,3,0]
Output: 2
Explanation:
1st move:
0 <-- 3
0
2nd move:
1
2 --> 0
Example3
Input: [0,2,0]
Output: -1
Explanation:
It's impossible to make all the three washing machines have the same
number of dresses.

Note:
The range of n is [1, 10000].
The range of dresses number in a super washing machine is [0, 1e5].
----------Detect Capital
----------Given a word, you need to judge whether the usage of capitals in it
is right or not.

We define the usage of capitals in a word to be right when one of the following cases holds:
All letters in this word are capitals, like "USA".
All letters in this word are not capitals, like "leetcode".
Only the first letter in this word is capital if it has more than one letter, like "Google".
Otherwise, we define that this word doesn't use capitals in a right way.

Example 1:
Input: "USA"
Output: True

Example 2:
Input: "FlaG"
Output: False

Note:
The input will be a non-empty word consisting of uppercase and
lowercase latin letters.
----------Longest Uncommon Subsequence II
----------Given a list of strings, you need to find the longest uncommon subsequence among them. The longest uncommon subsequence is defined as the longest subsequence of one of these strings and this subsequence should not be any subsequence of the other strings.

A subsequence is a sequence that can be derived from one sequence by deleting some characters without changing the order of the remaining elements. Trivially, any string is a subsequence of itself and an empty string is a subsequence of any string.

The input will be a list of strings, and the output needs to be the length of the longest uncommon subsequence. If the longest uncommon subsequence doesn't exist, return -1.
Example 1:
Input: "aba", "cdc", "eae"
Output: 3

Note:

All the given strings' lengths will not exceed 10.
The length of the given list will be in the range of [2, 50].
----------Continuous Subarray Sum
----------Given a list of non-negative numbers and a target integer k, write a function to check if the array has a continuous subarray of size at least 2 that sums up to the multiple of k, that is, sums up to n*k where n is also an integer.

Example 1:
Input: [23, 2, 4, 6, 7], k=6
Output: True
Explanation: Because [2, 4] is a continuous subarray of size 2 and sums up to 6.

Example 2:
Input: [23, 2, 6, 4, 7], k=6
Output: True
Explanation: Because [23, 2, 6, 4, 7] is an continuous subarray of size 5 and sums up to 42.

Note:
The length of the array won't exceed 10,000.
You may assume the sum of all the numbers is in the range of a signed 32-bit integer.
----------Longest Word in Dictionary through Deleting
----------Given a string and a string dictionary, find the longest string in the dictionary that can be formed by deleting some characters of the given string. If there are more than one possible results, return the longest word with the smallest lexicographical order. If there is no possible result, return the empty string.

Example 1:
Input:
s = "abpcplea", d = ["ale","apple","monkey","plea"]
Output:
"apple"

Example 2:
Input:
s = "abpcplea", d = ["a","b","c"]
Output:
"a"

Note:
All the strings in the input will only contain lower-case letters.
The size of the dictionary won't exceed 1,000.
The length of all the strings in the input won't exceed 1,000.
----------Contiguous Array
----------Given a binary array, find the maximum length of a contiguous subarray with equal number of 0 and 1.
Example 1:
Input: [0,1]
Output: 2
Explanation: [0, 1] is the longest contiguous subarray with equal number of 0 and 1.

Example 2:
Input: [0,1,0]
Output: 2
Explanation: [0, 1] (or [1, 0]) is a longest contiguous subarray with equal number of 0 and 1.

Note:
The length of the given binary array will not exceed 50,000.
----------Beautiful Arrangement
----------Suppose you have N integers from 1 to N. We define a beautiful arrangement as an array that is constructed by these N numbers successfully if one of the following is true for the ith position (1 ≤ i ≤ N) in this array:
The number at the ith position is divisible by i.
i is divisible by the number at the ith position.

Now given N, how many beautiful arrangements can you construct?
Example 1:
Input: 2
Output: 2
Explanation:
The first beautiful arrangement is [1, 2]:
Number at the 1st position (i=1) is 1, and 1 (i=1).
Number at the 2nd position (i=2) is 2, and 2 (i=2).
The second beautiful arrangement is [2, 1]:
Number at the 1st position (i=1) is 2, and 2 (i=1).
Number at the 2nd position (i=2) is 1, and i 1.

is divisible by i
is divisible by i
is divisible by i
(i=2) is divisible by

Note:
N is a positive integer and will not exceed 15.
----------Minesweeper
----------Let's play the minesweeper game (Wikipedia, online game)!
You are given a 2D char matrix representing the game board. 'M' represents an unrevealed mine, 'E' represents an unrevealed empty

square,
(above,
to '8')
square,

'B' represents a revealed blank square that has no adjacent
below, left, right, and all 4 diagonals) mines, digit ('1'
represents how many mines are adjacent to this revealed
and finally 'X' represents a revealed mine.

Now given the next click position (row and column indices) among all
the unrevealed squares ('M' or 'E'), return the board after
revealing this position according to the following rules:

If a mine ('M') is revealed, then the game is over - change it to
'X'.
If an empty square ('E') with no adjacent mines is revealed, then
change it to revealed blank ('B') and all of its adjacent unrevealed
squares should be revealed recursively.
If an empty square ('E') with at least one adjacent mine is
revealed, then change it to a digit ('1' to '8') representing the
number of adjacent mines.
Return the board when no more squares will be revealed.

Example 1:
Input:
[['E',
['E',
['E',
['E',

'E',
'E',
'E',
'E',

'E',
'M',
'E',
'E',

'E',
'E',
'E',
'E',

'E'],
'E'],
'E'],
'E']]

'E',
'M',
'1',

'B',

'1',
'1',
'1',
'B',

'B'],
'B'],
'B'],
'B']]

'E',
'M',
'1',
'B',

'1',
'1',
'1',
'B',

'B'],
'B'],
'B'],
'B']]

Click : [3,0]
Output:
[['B',
['B',
['B',
['B',

'1',
'1',
'1',
'B',

Explanation:

Example 2:
Input:
[['B',
['B',
['B',
['B',

'1',
'1',
'1',
'B',

Click : [1,2]
Output:
[['B',
['B',
['B',
['B',

'1',
'1',
'1',
'B',

'E',
'X',
'1',
'B',

'1',
'1',
'1',
'B',

'B'],
'B'],
'B'],
'B']]

Explanation:

Note:
The range of the input matrix's height and width is [1,50].
The click position will only be an unrevealed square ('M' or 'E'),
which also means the input board contains at least one clickable
square.
The input board won't be a stage when game is over (some mines have
been revealed).
For simplicity, not mentioned rules should be ignored in this
problem. For example, you don't need to reveal all the unrevealed
mines when the game is over, consider any cases that you will win
the game or flag any squares.
----------Minimum Absolute Difference in BST
----------Given a binary search tree with non-negative values, find the
minimum absolute difference between values of any two nodes.
Example:
Input:
1
\
3
/
2
Output:
1
Explanation:
The minimum absolute difference is 1, which is the difference

between 2 and 1 (or between 2 and 3).

Note:
There are at least two nodes in this BST.
----------K-diff Pairs in an Array
----------Given an array of integers and an integer k, you need to find the number of unique k-diff pairs in the array. Here a k-diff pair is defined as an integer pair (i, j), where i and j are both numbers in the array and their absolute difference is k.

Example 1:
Input: [3, 1, 4, 1, 5], k = 2
Output: 2
Explanation: There are two 2-diff pairs in the array, (1, 3) and (3, 5).Although we have two 1s in the input, we should only return the number of unique pairs.

Example 2:
Input:[1, 2, 3, 4, 5], k = 1
Output: 4
Explanation: There are four 1-diff pairs in the array, (1, 2), (2, 3), (3, 4) and (4, 5).

Example 3:
Input: [1, 3, 1, 5, 4], k = 0
Output: 1
Explanation: There is one 0-diff pair in the array, (1, 1).

Note:
The pairs (i, j) and (j, i) count as the same pair.
The length of the array won't exceed 10,000.
All the integers in the given input belong to the range: [-1e7, 1e7].

----------Encode and Decode TinyURL
----------Note: This is a companion problem to the System Design problem: Design TinyURL.
TinyURL is a URL shortening service where you enter a URL such as https://leetcode.com/problems/design-tinyurl and it returns a short URL such as http://tinyurl.com/4e9iAk.
Design the encode and decode methods for the TinyURL service. There is no restriction on how your encode/decode algorithm should work. You just need to ensure that a URL can be encoded to a tiny URL and the tiny URL can be decoded to the original URL.
----------Complex Number Multiplication
----------Given two strings representing two complex numbers. You need to return a string representing their multiplication. Note i2 = -1 according to the definition.
Example 1:
Input: "1+1i", "1+1i"
Output: "0+2i"
Explanation: (1 + i) * (1 + i) = 1 + i2 + 2 * i = 2i, and you need convert it to the form of 0+2i.

Example 2:
Input: "1+-1i", "1+-1i"
Output: "0+-2i"
Explanation: (1 - i) * (1 - i) = 1 + i2 - 2 * i = -2i, and you need convert it to the form of 0+-2i.

Note:
The input strings will not have extra blank.
The input strings will be given in the form of a+bi, where the integer a and b will both belong to the range of [-100, 100]. And

the output should be also in this form.
----------Convert BST to Greater Tree
----------Given a Binary Search Tree (BST), convert it to a Greater Tree such that every key of the original BST is changed to the original key
plus sum of all keys greater than the original key in BST.
Example:
Input: The root of a Binary Search Tree like this:
5
/
\
2
13
Output: The root of a Greater Tree like this:
18
/
\
20
13
----------Minimum Time Difference
----------Given a list of 24-hour clock time points in "Hour:Minutes" format, find the minimum minutes difference between any two time points in
the list.
Example 1:
Input: ["23:59","00:00"]
Output: 1

Note:
The number of time points in the given list is at least 2 and won't
exceed 20000.
The input time is legal and ranges from 00:00 to 23:59.
----------Reverse String II
----------Given a string and an integer k, you need to reverse the first k characters for every 2k characters counting from the start of the

string. If there are less than k characters left, reverse all of them. If there are less than 2k but greater than or equal to k characters, then reverse the first k characters and left the other as original.
Example:
Input: s = "abcdefg", k = 2
Output: "bacdfeg"

Restrictions:
The string consists of lower English letters only.
Length of the given string and k will in the range [1, 10000]
----------01 Matrix
----------Given a matrix consists of 0 and 1, find the distance of the nearest 0 for each cell.
The distance between two adjacent cells is 1.
Example 1:
Input:
0 0 0
0 1 0
0 0 0
Output:
0 0 0
0 1 0
0 0 0

Example 2:
Input:
0 0 0
0 1 0
1 1 1
Output:
0 0 0
0 1 0

1 2 1

Note:
The number of elements of the given matrix will not exceed 10,000.
There are at least one 0 in the given matrix.
The cells are adjacent in only four directions: up, down, left and
right.
----------Diameter of Binary Tree
----------Given a binary tree, you need to compute the length of the diameter
of the tree. The diameter of a binary tree is the length of the
longest path between any two nodes in a tree. This path may or may
not pass through the root.

Example:
Given a binary tree
1
/ \
2
/ \
4

3
5

Return 3, which is the length of the path [4,2,1,3] or [5,2,1,3].
Note:
The length of path between two nodes is represented by the number of
edges between them.
----------Remove Boxes
----------Given several boxes with different colors represented by different
positive numbers.
You may experience several rounds to remove boxes until there is no
box left. Each time you can choose some continuous boxes with the
same color (composed of k boxes, k >= 1), remove them and get k*k
points.

Find the maximum points you can get.
Example 1:
Input:
[1, 3, 2, 2, 2, 3, 4, 3, 1]
Output:
23
Explanation:
[1, 3, 2, 2, 2, 3, 4, 3, 1]
----> [1, 3, 3, 4, 3, 1] (3*3=9 points)
----> [1, 3, 3, 3, 1] (1*1=1 points)
----> [1, 1] (3*3=9 points)
----> [] (2*2=4 points)

Note:
The number of boxes n would not exceed 100.
----------Friend Circles
----------There are N students in a class. Some of them are friends, while some are not. Their friendship is transitive in nature. For example, if A is a direct friend of B, and B is a direct friend of C, then A is an indirect friend of C. And we defined a friend circle is a group of students who are direct or indirect friends.

Given a N*N matrix M representing the friend relationship between students in the class. If M[i][j] = 1, then the ith and jth students are direct friends with each other, otherwise not. And you have to output the total number of friend circles among all the students.
Example 1:
Input:
[[1,1,0],
[1,1,0],
[0,0,1]]
Output: 2
Explanation:The 0th and 1st students are direct friends, so they are in a friend circle. The 2nd student himself is in a friend circle. So return 2.

Example 2:
Input:
[[1,1,0],
[1,1,1],
[0,1,1]]
Output: 1
Explanation:The 0th and 1st students are direct friends, the 1st and 2nd students are direct friends, so the 0th and 2nd students are indirect friends. All of them are in the same friend circle, so return 1.

Note:
N is in range [1,200].
M[i][i] = 1 for all students.
If M[i][j] = 1, then M[j][i] = 1.
----------Design TinyURL----------Note: For the coding companion problem, please see: Encode and Decode TinyURL.
How would you design a URL shortening service that is similar to TinyURL?
Background:
TinyURL is a URL shortening service where you enter a URL such as https://leetcode.com/problems/design-tinyurl and it returns a short URL such as http://tinyurl.com/4e9iAk.
Requirements:
For instance, "http://tinyurl.com/4e9iAk" is the tiny url for the page "https://leetcode.com/problems/design-tinyurl". The identifier (the highlighted part) can be any string with 6 alphanumeric characters containing 0-9, a-z, A-Z.
Each shortened URL must be unique; that is, no two different URLs can be shortened to the same URL.

Note about Questions:Below are just a small subset of questions to get you started. In real world, there could be many follow ups and questions possible and the discussion is open-ended (No one true or correct way to solve a problem). If you have more ideas or questions, please ask in Discuss and we may compile it here!
Questions:
How many unique identifiers possible? Will you run out of unique URLs?
Should the identifier be increment or not? Which is easier to design? Pros and cons?
Mapping an identifier to an URL and its reversal - Does this problem ring a bell to you?
How do you store the URLs? Does a simple flat file database work?
What is the bottleneck of the system? Is it read-heavy or writeheavy?
Estimate the maximum number of URLs a single machine can store.
Estimate the maximum number of queries per second (QPS) for decoding a shortened URL in a single machine.
How would you scale the service? For example, a viral link which is shared in social media could result in a peak QPS at a moment's notice.
How could you handle redundancy? i,e, if a server is down, how could you ensure the service is still operational?
Keep URLs forever or prune, pros/cons? How we do pruning?
(Contributed by @alex_svetkin)
What API would you provide to a third-party developer? (Contributed by @alex_svetkin)
If you can enable caching, what would you cache and what's the expiry time? (Contributed by @Humandroid)

```
.hilight {
color: #d14;
background-color: #f7f7f9;
padding: 1px 3px;
border: 1px solid #e1e1e8"
}
```

License Key Formatting

Now you are given a string S, which represents a software license key which we would like to format. The string S is composed of alphanumerical characters and dashes. The dashes split the alphanumerical characters within the string into groups. (i.e. if there are M dashes, the string is split into M+1 groups). The dashes in the given string are possibly misplaced.
We want each group of characters to be of length K (except for

possibly the first group, which could be shorter, but still must contain at least one character). To satisfy this requirement, we will reinsert dashes. Additionally, all the lower case letters in the string must be converted to upper case.
So, you are given a non-empty string S, representing a license key to format, and an integer K. And you need to return the license key formatted according to the description above.
Example 1:
Input: S = "2-4A0r7-4k", K = 4
Output: "24A0-R74K"
Explanation: The string S has been split into two parts, each part has 4 characters.

Example 2:
Input: S = "2-4A0r7-4k", K = 3
Output: "24-A0R-74K"
Explanation: The string S has been split into three parts, each part has 3 characters except the first part as it could be shorter as said above.

Note:
The length of string S will not exceed 12,000, and K is a positive integer.
String S consists only of alphanumerical characters (a-z and/or A-Z and/or 0-9) and dashes(-).
String S is non-empty.
----------Longest Absolute File Path
----------Suppose we abstract our file system by a string in the following manner:
The string "dir\n\tsubdir1\n\tsubdir2\n\t\tfile.ext" represents:
dir
subdir1

subdir2
file.ext
The directory dir contains an empty sub-directory subdir1 and a subdirectory subdir2 containing a file file.ext.
The string
"dir\n\tsubdir1\n\t\tfile1.ext\n\t\tsubsubdir1\n\tsubdir2\n\t\tsubsu
bdir2\n\t\t\tfile2.ext" represents:
dir
subdir1
file1.ext
subsubdir1
subdir2
subsubdir2
file2.ext
The directory dir contains two sub-directories subdir1 and subdir2.
subdir1 contains a file file1.ext and an empty second-level subdirectory subsubdir1. subdir2 contains a second-level sub-directory
subsubdir2 containing a file file2.ext.
We are interested in finding the longest (number of characters)
absolute path to a file within our file system. For example, in the
second example above, the longest absolute path is "dir/subdir2/
subsubdir2/file2.ext", and its length is 32 (not including the
double quotes).
Given a string representing the file system in the above format,
return the length of the longest absolute path to file in the
abstracted file system. If there is no file in the system, return 0.
Note:
The name of a file contains at least a . and an extension.
The name of a directory or sub-directory will not contain a ..

Time complexity required: O(n) where n is the size of the input
string.
Notice that a/aa/aaa/file1.txt is not the longest file path, if
there is another path aaaaaaaaaaaaaaaaaaaa/sth.png.
----------Encode and Decode TinyURL
----------Note: This is a companion problem to the System Design problem:
Design TinyURL.

TinyURL is a URL shortening service where you enter a URL such as https://leetcode.com/problems/design-tinyurl and it returns a short URL such as http://tinyurl.com/4e9iAk.
Design the encode and decode methods for the TinyURL service. There is no restriction on how your encode/decode algorithm should work. You just need to ensure that a URL can be encoded to a tiny URL and the tiny URL can be decoded to the original URL.

----------Coin Change
----------You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Return 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.

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

Example 1:

Input: coins = [1,2,5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1
Example 2:

Input: coins = [2], amount = 3
Output: -1
Example 3:

Input: coins = [1], amount = 0
Output: 0
Example 4:

Input: coins = [1], amount = 1
Output: 1
Example 5:

Input: coins = [1], amount = 2
Output: 2

Constraints:

1 <= coins.length <= 12
1 <= coins[i] <= 231 - 1
0 <= amount <= 104

```
// DP knapsack solution
// DP[i][j] means the min coins we can use to get the amount
// Time complexity: O(N*amount)
// Space complexity: O(N*amount)
class Solution {
    public int coinChange(int[] coins, int amount) {
```

```
        if (coins == null || coins.length == 0) return -1; // or throw exception
        if (amount == 0) return 0;
        final int N = coins.length, MAX = amount+1;
        int[][] dp = new int[N+1][MAX]; // amount <= 10^4
        Arrays.fill(dp[0], MAX);
        dp[0][0] = 0;
        for (int i = 1; i <= N; i++) {
            int coin = coins[i-1];
            for (int j = 0; j <= amount; j++) {
                if (j >= coin) {
                    dp[i][j] = Math.min(dp[i-1][j], dp[i][j-coin] + 1);
                } else {
                    dp[i][j] = dp[i-1][j];
                }
            }
        }
        return dp[N][amount] >= MAX ? -1 : dp[N][amount];
    }
}
```

----------Poor Pigs----------There are buckets buckets of liquid, where exactly one of the buckets is poisonous. To figure out which one is poisonous, you feed some number of (poor) pigs the liquid to see whether they will die or not. Unfortunately, you only have minutesToTest minutes to determine which bucket is poisonous.

You can feed the pigs according to these steps:

Choose some live pigs to feed.
For each pig, choose which buckets to feed it. The pig will consume all the chosen buckets simultaneously and will take no time.
Wait for minutesToDie minutes. You may not feed any other pigs during this time.
After minutesToDie minutes have passed, any pigs that have been fed the poisonous bucket will die, and all others will survive.
Repeat this process until you run out of time.
Given buckets, minutesToDie, and minutesToTest, return the minimum number of pigs needed to figure out which bucket is poisonous within the allotted time.

Example 1:

Input: buckets = 1000, minutesToDie = 15, minutesToTest = 60
Output: 5
Example 2:

Input: buckets = 4, minutesToDie = 15, minutesToTest = 15
Output: 2
Example 3:

Input: buckets = 4, minutesToDie = 15, minutesToTest = 30
Output: 2

Constraints:

1 <= buckets <= 1000
1 <= minutesToDie <= minutesToTest <= 100

```
class Solution {
    public int poorPigs(int buckets, int minutesToDie, int minutesToTest) {
        int T = (minutesToTest/minutesToDie) + 1;
        int cnt = 0;
        int total = 1;
        while (total < buckets) {
            total *= T;
            cnt++;
        }
        return cnt;
    }
}
```

----------Minimum Genetic Mutation
----------A gene string can be represented by an 8-character long string, with choices from 'A', 'C', 'G', and 'T'.

Suppose we need to investigate a mutation from a gene string start to a gene string end where one mutation is defined as one single character changed in the gene string.

For example, "AACCGGTT" --> "AACCGGTA" is one mutation.
There is also a gene bank bank that records all the valid gene mutations. A gene must be in bank to make it a valid gene string.

Given the two gene strings start and end and the gene bank bank, return the minimum number of mutations needed to mutate from start to end. If there is no such a mutation, return -1.

Note that the starting point is assumed to be valid, so it might not be included in the bank.

Example 1:

Input: start = "AACCGGTT", end = "AACCGGTA", bank = ["AACCGGTA"]
Output: 1
Example 2:

Input: start = "AACCGGTT", end = "AAACGGTA", bank =
["AACCGGTA","AACCGCTA","AAACGGTA"]
Output: 2
Example 3:

Input: start = "AAAAACCC", end = "AACCCCCC", bank =
["AAAACCCC","AAACCCCC","AACCCCCC"]
Output: 3

Constraints:

start.length == 8

end.length == 8
0 <= bank.length <= 10
bank[i].length == 8
start, end, and bank[i] consist of only the characters ['A', 'C', 'G', 'T'].

```
class Solution {

    public int minMutation(String start, String end, String[] bank) {
        if(bank.length == 0){
            return -1;
        }
        //make bank set
        Set<String> bankSet = new HashSet<>();
        for(String gene:bank){
            bankSet.add(gene);
        }
        if(!bankSet.contains(end)){
            return -1;
        }
        Set<String> visited = new HashSet<>();
        Queue<String> dq = new ArrayDeque<>();
        char[] letterArr = {'A','C','G','T'};
        //bfs
        dq.add(start);
        int depht = 0;
        while(dq.size() != 0){
            int size = dq.size();
            for(int j=0;j<size;j++){
                String gene = dq.poll();
                if(gene.equals(end)){
                    return depht;
                }
                StringBuilder str = new StringBuilder(gene);
                for(int i=0;i<str.length();i++){
                    char currLetter = str.charAt(i);
                    for(char letter:letterArr){
                        if(letter != currLetter){
                            str.setCharAt(i,letter);
                            if(bankSet.contains(str.toString()) && !visited.contains(str.toString())){
                                dq.add(str.toString());
                                visited.add(str.toString());
                            }
                        }
                    }
                    str.setCharAt(i,currLetter);
                }
            }
            depht++;
        }
        return -1;
    }
}
```

LFU Cache

----------Design and implement a data structure for a Least Frequently Used (LFU) cache.

Implement the LFUCache class:

LFUCache(int capacity) Initializes the object with the capacity of the data structure.
int get(int key) Gets the value of the key if the key exists in the cache. Otherwise, returns -1.
void put(int key, int value) Update the value of the key if present, or inserts the key if not already present. When the cache reaches its capacity, it should invalidate and remove the least frequently used key before inserting a new item. For this problem, when there is a tie (i.e., two or more keys with the same frequency), the least recently used key would be invalidated.
To determine the least frequently used key, a use counter is maintained for each key in the cache. The key with the smallest use counter is the least frequently used key.

When a key is first inserted into the cache, its use counter is set to 1 (due to the put operation). The use counter for a key in the cache is incremented either a get or put operation is called on it.

The functions get and put must each run in O(1) average time complexity.

Example 1:

Input
["LFUCache", "put", "put", "get", "put", "get", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [3], [4, 4], [1], [3], [4]]
Output
[null, null, null, 1, null, -1, 3, null, -1, 3, 4]

Explanation
// cnt(x) = the use counter for key x
// cache=[] will show the last used order for tiebreakers (leftmost element is most recent)
LFUCache lfu = new LFUCache(2);
lfu.put(1, 1);   // cache=[1,_], cnt(1)=1
lfu.put(2, 2);   // cache=[2,1], cnt(2)=1, cnt(1)=1
lfu.get(1);   // return 1
// cache=[1,2], cnt(2)=1, cnt(1)=2
lfu.put(3, 3);   // 2 is the LFU key because cnt(2)=1 is the smallest, invalidate 2.
// cache=[3,1], cnt(3)=1, cnt(1)=2
lfu.get(2);   // return -1 (not found)
lfu.get(3);   // return 3
// cache=[3,1], cnt(3)=2, cnt(1)=2
lfu.put(4, 4);   // Both 1 and 3 have the same cnt, but 1 is LRU, invalidate 1.
// cache=[4,3], cnt(4)=1, cnt(3)=2
lfu.get(1);   // return -1 (not found)
lfu.get(3);   // return 3
// cache=[3,4], cnt(4)=1, cnt(3)=3
lfu.get(4);   // return 4
// cache=[3,4], cnt(4)=2, cnt(3)=3

Constraints:

0 <= capacity <= 104
0 <= key <= 105
0 <= value <= 109
At most 2 * 105 calls will be made to get and put.

```
class LFUCache {

    /*
    Approach: Create a class called Node which will be a node of a DoublyLinkedList having key, value, frequency,
        prevNode and nextNode.
        Use a HashMap (keyNodeMap: key -> Node) to handle data access by key.
        Then use a HashMap (freqNodeDLLMap: frequency -> DoublyLinkedList<Node>) to handle frequency.
        Also, maintain a variable (minumumFrequency) which will store the current minimum frequency of keys in cache.
        Thus if we want to add a new key, we just need to find (or create new) the likedlist by its frequency (which is 1),
        add the item to the start of the linked list.
        If cache is full and we need to remove an item, we will get the minimum frequency (minumumFrequency),
        get the appropriate linkedlist from freqNodeDLLMap by it, then remove the last item of that linkedlist.
        Also we'll use the key of that removed item to remove the item from our cache (keyNodeMap).
        If we want to increment the freqneucy of a key, we'll get the node, remove it from its current frequency linked list
        by joining it's prevNode and nextNode (This is why we're using DoublyLinkedList. No need to find a node by traversing
        to remove it. If we have the node, we can just join its previous and next node to remove it.). Then add the node to
        the linkedlist of the new (incremented) frequency.
        Thus, the frequency ranking management will be done in O(1) time.

    Complexity analysis: Time: O(1), Space: O(n).

    */


    int capacity;
    HashMap<Integer, Node> keyNodeMap = new HashMap<>();
    HashMap<Integer, NodeDLinkedList> freqNodeDLLMap = new HashMap<>();
    int minumumFrequency = 1;

    public LFUCache(int capacity) {
        this.capacity = capacity;
    }

    public int get(int key) {
        Node node = keyNodeMap.get(key);
        if(node != null){                                   //Item exists
            incrementFrequency(node);                               //Increment frequency
            return node.value;                                  //Return value
        }
        else{
```

```
            return -1;
        }
    }

    public void put(int key, int value) {

        if(capacity <= 0) return;

        if(keyNodeMap.containsKey(key)){                                    //Item exists

            Node node = keyNodeMap.get(key);                                //Get old node
            node.value = value;                                             //Update with new value
            incrementFrequency(node);                                       //Increment frequency
            keyNodeMap.put(key, node);                                      //Put (update) in cache
        }
        else{                                                       //Item doesn't exist

            Node node = new Node(key, value);                               //Create new node

            if(keyNodeMap.size() == capacity){                              //Cache is full
                Node removedLastNode = freqNodeDLLMap.get(minumumFrequency)
                                    .removeLastNode();                      //Remove LFU node from
removedLastNode
                keyNodeMap.remove(removedLastNode.key);                     //Remove LFU
node from cache
            }

            incrementFrequency(node);                                       //Add to frequency map
            keyNodeMap.put(key, node);                                      //Add to cache

            minumumFrequency = 1;                                           //Since new node is having
freqency as 1,
                                                                //update minumumFrequency to be 1
        }

    }


    private void incrementFrequency(Node node){

        int oldFrequency = node.frequency;

        if(freqNodeDLLMap.containsKey(oldFrequency)){                       //Frequency
already exists
            NodeDLinkedList oldNodeDLinkedList = freqNodeDLLMap.get(oldFrequency);      //Get
frequency linkedlist
            oldNodeDLinkedList.remove(node);                                //Remove current node
            if(node.frequency == minumumFrequency &&                        //If this frequency
was the minumum freq.
                oldNodeDLinkedList.length == 0){                            //and no node is having
this freq anymore
                minumumFrequency++;                                         //Increment minumum
frequency
            }
```

```
        }

        int newFrequency = oldFrequency + 1;                    //Increment frequency
        node.frequency = newFrequency;                          //Set new frequency to
node
        NodeDLinkedList newNodeDLinkedList =                    //Get or create the
LinkedList for
            freqNodeDLLMap.getOrDefault(                        //this new frequency
            newFrequency, new NodeDLinkedList()
        );
        newNodeDLinkedList.add(node);                           //Add node to the freq
linkedlist
        freqNodeDLLMap.put(newFrequency, newNodeDLinkedList);   //Put freq
linkedlist to freqNodeDLLMap
    }


    private class Node{
        int key;
        int value;
        int frequency;
        Node prev;
        Node next;
        Node(int key, int value){
            this.key = key;
            this.value = value;
            this.frequency = frequency;
        }
    }

    private class NodeDLinkedList{

        Node head, tail;
        int length;

        //Add a node to top
        void add(Node node){

            //Remove old pointers
            node.prev = null;
            node.next = null;

            if(head==null){                                     //Empty list
                head = node;
                tail = node;
            }
            else{
                node.next = head;                               //Forward linking
                head.prev = node;                               //Backward linking
                head = node;
            }

            length++;
        }
```

```
        //Remove a node
        void remove(Node node){

            if(node==head){                                     //Need to remove head node
                if(node==tail){                                 //Tail node is the same (list size =
1)
                    tail=null;                                  //Make tail null
                }
                head = head.next;                               //Make head point to the next
node
            }
            else{                                               //Need to remove later node
                node.prev.next = node.next;                     //Forward linking

                if(node==tail){                                 //Need to remove tail node
                    tail = node.prev;                           //Point tail to prev node
                }
                else{
                    node.next.prev = node.prev;                 //Backward linking
                }
            }

            length--;

        }

        //Remove last node
        Node removeLastNode(){

            Node tailNode = tail;

            if(tailNode != null){                               //LastNode exists
                remove(tailNode);
            }
            return tailNode;
        }
    }
}
```

----------Longest Palindromic Subsequence
----------Given a string s, find the longest palindromic subsequence's length in s.

A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

Example 1:

Input: s = "bbbab"
Output: 4

Explanation: One possible longest palindromic subsequence is "bbbb".
Example 2:

Input: s = "cbbd"
Output: 2
Explanation: One possible longest palindromic subsequence is "bb".

Constraints:

1 <= s.length <= 1000
s consists only of lowercase English letters.

```
public int longestPalindromeSubseq(String s) {
    char[] c = s.toCharArray();
    int[] dp = new int[c.length];
    for(int j = 0; j < dp.length; j++) {
        dp[j] = 1;
        int topRight = 0;
        for(int i = j-1; i >= 0; i--) {
            int temp = dp[i];
            dp[i] = c[i] == c[j] ? 2 + topRight : Math.max(dp[i], dp[i+1]);
            topRight = temp;
        }
    }
    return dp[0];
}
```