

```
'''  
    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(object):  
    def twoSum(self, nums, target):  
        mapping = {}  
  
        for index, val in enumerate(nums):  
            diff = target - val  
            if diff in mapping:  
                return [index, mapping[diff]]  
            else:  
                mapping[val] = index  
  
# Space: O(N)  
# Time: O(N)
```

```

from suffix_array import SuffixArray

class LCP(object):
    def __init__(self, s):
        self.s = s
        self.lcp_array = []
        self.suffix_array = SuffixArray(s)
        self.suffix_array.create_suffix_array()

    def lcp_w_suffix_str(self):
        N = len(self.suffix_array.suffix_array)
        array = self.suffix_array.suffix_array

        self.lcp_array = [0]*N
        inv_suffix = [0]*N

        for index in range(N):
            inv_suffix[array[index].index] = index

        maxLen = 0

        for index in range(N):
            if inv_suffix[index] == N-1:
                maxLen = 0
                continue

                index_j = array[inv_suffix[index]+1].index
                while(index+maxLen < N and index_j+maxLen < N and
self.s[index+maxLen] == self.s[index_j+maxLen]):
                    maxLen += 1

            self.lcp_array[inv_suffix[index]] = maxLen

            if maxLen > 0:
                maxLen -= 1

        return self.lcp_array

if __name__ == '__main__':
    lcp = LCP("banana")
    lcp.lcp_w_suffix_str()
    print lcp.lcp_array

```

```

class Suffix(object):
    def __init__(self):
        self.index = 0
        self.first_rank = -1
        self.adjacent_rank = -1

    def __lt__(self, other):
        if self.first_rank == other.first_rank:
            return self.adjacent_rank < other.adjacent_rank
        return self.first_rank < other.first_rank

class SuffixArray(object):
    def __init__(self, s):
        self.s = s
        self.suffix_array = []

    def print_suffix(self):
        for index in range(len(self.s)):
            ele = self.suffix_array[index]
            print("Suffix index {}, Suffix string\n{}".format(ele.index, self.s[ele.index:]))

    def create_suffix_array(self):
        N = len(self.s)

        for index, char in enumerate(self.s):
            suffix_obj = Suffix()
            suffix_obj.index = index
            suffix_obj.first_rank = ord(char)-ord('a')
            suffix_obj.adjacent_rank = ord(self.s[index+1])-ord('a')
if (index+1 < N) else -1
            self.suffix_array.append(suffix_obj)

        self.suffix_array.sort()

        no_char = 4
        index_map = {}
        while no_char < 2*N:
            rank = 0
            prev_rank, self.suffix_array[0].first_rank =
self.suffix_array[0].first_rank, rank
            index_map[self.suffix_array[0].index] = 0

            for index in range(1, N):
                if self.suffix_array[index].first_rank ==
prev_rank and self.suffix_array[index].adjacent_rank ==
self.suffix_array[index-1].adjacent_rank:
                    self.suffix_array[index].first_rank = rank
                else:
                    rank += 1
                    prev_rank,
self.suffix_array[index].first_rank =
self.suffix_array[index].first_rank, rank
                    index_map[self.suffix_array[index].index] = index

            for index in range(N):
                adjacent_index = self.suffix_array[index].index +
(no_char/2)

```

```
        self.suffix_array[index].adjacent_rank =
self.suffix_array[index_map[adjacent_index]] if adjacent_index < N else -
1

        self.suffix_array.sort()
no_char *= 2

if __name__ == '__main__':
    suffix_array = SuffixArray("banana")
    suffix_array.create_suffix_array()
    suffix_array.print_suffix()
```

```
'''
```

Given an array of integers `nums` and an integer `target`, return indices of the two numbers such that they add up to `target`.

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

You can return the answer in any order.

Example 1:

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

Output: `[0,1]`

Explanation: Because `nums[0] + nums[1] == 9`, we return `[0, 1]`.

Example 2:

Input: `nums = [3,2,4]`, `target = 6`

Output: `[1,2]`

Example 3:

Input: `nums = [3,3]`, `target = 6`

Output: `[0,1]`

Constraints:

`2 <= nums.length <= 104`

`-109 <= nums[i] <= 109`

`-109 <= target <= 109`

Only one valid answer exists.

Follow-up: Can you come up with an algorithm that is less than  $O(n^2)$  time complexity?

```
'''
```

```
class Solution():
    # Naive method to find a pair in a list with the given sum
    def twoSum(self, nums, target):
        # consider each element except the last
        for i in range(len(nums) - 1):
            # start from the i'th element until the last element
            for j in range(i + 1, len(nums)):
                # if the desired sum is found, print it
                if nums[i] + nums[j] == target:
                    print('Pair found', (nums[i], nums[j]))
                    return [i,j]

        # No pair with the given sum exists in the list
        print('Pair not found')

if __name__ == '__main__':
    nums = [2,7,11,15]
    target = 9
```

```
ret = Solution().twoSum(nums,target)
```

'''

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

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

Input: l1 = [2,4,3], l2 = [5,6,4]

Output: [7,0,8]

Explanation: 342 + 465 = 807.

Example 2:

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

Output: [0]

Example 3:

Input: l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]

Output: [8,9,9,9,0,0,0,1]

Constraints:

The number of nodes in each linked list is in the range [1, 100].

0 <= Node.val <= 9

It is guaranteed that the list represents a number that does not have leading zeros.

'''

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
        # Head of the new linked list - this is the head of the resultant
        list
        head = None
        # Reference of head which is null at this point
        temp = None
        # Carry
        carry = 0
        # Loop for the two lists
        while l1 is not None or l2 is not None:
            # At the start of each iteration, we should add carry from
            the last iteration
            sum_value = carry
            # Since the lengths of the lists may be unequal, we are
            checking if the
            # current node is null for one of the lists
            if l1 is not None:
                sum_value += l1.val
                l1 = l1.next
```

```

        if l2 is not None:
            sum_value += l2.val
            l2 = l2.next
    # At this point, we will add the total sum_value % 10 to the
new node
        # in the resultant list
    node = ListNode(sum_value % 10)
    # Carry to be added in the next iteration
    carry = sum_value // 10
    # If this is the first node or head
    if temp is None:
        temp = head = node
    # for any other node
    else:
        temp.next = node
        temp = temp.next
    # After the last iteration, we will check if there is carry left
    # If it's left then we will create a new node and add it
    if carry > 0:
        temp.next = ListNode(carry)
    return head

if __name__ == '__main__':
    l1 = ListNode(2, ListNode(4, ListNode(3)))
    l2 = ListNode(5, ListNode(6, ListNode(4)))
    target = ListNode(7, ListNode(0, ListNode(8)))
    ret = Solution().addTwoNumbers(l1,l2)

```

```
'''  
    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 "bbbbbb", 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(object):  
    def lengthOfLongestSubstring(self, s):  
        """  
        :type s: str  
        :rtype: int  
        """  
        mapSet = {}  
        start, result = 0, 0  
  
        for end in range(len(s)):  
            if s[end] in mapSet:  
                start = max(mapSet[s[end]], start)  
            result = max(result, end-start+1)  
            mapSet[s[end]] = end+1  
  
        return result
```

'''

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(object):
    def findMedianSortedArrays(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :rtype: float
        """

        if len(nums1) > len(nums2):
            nums1, nums2 = nums2, nums1

        x, y = len(nums1), len(nums2)
        low , high = 0, x

        while  low <= high:
            partitionx = (low+high)/2
            partitiony = (x+y+1)/2 - partitionx
            if partitionx == 0:
                maxLeftX = float('-inf')
            else:
                maxLeftX = nums1[partitionx-1]

            if partitionx == x:
                minRightX = float('inf')
            else:
                minRightX = nums1[partitionx]

            if partitiony == 0:
                maxLeftY = float('-inf')
            else:
                maxLeftY = nums2[partitiony-1]

            if partitiony == y:
                minRightY = float('inf')
            else:
                minRightY = nums2[partitiony]

            if maxLeftX <= minRightY and maxLeftY <= minRightX:
                if((x+y)%2 == 0):
```

```
        return (max(maxLeftX, maxLeftY) + min(minRightX,
minRightY))/2.0
    else:
        return max(maxLeftY, maxLeftX)
elif maxLeftX > minRightY:
    high = partitionx - 1
else:
    low = partitionx + 1

print Solution().findMedianSortedArrays([1,2], [3, 4])
```

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

```
Example 1:
```

```
Input: "babad"  
Output: "bab"  
Note: "aba" is also a valid answer.  
Example 2:
```

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

```
'''  
  
class Solution(object):  
    def longestPalindrome(self, s):  
        """  
        :type s: str  
        :rtype: str  
        """  
        dp = [[0 for _ in range(len(s))] for _ in range(len(s))]  
        maxLength, result = 1, ""  
        for index in range(len(s)):  
            dp[index][index] = 1  
            result = s[index]  
  
        length = 2  
  
        while length <= len(s):  
            index_i = 0  
            while index_i < len(s) - length + 1:  
                index_j = index_i + length - 1  
  
                if length == 2 and s[index_i] == s[index_j]:  
                    dp[index_i][index_j] = 1  
                    maxLength = max(maxLength, 2)  
                    result = s[index_i:index_j+1]  
                elif s[index_i] == s[index_j] and dp[index_i+1][index_j-1]:  
                    dp[index_i][index_j] = 1  
                    if length > maxLength:  
                        maxLength = length  
                        result = s[index_i:index_j+1]  
  
                index_i += 1  
            length += 1  
  
        return result  
  
# Space: O(N^2)  
# Time: O(N^2)  
  
class Solution(object):  
    def longestPalindrome(self, s):  
        """  
        :type s: str  
        :rtype: str
```

```
"""

def expand(s, left, right):
    while left >= 0 and right < len(s) and s[left] == s[right]:
        left -= 1
        right += 1
    return right-left-1

start, end = 0, 0
for index in range(len(s)):
    even_len = expand(s, index, index+1)
    odd_len = expand(s, index, index)
    length = max(even_len, odd_len)
    if length > (end-start):
        start = index - (length-1)/2
        end = index +length/2

return s[start:end+1]
```

```

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

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

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

class Solution(object):
    def convert(self, s, numRows):
        """
        :type s: str
        :type numRows: int
        :rtype: str
        """

        if numRows == 1:
            return s

        result = ["" for _ in range(numRows)]
        row, down = 0, 1
        for char in s:
            result[row] += char

            if row == numRows - 1:
                down = 0
            if row == 0:
                down = 1

            if down:
                row += 1
            else:
                row -= 1
        final_string = ""
        for value in result:
            final_string += value
        return final_string

print Solution().convert("PAYPALISHIRING", 3)

```

```
'''  
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).

Example 1:

Input: x = 123

Output: 321

Example 2:

Input: x = -123

Output: -321

Example 3:

Input: x = 120

Output: 21

Constraints:

-2<sup>31</sup> <= x <= 2<sup>31</sup> - 1

```
'''
```

```
class Solution:  
    def reverse(self, x: int) -> int:  
        y = str(abs(x))  
        y = y.strip()  
        y = y[::-1]  
        output = int(y)  
        if output >= 2** 31 - 1 or output <= -2** 31:  
            return 0  
        elif x < 0:  
            return -1 * output  
        else:  
            return output
```

```
class Solution:
    def myAtoi(self, str):
        """
        :type str: str
        :rtype: int
        """
        str = str.strip()
        number = ""

        for x in str:
            if x.isalpha() and number == "":
                return 0
            elif x.isalpha():
                break
            elif x == ".":
                break
            elif x == " ":
                break
            elif (x == "+" or x == "-") and number == "":
                number = number + x
            elif (x == "+" or x == "-") and number != "":
                break
            elif (x == "+" or x == "-") and (number[-1] == "+" or
number[-1] == "-")):
                return 0
            elif (x == "+" or x == "-") and ("+" in number or "-" in
number):
                break
            elif x.isdigit():
                number = number + x
        if number == "" or number == "+" or number == "-":
            return 0
        else:
            if int(number) > ((2**31)-1):
                return (2**31)-1
            elif int(number) < -(2**31):
                return -(2**31)
            else:
                return int(number)
```

```
'''
```

## 9. Palindrome Number

Given an integer  $x$ , return true if  $x$  is a palindrome, and false otherwise.

Example 1:

Input:  $x = 121$

Output: true

Explanation: 121 reads as 121 from left to right and from right to left.

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.

Constraints:

$-2^{31} \leq x \leq 2^{31} - 1$

```
'''
```

```
class Solution:
    def isPalindrome(self, x: int) -> bool:
        # Base condition
        if x < 0:
            return False
        # Store the number in a variable
        number = x
        # This will store the reverse of the number
        reverse = 0
        while number:
            reverse = reverse * 10 + number % 10
            number //= 10
        return x == reverse
```

```

"""
Given an input string (s) and a pattern (p), 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).

Note:

s could be empty and contains only lowercase letters a-z.
p could be empty and contains only lowercase letters a-z, and
characters like . or *.
"""

class Solution(object):
    def isMatch(self, s, p):
        """
        :type s: str
        :type p: str
        :rtype: bool
        """
        dp = [[False for _ in range(len(p) + 1)] for _ in range(len(s) + 1)]
        dp[0][0] = True

        for index in range(1, len(dp[0])):
            if p[index-1] == '*':
                dp[0][index] = dp[0][index - 2]

        for index_i in range(1, len(dp)):
            for index_j in range(1, len(dp[0])):
                if s[index_i - 1] == p[index_j - 1] or p[index_j - 1] ==
'.':
                    dp[index_i][index_j] = dp[index_i-1][index_j-1]
                elif p[index_j-1] == '*':
                    dp[index_i][index_j] = dp[index_i][index_j-2]

                    if s[index_i-1] == p[index_j-2] or p[index_j-2] ==
'.':
                        dp[index_i][index_j] = dp[index_i-1][index_j] or dp[index_i][index_j]

                else:
                    dp[index_i][index_j] = False

        return dp[len(s)][len(p)]

```

```
'''  
    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.  
'''
```

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

```
'''
```

```
class Solution(object):  
    def maxArea(self, height):  
        """  
        :type height: List[int]  
        :rtype: int  
        """  
        left, right, maxArea = 0, len(height) - 1, 0  
  
        while left < right:  
            maxArea = max(maxArea, min(height[left],  
height[right])*(right-left))  
            if height[left] < height[right]:  
                left += 1  
            else:  
                right -= 1  
  
        return maxArea
```

```
# Space : O(1)  
# Time: O(N)
```

'''

Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.

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"

Explanation: 3 is represented as 3 ones.

Example 2:

Input: num = 58

Output: "LVIII"

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

Example 3:

Input: num = 1994

Output: "MCMXCIV"

Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.

Constraints:

1 <= num <= 3999

'''

```
class Solution:
    def intToRoman(self, num: int) -> str:
        M = ["", "M", "MM", "MMM"]
        C = ["", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCC", "CM"]
        X = ["", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"]
```

```
I = ["", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"]
    return M[num // 1000] + C[(num % 1000) // 100] + X[(num % 100) //
10] + I[num % 10]
```

'''

### 13. Roman to Integer

Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.

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 ones 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

Explanation: III = 3.

Example 2:

Input: s = "LVIII"

Output: 58

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

Example 3:

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:

```
def romanToInt(self, s: str) -> int:  
    # Dictionary of roman numerals
```

```
roman_map = {'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C': 100, 'D':  
500, 'M': 1000}  
    # Length of the given string  
n = len(s)  
    # This variable will store result  
num = roman_map[s[n - 1]]  
    # Loop for each character from right to left  
for i in range(n - 2, -1, -1):  
    # Check if the character at right of current character is  
bigger or smaller  
    if roman_map[s[i]] >= roman_map[s[i + 1]]:  
        num += roman_map[s[i]]  
    else:  
        num -= roman_map[s[i]]  
return num
```

```
'''
```

```
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: ["flower","flow","flight"]
```

```
Output: "fl"
```

```
Example 2:
```

```
Input: ["dog","racecar","car"]
```

```
Output: ""
```

```
Explanation: There is no common prefix among the input strings.
```

```
Note:
```

```
All given inputs are in lowercase letters a-z.
```

```
'''
```

```
class Solution(object):
    def longestCommonPrefix(self, strs):
        """
        :type strs: List[str]
        :rtype: str
        """
        def prefix(strs, index):
            check_prefix = strs[0][:index]
            for index in range(1, len(strs)):
                if not strs[index].startswith(check_prefix):
                    return False
            return True

        if not strs:
            return ""

        minLength = float('inf')
        for string in strs:
            minLength = min(minLength, len(string))

        low, high = 0, minLength

        while low <= high:
            mid = (low+high)/2
            if (prefix(strs, mid)):
                low = mid + 1
            else:
                high = mid - 1

        return strs[0][: (low+high)/2]
```

```

"""
    Given an array nums of n integers, are there elements a, b, c in
    nums 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.

Example:
Given array nums = [-1, 0, 1, 2, -1, -4],
A solution set is:
[
    [-1, 0, 1],
    [-1, -1, 2]
]
"""

class Solution(object):
    def threeSum(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        nums.sort()

        if (len(nums) >= 3) and (nums[0] == nums[len(nums) - 1]) and
        (nums[0] == 0):
            return [[0, 0, 0]]

        result = []
        for index in range(len(nums) - 1):
            left = index+1
            right = len(nums) - 1

            while left < right:
                currSum = nums[index] + nums[left] + nums[right]
                if currSum == 0:
                    result.append([nums[index], nums[left],
numbs[right]])
                    left += 1
                    right -= 1
                elif currSum < 0:
                    left += 1
                else:
                    right -= 1
        return [list(t) for t in set(tuple(element) for element in
result)]


# Space: O(1)
# Time: O(N^2)

```

```
'''  
    Given an array nums of n integers and an integer target, find three  
    integers in nums such that the sum is closest to target. Return the sum  
    of the three integers. You may assume that each input would have exactly  
    one solution.
```

Example:

Given array nums = [-1, 2, 1, -4], and target = 1.

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

```
'''  
  
class Solution(object):  
    def threeSumClosest(self, nums, target):  
        """  
        :type nums: List[int]  
        :type target: int  
        :rtype: int  
        """  
  
        nums.sort()  
        result, min_diff = 0, float('inf')  
  
        for index in range(len(nums)-1):  
            left = index + 1  
            right = len(nums) - 1  
  
            while left < right:  
                currSum = nums[index] + nums[left] + nums[right]  
                diff = abs(target - currSum)  
  
                if diff == 0:  
                    return target  
                if diff < min_diff:  
                    min_diff = diff  
                    result = currSum  
  
                if currSum < target:  
                    left += 1  
                else:  
                    right -= 1  
        return result  
  
# Space: O(1)  
# Time: O(N^2)
```

```

"""
    Given a string containing digits from 2-9 inclusive, 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. Note that 1 does not map to any letters.
"""

class Solution(object):
    def letterCombinations(self, digits):
        """
        :type digits: str
        :rtype: List[str]
        """

        phoneMap = { '2': 'abc', '3': 'def', '4': 'ghi', '5': 'jkl', '6':
'mno', '7' : 'pqrs', '8': 'tuv', '9':'wxyz' }
        number = str(digits)

        if number == "":
            return []

        result = ['']
        for char in number:
            values = phoneMap[char]
            new_result = []
            for prefix in result:
                currElement = prefix
                for value in values:
                    new_result.append(currElement+value)

            result = new_result
            # result = [prefix+value for prefix in result for value in
values]
        return result

print Solution().letterCombinations("23")

```

```
'''  
    Given an array nums of n integers and an integer target, are there  
elements a, b, c, and d in nums 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.

Example:

Given array nums = [1, 0, -1, 0, -2, 2], and target = 0.

A solution set is:

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

```
'''
```

```
class Solution(object):  
    def fourSum(self, nums, target):  
        """  
        :type nums: List[int]  
        :type target: int  
        :rtype: List[List[int]]  
        """  
  
        sumMapping = {}  
        for index_i in range(len(nums)-1):  
            for index_j in range(index_i+1, len(nums)):  
                currSum = nums[index_i] + nums[index_j]  
                if currSum in sumMapping:  
                    sumMapping[currSum].append((index_i, index_j))  
                else:  
                    sumMapping[currSum] = [(index_i, index_j)]  
  
        result = set()  
        for key, value in sumMapping.iteritems():  
            diff = target - key  
            if diff in sumMapping:  
                firstSet = value  
                secondSet = sumMapping[diff]  
  
                for (i, j) in firstSet:  
                    for (k, l) in secondSet:  
                        fourlet = [i, j, k, l]  
                        if len(set(fourlet)) != len(fourlet):  
                            continue  
                        fourlist = [nums[i], nums[j], nums[k],  
                                   nums[l]]  
                        fourlist.sort()  
                        result.add(tuple(fourlist))  
  
        return list(result)
```

```
# Space : O(N)
# Time: O(N^3)
```

```
'''  
    Given a linked list, remove the n-th node from the end of list and  
    return its head.
```

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.

```
'''
```

```
# Definition for singly-linked list.  
# class ListNode(object):  
#     def __init__(self, x):  
#         self.val = x  
#         self.next = None  
  
class Solution(object):  
    def removeNthFromEnd(self, head, n):  
        """  
        :type head: ListNode  
        :type n: int  
        :rtype: ListNode  
        """  
        if not head:  
            return None  
  
        ref = head  
        while n > 0:  
            ref = ref.next  
            n -= 1  
  
        if ref is None:  
            return head.next  
        else:  
            main = head  
            while ref.next:  
                main = main.next  
                ref = ref.next  
  
            main.next = main.next.next  
        return head
```

```
'''  
20. Valid Parentheses  
Given a string s containing just the characters '(', ')', '{', '}', '[  
and ']', determine if the input string is valid.
```

An input string is valid if:

Open brackets must be closed by the same type of brackets.  
Open brackets must be closed in the correct order.  
Every close bracket has a corresponding open bracket of the same type.

Example 1:

```
Input: s = "()"  
Output: true  
Example 2:
```

```
Input: s = "()[]{}"  
Output: true  
Example 3:
```

```
Input: s = "()"  
Output: false
```

Constraints:

```
1 <= s.length <= 104  
s consists of parentheses only '()[]{}'.
```

```
'''
```

```
class Solution:  
    def isValid(self, s: str) -> bool:  
        # Stack for left symbols  
        leftSymbols = []  
        # Loop for each character of the string  
        for c in s:  
            # If left symbol is encountered  
            if c in ['(', '{', '[']:  
                leftSymbols.append(c)  
            # If right symbol is encountered  
            elif c == ')' and len(leftSymbols) != 0 and leftSymbols[-1]  
== '(':  
                leftSymbols.pop()  
            elif c == '}' and len(leftSymbols) != 0 and leftSymbols[-1]  
== '{':  
                leftSymbols.pop()  
            elif c == ']' and len(leftSymbols) != 0 and leftSymbols[-1]  
== '[':  
                leftSymbols.pop()  
            # If none of the valid symbols is encountered  
            else:  
                return False  
        return leftSymbols == []
```



'''

## 21. Merge Two Sorted Lists

You are given the heads of two sorted linked lists list1 and list2.

Merge the two lists in a one sorted list. The list should be made by splicing together the nodes of the first two lists.

Return the head of the merged linked list.

Example 1:

Input: list1 = [1,2,4], list2 = [1,3,4]

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

Example 2:

Input: list1 = [], list2 = []

Output: []

Example 3:

Input: list1 = [], list2 = [0]

Output: [0]

Constraints:

The number of nodes in both lists is in the range [0, 50].

-100 <= Node.val <= 100

Both list1 and list2 are sorted in non-decreasing order.

'''

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def mergeTwoLists(self, list1: Optional[ListNode], list2: Optional[ListNode]) -> Optional[ListNode]:
        # Check if either of the lists is null
        if list1 is None:
            return list2
        if list2 is None:
            return list1
        # Choose head which is smaller of the two lists
        if list1.val < list2.val:
            temp = head = ListNode(list1.val)
            list1 = list1.next
        else:
            temp = head = ListNode(list2.val)
            list2 = list2.next
        # Loop until any of the list becomes null
        while list1 is not None and list2 is not None:
            if list1.val < list2.val:
                temp.next = ListNode(list1.val)
                list1 = list1.next
            else:
                temp.next = ListNode(list2.val)
                list2 = list2.next
```

```
        temp = temp.next
# Add all the nodes in l1, if remaining
while list1 is not None:
    temp.next = ListNode(list1.val)
    list1 = list1.next
    temp = temp.next
# Add all the nodes in l2, if remaining
while list2 is not None:
    temp.next = ListNode(list2.val)
    list2 = list2.next
    temp = temp.next
return head
```

```
'''  
    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(object):  
    def generateParenthesis(self, n):  
        """  
        :type n: int  
        :rtype: List[str]  
        """  
  
        result = []  
  
        def backtracking(S, left, right):  
            if len(S) == 2*n:  
                result.append(S)  
                return  
  
            if left < n:  
                backtracking(S+'(', left+1, right)  
  
            if right < left:  
                backtracking(S+')', left, right+1)  
  
        backtracking('', 0, 0)  
        return result
```

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

Example:

```
Input: [ 1->4->5, 1->3->4, 2->6 ] Output: 1-
```

```

class Solution(object):
    def mergeKLists(self, lists):
        """
        :type lists: List[ListNode]
        :rtype: ListNode
        """

        from heapq import heappush, heappop

        heap = []
        head = point = ListNode(0)
        for element in lists:
            if element:
                heapq.heappush(heap, (element.val, element))

        while heap:
            value, node = heapq.heappop(heap)
            head.next = ListNode(value)
            head = head.next
            node = node.next
            if node:
                heapq.heappush(heap, (node.val, node))

        return point.next

# Space: O(K)
# Time: O(N*log(K))

class Solution(object):
    def mergeKLists(self, lists):
        """
        :type lists: List[ListNode]
        :rtype: ListNode
        """

        def merge2Lists(l1, l2):
            head = point = ListNode(0)
            while l1 and l2:
                if l1.val <= l2.val:
                    point.next = ListNode(l1.val)
                    l1 = l1.next
                else:
                    point.next = ListNode(l2.val)
                    l2 = l2.next
            point.next = l1 or l2
            return head.next

        if len(lists) == 1:
            return lists[0]
        if len(lists) == 0:
            return None
        if len(lists) == 2:
            return self.merge2Lists(lists[0], lists[1])
        else:
            mid = len(lists) // 2
            left = self.mergeKLists(lists[:mid])
            right = self.mergeKLists(lists[mid:])
            return self.merge2Lists(left, right)

```

```
        point.next = ListNode(l2.val)
        l2 = l2.next
        point = point.next

    if l1:
        point.next = l1
    else:
        point.next = l2
    return head.next

if not lists:
    return lists

interval = 1
while interval < len(lists):
    for index in range(0, len(lists) - interval ,interval*2):
        lists[index] = merge2Lists(lists[index],
list[index+interval])

    interval *= 2

return lists[0]

# Time: O(N*log(k))
# Space: O(1)
```

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

Example:

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

```
# Definition for singly-linked list.  
# class ListNode(object):  
#     def __init__(self, x):  
#         self.val = x  
#         self.next = None  
  
class Solution(object):  
    def swapPairs(self, head):  
        """  
        :type head: ListNode  
        :rtype: ListNode  
        """  
  
        if head is None:  
            return head  
  
        ref = head  
  
        while ref is not None and ref.next is not None:  
            ref.val, ref.next.val = ref.next.val, ref.val  
            ref = ref.next.next  
  
        return head
```

```

"""
    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.

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.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def reverseKGroup(self, head, k):
        if head:
            slow = head # the mover
            while slow:
                group = []
                while slow and len(group) < k:
                    group.append(slow)
                    slow = slow.next
                if not slow and len(group) < k:
                    return head
                for i in range(k/2):
                    print i,k-i-1
                    group[i].val,group[k-i-1].val = group[k-i-1].val,group[i].val
            return head

# Space: O(k)
# Time: O(N)

```

'''

Given a sorted array `nums`, 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 by modifying the input array in-place with O(1) extra memory.

Example 1:

Given `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 returned length.

'''

```
class Solution(object):
    def removeDuplicates(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        if not nums:
            return 0

        index_i = 0

        for index_j in range(1, len(nums)):
            if nums[index_i] != nums[index_j]:
                index_i += 1
                nums[index_i] = nums[index_j]

        return index_i + 1
```

'''

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

Example 1:

Input:

```
s = "barfoothefoobarman",
words = ["foo", "bar"]
```

Output: [0,9]

Explanation: Substrings starting at index 0 and 9 are "barfoor" and "foobar" respectively.

The output order does not matter, returning [9,0] is fine too.

Example 2:

Input:

```
s = "wordgoodstudentgoodword",
words = ["word", "student"]
```

Output: []

'''

```
class Solution(object):
    def findSubstring(self, s, words):
        """
        :type s: str
        :type words: List[str]
        :rtype: List[int]
        """
        if not str or not words:
            return []

        counts = {}
        for word in words:
            if word in counts:
                counts[word] += 1
            else:
                counts[word] = 1

        result = []
        n, numWords, fixLen = len(s), len(words), len(words[0])

        for index in range(0, n-numWords*fixLen+1):
            seen = {}

            index_j = 0
            while index_j < numWords:
                word = s[index + index_j*fixLen: index +
(index_j+1)*fixLen]
                if word in counts:
                    if word in seen:
                        seen[word] += 1
                    else:
                        seen[word] = 1

                    if seen[word] > counts[word]:
                        break
                index_j += 1
            if len(seen) == numWords:
                result.append(index)
```

```
        else:
            break
        index_j += 1

    if index_j == numWords:
        result.append(index)

return

# Time: O(N^2)
# Space: O(N)
```

```

"""
    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 and use only constant extra
memory.

    Here are some examples. Inputs are in the left-hand column and its
corresponding outputs are in the right-hand column.

    1,2,3 → 1,3,2
    3,2,1 → 1,2,3
    1,1,5 → 1,5,1
"""

class Solution(object):
    def nextPermutation(self, nums):
        """
        :type nums: List[int]
        :rtype: void Do not return anything, modify nums in-place
instead.
        """

        index_i = len(nums) - 2
        while index_i >= 0 and nums[index_i] >= nums[index_i+1]:
            index_i -= 1

        if index_i >= 0:
            index_j = len(nums) - 1
            while index_j >= index_i and nums[index_j] <= nums[index_i]:
                index_j -= 1

            nums[index_i], nums[index_j] = nums[index_j], nums[index_i]

        start = index_i + 1
        end = len(nums) - 1

        while start < end:
            nums[start], nums[end] = nums[end], nums[start]
            start += 1
            end -= 1

```

```
class Solution(object):
    def longestValidParentheses(self, s):
        """
        :type s: str
        :rtype: int
        """
        stack, result = [-1], 0

        for index in range(len(s)):
            if s[index] == '(':
                stack.append(index)
            else:
                currIndex = stack.pop()
                if currIndex == -1:
                    stack.append(index)
                else:
                    result = max(result, index-currIndex+1)
        return result
```

```
"""
    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.
```

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

class Solution(object):
    def search(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: int
        """
        if not nums:
            return -1
```

```
        left, right = 0, len(nums) - 1
```

```
        while left <= right:
```

```
            mid = (left + right) / 2
```

```
            if nums[mid] == target:
                return mid
```

```
            if nums[left] <= nums[mid]:
```

```
                if target >= nums[left] and target <= nums[mid]:
```

```
                    right = mid - 1
```

```
                else:
```

```
                    left = mid + 1
```

```
            else:
```

```
                if target >= nums[mid] and target <= nums[right]:
```

```
                    left = mid + 1
```

```
                else:
```

```
                    right = mid - 1
```

```
        return -1
```

```
class Solution(object):
    def search(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: int
        """

        def searchRecursive(nums, left, right, target):
```

```
            if left > right:
                return -1
```

```
mid = (left + right) / 2
if nums[mid] == target:
    return mid

if nums[left] <= nums[mid]:
    if nums[left] <= target < nums[mid]:
        return searchRecursive(nums, left, mid-1, target)
    else:
        return searchRecursive(nums, mid+1, right, target)
else:
    if nums[mid] < target <= nums[right]:
        return searchRecursive(nums, mid+1, right, target)
    else:
        return searchRecursive(nums, left, mid-1, target)

return searchRecursive(nums, 0, len(nums)-1, target)
```

```

"""
Given an array of integers nums 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]."

Example 1:

Input: nums = [5,7,7,8,8,10], target = 8
Output: [3,4]
"""

class Solution(object):
    def searchRange(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[int]
        """
        left, right = 0, len(nums)-1

        while left <= right:
            mid = (left + right) / 2
            if target > nums[mid]:
                left = mid + 1
            else:
                right = mid

        if left == len(nums) or nums[left] != target:
            return [-1, -1]

        result = [left]
        left, right = 0, len(nums) - 1
        while left <= right:
            mid = (left + right) / 2
            if nums[mid] > target:
                right = mid
            else:
                left = mid + 1

        result.append(left + 1)
        return result

```

'''

Determine if a 9x9 Sudoku board is valid. Only the filled cells need to be validated according to the following rules:

Each row must contain the digits 1-9 without repetition.

Each column must contain the digits 1-9 without repetition.

Each of the 9 3x3 sub-boxes of the grid must contain the digits 1-9 without repetition.

A partially filled sudoku which is valid.

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

Example 1:

Input:

```
[  
    ["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:

```
[  
    ["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.

Note:

A Sudoku board (partially filled) could be valid but is not necessarily solvable.

Only the filled cells need to be validated according to the mentioned rules.

The given board contain only digits 1-9 and the character '.'.

The given board size is always 9x9.

'''

```
class Solution(object):
    def isValidSudoku(self, board):
        """
        :type board: List[List[str]]
        :rtype: bool
        """
        import collections
        dict_row, dict_col, dict_cell = collections.defaultdict(set),
        collections.defaultdict(set), collections.defaultdict(set)

        for row_index in range(1, 4):
            for col_index in range(1, 4):
                for row in range(3*(row_index-1), 3*row_index):
                    for col in range(3*(col_index-1), 3*col_index):
                        cell_data = board[row][col]
                        if cell_data == '.':
                            continue
                        if cell_data in dict_row[row] or cell_data in
dict_col[col] or cell_data in dict_cell[(row_index, col_index)]:
                            return False

                        dict_row[row].add(cell_data)
                        dict_col[col].add(cell_data)
                        dict_cell[(row_index, col_index)].add(cell_data)

        return True
```

```
'''
```

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

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

```
1 is read off as "one 1" or 11.  
11 is read off as "two 1s" or 21.  
21 is read off as "one 2, then one 1" or 1211.
```

```
Given an integer n, generate the nth term of the count-and-say  
sequence.
```

```
'''
```

```
class Solution(object):  
    def countAndSay(self, n):  
        """  
        :type n: int  
        :rtype: str  
        """  
  
        if n == 1:  
            return "1"  
        new_num = ""  
        count_iter = 1  
        num = "1"  
  
        while count_iter < n:  
            index_i, index_j = 0, 0  
            count, new_num = 0, ""  
  
            while index_j < len(num):  
                if num[index_i] != num[index_j]:  
                    new_num += str(count) + str(num[index_i])  
                    count = 0  
                    index_i = index_j  
                else:  
                    count += 1  
                    index_j += 1  
  
            if count > 0:  
                new_num += str(count) + str(num[index_i])  
            num = new_num  
            count_iter += 1  
  
        return new_num  
  
# Space: O(1)  
# Time: O(N*k) k= length of string
```

```

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

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

Note:

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

class Solution(object):
    def combinationSum(self, candidates, target):
        """
        :type candidates: List[int]
        :type target: int
        :rtype: List[List[int]]
        """

        result = []

        def recursive(candidates, target, currList, index):
            if target < 0:
                return
            if target == 0:
                result.append(currList)
                return

            for start in range(index, len(candidates)):
                recursive(candidates, target - candidates[start],
                          currList + [candidates[start]], start)

        recursive(candidates, target, [], 0)
        return result

```

```

"""
    Given a collection of candidate numbers (candidates) and a target
    number (target), find all unique combinations in candidates where the
    candidate numbers sums to target.

    Each number in candidates 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.

    Example 1:

    Input: candidates = [10,1,2,7,6,1,5], target = 8,
    A solution set is:
    [
        [1, 7],
        [1, 2, 5],
        [2, 6],
        [1, 1, 6]
    ]
"""

class Solution(object):
    def combinationSum2(self, candidates, target):
        """
        :type candidates: List[int]
        :type target: int
        :rtype: List[List[int]]
        """
        result = []
        candidates.sort()

        def recursive(candidates, target, currList, index):
            if target < 0:
                return
            if target == 0:
                result.append(currList)
                return

            previous = -1
            for start in range(index, len(candidates)):
                if previous != candidates[start]:
                    recursive(candidates, target - candidates[start],
                              currList + [candidates[start]], start+1)
                    previous = candidates[start]

        recursive(candidates, target, [], 0)
        return result

```

```

"""
    Given an unsorted integer array, find the smallest missing positive
integer.

Example 1:

Input: [1,2,0]
Output: 3

Example 2:

Input: [3,4,-1,1]
Output: 2

Example 3:

Input: [7,8,9,11,12]
Output: 1

"""

class Solution(object):
    def firstMissingPositive(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """

        index_i = 0
        for index_j in range(len(nums)):
            if nums[index_j] <= 0:
                nums[index_i], nums[index_j] = nums[index_j],
                nums[index_i]
                index_i += 1

            for index in range(index_i, len(nums)):
                if abs(nums[index]) - 1 < len(nums) and nums[abs(nums[index]) - 1] > 0:
                    nums[abs(nums[index]) - 1] = -nums[abs(nums[index]) - 1]

        for index in range(len(nums)):
            if nums[index] > 0:
                return index + 1
        return len(nums) + 1

```

```
"""
    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.
"""

class Solution(object):
    def trap(self, height):
        """
        :type height: List[int]
        :rtype: int
        """

        if not height:
            return 0

        left, right = 0, len(height) - 1
        leftMax, rightMax = 0, 0
        result = 0
        while left < right:
            if height[left] < height[right]:
                if height[left] > leftMax:
                    leftMax = height[left]
                else:
                    result += (leftMax - height[left])
                left += 1
            else:
                if height[right] > rightMax:
                    rightMax = height[right]
                else:
                    result += (rightMax - height[right])
                right -= 1

        return result
```

```

"""
    Given an input string (s) and a pattern (p), 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).

Note:

    s could be empty and contains only lowercase letters a-z.
    p could be empty and contains only lowercase letters a-z, and
characters like ? or *.
"""

class Solution(object):
    def isMatch(self, s, p):
        """
        :type s: str
        :type p: str
        :rtype: bool
        """

        if len(p) == 0:
            return len(s) == 0

        dp = [[False for _ in range(len(p) + 1)] for _ in range(len(s) + 1)]
        dp[0][0] = True

        for index in range(1, len(dp[0])):
            if p[index - 1] == '*':
                dp[0][index] = dp[0][index - 1]

        for index_i in range(1, len(dp)):
            for index_j in range(1, len(dp[0])):
                if s[index_i - 1] == p[index_j - 1] or p[index_j - 1] == '?':
                    dp[index_i][index_j] = dp[index_i - 1][index_j - 1]
                elif p[index_j - 1] == '*':
                    dp[index_i][index_j] = dp[index_i][index_j - 1] or
dp[index_i - 1][index_j]
                else:
                    dp[index_i][index_j] = False
        return dp[len(s)][len(p)]

```

```
"""
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:

```
Input: [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.
```

```
"""
class Solution(object):
    def jump(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        steps, lastReach, maxReach = 0, 0 ,0

        for index in range(len(nums)):
            if index > lastReach:
                lastReach = maxReach
                steps += 1
            maxReach = max(maxReach, index + nums[index])

        return steps if maxReach == len(nums) - 1 else 0
```

```
class Solution(object):
    def permute(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """

        if len(nums) == 0:
            return []
        if len(nums) == 1:
            return [nums]

        result = []
        for index in range(len(nums)):
            for p in self.permute(nums[0:index] + nums[index+1:]):
                result.append([nums[index]] + p)

        return result
```

```
"""
You are given an n x n 2D matrix representing an image.

Rotate the image by 90 degrees (clockwise).
"""

class Solution(object):
    def rotate(self, matrix):
        """
        :type matrix: List[List[int]]
        :rtype: void Do not return anything, modify matrix in-place
instead.

        """
        n = len(matrix)
        if n%2 == 0:
            m = n/2
        else:
            m = n/2 + 1

        for i in range(n/2):
            for j in range(m):
                temp = matrix[i][j]
                matrix[i][j] = matrix[n-j-1][i]
                matrix[n-j-1][i] = matrix[n-i-1][n-j-1]
                matrix[n-i-1][n-j-1] = matrix[j][n-i-1]
                matrix[j][n-i-1] = temp
```

```
"""
    Given an integer array nums, find the contiguous subarray
(containing at least one number) which has the largest sum and return its
sum.
```

Example:

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

```
""
```

```
class Solution(object):
    def maxSubArray(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        if not nums:
            return 0

        currSum, result = nums[0], nums[0]

        for index in range(1, len(nums)):
            currSum = max(nums[index], currSum+nums[index])
            result = max(result, currSum)

        return result
```

```

"""
    Given a matrix of m x n elements (m rows, n columns), return all
elements of the matrix in spiral order.

Example 1:

Input:
[
  [ 1, 2, 3 ],
  [ 4, 5, 6 ],
  [ 7, 8, 9 ]
]
Output: [1,2,3,6,9,8,7,4,5]
"""

class Solution(object):
    def spiralOrder(self, matrix):
        """
        :type matrix: List[List[int]]
        :rtype: List[int]
        """
        if not matrix:
            return []
        R, C = len(matrix), len(matrix[0])
        dr = [0, 1, 0, -1]
        dc = [1, 0, -1, 0]

        result = []
        seen = [[False]*C for _ in range(R)]
        row = 0
        col = 0
        di = 0
        for _ in range(R*C):
            result.append(matrix[row][col])
            seen[row][col] = True
            rr, cc = row + dr[di], col + dc[di]
            if 0 <= rr < R and 0 <= cc < C and not seen[rr][cc]:
                row, col = rr, cc
            else:
                di = (di+1)%4
                row, col = row + dr[di], col + dc[di]

        return result

```

```

"""
Given a collection of intervals, merge all overlapping intervals.

Example 1:

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

# Definition for an interval.
# class Interval(object):
#     def __init__(self, s=0, e=0):
#         self.start = s
#         self.end = e

class compare(object):
    def __init__(self, interval):
        self.interval = interval

    def __lt__(self, other):
        if self.interval.start == other.interval.start:
            return self.interval.end < other.interval.end
        return self.interval.start < other.interval.end

class Solution(object):
    def merge(self, intervals):
        """
        :type intervals: List[Interval]
        :rtype: List[Interval]
        """

        intervals = sorted(intervals, key= compare)
        # intervals.sort(key=lambda x: x.start)
        merged = []
        for interval in intervals:
            if not merged or merged[-1].end < interval.start:
                merged.append(interval)
            else:
                merged[-1].end = max(merged[-1].end, interval.end)
        return merged

# Time: O(N*log(N))
# Space: O(1)

```

```
'''
```

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

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

```
Example 1:
```

```
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].
```

```
'''
```

```
# Definition for an interval.  
# class Interval(object):  
#     def __init__(self, s=0, e=0):  
#         self.start = s  
#         self.end = e  
  
class Solution(object):  
    def insert(self, intervals, newInterval):  
        """  
        :type intervals: List[Interval]  
        :type newInterval: Interval  
        :rtype: List[Interval]  
        """  
        result = []  
        for interval in intervals:  
            if newInterval.start > interval.end:  
                result.append(interval)  
            elif newInterval.end < interval.start:  
                result.append(newInterval)  
                newInterval = interval  
            elif newInterval.start <= interval.end or newInterval.end >=  
interval.start:  
                newInterval = Interval(min(newInterval.start,  
interval.start), max(interval.end, newInterval.end))  
  
            result.append(newInterval)  
        return result
```

```
class Solution(object):
    def getPermutation(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: str
        """

        nums = []
        for index in range(1, n+1):
            nums.append(index)

        def permute(nums):
            if len(nums) == 0:
                return []
            if len(nums) == 1:
                return [nums]

            result = []
            for index in range(len(nums)):
                for p in permute(nums[0:index] + nums[index+1:]):
                    result.append([nums[index]] + p)

            return result

        value = permute(nums)[k-1]
        result = ""
        for val in value:
            result += str(val)
        return result
```

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def rotateRight(self, head, k):
        """
        :type head: ListNode
        :type k: int
        :rtype: ListNode
        """
        if k == 0:
            return head
        if not head:
            return None

        tempHead, length = head, 1
        while tempHead.next:
            length += 1
            tempHead = tempHead.next

        tempHead.next = head
        jump = (length-k)%length

        previous = tempHead
        while jump > 0:
            previous = previous.next
            jump -= 1
        head = previous.next
        previous.next = None

        return head
```

```
'''
```

```
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?
```

```
'''
```

```
class Solution(object):
    def uniquePaths(self, m, n):
        """
        :type m: int
        :type n: int
        :rtype: int
        """
        dp = [[0 for _ in range(n)] for _ in range(m)]
        for index in range(m):
            dp[index][0] = 1

        for index in range(n):
            dp[0][index] = 1

        for index_i in range(1, m):
            for index_j in range(1, n):
                dp[index_i][index_j] = dp[index_i-1][index_j] +
dp[index_i][index_j-1]

        return dp[m-1][n-1]
```

'''

A robot is located at the top-left corner of a  $m \times n$  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?

'''

```
class Solution(object):
    def uniquePathsWithObstacles(self, obstacleGrid):
        """
        :type obstacleGrid: List[List[int]]
        :rtype: int
        """
        m, n = len(obstacleGrid), len(obstacleGrid[0])
        dp = [[0 for _ in range(n)] for _ in range(m)]

        if obstacleGrid[0][0] == 1 or obstacleGrid[m-1][n-1] == 1:
            return 0

        dp[0][0] = 1
        for index in range(1, m):
            if obstacleGrid[index][0] == 1:
                dp[index][0] = 0
            else:
                dp[index][0] = dp[index-1][0]

        for index in range(1, n):
            if obstacleGrid[0][index] == 1:
                dp[0][index] = 0
            else:
                dp[0][index] = dp[0][index-1]

        for index_i in range(1, m):
            for index_j in range(1, n):
                if obstacleGrid[index_i][index_j] == 1:
                    dp[index_i][index_j] = 0
                else:
                    dp[index_i][index_j] = dp[index_i-1][index_j] +
dp[index_i][index_j-1]

        return dp[m-1][n-1]
```

```

"""
    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.
"""

class Solution(object):
    def minPathSum(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """

        if not grid:
            return 0

        row, col = len(grid), len(grid[0])
        dp = [[0 for _ in range(col)] for _ in range(row)]
        dp[0][0] = grid[0][0]

        for index in range(1, row):
            dp[index][0] = dp[index-1][0] + grid[index][0]

        for index in range(1, col):
            dp[0][index] = dp[0][index-1] + grid[0][index]

        print dp
        for index_i in range(1, row):
            for index_j in range(1, col):
                dp[index_i][index_j] = min(dp[index_i-1][index_j],
dp[index_i][index_j-1]) + grid[index_i][index_j]

        return dp[row-1][col-1]

```

```
'''  
    Validate if a given string is numeric.
```

```
Some examples:
```

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

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

```
'''  
  
class Solution(object):  
    def isNumber(self, s):  
        """  
        :type s: str  
        :rtype: bool  
        """  
        s = s.strip()  
        try:  
            if isinstance(float(s), float) or isinstance(int(s), int):  
                return True  
        except Exception as e:  
            return False  
  
    # Time: O(1)  
    # Space: O(1)
```

```

"""
    Given a non-empty array of digits representing a non-negative
integer, plus one to the integer.

    The digits are stored such that the most significant digit is at
the head of the list, and each element in the array contain a single
digit.

    You may assume the integer does not contain any leading zero,
except the number 0 itself.

Example 1:

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

class Solution(object):
    def plusOne(self, digits):
        """
        :type digits: List[int]
        :rtype: List[int]
        """
        result = []
        if not digits:
            return []

        carry = 1
        new_digits = digits[::-1]

        for index in range(len(new_digits)):
            new_digits[index], carry = (new_digits[index] + carry)%10,
            (new_digits[index] + carry)/10

            if carry > 0:
                new_digits.append(carry)
        return new_digits[::-1]

Time: O(N)
Space: O(1)

```

```

"""
Given two binary strings, return their sum (also a binary string).

The input strings are both non-empty and contains only characters 1
or 0.

Example 1:

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

Example 2:

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

class Solution(object):
    def addBinary(self, a, b):
        """
        :type a: str
        :type b: str
        :rtype: str
        """

        result = ""

        carry = 0
        index_a, index_b = len(a)-1, len(b)-1
        while index_a >= 0 and index_b >= 0:
            result = (int(a[index_a]) + int(b[index_b]) + carry)%2 +
result
            carry = (int(a[index_a]) + int(b[index_b]) + carry)%2
            index_a -= 1
            index_b -= 1

        if index_a >= 0:
            while index_a >= 0:
                result = (int(a[index_a]) + carry)%2 + result
                carry = (int(a[index_a]) + carry)%2
                index_a -= 1
        elif index_b >= 0:
            while index_b >= 0:
                result = (int(b[index_b]) + carry)%2 + result
                carry = (int(b[index_b]) + carry)%2
                index_b -= 1
        else:
            if carry == 1:
                result = str(carry) + result
return result

```

```

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

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

Note: Given n will be a positive integer.

Example 1:

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

class Solution(object):
    def climbStairs(self, n):
        """
        :type n: int
        :rtype: int
        """
        if n == 0:
            return 0

        dp = [0]*n
        dp[0], dp[1] = 1, 2

        for index in range(2, n):
            dp[index] = dp[index-1] + dp[index-2]
        return dp[n-1]

# Time: O(N)
# Space: O(N)

```

```

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

For example,
path = "/home/", => "/home"
path = "/a/./b/../../c/", => "/c"
"""

class Solution(object):
    def simplifyPath(self, path):
        """
        :type path: str
        :rtype: str
        """

        result = "/"
        stack = []

        index = 0
        while index < len(path):
            if path[index] == '/':
                index += 1
                continue

            curr_str = ""
            while index < len(path) and path[index] != '/':
                curr_str += path[index]
                index += 1

            if curr_str == '.' or curr_str == "":
                index += 1
                continue
            elif curr_str == "..":
                if stack:
                    stack.pop()
                index += 1
            else:
                stack.append(curr_str)
                index += 1

        for index in range(len(stack)):
            if index != len(stack) - 1:
                result += stack[index] + '/'
            else:
                result += stack[index]

        return result

# Time: O(N)
# Space: O(N)

```

```
'''
```

```
    Given two words word1 and word2, find the minimum number of  
operations required to convert word1 to word2.
```

```
You have the following 3 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')
```

```
'''
```

```
class Solution(object):  
    def minDistance(self, word1, word2):  
        """  
        :type word1: str  
        :type word2: str  
        :rtype: int  
        """  
        m , n = len(word1), len(word2)  
  
        dp = [[0 for _ in range(n+1)] for _ in range(m+1)]  
        for index_i in range(m+1):  
            for index_j in range(n+1):  
                if index_i == 0:  
                    dp[index_i][index_j] = index_j  
                elif index_j == 0:  
                    dp[index_i][index_j] = index_i  
                elif word1[index_i-1] == word2[index_j-1]:  
                    dp[index_i][index_j] = dp[index_i-1][index_j-1]  
                else:  
                    dp[index_i][index_j] = 1 + min(dp[index_i-  
1][index_j], dp[index_i-1][index_j-1], dp[index_i][index_j-1])  
  
        return dp[m][n]
```

```
class Solution(object):
    def setZeroes(self, matrix):
        """
        :type matrix: List[List[int]]
        :rtype: void Do not return anything, modify matrix in-place
instead.
        """
        col0 = 1
        for row in range(len(matrix)):
            if matrix[row][0] == 0:
                col0 = 0
            for col in range(1, len(matrix[0])):
                if matrix[row][col] == 0:
                    matrix[row][0] = 0
                    matrix[0][col] = 0
        for row in range(len(matrix)-1, -1, -1):
            for col in range(len(matrix[0])-1, 0, -1):
                if matrix[row][0] == 0 or matrix[0][col] == 0:
                    matrix[row][col] = 0
        if col0 == 0:
            matrix[row][0] = 0
```

```

"""
    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,  50]
]
target = 3
Output: true
"""

class Solution(object):
    def searchMatrix(self, matrix, target):
        """
        :type matrix: List[List[int]]
        :type target: int
        :rtype: bool
        """

        if not matrix:
            return 0
        left, right = 0, len(matrix[0])-1

        while left < len(matrix) and right >= 0:
            if matrix[left][right] == target:
                return True
            elif matrix[left][right] < target:
                left += 1
            else:
                right -= 1
        return False

```

```
'''
```

```
    Given an array 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.
```

```
    Here, we will use the integers 0, 1, and 2 to represent the color
    red, white, and blue respectively.
```

```
    Note: You are not suppose to use the library's sort function for
    this problem.
```

```
Example:
```

```
Input: [2,0,2,1,1,0]
Output: [0,0,1,1,2,2]
```

```
'''
```

```
class Solution(object):
    def sortColors(self, nums):
        """
        :type nums: List[int]
        :rtype: void Do not return anything, modify nums in-place
instead.
        """
        zero, last = 0, len(nums)-1
        index = 0
        while index <= last:
            if nums[index] == 1:
                index += 1
            elif nums[index] == 0:
                nums[index], nums[zero] = nums[zero], nums[index]
                index += 1
                zero += 1
            elif nums[index] == 2:
                nums[last], nums[index] = nums[index], nums[last]
                last -= 1
```

```
"""
Given a set of distinct integers, nums, return all possible subsets (the
power set).

"""

class Solution(object):
    def subsets(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        result = [[]]
        for num in nums:
            for j in range(len(result)):
                result.append(result[j] + [num])
        return result
```

```

"""
Given a 2D board and a word, find if the word exists in the grid.

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

Example:

board =
[
    ['A', 'B', 'C', 'E'],
    ['S', 'F', 'C', 'S'],
    ['A', 'D', 'E', 'E']
]

Given word = "ABCED", return true.
Given word = "SEE", return true.
Given word = "ABCB", return false.
"""

class Solution(object):
    def exist(self, board, word):
        """
        :type board: List[List[str]]
        :type word: str
        :rtype: bool
        """

        result = False
        for row in range(len(board)):
            for col in range(len(board[0])):
                if self.dfs(board, word, row, col, 0):
                    return True
        return False

    def dfs(self, board, word, row, col, curr_len):
        if row < 0 or col < 0 or row >= len(board) or col >= len(board[0]):
            return False
        if board[row][col] == word[curr_len]:
            c = board[row][col]
            board[row][col] = '#'

            if curr_len == len(word) - 1:
                return True
            elif (self.dfs(board, word, row-1, col, curr_len+1) or
                  self.dfs(board, word, row+1, col, curr_len+1) or
                  self.dfs(board, word, row, col-1, curr_len+1) or
                  self.dfs(board, word, row, col+1, curr_len+1)):
                return True

            board[row][col] = c
        return False

```

```
'''
```

```
    Given a sorted array nums, remove the duplicates in-place such that  
    duplicates appeared at most twice and return the new length.
```

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

```
Example 1:
```

```
Given nums = [1,1,1,2,2,3],
```

```
Your function should return length = 5, with the first five  
elements of nums being 1, 1, 2, 2 and 3 respectively.
```

```
It doesn't matter what you leave beyond the returned length.
```

```
'''
```

```
class Solution(object):  
    def removeDuplicates(self, nums):  
        """  
        :type nums: List[int]  
        :rtype: int  
        """  
        if len(nums) <= 2:  
            return len(nums)  
  
        prev, curr = 1, 2  
  
        while curr < len(nums):  
            if nums[prev] == nums[curr] and nums[curr] == nums[prev-1]:  
                curr += 1  
            else:  
                prev += 1  
                nums[prev] = nums[curr]  
                curr += 1  
        return prev+1
```

'''

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

(i.e., [0,0,1,2,2,5,6] might become [2,5,6,0,0,1,2]).

You are given a target value to search. If found in the array return true, otherwise return false.

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

'''

```
class Solution(object):
    def search(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: bool
        """
        left, right = 0, len(nums) - 1
        while left <= right:
            mid = (left + right) / 2
            if nums[mid] == target:
                return True
            if nums[left] < nums[mid]:
                if nums[left] <= target < nums[mid]:
                    right = mid - 1
                else:
                    left = mid + 1
            elif nums[mid] < nums[left]:
                if nums[mid] < target <= nums[right]:
                    left = mid + 1
                else:
                    right = mid - 1
            else:
                right = mid - 1
        left += 1

    return False
```

```
'''
```

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

```
Example 1:
```

```
Input: 1->2->3->3->4->4->5
```

```
Output: 1->2->5
```

```
Example 2:
```

```
Input: 1->1->1->2->3
```

```
Output: 2->3
```

```
'''
```

```
# Definition for singly-linked list.  
# class ListNode(object):  
#     def __init__(self, x):  
#         self.val = x  
#         self.next = None  
  
class Solution(object):  
    def deleteDuplicates(self, head):  
        """  
        :type head: ListNode  
        :rtype: ListNode  
        """  
        if not head:  
            return None  
  
        result = ListNode(0)  
        ans = result  
        curr = head  
        while curr:  
            value = curr.val  
            count = 0  
            while curr and curr.val == value:  
                curr = curr.next  
                count += 1  
            if count == 1:  
                result.next = ListNode(value)  
                result = result.next  
            curr = curr.next  
        return ans.next
```

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

```
Example 1:
```

```
Input: 1->1->2
```

```
Output: 1->2
```

```
Example 2:
```

```
Input: 1->1->2->3->3
```

```
Output: 1->2->3
```

```
'''  
  
# Definition for singly-linked list.  
# class ListNode(object):  
#     def __init__(self, x):  
#         self.val = x  
#         self.next = None  
  
class Solution(object):  
    def deleteDuplicates(self, head):  
        """  
        :type head: ListNode  
        :rtype: ListNode  
        """  
        if not head:  
            return None  
  
        curr = head  
        while curr and curr.next:  
            if curr.val == curr.next.val:  
                curr.next = curr.next.next  
            else:  
                curr = curr.next  
  
        return head
```

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

Example:

Input:

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

Output: 6

'''

```
class Solution(object):  
    def largestRectangleArea(self, heights):  
        """  
        :type heights: List[int]  
        :rtype: int  
        """  
        if not heights:  
            return 0  
  
        stack = []  
        result, index = 0, 0  
  
        while index < len(heights):  
            if not stack or heights[index] >= heights[stack[-1]]:  
                stack.append(index)  
                index += 1  
            else:  
                curr = stack.pop()  
                if not stack:  
                    area = heights[curr]*index  
                else:  
                    area = heights[curr] * (index-stack[-1]-1)  
                result = max(result, area)  
  
        while stack:  
            curr = stack.pop()  
            if not stack:  
                area = heights[curr]*index  
            else:  
                area = heights[curr] * (index-stack[-1]-1)  
            result = max(result, area)  
        return result  
  
    def maximalRectangle(self, matrix):  
        """  
        :type matrix: List[List[str]]  
        :rtype: int  
        """  
        if not matrix:  
            return 0  
        m, n = len(matrix), len(matrix[0])  
        heights = [0 for index in range(n)]
```

```
result = 0

for index_i in range(m):
    for index_j in range(n):
        if matrix[index_i][index_j] != '0':
            heights[index_j] = heights[index_j] + 1
        else:
            heights[index_j] = 0

    result = max(result, self.largestRectangleArea(heights))
return result
```

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

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

Example:

```
Input: head = 1->4->3->2->5->2, x = 3  
Output: 1->2->2->4->3->5
```

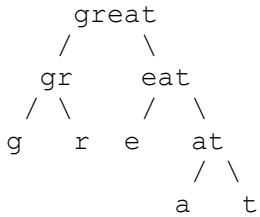
```
'''
```

```
# Definition for singly-linked list.  
# class ListNode(object):  
#     def __init__(self, x):  
#         self.val = x  
#         self.next = None  
  
class Solution(object):  
    def partition(self, head, x):  
        """  
        :type head: ListNode  
        :type x: int  
        :rtype: ListNode  
        """  
        if not head or not head.next:  
            return head  
  
        left, right = ListNode(0), ListNode(0)  
        leftPtr, rightPtr = left, right  
  
        while head:  
            if head.val < x:  
                leftPtr.next = ListNode(head.val)  
                leftPtr = leftPtr.next  
            else:  
                rightPtr.next = ListNode(head.val)  
                rightPtr = rightPtr.next  
            head = head.next  
  
        if not left.next:  
            return right.next  
        elif not right.next:  
            return left.next  
        else:  
            leftPtr.next = right.next  
            return left.next  
  
# Time: O(N)  
# Space: O(N)
```

'''

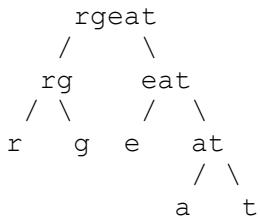
Given a string  $s_1$ , we may represent it as a binary tree by partitioning it to two non-empty substrings recursively.

Below is one possible representation of  $s_1 = "great"$ :



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

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



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

'''

```
class Solution(object):
    def __init__(self):
        self.cache = {}

    def isScramble(self, s1, s2):
        if s1 == s2:
            return True
        if s1+s2 in self.cache:
            return self.cache[s1+s2]
        if len(s1) != len(s2) or sorted(s1) != sorted(s2):
            self.cache[s1+s2] = False
            return False
        for index in range(1, len(s1)):
            if self.isScramble(s1[:index], s2[:index]) and
self.isScramble(s1[index:], s2[index:]):
                self.cache[s1+s2] = True
                return True
            if self.isScramble(s1[:index], s2[-index:]) and
self.isScramble(s1[index:], s2[0:-index]):
                self.cache[s1+s2] = True
                return True

        self.cache[s1+s2] = False
        return False
```

```
'''  
    Given a collection of integers that might contain duplicates, nums,  
    return all possible subsets (the power set).  
'''
```

Note: The solution set must not contain duplicate subsets.

Example:

Input: [1,2,2]

Output:

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

```
'''
```

```
class Solution(object):  
    def subsetsWithDup(self, nums):  
        """  
        :type nums: List[int]  
        :rtype: List[List[int]]  
        """  
        result = [[]]  
        for num in nums:  
            for index in range(len(result)):  
                new_list = result[index] + [num]  
                new_list.sort()  
                result.append(new_list)  
        unique = set(tuple(val) for val in result)  
        return list(list(val) for val in unique)
```

'''

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

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

Given a non-empty string containing only digits, determine the total number of ways to decode it.

Example 1:

Input: "12"

Output: 2

Explanation: It could be decoded as "AB" (1 2) or "L" (12).

Example 2:

Input: "226"

Output: 3

Explanation: It could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2 6).

'''

```
class Solution(object):  
    def numDecodings(self, s):  
        """  
        :type s: str  
        :rtype: int  
        """  
        if not s or s[0] == '0':  
            return 0  
        if len(s) == 1:  
            return 1  
  
        dp = [0]*len(s)  
        dp[0] = 1  
  
        if int(s[:2]) > 26:  
            if s[1] != '0':  
                dp[1] = 1  
            else:  
                dp[0] = 0  
        else:  
            if s[1] != '0':  
                dp[1] = 2  
            else:  
                dp[1] = 1  
  
        for index in range(2, len(s)):  
            if s[index] != '0':  
                dp[index] += dp[index-1]  
  
            val = int(s[index-1:index+1])  
            if val >= 10 and val <= 26:  
                dp[index] += dp[index-2]
```

```
    return dp[len(s)-1]

# Time: O(N)
# Space: O(N)
```

```

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

Note: 1 ≤ m ≤ n ≤ length of list.

Example:

Input: 1->2->3->4->5->NULL, m = 2, n = 4
Output: 1->4->3->2->5->NULL
"""

# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def reverseBetween(self, head, m, n):
        """
        :type head: ListNode
        :type m: int
        :type n: int
        :rtype: ListNode
        """
        if m == n:
            return head

        result = ListNode(0)
        result.next = head

        prev = result

        for index in range(m-1):
            prev = prev.next

        reverse = None
        curr = prev.next
        for i in range(n-m+1):
            temp = curr.next
            curr.next = reverse
            reverse = curr
            curr = temp

        prev.next.next = curr
        prev.next = reverse
        return result.next

```

```

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

Example:

Input: "25525511135"
Output: ["255.255.11.135", "255.255.111.35"]
"""

class Solution(object):
    def restoreIpAddresses(self, s):
        """
        :type s: str
        :rtype: List[str]
        """
        result = []

        def dfs(s, temp, count):
            if count == 4:
                if not s:
                    result.append(temp[:-1])
                    return
            for index in range(1, 4):
                if index <= len(s):
                    if index == 1:
                        dfs(s[index:], temp + s[:index] + ".", count+1)
                    elif index == 2 and s[0] != '0':
                        dfs(s[index:], temp + s[:index] + ".", count+1)
                    elif index == 3 and s[0] != '0' and int(s[:3]) <=
255:
                        dfs(s[index:], temp + s[:index] + ".", count+1)

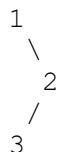
        dfs(s, "", 0)
        return result

```

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

Example:

Input: [1,null,2,3]



Output: [1,3,2]

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

```
'''  
  
# Definition for a binary tree node.  
# class TreeNode(object):  
#     def __init__(self, x):  
#         self.val = x  
#         self.left = None  
#         self.right = None  
  
class Solution(object):  
    def inorderTraversal(self, root):  
        """  
        :type root: TreeNode  
        :rtype: List[int]  
        """  
        if not root:  
            return []  
  
        stack, result = [root], []  
        while stack:  
            if root.left:  
                stack.append(root.left)  
                root = root.left  
            else:  
                node = stack.pop()  
                result.append(node.val)  
  
                if node.right:  
                    stack.append(node.right)  
                    root = node.right  
        return result
```

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

Example:

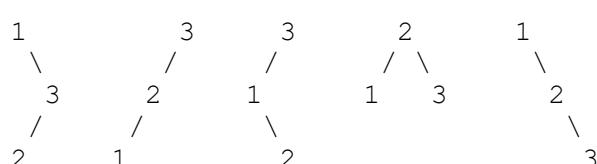
Input: 3

Output:

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

Explanation:

The above output corresponds to the 5 unique BST's shown below:



```
'''
```

```
# Definition for a binary tree node.  
# class TreeNode(object):  
#     def __init__(self, x):  
#         self.val = x  
#         self.left = None  
#         self.right = None  
  
class Solution(object):  
    def generateTrees(self, n):  
        """  
        :type n: int  
        :rtype: List[TreeNode]  
        """  
        if n == 0:  
            return []  
  
        def generate(start, end):  
            result = []  
            if start > end:  
                result.append(None)  
                return result  
  
            for index in range(start, end+1):  
                left = generate(start, index-1)  
                right = generate(index+1, end)  
  
                for l in left:  
                    for r in right:  
                        current = TreeNode(index)  
                        current.left = l  
                        current.right = r  
                        result.append(current)  
  
        return generate(1, n)
```

```
    return result  
  
return generate(1, n)
```

```

"""
    Given s1, s2, s3, find whether s3 is formed by the interleaving of
s1 and s2.

Example 1:

Input: s1 = "aabcc", s2 = "dbbca", s3 = "aadbcbcac"
Output: true
Example 2:

Input: s1 = "aabcc", s2 = "dbbca", s3 = "aadbcccacc"
Output: false
"""

class Solution(object):
    def isInterleave(self, s1, s2, s3):
        """
        :type s1: str
        :type s2: str
        :type s3: str
        :rtype: bool
        """

        if len(s3) != len(s1) + len(s2):
            return False

        dp = [[False for _ in range(len(s2)+1)] for _ in
range(len(s1)+1)]
        for row in range(len(s1)+1):
            for col in range(len(s2)+1):
                if row == 0 and col == 0:
                    dp[row][col] = True
                elif row == 0:
                    dp[row][col] = dp[row][col-1] and s2[col-1] ==
s3[row+col-1]
                elif col == 0:
                    dp[row][col] = dp[row-1][col] and s1[row-1] ==
s3[row+col-1]
                else:
                    dp[row][col] = (dp[row][col-1] and s2[col-1] ==
s3[row+col-1]) or (dp[row-1][col] and s1[row-1] == s3[row+col-1])

        return dp[len(s1)][len(s2)]

# Time: O(m*n)
# Space: O(m*n)

```

```
"""
    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.
```

```
"""
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def isValidBST(self, root):
        """
        :type root: TreeNode
        :rtype: bool
        """
        if not root:
            return True

        stack, result = [], []
        while stack or root:
            if root:
                stack.append(root)
                root = root.left
            else:
                root = stack.pop()
                result.append(root.val)
                root = root.right

        previous = result[0]
        for index in range(1, len(result)):
            if previous >= result[index]:
                return False
            previous = result[index]
        return True
```

```
'''
```

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

```
Recover the tree without changing its structure.
```

```
Example 1:
```

```
Input: [1,3,null,null,2]
```

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

```
Output: [3,1,null,null,2]
```

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

```
'''
```

```
# Definition for a binary tree node.  
# class TreeNode(object):  
#     def __init__(self, x):  
#         self.val = x  
#         self.left = None  
#         self.right = None  
  
class Solution(object):  
    def recoverTree(self, root):  
        """  
        :type root: TreeNode  
        :rtype: void Do not return anything, modify root in-place  
instead.  
        """  
  
        first, second, prev = None, None, None  
        def inorder(root):  
            if root:  
                inorder(root.left)  
                if prev is not None and root.val < prev.val:  
                    if first is None:  
                        first = root  
                    else:  
                        second = root  
                prev = root  
                inorder(root.right)  
  
        inorder(root)  
        if first and second:  
            first.val, second.val = second.val, first.val
```

'''

Given two binary trees, 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:        1                  1  
          / \                  / \  
          2    3              2    3  
  
[1,2,3],     [1,2,3]

Output: true

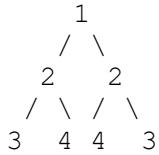
'''

```
# Definition for a binary tree node.  
# class TreeNode(object):  
#     def __init__(self, x):  
#         self.val = x  
#         self.left = None  
#         self.right = None  
  
class Solution(object):  
    def isSameTree(self, p, q):  
        """  
        :type p: TreeNode  
        :type q: TreeNode  
        :rtype: bool  
        """  
        if not p and not q:  
            return True  
  
        stack = [(p, q)]  
  
        while stack:  
            node1, node2 = stack.pop()  
            if node1 and node2 and node1.val == node2.val:  
                stack.append((node1.left, node2.left))  
                stack.append((node1.right, node2.right))  
            else:  
                if not node1 == node2:  
                    return False  
  
        return True
```

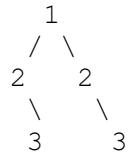
```
'''
```

```
    Given a binary tree, check whether it is a mirror of itself (ie,  
    symmetric around its center).
```

```
For example, this binary tree [1,2,2,3,4,4,3] is symmetric:
```



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



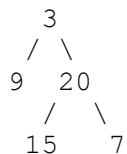
```
'''
```

```
# Definition for a binary tree node.  
# class TreeNode(object):  
#     def __init__(self, x):  
#         self.val = x  
#         self.left = None  
#         self.right = None  
  
class Solution(object):  
    def isSymmetric(self, root):  
        """  
        :type root: TreeNode  
        :rtype: bool  
        """  
        if not root:  
            return True  
  
        def dfs(left, right):  
            if not left and not right:  
                return True  
  
            if not left or not right:  
                return False  
            return (left.val == right.val) and dfs(left.left,  
right.right) and dfs(left.right, right.left)  
  
        return dfs(root, root)
```

```
'''  
    Given a binary tree, return the level order traversal of its nodes'  
values. (ie, from left to right, level by level).  
'''
```

For example:

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



return its level order traversal as:

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

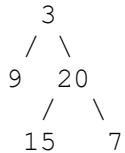
```
'''  
# Definition for a binary tree node.  
# class TreeNode(object):  
#     def __init__(self, x):  
#         self.val = x  
#         self.left = None  
#         self.right = None  
  
class Solution(object):  
    def levelOrder(self, root):  
        """  
        :type root: TreeNode  
        :rtype: List[List[int]]  
        """  
  
        if not root:  
            return []  
  
        queue = [(root, 0)]  
        levelMap = {}  
  
        while queue:  
            node, level = queue.pop(0)  
            if node.left:  
                queue.append((node.left, level+1))  
            if node.right:  
                queue.append((node.right, level+1))  
  
            if level in levelMap:  
                levelMap[level].append(node.val)  
            else:  
                levelMap[level] = [node.val]  
  
        result = []  
        for key, value in levelMap.items():  
            result.append(value)  
        return result
```

```
'''
```

```
    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],
```



```
return its zigzag level order traversal as:
```

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

```
'''
```

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def zigzagLevelOrder(self, root):
        """
        :type root: TreeNode
        :rtype: List[List[int]]
        """

        if not root:
            return []

        queue = [(root, 0)]
        levelMap = {}

        while queue:
            node, level = queue.pop(0)
            if node.left:
                queue.append((node.left, level+1))
            if node.right:
                queue.append((node.right, level+1))

            if level in levelMap:
                levelMap[level].append(node.val)
            else:
                levelMap[level] = [node.val]

        result = []
        spiral = False
        for key, value in levelMap.items():
            if spiral:
                result.append(value[::-1])
            else:
                result.append(value)
            spiral = not spiral

        return result
```

```
if spiral:  
    value = value[::-1]  
result.append(value)  
spiral = not spiral  
return result
```

```
'''
```

```
    Given a non-empty string s and a dictionary wordDict containing a
list of non-empty words, determine if s can be segmented into a space-
separated sequence of one or more dictionary words.
```

```
Note:
```

```
The same word in the dictionary may be reused multiple times in the
segmentation.
```

```
You may assume the dictionary does not contain duplicate words.
```

```
Example 1:
```

```
Input: s = "leetcode", wordDict = ["leet", "code"]  
Output: true
```

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

```
'''
```

```
# Definition for a binary tree node.
```

```
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
```

```
class Solution(object):
```

```
    def buildTree(self, preorder, inorder):
```

```
        """
```

```
        :type preorder: List[int]
        :type inorder: List[int]
        :rtype: TreeNode
        """
```

```
        self.index = 0
```

```
    def recursive(preorder, inorder, start, end):
```

```
        if start > end:
            return None
```

```
        node = TreeNode(preorder[self.index])
        self.index += 1
```

```
        if start == end:
            return node
```

```
        search_index = 0
```

```
        for i in range(start, end+1):
            if inorder[i] == node.val:
```

```
                search_index = i
                break
```

```
        node.left = recursive(preorder, inorder, start, search_index-1)
```

```
        node.right = recursive(preorder, inorder, search_index+1, end)
```

```
        return node
```

```
    return recursive(preorder, inorder, 0, len(inorder)-1)
```

```
'''
```

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

```
Note:
```

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

```
For example, given
```

```
inorder = [9,3,15,20,7]
postorder = [9,15,7,20,3]
Return the following binary tree:
```

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

```
'''
```

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def buildTree(self, inorder, postorder):
        """
        :type inorder: List[int]
        :type postorder: List[int]
        :rtype: TreeNode
        """
        self.index = len(inorder)-1
        def recursive(postorder, inorder, start, end):
            if start > end:
                return None

            node = TreeNode(postorder[self.index])
            self.index -= 1
            if start == end:
                return node

            search_index = 0
            for i in range(start, end+1):
                if inorder[i] == node.val:
                    search_index = i
                    break
            node.right = recursive(postorder, inorder, search_index+1,
end)
            node.left = recursive(postorder, inorder, start,
search_index-1)
            return node

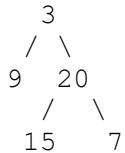
        return recursive(postorder, inorder, 0, len(inorder)-1)
```

'''

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],



return its bottom-up level order traversal as:

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

'''

```
# Definition for a binary tree node.  
# class TreeNode(object):  
#     def __init__(self, x):  
#         self.val = x  
#         self.left = None  
#         self.right = None  
  
class Solution(object):  
    def levelOrderBottom(self, root):  
        """  
        :type root: TreeNode  
        :rtype: List[List[int]]  
        """  
  
        if not root:  
            return []  
  
        queue = [(root, 0)]  
        levelMap = {}  
  
        while queue:  
            node, level = queue.pop(0)  
            if node.left:  
                queue.append((node.left, level+1))  
            if node.right:  
                queue.append((node.right, level+1))  
  
            if level in levelMap:  
                levelMap[level].append(node.val)  
            else:  
                levelMap[level] = [node.val]  
  
        result = []  
        for key, value in levelMap.items():  
            result.append(value)
```

```
return result[::-1]
```

```
'''
```

```
    Given an array 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:
```

```
Given the sorted array: [-10,-3,0,5,9],
```

```
One possible answer is: [0,-3,9,-10,null,5], which represents the  
following height balanced BST:
```

```
      0  
     / \  
    -3   9  
   /   /  
 -10  5
```

```
'''
```

```
# Definition for a binary tree node.  
# class TreeNode(object):  
#     def __init__(self, x):  
#         self.val = x  
#         self.left = None  
#         self.right = None  
  
class Solution(object):  
    def sortedArrayToBST(self, nums):  
        """  
        :type nums: List[int]  
        :rtype: TreeNode  
        """  
  
        def constructTree(nums, start, end):  
            if start > end:  
                return None  
  
            mid = (start+end)/2  
            node = TreeNode(nums[mid])  
  
            if start == end:  
                return node  
  
            node.left = constructTree(nums, start, mid-1)  
            node.right = constructTree(nums, mid+1, end)  
            return node  
  
        return constructTree(nums, 0, len(nums)-1)
```

```

''' 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:

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

      3
     / \
    9   20
     /   \
    15   7

return its minimum depth = 2.
'''

# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def minDepth(self, root):
        if not root:
            return 0
        depth = float('inf')
        stack = [(root, 1)]

        while stack:
            node, level = stack.pop()
            if node:
                if not node.left and not node.right:
                    depth = min(depth, level)
                stack.append((node.left, level+1))
                stack.append((node.right, level+1))

        return depth

```

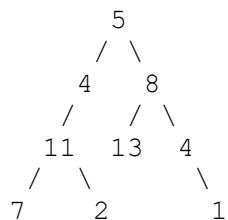
```
'''
```

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

```
Note: A leaf is a node with no children.
```

```
Example:
```

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



```
'''
```

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def hasPathSum(self, root, sum):
        """
        :type root: TreeNode
        :type sum: int
        :rtype: bool
        """
        if not root:
            return False

        if not root.left and not root.right and root.val == sum:
            return True

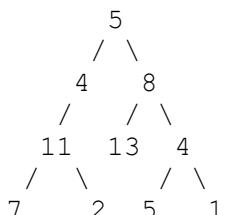
        return self.hasPathSum(root.left, sum-root.val) or
               self.hasPathSum(root.right, sum-root.val)
```

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

Note: A leaf is a node with no children.

Example:

Given the below binary tree and sum = 22,



Return:

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

```
'''  
  
# Definition for a binary tree node.  
# class TreeNode(object):  
#     def __init__(self, x):  
#         self.val = x  
#         self.left = None  
#         self.right = None  
  
class Solution(object):  
    def pathSum(self, root, sum):  
        """  
        :type root: TreeNode  
        :type sum: int  
        :rtype: List[List[int]]  
        """  
  
        result = []  
  
        def dfs(root, curr_sum, sum, path, result):  
            if not root:  
                return  
  
            curr_sum += root.val  
            if curr_sum == sum and not root.left and not root.right:  
                result.append(path + [root.val])  
                return  
  
            if root.left:  
                dfs(root.left, curr_sum, sum, path + [root.val], result)  
            if root.right:  
                dfs(root.right, curr_sum, sum, path + [root.val],  
result)
```

```
dfs(root, 0, sum, [], result)
return result
```

```
'''
```

```
    Given a string S and a string T, count the number of distinct
    subsequences of S which equals T.
```

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

```
Example 1:
```

```
Input: S = "rabbbit", T = "rabbit"
```

```
Output: 3
```

```
Explanation:
```

```
As shown below, there are 3 ways you can generate "rabbit" from S.
(The caret symbol ^ means the chosen letters)
```

```
rabbbit
^ ^ ^ ^ ^ ^
```

```
rabbbit
^ ^ ^ ^ ^ ^
```

```
rabbbit
^ ^ ^ ^ ^ ^
```

```
'''
```

```
class Solution(object):
    def numDistinct(self, s, t):
        """
        :type s: str
        :type t: str
        :rtype: int
        """

        row, col = len(s), len(t)

        if col > row:
            return 0

        dp = [[0 for _ in range(col+1)] for _ in range(row+1)]

        for r in range(row+1):
            for c in range(col+1):
                if r == 0 and c == 0:
                    dp[r][c] = 1
                elif r == 0:
                    dp[r][c] = 0
                elif c == 0:
                    dp[r][c] = 1
                else:
                    dp[r][c] = dp[r-1][c]
                    if s[r-1] == t[c-1]:
                        dp[r][c] += dp[r-1][c-1]

        return dp[row][col]
```

'''

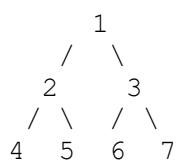
Given a binary tree

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

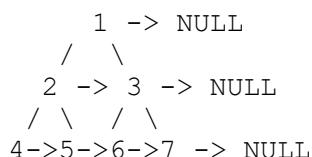
Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL  
Example:

Given the following perfect binary tree,



After calling your function, the tree should look like:



'''

```
# Definition for binary tree with next pointer.  
# class TreeLinkNode:  
#     def __init__(self, x):  
#         self.val = x  
#         self.left = None  
#         self.right = None  
#         self.next = None  
  
class Solution:  
    # @param root, a tree link node  
    # @return nothing  
    def connect(self, root):  
        def recursive(node):  
            if node is None:  
                return  
  
            if node.left:  
                node.left.next = node.right  
            if node.right:  
                if node.next:  
                    node.right.next = node.next.left  
                else:  
                    node.right.next = None  
            recursive(node.left)  
            recursive(node.right)  
  
            if root != None:  
                root.next = None
```

```
recursive(root)
```

'''

Given a binary tree

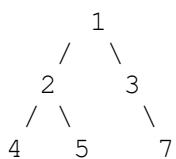
```
struct TreeLinkNode {  
    TreeLinkNode *left;  
    TreeLinkNode *right;  
    TreeLinkNode *next;  
}
```

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

Initially, all next pointers are set to NULL.

Example:

Given the following binary tree,



After calling your function, the tree should look like:

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

'''

```
# Definition for binary tree with next pointer.  
# class TreeLinkNode:  
#     def __init__(self, x):  
#         self.val = x  
#         self.left = None  
#         self.right = None  
#         self.next = None  
  
class Solution:  
    # @param root, a tree link node  
    # @return nothing  
    def connect(self, root):  
        if root == None:  
            return  
        queue = [root]  
        queue.append(None)  
  
        while queue:  
            front = queue.pop(0)  
            if front is not None:  
                front.next = queue[0]  
                if front.left:  
                    queue.append(front.left)  
                if front.right:  
                    queue.append(front.right)  
            elif queue:  
                queue.append(None)
```

```

# Definition for binary tree with next pointer.
# class TreeLinkNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
#         self.next = None

class Solution:
    # @param root, a tree link node
    # @return nothing
    def connect(self, root):
        if not root:
            return None

        root.next = None

        while root:
            temp = root
            while temp:
                if temp.left:
                    if temp.right:
                        temp.left.next = temp.right
                    else:
                        temp.left.next = self.getNext(temp)
                if temp.right:
                    temp.right.next = self.getNext(temp)

                temp = temp.next
            if root.left:
                root = root.left
            elif root.right:
                root = root.right
            else:
                root = self.getNext(root)

    def getNext(self, node):
        node = node.next
        while node:
            if node.left:
                return node.left
            if node.right:
                return node.right
            node = node.next
        return None

```

```

"""
    Given a non-negative integer numRows, generate the first numRows of
Pascal's triangle.

Example:

Input: 5
Output:
[
    [1],
    [1,1],
    [1,2,1],
    [1,3,3,1],
    [1,4,6,4,1]
]
"""

class Solution(object):
    def generate(self, numRows):
        """
:type numRows: int
:rtype: List[List[int]]
"""
        triangle = []

        for row in range(numRows):
            new_row = [0 for _ in range(row+1)]
            new_row[0], new_row[-1] = 1, 1

            for col in range(1, len(new_row)-1):
                new_row[col] = triangle[row-1][col-1] + triangle[row-1][col]

            triangle.append(new_row)
        return triangle

```

```
"""
    Given a non-negative index k where k ≤ 33, return the kth index row
of the Pascal's triangle.

    Note that the row index starts from 0.
"""

class Solution(object):
    def getRow(self, rowIndex):
        """
        :type rowIndex: int
        :rtype: List[int]
        """
        row = [1] * (rowIndex + 1)
        for i in range(1, rowIndex + 1):
            for j in range(i - 1, 0, -1):
                row[j] += row[j - 1]
        return row
```

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

For example, given the following triangle

```
[  
    [2],  
    [3, 4],  
    [6, 5, 7],  
    [4, 1, 8, 3]  
]  
The minimum path sum from top to bottom is 11 (i.e., 2 + 3 + 5 + 1 =  
11).  
'''  
  
class Solution(object):  
    def minimumTotal(self, triangle):  
        """  
        :type triangle: List[List[int]]  
        :rtype: int  
        """  
        length = len(triangle)  
        columns = len(triangle[length-1])  
  
        matrix = [[0 for col in range(columns)] for row in  
range(length)]  
        row_index = 0  
  
        for row in range(length):  
            elements = triangle[row]  
            col_index = 0  
  
            for val in elements:  
                matrix[row_index][col_index] = val  
                col_index += 1  
            row_index += 1  
  
        for row in range(length-2, -1, -1):  
            for col in range(row+1):  
                matrix[row][col] += min(matrix[row+1][col+1],  
matrix[row+1][col])  
        return matrix[0][0]
```

'''

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

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

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

Example 1:

Input: [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.

'''

```
class Solution(object):
    def maxProfit(self, prices):
        """
        :type prices: List[int]
        :rtype: int
        """
        if len(prices) < 2:
            return 0
        dp = [[0 for _ in range(len(prices))] for _ in range(3)]
        for i in range(1,3):
            maxDiff = -prices[0]
            for j in range(1,len(prices)):
                dp[i][j] = max(dp[i][j-1], prices[j] + maxDiff)
                maxDiff = max(maxDiff, dp[i-1][j] - prices[j])

        return dp[2][len(prices)-1]
```

'''

Given a non-empty binary tree, find the maximum path sum.

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

Example 1:

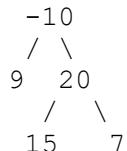
Input: [1,2,3]



Output: 6

Example 2:

Input: [-10,9,20,null,null,15,7]



Output: 42

'''

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def maxPathSum(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        self.result = float('-inf')
        self.dfs(root)
        return self.result

    def dfs(self, root):
        if not root:
            return 0

        l = self.dfs(root.left)
        r = self.dfs(root.right)

        max_one_end = max(max(l, r)+root.val, root.val)
        max_path = max(max_one_end, l+r+root.val)
        self.result = max(self.result, max_path)
        return max_one_end
```



```

class Solution(object):
    def numDistinct(self, s, t):
        """
        :type s: str
        :type t: str
        :rtype: int
        """

        row, col = len(s), len(t)

        if col > row:
            return 0

        dp = [[0 for _ in range(col+1)] for _ in range(row+1)]

        for r in range(row+1):
            for c in range(col+1):
                if r == 0 and c == 0:
                    dp[r][c] = 1
                elif r == 0:
                    dp[r][c] = 0
                elif c == 0:
                    dp[r][c] = 1
                else:
                    dp[r][c] = dp[r-1][c]
                    if s[r-1] == t[c-1]:
                        dp[r][c] += dp[r-1][c-1]
        return dp[row][col]

# Time: O(N^2)
# Space: O(N^2)

```

'''

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

Only one letter can be changed at a time.

Each transformed word must exist in the word list. Note that beginWord is not a transformed word.

Note:

Return 0 if there is no such transformation sequence.

All words have the same length.

All words contain only lowercase alphabetic characters.

You may assume no duplicates in the word list.

You may assume beginWord and endWord are non-empty and are not the same.

Example 1:

Input:

```
beginWord = "hit",
endWord = "cog",
wordList = ["hot", "dot", "dog", "lot", "log", "cog"]
```

Output: 5

Explanation: As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -> "cog",  
return its length 5.

'''

```
class Solution(object):
    def ladderLength(self, beginWord, endWord, wordList):
        """
        :type beginWord: str
        :type endWord: str
        :type wordList: List[str]
        :rtype: int
        """
        d = {}
        for word in wordList:
            for i in range(len(word)):
                s = word[:i] + "_" + word[i+1:]
                if s in d:
                    d[s].append(word)
                else:
                    d[s] = [word]
        print d
        queue, visited = [], set()
        queue.append((beginWord, 1))
        while queue:
            word, steps = queue.pop(0)
            if word not in visited:
                visited.add(word)

                if word == endWord:
                    return steps
                else:
                    for index in range(len(word)):
```

```
s = word[:index] + "_" + word[index+1:]
neigh_words = []
if s in d:
    neigh_words = d[s]

for neigh in neigh_words:
    if neigh not in visited:
        queue.append((neigh, steps+1))

return 0

Solution().ladderLength("hit", "cog",
["hot", "dot", "dog", "lot", "log", "cog"] )
```

```
'''
```

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

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

```
Example:
```

```
Input: [100, 4, 200, 1, 3, 2]
Output: 4
```

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

```
'''
```

```
class Solution(object):
    def longestConsecutive(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        result = 0
        nums = set(nums)

        for num in nums:
            if num-1 not in nums:
                curr = num
                length = 1

                while curr+1 in nums:
                    curr += 1
                    length += 1
                result = max(result, length)
        return result
```

'''

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

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

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

Note: A leaf is a node with no children.

Example:

Input: [1,2,3]



Output: 25

Explanation:

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

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

Therefore, sum = 12 + 13 = 25.

'''

```
# Definition for a binary tree node.
```

```
# class TreeNode(object):
```

```
#     def __init__(self, x):
```

```
#         self.val = x
```

```
#         self.left = None
```

```
#         self.right = None
```

```
class Solution(object):
```

```
    def sumNumbers(self, root):
```

```
        """
```

```
:type root: TreeNode
```

```
:rtype: int
```

```
"""
```

```
    if not root:
```

```
        return 0
```

```
    def dfs(root, num, total):
```

```
        if not root:
```

```
            return total
```

```
        num = num*10 + root.val
```

```
        if not root.left and not root.right:
```

```
            total += num
```

```
            return total
```

```
        return dfs(root.left, num) + dfs(root.right, num)
```

```
    return dfs(root, 0, 0)
```

```
'''  
    Given a 2D board containing 'X' and 'O' (the letter O), capture all  
regions surrounded by 'X'.  
'''
```

```
    A region is captured by flipping all 'O's into 'X's in that  
surrounded region.
```

Example:

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

After running your function, the board should be:

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

```
'''  
class Solution(object):  
    def solve(self, board):  
        """  
        :type board: List[List[str]]  
        :rtype: void Do not return anything, modify board in-place  
instead.  
        """  
        if len(board) == 0:  
            return  
        for row in range(len(board)):  
            if board[row][0] == 'O':  
                self.merge(board, row, 0)  
            if board[row][len(board[0])-1] == 'O':  
                self.merge(board, row, len(board[0])-1)  
  
        for col in range(len(board[0])):  
            if board[0][col] == 'O':  
                self.merge(board, 0, col)  
  
            if board[len(board)-1][col] == 'O':  
                self.merge(board, len(board)-1, col)  
  
        for row in range(len(board)):  
            for col in range(len(board[0])):  
                if board[row][col] == 'O':  
                    board[row][col] = 'X'  
                elif board[row][col] == '#':  
                    board[row][col] = 'O'  
  
    def merge(self, board, row, col):  
        if row < 0 or col < 0 or row >= len(board) or col >=  
len(board[0]):  
            return  
        if board[row][col] != 'O':  
            return  
  
        board[row][col] = '#'
```

```
self.merge(board, row+1, col)
self.merge(board, row, col-1)
self.merge(board, row, col+1)
self.merge(board, row-1, col)
```

```
'''  
    Given a string s, partition s such that every substring of the  
partition is a palindrome.  
  
    Return all possible palindrome partitioning of s.  
'''  
  
class Solution(object):  
    def partition(self, s):  
        result = []  
    def valid(s):  
        for i in range(len(s)/2):  
            if s[i] != s[-(i+1)]:  
                return False  
        return True  
  
    def partitionRec(curr, s, i):  
        if i == len(s):  
            result.append(curr)  
        else:  
            for j in range(i, len(s)):  
                if valid(s[i:j+1]):  
                    partitionRec(curr + [s[i:j+1]], s, j+1)  
  
partitionRec([], s, 0)  
return result
```

```

"""
    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:

Input: "aab"
Output: 1
Explanation: The palindrome partitioning ["aa","b"] could be
produced using 1 cut.

"""

class Solution(object):
    def minCut(self, s):
        """
        :type s: str
        :rtype: int
        """
        if not s:
            return 0

        P = [[False for _ in range(len(s))] for _ in range(len(s))]
        cuts = [0 for _ in range(len(s))]

        for index in range(len(s)):
            P[index][index] = True

        for length in range(2, len(s)+1):
            for i in range(len(s)-length+1):
                j = i+length - 1

                if length == 2:
                    P[i][j] = s[i] == s[j]
                else:
                    P[i][j] = (s[i] == s[j]) and P[i+1][j-1]

        for index in range(len(s)):
            if P[0][index]:
                cuts[index] = 0
            else:
                cuts[index] = float('inf')
                for j in range(index):
                    if P[j+1][index] and (cuts[index] > 1 + cuts[j]):
                        cuts[index] = 1+cuts[j]

        return cuts[len(s)-1]

# Time: O(N^2)
# Space: O(N^2)

```

'''

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

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

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

'''

```
class Solution(object):
    def canCompleteCircuit(self, gas, cost):
        """
        :type gas: List[int]
        :type cost: List[int]
        :rtype: int
        """
        start, curr_sum, total_sum = 0, 0, 0
        for index in range(len(gas)):
            diff = gas[index] - cost[index]
            total_sum += diff
            curr_sum += diff

            if curr_sum < 0:
                start = index + 1
                curr_sum = 0

        if total_sum >= 0:
            return start
        return -1
```



```
class Solution(object):
    def wordBreak(self, s, wordDict):
        """
        :type s: str
        :type wordDict: List[str]
        :rtype: List[str]
        """
        self.result = []
        self.dfs(s, wordDict, '')
        return self.result

    def dfs(self, s, wordDict, currStr):
        if self.check(s, wordDict):
            if len(s) == 0:
                self.result.append(currStr[1:])
            for i in range(1, len(s)+1):
                if s[:i] in wordDict:
                    self.dfs(s[i:], wordDict, currStr + ' ' + s[:i])

    def check(self, s, wordDict):
        dp = [False for _ in range(len(s)+1)]
        dp[0] = True

        for i in range(len(s)):
            for j in range(i, -1, -1):
                if dp[j] and s[j:i+1] in wordDict:
                    dp[i+1] = True
                    break

        return dp[len(s)]
```

```
'''  
    Given a linked list, determine if it has a cycle in it.  
  
    Follow up:  
    Can you solve it without using extra space?  
'''  
  
# Definition for singly-linked list.  
# class ListNode(object):  
#     def __init__(self, x):  
#         self.val = x  
#         self.next = None  
  
class Solution(object):  
    def hasCycle(self, head):  
        """  
        :type head: ListNode  
        :rtype: bool  
        """  
  
        if not head:  
            return False  
  
        slow, fast = head, head  
        while fast and fast.next:  
            slow = slow.next  
            fast = fast.next.next  
            if slow == fast:  
                return True  
        return False
```

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

Note: Do not modify the linked list.

Follow up:
Can you solve it without using extra space?
"""

# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def detectCycle(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        if not head:
            return None

        slow, fast = head, head.next

        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
            if slow == fast:
                break

        if slow == fast:
            slow = head
            while slow != fast:
                slow = slow.next
                fast = fast.next
            return slow
        return None
```

```

# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def reorderList(self, head):
        """
        :type head: ListNode
        :rtype: void Do not return anything, modify head in-place instead.
        """
        if not head:
            return None

        slow, fast = head, head.next

        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next

        head1, head2 = head, slow.next
        slow.next = None
        prev = None
        curr = head2
        while curr:
            nex = curr.next
            curr.next = prev
            prev = curr
            curr = nex
        head2 = prev

        while head2:
            n1 = head1.next
            n2 = head2.next
            head1.next = head2
            head1.next.next = n1
            head2 = n2
            head1 = head1.next.next

        head = head1

```

```

"""
    Given a binary tree, return the preorder traversal of its nodes' values.

Example:

Input: [1,null,2,3]
      1
       \
        2
       /
      3

Output: [1,2,3]
Follow up: Recursive solution is trivial, could you do it iteratively?
"""

# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def preorderTraversal(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        if not root:
            return []

        stack, result = [root], []
        while stack:
            element = stack.pop()
            result.append(element.val)

            if element.right:
                stack.append(element.right)
            if element.left:
                stack.append(element.left)

        return result

# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def preorderTraversal(self, root):
        """
        :type root: TreeNode

```

```
:rtype: List[int]
"""
result = []

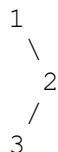
def recursive(root, result):
    if not root:
        return
    result.append(root.val)
    recursive(root.left, result)
    recursive(root.right, result)

recursive(root, result)
return result
```

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

Example:

Input: [1,null,2,3]



Output: [3,2,1]

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

```
# Definition for a binary tree node.  
# class TreeNode(object):  
#     def __init__(self, x):  
#         self.val = x  
#         self.left = None  
#         self.right = None  
  
class Solution(object):  
    def postorderTraversal(self, root):  
        """  
        :type root: TreeNode  
        :rtype: List[int]  
        """  
  
        result = []  
  
        def recursive(root, result):  
            if not root:  
                return  
            recursive(root.left, result)  
            recursive(root.right, result)  
            result.append(root.val)  
        recursive(root, result)  
        return result  
  
# Definition for a binary tree node.  
# class TreeNode(object):  
#     def __init__(self, x):  
#         self.val = x  
#         self.left = None  
#         self.right = None  
  
class Solution(object):  
    def postorderTraversal(self, root):  
        """  
        :type root: TreeNode  
        :rtype: List[int]  
        """
```

```
if not root:  
    return []  
  
stack, result = [], []  
  
while True:  
    while root:  
        if root.right:  
            stack.append(root.right)  
        stack.append(root)  
        root = root.left  
  
    root = stack.pop()  
  
    if root.right and stack and stack[-1] == root.right:  
        stack.pop()  
        stack.append(root)  
        root = root.right  
    else:  
        result.append(root.val)  
        root = None  
  
    if len(stack)<=0:  
        break  
  
return result
```

'''

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

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

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

Follow up:

Could you do both operations in O(1) time complexity?

Example:

```
LRUCache cache = new LRUCache( 2 /* capacity */ );
```

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

'''

```
class Node(object):
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.next = None
        self.prev = None
```

```
class LRUCache(object):
```

```
    def __init__(self, capacity):
        """
        :type capacity: int
        """
        self.capacity = capacity
        self.mapping = dict()
        self.head = Node(0, 0)
        self.tail = Node(0, 0)
        self.head.next = self.tail
        self.tail.prev = self.head
```

```
    def get(self, key):
        """
        :type key: int
        :rtype: int
        """
        if key in self.mapping:
            node = self.mapping[key]
            self.remove(node)
```

```

        self.add(node)
        return node.value
    return -1

def put(self, key, value):
    """
    :type key: int
    :type value: int
    :rtype: void
    """

    if key in self.mapping:
        self.remove(self.mapping[key])

    node = Node(key, value)
    if len(self.mapping) >= self.capacity:
        next_head = self.head.next
        self.remove(next_head)
        del self.mapping[next_head.key]

    self.add(node)
    self.mapping[key] = node

def add(self, node):
    tail = self.tail.prev
    tail.next = node
    self.tail.prev = node
    node.prev = tail
    node.next = self.tail

def remove(self, node):
    prev_node = node.prev
    prev_node.next = node.next
    node.next.prev = prev_node

# Your LRUCache object will be instantiated and called as such:
# obj = LRUCache(capacity)
# param_1 = obj.get(key)
# obj.put(key,value)

```

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def insertionSortList(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """

        if not head:
            return None

        sortedList = head
        head = head.next
        sortedList.next = None

        while head:
            curr = head
            head = head.next
            if curr.val <= sortedList.val:
                curr.next = sortedList
                sortedList = curr
            else:
                temp = sortedList
                while temp.next and temp.next.val < curr.val:
                    temp = temp.next
                curr.next = temp.next
                temp.next = curr
        return sortedList
```

```

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

Example 1:

Input: 4->2->1->3
Output: 1->2->3->4
"""

# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def sortList(self, head):
        """
:type head: ListNode
:rtype: ListNode
"""

        if not head or not head.next:
            return head

        slow, fast = head, head.next

        while fast.next and fast.next.next:
            slow = slow.next
            fast = fast.next.next

        head1, head2 = head, slow.next
        slow.next = None
        head1 = self.sortList(head1)
        head2 = self.sortList(head2)
        head = self.merge(head1, head2)
        return head

    def merge(self, head1, head2):
        if not head1:
            return head2
        if not head2:
            return head1

        result = ListNode(0)
        p = result

        while head1 and head2:
            if head1.val <= head2.val:
                p.next = ListNode(head1.val)
                head1 = head1.next
                p = p.next
            else:
                p.next = ListNode(head2.val)
                head2 = head2.next
                p = p.next

```

```
if head1:  
    p.next = head1  
if head2:  
    p.next = head2  
return result.next
```

```

"""
Evaluate the value of an arithmetic expression in Reverse Polish Notation.

Valid operators are +, -, *, /. Each operand may be an integer or another expression.

Note:

Division between two integers should truncate toward zero.
The given RPN expression is always valid. That means the expression would always evaluate to a result and there won't be any divide by zero operation.

Example 1:

Input: ["2", "1", "+", "3", "*"]
Output: 9
Explanation: ((2 + 1) * 3) = 9

Example 2:

Input: ["4", "13", "5", "/", "+"]
Output: 6
Explanation: (4 + (13 / 5)) = 6
"""

class Solution(object):
    def evalRPN(self, tokens):
        """
        :type tokens: List[str]
        :rtype: int
        """

        if not tokens:
            return 0

        stack = []
        for val in tokens:
            if val == '+':
                val1 = stack.pop()
                val2 = stack.pop()
                stack.append(val1 + val2)
            elif val == '-':
                val1 = stack.pop()
                val2 = stack.pop()
                stack.append(val2 - val1)
            elif val == '*':
                val1 = stack.pop()
                val2 = stack.pop()
                stack.append(val2 * val1)
            elif val == '/':
                val1 = stack.pop()
                val2 = stack.pop()
                if val1 * val2 < 0:
                    stack.append(-(-val2 / val1))
                else:
                    stack.append(val2 / val1)
            else:
                stack.append(int(val))

```

```
return stack[0]
```

```
'''
```

```
    Given an integer array nums, find the contiguous subarray within an
array (containing at least one number) which has the largest product.
```

```
Example 1:
```

```
Input: [2,3,-2,4]
```

```
Output: 6
```

```
Explanation: [2,3] has the largest product 6.
```

```
Example 2:
```

```
Input: [-2,0,-1]
```

```
Output: 0
```

```
Explanation: The result cannot be 2, because [-2,-1] is not a
subarray.
```

```
'''
```

```
class Solution(object):
    def maxProduct(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """

        if not nums:
            return 0

        max_so_far, min_so_far, result = nums[0], nums[0], nums[0]

        for index in range(1, len(nums)):
            if nums[index] > 0:
                max_so_far = max(max_so_far*nums[index], nums[index])
                min_so_far = min(min_so_far*nums[index], nums[index])
            else:
                temp = max_so_far
                max_so_far = max(min_so_far*nums[index], nums[index])
                min_so_far = min(temp*nums[index], nums[index])

            result = max(result, max_so_far)
        return result
```

'''

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

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

Find the minimum element.

You may assume no duplicate exists in the array.

Example 1:

Input: [3,4,5,1,2]

Output: 1

Example 2:

Input: [4,5,6,7,0,1,2]

Output: 0

'''

```
class Solution(object):
    def findMin(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        if not nums:
            return 0

        if len(nums) == 1:
            return nums[0]
        left, right = 0, len(nums)-1

        if nums[left] < nums[right]:
            return nums[left]
        while left <= right:
            while nums[left] == nums[right] and left != right:
                left += 1

            if nums[left] <= nums[right]:
                return nums[left]

            mid = (left + right)/2
            if nums[mid] >= nums[left]:
                left = mid+1
            else:
                right = mid
        return -1
```

```

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

push(x) -- Push element x onto stack.
pop() -- Removes the element on top of the stack.
top() -- Get the top element.
getMin() -- Retrieve the minimum element in the stack.
Example:
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();    --> Returns -3.
minStack.pop();
minStack.top();      --> Returns 0.
minStack.getMin();    --> Returns -2.
"""

class MinStack(object):

    def __init__(self):
        """
        initialize your data structure here.
        """
        self.stack = []
        self.minimum = float('inf')

    def push(self, x):
        """
        :type x: int
        :rtype: void
        """
        if not self.stack:
            self.stack.append(x)
            self.minimum = x
        else:
            if x < self.minimum:
                self.stack.append(2*x-self.minimum)
                self.minimum = x
            else:
                self.stack.append(x)

        print self.stack

    def pop(self):
        """
        :rtype: void
        """
        if self.stack:
            top = self.stack.pop()
            if top < self.minimum:
                self.minimum = 2*self.minimum - top

    def top(self):
        """

```

```
:rtype: int
"""
if not self.stack:
    return None
else:
    top = self.stack[-1]
    if top < self.minimum:
        return self.minimum
    else:
        return top

def getMin(self):
    """
:type: int
"""
if self.stack:
    return self.minimum
else:
    return None

# Your MinStack object will be instantiated and called as such:
# obj = MinStack()
# obj.push(x)
# obj.pop()
# param_3 = obj.top()
# param_4 = obj.getMin()
```

```
"""
Given a string, find the longest substring that contains only two unique
characters. For example, given "abcbbbcccbdddadacb", the longest
substring that contains 2 unique character is "bcbbbbccb".
"""

class Solution(object):
    def lengthOfLongestSubstringTwoDistinct(self, s):
        """
        :type s: str
        :rtype: int
        """
        if not s:
            return 0

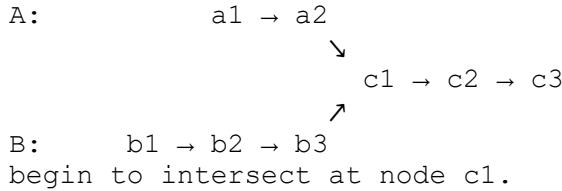
        unique_char, start, result = {}, 0, 0
        for index, char in enumerate(s):
            if char in unique_char:
                unique_char[s] += 1
            else:
                unique_char[s] = 1

            if len(unique_char) <= 2:
                result = max(result, index-start+1)
            else:
                while len(unique_char) > 2:
                    char_index = s[start]
                    count = unique_char[char_index]
                    if count == 1:
                        del unique_char[char_index]
                    else:
                        unique_char[char_index] -= 1
                    start += 1
        return result
```

'''

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

For example, the following two linked lists:



'''

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def getIntersectionNode(self, headA, headB):
        """
        :type head1, head2: ListNode
        :rtype: ListNode
        """
        if not headA or not headB:
            return None

        pa, pb = headA, headB
        while pa != pb:
            pa = pa.next if pa is not None else headB
            pb = pb.next if pb is not None else headA

        return pa if pa else None
```

```

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

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

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

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

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.

"""

class Solution(object):
    def findPeakElement(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        left, right = 0, len(nums)-1
        while left < right:
            mid = (left + right) / 2
            if nums[mid] > nums[mid+1]:
                right = mid
            else:
                left = mid + 1
        return left

class Solution(object):
    def findPeakElement(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        left = [False]*len(nums)
        right = [False]*len(nums)
        left[0], right[len(nums)-1] = True, True

        for index in range(1, len(nums)):
            if nums[index] > nums[index-1]:
                left[index] = True

        for index in range(len(nums)-2, -1, -1):
            if nums[index] > nums[index+1]:
                right[index] = True

        for index in range(len(left)):
            if left[index] and right[index]:
                return index
        return -1

```

```

"""
Given a sorted integer array where the range of elements are in the
inclusive range [lower, upper], return its missing ranges.

For example, given [0, 1, 3, 50, 75], lower = 0 and upper = 99, return
["2", "4->49", "51->74", "76->99"].
"""

class Solution(object):
    def missingRange(self, A, lower, upper):
        if not A:
            return []

        result = []
        if A[0] != lower:
            end = A[0] - 1
            if end == lower:
                m_r = str(lower)
            else:
                m_r = str(lower) + "->" + str(end)
            result.append(m_r)

        for index in range(1, len(A)):
            if A[index] != A[index-1] + 1:
                start = A[index-1] + 1
                end = A[index] - 1
                if start == end:
                    m_r = str(start)
                else:
                    m_r = str(start) + "->" + str(end)
                result.append(m_r)

        if A[len(A) - 1] != upper:
            start = A[len(A)-1] + 1
            if start == upper:
                m_r = str(start)
            else:
                m_r = str(start) + "->" + str(upper)
            result.append(m_r)
        return result

solution = Solution()
print solution.missingRange([0, 1, 3, 50, 75], 0, 99)
print solution.missingRange([4, 10, 50, 98], 0, 99)
print solution.missingRange([0], 0, 1)

```

```

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

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

For example,
add(1); add(3); add(5);
find(4) -> true
find(7) -> false
"""

class TwoSum(object):

    def __init__(self):
        """
        initialize your data structure here
        """
        self.value_count = {}

    def add(self, number):
        """
        Add the number to an internal data structure.
        :rtype: nothing
        """
        if number in self.value_count:
            self.value_count[number] += 1
        else:
            self.value_count[number] = 1

    def find(self, value):
        """
        Find if there exists any pair of numbers which sum is equal to
        the value.
        :type value: int
        :rtype: bool
        """
        for val in self.value_count:
            diff = value - val
            if diff in self.value_count and (diff != val or
self.value_count[val] > 1):
                return True
        return False

# Your TwoSum object will be instantiated and called as such:
# twoSum = TwoSum()
# twoSum.add(number)
# twoSum.find(value)

```

```

"""
Implement an iterator over a binary search tree (BST). Your
iterator will be initialized with the root node of a BST.

Calling next() will return the next smallest number in the BST.

Note: next() and hasNext() should run in average O(1) time and uses
O(h) memory, where h is the height of the tree.
"""

# Definition for a binary tree node
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class BSTIterator(object):
    def __init__(self, root):
        """
        :type root: TreeNode
        """
        self.stack = []
        while root:
            self.stack.append(root)
            root = root.left

    def hasNext(self):
        """
        :rtype: bool
        """
        return self.stack

    def next(self):
        """
        :rtype: int
        """
        node = self.stack.pop()
        new_node = node.right
        while new_node:
            self.stack.append(new_node)
            new_node = new_node.left
        return node.val

# Your BSTIterator will be called like this:
# i, v = BSTIterator(root), []
# while i.hasNext(): v.append(i.next())

```

```
'''
```

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

```
Example 1:
```

```
Input: [10,2]
```

```
Output: "210"
```

```
Example 2:
```

```
Input: [3,30,34,5,9]
```

```
Output: "9534330"
```

```
# '''
```

```
class Solution:  
    # @param {integer[]} nums  
    # @return {string}  
    def largestNumber(self, nums):  
        nums = [str(num) for num in nums]  
        nums.sort(cmp=lambda x, y : cmp(y+x, x+y))  
        return ''.join(nums).lstrip("0") or "0"
```

```
class Solution:
    # @param n, an integer
    # @return an integer
    def reverseBits(self, n):
        res = 0
        for i in range(32):
            res += n & 1
            n = n >> 1
            if i != 31:
                res = res << 1
        return res

print Solution().reverseBits(12)
```

```
class Solution(object):
    def hammingWeight(self, n):
        """
        :type n: int
        :rtype: int
        """
        bits = 0
        mask = 1

        for i in range(32):
            if (n&mask) != 0:
                bits +=1
            mask <<= 1

        return bits
```

```
'''
```

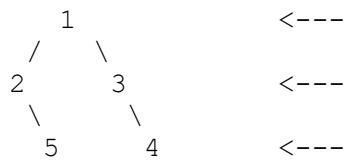
```
    Given a binary tree, imagine yourself standing on the right side of  
it, return the values of the nodes you can see ordered from top to  
bottom.
```

```
Example:
```

```
Input: [1,2,3,null,5,null,4]
```

```
Output: [1, 3, 4]
```

```
Explanation:
```



```
'''
```

```
# Definition for a binary tree node.  
# class TreeNode(object):  
#     def __init__(self, x):  
#         self.val = x  
#         self.left = None  
#         self.right = None  
  
class Solution(object):  
    def rightSideView(self, root):  
        """  
        :type root: TreeNode  
        :rtype: List[int]  
        """  
        if not root:  
            return []  
  
        stack, node_depth = [(root, 0)], {}  
  
        while stack:  
            node, depth = stack.pop(0)  
            if depth not in node_depth:  
                node_depth[depth] = node.val  
  
            if node.right:  
                stack.append((node.right, depth+1))  
            if node.left:  
                stack.append((node.left, depth+1))  
        return node_depth.values()
```

```
class Solution(object):
    def numIslands(self, grid):
        """
        :type grid: List[List[str]]
        :rtype: int
        """
        if not grid:
            return 0

        count = 0
        for row in range(len(grid)):
            for col in range(len(grid[0])):
                if grid[row][col] == '1':
                    count +=1
                    self.merge(grid, row, col)

        return count

    def merge(self, grid, row, col):
        if 0 > row or row >= len(grid) or col < 0 or col >= len(grid[0]):
            return

        if grid[row][col] != '1':
            return

        grid[row][col] = '#'
        self.merge(grid, row+1, col)
        self.merge(grid, row-1, col)
        self.merge(grid, row, col+1)
        self.merge(grid, row, col-1)
```



```
'''  
    Reverse a singly linked list.
```

```
Example:
```

```
Input: 1->2->3->4->5->NULL  
Output: 5->4->3->2->1->NULL
```

```
'''  
  
# Definition for singly-linked list.  
# class ListNode(object):  
#     def __init__(self, x):  
#         self.val = x  
#         self.next = None  
  
class Solution(object):  
    def reverseList(self, head):  
        """  
        :type head: ListNode  
        :rtype: ListNode  
        """  
        if not head:  
            return None  
  
        prev, curr = None, head  
        while curr:  
            temp = curr.next  
            curr.next = prev  
            prev = curr  
            curr = temp  
        return prev
```

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

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

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

Example 1:

Input: 2, [[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.

```
'''
```

```
class Solution(object):  
    def canFinish(self, numCourses, prerequisites):  
        """  
        :type numCourses: int  
        :type prerequisites: List[List[int]]  
        :rtype: bool  
        """  
  
        graph = [[] for _ in range(numCourses)]  
        visited = [False for _ in range(numCourses)]  
        stack = [False for _ in range(numCourses)]  
  
        for pair in prerequisites:  
            x, y = pair  
            graph[x].append(y)  
  
        for course in range(numCourses):  
            if visited[course] == False:  
                if self.dfs(graph, visited, stack, course):  
                    return False  
            return True  
  
    def dfs(self, graph, visited, stack, course):  
        visited[course] = True  
        stack[course] = True  
  
        for neigh in graph[course]:  
            if visited[neigh] == False:  
                if self.dfs(graph, visited, stack, neigh):  
                    return True  
  
            elif stack[neigh]:  
                return True  
        stack[course] = False  
        return False
```

```

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

class TreeNode(object):
    self.word = False
    self.children = {}

class Trie(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.root = TreeNode()

    def insert(self, word):
        """
        Inserts a word into the trie.
        :type word: str
        :rtype: void
        """
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TreeNode()
            node = node.children[char]
        node.word = True

    def search(self, word):
        """
        Returns if the word is in the trie.
        :type word: str
        :rtype: bool
        """
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.word

    def startsWith(self, prefix):
        """
        Returns if there is any word in the trie that starts with the
        given prefix.
        :type prefix: str
        :rtype: bool
        """
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True

```

```
# Your Trie object will be instantiated and called as such:  
# obj = Trie()  
# obj.insert(word)  
# param_2 = obj.search(word)  
# param_3 = obj.startsWith(prefix)
```

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

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

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

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

Example 1:

```
Input: 2, [[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] .
```

```
'''
```

```
class Solution(object):  
    def findOrder(self, numCourses, prerequisites):  
        """  
        :type numCourses: int  
        :type prerequisites: List[List[int]]  
        :rtype: List[int]  
        """  
        graph = [[] for _ in range(numCourses)]  
        visited = [False for _ in range(numCourses)]  
        stack = [False for _ in range(numCourses)]  
  
        for pair in prerequisites:  
            x, y = pair  
            graph[x].append(y)  
  
        result = []  
        for course in range(numCourses):  
            if visited[course] == False:  
                if self.dfs(graph, visited, stack, course, result):  
                    return []  
            return result  
  
    def dfs(self, graph, visited, stack, course, result):  
        visited[course] = True  
        stack[course] = True  
  
        for neigh in graph[course]:  
            if visited[neigh] == False:  
                if self.dfs(graph, visited, stack, neigh, result):  
                    return True  
  
            elif stack[neigh]:  
                return True  
        stack[course] = False  
        result.append(course)
```

```
return False
```

'''

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 <= 25  
word in addWord consists of lowercase English letters.  
word in search consist of '.' or lowercase English letters.  
There will be at most 3 dots in word for search queries.  
At most 104 calls will be made to addWord and search.  
'''
```

```
class TrieNode:  
    def __init__(self):  
        self.children = {}  
        self.is_word = False  
  
class WordDictionary:  
    def __init__(self):  
        self.root = TrieNode()  
  
    def addWord(self, word: str) -> None:  
        node = self.root  
        for ch in word:  
            if ch not in node.children:  
                node.children[ch] = TrieNode()  
            node = node.children[ch]
```

```

node.is_word = True

def search(self, word: str) -> bool:
    return self._search(self.root, word, 0)

def _search(self, node, word, index):
    if index == len(word):
        return node.is_word

    ch = word[index]
    if ch == '.':
        for child in node.children.values():
            if self._search(child, word, index+1):
                return True
        return False
    else:
        if ch not in node.children:
            return False
        return self._search(node.children[ch], word, index+1)

# Your WordDictionary object will be instantiated and called as such:
if __name__ == '__main__':
    inputs=[[],["bad"],["dad"],["mad"],["pad"],["bad"],[".ad"],["b.."]]
    output=[]
    for i in range(len(inputs)):
        obj = WordDictionary()
        obj.addWord(inputs[i])
        param_2 = obj.search(inputs[i])
        output.append(param_2)

```

```

class TrieNode(object):
    def __init__(self):
        self.value, self.links = None, [None]*26

class Trie(object):
    def __init__(self):
        self.root = TrieNode()
        return

    def insert(self, word):
        if word:
            curr = self.root
            for ch in word:
                offset = ord(ch)-ord('a')
                if curr.links[offset] == None:
                    curr.links[offset] = TrieNode()
                curr = curr.links[offset]
            curr.value = word
        return

class Solution(object):
    def helper(self, x, y, board, trie_node, result):
        if trie_node.value:
            result.add(trie_node.value) # Look for other soln even if a
soln is found. soln could a prefix of another soln.
        for x1,y1 in ((x+1,y), (x-1,y), (x, y+1), (x, y-1)):
            if 0<=x1<len(board) and 0<=y1<len(board[0]) and board[x1][y1]
!= -1 and trie_node.links[ord(board[x1][y1])-ord('a')]:
                ch, board[x1][y1] = board[x1][y1], -1
                self.helper(x1, y1, board, trie_node.links[ord(ch)-
ord('a')], result)
                board[x1][y1] = ch
        return

    def findWords(self, board, words):
        """
        :type board: List[List[str]]
        :type words: List[str]
        :rtype: List[str]
        """
        trie = Trie()
        for word in words:
            trie.insert(word)
        result = set([])
        for i in range(len(board)):
            for j in range(len(board[0])):
                if trie.root.links[ord(board[i][j])-ord('a')]:
                    ch, board[i][j] = board[i][j], -1
                    self.helper(i, j, board, trie.root.links[ord(ch)-
ord('a')], result)
                    board[i][j] = ch
        return [x for x in result]

```

```
'''  
    Find the kth largest element in an unsorted array. Note that it is  
the kth largest element in the sorted order, not the kth distinct  
element.  
  
Example 1:  
  
Input: [3,2,1,5,6,4] and k = 2  
Output: 5  
Example 2:  
  
Input: [3,2,3,1,2,4,5,5,6] and k = 4  
Output: 4  
'''  
  
class Solution(object):  
    def findKthLargest(self, nums, k):  
        """  
        :type nums: List[int]  
        :type k: int  
        :rtype: int  
        """  
        heap = []  
        import heapq  
        for num in nums:  
            heapq.heappush(heap, -(num))  
  
        result = 0  
        for _ in range(k):  
            result = heapq.heappop(heap)  
  
        return -(result)
```

```

import heapq

class f(object):
    def __init__(self, x, h, s):
        self.x = x
        self.h = h
        self.s = s

    def __lt__(self, other):
        if self.x != other.x:
            return self.x < other.x
        else:
            if self.s and other.s:
                return self.h > other.h
            elif not self.s and not other.s:
                return self.h < other.h
            else:
                return self.s > other.s

class Solution(object):
    def getSkyline(self, buildings):
        """
        :type buildings: List[List[int]]
        :rtype: List[List[int]]
        """
        if len(buildings) == 0:
            return []

        building_list = []
        for x in range(len(buildings)):
            building_list.append(f(buildings[x][0], buildings[x][2], 1))
            building_list.append(f(buildings[x][1], buildings[x][2], 0))

        building_list = sorted(building_list)
        for buil in building_list:
            print buil.x, buil.h, buil.s
        heap = [0]
        result = []
        curr_max = heap[0]

        for building in building_list:
            heapq._heapify_max(heap)

            if building.s:
                heap.append(building.h)
                heapq._heapify_max(heap)
                new_max = heap[0]

                if curr_max != new_max:
                    result.append([building.x, building.h])
                    curr_max = new_max
            else:
                heap.remove(building.h)
                heapq._heapify_max(heap)
                new_max = heap[0]

                if new_max != curr_max:
                    result.append([building.x, new_max])

        return result

```

```
curr_max = new_max  
return result
```

```

"""
Invert a binary tree.

Example:

Input:

    4
   / \
  2   7
 / \ / \
1  3 6  9

Output:

    4
   / \
  7   2
 / \ / \
9  6 3  1
"""

# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def invertTree(self, root):
        """
        :type root: TreeNode
        :rtype: TreeNode
        """

        if not root:
            return

        leftTree = self.invertTree(root.left)
        rightTree = self.invertTree(root.right)
        root.left = rightTree
        root.right = leftTree
        return root

# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def invertTree(self, root):
        """
        :type root: TreeNode
        :rtype: TreeNode
        """

```

```
"""
if not root:
    return None

queue = [root]
while queue:
    node = queue.pop(0)
    node.left, node.right = node.right, node.left
    if node.left:
        queue.append(node.left)
    if node.right:
        queue.append(node.right)

return root
```

```
'''  
    Given a binary search tree, write a function kthSmallest to find  
the kth smallest element in it.
```

Note:

You may assume k is always valid,  $1 \leq k \leq \text{BST's total elements}$ .

Example 1:

```
Input: root = [3,1,4,null,2], k = 1  
Output: 1
```

```
'''
```

```
# Definition for a binary tree node.  
# class TreeNode(object):  
#     def __init__(self, x):  
#         self.val = x  
#         self.left = None  
#         self.right = None  
  
class Solution(object):  
    def kthSmallest(self, root, k):  
        """  
        :type root: TreeNode  
        :type k: int  
        :rtype: int  
        """  
  
        if not root:  
            return 0  
  
        stack = [root]  
        count, curr = 0, root  
  
        while stack:  
            if curr.left:  
                stack.append(curr.left)  
                curr = curr.left  
            else:  
                val = stack.pop()  
                count += 1  
                if count == k:  
                    return val.val  
  
                if val.right:  
                    stack.append(val.right)  
                    curr = val.right  
return float('-inf')
```

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def isPalindrome(self, head):
        """
        :type head: ListNode
        :rtype: bool
        """
        rev = None
        slow, fast = head, head.next
        while fast and fast.next:
            fast = fast.next.next
            temp = slow
            slow = slow.next
            temp.next = rev
            rev = temp

        if fast:
            slow = slow.next

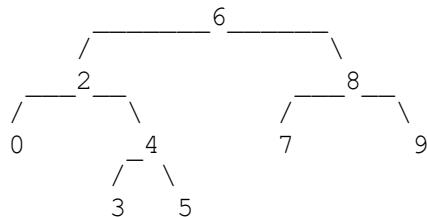
        while rev and rev.val == slow.val:
            rev = rev.next
            slow = slow.next
        return not rev
```

```
'''
```

```
    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)."
```

```
    Given binary search tree:  root = [6,2,8,0,4,7,9,null,null,3,5]
```



```
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
```

```
'''
```

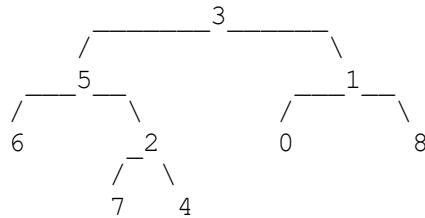
```
# Definition for a binary tree node.  
# class TreeNode(object):  
#     def __init__(self, x):  
#         self.val = x  
#         self.left = None  
#         self.right = None  
  
class Solution(object):  
    def lowestCommonAncestor(self, root, p, q):  
        """  
        :type root: TreeNode  
        :type p: TreeNode  
        :type q: TreeNode  
        :rtype: TreeNode  
        """  
        if not root:  
            return None  
  
        if root.val > p.val and root.val > q.val:  
            return self.lowestCommonAncestor(root.left, p, q)  
        elif root.val < p.val and root.val < q.val:  
            return self.lowestCommonAncestor(root.right, p, q)  
        else:  
            return root
```

```
'''
```

```
    Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.
```

```
    According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow a node to be a descendant of itself)."
```

```
    Given the following binary search tree: root = [3,5,1,6,2,0,8,null,null,7,4]
```



```
Example 1:
```

```
Input: root, p = 5, q = 1
```

```
Output: 3
```

```
Explanation: The LCA of of nodes 5 and 1 is 3.
```

```
'''
```

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def lowestCommonAncestor(self, root, p, q):
        """
        :type root: TreeNode
        :type p: TreeNode
        :type q: TreeNode
        :rtype: TreeNode
        """

        if not root:
            return None

        if root == p or root == q:
            return root

        l = self.lowestCommonAncestor(root.left, p, q)
        r = self.lowestCommonAncestor(root.right, p, q)

        if l and r:
            return root
        return l if l else r
```

```
'''  
    Given an array nums of n integers where n > 1, return an array  
    output such that output[i] is equal to the product of all the elements of  
    nums except nums[i].
```

Example:

```
Input: [1,2,3,4]  
Output: [24,12,8,6]  
      1 1 2 6  
      12 8 6
```

```
'''  
  
class Solution(object):  
    def productExceptSelf(self, nums):  
        """  
        :type nums: List[int]  
        :rtype: List[int]  
        """  
        if not nums:  
            return []  
  
        dp = [1]*len(nums)  
  
        for index in range(1,len(nums)):  
            dp[index] = dp[index-1]*nums[index-1]  
        print dp  
        right = 1  
        for index in range(len(nums)-1, -1, -1):  
            dp[index] *= right  
            right *= nums[index]  
        return dp
```

```
from collections import deque
class Solution(object):
    def maxSlidingWindow(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: List[int]
        """
        if len(nums) == 0:
            return []
        q = deque()
        for i in range(k):
            while q and nums[i] >= nums[q[-1]]:
                q.pop()
            q.append(i)

        result = []
        for i in range(k, len(nums)):
            result.append(nums[q[0]])

            while q and q[0] <= i-k:
                q.popleft()
            while q and nums[i] >= nums[q[-1]]:
                q.pop()

            q.append(i)

        result.append(nums[q[0]])
        return result
```

```
'''
```

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

```
    Integers in each row are sorted in ascending from left to right.
    Integers in each column are sorted in ascending from top to bottom.
    Consider the following matrix:
```

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

```
Example 1:
```

```
Input: matrix, target = 5
Output: true
```

```
'''
```

```
class Solution(object):
    def searchMatrix(self, matrix, target):
        """
        :type matrix: List[List[int]]
        :type target: int
        :rtype: bool
        """

        if not matrix:
            return False

        left, right = 0, len(matrix[0])-1
        while left < len(matrix) and right >= 0:
            if matrix[left][right] == target:
                return True
            elif matrix[left][right] < target:
                left += 1
            else:
                right -= 1
        return False
```

```

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

    For example,
    Given [[0, 30], [5, 10], [15, 20]],
    return 2.
"""

# Definition for an interval.
# class Interval(object):
#     def __init__(self, s=0, e=0):
#         self.start = s
#         self.end = e

class Solution:
    def minMeetingRooms(self, intervals):
        if not intervals or len(intervals) == 0:
            return 0

        import heapq

        sorted_intervals = sorted(intervals, key=lambda it:(it.start,
it.end))
        heap, result = [], 0

        for interval in sorted_intervals:
            start, end = interval.start, interval.end

            while heap and heap[0] <= start:
                heapq.heappop(heap)

            heapq.heappush(heap, end)

            result = max(result, len(heap))
        return result

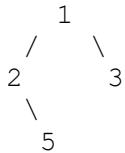
```

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

Note: A leaf is a node with no children.

Example:

Input:



Output: ["1->2->5", "1->3"]

Explanation: All root-to-leaf paths are: 1->2->5, 1->3

```
'''  
  
# Definition for a binary tree node.  
# class TreeNode(object):  
#     def __init__(self, x):  
#         self.val = x  
#         self.left = None  
#         self.right = None  
  
class Solution(object):  
    def binaryTreePaths(self, root):  
        """  
        :type root: TreeNode  
        :rtype: List[str]  
        """  
        if not root:  
            return []  
  
        paths = []  
        def dfs(root, curr):  
            if root.left is None and root.right is None:  
                paths.append(curr + str(root.val))  
                return  
  
            if root.left:  
                dfs(root.left, curr + str(root.val) + '->')  
            if root.right:  
                dfs(root.right, curr + str(root.val) + '->')  
  
        curr = ""  
        dfs(root, curr)  
        return paths
```

```
"""
    Given an array containing n distinct numbers taken from 0, 1, 2,
..., n, find the one that is missing from the array.

Example 1:

Input: [3,0,1]
Output: 2
Example 2:

Input: [9,6,4,2,3,5,7,0,1]
Output: 8
"""

class Solution(object):
    def missingNumber(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        if not nums:
            return 0
        totalSum, n = sum(nums), len(nums)
        expectedSum = (n*(n+1))/2
        return expectedSum - totalSum
```

```
'''  
    Given a positive integer n, find the least number of perfect square  
numbers (for example, 1, 4, 9, 16, ...) which sum to n.
```

Example 1:

```
Input: n = 12  
Output: 3  
Explanation: 12 = 4 + 4 + 4.  
Example 2:
```

```
Input: n = 13  
Output: 2  
Explanation: 13 = 4 + 9.
```

```
'''  
  
class Solution(object):  
    def numSquares(self, n):  
        """  
        :type n: int  
        :rtype: int  
        """  
        mapping = {}  
        squares = [num*num for num in range(1, int(pow(n, 0.5)) + 1)]  
        for square in squares:  
            mapping[square] = 1  
  
        for val in range(1, n+1):  
            if val not in mapping:  
                mapping[val] = float('inf')  
                for square in squares:  
                    if square < val:  
                        mapping[val] = min(mapping[val],  
mapping[square] + mapping[val-square])  
        return mapping[n]
```

```
'''
```

```
Given two 1d vectors, implement an iterator to return their elements  
alternately.
```

```
For example, given two 1d vectors:
```

```
v1 = [1, 2]  
v2 = [3, 4, 5, 6]
```

```
By calling next repeatedly until hasNext returns false, the order of  
elements returned by next should be: [1, 3, 2, 4, 5, 6].
```

```
'''
```

```
class Solution(object):  
    def __init__(self, v1, v2):  
        self.v1 = v1  
        self.v2 = v2  
        self.index_v1 = 0  
        self.index_v2 = 0  
  
    def next(self):  
        result = -1  
        if self.index_v1 != len(self.v1) and self.index_v1 <= self.index_v2:  
            result = self.v1[self.index_v1]  
            self.index_v1 += 1  
        else:  
            result = self.v2[self.index_v2]  
            self.index_v2 += 1  
  
        return result  
  
    def hasNext(self):  
        return self.index_v1 < len(self.v1) or self.index_v2 < len(self.v2)  
  
solution = Solution([1, 2], [3, 4, 5, 6])  
while solution.hasNext():  
    print solution.next()
```

```
'''
```

```
    Given an array nums, write a function to move all 0's to the end of  
it while maintaining the relative order of the non-zero elements.
```

```
Example:
```

```
Input: [0,1,0,3,12]
```

```
Output: [1,3,12,0,0]
```

```
Note:
```

```
You must do this in-place without making a copy of the array.  
Minimize the total number of operations.
```

```
'''
```

```
class Solution(object):  
    def moveZeroes(self, nums):  
        """  
        :type nums: List[int]  
        :rtype: void Do not return anything, modify nums in-place  
instead.  
        """  
        zeroIndex = 0  
        for index in range(len(nums)):  
            if nums[index] != 0:  
                nums[zeroIndex] = nums[index]  
                zeroIndex += 1  
  
        for index in range(zeroIndex, len(nums)):  
            nums[index] = 0
```

```

"""
    Given a binary search tree and a node in it, find the in-order
successor of that node in the BST.
    Note: If the given node has no in-order successor in the tree,
return null
"""

# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def inorderSuccessor(self, root, p):
        """
        :type root: TreeNode
        :type p: TreeNode
        :rtype: TreeNode
        """

        if not root or not p:
            return None

        if p.right:
            p = p.right
            while p.left:
                p = p.left
            return p

        successor = None
        while root and root != p:
            if root.val > p.val:
                successor = root
                root = root.left
            else:
                root = root.right
        return successor

```

```
'''
```

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

```
Example 1:
```

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

```
Output: 2
```

```
Example 2:
```

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

```
Output: 3
```

```
'''
```

```
class Solution(object):
    def findDuplicate(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        slow, fast = nums[0], nums[0]
        while True:
            slow = nums[slow]
            fast = nums[nums[fast]]
            if slow == fast:
                break

        num1= nums[0]
        num2 = slow
        while num1 != num2:
            num1 = nums[num1]
            num2 = nums[num2]
        return num2
```

```
class Solution(object):
    def gameOfLife(self, board):
        """
        :type board: List[List[int]]
        :rtype: void Do not return anything, modify board in-place
instead.
        """
        index = []

        def around(i, j, board):
            count = 0
            for k in range(i-1, i+2):
                for l in range(j-1, j+2):
                    if 0<=k < len(board) and 0 <= l < len(board[0]):
                        if board[k][l] == 1:
                            count += 1

            return count-1 if board[i][j] == 1 else count

        for i in range(len(board)):
            for j in range(len(board[0])):
                count = around(i, j, board)
                if board[i][j] == 1:
                    if count > 3 or count < 2:
                        index.append([i, j, 0])
                else:
                    if count == 3:
                        index.append([i, j, 1])

        while index:
            i, j, value = index.pop()
            board[i][j] = value
```

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

Examples:

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

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

Design a data structure that supports the following two operations:

```
void addNum(int num) - Add a integer number from the data stream to  
the data structure.
```

```
double findMedian() - Return the median of all elements so far.
```

For example:

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

```
'''
```

```
import heapq
```

```
class MedianFinder(object):
```

```
def __init__(self):
```

```
    """  
    initialize your data structure here.  
    """
```

```
    self.max_heap = []  
    self.min_heap = []
```

```
def addNum(self, num):
```

```
    """
```

```
    :type num: int  
    :rtype: void  
    """
```

```
    if not self.max_heap or num > -self.max_heap[0]:  
        heapq.heappush(self.min_heap, num)
```

```
    if len(self.min_heap) > len(self.max_heap) + 1:
```

```
        heapq.heappush(self.max_heap, -
```

```
heapq.heappop(self.min_heap))
```

```
    else:
```

```
        heapq.heappush(self.max_heap, -num)
```

```
        if len(self.max_heap) > len(self.min_heap):
```

```
            heapq.heappush(self.min_heap, -
```

```
heapq.heappop(self.max_heap))
```

```
def findMedian(self):
```

```
    """
```

```
    :rtype: float  
    """
```

```
    print self.max_heap, self.min_heap
```

```
    if len(self.max_heap) == len(self.min_heap):
```

```
        return (-self.max_heap[0]+self.min_heap[0] )/2.0
```

```
        else:
            return self.min_heap[0]

# Your MedianFinder object will be instantiated and called as such:
# obj = MedianFinder()
# obj.addNum(num)
# param_2 = obj.findMedian()
```

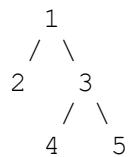
```
'''
```

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.

Example:

You may serialize the following tree:



as "[1,2,3,null,null,4,5]"

```
'''
```

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Codec:

    def serialize(self, root):
        """Encodes a tree to a single string.

        :type root: TreeNode
        :rtype: str
        """

        def preorder(root):
            if root:
                serializeTree.append(str(root.val) + ',')
                preorder(root.left)
                preorder(root.right)
            else:
                serializeTree.append('#,')

        serializeTree = []
        preorder(root)
        return ''.join(serializeTree)

    def deserialize(self, data):
        """Decodes your encoded data to tree.

        :type data: str
        """
```

```
:rtype: TreeNode
"""

def buildTree(preorder):
    value = preorder.pop(0)
    if value == '#':
        return None

    node = TreeNode(int(value))
    node.left = buildTree(preorder)
    node.right = buildTree(preorder)
    return node

preorder = data.split(',')[:-1]
return buildTree(preorder)

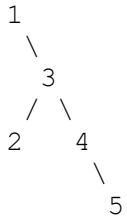
# Your Codec object will be instantiated and called as such:
# codec = Codec()
# codec.deserialize(codec.serialize(root))
```

```
'''
```

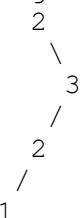
```
Given a binary tree, find the length of the longest consecutive sequence path.
```

The path refers to any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The longest consecutive path need to be from parent to child (cannot be the reverse).

For example,



Longest consecutive sequence path is 3-4-5, so return 3.



Longest consecutive sequence path is 2-3, not 3-2-1, so return 2.

```
'''
```

```
class Solution(object):
    def dfs(curr, parent, length):
        if not curr:
            return length
        if parent:
            length = length + 1 if curr.val == parent.val + 1
        else:
            length = 1

        return max(length, max(dfs(curr.left, curr, length),
                               dfs(curr.right, curr, length)))

    def longestConsecutive(TreeNode root):
        if not root:
            return 0

        return dfs(root, null, 0)
```

```

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

For example,
Given [10, 9, 2, 5, 3, 7, 101, 18],
The longest increasing subsequence is [2, 3, 7, 101], therefore the
length is 4. Note that there may be more than one LIS combination, it is
only necessary for you to return the length.

Your algorithm should run in O(n2) complexity.

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

class Solution(object):
    def lengthOfLIS(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """

        if len(nums) <= 1:
            return len(nums)

        count = [0 for _ in range(len(nums))]
        result = 1
        count[0] = nums[0]

        for index in range(1, len(nums)):
            if nums[index] < count[0]:
                count[0] = nums[index]
            elif nums[index] > count[result-1]:
                count[result] = nums[index]
                result += 1
            else:
                left, right = -1, result-1
                while (right-left > 1):
                    mid = (left+right)/2
                    if count[mid] >= nums[index]:
                        right = mid
                    else:
                        left = mid
                count[right] = nums[index]

        return result

```

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

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

Example 1:

```
Input: "()()()"  
Output: ["()", "(())()"]
```

Example 2:

```
Input: "(a)()()"  
Output: ["(a)()", "(a())()"]
```

Example 3:

```
Input: ")()  
Output: [""]
```

```
'''  
  
class Solution(object):  
    def removeInvalidParentheses(self, s):  
        """  
        :type s: str  
        :rtype: List[str]  
        """  
        if not s:  
            return [""]  
  
        def isValid(s):  
            count = 0  
            for char in s:  
                if char == '(':  
                    count += 1  
                elif char == ')':  
                    count -= 1  
                if count < 0:  
                    return False  
            return (count==0)  
  
        queue, result = [s], []  
        visited = set()  
        visited.add(s)  
        level = False  
  
        while queue:  
            new_str = queue.pop(0)  
            if isValid(new_str):  
                result.append(new_str)  
                level = True  
  
            if level:  
                continue  
  
            for index in range(len(new_str)):  
                if not (new_str[index] == "(" or new_str[index] == ")"):  
                    continue
```

```
partition_str = new_str[0:index] + new_str[index+1:]
if partition_str not in visited:
    queue.append(partition_str)
    visited.add(partition_str)
return result
```

```
'''  
Given an integer array nums, find the sum of the elements between indices  
i and j ( $i \leq j$ ), inclusive.
```

The update( $i$ ,  $val$ ) function modifies  $nums$  by updating the element at index  $i$  to  $val$ .

Example:

Given  $nums = [1, 3, 5]$

```
sumRange(0, 2) -> 9  
update(1, 2)  
sumRange(0, 2) -> 8
```

Note:

The array is only modifiable by the update function.

You may assume the number of calls to update and sumRange function is distributed evenly.

```
'''  
  
class Node(object):  
    def __init__(self, val, start, end):  
        self.sum = val  
        self.right, self.left = None, None  
        self.range = [start, end]  
  
class SegementTree(object):  
    def __init__(self, size):  
        self.root = self._build_segment_tree(0, size-1)  
  
    def _build_segment_tree(self, start, end):  
        if start > end:  
            return None  
        node = Node(0, start, end)  
        if start == end:  
            return node  
        mid = (start+end)/2  
        node.left, node.right = self._build_segment_tree(start, mid),  
        self._build_segment_tree(mid+1, end)  
        return node  
  
    def update(self, index, val, root=None):  
        root = root or self.root  
        if index < root.range[0] or index > root.range[1]:  
            return  
        root.sum += val  
        if index == root.range[0] == root.range[1]:  
            return  
        self.update(index, val, root.left)  
        self.update(index, val, root.right)  
  
    def range_sum(self, start, end, root=None):  
        root = root or self.root  
        if end < root.range[0] or start > root.range[1]:  
            return 0  
        if start <= root.range[0] and end >= root.range[1]:  
            return root.sum
```

```

        return self.range_sum(start, end, root.left) +
self.range_sum(start, end, root.right)

class NumArray(object):

    def __init__(self, nums):
        """
        :type nums: List[int]
        """
        self.nums = nums
        self.segment_tree = SegementTree(len(nums))
        for index, num in enumerate(nums):
            self.segment_tree.update(index, num)

    def update(self, i, val):
        """
        :type i: int
        :type val: int
        :rtype: None
        """
        diff = val-self.nums[i]
        self.segment_tree.update(i, diff)
        self.nums[i] = val

    def sumRange(self, i, j):
        """
        :type i: int
        :type j: int
        :rtype: int
        """
        return self.segment_tree.range_sum(i, j)

# Your NumArray object will be instantiated and called as such:
# obj = NumArray(nums)
# obj.update(i,val)
# param_2 = obj.sumRange(i,j)

```

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

Example:

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

```
To the right of 5 there are 2 smaller elements (2 and 1).  
To the right of 2 there is only 1 smaller element (1).  
To the right of 6 there is 1 smaller element (1).  
To the right of 1 there is 0 smaller element.  
Return the array [2, 1, 1, 0].
```

```
'''
```

```
class TreeNode(object):  
    def __init__(self, val):  
        self.right = None  
        self.left = None  
        self.val = val  
        self.count = 1  
  
class Solution(object):  
    def countSmaller(self, nums):  
        """  
        :type nums: List[int]  
        :rtype: List[int]  
        """  
        if len(nums) == 0:  
            return []  
  
        node = TreeNode(nums[len(nums)-1])  
        result = [0]  
        for index in range(len(nums)-2, -1, -1):  
            result.append(self.insertNode(node, nums[index]))  
  
        return result[::-1]  
  
    def insertNode(self, node, val):  
        totalCount = 0  
        while True:  
            if val <= node.val:  
                node.count += 1  
                if node.left is None:  
                    node.left = TreeNode(val)  
                    break  
                else:  
                    node = node.left  
            else:  
                totalCount += node.count  
                if node.right is None:  
                    node.right = TreeNode(val)  
                    break  
                else:  
                    node = node.right  
  
        return totalCount
```



'''

You want to build a house on an empty land which reaches all buildings in the shortest amount of distance. You can only move up, down, left and right. You are given a 2D grid of values 0, 1 or 2, where:

Each 0 marks an empty land which you can pass by freely.

Each 1 marks a building which you cannot pass through.

Each 2 marks an obstacle which you cannot pass through.

For example, given three buildings at (0,0), (0,4), (2,2), and an obstacle at (0,2):

```
1 - 0 - 2 - 0 - 1
|   |   |   |   |
0 - 0 - 0 - 0 - 0
|   |   |   |   |
0 - 0 - 1 - 0 - 0
```

The point (1,2) is an ideal empty land to build a house, as the total travel distance of  $3+3+1=7$  is minimal. So return 7.

Note:

There will be at least one building. If it is not possible to build such house according to the above rules, return -1.

'''

```
class Solution(object):
    def shortestDistance(self, grid):
        if not grid:
            return -1

        def bfs(grid, distance_reach_map, row, col):
            if(row < 0 or row > len(grid) or col < 0 or col >
len(grid[0])):
                return
            queue = [[row, col]]
            qdist = [1]

            direction = [[-1, 0], [0, 1], [1, 0], [0, -1]]
            while queue:
                x, y = queue.pop(0)
                curr_dist = qdist.pop(0)

                for dx, dy in direction:
                    new_x, new_y = x+dx, y+dy
                    if((0 <= new_x < len(grid)) and (0 <= new_y
< len(grid[0])) and grid[new_x][new_y] == 0):
                        grid[new_x][new_y] = -1
                        queue.append([new_x, new_y])

                        temp = distance_reach_map[new_x][new_y]
                        dist, reach = temp[0], temp[1]
                        dist += curr_dist
                        reach += 1
                        distance_reach_map[new_x][new_y] =
[dist, reach]
                        qdist.append(curr_dist+1)
```

```

        for row in range(len(grid)):
            for col in range(len(grid[0])):
                if grid[row][col] == -1:
                    grid[row][col] =0

r_len, c_len = len(grid), len(grid[0])
distance_reach_map = [[[0, 0]]*c_len for _ in range(r_len)]
buildings = 0
for row in range(len(grid)):
    for col in range(len(grid[0])):
        if grid[row][col] == 1:
            bfs(grid, distance_reach_map, row, col)
            buildings += 1

result = float('inf')
for row in range(r_len):
    for col in range(c_len):
        dist, reach = distance_reach_map[row][col]
        if reach == buildings:
            result = min(result, dist)
return result

solution = Solution()
grid = [[1, 0, 2, 0, 1],
        [0, 0, 0, 0, 0],
        [0, 0, 1, 0 ,0]]
print solution.shortestDistance(grid)

```

```
'''
```

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

```
Example 1:
```

```
coins = [1, 2, 5], amount = 11  
return 3 (11 = 5 + 5 + 1)
```

```
Example 2:
```

```
coins = [2], amount = 3  
return -1.
```

```
'''
```

```
class Solution(object):  
    def coinChange(self, coins, amount):  
        """  
        :type coins: List[int]  
        :type amount: int  
        :rtype: int  
        """  
        if not coins:  
            return 0  
  
        dp = [float('inf') for _ in range(amount+1)]  
        dp[0] = 0  
  
        for val in range(1, amount+1):  
            for coin in coins:  
                if coin <= val:  
                    dp[val] = min(dp[val-coin]+1, dp[val])  
        return dp[amount] if dp[amount] != float('inf') else -1
```

```
'''  
    Given an integer, write a function to determine if it is a power of  
three.  
'''
```

Follow up:

Could you do it without using any loop / recursion?

```
'''  
class Solution(object):  
    def isPowerOfThree(self, n):  
        """  
        :type n: int  
        :rtype: bool  
        """  
        if n <= 0:  
            return False  
  
        import math  
        return (math.log10(n)/math.log10(3))%1 == 0
```

```
'''
```

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

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

```
Example:
```

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

```
'''
```

```
# Definition for singly-linked list.  
# class ListNode(object):  
#     def __init__(self, x):  
#         self.val = x  
#         self.next = None  
  
class Solution(object):  
    def oddEvenList(self, head):  
        """  
        :type head: ListNode  
        :rtype: ListNode  
        """  
        if not head:  
            return None  
  
        odd, even = head, head.next  
        evenHead = even  
        while even and even.next:  
            odd.next = even.next  
            odd = odd.next  
            even.next = odd.next  
            even = even.next  
  
        odd.next = evenHead  
        return head
```

```

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

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

Example 1:

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

class Solution(object):
    def longestIncreasingPath(self, matrix):
        """
        :type matrix: List[List[int]]
        :rtype: int
        """
        result = 0
        dp = [[0 for col in range(len(matrix[0]))] for row in
range(len(matrix))]
        for row in range(len(matrix)):
            for col in range(len(matrix[0])):
                result = max(result, self.dfs(matrix, dp, row, col))

        return result

    def dfs(self, matrix, dp, i, j):
        if dp[i][j]:
            return dp[i][j]
        max_depth = 0
        direction = [[0, 1], [0, -1], [1, 0], [-1, 0]]
        for d in direction:
            x, y = i + d[0], j + d[1]
            if 0 <= x < len(matrix) and 0 <= y < len(matrix[0]) and
matrix[x][y] < matrix[i][j] :
                max_depth = max(max_depth, self.dfs(matrix, dp, x, y))

        dp[i][j] = max_depth + 1
        return dp[i][j]

```

```
'''
```

```
    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:
```

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

```
Output: [ "JFK", "MUC", "LHR", "SFO", "SJC"]
```

```
'''
```

```
from collections import defaultdict  
class Solution(object):  
    def findItinerary(self, tickets):  
        """  
        :type tickets: List[List[str]]  
        :rtype: List[str]  
        """  
        n = len(tickets)  
        trips = defaultdict(list)  
        for x in tickets:  
            trips[x[0]].append(x[1])  
        for x in trips:  
            trips[x].sort()  
        iter = ["JFK"]  
  
        def dfs(curr_iter):  
            if len(curr_iter) == n+1:  
                return curr_iter  
            curr_stop = curr_iter[-1]  
  
            if trips[curr_stop] == []:  
                return None  
  
            next_stops = trips[curr_stop]  
            i = 0  
            for stop in next_stops:  
                curr_iter.append(stop)  
                del trips[curr_stop][i]  
  
                if dfs(curr_iter):  
                    return curr_iter  
  
                curr_iter.pop()  
                trips[curr_stop].insert(i, stop)  
                i += 1  
        return None
```

```
return dfs(iterator)
```

```
'''  
    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.
```

```
'''
```

```
class Solution(object):  
    def increasingTriplet(self, nums):  
        """  
        :type nums: List[int]  
        :rtype: bool  
        """  
  
        first, second = float('inf'), float('inf')  
        for val in nums:  
            if val <= first:  
                first = val  
            elif val <= second:  
                second = val  
            else:  
                return True  
  
        return False
```

```

"""
Given a string, find the longest substring that contains only two unique
characters. For example, given "abcbbbbcccbdddadacb", the longest
substring that contains 2 unique character is "bcbbbbccb".
"""

class Solution(object):
    def lengthOfLongestSubstringKDistinct(self, S, K):
        charMapping, start = {}, 0
        result = 0
        for end, s in enumerate(S):
            if s in charMapping:
                charMapping[s] += 1
            else:
                charMapping[s] = 1

            if len(charMapping) <= K:
                result = max(result, end-start+1)
            else:
                while len(charMapping) > K :
                    character = S[start]
                    freq = charMapping[character]
                    if freq == 1:
                        del charMapping[character]
                    else:
                        charMapping[character] -= 1
                    start += 1

        return result

if __name__ == '__main__':
    print Solution().lengthOfLongestSubstringKDistinct("abcadacacacaca",
3)

```

```
'''  
Given a stream of integers and a window size, calculate the moving  
average of all integers in the sliding window.
```

For example,

```
MovingAverage m = new MovingAverage(3);  
m.next(1) = 1  
m.next(10) = (1 + 10) / 2  
m.next(3) = (1 + 10 + 3) / 3  
m.next(5) = (10 + 3 + 5) / 3  
'''
```

```
class Solution(object):  
    def __init__(self):  
        self.queue = []  
        self.curr_sum = 0  
  
    def movingAverage(self, num, size):  
        if len(self.queue) >= size:  
            val = self.queue.pop(0)  
            self.curr_sum -= val  
  
        self.curr_sum += num  
        self.queue.append(num)  
        return float(self.curr_sum)/len(self.queue)
```

```
solution = Solution()  
window_size = int(input())  
num = int(input())  
while num != -1:  
    print solution.movingAverage(num, window_size)  
    num = int(input())
```

```
"""
    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]
"""

class Solution(object):
    def topKFrequent(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: List[int]
        """

        if not nums:
            return []
        frequency = {}
        for num in nums:
            if num in frequency:
                frequency[num] += 1
            else:
                frequency[num] = 1

        result = []
        import heapq
        heap = []

        for key, value in frequency.iteritems():
            heapq.heappush(heap, (-value, key))

        for _ in range(k):
            result.append(heapq.heappop(heap)[1])
        return result
```

'''

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?

'''

```
class Solution(object):
    def intersect(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :rtype: List[int]
        """

        nums1.sort()
        nums2.sort()

        index_i, index_j = 0, 0
        result = []
        while index_i < len(nums1) and index_j < len(nums2):
            if nums1[index_i] == nums2[index_j]:
                result.append(nums1[index_i])
                index_i += 1
                index_j += 1
            elif nums1[index_i] > nums2[index_j]:
                index_j += 1
            else:
                index_i += 1
        return result
```

'''

Given an Android 3x3 key lock screen and two integers m and n, where  $1 \leq m \leq n \leq 9$ , count the total number of unlock patterns of the Android lock screen, which consist of minimum of m keys and maximum n keys.

Rules for a valid pattern:

Each pattern must connect at least m keys and at most n keys.

All the keys must be distinct.

If the line connecting two consecutive keys in the pattern passes through any other keys, the other keys must have previously selected in the pattern. No jumps through non selected key is allowed.

The order of keys used matters.

Explanation:

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

Invalid move: 4 - 1 - 3 - 6

Line 1 - 3 passes through key 2 which had not been selected in the pattern.

Invalid move: 4 - 1 - 9 - 2

Line 1 - 9 passes through key 5 which had not been selected in the pattern.

Valid move: 2 - 4 - 1 - 3 - 6

Line 1 - 3 is valid because it passes through key 2, which had been selected in the pattern

Valid move: 6 - 5 - 4 - 1 - 9 - 2

Line 1 - 9 is valid because it passes through key 5, which had been selected in the pattern.

Example:

Given m = 1, n = 1, return 9.

'''

```
class Solution(object):
    def numberOfPatterns(self, m, n):
        def dfs(reamin, current, visited, skip):
            if reamin < 0:
                return 0
            if reamin == 0:
                return 1
            visited[current] = True
            result = 0
            for index in range(1, 10):
                if not visited[index] and (skip[current][index] ==
0 || visited[skip[current][index]])
                    result += dfs(reamin-1, index, visited,
skip)
            visited[current] = False
            return result

        skip = [[0 for _ in range(10)] for _ in range(10)]
        skip[1][3] = skip[3][1] = 2
        skip[1][7] = skip[7][1] = 4
        skip[3][9] = skip[9][3] = 6
```

```
skip[7][9] = skip[9][7] = 8
skip[1][9] = skip[9][1] = skip[2][8] = skip[8][2] = skip[3][7] =
skip[7][3] = skip[4][6] = skip[6][4] = 5
visited = [False for _ in range(10)]

result = 0
for index in range(m, n):
    result += dfs(index-1, 1, visited, skip)
    result += dfs(index-1, 2, visited, skip)
    result += dfs(index-1, 5, visited, skip)
return result
```

```
'''  
Given a 2D grid, each cell is either a wall 'W', an enemy 'E' or empty  
'0' (the number zero), return the maximum enemies you can kill using one  
bomb.
```

```
The bomb kills all the enemies in the same row and column from the  
planted point until it hits the wall since the wall is too strong to be  
destroyed.
```

```
Note that you can only put the bomb at an empty cell.
```

Example:

```
For the given grid  
0 E 0 0  
E 0 W E  
0 E 0 0  
return 3. (Placing a bomb at (1,1) kills 3 enemies)
```

```
class Solution(object):  
    def maxKilledEnemies(self, grid):  
        if not grid or len(grid) == 0 or len(grid[0]) == 0:  
            return 0  
  
        result, row_count = float('-inf'), 0  
        column_count = [0]*len(grid[0])  
        for row in range(len(grid)):  
            for column in range(len(grid[0])):  
                if column == 0 or grid[row][column-1] == 'W':  
                    row_count = 0  
                for index in range(column, len(grid[0])):  
                    if grid[row][index] == 'W':  
                        break  
                    row_count += 1 if grid[row][index] ==  
'E' else 0  
  
                if row == 0 or grid[row-1][column] == 'W':  
                    column_count[column] = 0  
                for index in range(row, len(grid)):  
                    if grid[index][column] == 'W':  
                        break  
                    column_count[column] += 1 if  
grid[index][column] == 'E' else 0  
  
                if grid[row][column] == '0':  
                    result = max(result, row_count +  
column_count[column])  
        return result  
  
solution = Solution()  
grid = [['0', 'E', '0', '0'],  
['E', '0', 'W', 'E'],  
['0', 'E', '0', '0']]  
print solution.maxKilledEnemies(grid)
```

```

"""
    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.
"""

class Solution(object):
    def kthSmallest(self, matrix, k):
        """
        :type matrix: List[List[int]]
        :type k: int
        :rtype: int
        """

        if not matrix:
            return 0

        import heapq
        heap = []
        for col in range(len(matrix[0])):
            heapq.heappush(heap, (matrix[0][col], 0, col))

        val = 0
        for index in range(k):
            val, row, col = heapq.heappop(heap)
            new_val = float('inf')
            if row < len(matrix)-1:
                new_val = matrix[row+1][col]
            heapq.heappush(heap, (new_val, row+1, col))
        return val

```

```

"""
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();
"""

class RandomizedSet(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.values = []
        self.positions = {}

    def insert(self, val):
        """
        Inserts a value to the set. Returns true if the set did not
already contain the specified element.
        :type val: int
        :rtype: bool
        """
        if val not in self.positions:
            self.values.append(val)
            self.positions[val] = len(self.values) - 1
        return True

```

```

        return False

    def remove(self, val):
        """
            Removes a value from the set. Returns true if the set contained
            the specified element.
            :type val: int
            :rtype: bool
        """
        if val in self.positions:
            valIdx, lastEle = self.positions[val], self.values[-1]
            self.positions[lastEle], self.values[valIdx] = valIdx,
lastEle
            self.values.pop()
            self.positions.pop(val, 0)
            return True
        return False

    def getRandom(self):
        """
            Get a random element from the set.
            :rtype: int
        """
        return self.values[random.randint(0, len(self.values)-1)]

# Your RandomizedSet object will be instantiated and called as such:
# obj = RandomizedSet()
# param_1 = obj.insert(val)
# param_2 = obj.remove(val)
# param_3 = obj.getRandom()

```

```
'''  
    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.
```

```
'''
```

```
class Solution(object):  
    def firstUniqChar(self, s):  
        """  
        :type s: str  
        :rtype: int  
        """  
        letters='abcdefghijklmnopqrstuvwxyz'  
        index=[s.index(l) for l in letters if s.count(l) == 1]  
        return min(index) if len(index) > 0 else -1
```

```
'''
```

Suppose we abstract our file system by a string in the following manner:

The string "dir\n\tsubdir1\n\tsubdir2\n\t\tfile.ext" represents:

```
dir
    subdir1
    subdir2
        file.ext
```

The directory dir contains an empty sub-directory subdir1 and a sub-directory subdir2 containing a file file.ext.

The string  
"dir\n\tsubdir1\n\t\tfile1.ext\n\t\tsubsubdir1\n\t\tsubdir2\n\t\t\tsubsubdir2  
\n\t\t\tfile2.ext" represents:

```
dir
    subdir1
        file1.ext
        subsubdir1
    subdir2
        subsubdir2
            file2.ext
```

The directory dir contains two sub-directories subdir1 and subdir2. subdir1 contains a file file1.ext and an empty second-level sub-directory subsubdir1. subdir2 contains a second-level sub-directory subsubdir2 containing a file file2.ext.

We are interested in finding the longest (number of characters) absolute path to a file within our file system. For example, in the second example above, the longest absolute path is "dir/subdir2/subsubdir2/file2.ext", and its length is 32 (not including the double quotes).

Given a string representing the file system in the above format, return the length of the longest absolute path to file in the abstracted file system. If there is no file in the system, return 0.

Note:

The name of a file contains at least a . and an extension.

The name of a directory or sub-directory will not contain a ..

Time complexity required: O(n) where n is the size of the input string.

Notice that a/aa/aaa/file1.txt is not the longest file path, if there is another pathaaaaaaaaaaaaaaa/sth.png

```
'''
```

```
class Solution(object):
    def lengthLongestPath(self, input):
        """
        :type input: str
        :rtype: int
        """
        if not input:
            return 0
        directories = input.split('\n')
        stack = [[-1, 0]] # \t level, total dir length
        result = 0
        for direct in directories:
```

```
n_tabs = direct.count('\t')
while stack and stack[-1][0] >= n_tabs:
    stack.pop()
if "." in direct:
    result = max(result, stack[-1][1] + len(direct)-n_tabs)
stack.append([n_tabs, stack[-1][1] + len(direct) + 1 -
n_tabs])
return result
```

'''

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 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 (hexadecimal)	UTF-8 octet sequence (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 3-bytes character.

The next byte is a continuation byte which starts with 10 and that's correct.

But the second continuation byte does not start with 10, so it is invalid.

'''

```
class Solution(object):  
    def validUtf8(self, data):  
        """  
        :type data: List[int]  
        :rtype: bool  
        """  
        seventh_mask = 1 << 7  
        sixth_mask = 1 << 6  
        no_bytes = 0  
  
        if len(data) == 1:  
            return not(data[0] & seventh_mask)
```

```
for num in data:
    if no_bytes == 0:
        mask = 1 << 7

        while num & mask:
            no_bytes = 1
            mask >= 1

    if no_bytes == 0:
        continue

    if no_bytes == 1 or no_bytes > 4:
        return False
    else:
        if not(num & seveneth_mask and not(num & sixth_mask)):
            return False
        no_bytes -= 1
return no_bytes == 0
```

```
'''  
    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.

```
'''  
  
class Solution(object):  
    def longestSubstring(self, s, k):  
        """  
        :type s: str  
        :type k: int  
        :rtype: int  
        """  
        dict = {}  
        for c in s:  
            if c not in dict:  
                dict[c] = 0  
            dict[c] += 1  
        if all(dict[i] >= k for i in dict):  
            return len(s)  
  
        longest = 0  
        start = 0  
        for i in range(len(s)):  
            c = s[i]  
            if dict[c] < k:  
                longest = max(longest, self.longestSubstring(s[start:i],  
k))  
                start = i + 1  
  
        return max(longest, self.longestSubstring(s[start:], k))
```

'''

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:

$1 \leq n \leq 1000$   
 $1 \leq m \leq \min(50, n)$

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.

'''

```
class Solution(object):
    def splitArray(self, nums, m):
        """
        :type nums: List[int]
        :type m: int
        :rtype: int
        """

        left, right = max(nums), sum(nums)

        while left < right:
            mid = left + ((right-left) >> 1)
            curr_sum, invalid, groups = 0, True, 0
            for num in nums:
                if num > mid:
                    invalid = False
                    break
                if num + curr_sum > mid:
                    groups += 1
                    curr_sum = 0
                curr_sum += num
            if invalid and groups < m:
                right = mid
            else:
                left = mid + 1
        return left
```

'''

Given a rows x cols screen and a sentence represented by a list of non-empty words, find how many times the given sentence can be fitted on the screen.

Note:

A word cannot be split into two lines.

The order of words in the sentence must remain unchanged.

Two consecutive words in a line must be separated by a single space.

Total words in the sentence won't exceed 100.

Length of each word is greater than 0 and won't exceed 10.

$1 \leq \text{rows}, \text{cols} \leq 20,000$ .

Example 1:

Input:

```
rows = 2, cols = 8, sentence = ["hello", "world"]
```

Output:

1

Explanation:

hello---

world---

The character '--' signifies an empty space on the screen.

Example 2:

Input:

```
rows = 3, cols = 6, sentence = ["a", "bcd", "e"]
```

Output:

2

Explanation:

a-bcd-

e-a---

bcd-e-

The character '--' signifies an empty space on the screen.

Example 3:

Input:

```
rows = 4, cols = 5, sentence = ["I", "had", "apple", "pie"]
```

Output:

1

Explanation:

I-had

apple

pie-I

had--

The character '-' signifies an empty space on the screen.

'''

```
class Solution(object):
```

```
def wordsTyping(self, sentences, rows, cols):
    """
    :sentences List<String>
    :rows int
    :cols int
    """
    sentence = '-'.join(sentences)
    sentence += '-'

    index_in_sentence = 0
    for row in range(rows):
        index_in_sentence += cols
        if sentence[(index_in_sentence%len(sentence))] == '-':
            index_in_sentence += 1
        else:
            while index_in_sentence > 0 and
sentence[((index_in_sentence - 1)%len(sentence))] != '-':
                index_in_sentence -= 1

    return index_in_sentence/len(sentence)

solution = Solution()
row, col = 3, 6
sentences = ["a", "bcd", "e"]
print solution.wordsTyping(sentences=sentences, rows=row, cols=col)
```

```
'''  
420. Strong Password Checker
```

A password is considered strong if the below conditions are all met:

It has at least 6 characters and at most 20 characters.  
It contains at least one lowercase letter, at least one uppercase letter,  
and at least one digit.  
It does not contain three repeating characters in a row (i.e., "Baaabb0"  
is weak, but "Baaba0" is strong).  
Given a string password, return the minimum number of steps required to  
make password strong. If password is already strong, return 0.

In one step, you can:

Insert one character to password,  
Delete one character from password, or  
Replace one character of password with another character.

Example 1:

Input: password = "a"  
Output: 5  
Example 2:

Input: password = "aA1"  
Output: 3  
Example 3:

Input: password = "1337C0d3"  
Output: 0

Constraints:

```
1 <= password.length <= 50
password consists of letters, digits, dot '.' or exclamation mark '!'.
'''
```

```
class Solution:
    def strongPasswordChecker(self, s: str) -> int:
        n = len(s)
        chars = list(s)
        missing = self.getMissing(chars)
        replaces = 0
        oneSeq = 0
        twoSeq = 0

        i = 2
        while i < n:
            if chars[i] == chars[i - 1] and chars[i - 1] == chars[i - 2]:
                length = 2
                while i < n and chars[i] == chars[i - 1]:
                    length += 1
                    i += 1
                replaces += length // 3
                if length % 3 == 0:
```

```

        oneSeq += 1
    if length % 3 == 1:
        twoSeq += 1
    else:
        i += 1

    if n < 6:
        return max(6 - n, missing)
    if n <= 20:
        return max(replaces, missing)

    deletes = n - 20
    replaces -= min(oneSeq, deletes)
    replaces -= min(max(deletes - oneSeq, 0), twoSeq * 2) // 2
    replaces -= max(deletes - oneSeq - twoSeq * 2, 0) // 3
    return deletes + max(replaces, missing)

def getMissing(self, chars):
    missing = 3
    for c in chars:
        if c.isupper():
            missing -= 1
            break
    for c in chars:
        if c.islower():
            missing -= 1
            break
    for c in chars:
        if c.isdigit():
            missing -= 1
            break
    return missing

```

```
'''
```

```
Given an array of integers,  $1 \leq a[i] \leq n$  ( $n = \text{size of array}$ ), 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(object):
    def findDuplicates(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        if not nums:
            return []
        result = []
        for _, num in enumerate(nums):
            index = abs(num) - 1
            if nums[index] < 0:
                result.append(index + 1)
            nums[index] *= -1
        return result
```

```
'''  
Given an array of integers where  $1 \leq a[i] \leq n$  ( $n = \text{size of array}$ ), 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]  
'''

```
class Solution(object):  
    def findDisappearedNumbers(self, nums):  
        """  
        :type nums: List[int]  
        :rtype: List[int]  
        """  
        if not nums:  
            return []  
        result = []  
        for num in nums:  
            index = abs(num) - 1  
            if nums[index] > 0:  
                nums[index] *= -1  
        for index, num in enumerate(nums):  
            if num > 0:  
                result.append(index+1)  
        return result
```

```
class Solution(object):
    def fourSumCount(self, A, B, C, D):
        """
        :type A: List[int]
        :type B: List[int]
        :type C: List[int]
        :type D: List[int]
        :rtype: int
        """
        hashTable ={}

        for a in A:
            for b in B:
                if a+b in hashTable:
                    hashTable[a+b] += 1
                else:
                    hashTable[a+b] = 1

        result = 0
        for c in C:
            for d in D:
                if -(c+d) in hashTable:
                    result += hashTable[-(c+d)]
        return result
```

```
'''
```

You are given a license key represented as a string S which consists only alphanumeric character and dashes. The string is separated into N+1 groups by N dashes.

Given a number K, we would want to reformat the strings such that each group contains exactly K characters, except for the first group which could be shorter than K, but still must contain at least one character. Furthermore, there must be a dash inserted between two groups and all lowercase letters should be converted to uppercase.

Given a non-empty string S and a number K, format the string according to the rules described above.

Example 1:

Input: S = "5F3Z-2e-9-w", K = 4

Output: "5F3Z-2E9W"

Explanation: The string S has been split into two parts, each part has 4 characters.

Note that the two extra dashes are not needed and can be removed.

Example 2:

Input: S = "2-5g-3-J", K = 2

Output: "2-5G-3J"

Explanation: The string S has been split into three parts, each part has 2 characters except the first part as it could be shorter as mentioned 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.

```
'''
```

```
class Solution(object):
    def licenseKeyFormatting(self, S, K):
        """
        :type S: str
        :type K: int
        :rtype: str
        """
        S = S.replace('-', '').upper()
        result = ""

        if len(S)%K == 0:
            for index in range(0, len(S), K):
                result += S[index:index+K] + "-"
        else:
            result = S[:len(S)%K] + "-"
            for index in range(len(S)%K, len(S), K):
                result += S[index:index+K] + "-"

        return result[:-1]
```

```
'''
```

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

```
Example 1:
```

```
Input: [1,3,5,4,7]
```

```
Output: 2
```

```
Explanation: The two longest increasing subsequence are [1, 3, 4, 7] and
[1, 3, 5, 7].
```

```
Example 2:
```

```
Input: [2,2,2,2,2]
```

```
Output: 5
```

```
Explanation: The length of longest continuous increasing subsequence is
1, and there are 5 subsequences' length is 1, so output 5.
```

```
Note: Length of the given array will be not exceed 2000 and the answer is
guaranteed to be fit in 32-bit signed int.
```

```
'''
```

```
class Solution(object):
    def findNumberOfLIS(self, nums):
        length = [1]*len(nums)
        count = [1]*len(nums)
        result = 0
        for end, num in enumerate(nums):
            for start in range(end):
                if num > nums[start]:
                    if length[start] >= length[end]:
                        length[end] = 1+length[start]
                        count[end] = count[start]
                    elif length[start] + 1 == length[end]:
                        count[end] += count[start]
        for index, max_subs in enumerate(count):
            if length[index] == max(length):
                result += max_subs
        return result
```

```
'''
```

```
Given an unsorted array of integers, find the length of longest  
continuous increasing subsequence (subarray).
```

```
Example 1:
```

```
Input: [1,3,5,4,7]
```

```
Output: 3
```

```
Explanation: The longest continuous increasing subsequence is [1,3,5],  
its length is 3.
```

```
Even though [1,3,5,7] is also an increasing subsequence, it's not a  
continuous one where 5 and 7 are separated by 4.
```

```
Example 2:
```

```
Input: [2,2,2,2,2]
```

```
Output: 1
```

```
Explanation: The longest continuous increasing subsequence is [2], its  
length is 1.
```

```
Note: Length of the array will not exceed 10,000.
```

```
'''
```

```
class Solution(object):  
    def findLengthOfLCIS(self, nums):  
        """  
        :type nums: List[int]  
        :rtype: int  
        """  
        if not nums:  
            return 0  
        start, result = 0, 1  
        for end in range(1, len(nums)):  
            if nums[end-1] >= nums[end]:  
                start = end  
            result = max(result, end-start+1)  
        return result
```

'''

Given a time represented in the format "HH:MM", form the next closest time by reusing the current digits. There is no limit on how many times a digit can be reused.

You may assume the given input string is always valid. For example, "01:34", "12:09" are all valid. "1:34", "12:9" are all invalid.

Example 1:

Input: "19:34"  
Output: "19:39"

Explanation: The next closest time choosing from digits 1, 9, 3, 4, is 19:39, which occurs 5 minutes later. It is not 19:33, because this occurs 23 hours and 59 minutes later.

Example 2:

Input: "23:59"  
Output: "22:22"

Explanation: The next closest time choosing from digits 2, 3, 5, 9, is 22:22. It may be assumed that the returned time is next day's time since it is smaller than the input time numerically.

'''

```
class Solution(object):
    def nextClosestTime(self, time):
        current_time = 60*int(time[:2]) + int(time[3:])
        allowed = {int(x) for x in time if x != ':'}
        result = 24*60
        ans = current_time
        for h1, h2, m1, m2 in itertools.product(allowed, repeat=4):
            hours, minutes = 10*h1+h2, 10*m1+m2
            if hours < 24 and minutes < 60:
                elapsed = 60*hours + minutes
                diff = (current_time - elapsed)%(24*60)
                if 0 < diff < result:
                    result = diff
                    ans = elapsed

        return "{:02d}:{:02d}".format(divmod(ans, 60))
```

'''

In a string S of lowercase letters, these letters form consecutive groups of the same character.

For example, a string like S = "abbxxxxzzy" has the groups "a", "bb", "xxxx", "z" and "yy".

Call a group large if it has 3 or more characters. We would like the starting and ending positions of every large group.

The final answer should be in lexicographic order.

Example 1:

Input: "abbxxxxzzy"

Output: [[3, 6]]

Explanation: "xxxx" is the single large group with starting 3 and ending positions 6.

Example 2:

Input: "abc"

Output: []

Explanation: We have "a", "b" and "c" but no large group.

Example 3:

Input: "abcddeeeeaabbbcd"

Output: [[3, 5], [6, 9], [12, 14]]

'''

```
class Solution(object):
    def largeGroupPositions(self, S):
        """
        :type S: str
        :rtype: List[List[int]]
        """
        if not S:
            return []

        result = []
        count = 1
        prevChar = S[0]
        index_i = 0
        for index in range(1, len(S)):
            if S[index] == prevChar:
                count += 1
            else:
                if count >= 3:
                    result.append([index_i, index-1])

                count = 1
                prevChar = S[index]
                index_i = index

            if count >= 3:
                result.append([index_i, len(S)-1])
        return result
```



'''

We are given a personal information string S, which may represent either an email address or a phone number.

We would like to mask this personal information according to the following rules:

#### 1. Email address:

We define a name to be a string of length  $\geq 2$  consisting of only lowercase letters a-z or uppercase letters A-Z.

An email address starts with a name, followed by the symbol '@', followed by a name, followed by the dot '.' and followed by a name.

All email addresses are guaranteed to be valid and in the format of "name1@name2.name3".

To mask an email, all names must be converted to lowercase and all letters between the first and last letter of the first name must be replaced by 5 asterisks '\*'.

#### 2. Phone number:

A phone number is a string consisting of only the digits 0-9 or the characters from the set {'+', '-', '(', ')', ' '}. You may assume a phone number contains 10 to 13 digits.

The last 10 digits make up the local number, while the digits before those make up the country code. Note that the country code is optional. We want to expose only the last 4 digits and mask all other digits.

The local number should be formatted and masked as "\*\*\*\*-\*\*\*-1111", where 1 represents the exposed digits.

To mask a phone number with country code like "+111 111 111 1111", we write it in the form "+\*\*\*-\*\*\*-\*\*\*-1111". The '+' sign and the first '-' sign before the local number should only exist if there is a country code. For example, a 12 digit phone number mask should start with "+\*\*-".

Note that extraneous characters like "(", ")", " ", as well as extra dashes or plus signs not part of the above formatting scheme should be removed.

Return the correct "mask" of the information provided.

#### Example 1:

Input: "LeetCode@LeetCode.com"  
Output: "l\*\*\*\*\*e@leetcode.com"

Explanation: All names are converted to lowercase, and the letters between the first and last letter of the first name is replaced by 5 asterisks.  
Therefore, "leetcode" -> "l\*\*\*\*\*e".

Example 2:

Input: "AB@qq.com"  
Output: "a\*\*\*\*\*b@qq.com"

Explanation: There must be 5 asterisks between the first and last letter of the first name "ab". Therefore, "ab" -> "a\*\*\*\*\*b".

Example 3:

Input: "1(234)567-890"  
Output: "\*\*\*-\*\*\*-7890"

Explanation: 10 digits in the phone number, which means all digits make up the local number.

Example 4:

Input: "86-(10)12345678"  
Output: "+\*\*-\*\*\*-\*\*-5678"

Explanation: 12 digits, 2 digits for country code and 10 digits for local number.

Notes:

S.length <= 40.  
Emails have length at least 8.  
Phone numbers have length at least 10.

'''

```
class Solution(object):
    def maskPII(self, S):
        """
        :type S: str
        :rtype: str
        """
        if '@' in S:
            S = S.lower()
            firstChar = S[0]
            asterix = S.find('@')
            return S[0] + "*****" + S[asterix-1:]
        else:
            S = S.replace('+', '')
            S = S.replace('(', '')
            S = S.replace(')', '')
            S = S.replace('-', '')
            S = S.replace(' ', '')

            if len(S) == 10:
                return "***-***-" + S[-4:]
            else:
                countryCode = len(S) - 10
                result = "+"
                for index in range(countryCode):
                    result += "*"
                return result + "-***-***-" + S[-4:]
```

'''

A string of '0's and '1's is monotone increasing if it consists of some number of '0's (possibly 0), followed by some number of '1's (also possibly 0.)

We are given a string S of '0's and '1's, and we may flip any '0' to a '1' or a '1' to a '0'.

Return the minimum number of flips to make S monotone increasing.

Example 1:

Input: "00110"

Output: 1

Explanation: We flip the last digit to get 00111.

Example 2:

Input: "010110"

Output: 2

Explanation: We flip to get 011111, or alternatively 000111.

Example 3:

Input: "00011000"

Output: 2

Explanation: We flip to get 00000000.

Note:

1 <= S.length <= 20000

S only consists of '0' and '1' characters.

'''

```
class Solution(object):
    def minFlipsMonoIncr(self, S):
        """
        :type S: str
        :rtype: int
        """
        ones = [0]
        for char in S:
            ones.append(ones[-1] + int(char))
        # print ones
        result = float('inf')
        for index in range(len(ones)):
            zeroes = len(S) - index - (ones[-1]-ones[index])
            result = min(zeroes+ones[index], result)
        return result
```

```
'''
```

```
Given an array of integers A sorted in non-decreasing order, return an
array of the squares of each number, also in sorted non-decreasing order.
```

```
Example 1:
```

```
Input: [-4,-1,0,3,10]
Output: [0,1,9,16,100]
```

```
Example 2:
```

```
Input: [-7,-3,2,3,11]
Output: [4,9,9,49,121]
'''
```

```
class Solution(object):
    def sortedSquares(self, A):
        """
        :type A: List[int]
        :rtype: List[int]
        """
        N = len(A)
        j = 0
        while j < N and A[j] < 0:
            j += 1
        i = j-1
        result = []
        while i >= 0 and j < N:
            if A[i]**2 < A[j]**2:
                result.append(A[i]**2)
                i -= 1
            else:
                result.append(A[j]**2)
                j += 1
        while i >= 0:
            result.append(A[i]**2)
            i -= 1

        while j < N:
            result.append(A[j]**2)
            j += 1

        return result
```

```

"""
Create a timebased key-value store class TimeMap, that supports two
operations.

1. set(string key, string value, int timestamp)

Stores the key and value, along with the given timestamp.
2. get(string key, int timestamp)

Returns a value such that set(key, value, timestamp_prev) was called
previously, with timestamp_prev <= timestamp.
If there are multiple such values, it returns the one with the largest
timestamp_prev.
If there are no values, it returns the empty string ("").
"""

import bisect
class TimeMap(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.time_dict = {}
        self.key_map = {}

    def set(self, key, value, timestamp):
        """
        :type key: str
        :type value: str
        :type timestamp: int
        :rtype: None
        """
        if key in self.time_dict:
            self.time_dict[key].append(timestamp)
            self.key_map[key].append(value)
        else:
            self.time_dict[key] = [timestamp]
            self.key_map[key] = [value]

    def get(self, key, timestamp):
        """
        :type key: str
        :type timestamp: int
        :rtype: str
        """
        if key in self.time_dict:
            t_values = self.time_dict[key]
            index = bisect.bisect_right(t_values, timestamp)
            if index-1 == len(t_values) or index == 0:
                return ''
            return self.key_map[key][index-1]

```

```
# Your TimeMap object will be instantiated and called as such:  
# obj = TimeMap()  
# obj.set(key,value,timestamp)  
# param_2 = obj.get(key,timestamp)
```

```
'''
```

In a country popular for train travel, you have planned some train travelling one year in advance. The days of the year that you will travel is given as an array days. Each day is an integer from 1 to 365.

Train tickets are sold in 3 different ways:

```
a 1-day pass is sold for costs[0] dollars;  
a 7-day pass is sold for costs[1] dollars;  
a 30-day pass is sold for costs[2] dollars.
```

The passes allow that many days of consecutive travel. For example, if we get a 7-day pass on day 2, then we can travel for 7 days: day 2, 3, 4, 5, 6, 7, and 8.

Return the minimum number of dollars you need to travel every day in the given list of days.

Example 1:

```
Input: days = [1,4,6,7,8,20], costs = [2,7,15]
```

```
Output: 11
```

```
'''
```

```
class Solution:  
    def mincostTickets(self, days: 'List[int]', costs: 'List[int}') ->  
        'int':  
            def get_days_ago(day, ago):  
                for i in range(len(days)):  
                    if days[i] > days[day-1] - ago:  
                        return i  
                out = [0] * (len(days) + 1)  
                for i in range(1, len(days) + 1):  
                    out[i] = min(out[i-1] + costs[0], out[get_days_ago(i, 7)] +  
                                costs[1], out[get_days_ago(i, 30)] + costs[2])  
                return out[-1]
```

```
'''
```

```
Given two integers A and B, return any string S such that:
```

```
S has length A + B and contains exactly A 'a' letters, and exactly B 'b' letters;
```

```
The substring 'aaa' does not occur in S;
```

```
The substring 'bbb' does not occur in S.
```

```
Example 1:
```

```
Input: A = 1, B = 2
```

```
Output: "abb"
```

```
Explanation: "abb", "bab" and "bba" are all correct answers.
```

```
'''
```

```
class Solution(object):
    def strWithout3a3b(self, A, B):
        """
        :type A: int
        :type B: int
        :rtype: str
        """

        result = ''
        if A > B:
            while B > 0 and A > 0:
                if A-B >= 3:
                    if A > 1:
                        result += 'aab'
                        A -= 2
                    else:
                        result += 'ab'
                        A -= 1
                B -= 1
            else:
                result += 'ab'
                A -= 1
                B -= 1
            if A > 0:
                result += 'a'*A
            if B > 0:
                result += 'b'*B
        else:
            while B > 0 and A > 0:
                if B-A >= 3:
                    if B > 1:
                        result += 'bba'
                        B -= 2
                    else:
                        result += 'ba'
                        B -= 1
                A -= 1
            else:
                result += 'ba'
                A -= 1
                B -= 1
            if A > 0:
```

```
    result += 'a'*A
if B > 0:
    result += 'b'*B

return result
```

```
'''
```

```
We have an array A of integers, and an array queries of queries.
```

```
For the i-th query val = queries[i][0], index = queries[i][1], we add val  
to A[index]. Then, the answer to the i-th query is the sum of the even  
values of A.
```

```
(Here, the given index = queries[i][1] is a 0-based index, and each query  
permanently modifies the array A.)
```

```
Return the answer to all queries. Your answer array should have  
answer[i] as the answer to the i-th query.
```

```
Example 1:
```

```
Input: A = [1,2,3,4], queries = [[1,0],[-3,1],[-4,0],[2,3]]  
Output: [8,6,2,4]
```

```
'''
```

```
class Solution(object):  
    def sumEvenAfterQueries(self, A, queries):  
        """  
        :type A: List[int]  
        :type queries: List[List[int]]  
        :rtype: List[int]  
        """  
        result = 0  
        for val in A:  
            if val%2 == 0:  
                result += val  
  
        f_result = []  
        for val_index in queries:  
            val, index = val_index[0], val_index[1]  
            prev_val = A[index]  
            if prev_val%2 == 0:  
                result -= prev_val  
            new_val = prev_val + val  
            if new_val %2 == 0:  
                result += new_val  
            A[index] = new_val  
            f_result.append(result)  
        return f_result
```

```

"""
Given the root of a binary tree, each node has a value from 0 to 25
representing the letters 'a' to 'z': a value of 0 represents 'a', a value
of 1 represents 'b', and so on.

Find the lexicographically smallest string that starts at a leaf of this
tree and ends at the root.
"""

# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def smallestFromLeaf(self, root):
        """
        :type root: TreeNode
        :rtype: str
        """
        self.result = "~"

        def dfs(node, A):
            if node:
                A.append(chr(node.val + ord('a')))
                if not node.left and not node.right:
                    self.result = min(self.result, "".join(reversed(A)))

                dfs(node.left, A)
                dfs(node.right, A)
                A.pop()
        dfs(root, [])
        return self.result

```

'''

For a non-negative integer X, the array-form of X is an array of its digits in left to right order. For example, if X = 1231, then the array form is [1,2,3,1].

Given the array-form A of a non-negative integer X, return the array-form of the integer X+K.

Example 1:

Input: A = [1,2,0,0], K = 34

Output: [1,2,3,4]

Explanation: 1200 + 34 = 1234

'''

```
class Solution(object):
    def addToArrayForm(self, A, K):
        """
        :type A: List[int]
        :type K: int
        :rtype: List[int]
        """
        arr_k = []
        while K >0:
            digit = K%10
            K /= 10
            arr_k.append(digit)

        arr_k.reverse()
        if len(arr_k) > len(A):
            A, arr_k = arr_k, A

        sum_arr = [0]*len(A)
        i, j = len(A)-1, len(arr_k)-1
        k = len(A) - 1
        digit_sum, carry = 0, 0
        while j >= 0:
            curr_sum = A[i] + arr_k[j] + carry
            sum_arr[k] = (curr_sum%10)
            carry = curr_sum//10
            i -= 1
            k -= 1
            j -= 1

        while i >= 0:
            curr_sum = A[i] + carry
            sum_arr[k] = (curr_sum%10)
            carry = curr_sum//10
            i -= 1
            k -= 1

        if carry:
            sum_arr = [carry] + sum_arr
        return sum_arr
```

'''

Given an array equations of strings that represent relationships between variables, each string equations[i] has length 4 and takes one of two different forms: "a==b" or "a!=b". Here, a and b are lowercase letters (not necessarily different) that represent one-letter variable names.

Return true if and only if it is possible to assign integers to variable names so as to satisfy all the given equations.

Example 1:

Input: ["a==b", "b!=a"]

Output: false

Explanation: If we assign say, a = 1 and b = 1, then the first equation is satisfied, but not the second. There is no way to assign the variables to satisfy both equations.

'''

```
class Solution(object):
    def equationsPossible(self, equations):
        """
        :type equations: List[str]
        :rtype: bool
        """
        equal_list, unequal_list = [], []
        for equation in equations:
            x, y = equation[0], equation[3]
            if '==' in equation:
                if not equal_list:
                    equal_list.append(x+y)
                else:
                    found = False
                    for index in range(0, len(equal_list)):
                        val = equal_list[index]
                        if x in val or y in val:
                            val = val+x+y
                            equal_list[index] = val
                            found = True
                    if not found:
                        equal_list.append(x+y)
            else:
                if x == y:
                    return False
                unequal_list.append([x, y])

        for val in unequal_list:
            for equal in equal_list:
                if val[0] in equal and val[1] in equal:
                    return False
return True
```

```
'''
```

On a broken calculator that has a number showing on its display, we can perform two operations:

Double: Multiply the number on the display by 2, or;  
Decrement: Subtract 1 from the number on the display.  
Initially, the calculator is displaying the number X.

Return the minimum number of operations needed to display the number Y.

Example 1:

```
Input: X = 2, Y = 3
Output: 2
Explanation: Use double operation and then decrement operation {2 -> 4 ->
3}.
'''
```

```
class Solution(object):
    def brokenCalc(self, X, Y):
        """
        :type X: int
        :type Y: int
        :rtype: int
        """
        if X == Y:
            return 0
        if X > Y:
            return X-Y
        if(Y%2 == 1):
            return 1 + self.brokenCalc(X, Y+1)
        else:
            return 1 + self.brokenCalc(X, Y/2)
```

'''

In a binary tree, the root node is at depth 0, and children of each depth  $k$  node are at depth  $k+1$ .

Two nodes of a binary tree are cousins if they have the same depth, but have different parents.

We are given the root of a binary tree with unique values, and the values  $x$  and  $y$  of two different nodes in the tree.

Return true if and only if the nodes corresponding to the values  $x$  and  $y$  are cousins.

Input: root = [1,2,3,4], x = 4, y = 3

Output: false

'''

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def adjacent(self, root, node1, node2):
        if not root:
            return False

        value = False
        if (root.right and root.left):
            value = ((root.left.val == node1 and root.right.val ==
node2) or
                     (root.left.val == node2 and root.right.val ==
node1))

        return (value or
                self.adjacent(root.left, node1, node2) or
                self.adjacent(root.right, node1, node2))

    def _level(self, root, node, level):
        if not root:
            return 0
        if root.val == node:
            return level

        left_level = self._level(root.left, node, level+1)
        if left_level != 0:
            return left_level
        return self._level(root.right, node, level+1)

    def isCousins(self, root, x, y):
        """
        :type root: TreeNode
        :type x: int
        :type y: int
        :rtype: bool
        """
```

```
        if ((self._level(root, x, 1) == self._level(root, y, 1)) and not
self.adjacent(root, x, y)):
            return True
    return False
```

```
'''
```

```
In a given grid, each cell can have one of three values:
```

```
the value 0 representing an empty cell;  
the value 1 representing a fresh orange;  
the value 2 representing a rotten orange.
```

```
Every minute, any fresh orange that is adjacent (4-directionally) to a  
rotten orange becomes rotten.
```

```
Return the minimum number of minutes that must elapse until no cell has a  
fresh orange. If this is impossible, return -1 instead.
```

```
'''
```

```
class Solution(object):  
    def valid(self, row, col, row_size, col_size):  
        return row >= 0 and col >= 0 and row < row_size and col <  
col_size  
  
    def orangesRotting(self, grid):  
        """  
        :type grid: List[List[int]]  
        :rtype: int  
        """  
        queue = []  
        for row_index in range(len(grid)):  
            for col_index in range(len(grid[0])):  
                if grid[row_index][col_index] == 2:  
                    queue.append((row_index, col_index))  
  
        result = 0  
        queue.append((-1, -1))  
        while queue:  
            flag = False  
            print queue  
            while(queue[0][0] != -1 and queue[0][1] != -1):  
                (row, col) = queue[0]  
                if self.valid(row+1, col, len(grid), len(grid[0])) and  
grid[row+1][col] == 1 :  
                    if not flag:  
                        result += 1  
                        flag =True  
                    grid[row+1][col] = 2  
                    row += 1  
                    queue.append((row, col))  
                    row -= 1  
                if self.valid(row-1, col, len(grid), len(grid[0])) and  
grid[row-1][col] == 1 :  
                    if not flag:  
                        result += 1  
                        flag =True  
                    grid[row-1][col] = 2  
                    row -= 1  
                    queue.append((row, col))  
                    row += 1  
                if self.valid(row, col+1, len(grid), len(grid[0])) and  
grid[row][col+1] == 1 :  
                    if not flag:  
                        result += 1
```

```
        flag =True
        grid[row][col+1] = 2
        col += 1
        queue.append((row, col))
        col -= 1
        if self.valid(row, col-1, len(grid), len(grid[0])) and
grid[row][col-1] == 1 :
            if not flag:
                result += 1
                flag =True
                grid[row][col-1] = 2
                col -= 1
                queue.append((row, col))
                col += 1
            queue.pop(0)
        queue.pop(0)
    if queue:
        queue.append((-1, -1))
for row_index in range(len(grid)):
    for col_index in range(len(grid[0])):
        if grid[row_index][col_index] == 1:
            return -1
return result
```

```
'''
```

```
In an array A containing only 0s and 1s, a K-bit flip consists of  
choosing a (contiguous) subarray of length K and simultaneously changing  
every 0 in the subarray to 1, and every 1 in the subarray to 0.
```

```
Return the minimum number of K-bit flips required so that there is no 0  
in the array. If it is not possible, return -1.
```

```
Input: A = [0,1,0], K = 1
```

```
Output: 2
```

```
Explanation: Flip A[0], then flip A[2]
```

```
'''
```

```
class Solution:
```

```
    def minKBitFlips(self, a: 'List[int]', k: 'int') -> 'int':  
        from collections import deque  
        q = deque()  
        res = 0  
        for i in range(len(a)):  
            if len(q) % 2 != 0:  
                if a[i] == 1:  
                    res += 1  
                    q.append(i+k-1)  
            else:  
                if a[i] == 0:  
                    res += 1  
                    q.append(i+k-1)  
            if q and q[0] == i: q.popleft()  
            if q and q[-1] >= len(a): return -1  
        return res
```

'''

In a town, there are N people labelled from 1 to N. There is a rumor that one of these people is secretly the town judge.

If the town judge exists, then:

The town judge trusts nobody.

Everybody (except for the town judge) trusts the town judge.

There is exactly one person that satisfies properties 1 and 2.

You are given trust, an array of pairs trust[i] = [a, b] representing that the person labelled a trusts the person labelled b.

If the town judge exists and can be identified, return the label of the town judge. Otherwise, return -1.

Example 1:

Input: N = 2, trust = [[1,2]]

Output: 2

Example 2:

Input: N = 3, trust = [[1,3],[2,3]]

Output: 3

'''

```
class Solution(object):
    def findJudge(self, N, trust):
        """
        :type N: int
        :type trust: List[List[int]]
        :rtype: int
        """
        if not trust:
            return 1
        mapping = {}
        unique = set()
        for trustee_list in trust:
            unique.add(trustee_list[0])
            if trustee_list[1] in mapping:
                mapping[trustee_list[1]] += 1
            else:
                mapping[trustee_list[1]] = 1

        unique_set = len(unique)
        for key, value in mapping.items():
            if value == unique_set:
                return key
        return -1
```

```
'''
```

We are given the root node of a maximum tree: a tree where every node has a value greater than any other value in its subtree.

Just as in the previous problem, the given tree was constructed from an list A (root = Construct(A)) recursively with the following Construct(A) routine:

If A is empty, return null.

Otherwise, let A[i] be the largest element of A. Create a root node with value A[i].

The left child of root will be Construct([A[0], A[1], ..., A[i-1]])

The right child of root will be Construct([A[i+1], A[i+2], ..., A[A.length - 1]])

Return root.

Note that we were not given A directly, only a root node root = Construct(A).

Suppose B is a copy of A with the value val appended to it. It is guaranteed that B has unique values.

Return Construct(B).

Example 1:

Input: root = [4,1,3,null,null,2], val = 5

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

Explanation: A = [1,4,2,3], B = [1,4,2,3,5]

Example 2:

Input: root = [5,2,4,null,1], val = 3

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

Explanation: A = [2,1,5,4], B = [2,1,5,4,3]

```
'''
```

# Definition for a binary tree node.

# class TreeNode(object):

# def \_\_init\_\_(self, x):

# self.val = x

# self.left = None

# self.right = None

class Solution(object):

def insertIntoMaxTree(self, root, val):

```
        """
```

:type root: TreeNode

:type val: int

:rtype: TreeNode

```
        """
```

new\_node = TreeNode(val)

if not root:

return new\_node

```
if root.val < val:
    new_node.left = root
    return new_node

nrwrt = root
start, prev = root.right, root

while start:
    if(start.val > val):
        prev = start
        start = start.right
    else:
        break

prev.right = new_node
if not start:
    new_node.right = start
else:
    new_node.left = start

return root
```

'''

On an 8 x 8 chessboard, there is one white rook. There also may be empty squares, white bishops, and black pawns. These are given as characters 'R', '.', 'B', and 'p' respectively. Uppercase characters represent white pieces, and lowercase characters represent black pieces.

The rook moves as in the rules of Chess: it chooses one of four cardinal directions (north, east, west, and south), then moves in that direction until it chooses to stop, reaches the edge of the board, or captures an opposite colored pawn by moving to the same square it occupies. Also, rooks cannot move into the same square as other friendly bishops.

Return the number of pawns the rook can capture in one move.

Input:

```
[[".", ".", ".", ".", ".", ".", ".", "."], [".", ".", ".", "p", ".", ".", ".", "."], [".",
.", ".", "R", ".", ".", "p"], [".", ".", ".", ".", ".", ".", ".", "."], [".", ".", ".",
".", ".", ".", "."], [".", ".", ".", "p", ".", ".", ".", "."], [".", ".", ".", ".",
".", "."]]
```

Output: 3

Explanation:

In this example the rook is able to capture all the pawns.

'''

```
class Solution(object):
    def numRookCaptures(self, board):
        """
        :type board: List[List[str]]
        :rtype: int
        """
        result = 0
        rook_index = (0, 0)
        for row in range(len(board)):
            for col in range(len(board[0])):
                if board[row][col] == 'R':
                    rook_index = (row, col)
                    break

        flag = True
        col = rook_index[1]-1
        pawn = 0
        while col >= 0:
            if board[rook_index[0]][col] == 'B':
                flag = False
                break
            if board[rook_index[0]][col] == 'p':
                pawn += 1
                break
            col -= 1
        if flag and pawn != 0:
            result += 1

        flag = True
        col = rook_index[1]+1
        pawn = 0
        while col < len(board[0]):
            if board[rook_index[0]][col] == 'B':
                flag = False
```

```

        break
    if board[rook_index[0]][col] == 'p':
        pawn += 1
        break
    col += 1

    if flag and pawn != 0:
        result += 1

    flag = True
    row = rook_index[0]+1
    pawn = 0
    while row < len(board):
        if board[row][rook_index[1]] == 'B':
            flag = False
            break

        if board[row][rook_index[1]] == 'p':
            pawn += 1
            break
        row += 1

    if flag and pawn != 0:
        result += 1

    pawn = 0
    flag = True
    row = rook_index[0]-1
    while row >= 0:
        if board[row][rook_index[1]] == 'B':
            flag = False
            break
        if board[row][rook_index[1]] == 'p':
            pawn += 1
            break
        row -= 1
    if flag and pawn != 0:
        result += 1

return result

```

```
'''
```

```
Given an array A of strings made only from lowercase letters, return a
list of all characters that show up in all strings within the list
(including duplicates). For example, if a character occurs 3 times in
all strings but not 4 times, you need to include that character three
times in the final answer.
```

```
You may return the answer in any order.
```

```
Example 1:
```

```
Input: ["bella","label","roller"]
```

```
Output: ["e","l","l"]
```

```
Example 2:
```

```
Input: ["cool","lock","cook"]
```

```
Output: ["c","o"]
```

```
'''
```

```
class Solution(object):
    def commonChars(self, A):
        """
        :type A: List[str]
        :rtype: List[str]
        """
        char_map = {}
        for char in A[0]:
            if char in char_map:
                char_map[char] += 1
            else:
                char_map[char] = 1

        int_map = {}
        for index in range(1, len(A)):
            for char in char_map.keys():
                if char in A[index]:
                    char_count = min(A[index].count(char),
char_map[char])
                    char_map[char] = char_count
                else:
                    del char_map[char]

        result = []
        for key, value in char_map.items():
            result.extend([key]*value)

        return result
```

```
'''
```

We are given that the string "abc" is valid.

From any valid string V, we may split V into two pieces X and Y such that X + Y (X concatenated with Y) is equal to V. (X or Y may be empty.) Then, X + "abc" + Y is also valid.

If for example S = "abc", then examples of valid strings are: "abc", "aabcbc", "abcabc", "abcabcababcc". Examples of invalid strings are: "abccba", "ab", "cababc", "bac".

Return true if and only if the given string S is valid.

Example 1:

Input: "aabcbc"

Output: true

Explanation:

We start with the valid string "abc".

Then we can insert another "abc" between "a" and "bc", resulting in "a" + "abc" + "bc" which is "aabcbc".

Example 2:

Input: "abcabcababcc"

Output: true

Explanation:

"abcabcaabc" is valid after consecutive insertions of "abc".

Then we can insert "abc" before the last letter, resulting in "abcabcabab" + "abc" + "c" which is "abcabcababcc".

Example 3:

Input: "abccba"

Output: false

Example 4:

Input: "cababc"

Output: false

Note:

```
1 <= S.length <= 20000
S[i] is 'a', 'b', or 'c'
'''
```

```
class Solution(object):
    def isValid(self, S):
        """
        :type S: str
        :rtype: bool
        """
        stack = []
        if not S:
            return False

        for char in S:
```

```
if char == 'a':
    stack.append('a')
if char == 'b':
    if not stack:
        return False
    if stack[-1] == 'a':
        stack.pop()
        stack.append(char)
if char == 'c':
    if not stack:
        return False
    if stack[-1] == 'b':
        stack.pop()
return len(stack) == 0
```

```
'''
```

```
Given an array A of 0s and 1s, we may change up to K values from 0 to 1.
```

```
Return the length of the longest (contiguous) subarray that contains only 1s.
```

Example 1:

```
Input: A = [1,1,1,0,0,0,1,1,1,1,0], K = 2
```

```
Output: 6
```

```
Explanation:
```

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

```
Bolded numbers were flipped from 0 to 1. The longest subarray is underlined.
```

Example 2:

```
Input: A = [0,0,1,1,0,0,1,1,1,0,1,1,0,0,0,1,1,1,1], K = 3
```

```
Output: 10
```

```
Explanation:
```

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

```
Bolded numbers were flipped from 0 to 1. The longest subarray is underlined.
```

Note:

```
1 <= A.length <= 20000
```

```
0 <= K <= A.length
```

```
A[i] is 0 or 1
```

```
'''
```

```
class Solution(object):
    def longestOnes(self, A, K):
        """
        :type A: List[int]
        :type K: int
        :rtype: int
        """
        start_index = 0
        for end_index in range(0, len(A)):
            K -= 1-A[end_index]
            if K < 0:
                K += 1-A[start_index]
                start_index += 1
        return end_index-start_index+1
```

'''

Given an array A of integers, we must modify the array in the following way: we choose an i and replace A[i] with -A[i], and we repeat this process K times in total. (We may choose the same index i multiple times.)

Return the largest possible sum of the array after modifying it in this way.

Example 1:

Input: A = [4,2,3], K = 1

Output: 5

Explanation: Choose indices (1,) and A becomes [4,-2,3].

Example 2:

Input: A = [3,-1,0,2], K = 3

Output: 6

Explanation: Choose indices (1, 2, 2) and A becomes [3,1,0,2].

Example 3:

Input: A = [2,-3,-1,5,-4], K = 2

Output: 13

Explanation: Choose indices (1, 4) and A becomes [2,3,-1,5,4].

Note:

```
1 <= A.length <= 10000
1 <= K <= 10000
-100 <= A[i] <= 100
'''
```

```
class Solution(object):
    def largestSumAfterKNegations(self, A, K):
        """
        :type A: List[int]
        :type K: int
        :rtype: int
        """
        A.sort()
        index = 0
        while K > 0:
            if A[index] < 0:
                A[index] *= -1
            if A[index+1] < A[index] and index < len(A)-1:
                index += 1
            else:
                A[index] *= -1
            K -= 1
        return sum(A)
```

```
'''
```

```
Normally, the factorial of a positive integer n is the product of all  
positive integers less than or equal to n. For example, factorial(10) =  
10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1.
```

We instead make a clumsy factorial: using the integers in decreasing order, we swap out the multiply operations for a fixed rotation of operations: multiply (\*), divide (/), add (+) and subtract (-) in this order.

For example, clumsy(10) = 10 \* 9 / 8 + 7 - 6 \* 5 / 4 + 3 - 2 \* 1. However, these operations are still applied using the usual order of operations of arithmetic: we do all multiplication and division steps before any addition or subtraction steps, and multiplication and division steps are processed left to right.

Additionally, the division that we use is floor division such that 10 \* 9 / 8 equals 11. This guarantees the result is an integer.

Implement the clumsy function as defined above: given an integer N, it returns the clumsy factorial of N.

Example 1:

```
Input: 4  
Output: 7  
Explanation: 7 = 4 * 3 / 2 + 1
```

Example 2:

```
Input: 10  
Output: 12  
Explanation: 12 = 10 * 9 / 8 + 7 - 6 * 5 / 4 + 3 - 2 * 1
```

Note:

```
1 <= N <= 10000  
-2^31 <= answer <= 2^31 - 1 (The answer is guaranteed to fit within a  
32-bit integer.)
```

```
'''
```

```
class Solution(object):  
    def clumsy(self, N):  
        """  
        :type N: int  
        :rtype: int  
        """  
        return [0, 1, 2, 6, 7][N] if N < 5 else N + [1, 2, 2, -1][N % 4]
```

'''

In a row of dominoes, A[i] and B[i] represent the top and bottom halves of the i-th domino. (A domino is a tile with two numbers from 1 to 6 - one on each half of the tile.)

We may rotate the i-th domino, so that A[i] and B[i] swap values.

Return the minimum number of rotations so that all the values in A are the same, or all the values in B are the same.

If it cannot be done, return -1.

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

Output: 2

Explanation:

The first figure represents the dominoes as given by A and B: before we do any rotations.

If we rotate the second and fourth dominoes, we can make every value in the top row equal to 2, as indicated by the second figure.

'''

```
class Solution(object):
    def minDominoRotations(self, A, B):
        """
        :type A: List[int]
        :type B: List[int]
        :rtype: int
        """
        if len(A) != len(B):
            return -1
        if len(A) == 0:
            return 0

        for possibility in set([A[0], B[0]]):
            top_rotation, bottom_rotation = 0, 0
            for a_num, b_num in zip(A, B):
                if possibility not in [a_num, b_num]:
                    break
                top_rotation += int(b_num != possibility)
                bottom_rotation += int(a_num != possibility)
            else:
                return min(top_rotation, bottom_rotation)
        return -1
```

```

"""
Return the root node of a binary search tree that matches the given
preorder traversal.

(Recall that a binary search tree is a binary tree where for every node,
any descendant of node.left has a value < node.val, and any descendant of
node.right has a value > node.val. Also recall that a preorder traversal
displays the value of the node first, then traverses node.left, then
traverses node.right.)
"""

# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def bstFromPreorder(self, preorder):
        """
        :type preorder: List[int]
        :rtype: TreeNode
        """
        root = TreeNode(preorder[0])
        stack = [root]
        for index in range(1, len(preorder)):
            new_node = TreeNode(preorder[index])
            if new_node.val < stack[-1].val:
                stack[-1].left = new_node
            else:
                parent = None
                while stack and new_node.val > stack[-1].val:
                    parent = stack.pop()
                parent.right = new_node
            stack.append(new_node)
        return root

```

```
'''Every non-negative integer N has a binary representation. For example, 5 can be represented as "101" in binary, 11 as "1011" in binary, and so on. Note that except for N = 0, there are no leading zeroes in any binary representation.
```

The complement of a binary representation is the number in binary you get when changing every 1 to a 0 and 0 to a 1. For example, the complement of "101" in binary is "010" in binary.

For a given number N in base-10, return the complement of it's binary representation as a base-10 integer.

Example 1:

Input: 5

Output: 2

Explanation: 5 is "101" in binary, with complement "010" in binary, which is 2 in base-10.

Example 2:

Input: 7

Output: 0

Explanation: 7 is "111" in binary, with complement "000" in binary, which is 0 in base-10.

Example 3:

Input: 10

Output: 5

Explanation: 10 is "1010" in binary, with complement "0101" in binary, which is 5 in base-10.

Note:

$0 \leq N < 10^9$

'''

```
class Solution(object):
    def bitwiseComplement(self, N):
        """
        :type N: int
        :rtype: int
        """
        if N == 0:
            return 1
        import math
        bits = (int)(math.floor(math.log(N) /math.log(2))) + 1
        return ((1 << bits) - 1) ^ N
```

'''

In a list of songs, the i-th song has a duration of time[i] seconds.

Return the number of pairs of songs for which their total duration in seconds is divisible by 60. Formally, we want the number of indices i < j with  $(\text{time}[i] + \text{time}[j]) \% 60 == 0$ .

Example 1:

Input: [30, 20, 150, 100, 40]

Output: 3

Explanation: Three pairs have a total duration divisible by 60:

(time[0] = 30, time[2] = 150): total duration 180

(time[1] = 20, time[3] = 100): total duration 120

(time[1] = 20, time[4] = 40): total duration 60

Example 2:

Input: [60, 60, 60]

Output: 3

Explanation: All three pairs have a total duration of 120, which is divisible by 60.

Note:

1 <= time.length <= 60000

1 <= time[i] <= 500

'''

```
class Solution(object):
    def numPairsDivisibleBy60(self, time):
        """
        :type time: List[int]
        :rtype: int
        """
        count_arr = [0]*60
        result = 0
        for t in time:
            remainder = t%60
            complement = (60-remainder)%60
            result += count_arr[complement]
            count_arr[remainder] += 1
        return result

    """
    Explanation:
    Q1: why create array of size 60?
        it is similar to the map which store the count. Why only 60
    because 60 modulo of number cannot be more than 60
    Q2: why we need complement?
        to check the pair if it exist with given value or not
    example: if remainder is 20 then we need to check if we have any number
    with remainder 40 or not.
    Q3: why 60 modulo complement?
        for handle case when remainder is zero
    """
```

'''

A conveyor belt has packages that must be shipped from one port to another within D days.

The i-th package on the conveyor belt has a weight of weights[i]. Each day, we load the ship with packages on the conveyor belt (in the order given by weights). We may not load more weight than the maximum weight capacity of the ship.

Return the least weight capacity of the ship that will result in all the packages on the conveyor belt being shipped within D days.

Example 1:

Input: weights = [1,2,3,4,5,6,7,8,9,10], D = 5

Output: 15

Explanation:

A ship capacity of 15 is the minimum to ship all the packages in 5 days like this:

1st day: 1, 2, 3, 4, 5

2nd day: 6, 7

3rd day: 8

4th day: 9

5th day: 10

Note that the cargo must be shipped in the order given, so using a ship of capacity 14 and splitting the packages into parts like (2, 3, 4, 5), (1, 6, 7), (8), (9), (10) is not allowed.

Example 2:

Input: weights = [3,2,2,4,1,4], D = 3

Output: 6

Explanation:

A ship capacity of 6 is the minimum to ship all the packages in 3 days like this:

1st day: 3, 2

2nd day: 2, 4

3rd day: 1, 4

Note:

1 <= D <= weights.length <= 50000

1 <= weights[i] <= 500

'''

```
class Solution(object):
    def shipWithinDays(self, weights, D):
        """
        :type weights: List[int]
        :type D: int
        :rtype: int
        """
        high, low = sum(weights)+1, max(weights)
```

```

while(low < high):
    mid = (high+low)/2
    temp_left = mid
    packet_at_left = D-1
    for weight in weights:
        if weight <= mid:
            if temp_left < weight:
                if packet_at_left == 0:
                    low = mid+1
                    break
                packet_at_left -= 1
                temp_left = mid-weight
            else:
                temp_left -= weight
        else:
            high = mid

    return low

class Solution(object):
    def shipWithinDays(self, weights, D):
        """
        :type weights: List[int]
        :type D: int
        :rtype: int
        """
        left, right = max(weights), sum(weights)

        while left < right:
            curr_sum, groups, invalid = 0, 0, True
            mid = left + ((right-left) >> 1)
            for weight in weights:
                if weight > mid:
                    invalid = False
                    break
                if curr_sum + weight > mid:
                    groups += 1
                    curr_sum = 0
                curr_sum += weight
            if invalid and groups < D:
                right = mid
            else:
                left = mid + 1
        return left

```

```
'''
```

```
Given an array A of integers, return true if and only if we can partition  
the array into three non-empty parts with equal sums.
```

```
Formally, we can partition the array if we can find indexes i+1 < j with  
(A[0] + A[1] + ... + A[i] == A[i+1] + A[i+2] + ... + A[j-1] == A[j] +  
A[j+1] + ... + A[A.length - 1])
```

```
Example 1:
```

```
Input: [0,2,1,-6,6,-7,9,1,2,0,1]
```

```
Output: true
```

```
Explanation: 0 + 2 + 1 = -6 + 6 - 7 + 9 + 1 = 2 + 0 + 1
```

```
Example 2:
```

```
Input: [0,2,1,-6,6,7,9,-1,2,0,1]
```

```
Output: false
```

```
Example 3:
```

```
Input: [3,3,6,5,-2,2,5,1,-9,4]
```

```
Output: true
```

```
Explanation: 3 + 3 = 6 = 5 - 2 + 2 + 5 + 1 - 9 + 4
```

```
Note:
```

```
3 <= A.length <= 50000  
-10000 <= A[i] <= 10000  
'''
```

```
class Solution(object):  
    def canThreePartsEqualSum(self, A):  
        """  
        :type A: List[int]  
        :rtype: bool  
        """  
        total_sum = 0  
        for val in A:  
            total_sum += val  
  
        if(total_sum%3 != 0):  
            return False  
  
        curr_sum, groups = 0, 0  
        for val in A:  
            curr_sum += val  
            if curr_sum == total_sum/3:  
                curr_sum = 0  
                groups +=1  
        print groups  
        return groups == 3
```

'''

Given an array A of positive integers, A[i] represents the value of the i-th sightseeing spot, and two sightseeing spots i and j have distance  $j - i$  between them.

The score of a pair ( $i < j$ ) of sightseeing spots is  $(A[i] + A[j] + i - j)$  : the sum of the values of the sightseeing spots, minus the distance between them.

Return the maximum score of a pair of sightseeing spots.

Example 1:

Input: [8,1,5,2,6]

Output: 11

Explanation:  $i = 0, j = 2, A[i] + A[j] + i - j = 8 + 5 + 0 - 2 = 11$

Note:

```
2 <= A.length <= 50000
1 <= A[i] <= 1000
'''
class Solution(object):
    def maxScoreSightseeingPair(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        prev_best, result = 0, 0
        for index in range(0, len(A)):
            result = max(result, A[index]-index+prev_best)
            prev_best = max(prev_best, A[index]+index)
        return result
```

'''

Given a positive integer K, you need find the smallest positive integer N such that N is divisible by K, and N only contains the digit 1.

Return the length of N. If there is no such N, return -1.

Example 1:

Input: 1

Output: 1

Explanation: The smallest answer is N = 1, which has length 1.

Example 2:

Input: 2

Output: -1

Explanation: There is no such positive integer N divisible by 2.

Example 3:

Input: 3

Output: 3

Explanation: The smallest answer is N = 111, which has length 3.

'''

```
class Solution(object):
    def smallestRepunitDivByK(self, K):
        """
        :type K: int
        :rtype: int
        """
        length, value = 0, 0
        for no_one in range(100000):
            value = (10*value + 1)%K
            length += 1
            if value == 0:
                return length
        return -1
```

```
'''
```

```
Given a binary string S (a string consisting only of '0' and '1's) and a positive integer N, return true if and only if for every integer X from 1 to N, the binary representation of X is a substring of S.
```

Example 1:

Input: S = "0110", N = 3

Output: true

Example 2:

Input: S = "0110", N = 4

Output: false

Note:

```
1 <= S.length <= 1000
```

```
1 <= N <= 10^9
```

```
'''
```

```
class Solution(object):
    def queryString(self, S, N):
        """
        :type S: str
        :type N: int
        :rtype: bool
        """
        for num in range(1, N+1):
            binary_str = ''
            while (num != 0):
                binary_str += str(num%2)
                num /= 2
            reversed_str = binary_str[::-1]

            if reversed_str not in S:
                return False
        return True
```

'''

Given a number N, return a string consisting of "0"s and "1"s that represents its value in base -2 (negative two).

The returned string must have no leading zeroes, unless the string is "0".

Example 1:

Input: 2  
Output: "110"

Explantion:  $(-2)^2 + (-2)^1 = 2$

Example 2:

Input: 3  
Output: "111"  
Explantion:  $(-2)^2 + (-2)^1 + (-2)^0 = 3$

Example 3:

Input: 4  
Output: "100"  
Explantion:  $(-2)^2 = 4$

Note:

$0 \leq N \leq 10^9$

'''

```
class Solution(object):
    def baseNeg2(self, N):
        """
        :type N: int
        :rtype: str
        """
        if N == 0:
            digits = ['0']
        else:
            digits = []
            while N != 0:
                N, remainder = divmod(N, -2)
                if remainder < 0:
                    N, remainder = N+1, remainder + 2
                digits.append(str(remainder))
        return ''.join(digits[::-1])
```

'''

Given an array A of 0s and 1s, consider  $N_i$ : the i-th subarray from  $A[0]$  to  $A[i]$  interpreted as a binary number (from most-significant-bit to least-significant-bit.)

Return a list of booleans answer, where  $answer[i]$  is true if and only if  $N_i$  is divisible by 5.

Example 1:

Input: [0,1,1]  
Output: [true, false, false]

Explanation:

The input numbers in binary are 0, 01, 011; which are 0, 1, and 3 in base-10. Only the first number is divisible by 5, so  $answer[0]$  is true.

Example 2:

Input: [1,1,1]  
Output: [false, false, false]

Note:

$1 \leq A.length \leq 30000$   
 $A[i]$  is 0 or 1  
'''

```
class Solution(object):
    def prefixesDivBy5(self, A):
        """
        :type A: List[int]
        :rtype: List[bool]
        """
        result = []
        if not A:
            return []
        str_bin = ''
        for val in A:
            str_bin += str(val)
            if(int(str_bin, 2)%5 == 0):
                result.append(True)
            else:
                result.append(False)
        return result
```

```
'''
```

We are given a linked list with head as the first node. Let's number the nodes in the list: `node_1`, `node_2`, `node_3`, ... etc.

Each node may have a next larger value: for `node_i`, `next_larger(node_i)` is the `node_j.val` such that  $j > i$ , `node_j.val > node_i.val`, and  $j$  is the smallest possible choice. If such a  $j$  does not exist, the next larger value is 0.

Return an array of integers `answer`, where `answer[i] = next_larger(node_{i+1})`.

Note that in the example inputs (not outputs) below, arrays such as `[2,1,5]` represent the serialization of a linked list with a head node value of 2, second node value of 1, and third node value of 5.

Example 1:

Input: `[2,1,5]`

Output: `[5,5,0]`

Example 2:

Input: `[2,7,4,3,5]`

Output: `[7,0,5,5,0]`

Example 3:

Input: `[1,7,5,1,9,2,5,1]`

Output: `[7,9,9,9,0,5,0,0]`

Note:

$1 \leq \text{node.val} \leq 10^9$  for each node in the linked list.  
The given list has length in the range  $[0, 10000]$ .

```
'''
```

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def nextLargerNodes(self, head):
        """
        :type head: ListNode
        :rtype: List[int]
        """
        result = []
        while head:
            result.append(head.val)
            head = head.next

        stack = [result[-1]]
        ans = [0]
        for val in range(len(result)-2, -1, -1):
```

```
if result[val] < stack[-1]:
    ans.append(stack[-1])
else:
    while stack and stack[-1] <= result[val]:
        stack.pop()
    if stack:
        ans.append(stack[-1])
    else:
        ans.append(0)
    stack.append(result[val])
return ans[::-1]
```

'''

Given a 2D array A, each cell is 0 (representing sea) or 1 (representing land)

A move consists of walking from one land square 4-directionally to another land square, or off the boundary of the grid.

Return the number of land squares in the grid for which we cannot walk off the boundary of the grid in any number of moves.

Example 1:

Input: [[0,0,0,0],[1,0,1,0],[0,1,1,0],[0,0,0,0]]

Output: 3

Explanation:

There are three 1s that are enclosed by 0s, and one 1 that isn't enclosed because its on the boundary.

Example 2:

Input: [[0,1,1,0],[0,0,1,0],[0,0,1,0],[0,0,0,0]]

Output: 0

Explanation:

All 1s are either on the boundary or can reach the boundary.

Note:

1 <= A.length <= 500  
1 <= A[i].length <= 500  
0 <= A[i][j] <= 1  
All rows have the same size.  
'''

```
class Solution(object):
    def numEnclaves(self, A):
        """
        :type A: List[List[int]]
        :rtype: int
        """
        result = 0
        queue = []
        for row in range(len(A)):
            for col in range(len(A[0])):
                result += A[row][col]
                if (row*col == 0 or row == len(A)-1 or col == len(A[0])-1) and A[row][col] == 1:
                    queue.append((row, col))

        x_move = [-1, 0, 1, 0]
        y_move = [0, 1, 0, -1]

        while queue:
            x, y = queue.pop(0)
            A[x][y] = 0
            result -= 1
```

```
for xm, ym in zip(x_move, y_move):
    nx = x + xm
    ny = y + ym

    if 0 <= nx < len(A) and 0 <= ny < len(A[0]) and A[nx][ny]
    == 1 and (nx, ny) not in queue:
        queue.append((nx, ny))

return result
```

'''

A valid parentheses string is either empty (""), "(" + A + ")", or A + B, where A and B are valid parentheses strings, and + represents string concatenation. For example, "", "()", "(()())", and "((()()))" are all valid parentheses strings.

A valid parentheses string S is primitive if it is nonempty, and there does not exist a way to split it into  $S = A+B$ , with A and B nonempty valid parentheses strings.

Given a valid parentheses string S, consider its primitive decomposition:  $S = P_1 + P_2 + \dots + P_k$ , where  $P_i$  are primitive valid parentheses strings.

Return S after removing the outermost parentheses of every primitive string in the primitive decomposition of S.

Example 1:

Input: "(()())((())"

Output: "()()()

Explanation:

The input string is "(()())((())", with primitive decomposition "(()())" + "((())".

After removing outer parentheses of each part, this is "()()" + "()" = "()()()".

Example 2:

Input: "(()())((())((())"

Output: "()()()()()

Explanation:

The input string is "(()())((())((())", with primitive decomposition "(()())" + "((())" + "((()))".

After removing outer parentheses of each part, this is "()()" + "()" + "()" = "()()()()()".

Example 3:

Input: "()()

Output: ""

Explanation:

The input string is "()()", with primitive decomposition "()" + "()".

After removing outer parentheses of each part, this is "" + "" = "".

Note:

S.length <= 10000  
S[i] is "(" or ")"  
S is a valid parentheses string  
'''

```
class Solution(object):
    def removeOuterParentheses(self, S):
        """
        :type S: str
        :rtype: str
```

```
"""
temp, result = "", ""
start_bracket = 0
for char in S:
    temp += char
    if char == '(':
        start_bracket += 1
    else:
        start_bracket -= 1
    if start_bracket == 0:
        result += temp[1:-1]
        temp = ""
return result
```

'''

Given a binary tree, each node has value 0 or 1. Each root-to-leaf path represents a binary number starting with the most significant bit. For example, if the path is 0 -> 1 -> 1 -> 0 -> 1, then this could represent 01101 in binary, which is 13.

For all leaves in the tree, consider the numbers represented by the path from the root to that leaf.

Return the sum of these numbers.

Example 1:

Input: [1,0,1,0,1,0,1]

Output: 22

Explanation: (100) + (101) + (110) + (111) = 4 + 5 + 6 + 7 = 22

Note:

The number of nodes in the tree is between 1 and 1000.

node.val is 0 or 1.

The answer will not exceed  $2^{31} - 1$ .

'''

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def sumRootToLeaf(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        def traversal(root, paths, pathlen, allpaths):
            if not root:
                return
            if len(paths) > pathlen:
                paths[pathlen] = root.val
            else:
                paths.append(root.val)

            pathlen +=1
            if not root.left and not root.right:
                allpaths.append(int(''.join(str(val) for val in
paths[0:pathlen])), 2))
            else:
                traversal(root.left, paths, pathlen, allpaths)
                traversal(root.right, paths, pathlen, allpaths)
        paths = []
        traversal(root, [], 0, paths)
```

```
return sum(paths) %1000000007
```

```
'''
```

A query word matches a given pattern if we can insert lowercase letters to the pattern word so that it equals the query. (We may insert each character at any position, and may insert 0 characters.)

Given a list of queries, and a pattern, return an answer list of booleans, where answer[i] is true if and only if queries[i] matches the pattern.

Example 1:

```
Input: queries =
["FooBar","FooBarTest","FootBall","FrameBuffer","ForceFeedBack"], pattern
= "FB"
Output: [true,false,true,true,false]
Explanation:
"FooBar" can be generated like this "F" + "oo" + "B" + "ar".
"FootBall" can be generated like this "F" + "oot" + "B" + "all".
"FrameBuffer" can be generated like this "F" + "rame" + "B" + "uffer".
```

Example 2:

```
Input: queries =
["FooBar","FooBarTest","FootBall","FrameBuffer","ForceFeedBack"], pattern
= "FoBa"
Output: [true,false,true,false,false]
Explanation:
"FooBar" can be generated like this "Fo" + "o" + "Ba" + "r".
"FootBall" can be generated like this "Fo" + "ot" + "Ba" + "ll".
Example 3:
```

```
Input: queries =
["FooBar","FooBarTest","FootBall","FrameBuffer","ForceFeedBack"], pattern
= "FoBaT"
Output: [false,true,false,false,false]
Explanation:
"FooBarTest" can be generated like this "Fo" + "o" + "Ba" + "r" + "T" +
"est".
```

Note:

1.  $1 \leq \text{queries.length} \leq 100$
2.  $1 \leq \text{queries[i].length} \leq 100$
3.  $1 \leq \text{pattern.length} \leq 100$
4. All strings consists only of lower and upper case English letters.

```
'''
```

```
class Solution(object):
    def camelMatch(self, queries, pattern):
        """
        :type queries: List[str]
        :type pattern: str
        :rtype: List[bool]
        """
        import re
        result = []
```

```
patterns = re.findall('[A-Z][a-z]*', pattern)

for query in queries:
    splitter = re.findall('[A-Z][a-z]*', query)
    flag = True
    if len(patterns) == len(splitter):
        for index in range(len(patterns)):
            # print patterns[index], splitter[index]
            p_i, s_i = 1, 1
            if patterns[index][0] == splitter[index][0]:
                while p_i < len(patterns[index]) and s_i <
len(splitter[index]):
                    if patterns[index][p_i] ==
splitter[index][s_i]:
                        p_i += 1
                        s_i += 1
                    else:
                        s_i += 1
                    if p_i != len(patterns[index]):
                        flag = False
                        break
                else:
                    flag = False
                    break
            if flag:
                result.append(True)
            else:
                result.append(False)
    else:
        result.append(False)
return result
```

'''

Alice and Bob take turns playing a game, with Alice starting first.

Initially, there is a number N on the chalkboard. On each player's turn, that player makes a move consisting of:

Choosing any x with  $0 < x < N$  and  $N \% x == 0$ .

Replacing the number N on the chalkboard with  $N - x$ .

Also, if a player cannot make a move, they lose the game.

Return True if and only if Alice wins the game, assuming both players play optimally.

Example 1:

Input: 2

Output: true

Explanation: Alice chooses 1, and Bob has no more moves.

Example 2:

Input: 3

Output: false

Explanation: Alice chooses 1, Bob chooses 1, and Alice has no more moves.

Note:

$1 \leq N \leq 1000$

'''

```
class Solution(object):
    def divisorGame(self, N):
        """
        :type N: int
        :rtype: bool
        """
        if N == 0:
            return False

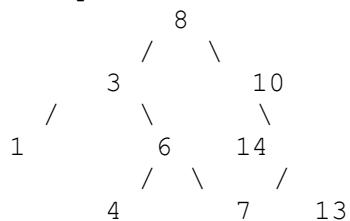
        move = 0
        while N > 1:
            for num in range(1, N):
                if N%num == 0:
                    N -= num
                    move += 1
                    break
        # print move
        if move%2:
            return True
        return False
```

1

Given the root of a binary tree, find the maximum value  $V$  for which there exists different nodes  $A$  and  $B$  where  $V = |A.val - B.val|$  and  $A$  is an ancestor of  $B$ .

(A node A is an ancestor of B if either: any child of A is equal to B, or any child of A is an ancestor of B.)

### Example 1:



Input: [8, 3, 10, 1, 6, null, 14, null, null, 4, 7, 13]

Output: 7

### Explanation:

We have various ancestor-node differences, some of which are given below  
:

1

$$| 3 | = 3$$

$$| 3 | = 4$$

$$|10 - 13| =$$

Among all nos

Among all possible differences, the maximum value of  $| \cdot |$  is obtained by  $| 18 - 1 | = 7$ .

Note:

The number of nodes in the tree is between 2 and 5000.  
Each node will have value between 0 and 100000.

7

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):

    def maxAncestorDiff(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """

    def utility_fun(root, res):
        if not root:
            return 2147483648, -
        if not root.left and not
            return root.val, ro
        left_t, lmax_t, res = ut
        right_t, rmax_t, res = u
```

```
m_val = min(left_t, right_t)
max_val = max(lmax_t, rmax_t)

res = max(res, max(abs(root.val-m_val), abs(root.val-
max_val)))
# print res
return min(m_val, root.val), max(max_val, root.val), res

x, x2, res = utility_fun(root, -2147483648)
return abs(res)
```

```
'''
```

```
Given an array A of integers, return the length of the longest arithmetic
subsequence in A.
```

```
Recall that a subsequence of A is a list A[i_1], A[i_2], ..., A[i_k] with
0 <= i_1 < i_2 < ... < i_k <= A.length - 1, and that a sequence B is
arithmetic if B[i+1] - B[i] are all the same value (for 0 <= i < B.length
- 1).
```

```
Example 1:
```

```
Input: [3,6,9,12]
```

```
Output: 4
```

```
Explanation:
```

```
The whole array is an arithmetic sequence with steps of length = 3.
```

```
Example 2:
```

```
Input: [9,4,7,2,10]
```

```
Output: 3
```

```
Explanation:
```

```
The longest arithmetic subsequence is [4,7,10].
```

```
'''
```

```
class Solution(object):
    def longestArithSeqLength(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        from collections import defaultdict

        dp = defaultdict(int)
        # print dp
        for index_i in range(len(A)):
            for index_j in range(index_i):
                diff = A[index_i] - A[index_j]
                dp[(index_i, diff)] = max(dp[(index_i, diff)], 
dp[(index_j, diff)]+1)
                # print dp
        return max(dp.itervalues())+1
```

```
'''
```

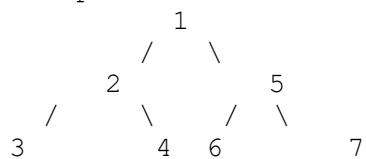
```
We run a preorder depth first search on the root of a binary tree.
```

```
At each node in this traversal, we output D dashes (where D is the depth  
of this node), then we output the value of this node. (If the depth of a  
node is D, the depth of its immediate child is D+1. The depth of the  
root node is 0.)
```

```
If a node has only one child, that child is guaranteed to be the left  
child.
```

```
Given the output S of this traversal, recover the tree and return its  
root
```

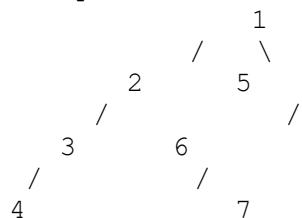
```
Example 1:
```



```
Input: "1-2--3--4-5--6--7"
```

```
Output: [1,2,5,3,4,6,7]
```

```
Example 2:
```



```
Input: "1-2--3---4-5--6---7"
```

```
Output: [1,2,5,3,null,6,null,4,null,7]
```

```
Example 3:
```

```
Input: "1-401--349---90--88"
```

```
Output: [1,401,null,349,88,90]
```

```
Note:
```

```
The number of nodes in the original tree is between 1 and 1000.  
Each node will have a value between 1 and 10^9.
```

```
'''
```

```
class Solution(object):  
    def longestArithSeqLength(self, A):  
        """  
        :type A: List[int]  
        :rtype: int  
        """  
        from collections import defaultdict
```

```
dp = defaultdict(int)
# print dp
for index_i in range(len(A)):
    for index_j in range(index_i):
        diff = A[index_i] - A[index_j]
        dp[(index_i, diff)] = max(dp[(index_i, diff)], 
dp[(index_j, diff)]+1)
        # print dp
return max(dp.itervalues())+1
```

'''

There are  $2N$  people a company is planning to interview. The cost of flying the  $i$ -th person to city A is  $\text{costs}[i][0]$ , and the cost of flying the  $i$ -th person to city B is  $\text{costs}[i][1]$ .

Return the minimum cost to fly every person to a city such that exactly  $N$  people arrive in each city.

Example 1:

Input: [[10,20],[30,200],[400,50],[30,20]]

Output: 110

Explanation:

The first person goes to city A for a cost of 10.

The second person goes to city A for a cost of 30.

The third person goes to city B for a cost of 50.

The fourth person goes to city B for a cost of 20.

The total minimum cost is  $10 + 30 + 50 + 20 = 110$  to have half the people interviewing in each city.

Note:

$1 \leq \text{costs.length} \leq 100$

It is guaranteed that  $\text{costs.length}$  is even.

$1 \leq \text{costs}[i][0], \text{costs}[i][1] \leq 1000$

'''

```
class Solution(object):
    def twoCitySchedCost(self, costs):
        """
        :type costs: List[List[int]]
        :rtype: int
        """
        result = 0
        costs = sorted(costs, key=lambda x : x[0] - x[1])
        for index in range(len(costs)):
            if index < len(costs)//2:
                result += costs[index][0]
            else:
                result += costs[index][1]
        return result
```

'''

We are given a matrix with R rows and C columns has cells with integer coordinates (r, c), where  $0 \leq r < R$  and  $0 \leq c < C$ .

Additionally, we are given a cell in that matrix with coordinates (r0, c0).

Return the coordinates of all cells in the matrix, sorted by their distance from (r0, c0) from smallest distance to largest distance. Here, the distance between two cells (r1, c1) and (r2, c2) is the Manhattan distance,  $|r1 - r2| + |c1 - c2|$ . (You may return the answer in any order that satisfies this condition.)

Example 1:

Input: R = 1, C = 2, r0 = 0, c0 = 0

Output: [[0,0],[0,1]]

Explanation: The distances from (r0, c0) to other cells are: [0,1]

Example 2:

Input: R = 2, C = 2, r0 = 0, c0 = 1

Output: [[0,1],[0,0],[1,1],[1,0]]

Explanation: The distances from (r0, c0) to other cells are: [0,1,1,2]

The answer [[0,1],[1,1],[0,0],[1,0]] would also be accepted as correct.

Example 3:

Input: R = 2, C = 3, r0 = 1, c0 = 2

Output: [[1,2],[0,2],[1,1],[0,1],[1,0],[0,0]]

Explanation: The distances from (r0, c0) to other cells are:

[0,1,1,2,2,3]

There are other answers that would also be accepted as correct, such as [[1,2],[1,1],[0,2],[1,0],[0,1],[0,0]].

Note:

```
1 <= R <= 100
1 <= C <= 100
0 <= r0 < R
0 <= c0 < C
'''
```

```
class Solution(object):
    def allCellsDistOrder(self, R, C, r0, c0):
        """
        :type R: int
        :type C: int
        :type r0: int
        :type c0: int
        :rtype: List[List[int]]
        """
        cells = [[x, y] for x in range(R) for y in range(C)]
        distance = {}
        for cell in cells:
            diff = abs(cell[0]-r0) + abs(cell[1]-c0)
            if diff in distance:
                distance[diff].append(cell)
            else:
                distance[diff] = [cell]
        result = []
        for dist in sorted(distance):
            result += distance[dist]
        return result
```

```
        distance[diff].append(cell)
    else:
        distance[diff] = [cell]
result = []
for key in sorted(distance):
    for value in distance[key]:
        result.append(value)
return result
```

'''

Given an array A of non-negative integers, return the maximum sum of elements in two non-overlapping (contiguous) subarrays, which have lengths L and M. (For clarification, the L-length subarray could occur before or after the M-length subarray.)

Formally, return the largest V for which  $V = (A[i] + A[i+1] + \dots + A[i+L-1]) + (A[j] + A[j+1] + \dots + A[j+M-1])$  and either:

$0 \leq i < i + L - 1 < j < j + M - 1 < A.length$ , or  
 $0 \leq j < j + M - 1 < i < i + L - 1 < A.length$ .

Example 1:

Input: A = [0,6,5,2,2,5,1,9,4], L = 1, M = 2

Output: 20

Explanation: One choice of subarrays is [9] with length 1, and [6,5] with length 2.

Example 2:

Input: A = [3,8,1,3,2,1,8,9,0], L = 3, M = 2

Output: 29

Explanation: One choice of subarrays is [3,8,1] with length 3, and [8,9] with length 2.

Example 3:

Input: A = [2,1,5,6,0,9,5,0,3,8], L = 4, M = 3

Output: 31

Explanation: One choice of subarrays is [5,6,0,9] with length 4, and [3,8] with length 3.

Note:

```
L >= 1
M >= 1
L + M <= A.length <= 1000
0 <= A[i] <= 1000
'''
```

```
class Solution(object):
    def maxSumTwoNoOverlap(self, A, L, M):
        """
        :type A: List[int]
        :type L: int
        :type M: int
        :rtype: int
        """
        cumm_sum = [0]
        for index in range(len(A)):
            cumm_sum.append(cumm_sum[index]+A[index])
        result = 0

        def valid(index_i, index_j):
            return index_i+L <= len(A) and index_j+M <= len(A)
            and(index_j>=index_i+L or index_i>=index_j+M)
```

```
for index_i in range(len(A)):
    for index_j in range(len(A)):
        if valid(index_i, index_j):
            result = max(result, cumm_sum[index_i+L]-
cumm_sum[index_i] + cumm_sum[index_j+M]-cumm_sum[index_j])
return result
```

'''

Implement the StreamChecker class as follows:

StreamChecker(words): Constructor, init the data structure with the given words.

query(letter): returns true if and only if for some  $k \geq 1$ , the last  $k$  characters queried (in order from oldest to newest, including this letter just queried) spell one of the words in the given list.

Example:

```
StreamChecker streamChecker = new StreamChecker(["cd","f","kl"]); // init
the dictionary.
streamChecker.query('a');           // return false
streamChecker.query('b');           // return false
streamChecker.query('c');           // return false
streamChecker.query('d');           // return true, because 'cd' is in the
wordlist
streamChecker.query('e');           // return false
streamChecker.query('f');           // return true, because 'f' is in the
wordlist
streamChecker.query('g');           // return false
streamChecker.query('h');           // return false
streamChecker.query('i');           // return false
streamChecker.query('j');           // return false
streamChecker.query('k');           // return false
streamChecker.query('l');           // return true, because 'kl' is in the
wordlist
```

Note:

```
1 <= words.length <= 2000
1 <= words[i].length <= 2000
Words will only consist of lowercase English letters.
Queries will only consist of lowercase English letters.
The number of queries is at most 40000.
```

'''

```
class Trie(object):
    def __init__(self):
        self.nodes = {}
        self.word = False

class StreamChecker(object):

    def __init__(self, words):
        """
        :type words: List[str]
        """
        self.trie_node = Trie()
        for word in words:
            ptr = self.trie_node
            for char in reversed(word):
                if char not in ptr.nodes:
                    ptr.nodes[char] = Trie()
                ptr = ptr.nodes[char]
```

```
        ptr.word = True
        self.stream = []

def query(self, letter):
    """
    :type letter: str
    :rtype: bool
    """
    self.stream.append(letter)
    root = self.trie_node
    for char in reversed(self.stream):
        if char not in root.nodes:
            return False
        if root.nodes[char].word:
            return True
        root = root.nodes[char]

    return root.word

# Your StreamChecker object will be instantiated and called as such:
# obj = StreamChecker(words)
# param_1 = obj.query(letter)
```

```
'''
```

```
Three stones are on a number line at positions a, b, and c.
```

```
Each turn, let's say the stones are currently at positions x, y, z with x < y < z. You pick up the stone at either position x or position z, and move that stone to an integer position k, with x < k < z and k != y.
```

```
The game ends when you cannot make any more moves, ie. the stones are in consecutive positions.
```

```
When the game ends, what is the minimum and maximum number of moves that you could have made? Return the answer as an length 2 array: answer = [minimum_moves, maximum_moves]
```

Example 1:

```
Input: a = 1, b = 2, c = 5
```

```
Output: [1, 2]
```

```
Explanation: Move stone from 5 to 4 then to 3, or we can move it directly to 3.
```

Example 2:

```
Input: a = 4, b = 3, c = 2
```

```
Output: [0, 0]
```

```
Explanation: We cannot make any moves.
```

Note:

```
1 <= a <= 100
1 <= b <= 100
1 <= c <= 100
a != b, b != c, c != a
'''
```

```
class Solution(object):
    def numMovesStones(self, a, b, c):
        """
        :type a: int
        :type b: int
        :type c: int
        :rtype: List[int]
        """
        lista = [a, b, c]
        lista.sort()
        a, b, c = lista[0], lista[1], lista[2]
        minsteps = 0
        if b == a+1 and c == a+2:
            return [0, 0]
        elif b == a+1 or c == b+1 or c == b+2 or b == a+2:
            minsteps = 1
        else:
            minsteps = 2
        return [minsteps, b-a-1+c-b-1]
```

'''

Given a 2-dimensional grid of integers, each value in the grid represents the color of the grid square at that location.

Two squares belong to the same connected component if and only if they have the same color and are next to each other in any of the 4 directions.

The border of a connected component is all the squares in the connected component that are either 4-directionally adjacent to a square not in the component, or on the boundary of the grid (the first or last row or column).

Given a square at location ( $r_0, c_0$ ) in the grid and a color, color the border of the connected component of that square with the given color, and return the final grid.

Example 1:

Input: grid = [[1,1],[1,2]], r0 = 0, c0 = 0, color = 3  
Output: [[3, 3], [3, 2]]

Example 2:

Input: grid = [[1,2,2],[2,3,2]], r0 = 0, c0 = 1, color = 3  
Output: [[1, 3, 3], [2, 3, 3]]

Example 3:

Input: grid = [[1,1,1],[1,1,1],[1,1,1]], r0 = 1, c0 = 1, color = 2  
Output: [[2, 2, 2], [2, 1, 2], [2, 2, 2]]

Note:

```
1 <= grid.length <= 50
1 <= grid[0].length <= 50
1 <= grid[i][j] <= 1000
0 <= r0 < grid.length
0 <= c0 < grid[0].length
1 <= color <= 1000
"""

class Solution(object):
    def colorBorder(self, grid, r0, c0, color):
        """
        :type grid: List[List[int]]
        :type r0: int
        :type c0: int
        :type color: int
        :rtype: List[List[int]]
        """
        if not grid:
            return grid
        visited, border = [], []
        m, n = len(grid), len(grid[0])

        def dfs(r, c):
            if r < 0 or c < 0 or r >= m or c >= n or grid[r][c] != grid[r0][c0] or (r,c) in visited:
```

```

        return
    visited.append((r,c))

    # check if the current row, col index is edge of the matrix
    # if not then check adjacent cells doesnt have same value as
    grid[r0][c0] then add in border
    if (r == 0 or c == 0 or r == m-1 or c == n-1 or
        (r+1 < m and grid[r+1][c] != grid[r0][c0]) or
        (r-1 >= 0 and grid[r-1][c] != grid[r0][c0]) or
        (c+1 < n and grid[r][c+1] != grid[r0][c0]) or
        (c-1 >= 0 and grid[r][c-1] != grid[r0][c0])):
        border.append((r,c))
    dfs(r-1, c)
    dfs(r+1, c)
    dfs(r, c-1)
    dfs(r, c+1)

    dfs(r0, c0)
    for (x, y) in border:
        grid[x][y] = color
    return grid

```

'''

We write the integers of A and B (in the order they are given) on two separate horizontal lines.

Now, we may draw a straight line connecting two numbers A[i] and B[j] as long as A[i] == B[j], and the line we draw does not intersect any other connecting (non-horizontal) line.

Return the maximum number of connecting lines we can draw in this way.

Example 1:

Input: A = [1,4,2], B = [1,2,4]

Output: 2

Explanation: We can draw 2 uncrossed lines as in the diagram.

We cannot draw 3 uncrossed lines, because the line from A[1]=4 to B[2]=4 will intersect the line from A[2]=2 to B[1]=2.

Example 2:

Input: A = [2,5,1,2,5], B = [10,5,2,1,5,2]

Output: 3

Example 3:

Input: A = [1,3,7,1,7,5], B = [1,9,2,5,1]

Output: 2

Note:

```
1 <= A.length <= 500
1 <= B.length <= 500
1 <= A[i], B[i] <= 2000
'''

class Solution(object):
    def maxUncrossedLines(self, A, B):
        """
        :type A: List[int]
        :type B: List[int]
        :rtype: int
        """
        dp = [[0]*len(A) for _ in range(len(B))]

        dp[0][0] = 1 if A[0] == B[0] else 0
        for index_i in range(1, len(dp)):
            dp[index_i][0] = dp[index_i-1][0]
            if A[0] == B[index_i]:
                dp[index_i][0] = 1

        for index_j in range(1, len(dp[0])):
            dp[0][index_j] = dp[0][index_j-1]
            if B[0] == A[index_j]:
                dp[0][index_j] = 1

        for index_i in range(1, len(dp)):
```

```
for index_j in range(1, len(dp[0])):
    if A[index_j] == B[index_i]:
        dp[index_i][index_j] = max(dp[index_i-1][index_j-1] +
1, max(dp[index_i-1][index_j], dp[index_i][index_j-1]))
    else:
        dp[index_i][index_j] = max(dp[index_i-1][index_j-1],
max(dp[index_i-1][index_j], dp[index_i][index_j-1]))
return dp[len(B)-1][len(A)-1]
```

```
'''  
A boomerang is a set of 3 points that are all distinct and not in a  
straight line.
```

Given a list of three points in the plane, return whether these points  
are a boomerang.

Example 1:

Input: [[1,1],[2,3],[3,2]]

Output: true

Example 2:

Input: [[1,1],[2,2],[3,3]]

Output: false

Note:

```
points.length == 3  
points[i].length == 2  
0 <= points[i][j] <= 100  
'''  
  
class Solution(object):  
    def isBoomerang(self, points):  
        """  
        :type points: List[List[int]]  
        :rtype: bool  
        """  
        x1, x2, x3, y1, y2, y3 = points[0][0], points[1][0],  
        points[2][0], points[0][1], points[1][1], points[2][1]  
        if ((y3 - y2)*(x2 - x1) == (y2 - y1)*(x3 - x2)):  
            return False  
        return True
```

```
'''
```

```
Given the root of a binary search tree with distinct values, modify it so  
that every node has a new value equal to the sum of the values of the  
original tree that are greater than or equal to node.val.
```

```
As a reminder, a binary search tree is a tree that satisfies these  
constraints:
```

```
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:

```
Input: [4,1,6,0,2,5,7,null,null,null,3,null,null,null,8]
```

```
Output: [30,36,21,36,35,26,15,null,null,null,33,null,null,null,8]
```

Note:

```
The number of nodes in the tree is between 1 and 100.
```

```
Each node will have value between 0 and 100.
```

```
The given tree is a binary search tree.
```

```
'''
```

```
# Definition for a binary tree node.  
# class TreeNode(object):  
#     def __init__(self, x):  
#         self.val = x  
#         self.left = None  
#         self.right = None  
  
class Solution(object):  
    def bstToGst(self, root):  
        """  
        :type root: TreeNode  
        :rtype: TreeNode  
        """  
        self.curr_sum = 0  
        def greaterSum(root):  
            if not root:  
                return  
            greaterSum(root.right)  
            self.curr_sum += root.val  
            root.val = self.curr_sum  
            greaterSum(root.left)  
  
        greaterSum(root)  
        return root
```

'''

Given  $N$ , consider a convex  $N$ -sided polygon with vertices labelled  $A[0]$ ,  $A[1]$ , ...,  $A[N-1]$  in clockwise order.

Suppose you triangulate the polygon into  $N-2$  triangles. For each triangle, the value of that triangle is the product of the labels of the vertices, and the total score of the triangulation is the sum of these values over all  $N-2$  triangles in the triangulation.

Return the smallest possible total score that you can achieve with some triangulation of the polygon.

Example 1:

Input: [1,2,3]

Output: 6

Explanation: The polygon is already triangulated, and the score of the only triangle is 6.

Example 2:

```
3 - 7 3 - 7
| / | | \ |
5 - 4 5 - 4
```

Input: [3,7,4,5]

Output: 144

Explanation: There are two triangulations, with possible scores:  $3*7*5 + 4*5*7 = 245$ , or  $3*4*5 + 3*4*7 = 144$ . The minimum score is 144.

Example 3:

Input: [1,3,1,4,1,5]

Output: 13

Explanation: The minimum score triangulation has score  $1*1*3 + 1*1*4 + 1*1*5 + 1*1*1 = 13$ .

Note:

```
3 <= A.length <= 50
1 <= A[i] <= 100
'''
```

```
class Solution(object):
    def minScoreTriangulation(self, A):
        """
        :type A: List[int]
        :rtype: int
        """

        n = len(A)
        dp = [[0]*n for _ in range(n)]
        for length in range(n):
            index_i = 0
            for index_j in range(length, n):
                if index_j < index_i+2:
                    dp[index_i][index_j] = 0
```

```
        else:
            dp[index_i][index_j] = float('inf')
            for index_k in range(index_i+1, index_j):
                val = dp[index_i][index_k] + dp[index_k][index_j]
+ (A[index_i]*A[index_k]*A[index_j])
                dp[index_i][index_j] = min(dp[index_i][index_j],
val)
            index_i += 1
    return dp[0][n-1]
```

'''

On an infinite plane, a robot initially stands at (0, 0) and faces north. The robot can receive one of three instructions:

"G": go straight 1 unit;  
"L": turn 90 degrees to the left;  
"R": turn 90 degrees to the right.

The robot performs the instructions given in order, and repeats them forever.

Return true if and only if there exists a circle in the plane such that the robot never leaves the circle.

Example 1:

Input: "GGLLGG"  
Output: true  
Explanation:  
The robot moves from (0,0) to (0,2), turns 180 degrees, and then returns to (0,0).  
When repeating these instructions, the robot remains in the circle of radius 2 centered at the origin.  
Example 2:

Input: "GG"  
Output: false  
Explanation:  
The robot moves north indefinitely.  
Example 3:

Input: "GL"  
Output: true  
Explanation:  
The robot moves from (0, 0) -> (0, 1) -> (-1, 1) -> (-1, 0) -> (0, 0) -> ...

Note:

```
1 <= instructions.length <= 100
instructions[i] is in {'G', 'L', 'R'}
'''

class Solution(object):
    def isRobotBounded(self, instructions):
        """
        :type instructions: str
        :rtype: bool
        """
        start_x, start_y = 0, 0
        left, direct = 0, 0
        moves = [[0, 1], [-1, 0], [0, -1], [1, 0]]
        instructions = instructions*4
        for instruction in instructions:
            if instruction == 'G':
                start_x += moves[direct][0]
```

```
start_y += moves[direct][1]
elif instruction == 'L':
    direct = (direct+1)%4
elif instruction == 'R':
    direct = (direct+3)%4

if(start_x == 0 and start_y == 0):
    return True
return False
```

'''

You have N gardens, labelled 1 to N. In each garden, you want to plant one of 4 types of flowers.

paths[i] = [x, y] describes the existence of a bidirectional path from garden x to garden y.

Also, there is no garden that has more than 3 paths coming into or leaving it.

Your task is to choose a flower type for each garden such that, for any two gardens connected by a path, they have different types of flowers.

Return any such a choice as an array answer, where answer[i] is the type of flower planted in the (i+1)-th garden. The flower types are denoted 1, 2, 3, or 4. It is guaranteed an answer exists.

Example 1:

Input: N = 3, paths = [[1,2], [2,3], [3,1]]  
Output: [1,2,3]

Example 2:

Input: N = 4, paths = [[1,2], [3,4]]  
Output: [1,2,1,2]

Example 3:

Input: N = 4, paths = [[1,2], [2,3], [3,4], [4,1], [1,3], [2,4]]  
Output: [1,2,3,4]

Note:

1 <= N <= 10000  
0 <= paths.size <= 20000  
No garden has 4 or more paths coming into or leaving it.  
It is guaranteed an answer exists.

'''

```
class Solution(object):
    def gardenNoAdj(self, N, paths):
        """
        :type N: int
        :type paths: List[List[int]]
        :rtype: List[int]
        """
        plant = [1, 2, 3, 4]
        result = [0 for _ in range(N)]
        if not paths:
            return [plant[index%4] for index in range(N)]
        # print result
        change = {}
        update = []
        for path in paths:
            x, y = path[0]-1, path[1]-1
            if result[x] == result[y]:
                if x in change:
                    if change[x] == result[y]:
                        change.pop(x)
                else:
                    change[x] = result[y]
                if y in change:
                    if change[y] == result[x]:
                        change.pop(y)
                else:
                    change[y] = result[x]
            else:
                if x in change:
                    if change[x] == result[y]:
                        change.pop(x)
                else:
                    change[x] = result[y]
                if y in change:
                    if change[y] == result[x]:
                        change.pop(y)
                else:
                    change[y] = result[x]
            result[x] = 1
            result[y] = 2
            update.append([x, y])
        for i in update:
            if i[0] in change:
                if change[i[0]] == 1:
                    change.pop(i[0])
                    result[i[0]] = 2
                else:
                    change.pop(i[0])
                    result[i[0]] = 1
            if i[1] in change:
                if change[i[1]] == 1:
                    change.pop(i[1])
                    result[i[1]] = 2
                else:
                    change.pop(i[1])
                    result[i[1]] = 1
```

```
if x in change:
    change[x].append(y)
else:
    change[x] = [y]

if y in change:
    change[y].append(x)
else:
    change[y] = [x]

for garden in range(N):
    color_used = []
    if garden in change:
        subgarden = change[garden]
        for subgarden in change[garden]:
            if result[subgarden]:
                color_used.append(result[subgarden])
    color_rem = list(set([1, 2, 3, 4]) - set(color_used))
    for color in color_rem:
        result[garden] = color
        break
return result
```

```
'''
```

Given an integer array A, you partition the array into (contiguous) subarrays of length at most K. After partitioning, each subarray has their values changed to become the maximum value of that subarray.

Return the largest sum of the given array after partitioning.

Example 1:

Input: A = [1,15,7,9,2,5,10], K = 3

Output: 84

Explanation: A becomes [15,15,15,9,10,10,10]

Note:

1 <= K <= A.length <= 500

0 <= A[i] <= 10^6

```
'''
```

```
class Solution(object):
    def maxSumAfterPartitioning(self, A, K):
        """
        :type A: List[int]
        :type K: int
        :rtype: int
        """
        if not A:
            return 0

        N = len(A)
        dp = [0] * (N+1)
        for index_i in range(N):
            maxi = 0
            for index_j in range(index_i, index_i-K, -1):
                if index_j >= 0 and index_j < len(A):
                    maxi = max(maxi, A[index_j])

            dp[index_i+1] = max(dp[index_i+1], maxi*(index_i-
index_j+1)+dp[index_j])
            # print index_i, maxi
            # print dp

        return dp[-1]
```

```

"""
Given a string S, consider all duplicated substrings: (contiguous)
substrings of S that occur 2 or more times. (The occurrences may
overlap.)

Return any duplicated substring that has the longest possible length.
(If S does not have a duplicated substring, the answer is "").
```

Example 1:

Input: "banana"

Output: "ana"

Example 2:

Input: "abcd"

Output: ""

Note:

$2 \leq S.length \leq 10^5$   
 $S$  consists of lowercase English letters.

```

class Suffix(object):
    def __init__(self):
        self.index = 0
        self.first_rank = -1
        self.adjacent_rank = -1

    def __lt__(self, other):
        if self.first_rank == other.first_rank:
            return self.adjacent_rank < other.adjacent_rank
        return self.first_rank < other.first_rank

def create_suffix_array(s):
    N = len(s)
    suffix_array = []

    for index, char in enumerate(s):
        suffix_obj = Suffix()
        suffix_obj.index = index
        suffix_obj.first_rank = ord(char)-ord('a')
        suffix_obj.adjacent_rank = ord(s[index+1])-ord('a') if
(index+1 < N) else -1
        suffix_array.append(suffix_obj)

    suffix_array.sort()

    no_char = 4
    index_map = {}
    while no_char < 2*N:
        rank = 0
        prev_rank, suffix_array[0].first_rank =
suffix_array[0].first_rank, rank
        index_map[suffix_array[0].index] = 0
```

```

        for index in range(1, N):
            if suffix_array[index].first_rank == prev_rank and
suffix_array[index].adjacent_rank == suffix_array[index-1].adjacent_rank:
                suffix_array[index].first_rank = rank
            else:
                rank += 1
                prev_rank, suffix_array[index].first_rank =
suffix_array[index].first_rank, rank
                index_map[suffix_array[index].index] = index

        for index in range(N):
            adjacent_index = suffix_array[index].index + (no_char/2)
            suffix_array[index].adjacent_rank =
suffix_array[index_map[adjacent_index]] if adjacent_index < N else -1

        suffix_array.sort()
        no_char *= 2

    return [suffix.index for suffix in suffix_array]

def lcp_w_suffix_str(array, s):
    N = len(array)

    lcp_array = [0]*N
    inv_suffix = [0]*N

    for index in range(N):
        inv_suffix[array[index]] = index

    maxLen = 0

    for index in range(N):
        if inv_suffix[index] == N-1:
            maxLen = 0
            continue

            index_j = array[inv_suffix[index]+1]
            while(index+maxLen < N and index_j+maxLen < N and
s[index+maxLen] == s[index_j+maxLen]):
                maxLen += 1

        lcp_array[inv_suffix[index]] = maxLen

        if maxLen > 0:
            maxLen -= 1

    return lcp_array

class Solution(object):
    def longestDupSubstring(self, S):
        """
        :type S: str
        :rtype: str
        """
        suffix_array = create_suffix_array(S)
        lcp_array = lcp_w_suffix_str(suffix_array, S)

```

```
start, end = 0, 0

for index in range(len(S)):
    if lcp_array[index] > end:
        end = lcp_array[index]
        start = suffix_array[index]

if end == 0:
    return ""
# print start, end
return S[start:start+end]
```

```
'''
```

We have a collection of rocks, each rock has a positive integer weight.

Each turn, we choose the two heaviest rocks and smash them together. Suppose the stones have weights  $x$  and  $y$  with  $x \leq y$ . The result of this smash is:

If  $x == y$ , both stones are totally destroyed;  
If  $x != y$ , the stone of weight  $x$  is totally destroyed, and the stone of weight  $y$  has new weight  $y-x$ .  
At the end, there is at most 1 stone left. Return the weight of this stone (or 0 if there are no stones left.)

Example 1:

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

Output: 1

Explanation:

We combine 7 and 8 to get 1 so the array converts to [2,4,1,1,1] then, we combine 2 and 4 to get 2 so the array converts to [2,1,1,1] then, we combine 2 and 1 to get 1 so the array converts to [1,1,1] then, we combine 1 and 1 to get 0 so the array converts to [1] then that's the value of last stone.

Note:

```
1 <= stones.length <= 30
```

```
1 <= stones[i] <= 1000
```

```
'''
```

```
class Solution(object):
    def lastStoneWeight(self, stones):
        """
        :type stones: List[int]
        :rtype: int
        """
        while len(stones) > 1:
            max_x = max(stones)
            stones.remove(max_x)
            max_y = max(stones)
            stones.remove(max_y)

            if max_x != max_y:
                stones.append(max_x - max_y)
        return stones[0] if stones else 0
```

```
'''
```

Given a string S of lowercase letters, a duplicate removal consists of choosing two adjacent and equal letters, and removing them.

We repeatedly make duplicate removals on S until we no longer can.

Return the final string after all such duplicate removals have been made. It is guaranteed the answer is unique.

Example 1:

Input: "abbaca"

Output: "ca"

Explanation:

For example, in "abbaca" we could remove "bb" since the letters are adjacent and equal, and this is the only possible move. The result of this move is that the string is "aaca", of which only "aa" is possible, so the final string is "ca".

Note:

```
1 <= S.length <= 20000
S consists only of English lowercase letters.
```

```
'''
```

```
class Solution(object):
    def removeDuplicates(self, S):
        """
        :type S: str
        :rtype: str
        """
        stack = []
        if not S:
            return ""
        for char in S:
            if not stack:
                stack.append(char)
            else:
                first = stack[-1]
                if first == char:
                    stack.pop()
                else:
                    stack.append(char)
        if not stack:
            return ""
        return ''.join(stack)
```

```
'''
```

Given a list of words, each word consists of English lowercase letters.

Let's say word1 is a predecessor of word2 if and only if we can add exactly one letter anywhere in word1 to make it equal to word2. For example, "abc" is a predecessor of "abac".

A word chain is a sequence of words [word\_1, word\_2, ..., word\_k] with k >= 1, where word\_1 is a predecessor of word\_2, word\_2 is a predecessor of word\_3, and so on.

Return the longest possible length of a word chain with words chosen from the given list of words.

Example 1:

Input: ["a", "b", "ba", "bca", "bda", "bdca"]

Output: 4

Explanation: one of the longest word chain is "a", "ba", "bda", "bdca".

Note:

```
1 <= words.length <= 1000
1 <= words[i].length <= 16
words[i] only consists of English lowercase letters.
'''
```

```
class Solution(object):
    def longestStrChain(self, words):
        """
        :type words: List[str]
        :rtype: int
        """
        if not words:
            return 0
        words.sort(key=len)
        dp = collections.defaultdict(int)
        result = 0
        for word in words:
            for index in range(len(word)):
                char_excluded_string = word[:index] + word[index+1:]
                if char_excluded_string in dp:
                    dp[word] = max(dp[char_excluded_string]+1, dp[word])
                else:
                    dp[word] = max(dp[word], 1)
            result = max(dp[word], result)
        return result
```

'''

Students are asked to stand in non-decreasing order of heights for an annual photo.

Return the minimum number of students not standing in the right positions. (This is the number of students that must move in order for all students to be standing in non-decreasing order of height.)

Example 1:

Input: [1,1,4,2,1,3]

Output: 3

Explanation:

Students with heights 4, 3 and the last 1 are not standing in the right positions.

Note:

1 <= heights.length <= 100

1 <= heights[i] <= 100

'''

```
class Solution(object):
    def heightChecker(self, heights):
        """
        :type heights: List[int]
        :rtype: int
        """
        result = 0
        for new_h, hei in zip(heights, sorted(heights)):
            if new_h != hei:
                result += 1
        return result
```

'''

Today, the bookstore owner has a store open for `customers.length` minutes. Every minute, some number of customers (`customers[i]`) enter the store, and all those customers leave after the end of that minute.

On some minutes, the bookstore owner is grumpy. If the bookstore owner is grumpy on the  $i$ -th minute, `grumpy[i] = 1`, otherwise `grumpy[i] = 0`. When the bookstore owner is grumpy, the customers of that minute are not satisfied, otherwise they are satisfied.

The bookstore owner knows a secret technique to keep themselves not grumpy for  $X$  minutes straight, but can only use it once.

Return the maximum number of customers that can be satisfied throughout the day.

Example 1:

Input: `customers = [1,0,1,2,1,1,7,5]`, `grumpy = [0,1,0,1,0,1,0,1]`, `X = 3`  
Output: 16  
Explanation: The bookstore owner keeps themselves not grumpy for the last 3 minutes.  
The maximum number of customers that can be satisfied =  $1 + 1 + 1 + 1 + 7 + 5 = 16$ .

Note:

```
1 <= X <= customers.length == grumpy.length <= 20000
0 <= customers[i] <= 1000
0 <= grumpy[i] <= 1
"""
class Solution(object):
    def maxSatisfied(self, customers, grumpy, X):
        """
        :type customers: List[int]
        :type grumpy: List[int]
        :type X: int
        :rtype: int
        """
        result = 0

        prefix_sum = [0]*(len(customers)+1)
        index = 0
        for customer, grump in zip(customers, grumpy):
            prefix_sum[index+1] = prefix_sum[index]
            if grump == 0:
                result += customer
            else:
                prefix_sum[index+1] += customer
            index += 1
        # print prefix_sum
        curr_max = result + prefix_sum[X]
        # print curr_max
        for index in range(X+1, len(prefix_sum)):
            temp_max = result + prefix_sum[index] - prefix_sum[index-X]
```

```
# print temp_max
curr_max = max(curr_max, temp_max)
return curr_max
```

'''

Given an array A of positive integers (not necessarily distinct), return the lexicographically largest permutation that is smaller than A, that can be made with one swap (A swap exchanges the positions of two numbers A[i] and A[j]). If it cannot be done, then return the same array.

Example 1:

Input: [3,2,1]

Output: [3,1,2]

Explanation: Swapping 2 and 1.

Example 2:

Input: [1,1,5]

Output: [1,1,5]

Explanation: This is already the smallest permutation.

Example 3:

Input: [1,9,4,6,7]

Output: [1,7,4,6,9]

Explanation: Swapping 9 and 7.

Example 4:

Input: [3,1,1,3]

Output: [1,3,1,3]

Explanation: Swapping 1 and 3.

Note:

```
1 <= A.length <= 10000
1 <= A[i] <= 10000
"""
class Solution(object):
    def prevPermOpt1(self, A):
        """
        :type A: List[int]
        :rtype: List[int]
        """

        left, right = len(A)-2, len(A)-1
        for left in range(len(A)-2, -1, -1):
            if A[left] > A[left+1]:
                break
            else:
                return A
        right = A.index(max(ele for ele in A[left+1:] if ele < A[left]),
left)
        A[left], A[right] = A[right], A[left]
        return A
```

```
'''
```

```
In a warehouse, there is a row of barcodes, where the i-th barcode is  
barcodes[i].
```

```
Rearrange the barcodes so that no two adjacent barcodes are equal. You  
may return any answer, and it is guaranteed an answer exists.
```

```
Example 1:
```

```
Input: [1,1,1,2,2,2]  
Output: [2,1,2,1,2,1]
```

```
Example 2:
```

```
Input: [1,1,1,1,2,2,3,3]  
Output: [1,3,1,3,2,1,2,1]
```

```
Note:
```

```
1 <= barcodes.length <= 10000  
1 <= barcodes[i] <= 10000  
'''
```

```
class Solution(object):  
    def rearrangeBarcodes(self, barcodes):  
        """  
        :type barcodes: List[int]  
        :rtype: List[int]  
        """  
        import heapq  
        di = collections.Counter(barcodes)  
        pq = [(-value, key) for key, value in di.items()]  
        heapq.heapify(pq)  
        # print pq  
        result = []  
        while len(pq) >= 2:  
            freq1, barcode1 = heapq.heappop(pq)  
            freq2, barcode2 = heapq.heappop(pq)  
            result.extend([barcode1, barcode2])  
  
            if freq1 + 1:  
                heapq.heappush(pq, (freq1 + 1, barcode1))  
            if freq2 + 1:  
                heapq.heappush(pq, (freq2 + 1, barcode2))  
  
        if pq:  
            result.append(pq[0][1])  
  
        return result
```

'''

Given an array A of distinct integers sorted in ascending order, return the smallest index i that satisfies A[i] == i. Return -1 if no such i exists.

Example 1:

Input: [-10,-5,0,3,7]

Output: 3

Explanation:

For the given array, A[0] = -10, A[1] = -5, A[2] = 0, A[3] = 3, thus the output is 3.

Example 2:

Input: [0,2,5,8,17]

Output: 0

Explanation:

A[0] = 0, thus the output is 0.

Example 3:

Input: [-10,-5,3,4,7,9]

Output: -1

Explanation:

There is no such i that A[i] = i, thus the output is -1.

'''

```
class Solution(object):
    def fixedPoint(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        if not A:
            return -1
        for index, num in enumerate(A):
            if num == index:
                return index
        return -1
```

```
'''
```

```
Given a text string and words (a list of strings), return all index pairs [i, j] so that the substring text[i]...text[j] is in the list of words.
```

Example 1:

```
Input: text = "thestoryofleetcodeandme", words = ["story", "fleet", "leetcode"]
```

```
Output: [[3,7], [9,13], [10,17]]
```

Example 2:

```
Input: text = "ababa", words = ["aba", "ab"]
```

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

Explanation:

```
Notice that matches can overlap, see "aba" is found in [0,2] and [2,4].
```

Note:

All strings contains only lowercase English letters.

It's guaranteed that all strings in words are different.

1 <= text.length <= 100

1 <= words.length <= 20

1 <= words[i].length <= 50

Return the pairs [i,j] in sorted order (i.e. sort them by their first coordinate in case of ties sort them by their second coordinate).

```
'''
```

```
class Solution(object):
    def indexPairs(self, text, words):
        """
        :type text: str
        :type words: List[str]
        :rtype: List[List[int]]
        """
        if not words:
            return []
        result = []
        for word in words:
            starting = [index for index in range(len(text)) if text.startswith(word, index)]
            for start in starting:
                result.append([start, start+len(word)-1])
        # print starting
        result.sort()
        return result
```

```
'''
```

For strings S and T, we say "T divides S" if and only if  $S = T + \dots + T$  ( $T$  concatenated with itself 1 or more times)

Return the largest string X such that X divides str1 and X divides str2.

Example 1:

Input: str1 = "ABCABC", str2 = "ABC"  
Output: "ABC"

Example 2:

Input: str1 = "ABABAB", str2 = "ABAB"  
Output: "AB"  
Example 3:

Input: str1 = "LEET", str2 = "CODE"  
Output: ""

Note:

$1 \leq \text{str1.length} \leq 1000$   
 $1 \leq \text{str2.length} \leq 1000$   
 $\text{str1}[i]$  and  $\text{str2}[i]$  are English uppercase letters.

```
'''
```

```
class Solution(object):
    def gcdOfStrings(self, str1, str2):
        """
        :type str1: str
        :type str2: str
        :rtype: str
        """
        if len(str1) > len(str2):
            str1, str2 = str2, str1

        l_str1 = len(str1)
        for index in range(1, len(str1)+1):
            if l_str1%index != 0:
                continue

            size_to_take = int(l_str1/index)
            substr1 = str1[:size_to_take]
            substr2 = str2

            while substr1 == substr2[:size_to_take]:
                substr2 = substr2[size_to_take:]

            if substr2 == "":
                return substr1
        return ""
```

'''

Given a matrix consisting of 0s and 1s, we may choose any number of columns in the matrix and flip every cell in that column. Flipping a cell changes the value of that cell from 0 to 1 or from 1 to 0.

Return the maximum number of rows that have all values equal after some number of flips.

Example 1:

Input: [[0,1],[1,1]]

Output: 1

Explanation: After flipping no values, 1 row has all values equal.

Example 2:

Input: [[0,1],[1,0]]

Output: 2

Explanation: After flipping values in the first column, both rows have equal values.

Example 3:

Input: [[0,0,0],[0,0,1],[1,1,0]]

Output: 2

Explanation: After flipping values in the first two columns, the last two rows have equal values.

Note:

1 <= matrix.length <= 300

1 <= matrix[i].length <= 300

All matrix[i].length's are equal

matrix[i][j] is 0 or 1

'''

'''

Given two numbers arr1 and arr2 in base -2, return the result of adding them together.

Each number is given in array format: as an array of 0s and 1s, from most significant bit to least significant bit. For example, arr = [1,1,0,1] represents the number  $(-2)^3 + (-2)^2 + (-2)^0 = -3$ . A number arr in array format is also guaranteed to have no leading zeros: either arr == [0] or arr[0] == 1.

Return the result of adding arr1 and arr2 in the same format: as an array of 0s and 1s with no leading zeros.

Example 1:

Input: arr1 = [1,1,1,1,1], arr2 = [1,0,1]

Output: [1,0,0,0,0]

Explanation: arr1 represents 11, arr2 represents 5, the output represents 16.

Note:

1 <= arr1.length <= 1000

1 <= arr2.length <= 1000

arr1 and arr2 have no leading zeros

arr1[i] is 0 or 1

arr2[i] is 0 or 1

'''

```
'''  
Given a matrix, and a target, return the number of non-empty submatrices  
that sum to target.
```

A submatrix  $x_1, y_1, x_2, y_2$  is the set of all cells  $\text{matrix}[x][y]$  with  $x_1 \leq x \leq x_2$  and  $y_1 \leq y \leq y_2$ .

Two submatrices  $(x_1, y_1, x_2, y_2)$  and  $(x'_1, y'_1, x'_2, y'_2)$  are different  
if they have some coordinate that is different: for example, if  $x_1 \neq x'_1$ .

Example 1:

Input:  $\text{matrix} = [[0,1,0],[1,1,1],[0,1,0]]$ , target = 0

Output: 4

Explanation: The four  $1 \times 1$  submatrices that only contain 0.

Example 2:

Input:  $\text{matrix} = [[1,-1],[-1,1]]$ , target = 0

Output: 5

Explanation: The two  $1 \times 2$  submatrices, plus the two  $2 \times 1$  submatrices, plus  
the  $2 \times 2$  submatrix.

Note:

```
1 <= matrix.length <= 300  
1 <= matrix[0].length <= 300  
-1000 <= matrix[i] <= 1000  
-10^8 <= target <= 10^8  
'''
```

```
'''
```

Given words first and second, consider occurrences in some text of the form "first second third", where second comes immediately after first, and third comes immediately after second.

For each such occurrence, add "third" to the answer, and return the answer.

Example 1:

```
Input: text = "alice is a good girl she is a good student", first = "a",
second = "good"
```

```
Output: ["girl", "student"]
```

Example 2:

```
Input: text = "we will we will rock you", first = "we", second = "will"
```

```
Output: ["we", "rock"]
```

Note:

```
1 <= text.length <= 1000
text consists of space separated words, where each word consists of
lowercase English letters.
```

```
1 <= first.length, second.length <= 10
```

```
first and second consist of lowercase English letters.
```

```
'''
```

```
class Solution(object):
    def findOccurrences(self, text, first, second):
        """
        :type text: str
        :type first: str
        :type second: str
        :rtype: List[str]
        """

        result = []
        if not text:
            return []
        splitted_text = text.split(' ')
        indi = 0
        for index in range(len(splitted_text)-1):
            if splitted_text[index] == first and splitted_text[index+1]
== second:
                index = index+2
                if index < len(splitted_text):
                    result.append(splitted_text[index])
return result
```

'''

You have a set of tiles, where each tile has one letter tiles[i] printed on it. Return the number of possible non-empty sequences of letters you can make.

Example 1:

Input: "AAB"

Output: 8

Explanation: The possible sequences are "A", "B", "AA", "AB", "BA", "AAB", "ABA", "BAA".

Example 2:

Input: "AAABBC"

Output: 188

'''

```
class Solution(object):
    def numTilePossibilities(self, tiles):
        """
        :type tiles: str
        :rtype: int
        """

        if not tiles:
            return 0

        import collections
        unique = set(tiles)
        freq_map = collections.Counter(tiles)
        total_len = 1
        while total_len < len(tiles):
            new = set()
            for char in tiles:
                for comb in unique:
                    new_seq = comb+char
                    up_freq = collections.Counter(new_seq)
                    flag = True
                    for key, val in up_freq.items():
                        if val > freq_map[key]:
                            flag = False
                    if flag:
                        new.add(new_seq)
            # print new
            unique.update(new)

            total_len += 1
        return len(unique)
```

```
'''
```

```
Given the root of a binary tree, consider all root to leaf paths: paths  
from the root to any leaf. (A leaf is a node with no children.)
```

```
A node is insufficient if every such root to leaf path intersecting this  
node has sum strictly less than limit.
```

```
Delete all insufficient nodes simultaneously, and return the root of the  
resulting binary tree.
```

```
Example 1:
```

```
Input: root = [1,2,3,4,-99,-99,7,8,9,-99,-99,12,13,-99,14], limit = 1
```

```
Output: [1,2,3,4,null,null,7,8,9,null,14]
```

```
Example 2:
```

```
Input: root = [5,4,8,11,null,17,4,7,1,null,null,5,3], limit = 22
```

```
Output: [5,4,8,11,null,17,4,7,null,null,null,5]
```

```
Example 3:
```

```
Input: root = [1,2,-3,-5,null,4,null], limit = -1
```

```
Output: [1,null,-3,4]
```

```
Note:
```

```
The given tree will have between 1 and 5000 nodes.
```

```
-10^5 <= node.val <= 10^5
```

```
-10^9 <= limit <= 10^9
```

```
'''
```

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def sufficientSubset(self, root, limit):
        """
        :type root: TreeNode
        :type limit: int
        :rtype: TreeNode
        """
        def reduce_tree(root, limit, curr_sum):
            if not root:
                return None
```

```
l_sum = [curr_sum[0] + root.val]
r_sum = [l_sum[0]]

root.left = reduce_tree(root.left, limit, l_sum)
root.right = reduce_tree(root.right, limit, r_sum)

curr_sum[0] = max(l_sum[0], r_sum[0])
if curr_sum[0] < limit:
    root = None
return root
curr_sum = [0]
return reduce_tree(root, limit, curr_sum)
```

```
'''
```

```
Return the lexicographically smallest subsequence of text that contains  
all the distinct characters of text exactly once.
```

Example 1:

```
Input: "cdadabcc"
```

```
Output: "adbc"
```

Example 2:

```
Input: "abcd"
```

```
Output: "abcd"
```

Example 3:

```
Input: "ecbacba"
```

```
Output: "eacb"
```

Example 4:

```
Input: "leetcode"
```

```
Output: "letcod"
```

Note:

```
1 <= text.length <= 1000
text consists of lowercase English letters.
'''

class Solution(object):
    def smallestSubsequence(self, text):
        """
        :type text: str
        :rtype: str
        """
        if not text:
            return ''
        import collections
        freq_map = collections.Counter(text)
        used = [False]*26
        result = ''

        for char in text:
            freq_map[char] -= 1
            if used[ord(char)-97]:
                continue
            while (result and result[-1] > char and freq_map[result[-1]] > 0):
                used[ord(result[-1])-97] = False
                result = result[:-1]

            used[ord(char)-97] = True
            result += char
        return result
```

'''

Given an array A of positive integers, let S be the sum of the digits of the minimal element of A.

Return 0 if S is odd, otherwise return 1.

Example 1:

Input: [34, 23, 1, 24, 75, 33, 54, 8]

Output: 0

Explanation:

The minimal element is 1, and the sum of those digits is  $S = 1$  which is odd, so the answer is 0.

Example 2:

Input: [99, 77, 33, 66, 55]

Output: 1

Explanation:

The minimal element is 33, and the sum of those digits is  $S = 3 + 3 = 6$  which is even, so the answer is 1.

Note:

```
1 <= A.length <= 100
1 <= A[i].length <= 100
'''
class Solution(object):
    def sumOfDigits(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        if not A:
            return 0

        mini = min(A)
        result = 0
        while mini > 0:
            quo = mini%10
            rem = mini/10
            result += quo
            mini = rem

        return 0 if result%2 else 1
```

```
'''
```

```
Given a list of scores of different students, return the average score of  
each student's top five scores in the order of each student's id.
```

```
Each entry items[i] has items[i][0] the student's id, and items[i][1] the  
student's score. The average score is calculated using integer division.
```

```
Example 1:
```

```
Input:
```

```
[[1,91],[1,92],[2,93],[2,97],[1,60],[2,77],[1,65],[1,87],[1,100],[2,100],  
[2,76]]
```

```
Output: [[1,87],[2,88]]
```

```
Explanation:
```

```
The average of the student with id = 1 is 87.
```

```
The average of the student with id = 2 is 88.6. But with integer division  
their average converts to 88.
```

```
Note:
```

```
1 <= items.length <= 1000  
items[i].length == 2
```

```
The IDs of the students is between 1 to 1000
```

```
The score of the students is between 1 to 100
```

```
For each student, there are at least 5 scores
```

```
'''
```

```
class Solution(object):  
    def highFive(self, items):  
        """  
        :type items: List[List[int]]  
        :rtype: List[List[int]]  
        """  
  
        if not items:  
            return []  
  
        score_map = {}  
        for item in items:  
            if item[0] in score_map:  
                score_map[item[0]].append(item[1])  
            else:  
                score_map[item[0]] = [item[1]]  
  
        result = []  
        for key, value in score_map.items():  
            value.sort(reverse=True)  
            if len(value) >= 5:  
                average = value[:5]  
            else:  
                average = value  
            score_map[key] = sum(average)/len(average)  
            result.append([key, score_map[key]])  
  
        return result
```



A string S represents a list of words.

Each letter in the word has 1 or more options. If there is one option, the letter is represented as is. If there is more than one option, then curly braces delimit the options. For example, "{a,b,c}" represents options ["a", "b", "c"].

For example, "`{a,b,c}d{e,f}`" represents the list `["ade", "adf", "bde", "bdf", "cde", "cdf"]`.

Return all words that can be formed in this manner, in lexicographical order.

### Example 1:

Input: "{a,b}c{d,e}f"  
Output: ["acdf", "acef", "bcdf", "bcef"]  
Example 2:

Input: "abcd"  
Output: ["abcd"]

### Note:

$1 \leq S.length \leq 50$   
There are no nested curly brackets.  
All characters inside a pair of consecutive opening and ending curly brackets are different.

```
class Solution(object):
    def permute(self, S):
        """
        :type S: str
        :rtype: List[str]
        """

        if not S:
            return []
        if '{' not in S:
            return [S]
        stack, stack2 = [], []
        brace = 0
        for char in S:
            if char == '{':
                brace = 1
            elif char == '}':
                if not stack:
                    stack = stack2
                else:
                    new_stack = []
                    for char in stack:
                        for char2 in stack2:
                            new_stack.append(char + char2)
                    stack = new_stack
            brace -= 1
            if brace < 0:
                return []
        return stack
```

```
        stack = new_stack
stack2 = []
brace = 2
elif char != ',',':
    if brace == 1:
        stack2.append(char)
    elif brace == 2:
        stack = [c + char for c in stack]
        stack2 = []
    else:
        stack.append(char)
    # print stack

stack.sort()
stack.sort(key = len)
return stack
```

```
'''
```

We can rotate digits by 180 degrees to form new digits. When 0, 1, 6, 8, 9 are rotated 180 degrees, they become 0, 1, 9, 8, 6 respectively. When 2, 3, 4, 5 and 7 are rotated 180 degrees, they become invalid.

A confusing number is a number that when rotated 180 degrees becomes a different number with each digit valid. (Note that the rotated number can be greater than the original number.)

Given a positive integer N, return the number of confusing numbers between 1 and N inclusive.

Example 1:

Input: 20

Output: 6

Explanation:

The confusing numbers are [6, 9, 10, 16, 18, 19].

6 converts to 9.

9 converts to 6.

10 converts to 01 which is just 1.

16 converts to 91.

18 converts to 81.

19 converts to 61.

Example 2:

Input: 100

Output: 19

Explanation:

The confusing numbers are

[6, 9, 10, 16, 18, 19, 60, 61, 66, 68, 80, 81, 86, 89, 90, 91, 98, 99, 100].

Note:

```
1 <= N <= 10^9
```

```
'''
```

```
class Solution(object):
    result = 0
    def confusingNumberII(self, N):
        """
        :type N: int
        :rtype: int
        """
        original_a = [0, 1, 6, 8, 9]
        o_rotation = [0, 1, 9, 8, 6]

        def recursive(original, rotation, digit, N):
            if original > N:
                return
            if original and original != rotation:
                self.result += 1

            start = original == 0
            if digit >= 1000000000:
```

```
        return
    for index in range(start, 5):
        recursive(original * 10 + original_a[index], rotation +
o_rotation[index]*digit, digit*10, N)

recursive(0, 0, 1, N)
if (N == 1000000000):
    self.result += 1
return self.result
```

'''

Given a fixed length array arr of integers, duplicate each occurrence of zero, shifting the remaining elements to the right.

Note that elements beyond the length of the original array are not written.

Do the above modifications to the input array in place, do not return anything from your function.

Example 1:

Input: [1,0,2,3,0,4,5,0]

Output: null

Explanation: After calling your function, the input array is modified to:  
[1,0,0,2,3,0,0,4]

Example 2:

Input: [1,2,3]

Output: null

Explanation: After calling your function, the input array is modified to:  
[1,2,3]

Note:

1 <= arr.length <= 10000

0 <= arr[i] <= 9

'''

```
class Solution(object):
    def duplicateZeros(self, arr):
        """
        :type arr: List[int]
        :rtype: None Do not return anything, modify arr in-place instead.
        """
        arr_copy = arr[:]
        index, n = 0, len(arr_copy)
        for elem in arr_copy:
            arr[index] = elem
            index += 1
            if index >= n:
                break
            if elem == 0:
                arr[index] = elem
                index += 1
                if index >= n:
                    break
```

```
'''
```

```
We have a set of items: the i-th item has value values[i] and label  
labels[i].
```

```
Then, we choose a subset S of these items, such that:
```

```
|S| <= num_wanted
```

```
For every label L, the number of items in S with label L is <= use_limit.  
Return the largest possible sum of the subset S.
```

```
Example 1:
```

```
Input: values = [5,4,3,2,1], labels = [1,1,2,2,3], num_wanted = 3,  
use_limit = 1
```

```
Output: 9
```

```
Explanation: The subset chosen is the first, third, and fifth item.
```

```
Example 2:
```

```
Input: values = [5,4,3,2,1], labels = [1,3,3,3,2], num_wanted = 3,  
use_limit = 2
```

```
Output: 12
```

```
Explanation: The subset chosen is the first, second, and third item.
```

```
Example 3:
```

```
Input: values = [9,8,8,7,6], labels = [0,0,0,1,1], num_wanted = 3,  
use_limit = 1
```

```
Output: 16
```

```
Explanation: The subset chosen is the first and fourth item.
```

```
Example 4:
```

```
Input: values = [9,8,8,7,6], labels = [0,0,0,1,1], num_wanted = 3,  
use_limit = 2
```

```
Output: 24
```

```
Explanation: The subset chosen is the first, second, and fourth item.
```

```
Note:
```

```
1 <= values.length == labels.length <= 20000  
0 <= values[i], labels[i] <= 20000  
1 <= num_wanted, use_limit <= values.length  
'''  
class Solution(object):  
    def largestValsFromLabels(self, values, labels, num_wanted,  
    use_limit):  
        """  
        :type values: List[int]  
        :type labels: List[int]  
        :type num_wanted: int  
        :type use_limit: int  
        :rtype: int  
        """  
        sorted_values = sorted([(i, j) for i, j in zip(values, labels)]),  
key = lambda x : x[0]**-1  
label_used_count = {label: 0 for label in set(labels)}  
result = 0
```

```
for s_v in sorted_values:
    if num_wanted:
        if label_used_count[s_v[1]] < use_limit:
            result += s_v[0]
            label_used_count[s_v[1]] +=1
            num_wanted -= 1
    else:
        break
return result
```

'''

In an N by N square grid, each cell is either empty (0) or blocked (1).

A clear path from top-left to bottom-right has length k if and only if it is composed of cells C\_1, C\_2, ..., C\_k such that:

Adjacent cells C\_i and C\_{i+1} are connected 8-directionally (ie., they are different and share an edge or corner)  
C\_1 is at location (0, 0) (ie. has value grid[0][0])  
C\_k is at location (N-1, N-1) (ie. has value grid[N-1][N-1])  
If C\_i is located at (r, c), then grid[r][c] is empty (ie. grid[r][c] == 0).

Return the length of the shortest such clear path from top-left to bottom-right. If such a path does not exist, return -1.

Example 1:

Input: [[0,1],[1,0]]

Output: 2

Example 2:

Input: [[0,0,0],[1,1,0],[1,1,0]]

Output: 4

Note:

```
1 <= grid.length == grid[0].length <= 100
grid[r][c] is 0 or 1
'''
class Solution(object):
    def shortestPathBinaryMatrix(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        if not grid:
            return -1

        rows, cols = len(grid), len(grid[0])
        if grid[0][0] or grid[rows-1][cols-1]:
            return -1

        queue = [[0, 0, 1]]
        for row, col, dist in queue:
            if row == rows-1 and col == cols-1:
                return dist
            for di, dj in [(-1, -1), (0, -1), (-1, 1), (-1, 0), (1, 0),
(1, -1), (0, 1), (1, 1)]:
                n_row, n_col = row + di, col + dj
                if 0 <= n_row < rows and 0 <= n_col < cols and not
grid[n_row][n_col]:
                    grid[n_row][n_col] = 1
                    queue.append([n_row, n_col, dist + 1])

return -1
```



```
'''
```

Given two strings str1 and str2, return the shortest string that has both str1 and str2 as subsequences. If multiple answers exist, you may return any of them.

(A string S is a subsequence of string T if deleting some number of characters from T (possibly 0, and the characters are chosen anywhere from T) results in the string S.)

Example 1:

```
Input: str1 = "abac", str2 = "cab"
Output: "cabac"
Explanation:
str1 = "abac" is a substring of "cabac" because we can delete the first "c".
str2 = "cab" is a substring of "cabac" because we can delete the last "ac".
The answer provided is the shortest such string that satisfies these properties.
```

Note:

```
1 <= str1.length, str2.length <= 1000
str1 and str2 consist of lowercase English letters.
'''
```

```
class Solution(object):
    def shortestCommonSupersequence(self, str1, str2):
        """
        :type str1: str
        :type str2: str
        :rtype: str
        """
        def lcs(A, B):
            n, m = len(A)+1, len(B)+1
            dp = [["" for _ in range(m)] for _ in range(n)]
            for index_i in range(1, n):
                for index_j in range(1, m):
                    if A[index_i-1] == B[index_j-1]:
                        dp[index_i][index_j] = dp[index_i-1][index_j-1] +
A[index_i - 1]
                    else:
                        dp[index_i][index_j] = max(dp[index_i-
1][index_j], dp[index_i][index_j-1], key=len)
            return dp[-1][-1]

        result = ""
        index_i, index_j = 0, 0
        for s in lcs(str1, str2):
            while str1[index_i] != s:
                result += str1[index_i]
                index_i += 1
            while str2[index_j] != s:
                result += str2[index_j]
                index_j += 1
```

```
    index_j += 1  
    result += s  
    index_i, index_j = index_i+1, index_j+1  
  
return result + str1[index_i:] + str2[index_j:]
```

```
'''
```

```
A bus has n stops numbered from 0 to n - 1 that form a circle. We know  
the distance between all pairs of neighboring stops where distance[i] is  
the distance between the stops number i and (i + 1) % n.
```

```
The bus goes along both directions i.e. clockwise and counterclockwise.
```

```
Return the shortest distance between the given start and destination  
stops.
```

Example 1:

```
Input: distance = [1,2,3,4], start = 0, destination = 1
```

```
Output: 1
```

```
Explanation: Distance between 0 and 1 is 1 or 9, minimum is 1.
```

Example 2:

```
Input: distance = [1,2,3,4], start = 0, destination = 2
```

```
Output: 3
```

```
Explanation: Distance between 0 and 2 is 3 or 7, minimum is 3.
```

Example 3:

```
Input: distance = [1,2,3,4], start = 0, destination = 3
```

```
Output: 4
```

```
Explanation: Distance between 0 and 3 is 6 or 4, minimum is 4.
```

Constraints:

```
1 <= n <= 10^4  
distance.length == n  
0 <= start, destination < n  
0 <= distance[i] <= 10^4  
'''
```

```
class Solution(object):  
    def distanceBetweenBusStops(self, distance, start, destination):  
        """  
        :type distance: List[int]  
        :type start: int  
        :type destination: int  
        :rtype: int  
        """  
        start, destination = min(start, destination), max(start,  
destination)  
        clock_dist = sum(distance[start:destination])  
        anti_clock_dist = sum(distance[:start]) +  
sum(distance[destination:])  
        return min(clock_dist, anti_clock_dist)
```

```
'''
```

```
Given a date, return the corresponding day of the week for that date.
```

```
The input is given as three integers representing the day, month and year respectively.
```

```
Return the answer as one of the following values {"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"}.
```

```
Example 1:
```

```
Input: day = 31, month = 8, year = 2019
```

```
Output: "Saturday"
```

```
Example 2:
```

```
Input: day = 18, month = 7, year = 1999
```

```
Output: "Sunday"
```

```
Example 3:
```

```
Input: day = 15, month = 8, year = 1993
```

```
Output: "Sunday"
```

```
Constraints:
```

```
The given dates are valid dates between the years 1971 and 2100.
```

```
'''
```

```
class Solution(object):
    def dayOfTheWeek(self, day, month, year):
        """
        :type day: int
        :type month: int
        :type year: int
        :rtype: str
        """
        day_of_week_map = ["Sunday", "Monday", "Tuesday", "Wednesday",
"Thursday", "Friday", "Saturday"]
        t = [ 0, 3, 2, 5, 0, 3, 5, 1, 4, 6, 2, 4 ]
        year -= month < 3
        return day_of_week_map[((year + int(year / 4) - int(year / 100)
+ int(year / 400) + t[month - 1] + day) % 7)]
```

```
'''
```

Given an array of integers, return the maximum sum for a non-empty subarray (contiguous elements) with at most one element deletion. In other words, you want to choose a subarray and optionally delete one element from it so that there is still at least one element left and the sum of the remaining elements is maximum possible.

Note that the subarray needs to be non-empty after deleting one element.

Example 1:

Input: arr = [1,-2,0,3]

Output: 4

Explanation: Because we can choose [1, -2, 0, 3] and drop -2, thus the subarray [1, 0, 3] becomes the maximum value.

Example 2:

Input: arr = [1,-2,-2,3]

Output: 3

Explanation: We just choose [3] and it's the maximum sum.

Example 3:

Input: arr = [-1,-1,-1,-1]

Output: -1

Explanation: The final subarray needs to be non-empty. You can't choose [-1] and delete -1 from it, then get an empty subarray to make the sum equals to 0.

Constraints:

```
1 <= arr.length <= 10^5
```

```
-10^4 <= arr[i] <= 10^4
```

```
'''
```

```
class Solution(object):
    def maximumSum(self, arr):
        """
        :type arr: List[int]
        :rtype: int
        """
        forward = [0] * len(arr)
        backward = [0] * len(arr)

        curr_max, max_so_far = arr[0], arr[0]
        forward[0] = arr[0]
        for index in range(1, len(arr)):
            curr_max = max(arr[index], curr_max + arr[index])
            max_so_far = max(max_so_far, curr_max)

            forward[index] = curr_max

        curr_max = arr[len(arr) - 1]
        max_so_far = arr[len(arr) - 1]
        backward[len(arr) - 1] = arr[len(arr) - 1]
```

```
index = len(arr) - 2
while index >= 0:
    curr_max = max(arr[index], curr_max + arr[index])
    max_so_far = max(max_so_far, curr_max)

    backward[index] = curr_max
    index -= 1

result = max_so_far
for index in range(1, len(arr)-1):
    result = max(result, forward[index-1] + backward[index + 1])
return result
```

```
'''
```

```
Given a string text, you want to use the characters of text to form as many instances of the word "balloon" as possible. You can use each character in text at most once. Return the maximum number of instances that can be formed.
```

```
Example 1:
```

```
Input: text = "nlaebolko"
```

```
Output: 1
```

```
Example 2:
```

```
Input: text = "loonbalxballpoon"
```

```
Output: 2
```

```
Example 3:
```

```
Input: text = "leetcode"
```

```
Output: 0
```

```
Constraints:
```

```
1 <= text.length <= 10^4
```

```
text consists of lower case English letters only.
```

```
'''
```

```
class Solution(object):
    def maxNumberOfBalloons(self, text):
        """
        :type text: str
        :rtype: int
        """
        if not text:
            return 0

        import collections
        cnt = collections.Counter(text)
        cnt_balloon = collections.Counter('balloon')

        return min([cnt[c]//cnt_balloon[c] for c in cnt_balloon])
```

```
'''
```

```
Given a string s that consists of lower case English letters and  
brackets.
```

```
Reverse the strings in each pair of matching parentheses, starting from  
the innermost one.
```

```
Your result should not contain any bracket.
```

```
Example 1:
```

```
Input: s = "(abcd)"
```

```
Output: "dcba"
```

```
Example 2:
```

```
Input: s = "(u(love)i)"
```

```
Output: "iloveu"
```

```
Example 3:
```

```
Input: s = "(ed(et(oc))el)"
```

```
Output: "leetcode"
```

```
Example 4:
```

```
Input: s = "a(bcdefghijkl(mno)p)q"
```

```
Output: "apmnolkjihgfedcbq"
```

```
Constraints:
```

```
0 <= s.length <= 2000
```

```
s only contains lower case English characters and parentheses.
```

```
It's guaranteed that all parentheses are balanced.
```

```
'''
```

```
class Solution(object):  
    def reverseParentheses(self, s):  
        """  
        :type s: str  
        :rtype: str  
        """  
        if not s:  
            return ''  
  
        stack = []  
        for char in s:  
            if char == ')':  
                combine_str = ''  
                while stack and stack[-1] != '(':  
                    elem = stack.pop()[:-1]  
                    combine_str += elem  
                stack.pop()  
                stack.append(combine_str)  
            else:  
                stack.append(char)  
        return ''.join(stack)
```

'''

Given an integer array arr and an integer k, modify the array by repeating it k times.

For example, if arr = [1, 2] and k = 3 then the modified array will be [1, 2, 1, 2, 1, 2].

Return the maximum sub-array sum in the modified array. Note that the length of the sub-array can be 0 and its sum in that case is 0.

As the answer can be very large, return the answer modulo  $10^9 + 7$ .

Example 1:

Input: arr = [1,2], k = 3

Output: 9

Example 2:

Input: arr = [1,-2,1], k = 5

Output: 2

Example 3:

Input: arr = [-1,-2], k = 7

Output: 0

Constraints:

```
1 <= arr.length <= 10^5
1 <= k <= 10^5
-10^4 <= arr[i] <= 10^4
"""
class Solution(object):
    def kConcatenationMaxSum(self, arr, k):
        """
        :type arr: List[int]
        :type k: int
        :rtype: int
        """
        def kadane(arr):
            curr_sum, max_sum = arr[0], arr[0]
            for index in range(1, len(arr)):
                curr_sum = max(arr[index], curr_sum + arr[index])
                max_sum = max(max_sum, curr_sum)
            return max_sum

        def prefix(arr):
            curr_sum, max_val = 0, float('-inf')
            for index, val in enumerate(arr):
                curr_sum += val
                max_val = max(max_val, curr_sum)
            return max_val

        def suffix(arr):
            curr_sum, max_val = 0, float('-inf')
            for index in range(len(arr)-1, -1, -1):
                curr_sum += val
                max_val = max(max_val, curr_sum)
            return max_val

        if k == 1:
            return kadane(arr)

        if k == 2:
            max_val = max(prefix(arr), suffix(arr))
            if max_val == float('-inf'):
                return 0
            else:
                return max_val

        if k == 3:
            max_val = max(prefix(arr), suffix(arr), kadane(arr))
            if max_val == float('-inf'):
                return 0
            else:
                return max_val

        if k > 3:
            max_val = max(prefix(arr), suffix(arr))
            if max_val == float('-inf'):
                return 0
            else:
                return max_val * (k-3) + kadane(arr)
```

```
curr_sum += arr[index]
max_val = max(max_val, curr_sum)
return max_val

if not arr:
    return 0
if k == 1:
    return max(0, kadane(arr)) % (10 ** 9 + 7)
else:
    return max(0, max((prefix(arr) + suffix(arr) + (k-2)*max(sum(arr), 0), kadane(arr)))) % (10 ** 9 + 7)
```

'''

Given an array of distinct integers arr, find all pairs of elements with the minimum absolute difference of any two elements.

Return a list of pairs in ascending order (with respect to pairs), each pair [a, b] follows

a, b are from arr  
a < b  
b - a equals to the minimum absolute difference of any two elements in arr

Example 1:

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

Explanation: The minimum absolute difference is 1. List all pairs with difference equal to 1 in ascending order.

Example 2:

Input: arr = [1,3,6,10,15]  
Output: [[1,3]]

Example 3:

Input: arr = [3,8,-10,23,19,-4,-14,27]  
Output: [[-14,-10],[19,23],[23,27]]

Constraints:

2 <= arr.length <= 10^5  
-10^6 <= arr[i] <= 10^6  
'''

```
class Solution(object):
    def minimumAbsDifference(self, arr):
        """
        :type arr: List[int]
        :rtype: List[List[int]]
        """
        if not arr:
            return []

        arr.sort()
        mindiff = arr[1] - arr[0]
        for index in range(2, len(arr)):
            mindiff = min(mindiff, (arr[index] - arr[index-1]))

        result = []
        for index in range(1, len(arr)):
            if arr[index] - arr[index-1] == mindiff:
                result.append([arr[index-1], arr[index]])
        return result
```

```
class Solution:
    def nthUglyNumber(self, n: int, a: int, b: int, c: int) -> int:
        def gcd(a, b):
            return a if not b else gcd(b, a % b)
        ab, ac, bc = a * b // gcd(a, b), a * c // gcd(a, c), b * c //
gcd(b, c)
        abc = ab * c // gcd(ab, c)
        l, r = 1, 2 * 10 ** 9
        while l < r:
            mid = (l + r) // 2
            if mid // a + mid // b + mid // c - mid // ab - mid // ac -
mid // bc + mid // abc < n:
                l = mid + 1
            else:
                r = mid
        return l
```

```
class Solution:
    def smallestStringWithSwaps(self, s: str, pairs: List[List[int]]) ->
str:
    class UF:
        def __init__(self, n): self.p = list(range(n))
        def union(self, x, y): self.p[self.find(x)] = self.find(y)
        def find(self, x):
            if x != self.p[x]: self.p[x] = self.find(self.p[x])
            return self.p[x]
    uf, res, m = UF(len(s)), [], collections.defaultdict(list)
    for x,y in pairs:
        uf.union(x,y)
    for i in range(len(s)):
        m[uf.find(i)].append(s[i])
    for comp_id in m.keys():
        m[comp_id].sort(reverse=True)
    for i in range(len(s)):
        res.append(m[uf.find(i)].pop())
    return ''.join(res)
```

```

class Solution:
    def sortItems(self, n: int, m: int, group: List[int], beforeItems: List[List[int]]) -> List[int]:
        def topo_sort(points, pre, suc):
            order = []
            sources = [p for p in points if not pre[p]]
            while sources:
                s = sources.pop()
                order.append(s)
                for u in suc[s]:
                    pre[u].remove(s)
                    if not pre[u]:
                        sources.append(u)
            return order if len(order) == len(points) else []

        # find the group of each item
        group2item = collections.defaultdict(set)
        for i in range(n):
            if group[i] == -1:
                group[i] = m
                m += 1
            group2item[group[i]].add(i)

        # find the relationships between the groups and each items in the same group
        t_pre, t_suc = collections.defaultdict(set),
        collections.defaultdict(set)
        g_pre, g_suc = collections.defaultdict(set),
        collections.defaultdict(set)
        for i in range(n):
            for j in beforeItems[i]:
                if group[i] == group[j]:
                    t_pre[i].add(j)
                    t_suc[j].add(i)
                else:
                    g_pre[group[i]].add(group[j])
                    g_suc[group[j]].add(group[i])

        # topological sort the groups
        groups_order = topo_sort([i for i in group2item], g_pre, g_suc)
        # topological sort the items in each group
        t_order = []
        for i in groups_order:
            items = group2item[i]
            i_order = topo_sort(items, t_pre, t_suc)
            if len(i_order) != len(items):
                return []
            t_order += i_order
        return t_order if len(t_order) == n else []

```

```
from collections import Counter as cnt
class Solution:
    def uniqueOccurrences(self, arr: List[int]) -> bool:
        return all(v == 1 for v in cnt(cnt(arr).values()).values())
```

```
class Solution:
    def equalSubstring(self, s: str, t: str, mx: int) -> int:
        i = 0
        for j in range(len(s)):
            mx -= abs(ord(s[j]) - ord(t[j]))
            if mx < 0:
                mx += abs(ord(s[i]) - ord(t[i]))
                i += 1
        return j - i + 1
```

```
class Solution:
    def removeDuplicates(self, s: str, k: int) -> str:
        stack = []
        for i, c in enumerate(s):
            if not stack or stack[-1][0] != c:
                stack.append([c, 1])
            else:
                stack[-1][1] += 1
            if stack[-1][1] == k:
                stack.pop()
        return ''.join(k * v for k, v in stack)
```

```

class Solution:
    def minimumMoves(self, grid: List[List[int]]) -> int:
        q, move, n, seen = {(0, 1, 0)}, 0, len(grid), set()
        while q:
            new = set()
            for i, j, hv in q:
                if i == j == n - 1 and not hv:
                    return move
                if hv and i < n - 1 and not grid[i + 1][j]:
                    if (i + 1, j, 1) not in seen:
                        new.add((i + 1, j, 1))
                if hv and j + 1 < n and grid[i][j + 1] == grid[i - 1][j + 1] == 0:
                    if (i, j + 1, 1) not in seen:
                        new.add((i, j + 1, 1))
                    if (i - 1, j + 1, 0) not in seen:
                        new.add((i - 1, j + 1, 0))
                if not hv and j + 1 < n and not grid[i][j + 1]:
                    if (i, j + 1, 0) not in seen:
                        new.add((i, j + 1, 0))
                    if not hv and i + 1 < n and grid[i + 1][j] == grid[i + 1][j - 1] == 0:
                        if (i + 1, j, 0) not in seen:
                            new.add((i + 1, j, 0))
                        if (i + 1, j - 1, 1) not in seen:
                            new.add((i + 1, j - 1, 1))
            q = new
            seen |= new
            move += 1
        return -1

```

```
class Solution:
    def arraysIntersection(self, arr1: List[int], arr2: List[int], arr3: List[int]) -> List[int]:
        return sorted(set(arr1) & set(arr2) & set(arr3))
```

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
from functools import lru_cache
class Solution:
    def twoSumBSTs(self, root1: TreeNode, root2: TreeNode, target: int) -> bool:
```

```
        def dfs(node):
            return dfs(node.left) | dfs(node.right) | {node.val} if node
        else set()
        q1 = dfs(root1)
        return any(target - a in q1 for a in dfs(root2))
```

```
class Solution:
    def countSteppingNumbers(self, low: int, high: int) -> List[int]:
        def dfs(n):
            if n > high:
                return
            if n >= low:
                q.add(n)
            d = n % 10
            if d == 0:
                dfs(n * 10 + 1)
            elif d == 9:
                dfs(n * 10 + 8)
            else:
                dfs(n * 10 + d + 1)
                dfs(n * 10 + d - 1)
        q = set()
        for i in range(10):
            dfs(i)
        return sorted(q)
```

```
class Solution:
    def isValidPalindrome(self, s: str, k: int) -> bool:
        n = len(s)
        dp = [[0] * (n + 1) for _ in range(n + 1)]
        for i in range(n + 1):
            for j in range(n + 1):
                if not i or not j:
                    dp[i][j] = i or j
                elif s[i - 1] == s[n - j]:
                    dp[i][j] = dp[i - 1][j - 1]
                else:
                    dp[i][j] = 1 + min(dp[i - 1][j], dp[i][j - 1])
        return dp[n][n] <= k * 2
```

```
class Solution:  
    def minCostToMoveChips(self, chips: List[int]) -> int:  
        return min(sum((c1 - c2) % 2 for c2 in chips) for c1 in chips)
```

```
class Solution:
    def longestSubsequence(self, arr: List[int], d: int) -> int:
        dp = collections.Counter()
        for a in arr:
            dp[a] = max(dp[a], dp[a - d] + 1)
        return max(dp.values())
```

```
class Solution:
    def getMaximumGold(self, grid: List[List[int]]) -> int:
        def dfs(i, j, v):
            seen.add((i, j))
            dp[i][j] = max(dp[i][j], v)
            for x, y in (i - 1, j), (i + 1, j), (i, j - 1), (i, j + 1):
                if 0 <= x < m and 0 <= y < n and grid[x][y] and (x, y)
not in seen:
                dfs(x, y, v + grid[x][y])
                seen.discard((i, j))
        m, n = len(grid), len(grid[0])
        dp = [[0] * n for _ in range(m)]
        for i in range(m):
            for j in range(n):
                if grid[i][j]:
                    seen = set()
                    dfs(i, j, grid[i][j])
        return max(c for row in dp for c in row)
```

```
class Solution:
    def countVowelPermutation(self, n: int) -> int:
        mod = 10 ** 9 + 7
        dp = [1] * 5
        for _ in range(n - 1):
            add = [0] * 5
            # from a
            add[1] = (add[1] + dp[0]) % mod
            # from e
            add[0] = (add[0] + dp[1]) % mod
            add[2] = (add[2] + dp[1]) % mod
            # from i
            add[0] = (add[0] + dp[2]) % mod
            add[1] = (add[1] + dp[2]) % mod
            add[3] = (add[3] + dp[2]) % mod
            add[4] = (add[4] + dp[2]) % mod
            # from o
            add[2] = (add[2] + dp[3]) % mod
            add[4] = (add[4] + dp[3]) % mod
            # from u
            add[0] = (add[0] + dp[4]) % mod
        for i in range(5):
            dp[i] = add[i] % mod
        return sum(dp) % mod
```

```
class Solution:
    def balancedStringSplit(self, s: str) -> int:
        res = cnt = 0
        for c in s:
            cnt += c == 'L'
            cnt -= c == 'R'
            res += cnt == 0
        return res
```

```
class Solution:
    def queensAttacktheKing(self, queens: List[List[int]], king:
List[int]) -> List[List[int]]:
        def dfs(dr, dc, r, c):
            while 0 <= r <= 7 and 0 <= c <= 7:
                if (r, c) in q:
                    res.append([r, c])
                    break
                r += dr
                c += dc
        q = set((r,c) for r, c in queens)
        res = []
        for dr, dc in (-1, -1), (-1, 0), (-1, 1), (0, 1),
(1, 1), (1, 0),
(1, -1), (0, -1):
            dfs(dr, dc, *king)
        return res
```

```
class Solution:
    def dieSimulator(self, n: int, r: List[int]) -> int:
        K = max(r)
        dp = [[[0 for k in range(K)] for j in range(6)] for i in
range(n)]
        for j in range(6): dp[0][j][0] = 1
        for i in range(1, n):
            for j in range(6):
                dp[i][j][0] += sum(dp[i-1][t][k] for t in range(6) for k
in range(r[t])) if t != j
                for k in range(1, r[j]):
                    dp[i][j][k] = dp[i-1][j][k-1]
        return sum(dp[n-1][j][k] for j in range(6) for k in range(K)) %
(10**9+7)
```

```
class Solution:
    def maxEqualFreq(self, nums: List[int]) -> int:
        def okay():
            if len(dic) == 1 and (1 in dic or 1 in dic.values()):
                return True
            if len(dic) == 2:
                c1, c2 = sorted(dic.keys())
                if c2 - c1 == 1 and dic[c2] == 1 or (c1 == 1 and dic[1]
== 1):
                    return True
            cnt = collections.Counter(nums)
            dic = collections.Counter(cnt.values())
            l = len(nums)
            for num in nums[::-1]:
                if okay():
                    return 1
                dic[cnt[num]] -= 1
                if not dic[cnt[num]]:
                    dic.pop(cnt[num])
                cnt[num] -= 1
                if cnt[num]:
                    dic[cnt[num]] += 1
                l -= 1
            if okay():
                return 1
```

```
class Solution:  
    def nthPersonGetsNthSeat(self, n: int) -> float:  
        return max(0.5, 1 / n)
```

```
class Solution:
    def missingNumber(self, arr: List[int]) -> int:
        d = (arr[-1] - arr[0]) // len(arr)
        for a, b in zip(arr, arr[1:]):
            if b != a + d:
                return a + d
    return 0
```

```
class Solution:
    def minAvailableDuration(self, s1: List[List[int]], s2:
List[List[int]], d: int) -> List[int]:
        s2.sort()
        j = 0
        for s, e in sorted(s1):
            while j < len(s2) - 1 and s2[j][1] < s:
                j += 1
            if s2[j][0] <= e:
                l, r = max(s, s2[j][0]), min(e, s2[j][1])
                if r - l >= d:
                    return [l, l + d]
```

```
class Solution:
    def probabilityOfHeads(self, p: List[float], t: int) -> float:
        n = len(p)
        dp = [[0] * (n + 1) for _ in range(n + 1)]
        dp[0][0] = 1
        for i in range(1, n + 1):
            for j in range(i + 1):
                if j == 0:
                    dp[i][j] = dp[i - 1][j] * (1 - p[i - 1])
                else :
                    dp[i][j] = (dp[i - 1][j] * (1.0 - p[i - 1])) + (dp[i - 1][j - 1] * p[i - 1])
        return dp[-1][t]
```

```
class Solution:
    def maximizeSweetness(self, sw: List[int], K: int) -> int:
        def ok(m):
            c = sm = 0
            for s in sw:
                sm += s
                if sm >= m:
                    sm = 0
                    c += 1
            return c >= K + 1
        l, r = 1, sum(sw)
        while l < r:
            m = (l + r) // 2
            if ok(m):
                l = m + 1
            else:
                r = m - 1
        print(l, r)
        return r if ok(r) else l - 1
```

```
class Solution:
    def checkStraightLine(self, c: List[List[int]]) -> bool:
        return len(set((b[0] - a[0]) / (b[1] - a[1]) for a, b in zip(c, c[1:])) == 1
```

```
class Solution:
    def removeSubfolders(self, folder: List[str]) -> List[str]:
        st = set(folder)
        for f in folder:
            if any(p in st for p in itertools.accumulate(f.split('/'),
lambda x, y: x + '/' + y) if p and p != f):
                st.discard(f)
        return list(st)
```

```
class Solution:
    def balancedString(self, s: str) -> int:
        cnt, i, res = {c: max(s.count(c) - len(s) // 4, 0) for c in 'QWER'}, 0, len(s)
        for j, c in enumerate(s):
            cnt[c] -= 1
            while i < len(s) and cnt[s[i]] < 0:
                cnt[s[i]] += 1
                i += 1
            if not any(cnt[c] > 0 for c in 'QWER'):
                res = min(res, j - i + 1)
        return res
```

```
class Solution:
    def jobScheduling(self, startTime: List[int], endTime: List[int],
profit: List[int]) -> int:
        jobs = sorted(zip(startTime, endTime, profit), key=lambda v:
v[1])
        dp = [[0, 0]]
        for s, e, p in jobs:
            i = bisect.bisect(dp, [s + 1]) - 1
            if dp[i][1] + p > dp[-1][1]:
                dp.append([e, dp[i][1] + p])
        return dp[-1][1]
```

```
# """
# This is HtmlParser's API interface.
# You should not implement it, or speculate about its implementation
# """
#class HtmlParser(object):
#    def getUrls(self, url):
#        """
#            :type url: str
#            :rtype List[str]
#        """
#
#class Solution:
#    def crawl(self, startUrl: str, htmlParser: 'HtmlParser') ->
List[str]:
        host = startUrl[::(startUrl + '/').find('/', startUrl.find('///') + 2)]
        q, seen = [startUrl], {startUrl}
        for url in q:
            for nex in htmlParser.getUrls(url):
                if nex[::(nex + '/').find('/', nex.find('///') + 2)] ==
host and nex not in seen:
                    q.append(nex)
                    seen.add(nex)
    return q
```

```
from itertools import product as pr

class Solution(object):
    def findSolution(self, customfunction, z):
        return [
            [i, j]
            for i, j in pr(range(1, z + 1), repeat=2)
            if customfunction.f(i, j) == z
        ]
```

```
class Solution:  
    def circularPermutation(self, n: int, start: int) -> List[int]:  
        return [start ^ i ^ i >> 1 for i in range(1 << n)]
```

```
class Solution:
    def maxLength(self, arr: List[str]) -> int:
        bfs = [""]
        for b in filter(lambda x: len(x) == len(set(x)), arr):
            bfs += [a + b for a in bfs if not set(a) & set(b)]
        return max(map(len, bfs))
```

```
class Solution:
    memo = {}

    def tilingRectangle(self, n: int, m: int) -> int:
        if (n, m) in {(11, 13), (13, 11)}:
            return 6
        if n == m:
            return 1
        if (n, m) not in self.memo:
            nMin = mMin = float("inf")
            for i in range(1, n // 2 + 1):
                nMin = min(
                    nMin, self.tilingRectangle(i, m) +
self.tilingRectangle(n - i, m)
                )
                for j in range(1, m // 2 + 1):
                    mMin = min(
                        mMin, self.tilingRectangle(n, j) +
self.tilingRectangle(n, m - j)
                    )
            self.memo[(n, m)] = min(nMin, mMin)
        return self.memo[(n, m)]
```

```
class Solution:
    def transformArray(self, arr: List[int], change: bool = True) ->
List[int]:
        while change:
            new = (
                arr[:1]
                + [
                    b + (a > b < c) - (a < b > c)
                    for a, b, c in zip(arr, arr[1:], arr[2:])]
                ]
                + arr[-1:]
            )
            arr, change = new, arr != new
        return arr
```

```
class Leaderboard:
    def __init__(self):
        self.scores = collections.defaultdict(set)
        self.p = collections.defaultdict(int)

    def addScore(self, playerId: int, score: int) -> None:
        self.scores[self.p[playerId]].discard(playerId)
        self.p[playerId] += score
        self.scores[self.p[playerId]].add(playerId)

    def top(self, K: int) -> int:
        sm = cnt = 0
        for score, players in sorted(self.scores.items())[::-1]:
            if len(players) + cnt <= K:
                sm += len(players) * score
                cnt += len(players)
            else:
                sm += (K - cnt) * score
                cnt = K
        return sm

    def reset(self, playerId: int) -> None:
        self.scores[self.p[playerId]].discard(playerId)
        self.p[playerId] = 0
        self.scores[0].add(playerId)
```

```
class Solution:
    def treeDiameter(self, edges: List[List[int]], move: int = 0) -> int:
        graph = collections.defaultdict(set)
        for a, b in edges:
            graph[a].add(b)
            graph[b].add(a)
        bfs = {(u, None) for u, nex in graph.items() if len(nex) == 1}
        while bfs:
            bfs, move = (
                {(v, u) for u, pre in bfs for v in graph[u] if v != pre},
                move + 1,
            )
        return max(move - 1, 0)
```

```
class Solution:
    def minimumMoves(self, arr: List[int]) -> int:
        n = len(arr)
        dp = [[0] * (n + 1) for _ in range(n + 1)]
        for l in range(1, n + 1):
            i, j = 0, l - 1
            while j < n:
                if l == 1:
                    dp[i][j] = 1
                else:
                    dp[i][j] = 1 + dp[i + 1][j]
                    if arr[i] == arr[i + 1]:
                        dp[i][j] = min(1 + dp[i + 2][j], dp[i][j])
                    for k in range(i + 2, j + 1):
                        if arr[i] == arr[k]:
                            dp[i][j] = min(dp[i + 1][k - 1] + dp[k + 1][j], dp[i][j])
                i, j = i + 1, j + 1
        return dp[0][n - 1]
```

```
class Solution:
    def minimumSwap(self, s1: str, s2: str, xy: int = 0, yx: int = 0) ->
int:
    for a, b in zip(s1, s2):
        xy += a == "x" and b == "y"
        yx += a == "y" and b == "x"
    return (xy + yx) // 2 + (xy % 2) * 2 if xy % 2 == yx % 2 else -1
```

```
class Solution:
    def numberOfSubarrays(self, nums: List[int], k: int, cnt: int = 0) ->
int:
        odds = [-1] + [i for i, num in enumerate(nums) if num % 2] +
[len(nums)]
        return sum(
            (odds[j - k + 1] - odds[j - k]) * (odds[j + 1] - odds[j])
            for j in range(k, len(odds) - 1)
        )
```

```
class Solution:
    def minRemoveToMakeValid(
        self, s: str, res: str = "", l: str = "(", r: str = ")", b: int =
0
    ) -> str:
        for _ in range(2):
            for c in s:
                if c == r and b <= 0:
                    continue
                b += c == l
                b -= c == r
                res += c
            res, s, l, r, b = "", res[::-1], r, l, 0
    return s
```

```
from functools import reduce

class Solution:
    def isGoodArray(self, nums: List[int]) -> bool:
        return reduce(math.gcd, nums) == 1
```

```
from collections import Counter as cnt

class Solution:
    def oddCells(self, n: int, m: int, indices: List[List[int]]) -> int:
        row, col = cnt(r for r, c in indices), cnt(c for r, c in indices)
        return sum((row[i] + col[j]) % 2 for i in range(n) for j in
range(m))
```

```
class Solution:
    def reconstructMatrix(
        self, upper: int, lower: int, colsum: List[int]
    ) -> List[List[int]]:
        res = [[0] * len(colsum) for _ in range(2)]
        for j, sm in enumerate(colsum):
            if sm == 2:
                if upper == 0 or lower == 0:
                    return []
                upper -= 1
                lower -= 1
                res[0][j] = res[1][j] = 1
            elif sm:
                if upper == lower == 0:
                    return []
                if upper >= lower:
                    upper -= 1
                    res[0][j] = 1
                else:
                    lower -= 1
                    res[1][j] = 1
        return res if upper == lower == 0 else []
```

```
class Solution:
    def closedIsland(self, grid: List[List[int]]) -> int:
        def dfs(i, j, ret=True):
            grid[i][j] = -1
            for x, y in (i - 1, j), (i + 1, j), (i, j - 1), (i, j + 1):
                if 0 <= x < m and 0 <= y < n:
                    if not grid[x][y]:
                        ret &= dfs(x, y)
                    else:
                        ret = False
            return ret

        m, n = len(grid), len(grid[0])
        return sum(dfs(i, j) for i in range(m) for j in range(n) if
grid[i][j] == 0)
```

```
from collections import Counter as cnt

class Solution:
    def maxScoreWords(self, w: List[str], l: List[str], s: List[int]) -> int:
        def dfs(use, i):
            return (
                use
                and i < len(w)
                and max(
                    dfs(use, i + 1),
                    not cnt(w[i]) - use
                    and sum(s[ord(c) - ord("a")] for c in w[i])
                    + dfs(use - cnt(w[i]), i + 1),
                )
            )
        return int(dfs(cnt(l), 0))
```

```
class Solution:  
    def encode(self, num: int) -> str:  
        return bin(num + 1)[3:]
```

```
class Solution:
    def findSmallestRegion(
        self, regions: List[List[str]], region1: str, region2: str
    ) -> str:
        nex = {r: region[0] for region in regions for r in region[1:]}
        r1, r2 = region1, region2
        while r1 != r2:
            r1 = nex[r1] if r1 in nex else region2
            r2 = nex[r2] if r2 in nex else region1
        return r1
```

```
class Solution:
    def generateSentences(self, synonyms: List[List[str]], text: str) ->
List[str]:
    def root(s):
        return s if parent[s] == s else root(parent[s])

        parent = {s: s for s in [c for sy in synonyms for c in sy] +
text.split()}

        for a, b in synonyms:
            parent[root(a)] = root(b)
        bfs = [""]

        for t in text.split():
            r = root(t)
            bfs = [s + " " + w for s in bfs for w in parent if root(w) ==
r]

        return sorted(s[1:] for s in bfs)
```

```
class Solution:
    def numberOfWays(self, num_people):
        self.memo = {0: 1}

        def dp(n):
            if n not in self.memo:
                self.memo[n] = sum(
                    [dp(i - 2) * dp(n - i) for i in range(2, n + 1, 2)]
                ) % (10 ** 9 + 7)
            return self.memo[n]

        return dp(num_people)
```

```
class Solution:
    def shiftGrid(self, grid: List[List[int]], k: int) ->
List[List[int]]:
    chain = [r for row in grid for r in row]
    k %= len(chain)
    chain = chain[-k:] + chain[:-k]
    return [chain[i : i + len(grid[0])] for i in range(0, len(chain),
len(grid[0]))]
```

```
class FindElements:
    def dfs(self, node: TreeNode, real: int = 0):
        if node:
            node.val = real
            self.nums.add(node.val)
            self.dfs(node.left, real * 2 + 1)
            self.dfs(node.right, real * 2 + 2)

    def __init__(self, root: TreeNode):
        self.root = root
        self.nums = set()
        self.dfs(root)

    def find(self, target: int) -> bool:
        return target in self.nums
```

```
class Solution:
    def maxSumDivThree(
        self, nums: List[int], dp: list = [0, -float("inf"), -
float("inf")])
    ) -> int:
        for num in nums:
            dp = [max(dp[i], dp[(i - num) % 3] + num) for i in range(3)]
        return dp[0]
```

```

class Solution:
    def minPushBox(self, grid: List[List[str]]) -> int:
        m, n = len(grid), len(grid[0])
        for r in range(m):
            for c in range(n):
                if grid[r][c] == "T":
                    tX, tY = r, c
                if grid[r][c] == "B":
                    bX, bY = r, c
                if grid[r][c] == "S":
                    pX, pY = r, c

        def heuristic(bX, bY):
            return abs(tX - bX) + abs(tY - bY)

        heap = [[heuristic(bX, bY), 0, pX, pY, bX, bY]]
        visited = set()
        while heap:
            _, moves, pX, pY, bX, bY = heapq.heappop(heap)
            if bX == tX and bY == tY:
                return moves
            if (pX, pY, bX, bY) not in visited:
                visited.add((pX, pY, bX, bY))
                for dx, dy in (0, 1), (1, 0), (-1, 0), (0, -1):
                    pX += dx
                    pY += dy
                    if 0 <= pX < m and 0 <= pY < n and grid[pX][pY] != "#":
                        if pX == bX and pY == bY:
                            bX += dx
                            bY += dy
                            if 0 <= bX < m and 0 <= bY < n and grid[bX][bY] != "#":
                                heapq.heappush(
                                    heap,
                                    [
                                        heuristic(bX, bY) + moves + 1,
                                        moves + 1,
                                        pX,
                                        pY,
                                        bX,
                                        bY,
                                    ],
                                )
                            bX -= dx
                            bY -= dy
                        else:
                            heapq.heappush(
                                heap, [heuristic(bX, bY) + moves, moves,
                                       pX, pY, bX, bY]
                            )
                    pX -= dx
                    pY -= dy
        return -1

```

'''

On a plane there are n points with integer coordinates  $\text{points}[i] = [\text{xi}, \text{yi}]$ . Your task is to find the minimum time in seconds to visit all points.

You can move according to the next rules:

In one second always you can either move vertically, horizontally by one unit or diagonally (it means to move one unit vertically and one unit horizontally in one second).

You have to visit the points in the same order as they appear in the array.

Input:  $\text{points} = [[1,1], [3,4], [-1,0]]$

Output: 7

Explanation: One optimal path is  $[1,1] \rightarrow [2,2] \rightarrow [3,3] \rightarrow [3,4] \rightarrow [2,3] \rightarrow [1,2] \rightarrow [0,1] \rightarrow [-1,0]$

Time from  $[1,1]$  to  $[3,4]$  = 3 seconds

Time from  $[3,4]$  to  $[-1,0]$  = 4 seconds

Total time = 7 seconds

Example 2:

Input:  $\text{points} = [[3,2], [-2,2]]$

Output: 5

Constraints:

```
points.length == n
1 <= n <= 100
points[i].length == 2
-1000 <= points[i][0], points[i][1] <= 1000
'''

class Solution(object):
    def minTimeToVisitAllPoints(self, points):
        """
        :type points: List[List[int]]
        :rtype: int
        """
        if not points:
            return 0
        result = 0
        for index in range(1, len(points)):
            result += max(abs(points[index][0]-points[index-1][0]),
abs(points[index][1]-points[index-1][1]))
        return result
```

```
'''
```

You are given a map of a server center, represented as a  $m * n$  integer matrix grid, where 1 means that on that cell there is a server and 0 means that it is no server. Two servers are said to communicate if they are on the same row or on the same column.

Return the number of servers that communicate with any other server.

Example 1:

Input: grid = [[1,0],[0,1]]

Output: 0

Explanation: No servers can communicate with others.

Example 2:

Input: grid = [[1,0],[1,1]]

Output: 3

Explanation: All three servers can communicate with at least one other server.

Example 3:

Input: grid = [[1,1,0,0],[0,0,1,0],[0,0,1,0],[0,0,0,1]]

Output: 4

Explanation: The two servers in the first row can communicate with each other. The two servers in the third column can communicate with each other. The server at right bottom corner can't communicate with any other server.

Constraints:

```
m == grid.length
n == grid[i].length
1 <= m <= 250
1 <= n <= 250
grid[i][j] == 0 or 1
'''
```

```
class Solution(object):
    def countServers(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        if not grid:
            return 0

        row_count = [0] * len(grid)
        col_count = [0] * len(grid[0])
        for index_r in range(len(grid)):
            for index_c in range(len(grid[0])):
                if grid[index_r][index_c] == 1:
                    row_count[index_r] += 1
                    col_count[index_c] += 1

        result = 0
```

```
for index_r in range(len(grid)):
    for index_c in range(len(grid[0])):
        if grid[index_r][index_c] == 1 and (row_count[index_r] >
1 or col_count[index_c] > 1):
            result += 1
return result
```

```
'''
```

Given an array of strings products and a string searchWord. We want to design a system that suggests at most three product names from products after each character of searchWord is typed. Suggested products should have common prefix with the searchWord. If there are more than three products with a common prefix return the three lexicographically minimums products.

Return list of lists of the suggested products after each character of searchWord is typed.

Example 1:

Input: products = ["mobile", "mouse", "moneypot", "monitor", "mousepad"],  
searchWord = "mouse"

Output: [  
["mobile", "moneypot", "monitor"],  
["mobile", "moneypot", "monitor"],  
["mouse", "mousepad"],  
["mouse", "mousepad"],  
["mouse", "mousepad"]  
]

Explanation: products sorted lexicographically =  
["mobile", "moneypot", "monitor", "mouse", "mousepad"]

After typing m and mo all products match and we show user  
["mobile", "moneypot", "monitor"]

After typing mou, mous and mouse the system suggests ["mouse", "mousepad"]

Example 2:

Input: products = ["havana"], searchWord = "havana"

Output:

[["havana"], ["havana"], ["havana"], ["havana"], ["havana"], ["havana"]]

Example 3:

Input: products = ["bags", "baggage", "banner", "box", "cloths"], searchWord = "bags"

Output:

[["baggage", "bags", "banner"], ["baggage", "bags", "banner"], ["baggage", "bags"], ["bags"]]

Example 4:

Input: products = ["havana"], searchWord = "tatiana"

Output: [[], [], [], [], [], []]

Constraints:

1 <= products.length <= 1000

There are no repeated elements in products.

1 <= products[i].length <= 2 \* 10^4

All characters of products[i] are lower-case English letters.

1 <= searchWord.length <= 1000

All characters of searchWord are lower-case English letters.

```
'''
```

```
class TrieNode(object):  
    def __init__(self):
```

```

        self.words = []
        self.children = {}

class Trie(object):
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.words.append(word)
        node.words.sort()
        if len(node.words) > 3:
            node.words = node.words[:3]

    def search(self, word):
        result, node = [], self.root
        for char in word:
            if char not in node.children:
                break
            node = node.children[char]
            result.append(node.words[:])
        for _ in range(len(word) - len(result)):
            result.append([])
        return result

class Solution(object):
    def suggestedProducts(self, products, searchWord):
        """
        :type products: List[str]
        :type searchWord: str
        :rtype: List[List[str]]
        """
        trie = Trie()
        for product in products:
            trie.insert(product)
        return trie.search(searchWord)

```

1

Tic-tac-toe is played by two players A and B on a  $3 \times 3$  grid.

Here are the rules of Tic-Tac-Toe:

Players take turns placing characters into empty squares (" "). The first player A always places "X" characters, while the second player B always places "O" characters. "X" and "O" characters are always placed into empty squares, never on filled ones. The game ends when there are 3 of the same (non-empty) character filling any row, column, or diagonal. The game also ends if all squares are non-empty. No more moves can be played if the game is over. Given an array moves where each element is another array of size 2 corresponding to the row and column of the grid where they mark their respective character in the order in which A and B play.

Return the winner of the game if it exists (A or B), in case the game ends in a draw return "Draw", if there are still movements to play return "Pending".

You can assume that moves is valid (It follows the rules of Tic-Tac-Toe), the grid is initially empty and A will play first.

### Example 1:

```
Input: moves = [[0,0],[2,0],[1,1],[2,1],[2,2]]
Output: "A"
Explanation: "A" wins, he always plays first.
"X" "X" "X" "X" "X"
" " -> " " -> " X" -> " X" -> " X"
" " "O" "O" "OO" "OOX"
```

### Example 2:

```
Input: moves = [[0,0],[1,1],[0,1],[0,2],[1,0],[2,0]]
Output: "B"
Explanation: "B" wins.
"X"      "X"      "XX"      "XXO"      "XXO"      "XXO"
" " -> " O " -> " O " -> " O " -> " X O " -> " X O "
" "      " "      " "      " "      " "      " "      " "
```

### Example 3:

**Input:** moves = [[0,0],[1,1],[2,0],[1,0],[1,2],[2,1],[0,1],[0,2],[2,2]]  
**Output:** "Draw"

**Explanation:** The game ends in a draw since there are no moves to make

## Бапта "УУГО"

XXX

"YOK"

Example 4:

```
Input: moves = [[0,0],[1,1]]
```

Output: "Pending"

Explanation: The game has not finished yet.

"X "

" Q "

```
"" "
```

Constraints:

```
1 <= moves.length <= 9
moves[i].length == 2
0 <= moves[i][j] <= 2
There are no repeated elements on moves.
moves follow the rules of tic tac toe.
"""

class Solution(object):
    def tictactoe(self, moves):
        """
        :type moves: List[List[int]]
        :rtype: str
        """
        def check(grid):
            for x in range(3):
                row = set([grid[x][0], grid[x][1], grid[x][2]])
                if len(row) == 1 and grid[x][0] != 0:
                    return grid[x][0]

            for x in range(3):
                column = set([grid[0][x], grid[1][x], grid[2][x]])
                if len(column) == 1 and grid[0][x] != 0:
                    return grid[0][x]

            diag1 = set([grid[0][0], grid[1][1], grid[2][2]])
            diag2 = set([grid[0][2], grid[1][1], grid[2][0]])
            if len(diag1) == 1 or len(diag2) == 1 and grid[1][1] != 0:
                return grid[1][1]

            return 0

        if not moves:
            return ""
        grid = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
        user = 1
        for move in moves:
            grid[move[0]][move[1]] = user
            if user == 1:
                user = 2
            else:
                user = 1

        result = check(grid)
        if result == 1:
            return "A"
        elif result == 2:
            return "B"
        else:
            if len(moves) == 9:
                return "Draw"
            else:
                return "Pending"
```

```
'''
```

```
Given two integers tomatoSlices and cheeseSlices. The ingredients of  
different burgers are as follows:
```

```
Jumbo Burger: 4 tomato slices and 1 cheese slice.
```

```
Small Burger: 2 Tomato slices and 1 cheese slice.
```

```
Return [total_jumbo, total_small] so that the number of remaining  
tomatoSlices equal to 0 and the number of remaining cheeseSlices equal to  
0. If it is not possible to make the remaining tomatoSlices and  
cheeseSlices equal to 0 return [].
```

```
Example 1:
```

```
Input: tomatoSlices = 16, cheeseSlices = 7
```

```
Output: [1,6]
```

```
Explanation: To make one jumbo burger and 6 small burgers we need 4*1 +  
2*6 = 16 tomato and 1 + 6 = 7 cheese. There will be no remaining  
ingredients.
```

```
Example 2:
```

```
Input: tomatoSlices = 17, cheeseSlices = 4
```

```
Output: []
```

```
Explanation: There will be no way to use all ingredients to make small and  
jumbo burgers.
```

```
Example 3:
```

```
Input: tomatoSlices = 4, cheeseSlices = 17
```

```
Output: []
```

```
Explanation: Making 1 jumbo burger there will be 16 cheese remaining and  
making 2 small burgers there will be 15 cheese remaining.
```

```
Example 4:
```

```
Input: tomatoSlices = 0, cheeseSlices = 0
```

```
Output: [0,0]
```

```
Example 5:
```

```
Input: tomatoSlices = 2, cheeseSlices = 1
```

```
Output: [0,1]
```

```
Constraints:
```

```
0 <= tomatoSlices <= 10^7
0 <= cheeseSlices <= 10^7
'''
class Solution(object):
    def numOfBurgers(self, tomatoSlices, cheeseSlices):
        """
        :type tomatoSlices: int
        :type cheeseSlices: int
        :rtype: List[int]
        """
        jumbo = tomatoSlices - 2*cheeseSlices
        if jumbo >= 0 and jumbo%2 == 0:
            x = jumbo/2
            y = cheeseSlices-(jumbo/2)
```

```
if x >= 0 and y >= 0:  
    return [x, y]  
else:  
    return []  
return []
```

```
'''
```

```
Given a m * n matrix of ones and zeros, return how many square  
submatrices have all ones.
```

Example 1:

```
Input: matrix =  
[  
    [0,1,1,1],  
    [1,1,1,1],  
    [0,1,1,1]  
]  
Output: 15  
Explanation:  
There are 10 squares of side 1.  
There are 4 squares of side 2.  
There is 1 square of side 3.  
Total number of squares = 10 + 4 + 1 = 15.
```

Example 2:

```
Input: matrix =  
[  
    [1,0,1],  
    [1,1,0],  
    [1,1,0]  
]  
Output: 7  
Explanation:  
There are 6 squares of side 1.  
There is 1 square of side 2.  
Total number of squares = 6 + 1 = 7.
```

Constraints:

```
1 <= arr.length <= 300  
1 <= arr[0].length <= 300  
0 <= arr[i][j] <= 1  
'''  
class Solution(object):  
    def countSquares(self, matrix):  
        """  
        :type matrix: List[List[int]]  
        :rtype: int  
        """  
  
        p_arr = [[0 for i in range(len(matrix[0]))] for j in  
range(len(matrix))]  
        result = 0  
  
        for index_i in range(1, len(matrix)):  
            for index_j in range(1, len(matrix[0])):  
                if matrix[index_i][index_j] == 1:  
                    matrix[index_i][index_j] = min(matrix[index_i-  
1][index_j-1], min(matrix[index_i-1][index_j], matrix[index_i][index_j-  
1]))+1
```

```
# print p_arr
return sum([ sum(x) for x in matrix])
```

```
'''
```

```
Given an integer number n, return the difference between the product of  
its digits and the sum of its digits.
```

```
Example 1:
```

```
Input: n = 234
```

```
Output: 15
```

```
Explanation:
```

```
Product of digits = 2 * 3 * 4 = 24
```

```
Sum of digits = 2 + 3 + 4 = 9
```

```
Result = 24 - 9 = 15
```

```
Example 2:
```

```
Input: n = 4421
```

```
Output: 21
```

```
Explanation:
```

```
Product of digits = 4 * 4 * 2 * 1 = 32
```

```
Sum of digits = 4 + 4 + 2 + 1 = 11
```

```
Result = 32 - 11 = 21
```

```
'''
```

```
class Solution(object):  
    def subtractProductAndSum(self, n):  
        """  
        :type n: int  
        :rtype: int  
        """  
  
        from functools import reduce  
        from operator import mul  
        digits = [int(x) for x in str(n)]  
        return reduce(mul, digits) - sum(digits)
```

'''

There are  $n$  people whose IDs go from 0 to  $n - 1$  and each person belongs exactly to one group. Given the array `groupSizes` of length  $n$  telling the group size each person belongs to, return the groups there are and the people's IDs each group includes.

You can return any solution in any order and the same applies for IDs. Also, it is guaranteed that there exists at least one solution.

Example 1:

Input: `groupSizes = [3,3,3,3,3,1,3]`

Output: `[[5],[0,1,2],[3,4,6]]`

Explanation:

Other possible solutions are `[[2,1,6],[5],[0,4,3]]` and `[[5],[0,6,2],[4,3,1]]`.

Example 2:

Input: `groupSizes = [2,1,3,3,3,2]`

Output: `[[1],[0,5],[2,3,4]]`

'''

```
class Solution(object):
    def groupThePeople(self, groupSizes):
        """
        :type groupSizes: List[int]
        :rtype: List[List[int]]
        """
        count = collections.defaultdict(list)
        for i, size in enumerate(groupSizes):
            count[size].append(i)
        result = []
        for s, value in count.items():
            for index in range(0, len(value), s):
                result.append(value[index:index + s])
        return result
```

'''

Given an array of integers `nums` and an integer `threshold`, we will choose a positive integer divisor and divide all the array by it and sum the result of the division. Find the smallest divisor such that the result mentioned above is less than or equal to `threshold`.

Each result of division is rounded to the nearest integer greater than or equal to that element. (For example:  $7/3 = 3$  and  $10/2 = 5$ ).

It is guaranteed that there will be an answer.

Example 1:

Input: `nums = [1,2,5,9]`, `threshold = 6`  
Output: 5

Explanation: We can get a sum to 17 ( $1+2+5+9$ ) if the divisor is 1.  
If the divisor is 4 we can get a sum to 7 ( $1+1+2+3$ ) and if the divisor is 5 the sum will be 5 ( $1+1+1+2$ ).

Example 2:

Input: `nums = [2,3,5,7,11]`, `threshold = 11`  
Output: 3  
Example 3:

```
Input: nums = [19], threshold = 5
Output: 4
'''
class Solution(object):
    def smallestDivisor(self, nums, threshold):
        """
        :type nums: List[int]
        :type threshold: int
        :rtype: int
        """
        def getSum(divisor, xs):
            return sum([x // divisor + 1 if x % divisor else x // divisor
for x in xs])

        left, right = 1, 10 ** 6
        while left + 1 < right:
            mid = (left + right) // 2
            if getSum(mid, nums) > threshold:
                left = mid
            else:
                right = mid

        return left if getSum(left, nums) <= threshold else right
```

```
'''
```

Given head which is a reference node to a singly-linked list. The value of each node in the linked list is either 0 or 1. The linked list holds the binary representation of a number.

Return the decimal value of the number in the linked list.

Example 1:

Input: head = [1,0,1]

Output: 5

Explanation: (101) in base 2 = (5) in base 10

Example 2:

Input: head = [0]

Output: 0

Example 3:

Input: head = [1]

Output: 1

Example 4:

Input: head = [1,0,0,1,0,0,1,1,1,0,0,0,0,0,0]

Output: 18880

Example 5:

Input: head = [0,0]

Output: 0

```
'''
```

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def getDecimalValue(self, head):
        """
        :type head: ListNode
        :rtype: int
        """
        result = ''
        if not head:
            return 0
        while head:
            result+= str(head.val)
            head = head.next
        return int(result, 2)
```

```
'''
```

```
An integer has sequential digits if and only if each digit in the number  
is one more than the previous digit.
```

```
Return a sorted list of all the integers in the range [low, high]  
inclusive that have sequential digits.
```

Example 1:

```
Input: low = 100, high = 300
```

```
Output: [123,234]
```

Example 2:

```
Input: low = 1000, high = 13000
```

```
Output: [1234,2345,3456,4567,5678,6789,12345]
```

```
'''
```

```
class Solution(object):  
    def sequentialDigits(self, low, high):  
        """  
        :type low: int  
        :type high: int  
        :rtype: List[int]  
        """  
        result = []  
        start = int(str(low)[0])  
        for val in range(1, len(str(low))):  
            new_val = start%10 + 1  
            start = start*10 + new_val  
        if start > high:  
            return result  
  
        result.append(start)  
  
        while result[-1] <= high:  
            temp = str(result[-1])  
            next_elem = int(temp[-1]) + 1  
  
            if next_elem > 9:  
                next_greater = 0  
                for index in range(len(temp) + 1):  
                    next_greater = next_greater*10 + (index+1)  
            else:  
                next_greater = int(temp[1:]) * 10 + next_elem  
            if next_greater <= high:  
                result.append(next_greater)  
            else:  
                break  
            # print next_greater  
        final_result = []  
        for val in result:  
            if '0' not in str(val) and val >= low:  
                final_result.append(val)  
        return final_result
```

```
'''  
Given an array nums of integers, return how many of them contain an even  
number of digits.
```

Example 1:

Input: nums = [12,345,2,6,7896]

Output: 2

Explanation:

12 contains 2 digits (even number of digits).

345 contains 3 digits (odd number of digits).

2 contains 1 digit (odd number of digits).

6 contains 1 digit (odd number of digits).

7896 contains 4 digits (even number of digits).

Therefore only 12 and 7896 contain an even number of digits.

Example 2:

Input: nums = [555,901,482,1771]

Output: 1

Explanation:

Only 1771 contains an even number of digits.

'''

```
class Solution(object):
```

```
    def findNumbers(self, nums):
```

```
        """
```

```
:type nums: List[int]
```

```
:rtype: int
```

```
"""
```

```
    return len([num for num in nums if len(str(num))%2 == 0])
```

