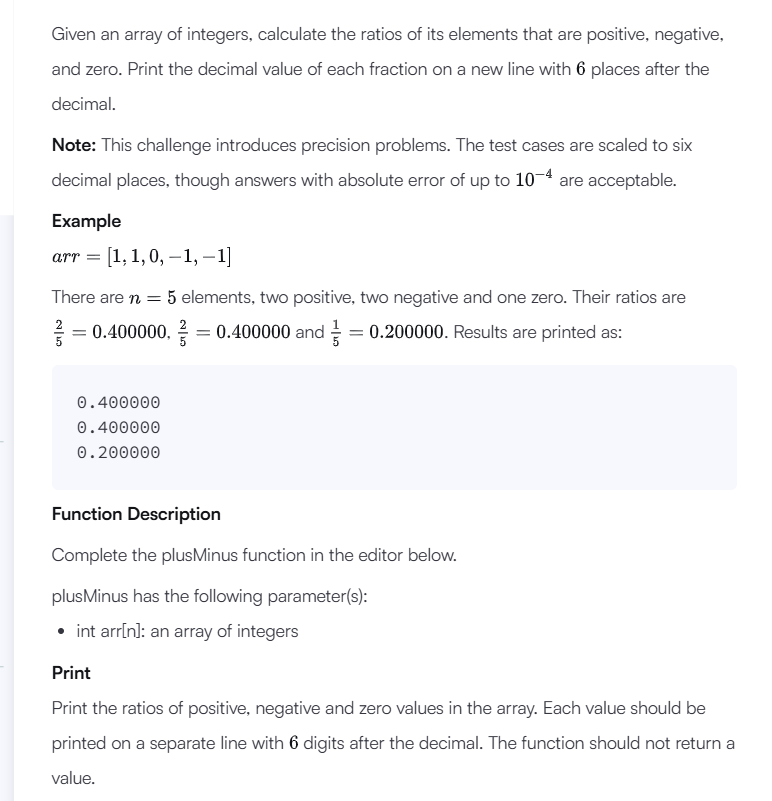
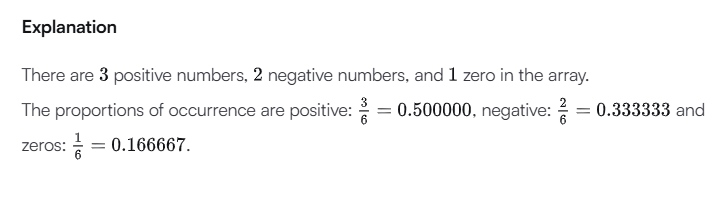
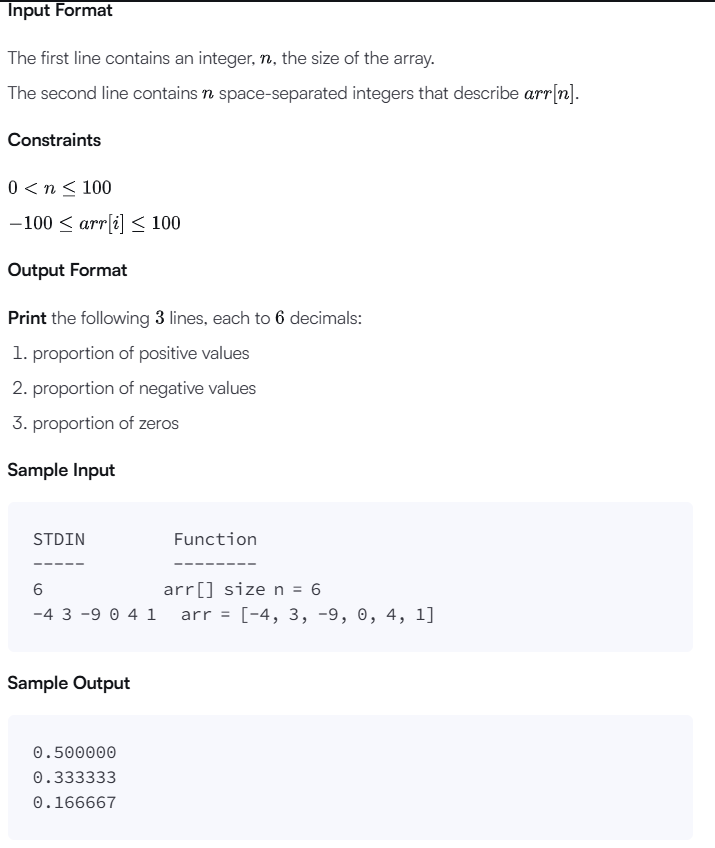
**Day 1:**





**Code:**

#!/bin/python3

import math

import os

import random

import re

import sys

#

# Complete the 'plusMinus' function below.

#

# The function accepts INTEGER\_ARRAY arr as parameter.

#

def plusMinus(arr):

    # Write your code here

     n = len(arr)

     pos\_count = 0

     neg\_count = 0

     zero\_count = 0

     for num in arr:

         if num > 0:

             pos\_count += 1

         elif num < 0:

             neg\_count += 1

         else:

             zero\_count += 1

     pos\_ratio = pos\_count / n

     neg\_ratio = neg\_count / n

     zero\_ratio = zero\_count / n

     print(f"{pos\_ratio:.6f}")

     print(f"{neg\_ratio:.6f}")

     print(f"{zero\_ratio:.6f}")

if \_\_name\_\_ == '\_\_main\_\_':

    n = int(input().strip())

    arr = list(map(int, input().rstrip().split()))

    plusMinus(arr)

Input (stdin)

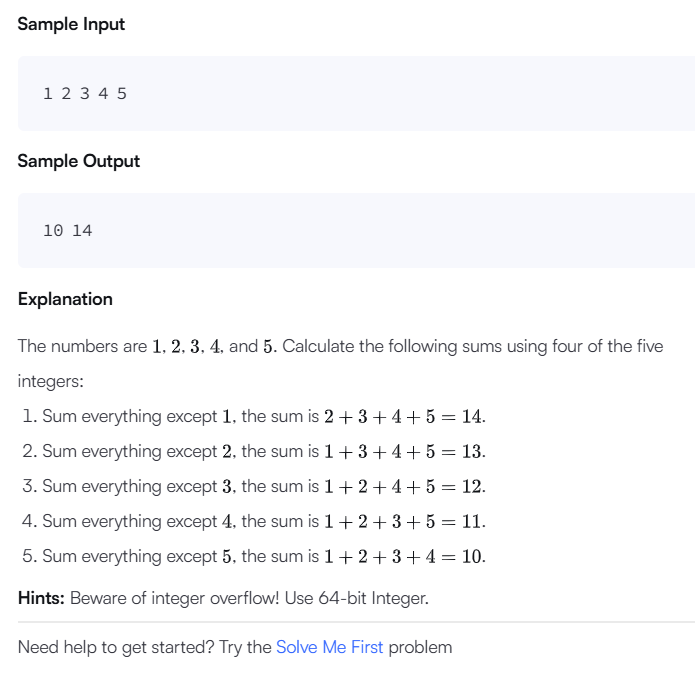
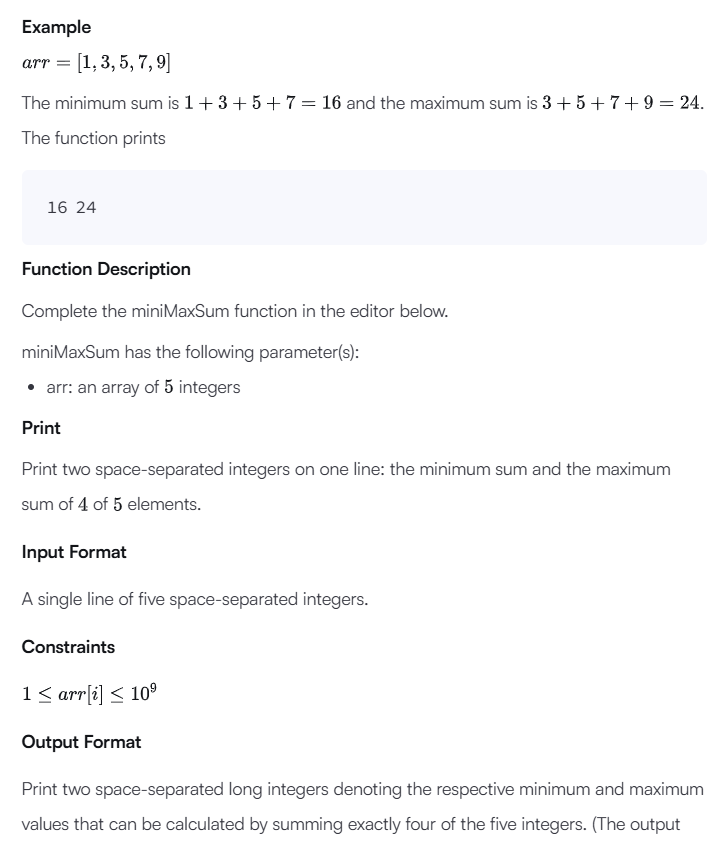
* **6**
* **-4 3 -9 0 4 1**

Your Output (stdout)

* **0.500000**
* **0.333333**
* **0.166667**

Expected Output

* **0.500000**
* **0.333333**
* **0.166667**

****

**Code:** #!/bin/python3

import math

import os

import random

import re

import sys

#

# Complete the 'miniMaxSum' function below.

#

# The function accepts INTEGER\_ARRAY arr as parameter.

#

def miniMaxSum(arr):

    # Calculate the total sum of the array

    total\_sum = sum(arr)

    # Minimum sum is obtained by excluding the maximum element

    min\_sum = total\_sum - max(arr)

    # Maximum sum is obtained by excluding the minimum element

    max\_sum = total\_sum - min(arr)

    # Print the results as space-separated integers

    print(f"{min\_sum} {max\_sum}")

if \_\_name\_\_ == '\_\_main\_\_':

    arr = list(map(int, input().rstrip().split()))

    miniMaxSum(arr)

Input (stdin)

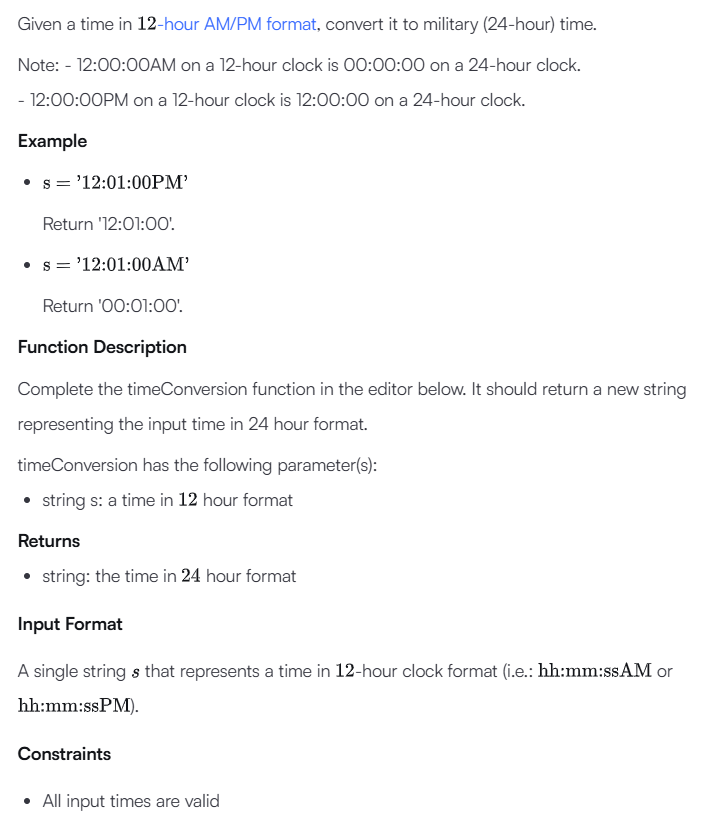
* **1 2 3 4 5**

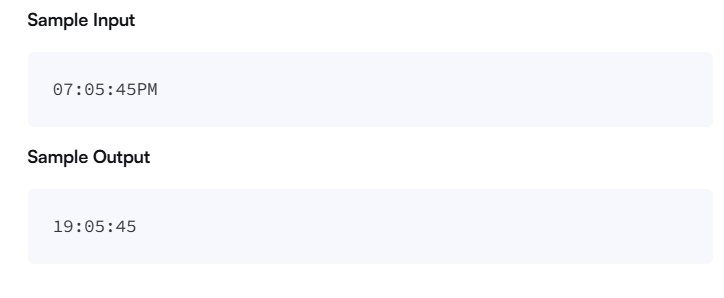
Your Output (stdout)

* **10 14**

Expected Output

* **10 14**

****

****

**Code:** #!/bin/python3

import math

import os

import random

import re

import sys

#

# Complete the 'timeConversion' function below.

#

# The function is expected to return a STRING.

# The function accepts STRING s as parameter.

#

def timeConversion(s):

    # Extract the AM/PM part

    period = s[-2:]

    hour = int(s[:2])

    minutes\_seconds = s[2:8]

    if period == "AM":

        if hour == 12:

            return f"00{minutes\_seconds}"

        else:

            return f"{hour:02}{minutes\_seconds}"

    else:  # PM case

        if hour == 12:

            return f"12{minutes\_seconds}"

        else:

            return f"{hour + 12:02}{minutes\_seconds}"

if \_\_name\_\_ == '\_\_main\_\_':

    fptr = open(os.environ['OUTPUT\_PATH'], 'w')

    s = input()

    result = timeConversion(s)

    fptr.write(result + '\n')

    fptr.close()

Input (stdin)

* **07:05:45PM**

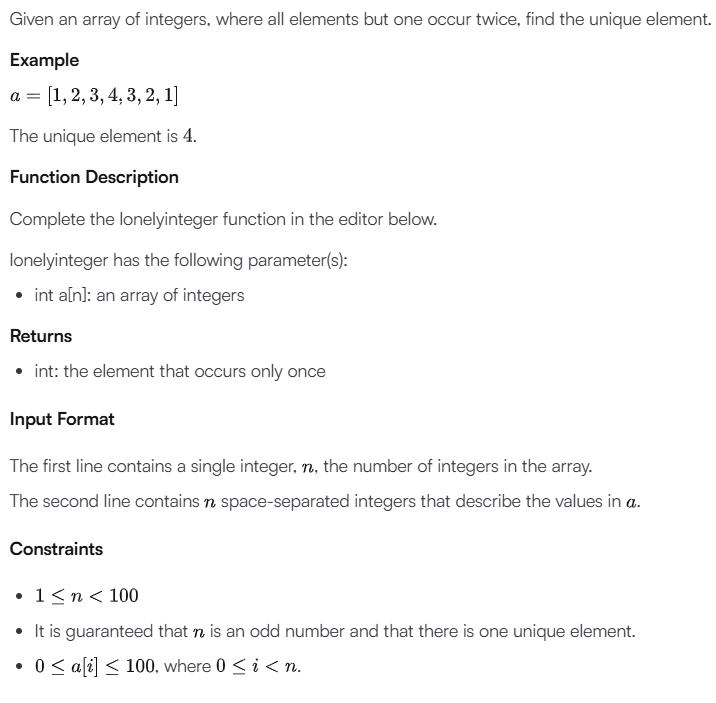
Your Output (stdout)

* **19:05:45**

Expected Output

* **19:05:45**

**Day2:**

****

Code: #!/bin/python3

import math

import os

import random

import re

import sys

#

# Complete the 'lonelyinteger' function below.

#

# The function is expected to return an INTEGER.

# The function accepts INTEGER\_ARRAY a as parameter.

#

def lonelyinteger(a):

    result = 0

    for num in a:

        result ^= num

    return result

if \_\_name\_\_ == '\_\_main\_\_':

    fptr = open(os.environ['OUTPUT\_PATH'], 'w')

    n = int(input().strip())

    a = list(map(int, input().rstrip().split()))

    result = lonelyinteger(a)

    fptr.write(str(result) + '\n')

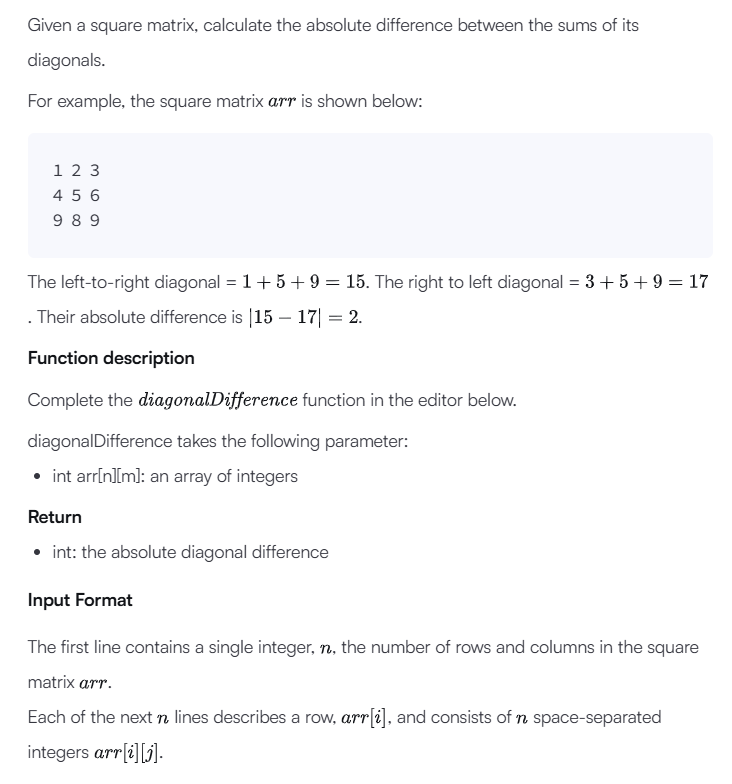
    fptr.close()

Input (stdin)

* **1**
* **1**

Your Output (stdout)

* **1**





#!/bin/python3

import math

import os

import random

import re

import sys

#

# Complete the 'diagonalDifference' function below.

#

# The function is expected to return an INTEGER.

# The function accepts 2D\_INTEGER\_ARRAY arr as parameter.

#

def diagonalDifference(arr):

    primary\_diagonal\_sum = 0

    secondary\_diagonal\_sum = 0

    n = len(arr)

    for i in range(n):

        primary\_diagonal\_sum += arr[i][i]  # Add the element from the primary diagonal

        secondary\_diagonal\_sum += arr[i][n-1-i]  # Add the element from the secondary diagonal

    return abs(primary\_diagonal\_sum - secondary\_diagonal\_sum)

if \_\_name\_\_ == '\_\_main\_\_':

    fptr = open(os.environ['OUTPUT\_PATH'], 'w')

    n = int(input().strip())

    arr = []

    for \_ in range(n):

        arr.append(list(map(int, input().rstrip().split())))

    result = diagonalDifference(arr)

    fptr.write(str(result) + '\n')

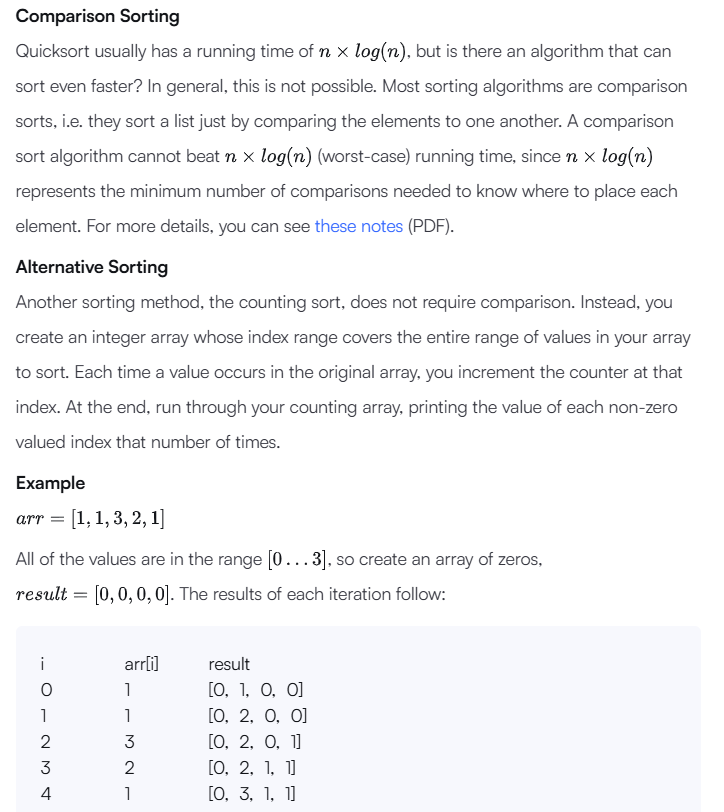
    fptr.close()

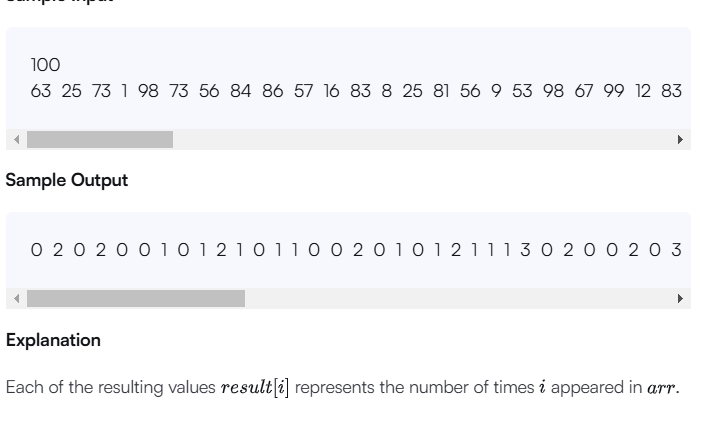
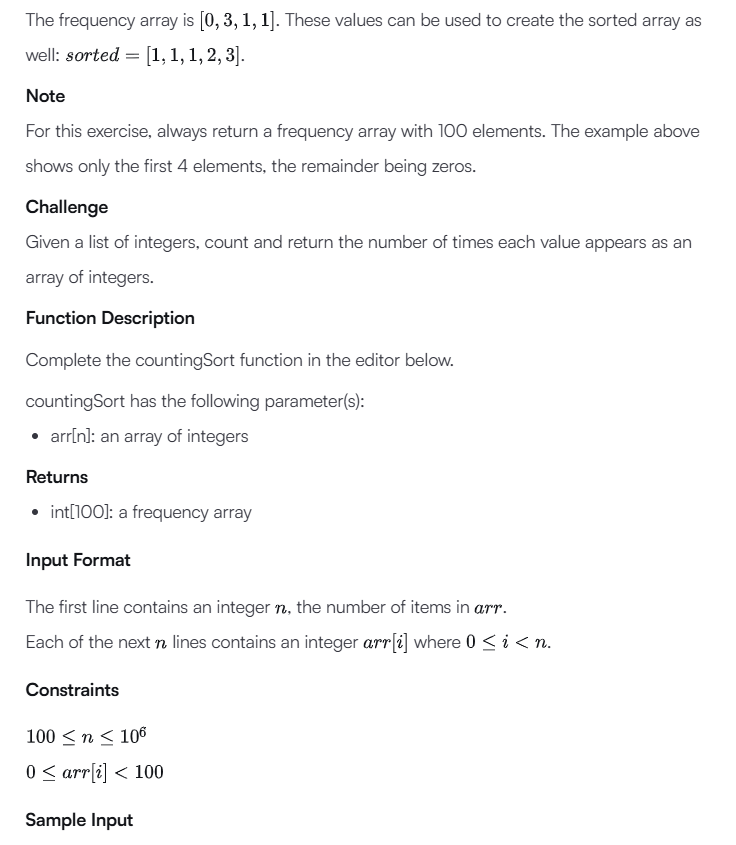
Input (stdin)

* **3**
* **11 2 4**
* **4 5 6**
* **10 8 -12**

Your Output (stdout)

* **15**





#!/bin/python3

import math

import os

import random

import re

import sys

#

# Complete the 'countingSort' function below.

#

# The function is expected to return an INTEGER\_ARRAY.

# The function accepts INTEGER\_ARRAY arr as parameter.

#

def countingSort(arr):

    # Create an array of size 100 initialized to 0

    freq = [0] \* 100

    # Count the occurrences of each number in the input array

    for num in arr:

        freq[num] += 1

    return freq

if \_\_name\_\_ == '\_\_main\_\_':

    fptr = open(os.environ['OUTPUT\_PATH'], 'w')

    n = int(input().strip())

    arr = list(map(int, input().rstrip().split()))

    result = countingSort(arr)

    fptr.write(' '.join(map(str, result)))

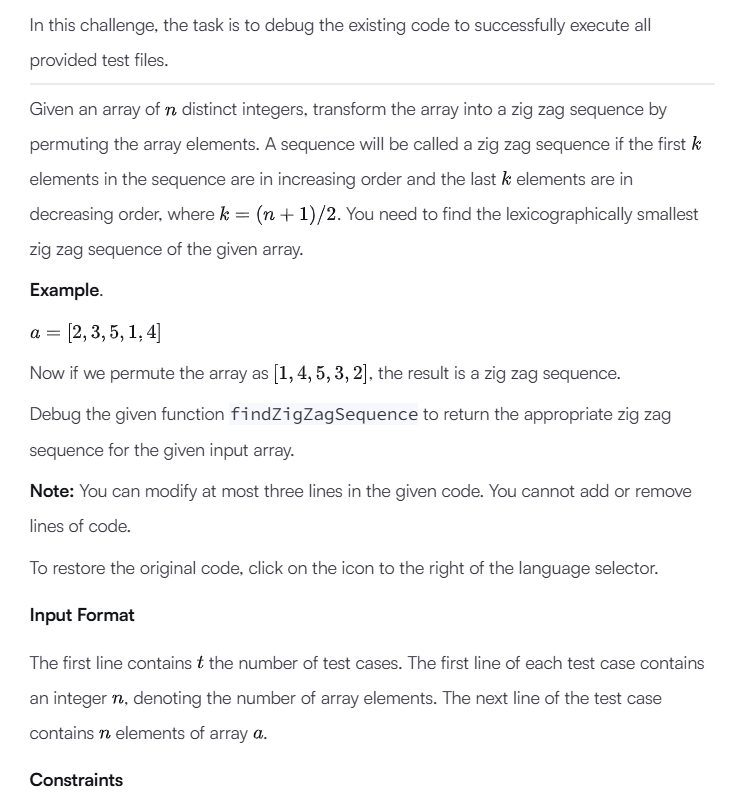
    fptr.write('\n')

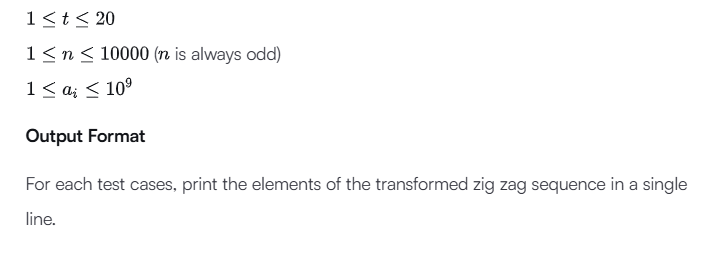
    fptr.close()

Your Output (stdout)

* **0 2 0 2 0 0 1 0 1 2 1 0 1 1 0 0 2 0 1 0 1 2 1 1 1 3 0 2 0 0 2 0 3 3 1 0 0 0 0 2 2 1 1 1 2 0 2 0 1 0 1 0 0 1 0 0 2 1 0 1 1 1 0 1 0 1 0 2 1 3 2 0 0 2 1 2 1 0 2 2 1 2 1 2 1 1 2 2 0 3 2 1 1 0 1 1 1 0 2 2**

**Day 3:**





def findZigZagSequence(a, n):

    a.sort()

    mid = int((n - 1)/2)

    a[mid], a[n-1] = a[n-1], a[mid]

    st = mid + 1

    ed = n - 2

    while(st <= ed):

        a[st], a[ed] = a[ed], a[st]

        st = st + 1

        ed = ed - 1

    for i in range (n):

        if i == n-1:

            print(a[i])

        else:

            print(a[i], end = ' ')

    return

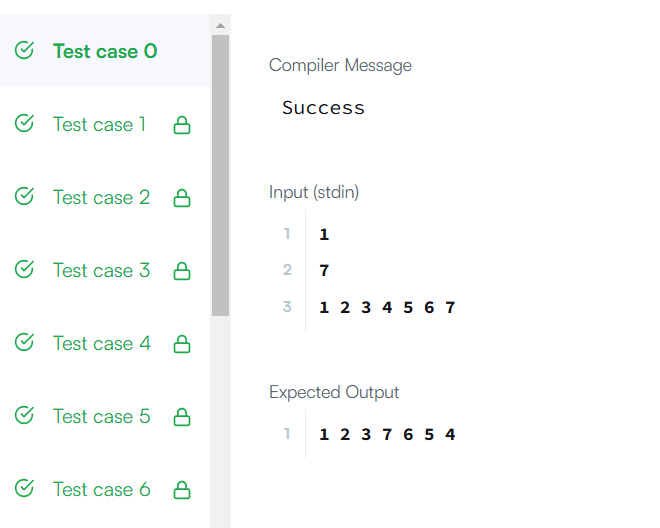
test\_cases = int(input())

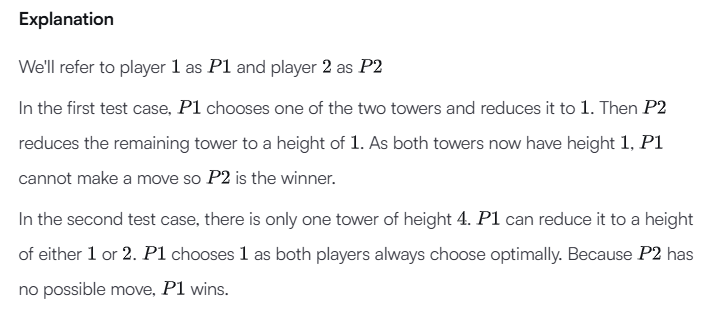
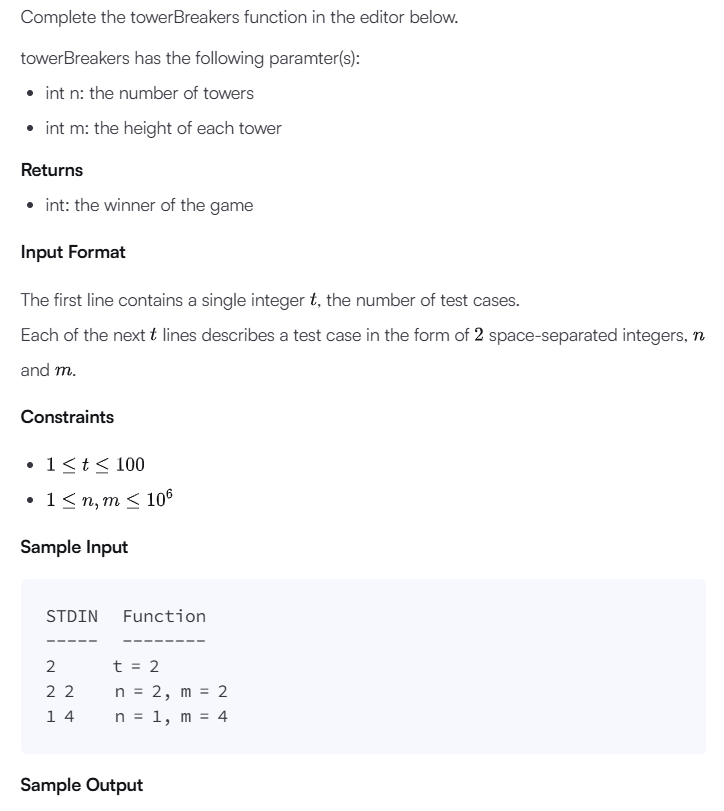
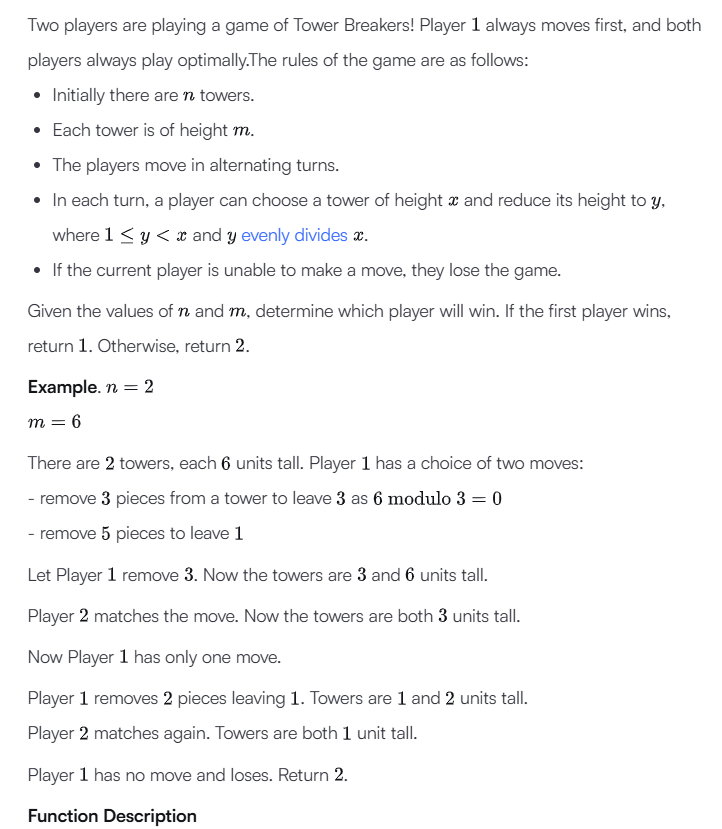
for cs in range (test\_cases):

    n = int(input())

    a = list(map(int, input().split()))

    findZigZagSequence(a, n)





#!/bin/python3

import math

import os

import random

import re

import sys

#

# Complete the 'towerBreakers' function below.

#

# The function is expected to return an INTEGER.

# The function accepts following parameters:

#  1. INTEGER n

#  2. INTEGER m

#

def towerBreakers(n, m):

    # If the height of towers is 1, Player 2 wins automatically

    if m == 1:

        return 2

    # If the number of towers is odd, Player 1 wins

    if n % 2 == 1:

        return 1

    # If the number of towers is even, Player 2 wins

    return 2

if \_\_name\_\_ == '\_\_main\_\_':

    fptr = open(os.environ['OUTPUT\_PATH'], 'w')

    t = int(input().strip())

    for t\_itr in range(t):

        first\_multiple\_input = input().rstrip().split()

        n = int(first\_multiple\_input[0])

        m = int(first\_multiple\_input[1])

        result = towerBreakers(n, m)

        fptr.write(str(result) + '\n')

    fptr.close()

* output : **2**
* **1**



#!/bin/python3

import math

import os

import random

import re

import sys

#

# Complete the 'caesarCipher' function below.

#

# The function is expected to return a STRING.

# The function accepts following parameters:

#  1. STRING s

#  2. INTEGER k

#

def caesarCipher(s, k):

    result = []

    k = k % 26

    for char in s:

        if char.isalpha():

            if char.isupper():

                new\_char = chr(((ord(char) - ord('A') + k) % 26) + ord('A'))

            else:

                new\_char = chr(((ord(char) - ord('a') + k) % 26) + ord('a'))

            result.append(new\_char)

        else:

            result.append(char)

    return ''.join(result)

if \_\_name\_\_ == '\_\_main\_\_':

    fptr = open(os.environ['OUTPUT\_PATH'], 'w')

    n = int(input().strip())

    s = input()

    k = int(input().strip())

    result = caesarCipher(s, k)

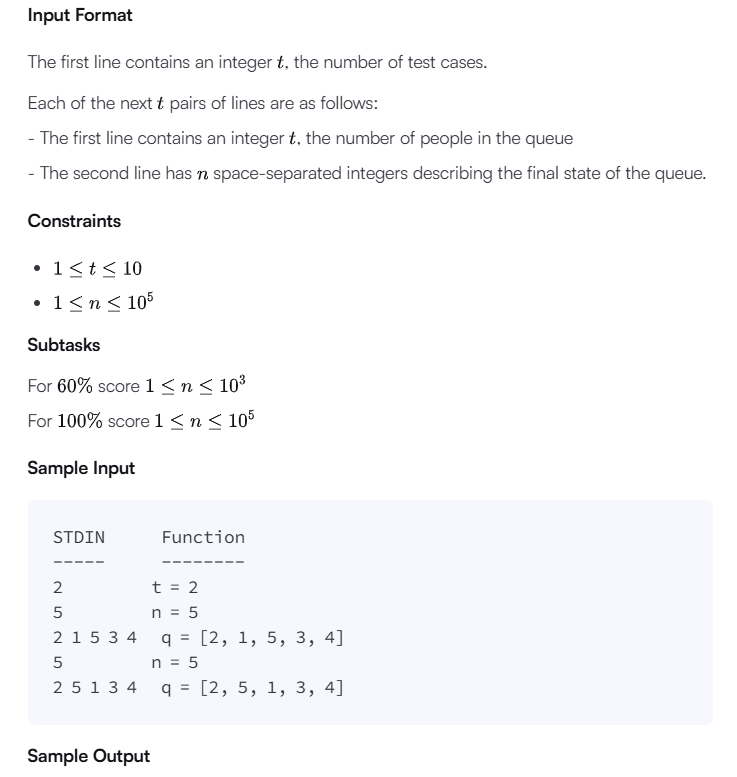
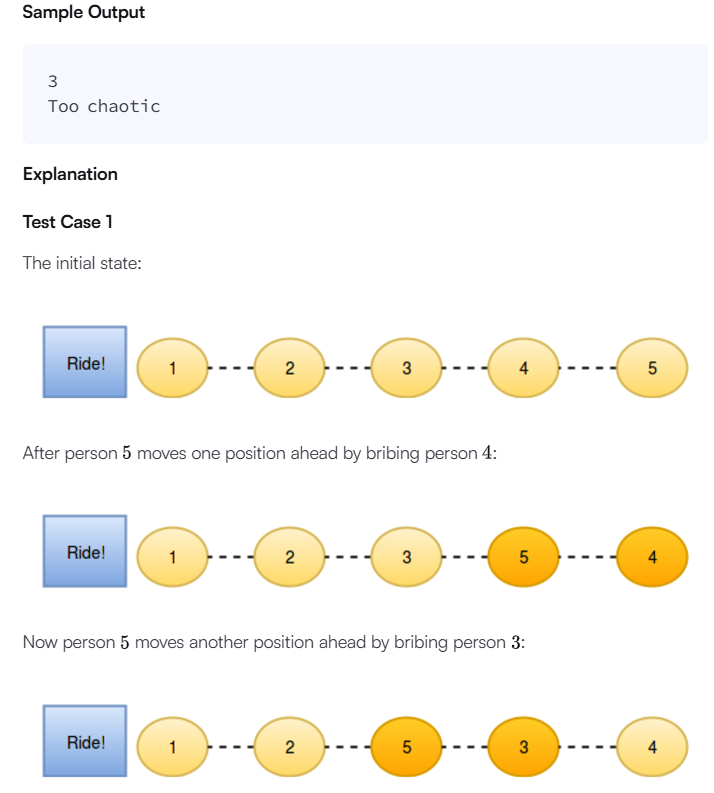
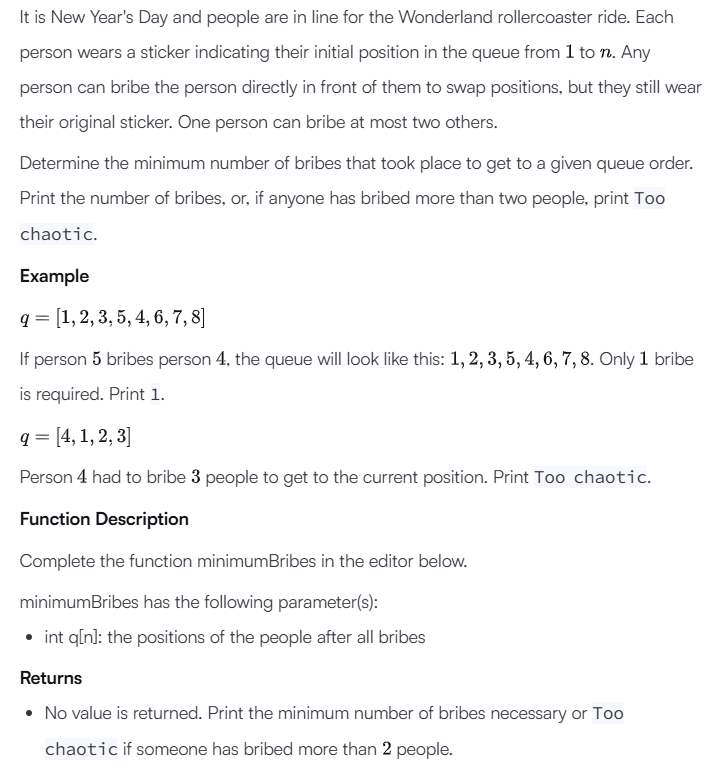
    fptr.write(result + '\n')

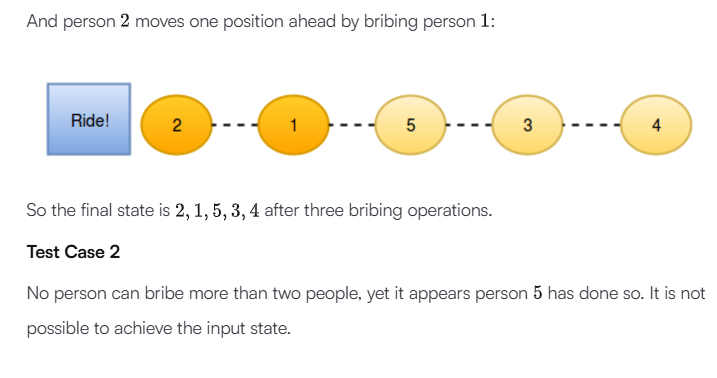
    fptr.close()

Output (stdout)

* **okffng-Qwvb**

**Day 4:**





#!/bin/python3

import math

import os

import random

import re

import sys

#

# Complete the 'minimumBribes' function below.

#

# The function accepts INTEGER\_ARRAY q as parameter.

#

def minimumBribes(q):

    bribes = 0

    for i in range(len(q) - 1, -1, -1):

        if q[i] - (i + 1) > 2:

            print("Too chaotic")

            return

        for j in range(max(0, q[i] - 2), i):

            if q[j] > q[i]:

                bribes += 1

    print(bribes)

if \_\_name\_\_ == '\_\_main\_\_':

    t = int(input().strip())

    for t\_itr in range(t):

        n = int(input().strip())

        q = list(map(int, input().rstrip().split()))

        minimumBribes(q)

Input (stdin)

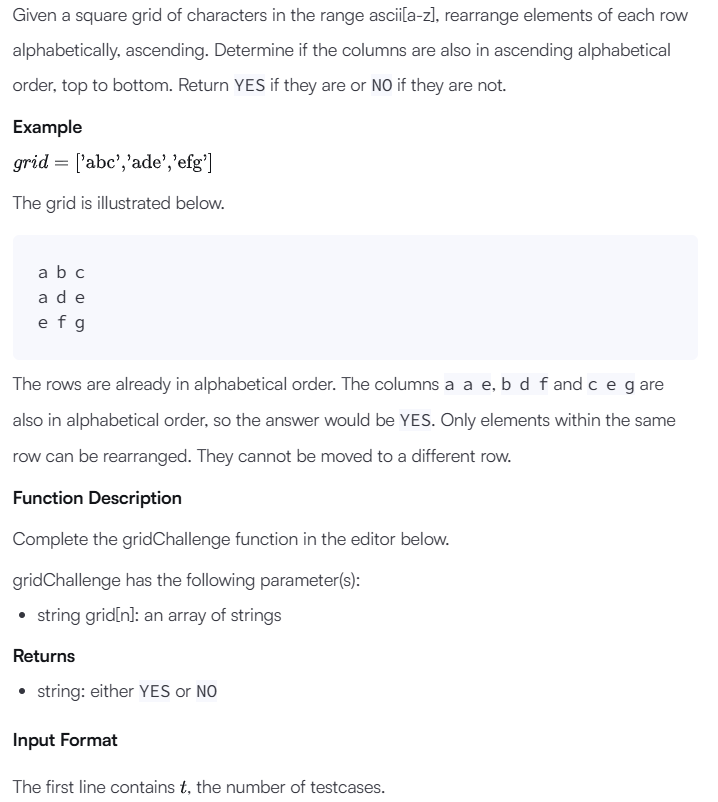
* **2**
* **5**
* **2 1 5 3 4**
* **5**
* **2 5 1 3 4**

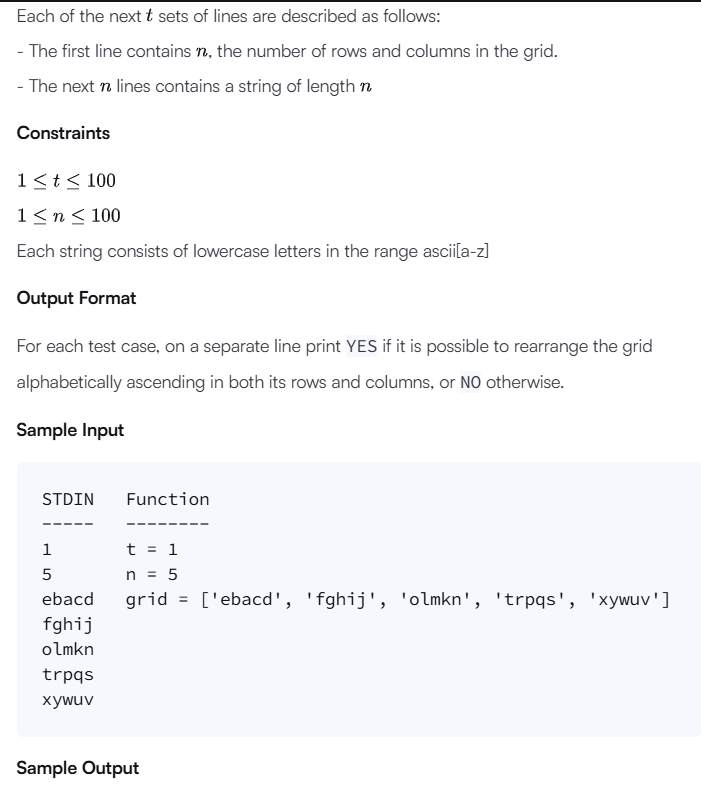
Your Output (stdout)

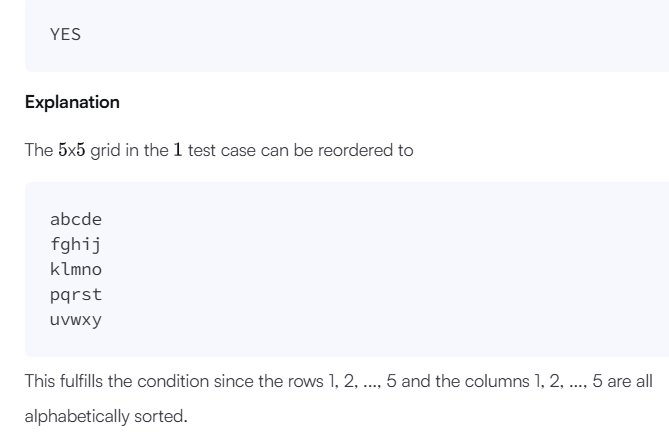
* **3**
* **Too chaotic**

Expected Output

* **3**
* **Too chaotic**







#!/bin/python3

import math

import os

import random

import re

import sys

#

# Complete the 'gridChallenge' function below.

#

# The function is expected to return a STRING.

# The function accepts STRING\_ARRAY grid as parameter.

#

def gridChallenge(grid):

    grid = [sorted(list(item)) for item in grid]

    gridt = [[grid[i][j] for i in range(len(grid))] for j in range(len(grid[0]))]

    result = all(gridt[i] == sorted(gridt[i]) for i in range(len(gridt)))

    return 'YES' if result else 'NO'

if \_\_name\_\_ == '\_\_main\_\_':

    fptr = open(os.environ['OUTPUT\_PATH'], 'w')

    t = int(input().strip())

    for t\_itr in range(t):

        n = int(input().strip())

        grid = []

        for \_ in range(n):

            grid\_item = input()

            grid.append(grid\_item)

        result = gridChallenge(grid)

        fptr.write(result + '\n')

    fptr.close()

Input (stdin)

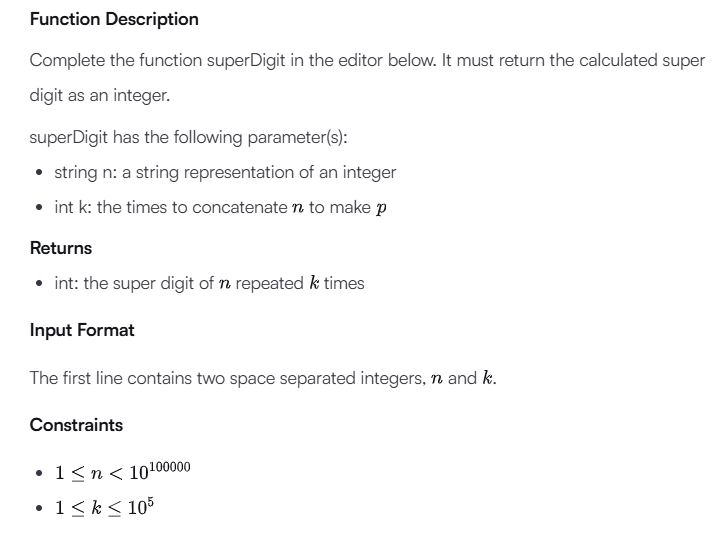
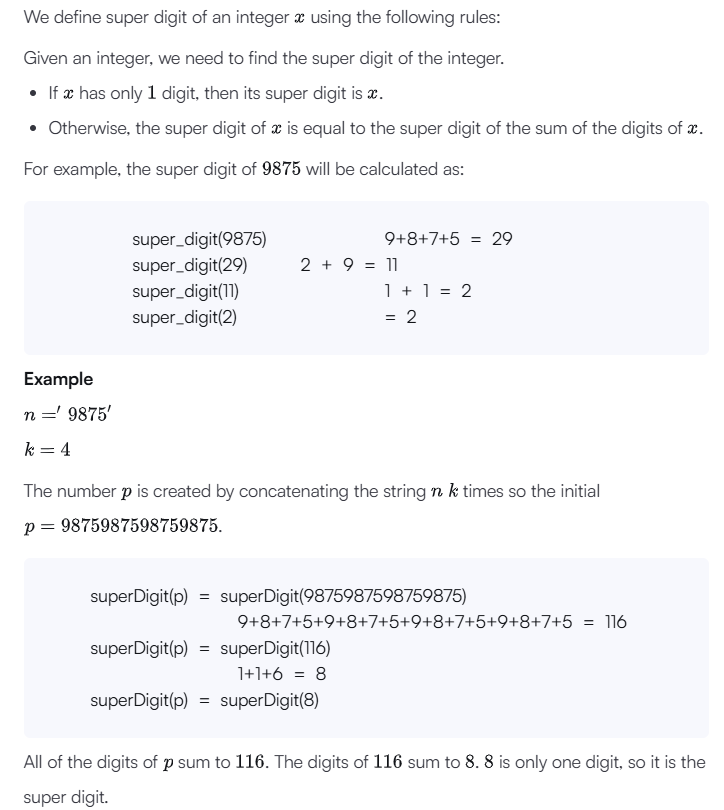
* **1**
* **5**
* **fghij**
* **olkmn**
* **trpqs**
* **xywuv**

Your Output (stdout)

* **YES**

Expected Output

* **YES**

****

#!/bin/python3

import math

import os

import random

import re

import sys

#

# Complete the 'superDigit' function below.

#

# The function is expected to return an INTEGER.

# The function accepts following parameters:

#  1. STRING n

#  2. INTEGER k

#

def superDigit(n, k):

    n = sum([int(n[i]) for i in range(len(n))]) \* k

    return superDigit(str(n), 1) if n > 9 else n

if \_\_name\_\_ == '\_\_main\_\_':

    fptr = open(os.environ['OUTPUT\_PATH'], 'w')

    first\_multiple\_input = input().rstrip().split()

    n = first\_multiple\_input[0]

    k = int(first\_multiple\_input[1])

    result = superDigit(n, k)

    fptr.write(str(result) + '\n')

    fptr.close()

Input (stdin)

* **148 3**

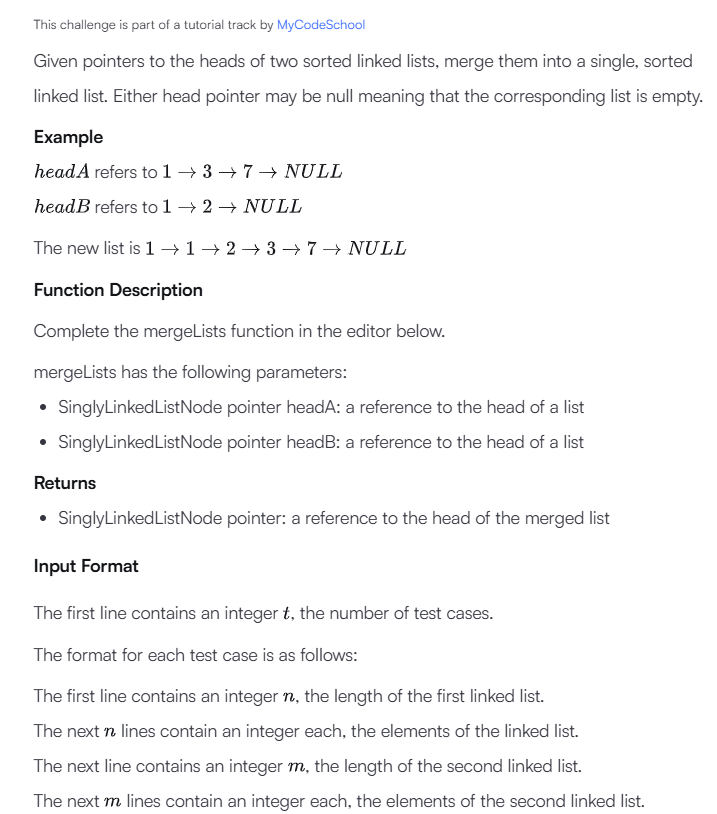
Your Output (stdout)

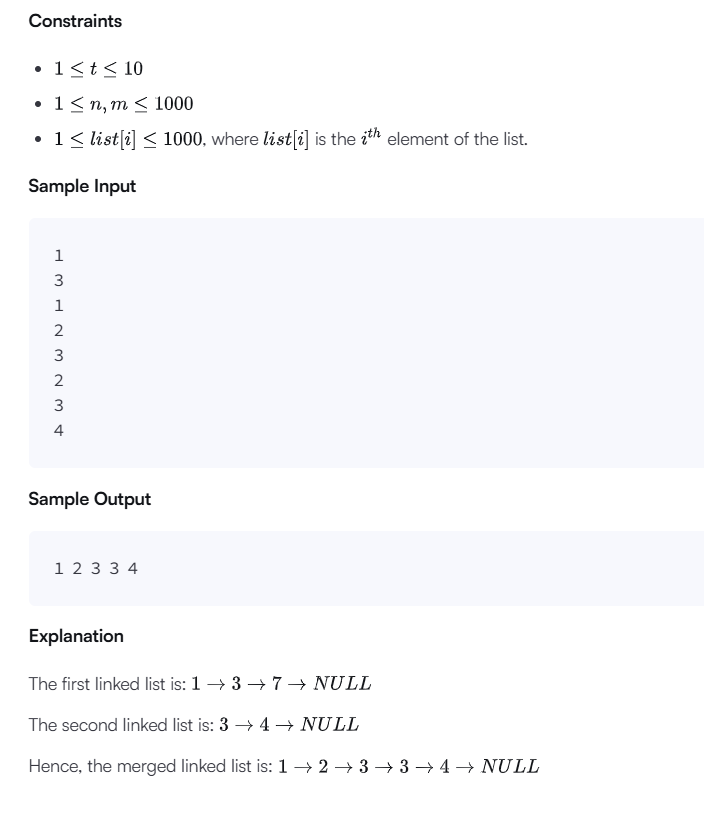
* **3**

Expected Output

* **3**

**Day 5:**

****

****

#!/bin/python3

import math

import os

import random

import re

import sys

class SinglyLinkedListNode:

    def \_\_init\_\_(self, node\_data):

        self.data = node\_data

        self.next = None

class SinglyLinkedList:

    def \_\_init\_\_(self):

        self.head = None

        self.tail = None

    def insert\_node(self, node\_data):

        node = SinglyLinkedListNode(node\_data)

        if not self.head:

            self.head = node

        else:

            self.tail.next = node

        self.tail = node

def print\_singly\_linked\_list(node, sep, fptr):

    while node:

        fptr.write(str(node.data))

        node = node.next

        if node:

            fptr.write(sep)

# Complete the mergeLists function below.

#

# For your reference:

#

# SinglyLinkedListNode:

#     int data

#     SinglyLinkedListNode next

#

#

def mergeLists(head1, head2):

    if head1 is None:

        return head2

    if head2 is None:

        return head1

    if head1.data < head2.data:

        merged\_head = head1

        head1 = head1.next

    else:

        merged\_head = head2

        head2 = head2.next

    current = merged\_head

    while head1 and head2:

        if head1.data < head2.data:

            current.next = head1

            head1 = head1.next

        else:

            current.next = head2

            head2 = head2.next

        current = current.next

    if head1:

        current.next = head1

    if head2:

        current.next = head2

    return merged\_head

if \_\_name\_\_ == '\_\_main\_\_':

    fptr = open(os.environ['OUTPUT\_PATH'], 'w')

    tests = int(input())

    for tests\_itr in range(tests):

        llist1\_count = int(input())

        llist1 = SinglyLinkedList()

        for \_ in range(llist1\_count):

            llist1\_item = int(input())

            llist1.insert\_node(llist1\_item)

        llist2\_count = int(input())

        llist2 = SinglyLinkedList()

        for \_ in range(llist2\_count):

            llist2\_item = int(input())

            llist2.insert\_node(llist2\_item)

        llist3 = mergeLists(llist1.head, llist2.head)

        print\_singly\_linked\_list(llist3, ' ', fptr)

        fptr.write('\n')

    fptr.close()

Input (stdin)

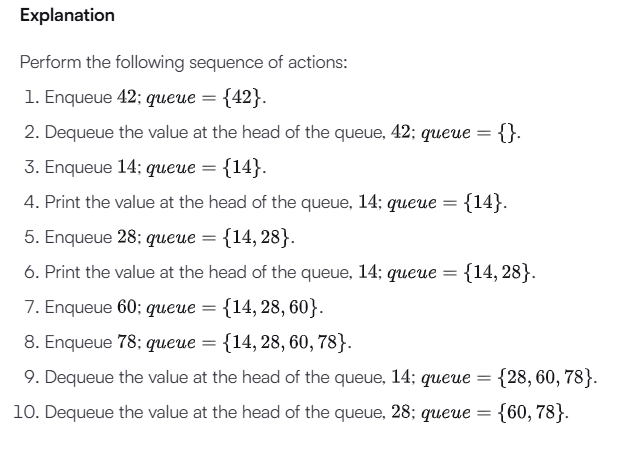
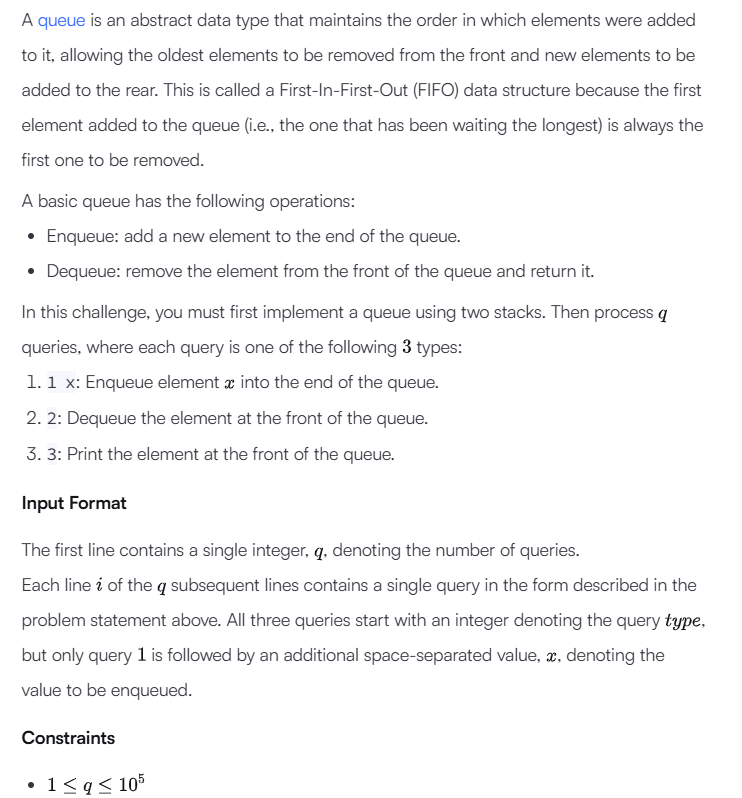
* **1**
* **3**
* **1**
* **2**
* **3**
* **2**
* **3**
* **4**

Your Output (stdout)

* **1 2 3 3 4**

Expected Output

* **1 2 3 3 4**

****

# Enter your code here. Read input from STDIN. Print output to STDOUT

class QueueUsingTwoStacks:

    def \_\_init\_\_(self):

        self.enqueue\_stack = []

        self.dequeue\_stack = []

    def enqueue(self, x):

        self.enqueue\_stack.append(x)

    def dequeue(self):

        if not self.dequeue\_stack:

            while self.enqueue\_stack:

                self.dequeue\_stack.append(self.enqueue\_stack.pop())

        if self.dequeue\_stack:

            return self.dequeue\_stack.pop()

        return None

    def front(self):

        if not self.dequeue\_stack:

            while self.enqueue\_stack:

                self.dequeue\_stack.append(self.enqueue\_stack.pop())

        if self.dequeue\_stack:

            return self.dequeue\_stack[-1]

        return None

q = int(input())

queue = QueueUsingTwoStacks()

for \_ in range(q):

    query = input().split()

    query\_type = int(query[0])

    if query\_type == 1:

        x = int(query[1])

        queue.enqueue(x)

    elif query\_type == 2:

        queue.dequeue()

    elif query\_type == 3:

        print(queue.front())

Input (stdin)

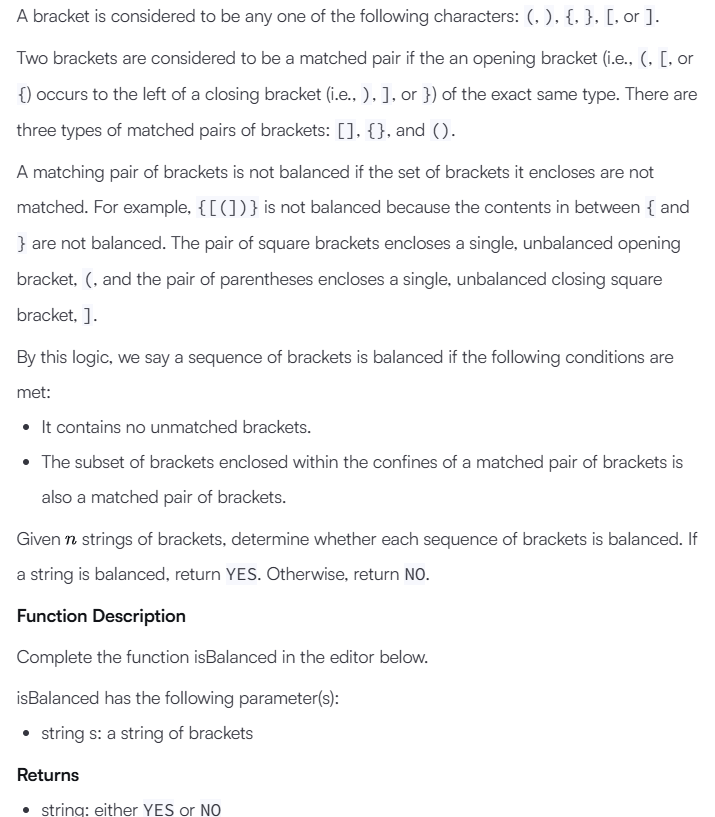
* **10**
* **1 42**
* **2**
* **1 14**
* **3**
* **1 28**
* **3**
* **1 60**
* **1 78**
* **2**
* **2**

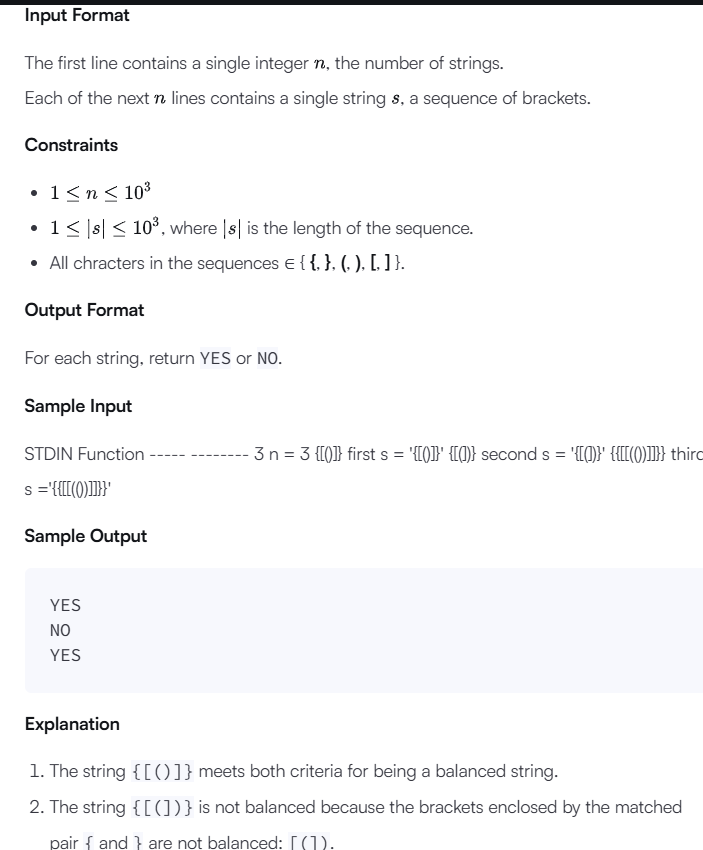
Your Output (stdout)

* **14**
* **14**

Expected Output

* **14**
* **14**

****

****

#!/bin/python3

import math

import os

import random

import re

import sys

#

# Complete the 'isBalanced' function below.

#

# The function is expected to return a STRING.

# The function accepts STRING s as parameter.

#

def isBalanced(s):

    stack = []

    matching\_brackets = {')': '(', '}': '{', ']': '['}

    for char in s:

        if char in matching\_brackets.values():

            stack.append(char)

        elif char in matching\_brackets.keys():

            if stack and stack[-1] == matching\_brackets[char]:

                stack.pop()

            else:

                return "NO"

    return "YES" if not stack else "NO"

    # Write your code here

if \_\_name\_\_ == '\_\_main\_\_':

    fptr = open(os.environ['OUTPUT\_PATH'], 'w')

    t = int(input().strip())

    for t\_itr in range(t):

        s = input()

        result = isBalanced(s)

        fptr.write(result + '\n')

    fptr.close()

Input (stdin)

* **3**
* **{[()]}**
* **{[(])}**
* **{{[[(())]]}}**

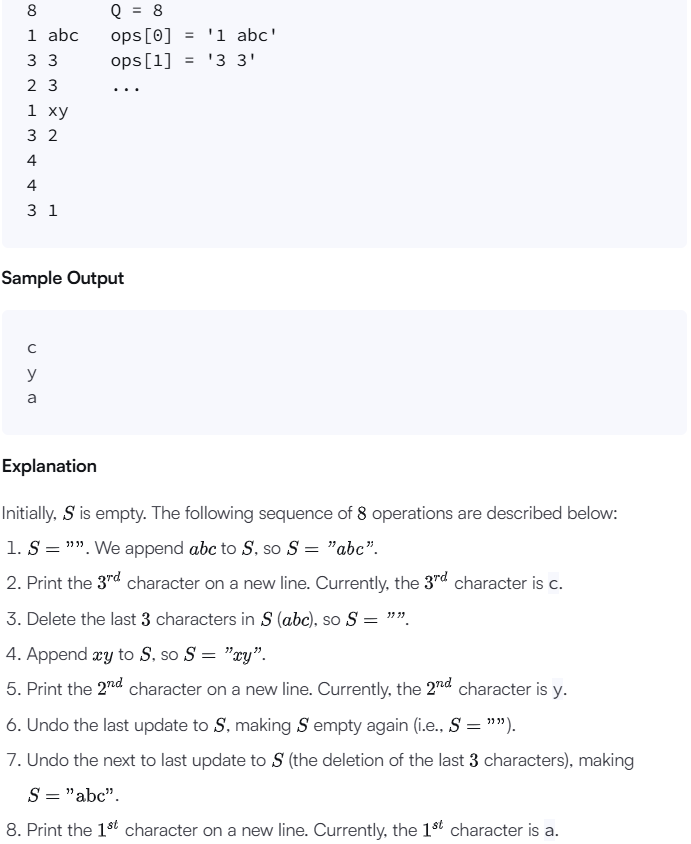
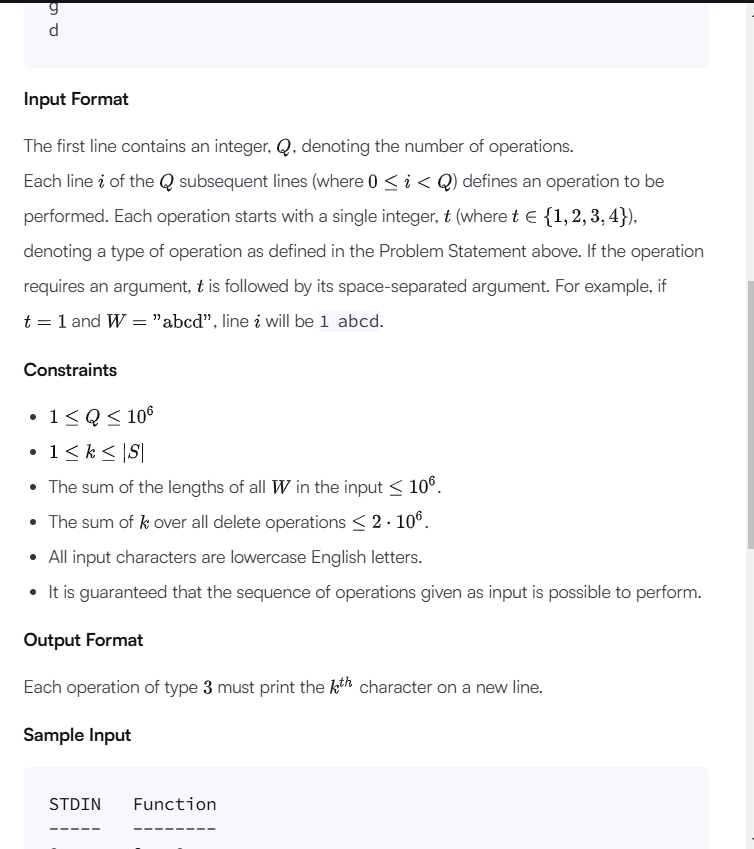
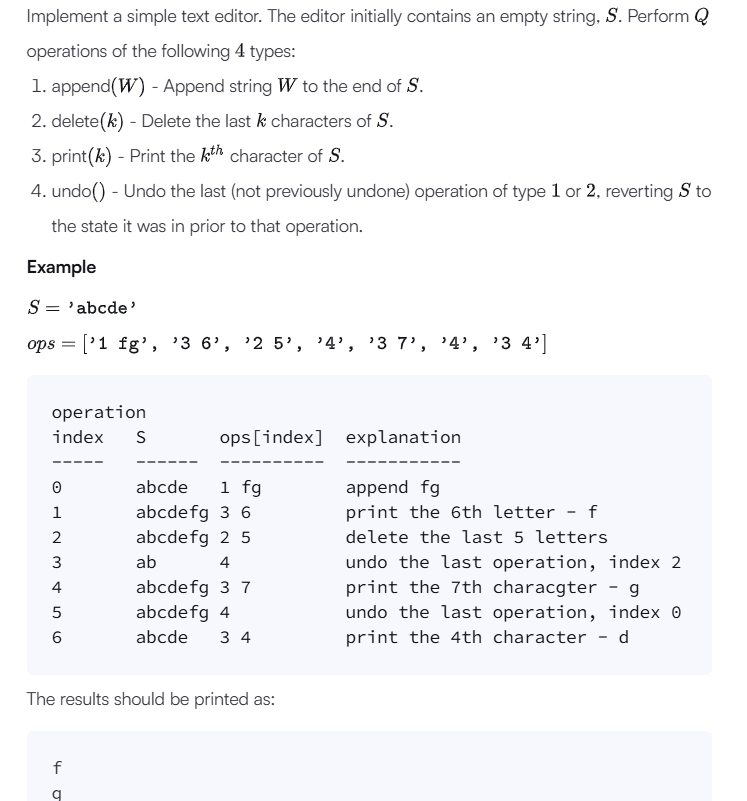
Your Output (stdout)

* **YES**
* **NO**
* **YES**

Expected Output

* **YES**
* **NO**
* **YES**

**Day 6:**

****

def text\_editor(operations):

    s = ""

    history = []

    for op in operations:

        op\_details = op.split()

        op\_type = int(op\_details[0])

        if op\_type == 1:

            string\_to\_append = op\_details[1]

            s += string\_to\_append

            history.append(("append", string\_to\_append))

        elif op\_type == 2:

            k = int(op\_details[1])

            deleted\_string = s[-k:]

            s = s[:-k]

            history.append(("delete", deleted\_string))

        elif op\_type == 3:

            k = int(op\_details[1]) - 1

            print(s[k])

        elif op\_type == 4:

            last\_op, value = history.pop()

            if last\_op == "append":

                s = s[:-len(value)]

            elif last\_op == "delete":

                s += value

# Input reading and handling

if \_\_name\_\_ == "\_\_main\_\_":

    q = int(input())  # Read the number of operations

    operations = [input().strip() for \_ in range(q)]

    text\_editor(operations)

Input (stdin)

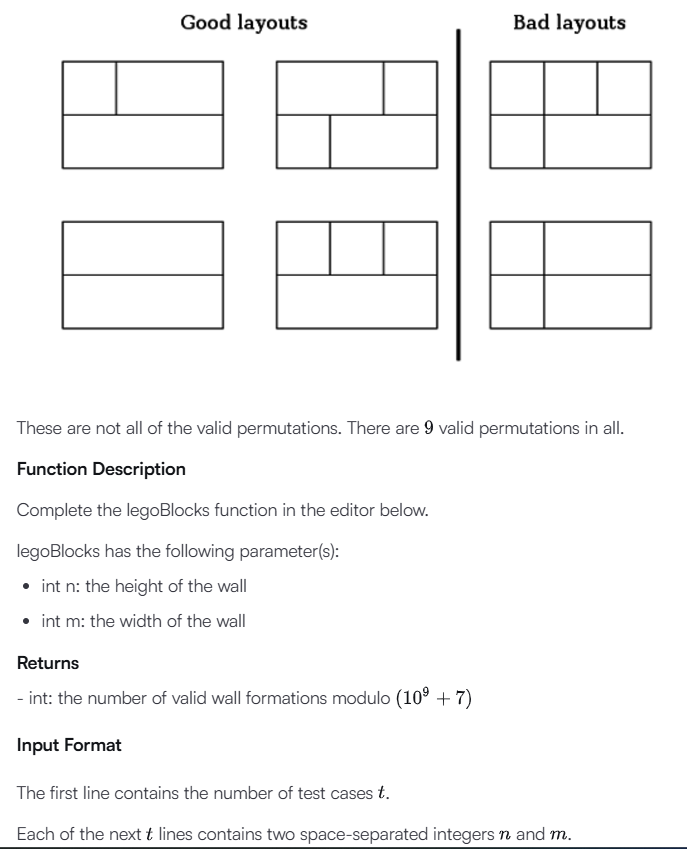
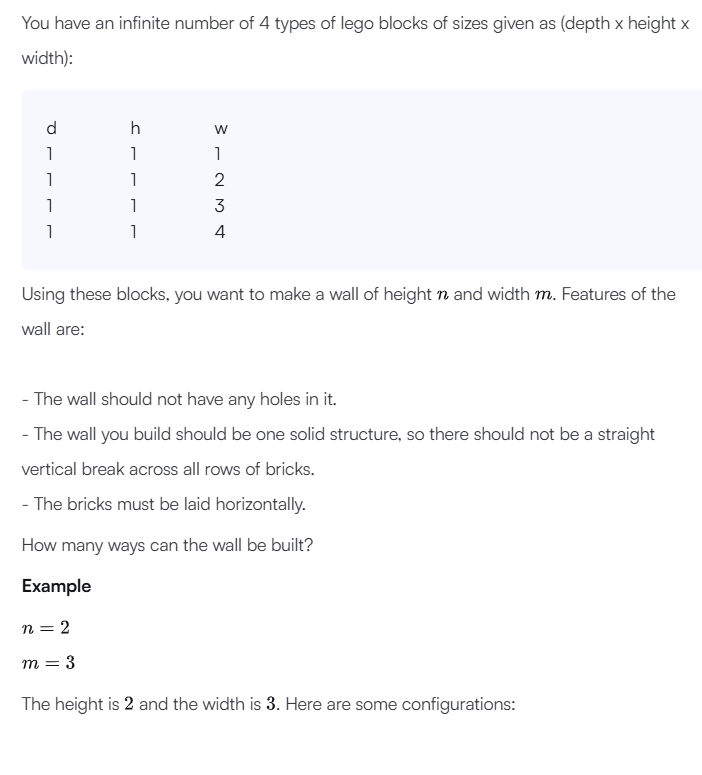
* **8**
* **1 abc**
* **3 3**
* **2 3**
* **1 xy**
* **3 2**
* **4**
* **4**
* **3 1**

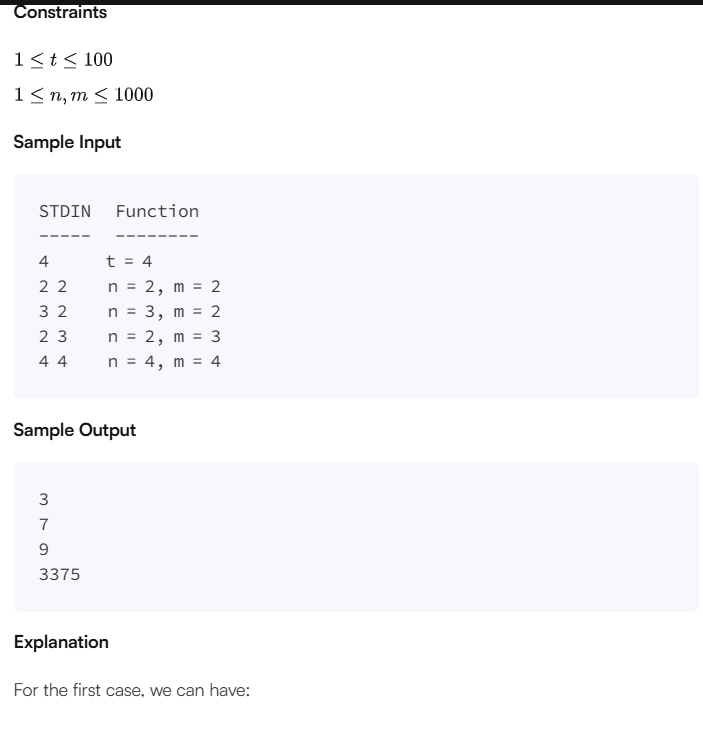
Your Output (stdout)

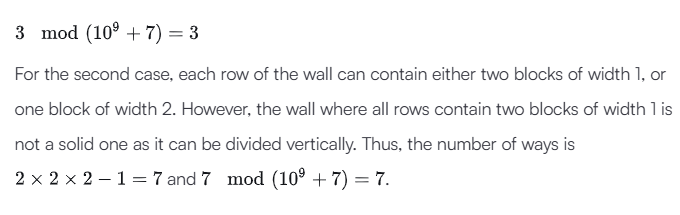
* **c**
* **y**
* **a**

Expected Output

* **c**
* **y**
* **a**

****

****

****

import math

import os

import random

import re

import sys

#

# Complete the 'legoBlocks' function below.

#

# The function is expected to return an INTEGER.

# The function accepts following parameters:

#  1. INTEGER n

#  2. INTEGER m

#

def legoBlocks(height, width):

    mod = 10 \*\* 9 + 7

    valid\_perms = [0] \* (width + 1)

    valid\_perms[0] = 1

    for w in range(1, width + 1):

        valid\_perms[w] = sum(valid\_perms[max(0, w - 4):w])

        valid\_perms[w] %= mod

    for w in range(width + 1):

        valid\_perms[w] = valid\_perms[w] \*\* height

        valid\_perms[w] %= mod

    valid = valid\_perms[:]

    for w in range(len(valid)):

        for separator in range(1, w):

            valid[w] -= valid[separator] \* valid\_perms[w-separator]

        valid[w] %= mod

    return valid[width]

if \_\_name\_\_ == '\_\_main\_\_':

    fptr = open(os.environ['OUTPUT\_PATH'], 'w')

    t = int(input().strip())

    for t\_itr in range(t):

        first\_multiple\_input = input().rstrip().split()

        n = int(first\_multiple\_input[0])

        m = int(first\_multiple\_input[1])

        result = legoBlocks(n, m)

        fptr.write(str(result) + '\n')

    fptr.close()

Input (stdin)

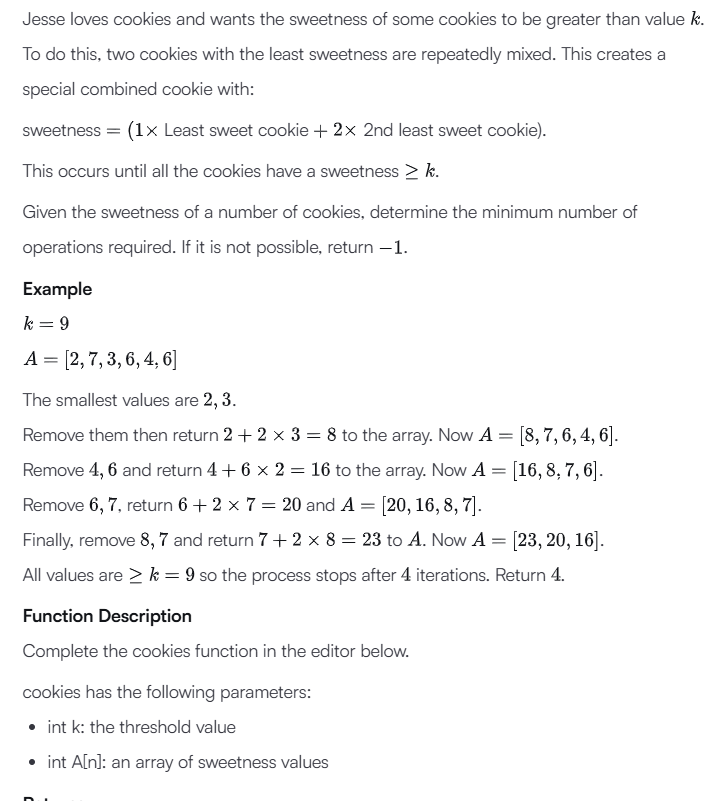
* **4**
* **2 2**
* **3 2**
* **2 3**
* **4 4**

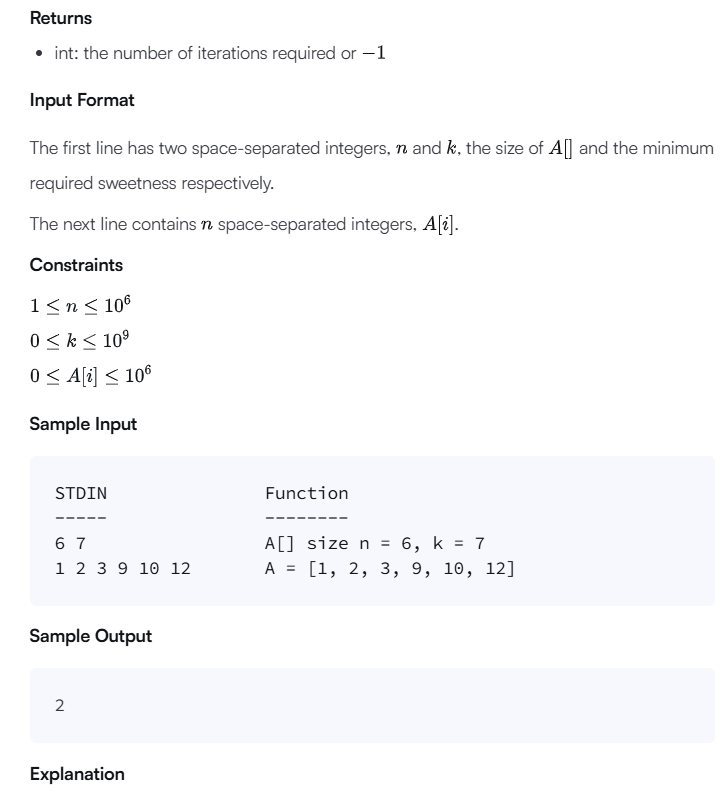
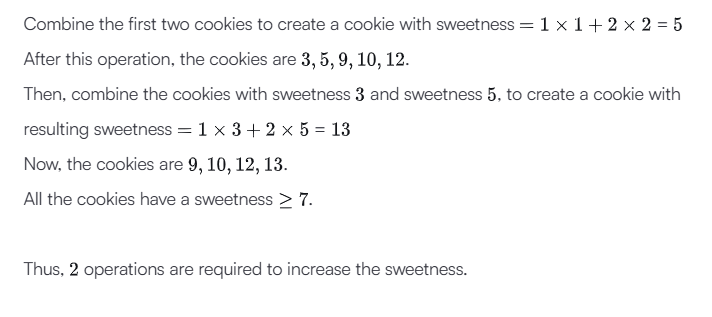
Your Output (stdout)

* **3**
* **7**
* **9**
* **3375**

Expected Output

* **3**
* **7**
* **9**
* **3375**

****

**  
**

import heapq

import os

def cookies(k, A):

    heapq.heapify(A)

    operations = 0

    while len(A) > 1:

        first = heapq.heappop(A)

        second = heapq.heappop(A)

        if first >= k:

            return operations

        new\_cookie = first + 2 \* second

        heapq.heappush(A, new\_cookie)

        operations += 1

    if A[0] < k:

        return -1

    return operations

if \_\_name\_\_ == '\_\_main\_\_':

    fptr = open(os.environ['OUTPUT\_PATH'], 'w')

    first\_multiple\_input = input().rstrip().split()

    n = int(first\_multiple\_input[0])

    k = int(first\_multiple\_input[1])

    A = list(map(int, input().rstrip().split()))

    result = cookies(k, A)

    fptr.write(str(result) + '\n')

    fptr.close()

Input (stdin)

* **6 7**
* **1 2 3 9 10 12**

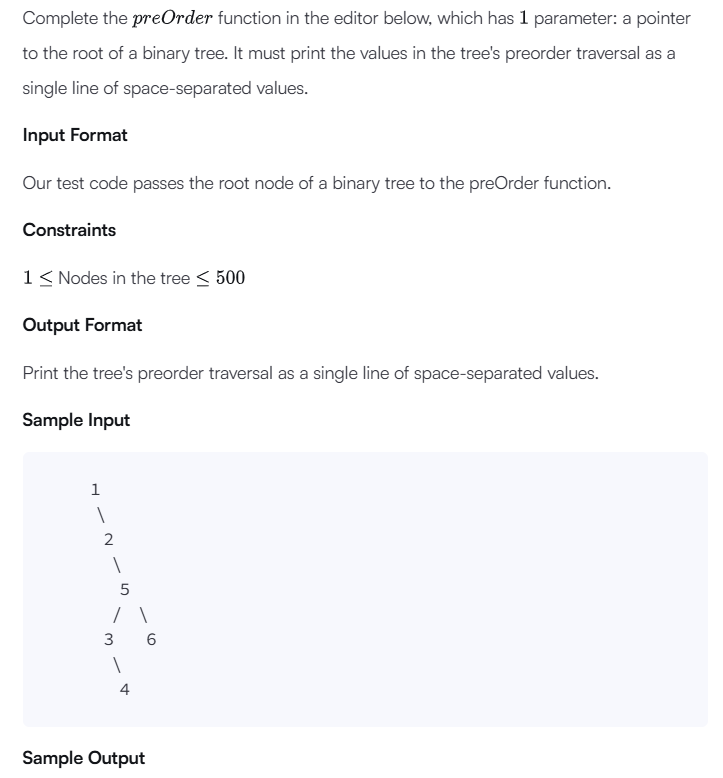
Your Output (stdout)

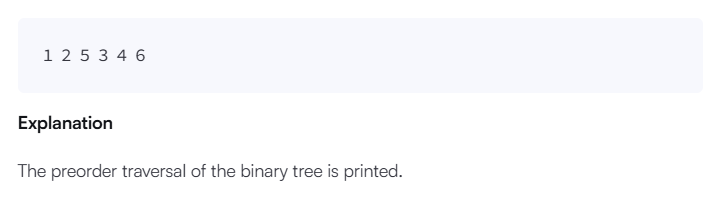
* **2**

Expected Output

* **2**

**Day 7:**

****



class Node:

    def \_\_init\_\_(self, info):

        self.info = info

        self.left = None

        self.right = None

        self.level = None

    def \_\_str\_\_(self):

        return str(self.info)

class BinarySearchTree:

    def \_\_init\_\_(self):

        self.root = None

    def create(self, val):

        if self.root == None:

            self.root = Node(val)

        else:

            current = self.root

            while True:

                if val < current.info:

                    if current.left:

                        current = current.left

                    else:

                        current.left = Node(val)

                        break

                elif val > current.info:

                    if current.right:

                        current = current.right

                    else:

                        current.right = Node(val)

                        break

                else:

                    break

"""

Node is defined as

self.left (the left child of the node)

self.right (the right child of the node)

self.info (the value of the node)

"""

def preOrder(root):

    if root:

        print(root.info, end=" ")

        preOrder(root.left)

        preOrder(root.right)

tree = BinarySearchTree()

t = int(input())

arr = list(map(int, input().split()))

for i in range(t):

    tree.create(arr[i])

preOrder(tree.root)

Input (stdin)

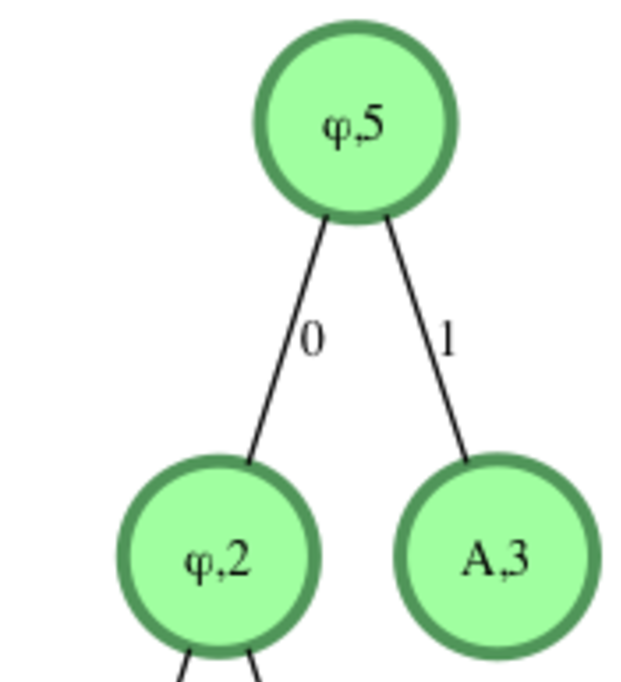
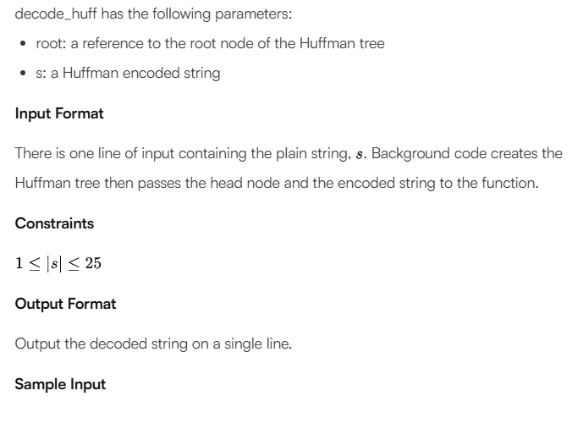
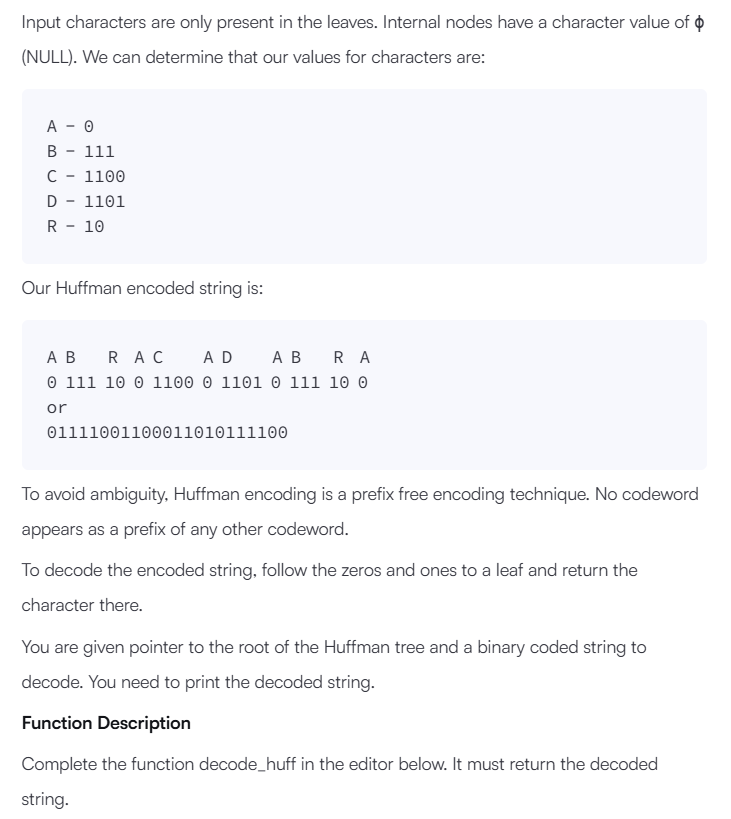
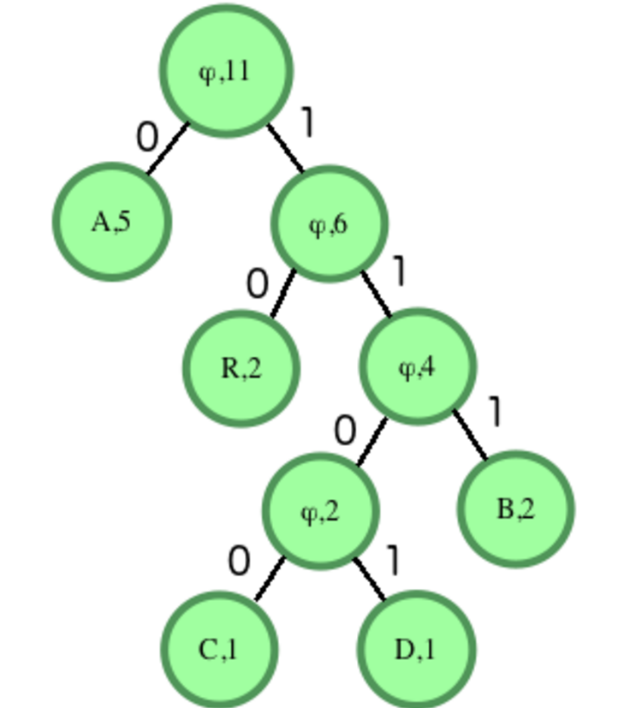
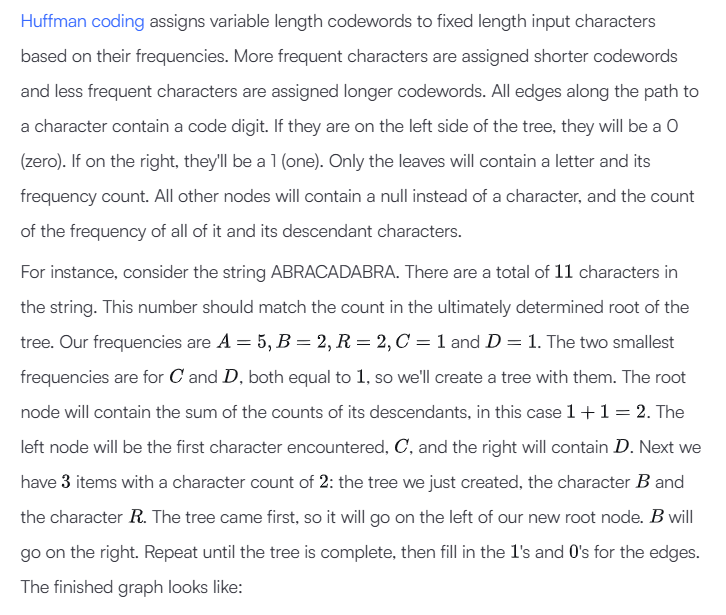
* **6**
* **1 2 5 3 6 4**

Your Output (stdout)

* **1 2 5 3 4 6**

Expected Output

* **1 2 5 3 4 6**

****

import queue as Queue

cntr = 0

class Node:

    def \_\_init\_\_(self, freq, data):

        self.freq = freq

        self.data = data

        self.left = None

        self.right = None

        global cntr

        self.\_count = cntr

        cntr = cntr + 1

    def \_\_lt\_\_(self, other):

        if self.freq != other.freq:

            return self.freq < other.freq

        return self.\_count < other.\_count

def huffman\_hidden():#builds the tree and returns root

    q = Queue.PriorityQueue()

    for key in freq:

        q.put((freq[key], key, Node(freq[key], key) ))

    while q.qsize() != 1:

        a = q.get()

        b = q.get()

        obj = Node(a[0] + b[0], '\0' )

        obj.left = a[2]

        obj.right = b[2]

        q.put((obj.freq, obj.data, obj ))

    root = q.get()

    root = root[2]#contains root object

    return root

def dfs\_hidden(obj, already):

    if(obj == None):

        return

    elif(obj.data != '\0'):

        code\_hidden[obj.data] = already

    dfs\_hidden(obj.right, already + "1")

    dfs\_hidden(obj.left, already + "0")

"""class Node:

    def \_\_init\_\_(self, freq,data):

        self.freq= freq

        self.data=data

        self.left = None

        self.right = None

"""

# Enter your code here. Read input from STDIN. Print output to STDOUT

def decodeHuff(root, s):

    decoded\_string = []

    current\_node = root

    for bit in s:

        if bit == '0':

            current\_node = current\_node.left

        else:

            current\_node = current\_node.right

        if current\_node.left is None and current\_node.right is None:

            decoded\_string.append(current\_node.data)

            current\_node = root

    print("".join(decoded\_string))

ip = input()

freq = {}#maps each character to its frequency

cntr = 0

for ch in ip:

    if(freq.get(ch) == None):

        freq[ch] = 1

    else:

        freq[ch]+=1

root = huffman\_hidden()#contains root of huffman tree

code\_hidden = {}#contains code for each object

dfs\_hidden(root, "")

if len(code\_hidden) == 1:#if there is only one character in the i/p

    for key in code\_hidden:

        code\_hidden[key] = "0"

toBeDecoded = ""

for ch in ip:

   toBeDecoded += code\_hidden[ch]

decodeHuff(root, toBeDecoded)

Input (stdin)

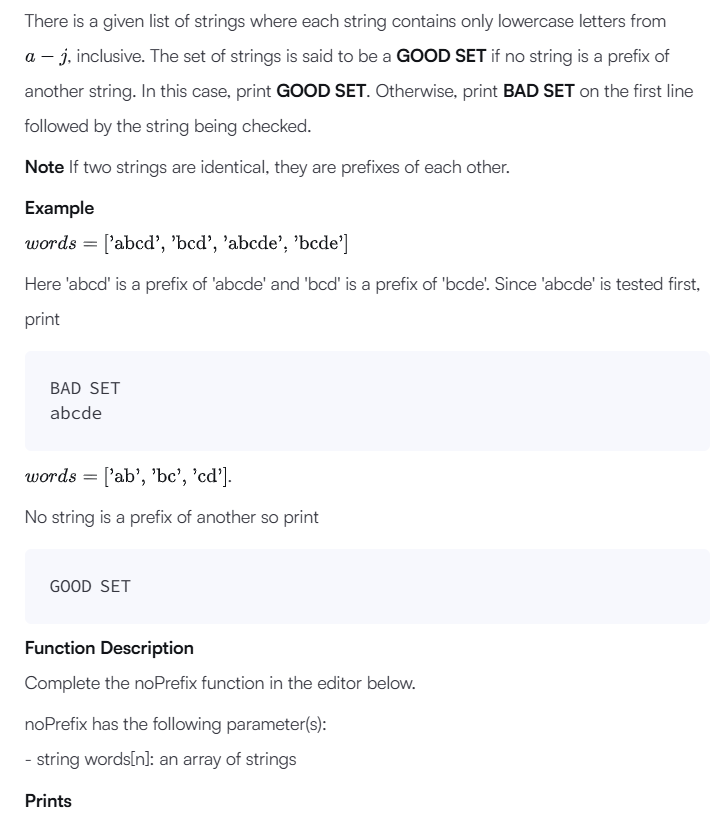
* **ABACA**

Your Output (stdout)

* **ABACA**

Expected Output

* **ABACA**

****

#!/bin/python3

import math

import os

import random

import re

import sys

#

# Complete the 'noPrefix' function below.

#

# The function accepts STRING\_ARRAY words as parameter.

#

class TrieNode:

    def \_\_init\_\_(self):

        self.children = {}

        self.is\_end\_of\_word = False

class Trie:

    def \_\_init\_\_(self):

        self.root = TrieNode()

    def insert(self, word):

        current = self.root

        for i, char in enumerate(word):

            if char not in current.children:

                current.children[char] = TrieNode()

            current = current.children[char]

            # If we are in the middle of the word and this is an end of another word

            if current.is\_end\_of\_word:

                print(f"BAD SET\n{word}")

                return False

        # If the current word is already a prefix of a previous word

        if current.children:

            print(f"BAD SET\n{word}")

            return False

        current.is\_end\_of\_word = True

        return True

def noPrefix(words):

    trie = Trie()

    for word in words:

        if not trie.insert(word):

            return

    print("GOOD SET")

if \_\_name\_\_ == '\_\_main\_\_':

    n = int(input().strip())

    words = []

    for \_ in range(n):

        words\_item = input()

        words.append(words\_item)

    noPrefix(words)

Input (stdin)

* **7**
* **aab**
* **defgab**
* **abcde**
* **aabcde**
* **cedaaa**
* **bbbbbbbbbb**
* **jabjjjad**

Your Output (stdout)

* **BAD SET**
* **aabcde**

Expected Output

* **BAD SET**
* **aabcde**