```python
import sys,math,random
from heapq import heappush,heappop
from bisect import bisect_right,bisect_left
from collections import Counter,deque,defaultdict
from itertools import permutations

# functions #
MOD = 998244353
MOD = 10**9 + 7
RANDOM = random.randrange(2**62)
def gcd(a,b):
    if a%b==0:
        return b
    else:
        return gcd(b,a%b)
def lcm(a,b):
    return a//gcd(a,b)*b
def w(x):
    return x ^ RANDOM
##

def solve():
    n = int(sys.stdin.readline().strip())
    L = list(map(int, sys.stdin.readline().split()))
    #L1 = list(map(int, sys.stdin.readline().split()))
    #st = sys.stdin.readline().strip()
for _ in range(int(sys.stdin.readline().strip())):
    solve()

HMOD = 2147483647
HBASE1 = random.randrange(HMOD)
HBASE2 = random.randrange(HMOD)
class Hashing:
    def __init__(self, s, mod=HMOD, base1=HBASE1, base2=HBASE2):
        self.mod, self.base1, self.base2 = mod, base1, base2
        self._len = _len = len(s)
        f_hash, f_pow = [0] * (_len + 1), [1] * (_len + 1)
        s_hash, s_pow = f_hash[:], f_pow[:]
        for i in range(_len):
            f_hash[i + 1] = (base1 * f_hash[i] + s[i]) % mod
            s_hash[i + 1] = (base2 * s_hash[i] + s[i]) % mod
            f_pow[i + 1] = base1 * f_pow[i] % mod
            s_pow[i + 1] = base2 * s_pow[i] % mod
        self.f_hash, self.f_pow = f_hash, f_pow
        self.s_hash, self.s_pow = s_hash, s_pow
    def hashed(self, start, stop):
        return (
            (self.f_hash[stop] - self.f_pow[stop - start] * self.f_hash[start]) % self.mod,
            (self.s_hash[stop] - self.s_pow[stop - start] * self.s_hash[start]) % self.mod,
        )
    def get_hashes(self, length):
        return (
            [(self.f_hash[i + length] - self.f_pow[length] * self.f_hash[i]) % self.mod for i in range(self._len - l
            [(self.s_hash[i + length] - self.s_pow[length] * self.s_hash[i]) % self.mod for i in range(self._len - l
        )

# def getSubHash(left,right,hash,power):
#     #returns hash value of [left,right]
#     #check for two diff hash,power to prevent collision
#     #change MOD according to how many values you want ,
#     #the more the value , lesser the collisiosn
#     left += 1
#     right += 1
#     h1 = hash[right]
#     h2 = (hash[left - 1] * power[right - left + 1]) % MOD
#     return (h1 + MOD - h2) % MOD
# n = len(st)
# hash = [0] * (n + 1)
# # hash1 = [0] * (n + 1)
# power = [0] * (n + 1)
# # power1 = [0] * (n + 1)
# P = 131
# # P1 = 132
# power[0] = 1
# # power1[0] = 1
# for i in range(n):
#     power[i + 1] = (power[i] * P) % MOD
#     # power1[i+1] = (power1[i] * P1) % MOD
#     hash[i + 1] = (hash[i] * P + ord(st[i])) % MOD
#     # hash1[i + 1] = (hash1[i] * P1 + ord(st[i])) % MOD

def z_function(S):
```

```
    # return: the Z array, where Z[i] = length of the longest common prefix of S[i:] and S
    n = len(S)
    Z = [0] * n
    l = r = 0
    for i in range(1, n):
        z = Z[i - l]
        if i + z >= r:
            z = max(r - i, 0)
            while i + z < n and S[z] == S[i + z]:
                z += 1
            l, r = i, i + z
        Z[i] = z
    Z[0] = n
    return Z

def matmul(A,B,MOD=(10**9 + 7)):
    ans = [[0 for i in range(len(B[0]))] for j in range(len(A))]
    for i in range(len(A)):
        for j in range(len(B[0])):
            for k in range(len(B)):
                ans[i][j] = (ans[i][j]+A[i][k]*B[k][j])%MOD
    return ans

def matpow(M,power):
    size = len(M)
    result = [[1 if i == j else 0 for j in range(size)] for i in range(size)]
    while power:
        if power % 2 == 1:
            result = matmul(result, M)
        M = matmul(M, M)
        power //= 2
    return result

def sieve(n):
    primes = []
    isp = [1] * (n+1)
    isp[0] = isp[1] = 0
    for i in range(2,n+1):
        if isp[i]:
            primes.append(i)
            for j in range(i*i,n+1,i):
                isp[j] = 0
    return primes

def miller_is_prime(n):
    """
        Miller-Rabin test - O(7 * log2n)
        Has 100% success rate for numbers less than 3e+9
        use it in case of TC problem
    """
    if n < 5 or n & 1 == 0 or n % 3 == 0:
        return 2 <= n <= 3
    s = ((n - 1) & (1 - n)).bit_length() - 1
    d = n >> s
    for a in [2, 325, 9375, 28178, 450775, 9780504, 1795265022]:
        p = pow(a, d, n)
        if p == 1 or p == n - 1 or a % n == 0:
            continue
        for _ in range(s):
            p = (p * p) % n
            if p == n - 1:
                break
        else:
            return False
    return True

def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

def sieve_unique(N):
    mini = [i for i in range(N)]
```

```
    for i in range(2,N):
        if mini[i]==i:
            for j in range(2*i,N,i):
                mini[j] = i
    return mini

def prime_factors(k):
    """
        When the numbers are large this is the best method to get
        unique prime factors, precompute n log n log n , then each query is log n
    """
    Lmini = [] #precalculate this upto the number required
    # this should not be here , it should be global and contain the mini made in sieve_unique
    # dont forget
    ans = []
    while k!=1:
        ans.append(Lmini[k])
        k //= Lmini[k]
    return ans

class SparseTable:
    @staticmethod
    def func(a,b):
        return gcd(a,b)
    def __init__(self, arr):
        self.n = len(arr)
        self.table = [[0 for i in range(int((math.log(self.n, 2)+1)))] for j in range(self.n)]
        self.build(arr)
    def build(self, arr):
        for i in range(0, self.n):
            self.table[i][0] = arr[i]
        j = 1
        while (1 << j) <= self.n:
            i = 0
            while i <= self.n - (1 << j):
                self.table[i][j] = self.func(self.table[i][j - 1], self.table[i + (1 << (j - 1))][j - 1])
                i += 1
            j += 1
    def query(self, L, R):
        j = int(math.log2(R - L + 1))
        return self.func(self.table[L][j], self.table[R - (1 << j) + 1][j])

class FenwickTree:
    def __init__(self, x):
        bit = self.bit = list(x)
        size = self.size = len(bit)
        for i in range(size):
            j = i | (i + 1)
            if j < size:
                bit[j] += bit[i]
    def update(self, idx, x):
        while idx < self.size:
            self.bit[idx] += x
            idx |= idx + 1
    def __call__(self, end):
        x = 0
        while end:
            x += self.bit[end - 1]
            end &= end - 1
        return x
    def find_kth(self, k):
        idx = -1
        for d in reversed(range(self.size.bit_length())):
            right_idx = idx + (1 << d)
            if right_idx < self.size and self.bit[right_idx] <= k:
                idx = right_idx
                k -= self.bit[idx]
        return idx + 1, k
class SortedList:
    block_size = 700
    def __init__(self, iterable=()):
        self.macro = []
        self.micros = [[]]
        self.micro_size = [0]
        self.fenwick = FenwickTree([0])
        self.size = 0
        for item in iterable:
            self.insert(item)
    def insert(self, x):
        i = bisect_left(self.macro, x)
        j = bisect_right(self.micros[i], x)
        self.micros[i].insert(j, x)
```

```
        self.size += 1
        self.micro_size[i] += 1
        self.fenwick.update(i, 1)
        if len(self.micros[i]) >= self.block_size:
            self.micros[i:i + 1] = self.micros[i][:self.block_size >> 1], self.micros[i][self.block_size >> 1:]
            self.micro_size[i:i + 1] = self.block_size >> 1, self.block_size >> 1
            self.fenwick = FenwickTree(self.micro_size)
            self.macro.insert(i, self.micros[i + 1][0])
    def add(self, x):
        self.insert(x)
    def pop(self, k=-1):
        i, j = self._find_kth(k)
        self.size -= 1
        self.micro_size[i] -= 1
        self.fenwick.update(i, -1)
        return self.micros[i].pop(j)
    def remove(self, N: int):
        idx = self.bisect_left(N)
        self.pop(idx)
    def __getitem__(self, k):
        i, j = self._find_kth(k)
        return self.micros[i][j]
    def count(self, x):
        return self.bisect_right(x) - self.bisect_left(x)
    def __contains__(self, x):
        return self.count(x) > 0
    def bisect_left(self, x):
        i = bisect_left(self.macro, x)
        return self.fenwick(i) + bisect_left(self.micros[i], x)
    def bisect_right(self, x):
        i = bisect_right(self.macro, x)
        return self.fenwick(i) + bisect_right(self.micros[i], x)
    def _find_kth(self, k):
        return self.fenwick.find_kth(k + self.size if k < 0 else k)
    def __len__(self):
        return self.size
    def __iter__(self):
        return (x for micro in self.micros for x in micro)
    def __repr__(self):
        return str(list(self))

from typing import Generic, Iterable, Iterator, List, Tuple, TypeVar, Optional
T = TypeVar('T')
class SortedList(Generic[T]):
    BUCKET_RATIO = 16
    SPLIT_RATIO = 24
    def __init__(self, a: Iterable[T] = []) -> None:
        a = list(a)
        n = self.size = len(a)
        if any(a[i] > a[i + 1] for i in range(n - 1)):
            a.sort()
        num_bucket = int(math.ceil(math.sqrt(n / self.BUCKET_RATIO)))
        self.a = [a[n * i // num_bucket : n * (i + 1) // num_bucket] for i in range(num_bucket)]
    def __iter__(self) -> Iterator[T]:
        for i in self.a:
            for j in i: yield j
    def __reversed__(self) -> Iterator[T]:
        for i in reversed(self.a):
            for j in reversed(i): yield j
    def __eq__(self, other) -> bool:
        return list(self) == list(other)
    def __len__(self) -> int:
        return self.size
    def __repr__(self) -> str:
        return "SortedMultiset" + str(self.a)
    def __str__(self) -> str:
        s = str(list(self))
        return "{" + s[1 : len(s) - 1] + "}"
    def _position(self, x: T) -> Tuple[List[T], int, int]:
        for i, a in enumerate(self.a):
            if x <= a[-1]: break
        return (a, i, bisect_left(a, x))
    def __contains__(self, x: T) -> bool:
        if self.size == 0: return False
        a, _, i = self._position(x)
        return i != len(a) and a[i] == x
    def count(self, x: T) -> int:
        return self.index_right(x) - self.index(x)
    def insert(self, x: T) -> None:
        if self.size == 0:
            self.a = [[x]]
            self.size = 1
```

```
            return
        a, b, i = self._position(x)
        a.insert(i, x)
        self.size += 1
        if len(a) > len(self.a) * self.SPLIT_RATIO:
            mid = len(a) >> 1
            self.a[b:b+1] = [a[:mid], a[mid:]]
    def _pop(self, a: List[T], b: int, i: int) -> T:
        ans = a.pop(i)
        self.size -= 1
        if not a: del self.a[b]
        return ans
    def remove(self, x: T) -> bool:
        if self.size == 0: return False
        a, b, i = self._position(x)
        if i == len(a) or a[i] != x: return False
        self._pop(a, b, i)
        return True
    def lt(self, x: T) -> Optional[T]:
        for a in reversed(self.a):
            if a[0] < x:
                return a[bisect_left(a, x) - 1]
    def le(self, x: T) -> Optional[T]:
        for a in reversed(self.a):
            if a[0] <= x:
                return a[bisect_right(a, x) - 1]
    def gt(self, x: T) -> Optional[T]:
        for a in self.a:
            if a[-1] > x:
                return a[bisect_right(a, x)]
    def ge(self, x: T) -> Optional[T]:
        for a in self.a:
            if a[-1] >= x:
                return a[bisect_left(a, x)]
    def __getitem__(self, i: int) -> T:
        if i < 0:
            for a in reversed(self.a):
                i += len(a)
                if i >= 0: return a[i]
        else:
            for a in self.a:
                if i < len(a): return a[i]
                i -= len(a)
        raise IndexError
    def pop(self, i: int = -1) -> T:
        if i < 0:
            for b, a in enumerate(reversed(self.a)):
                i += len(a)
                if i >= 0: return self._pop(a, ~b, i)
        else:
            for b, a in enumerate(self.a):
                if i < len(a): return self._pop(a, b, i)
                i -= len(a)
        raise IndexError
    def index(self, x: T) -> int:
        ans = 0
        for a in self.a:
            if a[-1] >= x:
                return ans + bisect_left(a, x)
            ans += len(a)
        return ans
    def index_right(self, x: T) -> int:
        ans = 0
        for a in self.a:
            if a[-1] > x:
                return ans + bisect_right(a, x)
            ans += len(a)
        return ans
    def find_closest(self, k: T) -> Optional[T]:
        if self.size == 0:
            return None
        ltk = self.le(k)
        gtk = self.ge(k)
        if ltk is None:
            return gtk
        if gtk is None:
            return ltk
        if abs(k-ltk)<=abs(k-gtk):
            return ltk
        else:
            return gtk
```

```python
from types import GeneratorType

def bootstrap(f, stack=[]):
    def wrappedfunc(*args, **kwargs):
        if stack:
            return f(*args, **kwargs)
        else:
            to = f(*args, **kwargs)
            while True:
                if type(to) is GeneratorType:
                    stack.append(to)
                    to = next(to)
                else:
                    stack.pop()
                    if not stack:
                        break
                    to = stack[-1].send(to)
            return to
    return wrappedfunc

class BIT:
    #Faster than segment tree so use if possbile
    def __init__(self, x):
        """transform list into BIT"""
        self.bit = x
        for i in range(len(x)):
            j = i | (i + 1)
            if j < len(x):
                x[j] += x[i]
    def update(self, idx, x):
        """updates bit[idx] += x"""
        #basically after that number greater greater than x will be added
        while idx < len(self.bit):
            self.bit[idx] += x
            idx |= idx + 1
    def query(self, end):
        """calc sum(bit[:end])"""
        #gives sum of element before end
        x = 0
        while end:
            x += self.bit[end - 1]
            end &= end - 1
        return x
    def findkth(self, k):
        """Find largest idx such that sum(bit[:idx]) <= k"""
        idx = -1
        for d in reversed(range(len(self.bit).bit_length())):
            right_idx = idx + (1 << d)
            if right_idx < len(self.bit) and k >= self.bit[right_idx]:
                idx = right_idx
                k -= self.bit[idx]
        return idx + 1

class SegmentTree:
    # all the operations in here are inclusive of l to r
    # later on make it custom for each func like seg point
    @staticmethod
    def func(a, b):
        # Change this function depending upon needs
        return a+b
    def __init__(self, arr):
        self.arr = arr
        self.n = len(arr)
        self.tree = [0] * (4 * self.n)
        self.lazy = [0] * (4 * self.n)
        self.build_tree(1, 0, self.n - 1)
    def build_tree(self, node, start, end):
        if start == end:
            self.tree[node] = self.arr[start]
        else:
            mid = (start + end) // 2
            self.build_tree(2 * node, start, mid)
            self.build_tree(2 * node + 1, mid + 1, end)
            self.tree[node] = self.func(self.tree[2 * node], self.tree[2 * node + 1])
    def propagate_lazy(self, node, start, end):
        if self.lazy[node] != 0:
            self.tree[node] += self.lazy[node]
            if start != end:
                self.lazy[2 * node] += self.lazy[node]
                self.lazy[2 * node + 1] += self.lazy[node]
            self.lazy[node] = 0
    def update(self, node, start, end, l, r, value):
```

```
        self.propagate_lazy(node, start, end)
        if start > r or end < l:
            return
        if start >= l and end <= r:
            self.tree[node] += value
            if start != end:
                self.lazy[2 * node] += value
                self.lazy[2 * node + 1] += value
            return
        mid = (start + end) // 2
        self.update(2 * node, start, mid, l, r, value)
        self.update(2 * node + 1, mid + 1, end, l, r, value)
        self.tree[node] = self.func(self.tree[2 * node], self.tree[2 * node + 1])
    def range_update(self, l, r, value):
        self.update(1, 0, self.n - 1, l, r, value)
    def query(self, node, start, end, l, r):
        self.propagate_lazy(node, start, end)
        if start > r or end < l:
            return 0
        if start >= l and end <= r:
            return self.tree[node]
        mid = (start + end) // 2
        return self.func(self.query(2 * node, start, mid, l, r),
                         self.query(2 * node + 1, mid + 1, end, l, r))
    def range_query(self, l, r):
        return self.query(1, 0, self.n - 1, l, r)
    def to_list(self):
        result = []
        for i in range(self.n):
            result.append(self.range_query(i, i))
        return result

class SegmentTree:
    def __init__(self, data, default=0, func=max):
        # don't forget to change func here
        # default is the value given to it byy default
        self._default = default
        self._func = func

        self._len = len(data)
        self._size = _size = 1 << (self._len - 1).bit_length()
        self._lazy = [0] * (2 * _size)

        self.data = [default] * (2 * _size)
        self.data[_size:_size + self._len] = data
        for i in reversed(range(_size)):
            self.data[i] = func(self.data[i + i], self.data[i + i + 1])
    def __len__(self):
        return self._len
    def _push(self, idx):
        q, self._lazy[idx] = self._lazy[idx], 0
        self._lazy[2 * idx] += q
        self._lazy[2 * idx + 1] += q
        self.data[2 * idx] += q
        self.data[2 * idx + 1] += q
    def _update(self, idx):
        for i in reversed(range(1, idx.bit_length())):
            self._push(idx >> i)
    def _build(self, idx):
        idx >>= 1
        while idx:
            self.data[idx] = self._func(self.data[2 * idx], self.data[2 * idx + 1]) + self._lazy[idx]
            idx >>= 1
    def add(self, start, stop, value):
        # lazily add value to [start, stop)
        start = start_copy = start + self._size
        stop = stop_copy = stop + self._size
        while start < stop:
            if start & 1:
                self._lazy[start] += value
                self.data[start] += value
                start += 1
            if stop & 1:
                stop -= 1
                self._lazy[stop] += value
                self.data[stop] += value
            start >>= 1
            stop >>= 1
        self._build(start_copy)
        self._build(stop_copy - 1)
    def query(self, start, stop, default=-float('inf')):
        # func of data[start, stop)
```

```
        # don't forget to update the default
        start += self._size
        stop += self._size
        self._update(start)
        self._update(stop - 1)
        res = default
        while start < stop:
            if start & 1:
                res = self._func(res, self.data[start])
                start += 1
            if stop & 1:
                stop -= 1
                res = self._func(res, self.data[stop])
            start >>= 1
            stop >>= 1
        return res
    def __repr__(self):
        return "LazySegmentTree({0})".format(self.data)

class SegmentTree:
    """
        Remember to change the func content as well as the initializer to display the content
    """
    @staticmethod
    def func(a, b):
        # Change this function depending upon needs
        return max(a, b)
    def __init__(self, data):
        self.n = len(data)
        self.tree = [0] * (2 * self.n)
        self.build(data)
    def build(self, data):
        for i in range(self.n):
            self.tree[self.n + i] = data[i]
        for i in range(self.n - 1, 0, -1):
            self.tree[i] = self.func(self.tree[i * 2], self.tree[i * 2 + 1])
    def update(self, pos, value):
        # Update the value at the leaf node
        pos += self.n
        # For updating
        self.tree[pos] = value
        # self.tree[pos] += value
        # If you want to add rather than update
        while pos > 1:
            pos //= 2
            self.tree[pos] = self.func(self.tree[2 * pos], self.tree[2 * pos + 1])
    def query(self, left, right):
        # Query the maximum value in the range [left, right)
        left += self.n
        right += self.n
        # Change the initializer depending upon the self.func
        max_val = float('-inf')
        ##
        while left < right:
            if left % 2:
                max_val = self.func(max_val, self.tree[left])
                left += 1
            if right % 2:
                right -= 1
                max_val = self.func(max_val, self.tree[right])
            left //= 2
            right //= 2
        return max_val

def convex_hull_trick(m, c, integer=True):
    """
    Given lines on the form y = m[i] * x + c[i] this function returns intervals,
    such that on each interval the convex hull is made up of a single line.
    Input:
        m: list of the slopes
        c: list of the constants (value at x = 0)
        integer: boolean for turning on / off integer mode. Integer mode is exact, it
            works by effectively flooring the separators of the intervals.
    Return:
        hull_i: on interval j, line i = hull_i[j] is >= all other lines
        hull_x: interval j and j + 1 is separated by x = hull_x[j], (hull_x[j] is the last x in interval j)
    """
    if integer:
        intersect = lambda i, j: (c[j] - c[i]) // (m[i] - m[j])
    else:
        intersect = lambda i, j: (c[j] - c[i]) / (m[i] - m[j])
    hull_i = []
```

```
    hull_x = []
    order = sorted(range(len(m)), key=m.__getitem__)
    for i in order:
        while True:
            if not hull_i:
                hull_i.append(i)
                break
            elif m[hull_i[-1]] == m[i]:
                if c[hull_i[-1]] >= c[i]:
                    break
                hull_i.pop()
                if hull_x: hull_x.pop()
            else:
                x = intersect(i, hull_i[-1])
                if hull_x and x <= hull_x[-1]:
                    hull_i.pop()
                    hull_x.pop()
                else:
                    hull_i.append(i)
                    hull_x.append(x)
                    break
    return hull_i, hull_x
def max_query(x, m, c, hull_i, hull_x):
    """ Find maximum value at x in O(log n) time """
    i = hull_i[bisect_left(hull_x, x)]
    return m[i] * x + c[i]

class Factorial:
    def __init__(self, N, mod):
        N += 1
        self.mod = mod
        self.f = [1 for _ in range(N)]
        self.g = [1 for _ in range(N)]
        for i in range(1, N):
            self.f[i] = self.f[i - 1] * i % self.mod
        self.g[-1] = pow(self.f[-1], mod - 2, mod)
        for i in range(N - 2, -1, -1):
            self.g[i] = self.g[i + 1] * (i + 1) % self.mod
    def fac(self, n):
        return self.f[n]
    def fac_inv(self, n):
        return self.g[n]
    def combi(self, n, m):
        if m == 0: return 1
        if n < m or m < 0 or n < 0: return 0
        return self.f[n] * self.g[m] % self.mod * self.g[n - m] % self.mod
    def permu(self, n, m):
        if n < m or m < 0 or n < 0: return 0
        return self.f[n] * self.g[n - m] % self.mod
    def catalan(self, n):
        return (self.combi(2 * n, n) - self.combi(2 * n, n - 1)) % self.mod
    def inv(self, n):
        return self.f[n-1] * self.g[n] % self.mod

class DiophantineEquations:
    """
        used for solving equations of the form a*x + b*y = c,
        soln1l takes the lower limit as well
    """
    def __init__(self):
        pass
    def euclidean_gcd(self,a, b):
        """
            euclidean gcd , returns x and y such that
            a*x + b*y = gcd(a,b)
        """
        if b == 0:
            return a,1,0
        g,x1,y1 = self.euclidean_gcd(b, a % b)
        x = y1
        y = x1-(a//b)*y1
        return g,x,y
    def soln(self,a,b,c,t=10**18,t1=10**18):
        """
            return m and n such that a*m + b*n = c and 0<=m<=t and 0<=n<=t1
            don't input t,t1 for any possible value
        """
        g = gcd(a, b)
        if c%g!=0:
            return -1,-1
        _, x, y = self.euclidean_gcd(a, b)
        k3 = x*(c//g)
```

```
        n1 = y*(c//g)
        k1 = max(math.ceil(-k3*g/b),math.ceil((n1-t1)/(a/g)))
        kmaxi = min(math.floor((t-k3)/(b/g)), math.floor(n1/(a/g)))
        if k1<=kmaxi:
            k = k1
            m = k3+k*(b//g)
            n = n1-k*(a//g)
            return m, n
        else:
            return -1,-1
    def sollnll(self,a,b,c,t,t1,m_lower,n_lower):
        """
            return m and n such that a*m + b*n = c and
            m_lower <= m <= t and n_lower <= n <= t1
        """
        g = gcd(a, b)
        if c % g != 0:
            return -1, -1
        _, x, y = self.euclidean_gcd(a, b)
        k3 = x * (c // g)
        n1 = y * (c // g)
        k1 = max(math.ceil((m_lower - k3 * g) / b), math.ceil((n1 - t1) / (a / g)))
        kmaxi = min(math.floor((t - k3) / (b / g)), math.floor((n1 - n_lower) / (a / g)))
        if k1 <= kmaxi:
            k = k1
            m = k3 + k * (b // g)
            n = n1 - k * (a // g)
            if m_lower <= m <= t and n_lower <= n <= t1:
                return m, n
            else:
                return -1,-1
        else:
            return -1, -1
Leq = DiophantineEquations()

def bellman_ford(n, edges, start):
    dist = [float("inf")] * n
    pred = [None] * n
    dist[start] = 0
    for _ in range(n):
        for u, v, d in edges:
            if dist[u] + d < dist[v]:
                dist[v] = dist[u] + d
                pred[v] = u
    # for u, v, d in edges:
    #  if dist[u] + d < dist[v]:
    #   return -1
    # This returns -1 , if there is a negative cycle

class binary_lift:
    def __init__(self, graph, data=(), f=min, root=0):
        n = len(graph)
        parent = [-1] * (n + 1)
        depth = self.depth = [-1] * n
        bfs = [root]
        depth[root] = 0
        for node in bfs:
            for nei in graph[node]:
                if depth[nei] == -1:
                    parent[nei] = node
                    depth[nei] = depth[node] + 1
                    bfs.append(nei)

        data = self.data = [data]
        parent = self.parent = [parent]
        self.f = f

        for _ in range(max(depth).bit_length()):
            old_data = data[-1]
            old_parent = parent[-1]
            data.append([f(val, old_data[p]) for val,p in zip(old_data, old_parent)])
            parent.append([old_parent[p] for p in old_parent])

    def lca(self, a, b):
        depth = self.depth
        parent = self.parent
        if depth[a] < depth[b]:
            a,b = b,a
        d = depth[a] - depth[b]
        for i in range(d.bit_length()):
            if (d >> i) & 1:
                a = parent[i][a]
```

```
        for i in range(depth[a].bit_length())[::-1]:
            if parent[i][a] != parent[i][b]:
                a = parent[i][a]
                b = parent[i][b]
        if a != b:
            return parent[0][a]
        else:
            return a
    def distance(self, a, b):
        return self.depth[a] + self.depth[b] - 2 * self.depth[self.lca(a,b)]
    def kth_ancestor(self, a, k):
        parent = self.parent
        if self.depth[a] < k:
            return -1
        for i in range(k.bit_length()):
            if (k >> i) & 1:
                a = parent[i][a]
        return a
    def __call__(self, a, b):
        depth = self.depth
        parent = self.parent
        data = self.data
        f = self.f
        c = self.lca(a, b)
        val = data[0][c]
        for x,d in (a, depth[a] - depth[c]), (b, depth[b] - depth[c]):
            for i in range(d.bit_length()):
                if (d >> i) & 1:
                    val = f(val, data[i][x])
                    x = parent[i][x]
        return val

def floyd_warshall(n, edges):
    dist = [[0 if i == j else float("inf") for i in range(n)] for j in range(n)]
    pred = [[None] * n for _ in range(n)]

    for u, v, d in edges:
        dist[u][v] = d
        pred[u][v] = u

    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
                    pred[i][j] = pred[k][j]
    # Sanity Check
    # for u, v, d in edges:
    #  if dist[u] + d < dist[v]:
    #   return None

    return dist, pred

def dijkstra(graph, start ,n):
    dist, parents = [float("inf")] * n, [-1] * n
    dist[start] = 0
    queue = [(0, start)]
    while queue:
        path_len, v = heappop(queue)
        if path_len == dist[v]:
            for w, edge_len in graph[v]:
                if edge_len + path_len < dist[w]:
                    dist[w], parents[w] = edge_len + path_len, v
                    heappush(queue, (edge_len + path_len, w))
    return dist, parents

def toposort(graph):
    res, found = [], [0] * len(graph)
    stack = list(range(len(graph)))
    while stack:
        node = stack.pop()
        if node < 0:
            res.append(~node)
        elif not found[node]:
            found[node] = 1
            stack.append(~node)
            stack += graph[node]
    for node in res:
        if any(found[nei] for nei in graph[node]):
            return None
        found[node] = 0
    return res[::-1]
```

```python
def kahn(graph):
    n = len(graph)
    indeg, idx = [0] * n, [0] * n
    for i in range(n):
        for e in graph[i]:
            indeg[e] += 1
    q, res = [], []
    for i in range(n):
        if indeg[i] == 0:
            q.append(i)
    nr = 0
    while q:
        res.append(q.pop())
        idx[res[-1]], nr = nr, nr + 1
        for e in graph[res[-1]]:
            indeg[e] -= 1
            if indeg[e] == 0:
                q.append(e)
    return res, idx, nr == n

def dfs(graph, start=0):
    n = len(graph)
    dp = [0] * n
    visited, finished = [False] * n, [False] * n
    stack = [start]
    while stack:
        start = stack[-1]
        if not visited[start]:
            visited[start] = True
            for child in graph[start]:
                if not visited[child]:
                    stack.append(child)
        else:
            stack.pop()
            dp[start] += 1
            for child in graph[start]:
                if finished[child]:
                    dp[start] += dp[child]
            finished[start] = True
    return visited, dp

def rec(cur,color):
    # If asking for SCC, rather than d, use the reversed graph
    # Also the traversel should be in reverse of topological order
    visited = [],ans = [],d = {} # remove this
    visited[cur] = True
    ans[cur] = color
    for i in d[cur]:
        if visited[i]:
            continue
        ans1 = (yield rec(i,color))
    yield -1

def euler_path(d):
    start = [1]
    ans = []
    while start:
        cur = start[-1]
        if len(d[cur])==0:
            ans.append(cur)
            start.pop()
            continue
        k1 = d[cur].pop()
        d[k1].remove(cur)
        start.append(k1)
    return ans

INF = float("inf")
class Dinic:
    def __init__(self, n):
        self.lvl = [0] * n
        self.ptr = [0] * n
        self.q = [0] * n
        self.adj = [[] for _ in range(n)]
    def add_edge(self, a, b, c, rcap=0):
        self.adj[a].append([b, len(self.adj[b]), c, 0])
        self.adj[b].append([a, len(self.adj[a]) - 1, rcap, 0])
    def dfs(self, v, t, f):
        if v == t or not f:
            return f
        for i in range(self.ptr[v], len(self.adj[v])):
```

```
            e = self.adj[v][i]
            if self.lvl[e[0]] == self.lvl[v] + 1:
                p = self.dfs(e[0], t, min(f, e[2] - e[3]))
                if p:
                    self.adj[v][i][3] += p
                    self.adj[e[0]][e[1]][3] -= p
                    return p
            self.ptr[v] += 1
        return 0
    def calc(self, s, t):
        flow, self.q[0] = 0, s
        for l in range(31):  # l = 30 maybe faster for random data
            while True:
                self.lvl, self.ptr = [0] * len(self.q), [0] * len(self.q)
                qi, qe, self.lvl[s] = 0, 1, 1
                while qi < qe and not self.lvl[t]:
                    v = self.q[qi]
                    qi += 1
                    for e in self.adj[v]:
                        if not self.lvl[e[0]] and (e[2] - e[3]) >> (30 - l):
                            self.q[qe] = e[0]
                            qe += 1
                            self.lvl[e[0]] = self.lvl[v] + 1
                p = self.dfs(s, t, INF)
                while p:
                    flow += p
                    p = self.dfs(s, t, INF)
                if not self.lvl[t]:
                    break
        return flow

def find_SCC(graph):
    SCC, S, P = [], [], []
    depth = [0] * len(graph)
    stack = list(range(len(graph)))
    while stack:
        node = stack.pop()
        if node < 0:
            d = depth[~node] - 1
            if P[-1] > d:
                SCC.append(S[d:])
                del S[d:], P[-1]
                for node in SCC[-1]:
                    depth[node] = -1
        elif depth[node] > 0:
            while P[-1] > depth[node]:
                P.pop()
        elif depth[node] == 0:
            S.append(node)
            P.append(len(S))
            depth[node] = len(S)
            stack.append(~node)
            stack += graph[node]
    return SCC[::-1]

class TwoSat:
    def __init__(self, n):
        self.n = n
        self.graph = [[] for _ in range(2 * n)]
    def _imply(self, x, y):
        self.graph[x].append(y if y >= 0 else 2 * self.n + y)
    def either(self, x, y):
        # either x or y must be True
        self._imply(~x, y)
        self._imply(~y, x)
    def set(self, x):
        # x must be True
        self._imply(~x, x)
    def solve(self):
        SCC = find_SCC(self.graph)
        order = [0] * (2 * self.n)
        for i, comp in enumerate(SCC):
            for x in comp:
                order[x] = i
        for i in range(self.n):
            if order[i] == order[~i]:
                return False, None
        return True, [+(order[i] > order[~i]) for i in range(self.n)]

class DisjointSetUnion:
    def __init__(self, n):
        self.parent = list(range(n))
```

```
        self.size = [1] * n
        self.num_sets = n

    def find(self, a):
        acopy = a
        while a != self.parent[a]:
            a = self.parent[a]
        while acopy != a:
            self.parent[acopy], acopy = a, self.parent[acopy]
        return a

    def union(self, a, b):
        a, b = self.find(a), self.find(b)
        if a != b:
            if self.size[a] < self.size[b]:
                a, b = b, a

            self.num_sets -= 1
            self.parent[b] = a
            self.size[a] += self.size[b]

    def set_size(self, a):
        return self.size[self.find(a)]

    def __len__(self):
        return self.num_sets


class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))

    def find(self, a):
        acopy = a
        while a != self.parent[a]:
            a = self.parent[a]
        while acopy != a:
            self.parent[acopy], acopy = a, self.parent[acopy]
        return a

    def union(self, a, b):
        self.parent[self.find(b)] = self.find(a)
```