

```

import sys,math,cmath,random,os,
psutil
from heapq import heappush,heappop
from bisect import bisect_right,
    bisect_left
from collections import Counter,
    deque, defaultdict
from itertools import permutations,
    combinations
from io import BytesIO, IOBase
from decimal import Decimal,
    getcontext
BUFSIZE = 8192
class FastIO(IOBase):
    newlines = 0
    def __init__(self, file):
        self._file = file
        self._fd = file.fileno()
        self.buffer = BytesIO()
        self.writable = "x" in file.mode or "r" not in file.mode
        self.write = self.buffer.write if self.writable else None
    def read(self):
        while True:
            b = os.read(self._fd, max(
                os.fstat(self._fd).st_size, BUFSIZE))
            if not b:
                break
            ptr = self.buffer.tell()
            self.buffer.seek(0, 2),
            self.buffer.write(b),
            self.buffer.seek(ptr)
        self.newlines = 0
        return self.buffer.read()
    def readline(self):
        while self.newlines == 0:
            b = os.read(self._fd, max(
                os.fstat(self._fd).st_size, BUFSIZE))
            self.newlines = b.count(b"\n") + (not b)
            ptr = self.buffer.tell()
            self.buffer.seek(0, 2),
            self.buffer.write(b),
            self.buffer.seek(ptr)
        self.newlines -= 1
        return self.buffer.readline()
    def flush(self):
        if self.writable:
            os.write(self._fd, self.
                buffer.getvalue())
            self.buffer.truncate(0),
            self.buffer.seek(0)
class IOWrapper(IOBase):
    def __init__(self, file):
        self.buffer = FastIO(file)

```

```

self.flush = self.buffer.flush
self.writable = self.buffer.
    writable
self.write = lambda s: self.
    buffer.write(s.encode("ascii"))
self.read = lambda: self.
    buffer.read().decode("ascii")
self.readline = lambda: self.
    buffer.readline().decode("ascii")
sys.stdin, sys.stdout = IOWrapper(
    sys.stdin), IOWrapper(sys.stdout)
MOD = 10**9 + 7; RANDOM = random.
    randrange(1, 2**62)
def gcd(a,b):while b:a,b = b,a%b;
    return a
LII = lambda : list(map(int, sys.
    stdin.readline().split()))
SI = lambda : sys.stdin.readline().
    strip()

def miller_is_prime(n):
    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 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
MAX_N = 10**6+1
Lmini = sieve_unique(MAX_N)
def prime_factors(k,typ=0):
    if typ==0:ans = Counter()
    elif typ==1:ans = set()
    else:ans = []
    while k!=1:
        if typ==0:ans[Lmini[k]] += 1
        elif typ==1:ans.add(Lmini[k])
        else:ans.append(Lmini[k])
        k // Lmini[k]

```

```

return ans
def all_factors(x):
    L = list(prime_factors(x).items())
    st = [1]
    for i in range(len(L)):
        for j in range(len(st)):
            st[-1] *= L[i][0]
        for l in range(L[i][1]):
            st.append(st[j]*k)
            k *= L[i][0]
    return st

def pollard_rho(n):
    if n & 1 == 0:
        return 2
    if n % 3 == 0:
        return 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):
            prev = p
            p = (p * p) % n
            if p == 1:
                return gcd(prev - 1, n)
            if p == n - 1:
                break
        else:
            for i in range(2, n):
                x, y = i, (i * i + 1) %
                    n
                f = gcd(abs(x - y), n)
                while f == 1:
                    x, y = (x * x + 1) %
                        n, (y * y + 1) % n
                    y = (y * y + 1) % n
                    f = gcd(abs(x - y), n)
                if f != n:
                    return f
    return n
def extended_gcd(a, b):
    # returns gcd(a, b), s, r s.t. a
    # * s + b * r == gcd(a, b)
    s, old_s = 0, 1
    r, old_r = b, a
    while r:
        q = old_r // r
        old_r, r = r, old_r - q * r
        old_s, s = s, old_s - q * s
    return old_r, old_s, (old_r -
        old_s * a) // b if b else 0

```

```

def composite_crt(b, m):
    # returns x s.t. x = b[i] (mod m[i]) for all i
    x, m_prod = 0, 1
    for bi, mi in zip(b, m):
        g, s, _ = extended_gcd(m_prod, mi)
        if ((bi - x) % mi) % g:
            return None
        x += m_prod * (s * ((bi - x) % mi)) // g
        m_prod = (m_prod * mi) // gcd(m_prod, mi)
    return x % m_prod

def phi(n):
    ph = [i if i & 1 else i // 2 for i in range(n + 1)]
    for i in range(3, n+1, 2):
        if ph[i]==i:
            for j in range(i, n+1, i):
                ph[j] = (ph[j]//i)*(i-1)
    return ph

```

```

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 derangements(self, n):
        return int(self.fac(n)/math.e + 0.5)
    def stirling_2(self, n, k):
        # Number of ways to partition n elements into k non-empty subsets
        return sum((((-1)**(k - j)) * math.comb(k, j) * (j**n) for j in range(k + 1))) // math.factorial(k)
    def partition(self, n, k):
        # Ways to partition n into k or fewer parts of size 1 or greater
        if n<0 or (k<1 and n>0):return 0
        if n==0:return 1
        return 1 if n==1 else self.partition(n, k-1)+self.partition(n-k, k)
    def lucas_nCk(self, n, k):

```

```

        # Lucas's theorem for finding ((nCk)%p) in log p(n)
        ans = 1;p = self.mod
        while n or k:
            x = n%p;y = k%p;ans *= self.combi(x,y)
            ans %= p;n //= p;k //= p
        return ans

    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

```

```

## here onwards, it is for intersection only, even the point class is different
class Point:
    def __init__(self, x, y):
        self.x = x;self.y = y
    def onSegment(p, q, r):
        return ((q.x <= max(p.x, r.x)) and (q.x >= min(p.x, r.x)) and (q.y <= max(p.y, r.y)) and (q.y >= min(p.y, r.y)))
    def orientation(p, q, r):
        # to find the orientation of an ordered triplet (p,q,r)
        # 0:Collinear points,1:Clockwise points,2:Counterclockwise
        val = ((q.y - p.y) * (r.x - q.x)) - ((q.x - p.x) * (r.y - q.y))
        if (val > 0):return 1
        elif (val < 0):return 2
        else:return 0
    def doIntersect(p1, q1, p2, q2):
        o1 = orientation(p1, q1, p2);o2 =
        orientation(p1, q1, q2)
        o3 = orientation(p2, q2, p1);o4 =
        orientation(p2, q2, q1)
        if ((o1 != o2) and (o3 != o4)):
            return True
        if ((o1 == 0) and onSegment(p1, p2, q1)):return True
        if ((o2 == 0) and onSegment(p1, q2, q1)):return True
        if ((o3 == 0) and onSegment(p2, q2, p1)):

```

```

            p1, q2)):return True
            if ((o4 == 0) and onSegment(p2, q1, q2)):return True
            return False
        ## lines
        # 2d line: ax + by + c = 0 is (a, b, c)
        # ax + by + c = 0 ((a, b, c), f),
        # gy + hz + i = 0 (g, h, i))
        def get_2dline(p1, p2):
            if p1 == p2:return (0, 0, 0)
            p1, p2 = min(p1, p2), max(p1, p2)
            a, b, c = p2[1] - p1[1], p1[1] - p2[0] -
            p1[0] * p2[1]
            g = gcd(gcd(a, b), c)
            return (a // g, b // g, c // g)
        dist = lambda p1, p2: sum((a - b) *
        (a - b) for a, b in zip(p1, p2)) **
        0.5
        get_line = lambda p1, p2: map(
            get_2dline, combinations(p1, 2),
            combinations(p2, 2))
        is_parallel = lambda l1, l2: l1[0] *
        l2[1] == l2[0] * l1[1]
        is_same = lambda l1, l2: is_parallel(l1, l2) and (l1[1] * l2[2] == l2[1] * l1[2])
        collinear = lambda p1, p2, p3:
            is_same(get_2dline(p1, p2),
            get_2dline(p2, p3))
        intersect = (lambda l1, l2: None if
            is_parallel(l1, l2) else (
                (l2[1] * l1[2] - l1[1] * l2[2]) /
                (l2[0] * l1[1] - l1[0] * l2[1]),
                (l1[0] * l2[2] - l1[2] * l2[0]) /
                (l2[0] * l1[1] - l1[0] * l2[1]),
                ))
        rotate = lambda p, theta, origin=(0, 0):
            origin[0] + (p[0] - origin[0]) *
            math.cos(theta) - (p[1] -
            origin[1]) * math.sin(theta),
            origin[1] + (p[0] - origin[0]) *
            math.sin(theta) + (p[1] -
            origin[1]) * math.cos(theta),
        ## polygons
        dist = lambda p1, p2: sum((a - b) *
        (a - b) for a, b in zip(p1, p2)) **
        0.5
        perimeter = lambda p: sum(dist(i, j)
        for i, j in zip(p, p[1:] + p[:1]))

```

```

area = lambda *p: abs(sum(i[0] * j
    [1] - j[0] * i[1] for i, j in zip(
        p, p[1:] + p[:1]))) / 2
is_in_circle = lambda p, c, r: sum(i
    * i - j * j for i, j in zip(p, c))
    ) < r * r
incircle_radius = lambda a, b, c:
    area(a, b, c) / (perimeter(a, b, c
    ) / 2)
circumcircle_radius = lambda a, b, c:
    : (dist(a, b) * dist(b, c) * dist(
        c, a)) / (4 * area(a, b, c))
##
```

```

class XorBasis:
    def __init__(self, A=[]):
        self.reduced_base = {}
        self.how = {}
        self.base = []
        for i in A:
            self.add(i)
    def max(self):
        base = self.base; reduced_base
        = self.reduced_base
        how = self.how
        x = 0; tmp = 0
        for j in sorted(reduced_base,
            reverse=True):
            if not x & (1 << j):
                x ^= reduced_base[j]
                tmp ^= how[j]
        # xor of these elements return
        # the max element
        return [base[j] for j in range(
            len(base)) if tmp & (1 << j
            )]
    def make(self, k):
        # returns True if can make x
        # with these elements
        reduced_base = self.
        reduced_base
        for j in range(k.bit_length()
            -1, -1, -1):
            if ((1 << j) & k):
                if j in reduced_base:
                    k ^= reduced_base[j]
                else:
                    return False
        return k==0
    def add(self, a):
        # adds a to the basis
        reduced_base = self.
        reduced_base
        how = self.how; base = self.
        base
        tmp = 0
        while a:
            b = a.bit_length() - 1
            if b in reduced_base:
```

```

a ^= reduced_base[b]
tmp ^= how[b]
else:
    reduced_base[b] = a
    how[b] = tmp | (1 << len
        (base))
    base.append(a)
break
def s_lr(l,r,pow=1):
    if l>r: return 0
    if pow==1: return (r*(r+1)//2) -
        ((l-1)*l//2)
    elif pow==2: return (r*(r+1)*(2*r
        +1)//6) - ((l-1)*l*(2*l-1)//6)
    elif pow==3: return (r*(r+1)//2)
        **2 - ((l-1)*l//2)**2
def gauss(A,mod=MOD):
    m,n = len(A),len(A[0])-1
    rank = 0; L = [-1]*n
    for col in range(n):
        for row in range(rank,m):
            if A[row][col]:
                A[rank],A[row] = A[row],
                A[rank]
                break
            else:
                continue
            k = pow(A[rank][col],-1,mod)
            for j in range(col,n+1):
                A[rank][j] = A[rank][j]*k%
                    mod
        for row in range(m):
            if row!=rank and A[row][col]:
                factor = A[row][col]
                for j in range(col,n+1):
                    A[row][j] -= factor*A
                        [rank][j]
                    A[row][j] %= mod
        L[col] = rank
        rank += 1
    for row in range(rank, m):
        if A[row][n]:
            return None
    return [A[L[i]][n] if L[i]!=-1
        else 0 for i in range(len(L))]
class DiophantineEquations:
    """
        used for solving equations of
        the form a*x + b*y = c,
        solnll 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 solnll(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

```

```

class PersistentDSU:
    def __init__(self, n):
        self.parent = list(range(n))
        self.size = [1] * n
        self.time = [float('inf')] * n
    def find(self, node, version):
        # returns root at given
        # version
        while not (self.parent[node] == node or self.time[node] >
                    version):
            node = self.parent[node]
        return node

```

```

def union(self, a, b, time):
    # merges a and b
    a = self.find(a, time)
    b = self.find(b, time)
    if a == b:
        return False
    if self.size[a] > self.size[b]:
        a, b = b, a
    self.parent[a] = b
    self.time[a] = time
    self.size[b] += self.size[a]
    return True
def isdisconnected(self, a, b, time):
    return self.find(a, time) != self
        .find(b, time)

```

```

class Node:
    def __init__(self, left=None, right
                 =None, parent=0, size=1):
        self.left = left
        self.right = right
        self.parent = parent
        self.size = size

```

```

class PersistentDSU:
    def __init__(self, n):
        self.n = n
        self.versions = [self._build
                        (0, n)]
    def _build(self, l, r):
        if r-l==1:
            return Node(parent=l, size
                        =1)
        mid=(l+r)//2
        return Node(self._build(l, mid)
                   ,self._build(mid, r))
    def _get(self, node, l, r, idx):

```

```

        if r-l==1:
            return node.parent, node.
                size
        mid=(l+r)//2
        if idx<mid:
            return self._get(node.left,
                             l, mid, idx)
        return self._get(node.right,
                         mid, r, idx)
    def _update(self, node, l, r, idx,
               parent=None, size=None):
        if r-l==1:
            return Node(parent if
                        parent is not None else
                        node.parent, size if size
                        is not None else node.
                        size)
        mid=(l+r)//2
        if idx<mid:
            return Node(self._update(
                            node.left, l, mid, idx,
                            parent, size), node.right)
        return Node(node.left, self.
                   _update(node.right, mid, r, idx
                          ,parent, size))
    def find(self, root, node):
        parent, _ = self._get(root, 0,
                             self.n, node)
        return node if parent==node
        else self.find(root, parent)
    def union(self, version, u, v):
        root=self.versions[version]
        u=self.find(root, u)
        v=self.find(root, v)
        if u==v:
            self.versions.append(root)
            return len(self.versions)-1
        size_u=self._get(root, 0, self
                         .n, u)
        size_v=self._get(root, 0, self
                         .n, v)
        if size_u>size_v:
            u, v=v, u
        root=self._update(root, 0, self.
                           n, u, parent=v)
        root=self._update(root, 0, self.
                           n, v, size=size_u+size_v)
        self.versions.append(root)
        return len(self.versions)-1

```

## Fenwick Tree

```

class BIT:
    #Faster than segment tree so use
    #if possible
    def __init__(self, x):
        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"""
        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

```

## 2-D Fenwick Tree

```

class BIT2D:
    def __init__(self, arr):
        self.n = len(arr)
        self.m = len(arr[0]) if self.n
        > 0 else 0
        # self.bit = [row[:] for row
        # in arr]
        self.bit = arr # assuming that
        # arr is not used after this
        for i in range(self.n):
            for j in range(self.m):
                ni = i | (i + 1)
                if ni < self.n:
                    self.bit[ni][j] +=
                        self.bit[i][j]
        for i in range(self.n):
            for j in range(self.m):
                nj = j | (j + 1)
                if nj < self.m:
                    self.bit[i][nj] +=
                        self.bit[i][j]
    def add(self, x, y, delta):
        # 0-based in log n * log m
        i = x
        while i < self.n:
            j = y

```

```

        while j < self.m:
            self.bit[i][j] += delta
            j |= j + 1
            i |= i + 1
    def sum(self, x, y):
        # sum from 0,0 to x,y
        # inclusive in log n * log m
        if not (0<=x<self.n) or not
            (0<=y<self.m):
            return 0
        res = 0
        i = x
        while i >= 0:
            j = y
            while j >= 0:
                res += self.bit[i][j]
                j = (j & (j + 1)) - 1
            i = (i & (i + 1)) - 1
        return res
    def query(self, x1, y1, x2, y2):
        # sum of L[x1:x2+1][y1:y2+1]
        return (self.sum(x2, y2)-self.
            sum(x1-1, y2)-self.sum(x2, y1
            -1)+(self.sum(x1-1, y1-1)))

```

## Bucket Sorted List

```

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]
                 = []):
        self.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 add(self, x): return self.
        insert(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 bisect_left(self, x): return
        self.index(x)
    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 bisect_right(self, x): return
        self.index_right(x)
    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
    return ltk if abs(k-ltk)<=abs(k-gtk) else gtk

```

## Lazy Segment Tree

```

class SegmentTree:
    @staticmethod
    def func(a, b):
        return a+b
    def __init__(self, data, default
    =0, mode='s'):
        self.mode = mode
        self._default = default
        self.n = len(data)
        self.size = 1 << (self.n - 1).
            bit_length()
        self.tree = [default] * (2 *
            self.size)
        self._size = [0] * (2 * self.
            size)
        self._size[self.size:] = [1] *
            self.size
        for i in range(self.size - 1,
            0, -1):
            self._size[i] = self._size[
                i << 1] + self._size[i <<
                1 | 1]
        self.lazy_add = 0 if self.mode
            == 's' else 0
        self.lazy_set = None
        self.lazy_add = [0] * self.
            size
        self.lazy_set = [None] * self.
            size
        for i in range(self.n):
            self.tree[self.size + i] =
                data[i]
        for i in range(self.size - 1,
            0, -1):
            self.tree[i] = self.func(
                self.tree[i << 1], self.
                tree[i << 1 | 1])
    def _apply_set(self, pos, value):
        if self.mode == 's':
            self.tree[pos] = value *

```

```

                self._size[pos]
        else:
            self.tree[pos] = value
            if pos < self.size:
                self.lazy_set[pos] = value
                self.lazy_add[pos] = 0
    def _apply_add(self, pos, value):
        if self.mode == 's':
            self.tree[pos] += value *
                self._size[pos]
        else:
            self.tree[pos] += value
            if pos < self.size:
                if self.lazy_set[pos] is
                    not None:
                    self.lazy_set[pos] +=
                        value
                else:
                    self.lazy_add[pos] +=
                        value
    def _build(self, pos):
        while pos > 1:
            pos >>= 1
            self.tree[pos] = self.func(
                self.tree[pos << 1], self.
                tree[pos << 1 | 1])
        if self.lazy_set[pos] is
            not None:
            if self.mode == 's':
                self.tree[pos] = self.
                    lazy_set[pos] *
                    self._size[pos]
            else:
                self.tree[pos] = self.
                    lazy_set[pos]
        if self.lazy_add[pos] != 0:
            if self.mode == 's':
                self.tree[pos] +=
                    self.lazy_add[pos] *
                    self._size[pos]
            else:
                self.tree[pos] +=
                    self.lazy_add[pos]
    def _push(self, pos):
        for shift in range(self.size.
            bit_length() - 1, 0, -1):
            i = pos >> shift
            set_val = self.lazy_set[i]
            if set_val is not None:
                self._apply_set(i << 1,
                    set_val)
                self._apply_set(i << 1 |
                    1, set_val)
            self.lazy_set[i] = None
            add_val = self.lazy_add[i]
            if add_val != 0:
                self._apply_add(i << 1,
                    add_val)
                self._apply_add(i << 1 |

```

```

                    1, add_val)
                self.lazy_add[i] = 0
    def range_update(self, left,
        right, value, flag=True):
        # Range Update in [L,R] if
        # flag, then add
        if flag:
            l = left + self.size
            r = right + self.size
            l0, r0 = l, r
            self._push(l0)
            self._push(r0)
            while l <= r:
                if l & 1: self.
                    _apply_add(l, value);
                l += 1
                if not r & 1: self.
                    _apply_add(r, value);
                r -= 1
            l >>= 1; r >>= 1
            self._build(l0)
            self._build(r0)
        else:
            l = left + self.size
            r = right + self.size
            l0, r0 = l, r
            self._push(l0)
            self._push(r0)
            while l <= r:
                if l & 1: self.
                    _apply_set(l, value);
                l += 1
                if not r & 1: self.
                    _apply_set(r, value);
                r -= 1
            l >>= 1; r >>= 1
            self._build(l0)
            self._build(r0)
    def range_query(self, left, right
    ):
        # Range Query in [L,R]
        l = left + self.size
        r = right + self.size
        self._push(l)
        self._push(r)
        res = self._default
        while l <= r:
            if l & 1: res = self.func(
                res, self.tree[l]); l +=
                1
            if not r & 1: res = self.
                func(res, self.tree[r]);
                r -= 1
            l >>= 1; r >>= 1
        return res

```

## Lazy Segment Tree 2

```

class SegmentTree:

```

```

def __init__(self, data, default=0, func=max):
    # don't forget to change func here
    # default is the value given to it by 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

```

### Trie

```

class Trie:
    def __init__(self):
        self.root = {}
    def add(self, word):
        current_dict = self.root
        for letter in word:
            current_dict = current_dict.setdefault(letter, {})
        current_dict[0] = True

```

### Convex Hull

```

def f(line, x):
    return line[0] * x + line[1]
class LiChao:
    def __init__(self, lo=0, hi=10**9):
        self.lo = lo
        self.hi = hi
        self.m = (lo + hi) // 2
        self.line = None
        self.left = None
        self.right = None
    def add_line(self, new_line):
        l, r, m = self.lo, self.hi,
        self.m
        if self.line is None:
            self.line = new_line

```

```

        return
if f(new_line, m) > f(self.line, m):
    self.line, new_line = new_line, self.line
if l == r:
    return
if f(new_line, l) > f(self.line, l):
    if self.left is None:
        self.left = LiChao(l, m)
        self.left.add_line(new_line)
    elif f(new_line, r) > f(self.line, r):
        if self.right is None:
            self.right = LiChao(m + 1, r)
            self.right.add_line(new_line)
def query(self, x):
    res = f(self.line, x) if self.line is not None else -10**18
    if self.lo == self.hi:
        return res
    if x <= self.m and self.left is not None:
        res = max(res, self.left.query(x))
    elif x > self.m and self.right is not None:
        res = max(res, self.right.query(x))
    return res

```

### Mono Deque

```

from typing import Callable, TypeVar, Generic, List
T = TypeVar('T')
class MonoStack(Generic[T]):
    def __init__(self, op: Callable[[T, T], T], e: Callable[[], T]):
        self.s: List[T] = []
        self.sMono: List[T] = []
        self.op = op
        self.e = e
    def push(self, x: T):
        self.s.append(x)
        if not self.sMono:
            self.sMono.append(x)
        else:
            self.sMono.append(self.op(
                self.sMono[-1], x))
    def pop(self) -> T:
        if not self.s:
            return self.e()
        self.sMono.pop()
        return self.s.pop()

```

```

def get(self) -> T:
    return self.sMono[-1] if self.
        sMono else self.e()
def empty(self) -> bool:
    return not self.s
class MonoDeque(Generic[T]):
    def __init__(self, op: Callable[[T, T], T] = lambda a,b: max(a,b),
                 e: Callable[[], T] = lambda : -float('inf')):
        # e is the unit value in form
        # of lambda : 0
        # The Function op must be
        # associative, f(a,b)==f(b,a)
        self.op = op
        self.e = e
        self.front = MonoStack(op, e)
        self.back = MonoStack(op, e)
    def push_back(self, x: T):
        self.back.push(x)
    def pop_front(self):
        if self.front.empty():
            while not self.back.empty():
                self.front.push(self.
                    back.pop())
            self.front.pop()
        def get(self) -> T:
            # returns the function op of
            # the elements present in the
            # deque
            return self.op(self.front.get(),
                           self.back.get())

```

## Persistent Segment Tree

```

class PersistentSegmentTree:
    class Node:
        def __init__(self, value=0,
                     left=None, right=None):
            self.value = value
            self.left = left
            self.right = right
    @staticmethod
    def func(a, b):
        return a+b
    def __init__(self, data):
        self.n = len(data)
        self.versions = []
        self.versions.append(self.
            build(data))
    def _build(self, data):
        stack = [(0, self.n - 1, False)]
        nodes = {}
        while stack:
            left, right, visited =
                stack.pop()
            if left == right:

```

```

                nodes[(left, right)] =
                    self.Node(data[left])
            else:
                if visited:
                    mid = (left + right) //
                        2
                    left_child = nodes[(left, mid)]
                    right_child = nodes[(mid + 1, right)]
                    nodes[(left, right)] =
                        self.Node(self.
                            func(left_child.
                                value, right_child.
                                value), left_child,
                                right_child)
                else:
                    stack.append((left,
                                  right, True))
                    mid = (left + right) //
                        2
                    stack.append((mid +
                                  1, right, False))
                    stack.append((left,
                                  mid, False))
            return nodes[(0, self.n - 1)]
    def update(self, version, pos,
               value):
        old_root = self.versions[
            version]
        stack, path = [(old_root, 0,
                       self.n - 1)], []
        while stack:
            node, left, right = stack.
                pop()
            path.append((node, left,
                         right))
            if left == right:
                break
            mid = (left + right) // 2
            if pos <= mid:
                stack.append((node.left,
                              left, mid))
            else:
                stack.append((node.right,
                              mid + 1, right))
        new_nodes = {}
        for node, left, right in
            reversed(path):
            if left == right:
                k = value
                # k = self.func(node.
                #     value, value) # if i
                # want to update
                new_nodes[(left, right)] =
                    self.Node(k)
            else:
                mid = (left + right) //
                    2
                left_child = new_nodes.
                    get((left, mid), node.
                        left)
                right_child = new_nodes.
                    get((mid + 1, right),
                        node.right)
                new_nodes[(left, right)] =
                    self.Node(self.func(
                        left_child.value,
                        right_child.value),
                        left_child,
                        right_child)
        return new_nodes[(0, self.n -
                          1)]
    def create_version(self, version,
                      pos, value):
        new_root = self.update(version,
                               pos, value)
        self.versions.append(new_root)
        return len(self.versions) - 1
    def query(self, version, ql, qr):
        node, left, right = self.
            versions[version], 0, self.n -
            1
        stack = [(node, left, right)]
        result = 0 # change this
        depending on the problem
        while stack:
            node, left, right = stack.
                pop()
            if ql > right or qr < left:
                continue
            if ql <= left and right <=
                qr:
                result = self.func(
                    result, node.value)
            else:
                mid = (left + right) //
                    2
                stack.append((node.left,
                              left, mid))
                stack.append((node.right,
                              mid + 1, right))
        return result

```

## Suffix Automaton

```

class State:
    def __init__(self):
        self.next = {}
        self.link = -1
        self.len = 0
        self.first_pos = -1
        self.occurrence = 0
    class SuffixAutomaton:
        def __init__(self, s):
            self.s = s
            self.states = [State()]
            self.size = 1

```

```

self.last = 0
for ch in s:
    self.add(ch)
self._prepare_occurrences() # comment out if taking time
self._count_substrings()
def add(self, ch):
    p = self.last
    cur = self.size
    self.states.append(State())
    self.size += 1
    self.states[cur].len = self.states[p].len + 1
    self.states[cur].first_pos = self.states[cur].len - 1
    self.states[cur].occurrence = 1
    while p != -1 and ch not in self.states[p].next:
        self.states[p].next[ch] = cur
        p = self.states[p].link
    if p == -1:
        self.states[cur].link = 0
    else:
        q = self.states[p].next[ch]
        if self.states[p].len + 1 == self.states[q].len:
            self.states[cur].link = q
        else:
            clone = self.size
            self.states.append(State())
            self.size += 1
            self.states[clone].len = self.states[p].len + 1
            self.states[clone].next = self.states[q].next.copy()
            self.states[clone].link = self.states[q].link
            self.states[clone].first_pos = self.states[q].first_pos
            while p != -1 and self.states[p].next[ch] == q:
                self.states[p].next[ch] = clone
                p = self.states[p].link
            self.states[q].link = self.states[cur].link = clone
            self.last = cur
def _prepare_occurrences(self):
    order = sorted(range(self.size), key=lambda x: -self.

```

```

states[x].len)
for i in order:
    if self.states[i].link != -1:
        self.states[self.states[i].link].occurrence += self.states[i].occurrence
def _count_substrings(self):
    self.dp = [0] * self.size
    for i in range(self.size):
        self.dp[i] = 1
    order = sorted(range(self.size), key=lambda x: self.states[x].len)
    for u in reversed(order):
        for v in self.states[u].next.values():
            self.dp[u] += self.dp[v]
def count_occurrences(self, s):
    current = 0
    for ch in s:
        if ch not in self.states[current].next:
            return 0
        current = self.states[current].next[ch]
    return self.states[current].occurrence
def count_distinct_substrings(self):
    return sum(self.states[i].len - self.states[self.states[i].link].len for i in range(1, self.size))
def kth_lex_substring(self, k):
    # kth distinct substring
    result = []
    current = 0
    while k:
        for ch in sorted(self.states[current].next):
            next_state = self.states[current].next[ch]
            if self.dp[next_state] < k:
                k -= self.dp[next_state]
            else:
                result.append(ch)
                k -= 1
                current = next_state
                break
    return ''.join(result)
def enumerate_all_substrings(self):
    result = []
    def dfs(state, path):
        for ch in sorted(self.states[state].next):

```

```

next_state = self.states[state].next[ch]
result.append(path + ch)
dfs(next_state, path + ch)
dfs(0, "")
return result
def longest_common_substring(self, t):
    v = 0; l = 0; best = 0; bestpos = 0
    for i in range(len(t)):
        while v and t[i] not in self.states[v].next:
            v = self.states[v].link
            l = self.states[v].len
        if t[i] in self.states[v].next:
            v = self.states[v].next[t[i]]
            l += 1
        if l > best:
            best = l
            bestpos = i
    return t[bestpos - best + 1: bestpos + 1]
def all_occurrences(self, s):
    current = 0
    for ch in s:
        if ch not in self.states[current].next:
            return []
        current = self.states[current].next[ch]
    positions = []
    def collect(state):
        if self.states[state].occurrence:
            pos = self.states[state].first_pos - len(s) + 1
            positions.append(pos)
            for v in self.states[state].next.values():
                collect(v)
    collect(current)
    return sorted(set(positions))
def missing_sub(self):
    visited = set()
    q = deque([(0, "")])
    while q:
        state, path = q.popleft()
        for c in map(chr, range(97, 123)):
            if c not in self.states[state].next:
                return path + c
            next_state = self.states[state].next[c]
            if (next_state, path + c)

```

```

) not in visited:
    visited.add((
        next_state, path +
        c))
    q.append((next_state,
              path + c))
return None

```

## Segment Tree

```

class SegmentTree:
    def __init__(self, data):
        self.n = len(data)
        self.tree = [0] * (self.n<<1)
        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] = max(self.
                tree[i<<1], self.tree[(i
                <<1) + 1])
    def update(self, pos, value):
        pos += self.n; self.tree[pos] =
            value
        while pos > 1:
            pos >= 1
            self.tree[pos] = max(self.
                tree[pos<<1], self.tree[(pos
                <<1) + 1])
    def query(self, left, right):
        # [left, right)
        left += self.n; right += self.n
        max_val = float('-inf')
        while left < right:
            if left&1:
                max_val = max(max_val,
                               self.tree[left])
                left += 1
            if right&1:
                right -= 1
                max_val = max(max_val,
                               self.tree[right])
                right >= 1; right >= 1
        return max_val

```

```

class SparseTable:
    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)]
        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] = max(
        self.table[i][j - 1],
        self.table[i + (1 << (j -
        1))][j - 1])
    i += 1
    j += 1
def query(self, L, R):
    # query from [L,R]
    j = int(math.log2(R - L + 1))
    return max(self.table[L][j],
               self.table[R - (1 << j) +
               1][j])

```

## Others

```

def euler_path(d):
    start = [1]
    ans = []
    while start:
        cur = start[-1]
        if len(d[cur]) == 0:
            ans.append(start.pop())
            continue
        k1 = d[cur].pop()
        d[k1].remove(cur) # if
                           undirected
        start.append(k1)
    return ans
def floyd_warshall(n, edges):
    dist = [[0 if i == j else float(""
        "inf") for i in range(n)] for j
        in range(n)]
    for u, v, d in edges:
        dist[u][v] = d
    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i
                ][j], dist[i][k] + dist
                [k][j])
    return dist
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]

## Advance

```

def centroid_decomposition_tree(
    graph):
    n = len(graph)
    graph = [c[:] for c in graph]
    bfs = [0]
    for node in bfs:
        bfs += graph[node]
        for nei in graph[node]:
            graph[nei].remove(node)
    size = [0] * n
    for node in reversed(bfs):
        size[node] = 1 + sum(size[
            child] for child in graph[
            node])
    decomposition_tree = [[] for _ in
        range(n)]
    def centroid_reroot(u):
        N = size[u]
        while True:
            for v in graph[u]:
                if size[v] > N // 2:
                    size[u] = N - size[v]
                    graph[u].remove(v)
                    graph[v].append(u)
                    u = v; break
            else:
                decomposition_tree[u] =
                    [centroid_reroot(v)
                     for v in graph[u]]
                return u
    decomposition_root =
        centroid_reroot(0)
    return decomposition_tree,
    decomposition_root
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
class AuxiliaryTree:
    def __init__(self, edge, root = 0):
        self.n = len(edge)
        self.order = [-1] * self.n
        self.path = [-1] * (self.n-1)
        self.depth = [0] * self.n
        if self.n == 1: return
        parent = [-1] * self.n
        que = [root]
        t = -1
        while que:
            u = que.pop()
            self.path[t] = parent[u]
            t += 1
            self.order[u] = t
for v in edge[u]:
    if self.order[v] == -1:
        que.append(v)
        parent[v] = u
        self.depth[v] = self.
            depth[u] + 1
self.n -= 1
self.h = self.n.bit_length()
self.data = [0] * (self.n *
    self.h)
self.data[:self.n] = [self.
    order[u] for u in self.path]
for i in range(1, self.h):
    for j in range(self.n -
        (1<<i) + 1):
        self.data[i*self.n + j] =
        min(self.data[(i-1)*
            self.n + j], self.data[
            ((i-1)*self.n + j
            +(1<<(i-1))])
def lca(self, u, v):
    if u == v: return u
    l = self.order[u]
    r = self.order[v]
    if l > r:
        l, r = r, l
    level = (r - l).bit_length() -
        1
    return self.path[min(self.data[
        level*self.n + l], self.
        data[level*self.n + r-(1<<
            level)))]
def dis(self, u, v):
    if u == v: return 0
    l = self.order[u]
    r = self.order[v]
    if l > r:
        l, r = r, l
    level = (r - l).bit_length() -
        1
    p = self.path[min(self.data[
        level*self.n + l], self.data[
        level*self.n + r-(1<<level)
        ])]
    return self.depth[u] + self.
        depth[v] - 2 * self.depth[p]
def make(self, vs):
    k = len(vs)
    vs.sort(key = self.order.
        __getitem__)
    par = dict()
    edge = dict()
    edge[vs[0]] = []
    st = [vs[0]]
    for i in range(k - 1):
        l = self.order[vs[i]]
        r = self.order[vs[i+1]]
        level = (r - l).bit_length()
        () - 1
        w = self.path[min(self.data[
            level*self.n + l], self.
            data[level*self.n + r
            -(1<<level))])
        if w != vs[i]:
            p = st.pop()
            while st and self.depth[st
                [-1]] < self.depth[st
                [-1]]:
                par[p] = st[-1]
                edge[st[-1]].append(p)
                p = st.pop()
            if not st or st[-1] != w
                :
                st.append(w)
                edge[w] = [p]
            else:
                edge[w].append(p)
            par[p] = w
            st.append(vs[i+1])
            edge[vs[i+1]] = []
        for i in range(len(st) - 1):
            edge[st[i]].append(st[i+1])
            par[st[i+1]] = st[i]
            par[st[0]] = -1
        return st[0], edge, par
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
DP-OPT
def Knuths(n, C = lambda x: x):
    # For all a<=b<=c<=d:
    # cost(a,c)+cost(b,d)<=cost(a,d) +
    cost(b,c)
    # opt[l][r-1] <= opt[l][r] <= opt
    [l+1][r]
    dp = [[0]*n for _ in range(n)]
    opt = [[0]*n for _ in range(n)]
    for i in range(n):
        dp[i][i] = 0
        opt[i][i] = i
    for i in range(n-2, -1, -1):

```

```

for j in range(i+1, n):
    cost = C(i, j)
    mini = float('inf')
    L = opt[i][j-1]
    R = min(j-1, opt[i+1][j])
    for k in range(L, R+1):
        val = dp[i][k] + dp[k+1][j] + cost
        if val <= mini:
            mini = val
            opt[i][j] = k
    dp[i][j] = mini
return dp[0][n-1]

def DnC(prev, dp, cost):
    # opt[i][j] <= opt[i][j+1]
    # for i in range(1, K + 1):
    # DnC(dp_prev, dp_cur, cost, n)
    # dp_prev, dp_cur = dp_cur,
    # dp_prev
    def solve(l, r, opt_l, opt_r):
        if l > r:
            return
        mid = (l + r) // 2
        best_k = -1
        best_val = float('inf')
        end = max(0, min(mid - 1, opt_r))
        for k in range(opt_l, end + 1):
            val = prev[k] + cost(k, mid)
            if val < best_val:
                best_val = val
                best_k = k
        dp[mid] = best_val
        solve(l, mid - 1, opt_l, best_k)
        solve(mid + 1, r, best_k, opt_r)
    solve(0, len(dp)-1, 0, len(dp)-1)

```

## Graph-Flatten

```

def dfs(graph):
    starttime = [[0, 0] for i in range(len(graph))]
    time = 0
    stack = [(0, -1, 0)]
    while stack:
        cur, prev, state = stack.pop()
        if state == 0:
            starttime[cur][0] = time
            time += 1
            stack.append((cur, prev, 1))
        for neighbor in graph[cur]:
            if neighbor == prev:
                continue
            stack.append((neighbor, cur, 0))

```

```

        cur, 0))
    elif state == 1:
        starttime[cur][1] = time
    return starttime

2-sat

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 negate(self, x):
        return x + self.n if x < self.n else x - self.n
    def _imply(self, x, y):
        # agar x hogा, toh y hogा
        self.graph[x].append(y)
        self.graph[self.negate(y)].append(self.negate(x))
    def either(self, x, y):
        # koi ek true ho sakta hain ya dono bhi
        self._imply(self.negate(x), y)
        self._imply(self.negate(y), x)
    def set(self, x):
        self._imply(self.negate(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[self.negate(i)]:

```

```

                return False, None
        return True, [+ (order[i] > order[self.negate(i)]) for i in range(self.n)]

def find_bridges(adj):
    # returns all bridges
    bridges = []
    n = len(adj)
    timer = 0
    visited = [False] * n
    tin = [-1] * n
    low = [-1] * n
    for start in range(n):
        if visited[start]:
            continue
        stack = [(start, -1, 0, False)]
        visited[start] = True
        tin[start] = low[start] = timer
        timer += 1
        while stack:
            v, parent, idx, backtrack = stack.pop()
            if backtrack:
                to = adj[v][idx]
                low[v] = min(low[v], low[to])
                if low[to] > tin[v]:
                    bridges.append((v, to))
            continue
        if idx < len(adj[v]):
            to = adj[v][idx]
            stack.append((v, parent, idx + 1, False))
            if to == parent:
                continue
            if visited[to]:
                low[v] = min(low[v], tin[to])
            else:
                visited[to] = True
                tin[to] = low[to] = timer
                timer += 1
                stack.append((v, parent, idx, True))
                stack.append((to, v, 0, False))
    return bridges

def lowlink(edge):
    n = len(edge)
    parent = [-1] * n
    visited = [False] * n
    for s in range(n):
        if not visited[s]:

```

```

que = [s]
while que:
    now = que.pop()
    if visited[now]:
        continue
    visited[now] = True
    for nxt in edge[now]:
        if not visited[nxt]:
            parent[nxt] = now
            que.append(nxt)

order = [-1] * n
low = [-1] * n
is_articulation = [False] * n
articulation = []
bridge = []
def dfs(s):
    idx = 0
    cnt = 0
    que = [~s,s]
    while que:
        now = que.pop()
        if now >= 0:
            order[now] = low[now] = idx
            idx += 1
        for nxt in edge[now]:
            if parent[nxt] == now:
                :
                que.append(~nxt)
                que.append(nxt)
            elif parent[now] != nxt and order[nxt] != -1:
                low[now] = min(low[now], order[nxt])
            else:
                now = ~now
                par = parent[now]
                if par == s: cnt += 1
                if now == s:
                    is_articulation[now] |= (cnt >= 2)
                if is_articulation[now]:
                    articulation.append(now)
    return
    if is_articulation[now]:
        articulation.append(now)
    if now != parent[par]:
        low[par] = min(low[par], low[now])
    is_articulation[par] |= (par != s) and (order[par] <= low[now])
    if order[par] < low[now]:
        :

```

```

bridge.append((par, now))
for i in range(n):
    if parent[i] == -1:
        dfs(i)
return articulation, bridge

def find_2ecc(edges,d):
    # returns a new graph, in which
    # two nodes are connected
    # if and only if they are part of
    # same cycle.
    _,bridges = lowlink(d)
    newd = [[] for i in range(len(d))]
    bridges = set((w(i[0]),w(i[1])))
    for i in bridges)
    for u,v in edges:
        if (w(u),w(v)) not in bridges
            and (w(v),w(u)) not in
            bridges:
            newd[u].append(v)
            newd[v].append(u)
    return newd

```

### String Hashing

```

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

```

```

def LCPArray(L, SA=None):
    # Longest Common prefix of S[i:]
    # and S[i+1:]
    if not SA:SA = SuffArr(L)
    n = len(L);rank = [0] * n
    for i in range(n):rank[SA[i]] = i
    LCP = [0] * (n - 1)
    k = 0
    for i in range(n):

```

```

        l = rank[i]
        if l==n-1:k = 0;continue
        j = SA[l+1]
        while i+k<n and L[i+k] == L[j+k]:
            k += 1
        LCP[l] = k;k -= k > 0
    return LCP
def z_function(S):
    # where Z[i] = 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 - 1]
        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 manacher(s):
    t = '#' + '#' .join(s) + '#'
    n = len(t);L = [0] * n
    c = r = 0;ml = 0;mc = 0
    for i in range(n):
        mirror = 2*c-i
        if i<r:L[i] = min(r-i,L[mirror])
        a = i+L[i]+1
        b = i-L[i]-1
        while a<n and b>=0 and t[a]==t[b]:
            L[i] += 1;a += 1;b -= 1
        if i+L[i]>r:c = i;r = i+L[i]
        if L[i]>ml:ml = L[i];mc = i
    start = (mc-ml)//2
    return s[start:start+ml]
def SuffArray(s):
    # Starting position of ith suffix
    # in lexicographic order
    s += "$"
    n = len(s)
    k = 0
    rank = [ord(c) for c in s]
    tmp = [0] * n
    sa = list(range(n))
    def sort_key(i):
        return (rank[i], rank[i + (1
        << k)] if i + (1 << k) < n
        else -1)
    while True:
        sa.sort(key=sort_key)
        tmp[sa[0]] = 0
        for i in range(1, n):
            tmp[sa[i]] = tmp[sa[i-1]] +
            (sort_key(sa[i-1]) !=

```

```

rank, tmp = rank, rank
k += 1
if (1 << k) >= n:
    break
return sa

```

## FFT

```

class FFT:
    def __init__(self, MOD=998244353,
                 MOD1=469762049):
        FFT.MOD = MOD
        FFT.MOD1 = MOD1
        FFT.MOD2 = pow(MOD, MOD1-2, MOD1)
        FFT.mod_inv = (self.XT_GCD(MOD,
                                     MOD1)[1])%MOD1
        # g = self.
        primitive_root_constexpr()
        g = 3
        ig = pow(g, FFT.MOD - 2, FFT.MOD)
        ig1 = pow(g, FFT.MOD1 - 2, FFT.MOD1)
        FFT.W = [pow(g, (FFT.MOD - 1)
                      >> i, FFT.MOD) for i in
                  range(30)]
        FFT.W1 = [pow(g, (FFT.MOD1 -
                           1) >> i, FFT.MOD1) for i in
                  range(30)]
        FFT.iW = [pow(ig, (FFT.MOD -
                            1) >> i, FFT.MOD) for i in
                  range(30)]
        FFT.iW1 = [pow(ig1, (FFT.MOD1 -
                               1) >> i, FFT.MOD1) for i
                  in range(30)]
    def primitive_root_constexpr(self):
        if FFT.MOD == 998244353:
            return 3
        elif FFT.MOD == 200003:
            return 2
        elif FFT.MOD == 167772161:
            return 3
        elif FFT.MOD == 469762049:
            return 3
        elif FFT.MOD == 754974721:
            return 11
        divs = [0] * 20
        divs[0] = 2
        cnt = 1
        x = (FFT.MOD - 1) // 2
        while x % 2 == 0:
            x /= 2
            i = 3
            while i * i <= x:
                if x % i == 0:
                    divs[cnt] = i
                    cnt += 1

```

```

while x % i == 0:
    x /= i
    i += 2
if x > 1:
    divs[cnt] = x
    cnt += 1
g = 2
while 1:
    ok = True
    for i in range(cnt):
        if pow(g, (FFT.MOD - 1)
               // divs[i], FFT.MOD)
               == 1:
            ok = False
            break
    if ok:
        return g
    g += 1
def fft(self, k, f, f1):
    for l in range(k, 0, -1):
        d = 1 << l - 1
        U = [(1, 1)]
        for i in range(d):
            U.append((U[-1][0] * FFT.W[l] % FFT.MOD,
                      U[-1][1] * FFT.W1[l] % FFT.MOD1))
        for i in range(1 << k - 1):
            for j in range(d):
                s = i * 2 * d + j
                f[s], f[s + d] = (f[s]
                                    + f[s + d]) % FFT.MOD,
                U[j][0] * (f[s] - f[s + d]) % FFT.MOD
                f1[s], f1[s + d] = (f1[s] + f1[s + d])
                % FFT.MOD1, U[j][1] * (f1[s] - f1[s + d]) % FFT.MOD1
def ifft(self, k, f, f1):
    for l in range(1, k + 1):
        d = 1 << l - 1
        for i in range(1 << k - 1):
            u = 1
            u1 = 1
            for j in range(i * 2 * d,
                           (i * 2 + 1) * d):
                f[j+d] *= u
                f[j], f[j + d] = (f[j]
                                   + f[j + d]) % FFT.MOD,
                (f[j] - f[j + d]) % FFT.MOD
            u = u * FFT.iW[l] % FFT.MOD
            f1[j+d] *= u1
            f1[j], f1[j + d] = (f1[j]
                                   + f1[j + d]) % FFT.MOD1, (f1[j]

```

```

- f1[j + d]) % FFT.MOD1
u1 = u1 * FFT.iW1[l] % FFT.MOD1
def XT_GCD(self, a, b):
    if b == 0:
        return a, 1, 0
    g, x1, y1 = self.XT_GCD(b, a % b)
    x = y1
    y = x1 - (a // b) * y1
    return g, x, y
def CRT(self, a, mod1, b, mod2):
    k = (a + (b-a)*self.mod_inv%mod2
         *mod1)%(mod1*mod2)
    return k
def convolve(self, A, B):
    n0 = len(A) + len(B) - 1
    k = (n0).bit_length()
    n = 1 << k
    A += [0] * (n - len(A))
    B += [0] * (n - len(B))
    A1 = A[:]
    B1 = B[:]
    self.fft(k, A, A1)
    self.fft(k, B, B1)
    A = [a * b % FFT.MOD for a, b
         in zip(A, B)]
    A1 = [a * b % FFT.MOD1 for a,
          b in zip(A1, B1)]
    self.ifft(k, A, A1)
    inv = pow(n, FFT.MOD - 2, FFT.MOD)
    inv1 = pow(n, FFT.MOD1 - 2,
               FFT.MOD1)
    del A[n0:]
    for i in range(n0):
        A[i] = self.CRT(A[i]*inv,
                         FFT.MOD, A1[i]*inv1, FFT.MOD1)
    return A

```

## FWHT

```

def fwht(a, invert=False):
    # In-place Fast Walsh Hadamard
    # Transform for XOR convolution.
    # invert: if True, computes
    # inverse transform
    n = len(a); step = 1
    while step < n:
        for i in range(0, n, step*2):
            for j in range(step):
                a[i+j], a[i+j+step] = a[i+j] + a[i+j+step], a[i+j] - a[i+j+step]
            step *= 2
def fwt_or(a, invert=False):
    n = len(a); step = 1
    while step < n:
        for i in range(0, n, step*2):

```

```

for j in range(step):
    u = a[i+j]
    v = a[i+j+step]
    if not invert:
        a[i+j+step] = u+v
    else:
        a[i+j+step] = v-u
    step *= 2
def fwt_and(a, invert=False):
    n = len(a); step = 1
    while step < n:
        for i in range(0, n, step*2):
            for j in range(step):
                u = a[i+j]
                v = a[i+j+step]
                if not invert:
                    a[i+j] = u+v
                else:
                    a[i+j] = u-v
            step *= 2
def convolution(A, B, fwt=fwht):
    # Computes XOR convolution of
    # arrays A and B using FWHT
    # res[i] = summation of A[j]*B[k]
    # such that j xor k = i
    n = 1; x = max(len(A), len(B))
    while n < x: n <<= 1
    n <= 1
    fa = A + [0] * (n - len(A))
    fb = B + [0] * (n - len(B))
    fwt(fa)
    fwt(fb)
    for i in range(n):
        fa[i] *= fb[i]
    fwt(fa, invert=True)
    return fa

```

## Bootstrap

```

from types import GeneratorType
def bootstrap(f, stack=[]):
    def wrappedfunc(*args, **kwargs):
        if stack:
            return f(*args, **kwargs)
        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

```

## Utils

```

getcontext().prec = 50; sys.
    setrecursionlimit(10**6)
sys.set_int_max_str_digits(10**5)
import builtins
globals()['print'] = lambda *args,
    **kwargs: builtins.print(*args,
    flush=True, **kwargs)

```

**Extended GCD** Computes  $x, y$  s.t.  $ax + by = \text{gcd}(a, b)$ .

**CRT** Finds smallest  $a$  satisfying  $a \equiv b_i \pmod{m_i}$ .

$\phi(n)$  Count of integers  $< n$  coprime with  $n$ .

**Suffix Array** Starting indices of suffixes sorted lexicographically.

**Euler Path** Uses every edge exactly once.

**Bridge** Edge whose removal increases components.

**Articulation Point** Vertex whose removal increases components.

**2ECC** Maximal subgraph where every pair lies on a cycle.

Use `get<i>(tuple)` for tuple access. Custom sort: `[](auto& a, auto& b){return get<2>(a)<get<2>(b);}`  
 2D vector init: `vector<vector<int>> dp(n, vector<int>(m, 0));` Prefer `emplace_back` for complex objects. Min-heap: `priority_queue<pi, vector<pi>, greater<pi> pq;`

```

#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.
    hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
using namespace std;
#define int long long
#define fastio ios::sync_with_stdio(
    false); cin.tie(0);
#pragma GCC optimize("O3,unroll-
    loops")
#pragma GCC target("avx2,bmi,bmi2,
    lzcnt,popcnt")
template <typename T>
using ordered_multiset = tree<
    std::pair<T, int>,
    null_type,
    less<std::pair<T, int>>,
    rb_tree_tag,
    tree_order_statistics_node_update
>;
int II(){int a;cin>>a;return a;}
string SI(){string s;cin>>s;return s
    ;}
vi LI(int n){vi a(n);cin>>a;return
    a;}
int32_t main() {
    fastio
    return 0;
}

```

```

template <typename T>
class SortedList {
    ordered_multiset<T> os;
    int uid = 0;
public:
    void insert(T x) {
        os.insert({x, uid++});
    }
    void erase_one(T x) {
        auto it = os.lower_bound({x,
        0});
        if (it != os.end() && it->
            first == x)
            os.erase(it);
    }
    int index(T x) {
        return os.order_of_key({x, 0});
    }
    int count(T x) {
        return os.order_of_key({x,
        INT_MAX}) - os.order_of_key
        ({x, 0});
    }
    operator[](int k) {
        return os.find_by_order(k)->
            first;
    }
};

class SegmentTree {
public:
    int n;
    vector<int> tree;
    vector<int> lazy_add;
    vector<int> lazy_set;
    int NO_ASSIGNMENT = LLONG_MIN;
    SegmentTree(vector<int>& arr) {
        n = arr.size();
        tree.assign(4 * n, 0);
        lazy_add.assign(4 * n, 0);
        lazy_set.assign(4 * n,
        NO_ASSIGNMENT);
    }
}

```

```

    build_tree(1, 0, n - 1, arr); ||
} static int func(int a, int b) {
    return a + b;
}
void build_tree(int node, int
    start, int end, vector<int>&
    arr) {
    if (start == end) {
        tree[node] = arr[start];
    } else {
        int mid = start + (end -
            start) / 2;
        build_tree(2 * node, start,
            mid, arr);
        build_tree(2 * node + 1,
            mid + 1, end, arr);
        tree[node] = func(tree[2 *
            node], tree[2 * node +
            1]);
    }
}
void propagate_lazy(int node, int
    start, int end) {
    int current_range_size = (end -
        start + 1);
    if (lazy_set[node] != NO_ASSIGNMENT) {
        tree[node] = lazy_set[node] *
            current_range_size;
        if (start != end) {
            lazy_set[2 * node] =
                lazy_set[node];
            lazy_set[2 * node + 1] =
                lazy_set[node];
            lazy_add[2 * node] = 0;
            lazy_add[2 * node + 1] =
                0;
        }
        lazy_set[node] =
            NO_ASSIGNMENT;
    }
    if (lazy_add[node] != 0) {
        tree[node] += lazy_add[node] *
            current_range_size;
        if (start != end) {
            lazy_add[2 * node] +=
                lazy_add[node];
            lazy_add[2 * node + 1] +=
                lazy_add[node];
        }
        lazy_add[node] = 0;
    }
}
void update(int node, int start,
    int end, int l, int r, int
    value, bool is_add) {
    propagate_lazy(node, start,
        end);
    if (start > r || end < l) {
        return;
    }
    if (start >= l && end <= r) {
        if (is_add) {
            tree[node] += value * (
                end - start + 1);
            if (start != end) {
                lazy_add[2 * node] +=
                    value;
                lazy_add[2 * node +
                    1] += value;
            }
        } else {
            tree[node] = value * (
                end - start + 1);
            if (start != end) {
                lazy_set[2 * node] =
                    value;
                lazy_set[2 * node +
                    1] = value;
                lazy_add[2 * node] =
                    0;
                lazy_add[2 * node +
                    1] = 0;
            }
        }
        return;
    }
    int mid = start + (end - start
        ) / 2;
    update(2 * node, start, mid, l
        , r, value, is_add);
    update(2 * node + 1, mid + 1,
        end, l, r, value, is_add);
    tree[node] = func(tree[2 *
        node], tree[2 * node + 1]);
}
int query(int node, int start,
    int end, int l, int r) {
    propagate_lazy(node, start,
        end);
    if (start > r || end < l) {
        return 0;
    }
    if (start >= l && end <= r) {
        return tree[node];
    }
    int mid = start + (end - start
        ) / 2;
    return func(query(2 * node,
        start, mid, l, r), query(2 *
        node + 1, mid + 1, end, l,
        r));
}
void range_update(int l, int r,
    int value, bool is_add = true)
{
    update(1, 0, n - 1, l, r,
        value, is_add);
}
int range_query(int l, int r) {
    return query(1, 0, n - 1, l, r
        );
}
vector<int> to_list() {
    vector<int> result(n);
    for (int i = 0; i < n; ++i) {
        result[i] = range_query(i,
            i);
    }
    return result;
}

```