Indian Institute of Information Technology, Nagpur

Team Blank

I/O

```
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
process = psutil.Process(os.getpid())
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
def lcm(a,b):return a//gcd(a,b)*b
II = lambda : int(sys.stdin.readline().strip())
LII = lambda : list(map(int, sys.stdin.readline().split()))
SI = lambda : sys.stdin.readline().strip()
LII_1 = lambda : list(map(lambda x:int(x)-1, sys.stdin.readline().split()))
if True:
   base = os.path.dirname(os.path.abspath(__file__))
   sys.stdin = open(os.path.join(base, "input.txt"), "r")
   sys.stdout = open(os.path.join(base, "output.txt"), "w")
   sys.stderr = open(os.path.join(base, "error.txt"), "w")
def solve():
   return
solve()
```

Number Theory

```
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 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(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 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)-1,-1,-1):
            k = L[i][0]
            for l in range(L[i][1]):
                st.append(st[j]*k)
                k *= L[i][0]
    return st
```

```
def memodict(f):
    class memodict(dict):
        def __missing__(self, key):
            ret = self[key] = f(key)
            return ret
    return memodict().__getitem__
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
@memodict
def prime_factors_large(n):
    # returns prime factor in n^(1/4) but is probablistic
    if n <= 1:
        return Counter()
    f = pollard_rho(n)
    return Counter([n]) if f == n else prime_factors_large(f) +
        prime_factors_large(n // f)
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
```

Combinatorics

```
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
```

Ways to partition n into k or fewer parts of size 1 or greater

```
@memoize
def partition(n, k):
    if n < 0:return 0
    if n == 0:return 1
    if k < 1:return 0
    return 1 if n == 1 else partition(n, k - 1) + partition(n - k, k)
```

Number of ways to partition n elements into k non-empty subsets

```
stirling_2 = lambda n, k: sum(((-1)**(k - j)) * nCr(k, j) * (j**n) for j in range
    (k + 1)) // math.factorial(k)
```

Number of permutations of n objects where no object appears in its original position

```
derangements = lambda n: int(math.factorial(n) / math.e + 0.5)
```

Geometry

```
## 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, 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),
# 3d line: dx + ez + f = 0 is (d, e, 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[0] - _p2[0], _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))
##
```

Linear Algebra

```
def max_xor(A,flag=True):
    base = [];how = {};reduced_base = {}
    for i in range(len(A)):
        a = A[i];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(i)
                break
    x = 0;tmp = 0
    for j in sorted(reduced_base, reverse=True):
        if not x & (1 << j):
            x ^= reduced_base[j]
            tmp ^= how[j]
    if flag:
        # elements whose combination returns all possible subet xors
        return list(reduced_base.values())
    # elements whose xor is maximum
    return [base[j] for j in range(len(base)) if tmp & (1 << j)]

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

Data Structures

Disjoint Set Union

```
class DisjointSetUnion:
    def __init__(self, n):
        self.parent = list(range(n))
        # self.s = set(self.parent)
        self.size = [1] * 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.s.remove(b)
            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 len(self.s)
    def notfind(self, a):
        k = self.find(a)
        for j in self.s:
            if j!=k:
                return j
        return -1
```

Persistent DSU

```
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 isconnected(self,a,b,time):
        return self.find(a,time)==self.find(b,time)
```

Fenwick Tree

```
class BIT:
    #Faster than segment tree so use if possbile
    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] = []) -> 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 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
    def __repr__(self):
        return f"SegmentTree({[self.range_query(i,i) for i in range(self.n)]})"
```

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 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)
```

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

```
class ConvexHull:
    def __init__(self, n=100000):
        # put n equal to max value of ai , bi , you may need to do coordinate
            compression in case it is upto 10**9
        # works for value which are not increasing as well
        self.n = n
        self.seg = [Line(0, float('inf'))] * (4 * n)
        self.lo = [0] * (4 * n)
        self.hi = [0] * (4 * n)
        self.build(1,1,n)
    def build(self, i, l, r):
        stack = [(i, l, r)]
        while stack:
            idx, left, right = stack.pop()
            self.lo[idx] = left
            self.hi[idx] = right
            self.seg[idx] = Line(0, float('inf'))
            if left == right:
                continue
            mid = (left + right) // 2
            stack.append((2 * idx + 1, mid + 1, right))
            stack.append((2 * idx, left, mid))
    def insert(self,L):
        pos = 1
        while True:
            l, r = self.lo[pos], self.hi[pos]
            if l == r:
                if L(l) < self.seg[pos](l):
                    self.seg[pos] = L
                break
            m = (l + r) // 2
            if self.seg[pos].m < L.m:
                self.seg[pos], L = L, self.seg[pos]
            if self.seg[pos](m) > L(m):
                self.seg[pos], L = L, self.seg[pos]
                pos = 2*pos
            else:
                pos = 2*pos+1
    def query(self,x):
        i = 1
        res = self.seg[i](x)
        pos = i
        while True:
            l, r = self.lo[pos], self.hi[pos]
            if l == r:
                return min(res, self.seg[pos](x))
            m = (l + r) // 2
            if x < m:
                res = min(res, self.seg[pos](x))
                pos = 2 * pos
            else:
                res = min(res, self.seg[pos](x))
                pos = (2 * pos + 1)
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
class Line:
    def __init__(self, m, b, c=0):
        # c is an identifier for the line
        self.m = m;self.b = b;self.c = c
    def __call__(self, x):
        return self.m * x + self.b
```

Heavy Light Decomposition

```
class HLD:
   def __init__(self, adj, values, root=0,func=max,unit=float('-inf')):
      self.adj = adj
      self.values = values
      self.parent = [-1] * len(adj)
      self.depth = [0] * len(adj)
      self.size = [0] * len(adj)
      self.heavy = [-1] * len(adj)
      self.head = [0] * len(adj)
      self.pos = [0] * len(adj)
      self.flat = [0] * len(adj)
      self.unit = unit
      self.func = func
      self._dfs(root)
      self._decompose(root)
      self.seg = SegmentTree([self.values[self.flat[i]] for i in range(len(self.
          adj))],func,unit)
   def _dfs(self,start=0):
      visited = [False] * len(self.adj)
      stack = [start]
      while stack:
         start = stack[-1]
         if not visited[start]:
            visited[start] = True
            for child in self.adj[start]:
               if not visited[child]:
                  self.parent[child] = start
                  self.depth[child] = self.depth[start]+1
                  stack.append(child)
         else:
            self.size[stack.pop()] = 1
            k = 0
            for child in self.adj[start]:
               if self.parent[start]!=child:
                  self.size[start] += self.size[child]
                  if self.size[child]>k:
                     k = self.size[child]
                     self.heavy[start] = child
      return visited
   def _decompose(self, root):
      stack = [(root,root)]
      time = 0
      while stack:
         u, h = stack.pop()
         self.head[u] = h
         self.flat[time] = u
         self.pos[u] = time
         time += 1
         for v in reversed(self.adj[u]):
            if v!=self.parent[u] and v!=self.heavy[u]:
               stack.append((v, v))
         if self.heavy[u] != -1:
            stack.append((self.heavy[u], h))
   def query(self, u, v):
      res = self.unit
      while self.head[u] != self.head[v]:
         if self.depth[self.head[u]] < self.depth[self.head[v]]:
            u, v = v, u
         res = self.func(res, self.seg.query(self.pos[self.head[u]], self.pos[u]
             + 1))
         u = self.parent[self.head[u]]
      if self.depth[u] > self.depth[v]:
         u, v = v, u
      return self.func(res, self.seg.query(self.pos[u], self.pos[v] + 1))
   def update(self, u, value):
      self.seg.update(self.pos[u], value)
   def update_path(self, u, v, value):
      while self.head[u] != self.head[v]:
         if self.depth[self.head[u]] < self.depth[self.head[v]]:
            u, v = v, u
         self.seg.range_update(self.pos[self.head[u]], self.pos[u], value)
         u = self.parent[self.head[u]]
      if self.depth[u] > self.depth[v]:
         u, v = v, u
      self.seg.range_update(self.pos[u], self.pos[v], value)
   def add_to_subtree(self, u, value):
      self.seg.range_update(self.pos[u], self.pos[u] + self.size[u] - 1, value)
```

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())
```

Merge Sort Tree

```
class MergeSortTree:
    def merge(L,L1):
        L2 = []
        L = list(L);L1 = list(L1)
        i = j = 0
        while j<len(L) and i<len(L1):
            if L[j]<L1[i]:
                L2.append(L[j])
                j += 1
            else:
                L2.append(L1[i])
                i += 1
        L2.extend(L[j:]+L1[i:])
        return L2
    def __init__(self, data):
        self.n = len(data)
        self.tree = [SortedList() for _ in range(2 * self.n)]
        self.build(data)
    def build(self, data):
        for i in range(self.n):
            self.tree[self.n + i].insert(data[i])
        for i in range(self.n - 1, 0, -1):
            self.tree[i] = SortedList(MergeSortTree.merge(self.tree[i << 1],self.
                tree[i<<1|1]))
    def query(self, left, right,l,r):
        res = 0
        left += self.n
        right += self.n
        while left < right:
            if left & 1:
                res += self.func(self.tree[left], l, r);left += 1
            if right & 1:
                right -= 1;res += self.func(self.tree[right], l, r)
            left >>= 1;right >>= 1
        return res
    def func(self, L, l, r):
        return L.bisect_right(r)-L.bisect_left(l)
    def update(self,pos,value):
        pos += self.n;old_val = self.tree[pos][0]
        self.tree[pos].remove(old_val)
        self.tree[pos].add(value)
        while pos:
            pos >>= 1
            self.tree[pos].remove(old_val)
            self.tree[pos].add(value)
```

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 is_substring(self, s):
      current = 0
      for ch in s:
         if ch not in self.states[current].next:
            return False
         current = self.states[current].next[ch]
      return True
   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:
    @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] * (self.n<<1)
        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<<1], self.tree[(i<<1) + 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 >>= 1
            self.tree[pos] = self.func(self.tree[pos<<1], self.tree[(pos<<1) + 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&1:
                max_val = self.func(max_val, self.tree[left])
                left += 1
            if right&1:
                right -= 1
                max_val = self.func(max_val, self.tree[right])
            left >>= 1
            right >>= 1
        return max_val
    def __repr__(self):
        values = [str(self.query(i, i + 1)) for i in range(self.n)]
        return f"Seg[{', '.join(values)}]"
```

Sparse Table

```
class SparseTable:
    @staticmethod
    def func(a,b):
        # func(a,a) should be a
        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):
        # query from [L,R]
        j = int(math.log2(R - L + 1))
        return self.func(self.table[L][j], self.table[R - (1 << j) + 1][j])
```

BitArray

```
class BitSet:
    ADDRESS_BITS_PER_WORD = 12
    BITS_PER_WORD = 1 << ADDRESS_BITS_PER_WORD
    WORD_MASK = -1
    def __init__(self, sz):
        self.sz = sz
        self.words = [0] * (self._wordIndex(sz - 1) + 1)
    def _wordIndex(self, bitIndex):
        if bitIndex >= self.sz:
            raise ValueError("out of bound index", bitIndex)
        return bitIndex >> BitSet.ADDRESS_BITS_PER_WORD
    def flip(self, bitIndex):
        wordIndex = self._wordIndex(bitIndex)
        self.words[wordIndex] ^= 1 << (bitIndex % BitSet.BITS_PER_WORD)
    def flip_range(self, l, r):
```

```
        startWordIndex = self._wordIndex(l)
        endWordIndex = self._wordIndex(r)
        firstWordMask = BitSet.WORD_MASK << (l % BitSet.BITS_PER_WORD)
        rem = (r+1) % BitSet.BITS_PER_WORD
        lastWordMask = BitSet.WORD_MASK if rem == 0 else ~(BitSet.WORD_MASK << rem)
        if startWordIndex == endWordIndex:
            self.words[startWordIndex] ^= (firstWordMask & lastWordMask)
        else:
            self.words[startWordIndex] ^= firstWordMask
            for i in range(startWordIndex + 1, endWordIndex):
                self.words[i] ^= BitSet.WORD_MASK
            self.words[endWordIndex] ^= lastWordMask
    def __setitem__(self, bitIndex, value):
        wordIndex = self._wordIndex(bitIndex)
        if value:
            self.words[wordIndex] |= 1 << (bitIndex % BitSet.BITS_PER_WORD)
        else:
            self.words[wordIndex] &= ~(1 << (bitIndex % BitSet.BITS_PER_WORD))
    def __getitem__(self, bitIndex):
        wordIndex = self._wordIndex(bitIndex)
        return self.words[wordIndex] & (1 << (bitIndex % BitSet.BITS_PER_WORD)) !=
             0
    def nextSetBit(self, fromIndex):
        wordIndex = self._wordIndex(fromIndex)
        word = self.words[wordIndex] & (BitSet.WORD_MASK << (fromIndex % BitSet.
            BITS_PER_WORD))

        while True:
            if word != 0:
                return wordIndex * BitSet.BITS_PER_WORD + (word & -word).bit_length()
                     - 1
            wordIndex += 1
            if wordIndex > len(self.words) - 1:
                return -1
            word = self.words[wordIndex]
    def nextClearBit(self, fromIndex):
        wordIndex = self._wordIndex(fromIndex)
        word = ~self.words[wordIndex] & (BitSet.WORD_MASK << (fromIndex % BitSet.
            BITS_PER_WORD))

        while True:
            if word != 0:
                index = wordIndex * BitSet.BITS_PER_WORD + (word & -word).bit_length
                    () - 1
                return index if index < self.sz else - 1
            wordIndex += 1
            if wordIndex > len(self.words) - 1:
                return -1
            word = ~self.words[wordIndex]
    def lastSetBit(self):
        wordIndex = len(self.words) - 1
        word = self.words[wordIndex]

        while wordIndex >= 0:
            if word != 0:
                return wordIndex * BitSet.BITS_PER_WORD + (word.bit_length() - 1 if
                     word > 0 else BitSet.BITS_PER_WORD - 1)
            wordIndex -= 1
            word = self.words[wordIndex]
        return -1
    def __str__(self):
        res = []
        st = 0
        while True:
            i = self.nextSetBit(st)
            if i != -1:
                res += [0] * (i - st)
                j = self.nextClearBit(i)
                if j != -1:
                    res += [1] * (j-i)
                    st = j
                else:
                    res += [1] * (self.sz - i)
                    break
            else:
                res += [0] * (self.sz - st)
                break

        return "".join(str(v) for v in res)
    def __repr__(self):
        return "Bitset(%s)" % str(self)
    def __iter__(self):
        for i in self[:]:
            yield i
    def __len__(self):
        return self.sz
    def __or__(self, other):
        if self.sz != other.sz:
            raise ValueError("BitSets must be of equal size")
        res = BitSet(self.sz)
        res.words = [a | b for a, b in zip(self.words, other.words)]
        return res
    def __add__(self, other):
        if self.sz != other.sz:
            raise ValueError("BitSets must be of equal size")
        res = BitSet(self.sz)
        carry = 0
        for i in range(len(self.words)):
            total = self.words[i] + other.words[i] + carry
            res.words[i] = total & BitSet.WORD_MASK
            carry = total >> BitSet.BITS_PER_WORD
        return res
    def __and__(self, other):
        if self.sz != other.sz:
            raise ValueError("BitSets must be of equal size")
        res = BitSet(self.sz)
        res.words = [a & b for a, b in zip(self.words, other.words)]
        return res
    def __xor__(self, other):
        if self.sz != other.sz:
            raise ValueError("BitSets must be of equal size")
        res = BitSet(self.sz)
        res.words = [a ^ b for a, b in zip(self.words, other.words)]
        return res
    def __invert__(self):
        res = BitSet(self.sz)
        res.words = [~a & BitSet.WORD_MASK for a in self.words]
        return res
    def add(self, val):
        self.flip_range(val, self.nextClearBit(val))
    def rem(self, val):
        self.flip_range(val, self.nextSetBit(val))
```

Graphs

Basics

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

def dijkstra(d,start=0):
    n = len(d)
    dist = [float("inf")]*n
    # parents = [-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 d[v]:
                new_dist = edge_len+path_len
                if new_dist<dist[w]:
                    dist[w] = new_dist
                    # parents[w] = v
                    heappush(queue, (new_dist, w))
    return dist
```

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)]
    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]
    return dist, pred

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

```
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[w] < 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

class binary_lift:
   def __init__(self, graph, f=max, root=0, flag=False):
      n = len(graph)
      parent = [-1] * (n + 1)
      depth = self.depth = [-1] * n
      bfs = [root]
      depth[root] = 0
      data = [0]*n
      for node in bfs:
         # for nei,w in graph[node]:
         for nei in graph[node]:
            if depth[nei] == -1:
               # data[nei] = w
               parent[nei] = node
               depth[nei] = depth[node] + 1
               bfs.append(nei)
      parent = self.parent = [parent]
      self.f = f
      if flag:
         data = self.data = [data]
         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])
        else:
            for _ in range(max(depth).bit_length()):
                old_parent = parent[-1]
                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, c=0):
        depth = self.depth
        parent = self.parent
        data = self.data
        f = self.f
        c = self.lca(a, b)
        val = 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
```

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))
        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 hoga , toh y hoga
        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)]
```

Bridges and Aritculation Points

```
    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 bridges_on_path(adj):
    # returns all bridges on path from 1 to n
    n = len(adj)
    timer = 0
    visited = [False]*n
    tin = [-1]*n
    low = [-1]*n
    bridges = []
    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))
    comp_id = [-1]*n
    comp = 0
    bridge_set = set(bridges)
    for i in range(n):
        if comp_id[i] != -1:
            continue
        stack = [i]
        comp_id[i] = comp
        while stack:
            v = stack.pop()
            for to in adj[v]:
                if comp_id[to] == -1 and (v, to) not in bridge_set and (to, v) not in
                    bridge_set:
                    comp_id[to] = comp
                    stack.append(to)
        comp += 1

    tree = [[] for _ in range(comp)]
    for u, v in bridges:
        cu, cv = comp_id[u], comp_id[v]
        tree[cu].append((cv, (u, v)))
        tree[cv].append((cu, (u, v)))

    c1, cN = comp_id[0], comp_id[n-1]
    if c1 == cN:
        return []
    parent = {c1: None}
    edge_used = {c1: None}
    stack = [c1]
    while stack:
        u = stack.pop()
        for v, e in tree[u]:
            if v not in parent:
                parent[v] = u
                edge_used[v] = e
                stack.append(v)
    path_bridges = []
    cur = cN
    while cur != c1:
        path_bridges.append(edge_used[cur])
        cur = parent[cur]
    return [(u,v) for u, v in path_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
```

Strings

String Hashing

```
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)
        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
```

String Functions

```
def LCPArray(L,SA=None):
    # Longest Common prefix in between 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):
    # 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 manacher(s):
    # returns longest palindrome in 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 lexographic 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]) != sort_key(sa[i]))
        rank, tmp = tmp, rank
        k += 1
        if (1 << k) >= n:
            break
    return sa
```

FFT

FFT

```
class FastFFT:
    # This is a bit faster, one log n factor is less but it's accuracy is not 100%
    # use this when coeffecient does not matter(set coeffecient to 1 repeatedly)
    # or you could just risk it :)
    def __init__(self, MOD=998244353):
        FastFFT.MOD = MOD
        # g = self.primitive_root_constexpr()
        g = 3
        ig = pow(g, FastFFT.MOD - 2, FastFFT.MOD)
        FastFFT.W = [pow(g, (FastFFT.MOD - 1) >> i, FastFFT.MOD) for i in range(30)
            ]
        FastFFT.iW = [pow(ig, (FastFFT.MOD - 1) >> i, FastFFT.MOD) for i in range
            (30)]
    def primitive_root_constexpr(self):
        if FastFFT.MOD == 998244353:
            return 3
        elif FastFFT.MOD == 200003:
            return 2
        elif FastFFT.MOD == 167772161:
            return 3
        elif FastFFT.MOD == 469762049:
            return 3
        elif FastFFT.MOD == 754974721:
            return 11
        divs = [0] * 20
        divs[0] = 2
        cnt = 1
        x = (FastFFT.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, (FastFFT.MOD - 1) // divs[i], FastFFT.MOD) == 1:
                    ok = False
                    break
            if ok:
                return g
            g += 1
    def fft(self, k, f):
        for l in range(k, 0, -1):
            d = 1 << l - 1
            U = [1]
            for i in range(d):
                U.append(U[-1] * FastFFT.W[l] % FastFFT.MOD)
            for i in range(1 << k - l):
                for j in range(d):
                    s = i * 2 * d + j
                    f[s], f[s + d] = (f[s] + f[s + d]) % FastFFT.MOD, U[j] * (f[s] - f
                        [s + d]) % FastFFT.MOD
    def ifft(self, k, f):
        for l in range(1, k + 1):
            d = 1 << l - 1
            for i in range(1 << k - l):
                u = 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]) % FastFFT.MOD, (f[j] - f[j + d
                        ]) % FastFFT.MOD
                    u = u * FastFFT.iW[l] % FastFFT.MOD
    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))
        self.fft(k, A)
        self.fft(k, B)
        A = [a * b % FastFFT.MOD for a, b in zip(A, B)]
        self.ifft(k, A)
        inv = pow(n, FastFFT.MOD - 2, FastFFT.MOD)
        del A[n0:]
        for i in range(n0):
            A[i] = (A[i]*inv)%FastFFT.MOD
        return A

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 - l):
                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 - l):
                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
    if invert:
        for i in range(n):
            a[i] //= n

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
```

Miscellaneous

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)
interactive()
```

Definitions

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

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.

C++ Notes

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;`

CP - C++

```cpp
#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 INF LLONG_MAX
#define fastio ios::sync_with_stdio(false); cin.tie(0);
#define len(arr) arr.size()
#define f first
#define s second
#define pb push_back
using vi = vector<int>;
using vii = vector<vector<int>>;
using pi = pair<int,int>;
#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,
   std::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 LII(int n){vi a(n);cin>>a;return a;}

void solve() {

}

int32_t main() {
   fastio
   int t=1;
   cin >> t;
   while (t--) solve();
   return 0;
}
```

Sorted List - C++

```cpp
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});
   }
   T operator[](int k) {
      return os.find_by_order(k)->first;
   }
   int size() const {
      return os.size();
   }
   void print() {
      for (int i = 0; i < size(); i++) std::cout << (*this)[i] << " ";
      std::cout << "\n";
   }
};
```

Lazy Segment Tree - C++

```cpp
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;
}
};
```

FFT - C++

```
class FFT {
public:
    static const int MOD = 998244353;
    static const int MOD1 = 469762049;
    int MOD2, mod_inv;
    vector<int> W, W1, iW, iW1;
    FFT() {
        MOD2 = power(MOD, MOD1 - 2, MOD1);
        mod_inv = XT_GCD(MOD, MOD1).second % MOD1;
        int g = 3;
        int ig = power(g, MOD - 2, MOD);
        int ig1 = power(g, MOD1 - 2, MOD1);
        W.resize(30); W1.resize(30); iW.resize(30); iW1.resize(30);
        for (int i = 0; i < 30; ++i) {
            W[i] = power(g, (MOD - 1) >> i, MOD);
            W1[i] = power(g, (MOD1 - 1) >> i, MOD1);
            iW[i] = power(ig, (MOD - 1) >> i, MOD);
            iW1[i] = power(ig1, (MOD1 - 1) >> i, MOD1);
        }
    }
    int power(int base, int exp, int mod) {
        int res = 1;
        while (exp) {
            if (exp % 2) res = res * base % mod;
            base = base * base % mod;
            exp /= 2;
        }
        return res;
    }
    pair<int, int> XT_GCD(int a, int b) {
        if (b == 0) return {a, 1};
        pair<int,int> k9 = XT_GCD(b, a % b);
        int g = k9.first;int x1 = k9.second;
        return {g, x1 - (a / b) * x1};
    }

    int CRT(int a, int mod1, int b, int mod2) {
        return (a + (b - a) * mod_inv % mod2 * mod1) % (mod1 * mod2);
    }
    void fft(int k, vector<int>& f, vector<int>& f1) {
        for (int l = k; l > 0; --l) {
            int d = 1 << (l - 1);
            vector<pair<int, int>> U = {{1, 1}};
            for (int i = 0; i < d; ++i)
                U.emplace_back(U.back().first * W[l] % MOD, U.back().second * W1[l] %
                    MOD1);
            for (int i = 0; i < (1 << (k - l)); ++i) {
                for (int j = 0; j < d; ++j) {
                    int s = i * 2 * d + j;
                    int tmp_f = f[s] - f[s + d];
                    int tmp_f1 = f1[s] - f1[s + d];
```

```
                f[s] = (f[s] + f[s + d]) % MOD;
                f[s + d] = U[j].first * tmp_f % MOD;
                f1[s] = (f1[s] + f1[s + d]) % MOD1;
                f1[s + d] = U[j].second * tmp_f1 % MOD1;
            }
        }
    }
}
void ifft(int k, vector<int>& f, vector<int>& f1) {
    for (int l = 1; l <= k; ++l) {
        int d = 1 << (l - 1);
        for (int i = 0; i < (1 << (k - l)); ++i) {
            int u = 1, u1 = 1;
            for (int j = i * 2 * d; j < (i * 2 + 1) * d; ++j) {
                f[j + d] = f[j + d] * u % MOD;
                f[j] = (f[j] + f[j + d]) % MOD;
                f[j + d] = (f[j] - f[j + d] + MOD) % MOD;
                u = u * iW[l] % MOD;
                f1[j + d] = f1[j + d] * u1 % MOD1;
                f1[j] = (f1[j] + f1[j + d]) % MOD1;
                f1[j + d] = (f1[j] - f1[j + d] + MOD1) % MOD1;
                u1 = u1 * iW1[l] % MOD1;
            }
        }
    }
}
vector<int> convolve(vector<int> A, vector<int> B) {
    int n0 = A.size() + B.size() - 1;
    int k = 0;
    while ((1 << k) < n0) ++k;
    int n = 1 << k;
    A.resize(n, 0);
    B.resize(n, 0);
    vector<int> A1 = A, B1 = B;
    fft(k, A, A1);
    fft(k, B, B1);
    for (int i = 0; i < n; ++i) {
        A[i] = A[i] * B[i] % MOD;
        A1[i] = A1[i] * B1[i] % MOD1;
    }
    ifft(k, A, A1);
    int inv = power(n, MOD - 2, MOD);
    int inv1 = power(n, MOD1 - 2, MOD1);
    A.resize(n0);
    for (int i = 0; i < n0; ++i) {
        A[i] = CRT(A[i] * inv % MOD, MOD, A1[i] * inv1 % MOD1, MOD1);
    }
    return A;
}
};
```