DAY-48

import sys, math

# Pretty important line here.

sys.setrecursionlimit(10\*\*6)

def preprocess\_original\_tree(node, parent):

    # "Jumping" in the current node.

    global timer

    tin[node] = timer

    timer += 1

    # Building a chart of the 2^i ancestors of each node.

    up[node][0] = parent

    # Setting each node's (which is the current child) 2^i ancestors' value.

    for i in range(1, LOG):

        if up[node][i - 1] != -1:

            up[node][i] = up[up[node][i - 1]][i - 1]

    # Perform DFS to set current node's children's depth recursively.

    for neighbor in adj[node]:

        if neighbor == parent:

            continue

        depth[neighbor] = depth[node] + 1

        preprocess\_original\_tree(neighbor, node)

    # "Jumping" out of the current node.

    tout[node] = timer

    timer += 1

def lift\_node(node, k):

    # Jumping to the greatest 2^i ancestor each time.

    for i in range(LOG - 1, -1, -1):

        if k & (1 << i):

            node = up[node][i]

    return node

def get\_lca(u, v):

    # Equalizing both node's depths.

    if depth[u] < depth[v]:

        u, v = v, u

    u = lift\_node(u, depth[u] - depth[v])

    if u == v:

        return u

    # Jumping to the greatest 2^i ancestor each time.

    for i in range(LOG - 1, -1, -1):

        if up[u][i] != up[v][i]:

            u = up[u][i]

            v = up[v][i]

    return up[u][0]

def get\_distance(u, v):

    ancestor = get\_lca(u, v)

    # It uses the original tree's preprocessed depths.

    return depth[u] + depth[v] - 2 \* depth[ancestor]

def build\_virtual\_tree(nodes):

    # Adding relevant nodes to virtual tree.

    nodes.sort(key=lambda x: tin[x])

    m = len(nodes)

    vt\_nodes = nodes[:]

    for i in range(m - 1):

        vt\_nodes.append(get\_lca(nodes[i], nodes[i + 1]))

    vt\_nodes = list(set(vt\_nodes))

    vt\_nodes.sort(key=lambda x: tin[x])

    # Connecting nodes in virtual tree.

    tree = {node: [] for node in vt\_nodes}

    stack = []

    # All virtual tree nodes are stored in the order in which they were found, thus preserving their hierarchy from left to right.

    for node in vt\_nodes:

        # Validating if the last ancestor in the stack is the ancestor of the current node.

        while stack and tout[stack[-1]] < tin[node]:

            stack.pop()

        if stack:

            tree[stack[-1]].append(node)

        stack.append(node)

    return tree, vt\_nodes

def solve\_query(query\_nodes):

    # Traversing query nodes (virtual tree's nodes)

    def dp(u):

        nonlocal res

        s = query\_val.get(u, 0)

        # Performing DFS.

        for v in vt[u]:

            sub = dp(v)

            # Since

            # sum(u in sub) \* sum(v not in sub)

            # = (sum(u in sub)) \* (sum(v not in sub))

            # = sub \* (S\_total - sub)

            res = (res + sub \* (S\_total - sub) % MOD \* get\_distance(u, v)) % MOD

            s += sub

        # Returning the total sum of the current subtree.

        return s

    if len(query\_nodes) < 2:

        return 0

    S\_total = sum(query\_nodes)

    query\_val = {node: node for node in query\_nodes}

    vt, vt\_nodes = build\_virtual\_tree(query\_nodes)

    res = 0

    dp(vt\_nodes[0])

    return res

MOD = 10\*\*9 + 7

timer = 0

data = sys.stdin.read().split()

it = iter(data)

n = int(next(it))

q = int(next(it))

LOG = int(math.log2(n)) + 1

up = [[-1] \* LOG for \_ in range(n + 1)]

depth = [0] \* (n + 1)

tin = [0] \* (n + 1)

tout = [0] \* (n + 1)

# Building original tree.

adj = [[] for \_ in range(n + 1)]

for \_ in range(n - 1):

    u, v = int(next(it)), int(next(it))

    adj[u].append(v)

    adj[v].append(u)

preprocess\_original\_tree(1, -1)

res = []

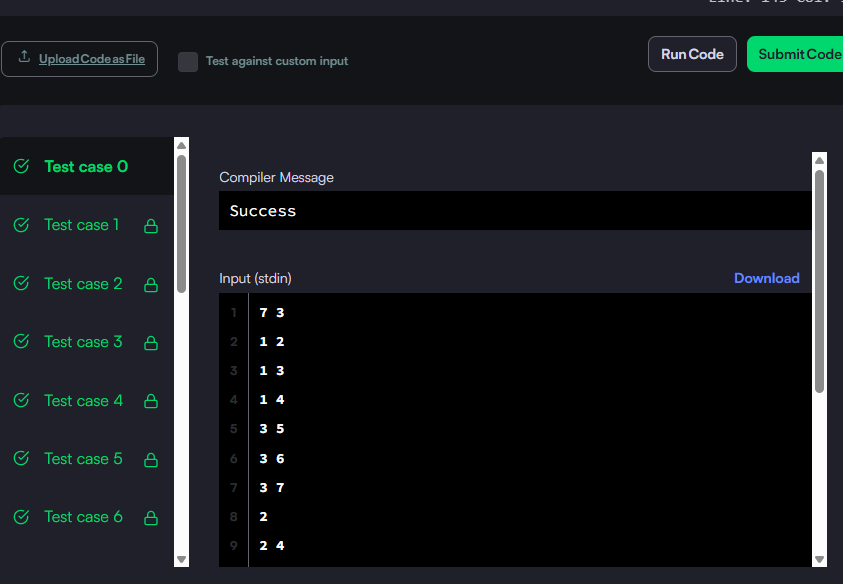
for \_ in range(q):

    k = int(next(it))

    query\_nodes = [int(next(it)) for \_ in range(k)]

    res.append(str(solve\_query(query\_nodes)))

sys.stdout.write("\n".join(res))



from enum import Enum

class State(Enum):

  INIT = 0

  VISITING = 1

  VISITED = 2

class Solution:

  def isPrintable(self, targetGrid: list[list[int]]) -> bool:

    MAX\_COLOR = 60

    m = len(targetGrid)

    n = len(targetGrid[0])

    graph = [set() for \_ in range(MAX\_COLOR + 1)]

    for color in range(1, MAX\_COLOR + 1):

      minI = m

      minJ = n

      maxI = -1

      maxJ = -1

      for i in range(m):

        for j in range(n):

          if targetGrid[i][j] == color:

            minI = min(minI, i)

            minJ = min(minJ, j)

            maxI = max(maxI, i)

            maxJ = max(maxJ, j)

      # Add any color covering the current as the children.

      for i in range(minI, maxI + 1):

        for j in range(minJ, maxJ + 1):

          if targetGrid[i][j] != color:

            graph[color].add(targetGrid[i][j])

    states = [State.INIT] \* (MAX\_COLOR + 1)

    def hasCycle(u: int) -> bool:

      if states[u] == State.VISITING:

        return True

      if states[u] == State.VISITED:

        return False

      states[u] = State.VISITING

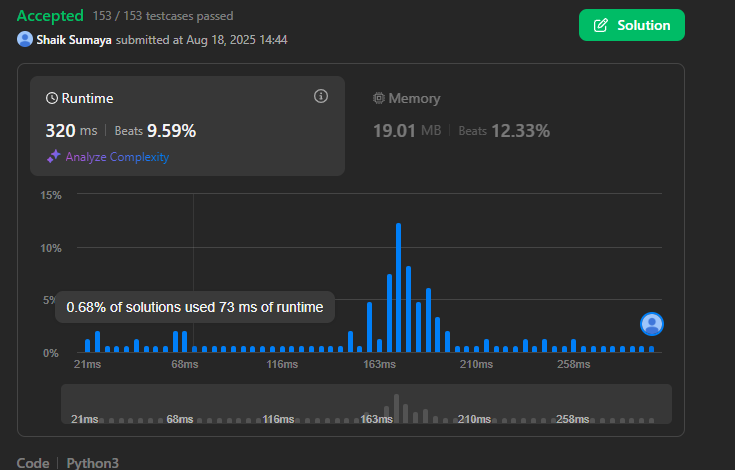
      if any(hasCycle(v) for v in graph[u]):

        return True

      states[u] = State.VISITED

      return False

    return not (any(hasCycle(i) for i in range(1, MAX\_COLOR + 1)))



# cook your dish here

MOD = 10\*\*9 + 7

def merge(a, size\_a, b, size\_b, c):

idx1, idx2, idx = 0, 0, 0

while idx1 < size\_a and idx2 < size\_b:

if a[idx1] < b[idx2]:

c[idx] = a[idx1]

idx1 += 1

else:

c[idx] = b[idx2]

idx2 += 1

idx += 1

while idx1 < size\_a:

c[idx] = a[idx1]

idx1 += 1

idx += 1

while idx2 < size\_b:

c[idx] = b[idx2]

idx2 += 1

idx += 1

def count\_inversions(a, size\_a, b, size\_b):

ptr = size\_b - 1

ans = 0

for i in range(size\_a):

while ptr >= 0 and b[ptr] < a[i]:

ptr -= 1

ans = (ans + (size\_b - 1 - ptr)) % MOD

return ans

def sort\_and\_count(a, size\_a):

if size\_a == 1:

return 0

mid = size\_a // 2

sz1 = mid

sz2 = size\_a - sz1

a1 = a[:mid]

a2 = a[mid:]

# Number of inversions in the left and right subarray

left = sort\_and\_count(a1, sz1)

right = sort\_and\_count(a2, sz2)

# Number of inversions when both halves are put together

num = count\_inversions(a1, sz1, a2, sz2)

merge(a1, sz1, a2, sz2, a)

return ((left + right) % MOD + num) % MOD

t = int(input())

for \_ in range(t):

n = int(input())

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

print(sort\_and\_count(a, n))

