**数算Cheat Sheet （ 4/6/2025）**

**01094: Sorting It All Out**

**from collections import deque**

**def topo\_sort(graph):**

**in\_degree = {u:0 for u in graph}**

**for u in graph:**

**for v in graph[u]:**

**in\_degree[v] += 1**

**q = deque([u for u in in\_degree if in\_degree[u] == 0])**

**topo\_order = [];flag = True**

**while q:**

**if len(q) > 1:**

**flag = False#topo\_sort不唯一确定**

**u = q.popleft()**

**topo\_order.append(u)**

**for v in graph[u]:**

**in\_degree[v] -= 1**

**if in\_degree[v] == 0:**

**q.append(v)**

**if len(topo\_order) != len(graph): return 0**

**return topo\_order if flag else None**

**while True:**

**n,m = map(int,input().split())**

**if n == 0: break**

**graph = {chr(x+65):[] for x in range(n)}**

**edges = [tuple(input().split('<')) for \_ in range(m)]**

**for i in range(m):**

**a,b = edges[i]**

**graph[a].append(b)**

**t = topo\_sort(graph)**

**if t:**

**s = ''.join(t)**

**print("Sorted sequence determined after {} relations: {}.".format(i+1,s))**

**break**

**elif t == 0:**

**print("Inconsistency found after {} relations.".format(i+1))**

**break**

**else:**

**print("Sorted sequence cannot be determined.")**

**01145: Tree Summing**

**class TreeNode:**

**def \_\_init\_\_(self, val=0, left=None, right=None):**

**self.val = val**

**self.left = left**

**self.right = right**

**def has\_path\_sum(root, target\_sum):**

**if root is None:**

**return False**

**if root.left is None and root.right is None: # The current node is a leaf node**

**return root.val == target\_sum**

**left\_exists = has\_path\_sum(root.left, target\_sum - root.val)**

**right\_exists = has\_path\_sum(root.right, target\_sum - root.val)**

**return left\_exists or right\_exists**

**def parse\_tree(s):**

**stack = []**

**i = 0**

**while i < len(s):**

**if s[i].isdigit() or s[i] == '-':**

**j = i**

**while j < len(s) and (s[j].isdigit() or s[j] == '-'):**

**j += 1**

**num = int(s[i:j])**

**node = TreeNode(num)**

**if stack:**

**parent = stack[-1]**

**if parent.left is None:**

**parent.left = node**

**else:**

**parent.right = node**

**stack.append(node)**

**i = j**

**elif s[i] == '[':**

**i += 1**

**elif s[i] == ']' and s[i - 1] != '[' and len(stack) > 1:**

**stack.pop()**

**i += 1**

**else:**

**i += 1**

**return stack[0] if len(stack) > 0 else None**

**while True:**

**try:**

**s = input()**

**except:**

**break**

**s = s.split()**

**target\_sum = int(s[0])**

**tree = ("").join(s[1:])**

**tree = tree.replace('(', ',[').replace(')', ']')**

**while True:**

**try:**

**tree = eval(tree[1:])**

**break**

**except SyntaxError:**

**s = input().split()**

**s = ("").join(s)**

**s = s.replace('(', ',[').replace(')', ']')**

**tree += s**

**tree = str(tree)**

**tree = tree.replace(',[', '[')**

**if tree == '[]':**

**print("no")**

**continue**

**root = parse\_tree(tree)**

**if has\_path\_sum(root, target\_sum):**

**print("yes")**

**else:**

**print("no")**

**01178: Camelot**

**import sys**

**inf = float('infinity')**

**kmove = [(1,0),(1,1),(0,1),(-1,1),(-1,0),(-1,-1),(0,-1),(1,-1)]**

**knmove = [(2,1),(1,2),(-1,2),(-2,1),(-2,-1),(-1,-2),(1,-2),(2,-1)]**

**kmap = [[inf]\*64 for \_ in range(64)]**

**knmap = [[inf]\*64 for \_ in range(64)]**

**def ok(x, y):**

**return 0 <= x < 8 and 0 <= y < 8**

**def getxy(p):**

**return p % 8, p // 8**

**def getPosition(x, y):**

**return x + y \* 8**

**def init():**

**for i in range(64):**

**kmap[i][i] = 0**

**knmap[i][i] = 0**

**x, y = getxy(i)**

**for j in range(8):**

**tx, ty = kmove[j][0] + x, kmove[j][1] + y**

**if ok(tx, ty):**

**next = getPosition(tx, ty)**

**kmap[i][next] = 1**

**tx, ty = knmove[j][0] + x, knmove[j][1] + y**

**if ok(tx, ty):**

**next = getPosition(tx, ty)**

**knmap[i][next] = 1**

**def floyd():**

**for k in range(64):**

**for i in range(64):**

**for j in range(64):**

**kmap[i][j] = min(kmap[i][j], kmap[i][k] + kmap[k][j])**

**knmap[i][j] = min(knmap[i][j], knmap[i][k] + knmap[k][j])**

**init()**

**floyd()**

**s = input().strip()**

**size = len(s)**

**num = 0**

**position = [0]\*64**

**for i in range(0, size, 2):**

**position[num] = ord(s[i]) - ord('A') + (ord(s[i+1]) - ord('1')) \* 8**

**num += 1**

**minmove = inf**

**total = 0 # Renamed 'sum' to 'total'**

**for ds in range(64):**

**for m in range(64):**

**for k in range(1, num):**

**total = sum(knmap[position[i]][ds] for i in range(1, num))**

**total += kmap[position[0]][m]**

**total += knmap[position[k]][m] + knmap[m][ds]**

**total -= knmap[position[k]][ds]**

**minmove = min(minmove, total)**

**print(minmove)**

**01182：食物链**

**class DisjointSet:**

**def \_\_init\_\_(self, n):**

**self.parent = [i for i in range(3 \* n + 1)]**

**self.rank = [0] \* (3 \* n + 1)**

**def find(self, u):**

**if self.parent[u] != u:**

**self.parent[u] = self.find(self.parent[u])**

**return self.parent[u]**

**def union(self, u, v):**

**pu, pv = self.find(u), self.find(v)**

**if pu == pv:**

**return False**

**if self.rank[pu] > self.rank[pv]:**

**self.parent[pv] = pu**

**elif self.rank[pu] < self.rank[pv]:**

**self.parent[pu] = pv**

**else:**

**self.parent[pv] = pu**

**self.rank[pu] += 1**

**return True**

**def is\_valid(n, k, statements):**

**dsu = DisjointSet(n)**

**def find\_disjoint\_set(x):**

**if x > n:**

**return False**

**return True**

**false\_count = 0**

**for d, x, y in statements:**

**if not find\_disjoint\_set(x) or not find\_disjoint\_set(y):**

**false\_count += 1**

**continue**

**if d == 1: # X and Y are of the same type**

**if dsu.find(x) == dsu.find(y + n) or dsu.find(x) == dsu.find(y + 2 \* n):**

**false\_count += 1**

**else:**

**dsu.union(x, y)**

**dsu.union(x + n, y + n)**

**dsu.union(x + 2 \* n, y + 2 \* n)**

**else:**

**if dsu.find(x) == dsu.find(y) or dsu.find(x + 2\*n) == dsu.find(y):**

**false\_count += 1**

**else:**

**dsu.union(x + n, y)**

**dsu.union(x, y + 2 \* n)**

**dsu.union(x + 2 \* n, y + n)**

**return false\_count**

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

**N, K = map(int, input().split())**

**statements = []**

**for \_ in range(K):**

**D, X, Y = map(int, input().split())**

**statements.append((D, X, Y))**

**result = is\_valid(N, K, statements)**

**print(result)**

**01258: Agri-Net**

**from heapq import heappop, heappush**

**while True:**

**try:**

**n = int(input())**

**except:**

**break**

**mat, cur = [], 0**

**for i in range(n):**

**mat.append(list(map(int, input().split())))**

**d, v, q, cnt = [100000 for i in range(n)], set(), [], 0**

**d[0] = 0**

**heappush(q, (d[0], 0))**

**while q:**

**x, y = heappop(q)**

**if y in v:**

**continue**

**v.add(y)**

**cnt += d[y]**

**for i in range(n):**

**if d[i] > mat[y][i]:**

**d[i] = mat[y][i]**

**heappush(q, (d[i], i))**

**print(cnt)**

**01321：棋盘问题**

**def count\_ways(board, n, k):**

**def backtrack(row, columns):**

**if len(columns) == k:**

**return 1**

**if row >= n:**

**return 0**

**count = 0**

**for col in range(n):**

**if board[row][col] == '#' and col not in columns :**

**columns.add(col)**

**count += backtrack(row + 1, columns)**

**columns.remove(col)**

**count += backtrack(row + 1, columns) # 考虑不放置棋子的情况**

**return count**

**return backtrack(0, set())**

**while True:**

**n, k = map(int, input().split())**

**if n == -1 and k == -1:**

**break**

**board = [input() for j in range(n)]**

**print(count\_ways(board, n, k))**

**01376: Robot**

**from collections import deque**

**def bfs\_min\_time(grid, start, end, direction):**

**N, M = len(grid), len(grid[0])**

**dir\_map = {'E': 0, 'S': 1, 'W': 2, 'N': 3}**

**start\_dir = dir\_map[direction]**

**sr, sc, tr, tc = start[0], start[1], end[0], end[1]**

**valid = [[False] \* (M) for \_ in range(N)]**

**for i in range(1, N):**

**for j in range(1, M):**

**if grid[i - 1][j - 1] == 0 and grid[i - 1][j] == 0 and grid[i][j - 1] == 0 and grid[i][j] == 0:**

**valid[i][j] = True**

**if not valid[sr][sc] or not valid[tr][tc]:**

**return -1**

**dr = [0, 1, 0, -1]**

**dc = [1, 0, -1, 0]**

**visited = [[[False] \* 4 for \_ in range(M)] for \_ in range(N)]**

**q = deque()**

**q.append((sr, sc, start\_dir, 0))**

**visited[sr][sc][start\_dir] = True**

**while q:**

**r, c, d, steps = q.popleft()**

**if r == tr and c == tc:**

**return steps**

**for nd in [(d + 3) % 4, (d + 1) % 4]:**

**if not visited[r][c][nd]:**

**visited[r][c][nd] = True**

**q.append((r, c, nd, steps + 1))**

**for k in range(1, 4):**

**nr = r + dr[d] \* k**

**nc = c + dc[d] \* k**

**if nr < 1 or nr >= N or nc < 1 or nc >= M:**

**break**

**if not valid[nr][nc]:**

**break**

**if not visited[nr][nc][d]:**

**visited[nr][nc][d] = True**

**q.append((nr, nc, d, steps + 1))**

**return -1**

**while True:**

**n, m = map(int, input().split())**

**if n == 0 and m == 0:**

**break**

**grid = [list(map(int, input().split())) for \_ in range(n)]**

**sx, sy, ex, ey, direction = input().split()**

**sx, sy, ex, ey = map(int, [sx, sy, ex, ey])**

**direction = direction.upper()**

**result = bfs\_min\_time(grid, (sx, sy), (ex, ey), direction[0])**

**print(result)**

**01426: Find The Multiple**

**from collections import deque**

**def find\_multiple(n):**

**q = deque()**

**q.append((1 % n, "1"))**

**visited = set([1 % n])**

**while q:**

**mod, num\_str = q.popleft()**

**if mod == 0:**

**return num\_str**

**for digit in ["0", "1"]:**

**new\_num\_str = num\_str + digit**

**new\_mod = (mod \* 10 + int(digit)) % n**

**if new\_mod not in visited:**

**q.append((new\_mod, new\_num\_str))**

**visited.add(new\_mod)**

**def main():**

**while True:**

**n = int(input())**

**if n == 0:**

**break**

**print(find\_multiple(n))**

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

**main()**

**01577: Falling Leaves**

**class TreeNode:**

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

**self.data = data**

**self.left = None**

**self.right = None**

**def build\_bst(leaves):**

**if not leaves:**

**return None**

**root = TreeNode(leaves[0])**

**for leaf in leaves[1:]:**

**insert\_node(root, leaf)**

**return root**

**def insert\_node(root, leaf):**

**if leaf < root.data:**

**if root.left is None:**

**root.left = TreeNode(leaf)**

**else:**

**insert\_node(root.left, leaf)**

**else:**

**if root.right is None:**

**root.right = TreeNode(leaf)**

**else:**

**insert\_node(root.right, leaf)**

**def preorder\_traversal(root):**

**if root is None:**

**return []**

**traversal = [root.data]**

**traversal.extend(preorder\_traversal(root.left))**

**traversal.extend(preorder\_traversal(root.right))**

**return traversal**

**flag = 0**

**while True:**

**leaves = []**

**while True:**

**line = input().strip()**

**if line == '\*':**

**break**

**elif line == '$':**

**flag = 1**

**break**

**else:**

**leaves.extend(line)**

**root = build\_bst(leaves[::-1])**

**traversal\_result = preorder\_traversal(root)**

**print(''.join(traversal\_result))**

**if flag:**

**break**

**01611: The Suspects**

**def find(x):**

**if parent[x] == x:**

**return parent[x]**

**else:**

**parent[x] = find(parent[x])**

**return parent[x]**

**def disjoint(x,y):**

**rep\_x,rep\_y = find(x),find(y)**

**if rep\_x != rep\_y:**

**if rank[rep\_x] < rank[rep\_y]:**

**parent[rep\_x] = rep\_y**

**elif rank[rep\_x] > rank[rep\_y]:**

**parent[rep\_y] = rep\_x**

**else:**

**parent[rep\_y] = rep\_x**

**rank[rep\_x] += 1**

**def joint(mylist):**

**node = mylist[0]**

**for element in mylist[1:]:**

**disjoint(node,element)**

**while True:**

**n,m = [int(i) for i in input().split()]**

**if n == 0 and m == 0:**

**break**

**parent = [i for i in range(n)]**

**rank = [0 for i in range(n)]**

**for i in range(m):**

**s = [int(i) for i in input().split()]**

**joint(s[1:])**

**rep\_0 = find(0)**

**print(len([i for i in parent if find(i) == rep\_0]))**

**01703：发现它，抓住它**

**class UnionFind:**

**def \_\_init\_\_(self, n):**

**self.parent = list(range(n))**

**self.rank = [0] \* n**

**def find(self, x):**

**if self.parent[x] != x:**

**self.parent[x] = self.find(self.parent[x])**

**return self.parent[x]**

**def union(self, x, y):**

**rootX = self.find(x)**

**rootY = self.find(y)**

**if rootX != rootY:**

**if self.rank[rootX] > self.rank[rootY]:**

**self.parent[rootY] = rootX**

**elif self.rank[rootX] < self.rank[rootY]:**

**self.parent[rootX] = rootY**

**else:**

**self.parent[rootY] = rootX**

**self.rank[rootX] += 1**

**def solve():**

**n, m = map(int, input().split())**

**uf = UnionFind(2 \* n) # 初始化并查集，每个案件对应两个节点**

**for \_ in range(m):**

**operation, a, b = input().split()**

**a, b = int(a) - 1, int(b) - 1**

**if operation == "D":**

**uf.union(a, b + n) # a与b的对立案件合并**

**uf.union(a + n, b) # a的对立案件与b合并**

**else: # "A"**

**if uf.find(a) == uf.find(b) or uf.find(a + n) == uf.find(b + n):**

**print("In the same gang.")**

**elif uf.find(a) == uf.find(b + n) or uf.find(a + n) == uf.find(b):**

**print("In different gangs.")**

**else:**

**print("Not sure yet.")**

**T = int(input())**

**for \_ in range(T):**

**solve()**

**01724: ROADS**

**import heapq**

**def dijkstra(g):**

**while pq:**

**dist,node,fee = heapq.heappop(pq)**

**if node == n-1 :**

**return dist**

**for nei,w,f in g[node]:**

**n\_dist = dist + w**

**n\_fee = fee + f**

**if n\_fee <= k:**

**dists[nei] = n\_dist**

**heapq.heappush(pq,(n\_dist,nei,n\_fee))**

**return -1**

**k,n,r = int(input()),int(input()),int(input())**

**g = [[] for \_ in range(n)]**

**for i in range(r):**

**s,d,l,t = map(int,input().split())**

**g[s-1].append((d-1,l,t)) #node,dist,fee**

**pq = [(0,0,0)] #dist,node,fee**

**dists = [float('inf')] \* n**

**dists[0] = 0**

**spend = 0**

**result = dijkstra(g)**

**print(result)**

**01760: Disk Tree**

**from collections import defaultdict**

**import sys**

**class TrieNode:**

**"""Trie 结点类"""**

**def \_\_init\_\_(self):**

**self.children = defaultdict(TrieNode) # 存储子目录**

**self.is\_end = False # 该标志在本题中可省略**

**class Trie:**

**"""Trie 前缀树"""**

**def \_\_init\_\_(self):**

**self.root = TrieNode()**

**def insert(self, path: str):**

**node = self.root**

**for folder in path.split("\\"): # 以 "\" 分割路径**

**node = node.children[folder] # 如果不存在则自动创建**

**def print\_tree(self, node=None, depth=0):**

**if node is None:**

**node = self.root**

**for folder in sorted(node.children): # 按字典序排序**

**print(" " \* depth + folder) # 根据深度打印**

**self.print\_tree(node.children[folder], depth + 1) # 递归打印子目**

**def main():**

**n = int(sys.stdin.readline().strip())**

**trie = Trie()**

**for \_ in range(n):**

**path = sys.stdin.readline().strip()**

**trie.insert(path)**

**trie.print\_tree()**

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

**main()**

**01789: Truck History**

**import heapq**

**def truck\_history():**

**while True:**

**n = int(input())**

**if n == 0:**

**break**

**trucks = [input() for \_ in range(n)]**

**trucks.sort()**

**graph = [[0]\*n for \_ in range(n)]**

**for i in range(n):**

**for j in range(i+1, n):**

**graph[i][j] = graph[j][i] = sum(a!=b for a, b in zip(trucks[i], trucks[j]))**

**visited = [False]\*n**

**min\_edge = [float('inf')]\*n**

**min\_edge[0] = 0**

**total\_distance = 0**

**min\_heap = [(0, 0)]**

**while min\_heap:**

**d, v = heapq.heappop(min\_heap)**

**if visited[v]:**

**continue**

**visited[v] = True**

**total\_distance += d**

**for u in range(n):**

**if not visited[u] and graph[v][u] < min\_edge[u]:**

**min\_edge[u] = graph[v][u]**

**heapq.heappush(min\_heap, (graph[v][u], u))**

**print(f"The highest possible quality is 1/{total\_distance}.")**

**truck\_history()**

**01860: Currency Exchange**

**import sys**

**def main():**

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

**N, M = map(int, data[:2])**

**S = int(data[2]) # 起始货币编号**

**V = float(data[3]) # 起始金额**

**edges = []**

**idx = 4**

**for \_ in range(M):**

**A = int(data[idx]); B = int(data[idx+1])**

**R\_ab = float(data[idx+2]); C\_ab = float(data[idx+3])**

**R\_ba = float(data[idx+4]); C\_ba = float(data[idx+5])**

**idx += 6**

**edges.append((A, B, R\_ab, C\_ab))**

**edges.append((B, A, R\_ba, C\_ba))**

**best = [0.0] \* (N + 1)**

**best[S] = V**

**for iteration in range(1, N + 1):**

**updated = False**

**for u, v, rate, fee in edges:**

**if best[u] > fee:**

**x = (best[u] - fee) \* rate**

**if x > best[v] + 1e-12: # 加一点 eps 防止浮点误差**

**best[v] = x**

**updated = True**

**if v == S and best[S] > V:**

**print("YES")**

**return**

**if iteration == N and updated:**

**print("YES")**

**return**

**if not updated:**

**break**

**print("NO")**

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

**main()**

**01941: The Sierpinski Fractal**

**def f(n):**

**if n == 1:**

**return [' /\\ ', '/\_\_\\']**

**t = f(n - 1)**

**x = 2 \*\* (n - 1)**

**res = [' ' \* x + u + ' ' \* x for u in t]**

**res.extend([u + u for u in t])**

**return res**

**al = [f(i) for i in range(1, 11)]**

**while True:**

**n = int(input())**

**if n == 0:**

**break**

**for u in al[n - 1]:**

**print(u)**

**print()**

**01944: Fiber Communications**

**N, P = map(int, input().split())**

**node\_one = []**

**for i in range(P):**

**Q1, Q2 = map(int, input().split())**

**node\_one.append({'start': min(Q1, Q2), 'end': max(Q1, Q2)}**

**node\_one.sort(key=lambda x: (x['start'], x['end']))**

**INF = float('inf')**

**ans = INF**

**for i in range(1, N + 1):**

**to = [0] \* (N + 1)**

**for j in range(P):**

**if node\_one[j]['end'] >= i + 1 and node\_one[j]['start'] <= i:**

**to[1] = max(to[1], node\_one[j]['start'])**

**to[node\_one[j]['end']] = N + 1**

**else:**

**to[node\_one[j]['start']] = max(to[node\_one[j]['start']], node\_one[j]['end'])**

**duandian = 0**

**result = 0**

**for j in range(1, N + 1):**

**if to[j] == 0:**

**continue**

**if to[j] > duandian:**

**if j >= duandian:**

**result += (to[j] - j)**

**else:**

**result += (to[j] - duandian)**

**duandian = to[j]**

**ans = min(ans, result)**

**print(ans)**

**02039：反反复复**

**cols = int(input())**

**encrypted = input()**

**rows = len(encrypted) // cols**

**matrix = [['' for \_ in range(cols)] for \_ in range(rows)]**

**index = 0**

**for row in range(rows):**

**if row % 2 == 0: # 从左到右填充**

**for col in range(cols):**

**matrix[row][col] = encrypted[index]**

**index += 1**

**else: # 从右到左填充**

**for col in range(cols - 1, -1, -1):**

**matrix[row][col] = encrypted[index]**

**index += 1**

**original = ''**

**for col in range(cols):**

**for row in range(rows):**

**original += matrix[row][col]**

**print(original)**

**02049: Finding Nemo**

**from collections import deque**

**N = 210**

**Size = 999999**

**INF = 1<<20**

**mv = [(1,0),(0,-1),(0,1),(-1,0)]**

**mapp = [[[0]\*2 for \_ in range(N)] for \_ in range(N)]**

**vis = [[0]\*N for \_ in range(N)]**

**def init():**

**global result**

**result = 0**

**for i in range(N):**

**for j in range(N):**

**mapp[i][j] = [0, 0]**

**vis[i][j] = 0**

**def BFS(x, y):**

**global result**

**q = deque()**

**q.append((x, y, 0))**

**vis[x][y] = 1**

**result = INF**

**while q:**

**t = q.popleft()**

**if t[0] == 0 or t[1] == 0 or t[0] > 198 or t[1] > 198:**

**result = min(result, t[2])**

**continue**

**for i in range(4):**

**f = [t[0] + mv[i][0], t[1] + mv[i][1]]**

**if i == 0 and not vis[f[0]][f[1]] and mapp[t[0]][t[1]][1] != 3:**

**f.append(t[2] + 1 if mapp[t[0]][t[1]][1] == 4 else t[2])**

**vis[f[0]][f[1]] = 1**

**q.append(tuple(f))**

**elif i == 1 and not vis[f[0]][f[1]] and mapp[f[0]][f[1]][0] != 3:**

**f.append(t[2] + 1 if mapp[f[0]][f[1]][0] == 4 else t[2])**

**vis[f[0]][f[1]] = 1**

**q.append(tuple(f))**

**elif i == 2 and not vis[f[0]][f[1]] and mapp[t[0]][t[1]][0] != 3:**

**f.append(t[2] + 1 if mapp[t[0]][t[1]][0] == 4 else t[2])**

**vis[f[0]][f[1]] = 1**

**q.append(tuple(f))**

**elif i == 3 and not vis[f[0]][f[1]] and mapp[f[0]][f[1]][1] != 3:**

**f.append(t[2] + 1 if mapp[f[0]][f[1]][1] == 4 else t[2])**

**vis[f[0]][f[1]] = 1**

**q.append(tuple(f))**

**while True:**

**m, n = map(int, input().split())**

**if m == -1 and n == -1:**

**break**

**init()**

**for \_ in range(m):**

**x, y, d, t = map(int, input().split())**

**if d:**

**for num in range(t):**

**mapp[x-1][y+num][1] = 3**

**else:**

**for num in range(t):**

**mapp[x+num][y-1][0] = 3**

**for \_ in range(n):**

**x, y, d = map(int, input().split())**

**if d:**

**mapp[x-1][y][1] = 4**

**else:**

**mapp[x][y-1][0] = 4**

**Nemo\_x, Nemo\_y = map(float, input().split())**

**xx, yy = int(Nemo\_x + 0.0001), int(Nemo\_y + 0.0001)**

**if n == 0 and m == 0:**

**print(0)**

**continue**

**if xx <= 0 or yy <= 0 or xx >= 199 or yy >= 199:**

**print(0)**

**else:**

**BFS(xx, yy)**

**print(result if result != INF else -1)**

**02092: Grandpa is Famous**

**while True:**

**n, m = map(int, input().split())**

**if n == 0 and m == 0:**

**break**

**count = [0] \* 10001**

**for \_ in range(n):**

**for player in map(int, input().split()):**

**count[player] += 1**

**max\_count = max(count)**

**second\_max\_count = max(x for x in count if x != max\_count)**

**for player, player\_count in enumerate(count):**

**if player\_count == second\_max\_count:**

**print(player, end=' ')**

**print()**

**02192: Zipper**

**from functools import lru\_cache**

**@lru\_cache**

**def f(a, b, c):**

**if len(c) == 0:**

**return True**

**else:**

**if len(a) and c[0] == a[0] and f(a[1:], b, c[1:]):**

**return True**

**elif len(b) and c[0] == b[0] and f(a, b[1:], c[1:]):**

**return True**

**else:**

**return False**

**n = int(input())**

**for \_ in range(n):**

**a, b, c = input().split()**

**x = len(c)**

**if f(a, b, c):**

**print('Data set %d: yes' % (\_ + 1))**

**else:**

**print('Data set %d: no' % (\_ + 1))**

**02226: Muddy Fields**

**def min\_boards(R, C, field):**

**hor = [[0] \* C for \_ in range(R)]**

**hor\_id = 0**

**for r in range(R):**

**c = 0**

**while c < C:**

**if field[r][c] == '\*':**

**hor\_id += 1**

**while c < C and field[r][c] == '\*':**

**hor[r][c] = hor\_id**

**c += 1**

**else:**

**c += 1**

**ver = [[0] \* C for \_ in range(R)]**

**ver\_id = 0**

**for c in range(C):**

**r = 0**

**while r < R:**

**if field[r][c] == '\*':**

**ver\_id += 1**

**while r < R and field[r][c] == '\*':**

**ver[r][c] = ver\_id**

**r += 1**

**else:**

**r += 1**

**graph = {i: set() for i in range(1, hor\_id + 1)}**

**for r in range(R):**

**for c in range(C):**

**if field[r][c] == '\*':**

**h = hor[r][c]**

**v = ver[r][c]**

**graph[h].add(v)**

**def dfs(u, seen):**

**for v in graph[u]:**

**if v in seen:**

**continue**

**seen.add(v)**

**if v not in match or dfs(match[v], seen):**

**match[v] = u**

**return True**

**return False**

**result = 0**

**for u in range(1, hor\_id + 1):**

**if dfs(u, set()):**

**result += 1**

**return result**

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

**import sys**

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

**if not data:**

**exit(0)**

**R = int(data[0])**

**C = int(data[1])**

**field = data[2:]**

**print(min\_boards(R, C, field))**

**02253: Frogger**

**import math**

**def frog\_distance(stones):**

**n = len(stones)**

**distances = [[float('inf')] \* n for \_ in range(n)]**

**for i in range(n):**

**for j in range(n):**

**if i == j:**

**distances[i][j] = 0**

**else:**

**x1, y1 = stones[i]**

**x2, y2 = stones[j]**

**distance = math.sqrt((x2 - x1) \*\* 2 + (y2 - y1) \*\* 2)**

**distances[i][j] = distance**

**for k in range(n):**

**for i in range(n):**

**for j in range(n):**

**distances[i][j] = min(distances[i][j], max(distances[i][k], distances[k][j]))**

**return distances[0][1]**

**test\_case = 1**

**while True:**

**n = int(input())**

**if n == 0:**

**break**

**stones = []**

**for \_ in range(n):**

**x, y = map(int, input().split())**

**stones.append((x, y))**

**distance = frog\_distance(stones)**

**print("Scenario #{}".format(test\_case))**

**print("Frog Distance = {:.3f}".format(distance))**

**print()**

**input()**

**test\_case += 1**

**02255：重建二叉树**

**def build\_tree(preorder, inorder):**

**if not preorder:**

**return ''**

**root = preorder[0]**

**root\_index = inorder.index(root)**

**left\_preorder = preorder[1:1 + root\_index]**

**right\_preorder = preorder[1 + root\_index:]**

**left\_inorder = inorder[:root\_index]**

**right\_inorder = inorder[root\_index + 1:]**

**left\_tree = build\_tree(left\_preorder, left\_inorder)**

**right\_tree = build\_tree(right\_preorder, right\_inorder)**

**return left\_tree + right\_tree + root**

**while True:**

**try:**

**preorder, inorder = input().split()**

**postorder = build\_tree(preorder, inorder)**

**print(postorder)**

**except EOFError:**

**break**

**02299: Ultra-QuickSort**

**def merge\_sort(lst):**

**if len(lst) <= 1:**

**return lst, 0**

**middle = len(lst) // 2**

**left, inv\_left = merge\_sort(lst[:middle])**

**right, inv\_right = merge\_sort(lst[middle:])**

**merged, inv\_merge = merge(left, right)**

**return merged, inv\_left + inv\_right + inv\_merge**

**def merge(left, right):**

**merged = []**

**inv\_count = 0**

**i = j = 0**

**while i < len(left) and j < len(right):**

**if left[i] <= right[j]:**

**merged.append(left[i])**

**i += 1**

**else:**

**merged.append(right[j])**

**j += 1**

**inv\_count += len(left) - i**

**merged += left[i:]**

**merged += right[j:]**

**return merged, inv\_count**

**while True:**

**n = int(input())**

**if n == 0:**

**break**

**lst = []**

**for \_ in range(n):**

**lst.append(int(input()))**

**\_, inversions = merge\_sort(lst)**

**print(inversions)**

**02337: Catenyms**

**import sys**

**import heapq**

**from collections import defaultdict, deque**

**def find\_eulerian\_path(words):**

**indeg = defaultdict(int)**

**outdeg = defaultdict(int)**

**adj = defaultdict(list)**

**used\_letters = set()**

**for w in words:**

**u, v = w[0], w[-1]**

**outdeg[u] += 1**

**indeg[v] += 1**

**used\_letters |= {u, v}**

**heapq.heappush(adj[u], (w, v))**

**start, plus1, minus1 = None, 0, 0**

**for ch in used\_letters:**

**o, i = outdeg[ch], indeg[ch]**

**if o == i + 1:**

**plus1 += 1**

**start = ch**

**elif i == o + 1:**

**minus1 += 1**

**elif i != o:**

**return None**

**if not ((plus1 == 1 and minus1 == 1) or (plus1 == 0 and minus1 == 0)):**

**return None**

**if start is None:**

**start = min(ch for ch in used\_letters if outdeg[ch] > 0)**

**seen = {start}**

**q = deque([start])**

**undirected = defaultdict(list)**

**for u in adj:**

**for \_, v in adj[u]:**

**undirected[u].append(v)**

**undirected[v].append(u)**

**while q:**

**u = q.popleft()**

**for v in undirected[u]:**

**if v not in seen:**

**seen.add(v)**

**q.append(v)**

**if seen != used\_letters:**

**return None**

**path = deque()**

**def dfs(u):**

**heap = adj[u]**

**while heap:**

**w, v = heapq.heappop(heap)**

**dfs(v)**

**path.appendleft(w)**

**dfs(start)**

**if len(path) != len(words):**

**return None**

**return '.'.join(path)**

**def solve():**

**input = sys.stdin.readline**

**t = int(input())**

**for \_ in range(t):**

**n = int(input())**

**words = [input().strip() for \_ in range(n)]**

**ans = find\_eulerian\_path(words)**

**print(ans if ans is not None else "\*\*\*")**

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

**sys.setrecursionlimit(1000000)**

**solve()**

**02442: Sequence**

**import sys**

**import heapq**

**def merge(arr1, arr2, n):**

**heap = []**

**visited = set()**

**heapq.heappush(heap, (arr1[0] + arr2[0], 0, 0))**

**visited.add((0, 0))**

**result = []**

**while len(result) < n:**

**s, i, j = heapq.heappop(heap)**

**result.append(s)**

**if i + 1 < n and (i + 1, j) not in visited:**

**heapq.heappush(heap, (arr1[i + 1] + arr2[j], i + 1, j))**

**visited.add((i + 1, j))**

**if j + 1 < n and (i, j + 1) not in visited:**

**heapq.heappush(heap, (arr1[i] + arr2[j + 1], i, j + 1))**

**visited.add((i, j + 1))**

**return result**

**def main():**

**input\_data = sys.stdin.read().split()**

**it = iter(input\_data)**

**T = int(next(it))**

**results = []**

**for \_ in range(T):**

**m = int(next(it))**

**n = int(next(it))**

**current = sorted(int(next(it)) for \_ in range(n))**

**for \_ in range(m - 1):**

**seq = sorted(int(next(it)) for \_ in range(n))**

**current = merge(current, seq, n)**

**results.append(" ".join(map(str, current)))**

**sys.stdout.write("\n".join(results))**

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

**main()**

**02488: A Knight's Journey**

**import heapq**

**def dfs(x,y,r,c):**

**global table,directions**

**q=[[table[x][y],(x,y)]]**

**while q:**

**way,(x,y)=heapq.heappop(q)**

**if len(way)==r\*c\*2:**

**return way**

**for dx,dy in directions:**

**nx,ny=x+dx,y+dy**

**if 0<=nx<r and 0<=ny<c and table[nx][ny] not in way:**

**heapq.heappush(q,[way+table[nx][ny],(nx,ny)])**

**return 0**

**n=int(input())**

**directions=[(-2,-1),(-2,1),(-1,2),(-1,-2),(1,-2),(1,2),(2,-1),(2,1)]**

**for \_ in range(n):**

**p,q=map(int,input().split())**

**table=[]**

**for i in range(p):**

**temp=[]**

**for j in range(q):**

**temp.append(chr(ord("A")+j)+str(i+1))**

**table.append(temp)**

**judge=False**

**for j in range(q):**

**for i in range(p):**

**if dfs(i,j,p,q):**

**judge=True**

**print("Scenario #",\_+1,":",sep="")**

**print(dfs(i,j,p,q))**

**break**

**if judge:**

**break**

**if not judge:**

**print("Scenario #",\_+1,":",sep="")**

**print("impossible")**

**if \_<n-1:**

**print()**

**02499: Binary Tree**

**def count\_moves(i, j):**

**left\_moves = 0**

**right\_moves = 0**

**while i != 1 and j != 1:**

**if i > j:**

**left\_moves += i // j**

**i %= j**

**if i == 0:**

**i = 1**

**else:**

**right\_moves += j // i # 计算可以跳跃多少次**

**j %= i # 直接更新 j，减少迭代次数**

**if j == 0: # 避免 ZeroDivisionError**

**j = 1**

**if i > 1:**

**left\_moves += i - 1**

**elif j > 1:**

**right\_moves += j - 1**

**return left\_moves, right\_moves**

**n = int(input())**

**for case\_num in range(1, n + 1):**

**i, j = map(int, input().split())**

**left, right = count\_moves(i, j)**

**print(f"Scenario #{case\_num}:")**

**print(left, right)**

**if case\_num != n:**

**print()**

**02502: Subway**

**import math**

**import heapq**

**def get\_distance(x1, y1, x2, y2):**

**return math.sqrt((x1 - x2) \*\* 2 + (y1 - y2) \*\* 2)**

**sx, sy, ex, ey = map(int, input().split())**

**min\_time = {}**

**rails = set()**

**while True:**

**try:**

**rail = list(map(int, input().split()))**

**if rail == [-1, -1]:**

**break**

**stations = [(rail[2 \* i], rail[2 \* i + 1]) for i in range(len(rail) // 2 - 1)]**

**for j, station in enumerate(stations):**

**min\_time[station] = float('inf')**

**if j != len(stations) - 1:**

**rails.add((station, stations[j + 1]))**

**rails.add((stations[j + 1], station))**

**except EOFError:**

**break # 输入结束**

**min\_time[(sx, sy)] = 0 # 起点时间为 0**

**min\_time[(ex, ey)] = float('inf') # 终点初始化为无穷大**

**min\_heap = [(0, sx, sy)] # (当前耗时, 当前x, 当前y)**

**while min\_heap:**

**curr\_time, x, y = heapq.heappop(min\_heap)**

**if curr\_time > min\_time[(x, y)]:**

**continue**

**if (x, y) == (ex, ey):**

**break**

**for position in min\_time.keys():**

**if position == (x, y):**

**continue # 自己跳过**

**nx, ny = position**

**dis = get\_distance(x, y, nx, ny)**

**rail\_factor = 4 if ((position, (x, y)) in rails or ((x, y), position) in rails) else 1**

**new\_time = curr\_time + dis / (10000 \* rail\_factor)**

**if new\_time < min\_time[position]:**

**min\_time[position] = new\_time**

**heapq.heappush(min\_heap, (new\_time, nx, ny))**

**print(round(min\_time[(ex, ey)] \* 60))**

**02524：宗教信仰**

**def init\_set(n):**

**return list(range(n))**

**def get\_father(x, father):**

**if father[x] != x:**

**father[x] = get\_father(father[x], father)**

**return father[x]**

**def join(x, y, father):**

**fx = get\_father(x, father)**

**fy = get\_father(y, father)**

**if fx == fy:**

**return**

**father[fx] = fy**

**def is\_same(x, y, father):**

**return get\_father(x, father) == get\_father(y, father)**

**def main():**

**case\_num = 0**

**while True:**

**n, m = map(int, input().split())**

**if n == 0 and m == 0:**

**break**

**count = 0**

**father = init\_set(n)**

**for \_ in range(m):**

**s1, s2 = map(int, input().split())**

**join(s1 - 1, s2 - 1, father)**

**for i in range(n):**

**if father[i] == i:**

**count += 1**

**case\_num += 1**

**print(f"Case {case\_num}: {count}")**

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

**main()**

**02694：波兰表达式**

**s = input().split()**

**def cal():**

**cur = s.pop(0)**

**if cur in "+-\*/":**

**return str(eval(cal() + cur + cal()))**

**else:**

**return cur**

**print("%.6f" % float(cal()))**

**02756：二叉树（1）**

**def common(x, y):**

**if x == y:**

**return x**

**if x < y:**

**return common(x, y//2)**

**else:**

**return common(x//2, y)**

**m, n = map(int, input().split())**

**print(common(m, n))**

**02766：最大子矩阵**

**def max\_submatrix(matrix):**

**def kadane(arr):**

**max\_end\_here = max\_so\_far = arr[0]**

**for x in arr[1:]:**

**max\_end\_here = max(x, max\_end\_here + x)**

**max\_so\_far = max(max\_so\_far, max\_end\_here)**

**return max\_so\_far**

**rows = len(matrix)**

**cols = len(matrix[0])**

**max\_sum = float('-inf')**

**for left in range(cols):**

**temp = [0] \* rows**

**for right in range(left, cols):**

**for row in range(rows):**

**temp[row] += matrix[row][right]**

**max\_sum = max(max\_sum, kadane(temp))**

**return max\_sum**

**n = int(input())**

**nums = []**

**while len(nums) < n \* n:**

**nums.extend(input().split())**

**matrix = [list(map(int, nums[i \* n:(i + 1) \* n])) for i in range(n)]**

**max\_sum = max\_submatrix(matrix)**

**print(max\_sum)**

**02773：采药**

**T, M = map(int, input().split())**

**dp = [ [0] + [0]\*T for \_ in range(M+1)]**

**t = [0]**

**v = [0]**

**for i in range(M):**

**ti, vi = map(int, input().split())**

**t.append(ti)**

**v.append(vi)**

**for i in range(1, M+1): # 外层循环（行）药草M**

**for j in range(0, T+1): # 内层循环（列）时间T**

**if j >= t[i]:**

**dp[i][j] = max(dp[i-1][j], dp[i-1][j-t[i]] + v[i])**

**else:**

**dp[i][j] = dp[i-1][j]**

**print(dp[M][T])**

**02774：木材加工**

**n, k = map(int, input().split())**

**expenditure = []**

**for \_ in range(n):**

**expenditure.append(int(input()))**

**def check(x):**

**num = 0**

**for i in range(n):**

**num += expenditure[i] // x**

**return num >= k**

**lo = 1**

**hi = max(expenditure) + 1**

**if sum(expenditure) < k:**

**print(0)**

**exit()**

**ans = 1**

**while lo < hi:**

**mid = (lo + hi) // 2**

**if check(mid):**

**ans = mid**

**lo = mid + 1**

**else:**

**hi = mid**

**print(ans)**

**02775：文件结构“图"**

**class Node:**

**def \_\_init\_\_(self,name):**

**self.name=name**

**self.dirs=[]**

**self.files=[]**

**def print\_(root,m):**

**pre='| '\*m**

**print(pre+root.name)**

**for Dir in root.dirs:**

**print\_(Dir,m+1)**

**for file in sorted(root.files):**

**print(pre+file)**

**tests,test=[],[]**

**while True:**

**s=input()**

**if s=='#':**

**break**

**elif s=='\*':**

**tests.append(test)**

**test=[]**

**else:**

**test.append(s)**

**for n,test in enumerate(tests,1):**

**root=Node('ROOT')**

**stack=[root]**

**print(f'DATA SET {n}:')**

**for i in test:**

**if i[0]=='d':**

**Dir=Node(i)**

**stack[-1].dirs.append(Dir)**

**stack.append(Dir)**

**elif i[0]=='f':**

**stack[-1].files.append(i)**

**else:**

**stack.pop()**

**print\_(root,0)**

**print()**

**02788：二叉树（2）**

**import sys**

**def count\_subtree\_nodes(m, n):**

**count = 0**

**left = m**

**right = m**

**while left <= n:**

**count += min(n, right) - left + 1**

**left \*= 2**

**right = right \* 2 + 1**

**return count**

**def main():**

**input\_stream = sys.stdin**

**for line in input\_stream:**

**m, n = map(int, line.split())**

**if m == 0 and n == 0:**

**break**

**print(count\_subtree\_nodes(m, n))**

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

**main()**

**02815：城堡问题**

**def dfs(x, y):**

**stack = [(x, y)]**

**room\_size = 0**

**while stack:**

**cx, cy = stack.pop()**

**if visited[cx][cy]:**

**continue**

**visited[cx][cy] = True**

**room\_size += 1**

**if not (castle[cx][cy] & 1): # 没有西墙**

**stack.append((cx, cy - 1))**

**if not (castle[cx][cy] & 2): # 没有北墙**

**stack.append((cx - 1, cy))**

**if not (castle[cx][cy] & 4): # 没有东墙**

**stack.append((cx, cy + 1))**

**if not (castle[cx][cy] & 8): # 没有南墙**

**stack.append((cx + 1, cy))**

**return room\_size**

**m = int(input())**

**n = int(input())**

**castle = []**

**for \_ in range(m):**

**castle.append(list(map(int, input().split())))**

**visited = [[False] \* n for \_ in range(m)]**

**room\_sizes = []**

**max\_room\_size = 0**

**for i in range(m):**

**for j in range(n):**

**if not visited[i][j]:**

**size = dfs(i, j)**

**room\_sizes.append(size)**

**max\_room\_size = max(max\_room\_size, size)**

**print(len(room\_sizes))**

**print(max\_room\_size)**

**02945：拦截导弹**

**ans=0**

**a=int(input())**

**l=list(map(int,input().split()))**

**def dfs(cnt, pos):**

**global ans**

**cnt+=1**

**if cnt>ans:**

**ans=cnt**

**if pos == a-1:**

**return**

**else:**

**for i in range(pos+1,a):**

**if l[i]<=l[pos]:**

**dfs(cnt,i)**

**return**

**for i in range(0,a):**

**dfs(0,i)**

**print(ans)**

**03151: Pots**

**def bfs(A, B, C):**

**start = (0, 0)**

**visited = set()**

**visited.add(start)**

**queue = [(start, [])]**

**while queue:**

**(a, b), actions = queue.pop(0)**

**if a == C or b == C:**

**return actions**

**next\_states = [(A, b), (a, B), (0, b), (a, 0), (min(a + b, A),\**

**max(0, a + b - A)), (max(0, a + b - B), min(a + b, B))]**

**for i in next\_states:**

**if i not in visited:**

**visited.add(i)**

**new\_actions = actions + [get\_action(a, b, i)]**

**queue.append((i, new\_actions))**

**return ["impossible"]**

**def get\_action(a, b, next\_state):**

**if next\_state == (A, b):**

**return "FILL(1)"**

**elif next\_state == (a, B):**

**return "FILL(2)"**

**elif next\_state == (0, b):**

**return "DROP(1)"**

**elif next\_state == (a, 0):**

**return "DROP(2)"**

**elif next\_state == (min(a + b, A), max(0, a + b - A)):**

**return "POUR(2,1)"**

**else:**

**return "POUR(1,2)"**

**A, B, C = map(int, input().split())**

**solution = bfs(A, B, C)**

**if solution == ["impossible"]:**

**print(solution[0])**

**else:**

**print(len(solution))**

**for i in solution:**

**print(i)**

**03447：银河贸易问题**

**from collections import defaultdict, deque**

**n = int(input())**

**graph = defaultdict(set)**

**to\_earth = set()**

**price = {}**

**for i in range(n):**

**a, b, c = input().split()**

**b = float(b)**

**price[a] = b if a not in price else max(price[a], b)**

**for x in c:**

**if x == "\*":**

**to\_earth.add(a)**

**else:**

**graph[a].add(x)**

**graph[x].add(a)**

**def bfs(start):**

**Q = deque([start])**

**visited = set()**

**visited.add(start)**

**cnt = 0**

**while Q:**

**l = len(Q)**

**for \_ in range(l):**

**f = Q.popleft()**

**if f in to\_earth:**

**return price[start] \* (0.95 \*\* cnt)**

**for x in graph[f]:**

**if x not in visited:**

**Q.append(x)**

**visited.add(x)**

**cnt += 1**

**return 0**

**ans = []**

**for planet in price.keys():**

**ans.append((bfs(planet), planet))**

**ans.sort(key=lambda x: [-x[0], x[1]])**

**print(ans[0][1])**

**03424: Candies**

**import sys**

**import threading**

**import heapq**

**def main():**

**input = sys.stdin.readline**

**N, M = map(int, input().split())**

**graph = [[] for \_ in range(N+1)]**

**for \_ in range(M):**

**A, B, c = map(int, input().split())**

**graph[A].append((B, c))**

**INF = 10\*\*30**

**dist = [INF] \* (N+1)**

**dist[1] = 0**

**pq = [(0, 1)] # (当前距离, 节点)**

**while pq:**

**d, u = heapq.heappop(pq)**

**if d > dist[u]:**

**continue**

**if u == N:**

**break # 提前退出**

**for v, w in graph[u]:**

**nd = d + w**

**if nd < dist[v]:**

**dist[v] = nd**

**heapq.heappush(pq, (nd, v))**

**# 输出从 1 到 N 的最短路距离，即为最大可实现的 x\_N - x\_1**

**print(dist[N])**

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

**threading.Thread(target=main).start()**

**03720：文本二叉树**

**class Node:**

**def \_\_init\_\_(self, x, depth):**

**self.x = x**

**self.depth = depth**

**self.lchild = None**

**self.rchild = None**

**def preorder\_traversal(self):**

**nodes = [self.x]**

**if self.lchild and self.lchild.x != '\*':**

**nodes += self.lchild.preorder\_traversal()**

**if self.rchild and self.rchild.x != '\*':**

**nodes += self.rchild.preorder\_traversal()**

**return nodes**

**def inorder\_traversal(self):**

**nodes = []**

**if self.lchild and self.lchild.x != '\*':**

**nodes += self.lchild.inorder\_traversal()**

**nodes.append(self.x)**

**if self.rchild and self.rchild.x != '\*':**

**nodes += self.rchild.inorder\_traversal()**

**return nodes**

**def postorder\_traversal(self):**

**nodes = []**

**if self.lchild and self.lchild.x != '\*':**

**nodes += self.lchild.postorder\_traversal()**

**if self.rchild and self.rchild.x != '\*':**

**nodes += self.rchild.postorder\_traversal()**

**nodes.append(self.x)**

**return nodes**

**def build\_tree():**

**n = int(input())**

**for \_ in range(n):**

**tree = []**

**stack = []**

**while True:**

**s = input()**

**if s == '0':**

**break**

**depth = len(s) - 1**

**node = Node(s[-1], depth)**

**tree.append(node)**

**while stack and tree[stack[-1]].depth >= depth:**

**stack.pop()**

**if stack: # There is a parent**

**parent = tree[stack[-1]]**

**if not parent.lchild:**

**parent.lchild = node**

**else:**

**parent.rchild = node**

**stack.append(len(tree) - 1)**

**for root in build\_tree():**

**print("".join(root.preorder\_traversal()))**

**print("".join(root.postorder\_traversal()))**

**print("".join(root.inorder\_traversal()))**

**print()**

**04077：出栈序列统计**

**def count\_sequences(n):**

**def dfs(push\_num, stack, popped):**

**nonlocal count**

**if popped == n:**

**count += 1**

**return**

**if push\_num <= n:**

**stack.append(push\_num)**

**dfs(push\_num + 1, stack, popped)**

**stack.pop()**

**if stack:**

**top = stack.pop()**

**dfs(push\_num, stack, popped + 1)**

**stack.append(top)**

**count = 0**

**dfs(1, [], 0)**

**return count**

**n = int(input())**

**print(count\_sequences(n))**

**04078：实现堆结构**

**class BinaryHeap:**

**def \_\_init\_\_(self):**

**self.\_heap = []**

**def \_perc\_up(self, i):**

**while (i - 1) // 2 >= 0:**

**parent\_idx = (i - 1) // 2**

**if self.\_heap[i] < self.\_heap[parent\_idx]:**

**self.\_heap[i], self.\_heap[parent\_idx] = (**

**self.\_heap[parent\_idx],**

**self.\_heap[i],**

**)**

**i = parent\_idx**

**def insert(self, item):**

**self.\_heap.append(item)**

**self.\_perc\_up(len(self.\_heap) - 1)**

**def \_perc\_down(self, i):**

**while 2 \* i + 1 < len(self.\_heap):**

**sm\_child = self.\_get\_min\_child(i)**

**if self.\_heap[i] > self.\_heap[sm\_child]:**

**self.\_heap[i], self.\_heap[sm\_child] = (**

**self.\_heap[sm\_child],**

**self.\_heap[i],**

**)**

**else:**

**break**

**i = sm\_child**

**def \_get\_min\_child(self, i):**

**if 2 \* i + 2 > len(self.\_heap) - 1:**

**return 2 \* i + 1**

**if self.\_heap[2 \* i + 1] < self.\_heap[2 \* i + 2]:**

**return 2 \* i + 1**

**return 2 \* i + 2**

**def delete(self):**

**self.\_heap[0], self.\_heap[-1] = self.\_heap[-1], self.\_heap[0]**

**result = self.\_heap.pop()**

**self.\_perc\_down(0)**

**return result**

**def heapify(self, not\_a\_heap):**

**self.\_heap = not\_a\_heap[:]**

**i = len(self.\_heap) // 2 - 1 # 超过中点的节点都是叶子节点**

**while i >= 0:**

**#print(f'i = {i}, {self.\_heap}')**

**self.\_perc\_down(i)**

**i = i - 1**

**n = int(input().strip())**

**bh = BinaryHeap()**

**for \_ in range(n):**

**inp = input().strip()**

**if inp[0] == '1':**

**bh.insert(int(inp.split()[1]))**

**else:**

**print(bh.delete())**

**04079：二叉搜索树**

**class TreeNode:**

**def \_\_init\_\_(self, val=0, left=None, right=None):**

**self.val = val**

**self.left = left**

**self.right = right**

**def insert\_into\_bst(root, val):**

**if root is None:**

**return TreeNode(val)**

**if val < root.val:**

**root.left = insert\_into\_bst(root.left, val)**

**elif val > root.val:**

**root.right = insert\_into\_bst(root.right, val)**

**return root**

**def preorder\_traversal(root):**

**return [root.val] + preorder\_traversal(root.left) + preorder\_traversal(root.right) if root else []**

**def preorderTraversal(root):**

**if root is None:**

**return []**

**stack = []**

**result = []**

**stack.append(root)**

**while stack:**

**node = stack.pop()**

**result.append(node.val)**

**if node.right:**

**stack.append(node.right)**

**if node.left:**

**stack.append(node.left)**

**return result**

**numbers = list(map(int, input().split()))**

**bst\_root = None**

**for num in numbers:**

**bst\_root = insert\_into\_bst(bst\_root, num)**

**print(' '.join(map(str, preorderTraversal(bst\_root))))**

**04080:Huffman编码树**

**import heapq**

**def min\_weighted\_path\_length(n, weights):**

**heapq.heapify(weights)**

**total = 0**

**while len(weights) > 1:**

**a = heapq.heappop(weights)**

**b = heapq.heappop(weights)**

**combined = a + b**

**total += combined**

**heapq.heappush(weights, combined)**

**return total**

**n = int(input())**

**weights = list(map(int, input().split()))**

**print(min\_weighted\_path\_length(n, weights))**

**04081：树的转换**

**class BinaryTreeNode:**

**def \_\_init\_\_(self):**

**self.parent = None**

**self.left = None**

**self.right = None**

**def tree\_height(root): # 计算二叉树高度**

**if not root:**

**return -1**

**else:**

**return max(tree\_height(root.left), tree\_height(root.right)) + 1**

**def original\_tree\_height(arr): # 原树高度**

**height, max\_height = 0, 0**

**for action in arr:**

**if action == 'd':**

**height += 1**

**elif action == 'u':**

**height -= 1**

**max\_height = max(max\_height, height)**

**return max\_height**

**def build\_binary\_tree(arr): # 根据输入序列建立二叉树**

**root = BinaryTreeNode()**

**current\_node = root**

**for action in arr:**

**if action == 'd':**

**current\_node.left = BinaryTreeNode()**

**current\_node.left.parent = current\_node**

**current\_node = current\_node.left**

**elif action == 'x':**

**current\_node.right = BinaryTreeNode()**

**current\_node.right.parent = current\_node.parent**

**current\_node = current\_node.right**

**elif action == 'u':**

**current\_node = current\_node.parent**

**return root**

**input\_sequence = input().replace('ud', 'x')**

**binary\_tree\_root = build\_binary\_tree(input\_sequence)**

**print(original\_tree\_height(input\_sequence), '=>', tree\_height(binary\_tree\_root))**

**04082：树的镜面映射**

**from collections import defaultdict**

**n = int(input())**

**if n == 0:**

**print()**

**exit()**

**preorder = input().split()**

**root = preorder[0][0]**

**root\_type = preorder[0][1]**

**tier = defaultdict(list)**

**tier[0].append(root)**

**nodes = [root]**

**level = 0**

**types = {root: root\_type}**

**for i in range(1, n):**

**current = preorder[i]**

**name = current[0]**

**typ = current[1]**

**types[name] = typ**

**prev\_node = nodes[-1]**

**prev\_type = types[prev\_node]**

**if prev\_type == '1':**

**level -= 1**

**else:**

**level += 1**

**nodes.append(name)**

**if name != '$':**

**tier[level].append(name)**

**sorted\_levels = sorted(tier.items(), key=lambda x: x[0])**

**result = []**

**for level, chars in sorted\_levels:**

**result.extend(reversed(chars))**

**print(' '.join(result))**

**04084：拓扑排序**

**import heapq**

**def topological\_sort(vertices, edges):**

**in\_edges = [0] \* (vertices + 1)**

**connect = [[0] \* (vertices + 1) for \_ in range(vertices + 1)]**

**for u, v in edges:**

**in\_edges[v] += 1**

**connect[u][v] += 1**

**queue = []**

**for i in range(1, vertices + 1):**

**if in\_edges[i] == 0:**

**heapq.heappush(queue, i)**

**order = []**

**while queue:**

**u = heapq.heappop(queue)**

**order.append(u)**

**for v in range(1, vertices + 1):**

**if connect[u][v] > 0:**

**in\_edges[v] -= connect[u][v]**

**if in\_edges[v] == 0:**

**heapq.heappush(queue, v)**

**if len(order) == vertices:**

**return order**

**else:**

**return None**

**vertices, num\_edges = map(int, input().split())**

**edges = []**

**for \_ in range(num\_edges):**

**u, v = map(int, input().split())**

**edges.append((u, v))**

**order = topological\_sort(vertices, edges)**

**if order:**

**for i, vertex in enumerate(order):**

**if i < len(order) - 1:**

**print(f"v{vertex}", end=" ")**

**else:**

**print(f"v{vertex}")**

**else:**

**print("No topological order exists due to a cycle in the graph.")**

**04089：电话号码**

**class TrieNode:**

**def \_\_init\_\_(self):**

**self.children = {}**

**self.is\_end\_of\_number = False**

**class Trie:**

**def \_\_init\_\_(self):**

**self.root = TrieNode()**

**def insert(self, number):**

**node = self.root**

**for digit in number:**

**if digit not in node.children:**

**node.children[digit] = TrieNode()**

**node = node.children[digit]**

**if node.is\_end\_of\_number:**

**return False**

**node.is\_end\_of\_number = True**

**return len(node.children) == 0**

**def is\_consistent(self, numbers):**

**numbers.sort(key=len)**

**for number in numbers:**

**if not self.insert(number):**

**return False**

**return True**

**def main():**

**import sys**

**input = sys.stdin.read**

**data = input().splitlines()**

**t = int(data[0])**

**index = 1**

**results = []**

**for \_ in range(t):**

**n = int(data[index]) # 当前测试样例的电话号码数量**

**index += 1**

**numbers = data[index:index + n]**

**index += n**

**trie = Trie()**

**if trie.is\_consistent(numbers):**

**results.append("YES")**

**else:**

**results.append("NO")**

**print("\n".join(results))**

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

**main()**

**04093：倒排索引查询**

**import sys**

**input = sys.stdin.read**

**data = input().split()**

**index = 0**

**N = int(data[index])**

**index += 1**

**word\_documents = []**

**for \_ in range(N):**

**ci = int(data[index])**

**index += 1**

**documents = sorted(map(int, data[index:index + ci]))**

**index += ci**

**word\_documents.append(documents)**

**M = int(data[index])**

**index += 1**

**results = []**

**for \_ in range(M):**

**query = list(map(int, data[index:index + N]))**

**index += N**

**included\_docs = []**

**excluded\_docs = set()**

**for i in range(N):**

**if query[i] == 1:**

**included\_docs.append(word\_documents[i])**

**elif query[i] == -1:**

**excluded\_docs.update(word\_documents[i])**

**if included\_docs:**

**result\_set = set(included\_docs[0])**

**for docs in included\_docs[1:]:**

**result\_set.intersection\_update(docs)**

**result\_set.difference\_update(excluded\_docs)**

**final\_docs = sorted(result\_set)**

**results.append(" ".join(map(str, final\_docs)) if final\_docs else "NOT FOUND")**

**else:**

**results.append("NOT FOUND")**

**for result in results:**

**print(result)**

**04117：简单的整数划分问题**

**def partition\_count(n):**

**dp = [[0] \* (n + 1) for \_ in range(n + 1)]**

**for j in range(n + 1):**

**dp[0][j] = 1**

**for i in range(1, n + 1):**

**for j in range(1, n + 1):**

**if i < j:**

**dp[i][j] = dp[i][i]**

**else:**

**dp[i][j] = dp[i][j - 1] + dp[i - j][j]**

**return dp[n][n]**

**try:**

**while True:**

**N = int(input())**

**print(partition\_count(N))**

**except EOFError:**

**pass**

**04130: Saving Tang Monk**

**import sys**

**import heapq**

**from collections import deque**

**def solve():**

**data = sys.stdin.read().splitlines()**

**if not data:**

**return**

**line\_index = 0**

**directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]**

**results = []**

**while line\_index < len(data):**

**if not data[line\_index].strip():**

**line\_index += 1**

**continue**

**parts = data[line\_index].split()**

**line\_index += 1**

**n = int(parts[0])**

**m = int(parts[1])**

**if n == 0 and m == 0:**

**break**

**g = []**

**xs = ys = xe = ye = None**

**snake\_index = {}**

**snake\_count = 0**

**for i in range(n):**

**row = list(data[line\_index].strip())**

**line\_index += 1**

**for j, ch in enumerate(row):**

**if ch == 'K':**

**xs, ys = i, j**

**elif ch == 'T':**

**xe, ye = i, j**

**elif ch == 'S':**

**snake\_index[(i, j)] = snake\_count**

**snake\_count += 1**

**g.append(row)**

**reachable = [[False]\*n for \_ in range(n)]**

**flag = [False]\*(m+1)**

**q = deque([(xs, ys)])**

**reachable[xs][ys] = True**

**while q:**

**x0, y0 = q.popleft()**

**for dx, dy in directions:**

**x1, y1 = x0 + dx, y0 + dy**

**if not (0 <= x1 < n and 0 <= y1 < n):**

**continue**

**if g[x1][y1] == '#':**

**continue**

**if not reachable[x1][y1]:**

**reachable[x1][y1] = True**

**q.append((x1, y1))**

**if g[x1][y1].isdigit():**

**key\_val = int(g[x1][y1])**

**if 1 <= key\_val <= m:**

**flag[key\_val] = True**

**elif x1 == xe and y1 == ye:**

**flag[0] = True**

**if not (flag[0] and all(flag[1:])):**

**results.append("impossible")**

**continue**

**def encode(keys, smask):**

**return keys \* (1 << snake\_count) + smask**

**visited = [[{} for \_ in range(n)] for \_ in range(n)]**

**init\_state = encode(0, 0)**

**visited[xs][ys][init\_state] = 0**

**heap = [(0, xs, ys, 0, 0)] # (耗时, x, y, keys, snake\_mask)**

**ans = -1**

**while heap:**

**t, x, y, keys, smask = heapq.heappop(heap)**

**state\_code = encode(keys, smask)**

**if visited[x][y].get(state\_code, float('inf')) < t:**

**continue**

**if x == xe and y == ye and keys == m:**

**ans = t**

**break**

**for dx, dy in directions:**

**nx, ny = x + dx, y + dy**

**if not (0 <= nx < n and 0 <= ny < n):**

**continue**

**if g[nx][ny] == '#':**

**continue**

**nkeys = keys**

**nsmask = smask**

**nt = t + 1 # 每走一步耗时1分钟**

**cell = g[nx][ny]**

**if cell == 'S':**

**idx = snake\_index[(nx, ny)]**

**if not (smask & (1 << idx)):**

**nt += 1**

**nsmask = smask | (1 << idx)**

**if cell.isdigit():**

**k = int(cell)**

**if keys < m and k == keys + 1:**

**nkeys = keys + 1**

**new\_state = encode(nkeys, nsmask)**

**if new\_state not in visited[nx][ny] or nt < visited[nx][ny][new\_state]:**

**visited[nx][ny][new\_state] = nt**

**heapq.heappush(heap, (nt, nx, ny, nkeys, nsmask))**

**results.append("impossible" if ans == -1 else str(ans))**

**sys.stdout.write("\n".join(results))**

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

**solve()**

**04135：月度开销**

**def minMaxMonthlyExpense(N, M, expenses):**

**def can\_split(max\_expense):**

**months = 1 # 记录当前使用的月份数**

**current\_sum = 0 # 当前月的开销**

**for cost in expenses:**

**if current\_sum + cost > max\_expense:**

**months += 1**

**if months > M:**

**return False**

**current\_sum = cost**

**else:**

**current\_sum += cost**

**return True**

**left, right = max(expenses), sum(expenses) + 1**

**ans = -1**

**while left < right: # 二分查找最小的 "最大月度开销"**

**mid = (left + right) // 2**

**if can\_split(mid):**

**ans = mid # 记录可行的 `mid`**

**right = mid # 继续尝试更小的值**

**else:**

**left = mid + 1**

**return ans**

**N, M = map(int, input().split())**

**expenses = [int(input()) for \_ in range(N)]**

**print(minMaxMonthlyExpense(N, M, expenses))**

**04136：矩形分割**

**R = int(input())**

**a = [0] \* R**

**n = int(input())**

**for i in range(n):**

**L, T, W, H = map(int, input().split())**

**for j in range(L, L + W):**

**a[j] += H**

**le = 0**

**ri = R**

**while True:**

**if le >= ri:**

**break**

**else:**

**mi = (le + ri) // 2**

**x = sum(a[:mi])**

**y = sum(a[mi:])**

**if x >= y:**

**ri = mi**

**else:**

**le = mi + 1**

**while True:**

**if le == R:**

**print(le)**

**break**

**elif a[le] == 0: #如果当前位置的小矩形高度为0，则将位置向右移动。**

**le += 1**

**else:**

**print(le)**

**break**

**04137：最小新整数**

**def removeKDigits(num, k):**

**stack = []**

**for digit in num:**

**while k and stack and stack[-1] > digit:**

**stack.pop()**

**k -= 1**

**stack.append(digit)**

**while k:**

**stack.pop()**

**k -= 1**

**return int(''.join(stack))**

**t = int(input())**

**results = []**

**for \_ in range(t):**

**n, k = input().split()**

**results.append(removeKDigits(n, int(k)))**

**for result in results:**

**print(result)**

**04140：方程求解**

**def f(x):**

**return x\*\*3 - 5\*x\*\*2 + 10\*x - 80**

**def df(x):**

**return 3\*x\*\*2 - 10\*x + 10**

**def newton\_method(x0, tol=1e-9, max\_iter=1000):**

**x = x0**

**for \_ in range(max\_iter):**

**fx = f(x)**

**dfx = df(x)**

**if abs(fx) < tol:**

**return x**

**x -= fx / dfx # 牛顿迭代公式**

**return x**

**root2 = newton\_method(3)**

**print(f"{root2:.9f}")**

**04143：和为给定数**

**n=int(input())-1;m=0**

**A=sorted(map(int,input().split()))**

**s=int(input())**

**while m<n:**

**while m<n and A[m]+A[n]>s:n-=1**

**while m<n and A[m]+A[n]<s:m+=1**

**if m<n and A[m]+A[n]==s:print(A[m],A[n]);break**

**else:print("No")**

**05343：用队列对扑克牌排序**

**from collections import deque**

**n = int(input())**

**queues = [deque() for \_ in range(9)]**

**cards = deque(list(input().split()))**

**while cards:**

**card = cards.popleft()**

**queues[int(card[1])-1].append(card)**

**qs = {'A': deque(), 'B': deque(), 'C': deque(), 'D': deque()}**

**for i in range(9):**

**tmp = []**

**while queues[i]:**

**card = queues[i].popleft()**

**qs[card[0]].append(card)**

**tmp.append(card)**

**print(f'Queue{i+1}:'+' '.join(tmp))**

**result = []**

**for char in qs.keys():**

**tmp = []**

**while qs[char]:**

**card = qs[char].popleft()**

**result.append(card)**

**tmp.append(card)**

**print(f'Queue{char}:' + ' '.join(tmp))**

**print(\*result)**

**05344：最后的最后**

**from collections import deque**

**n,k=map(int,input().split())**

**queue=deque(i for i in range(1,n+1))**

**flag=k**

**res=[]**

**# 1 2 3 4 5 6 7 8 9 10**

**while len(queue)>=2:**

**a=queue.popleft()**

**queue.append(a)**

**if k-2!=0:**

**for \_ in range(k-2):**

**a = queue.popleft()**

**queue.append(a)**

**b=queue.popleft()**

**res.append(b)**

**res\_new=[str(i) for i in res]**

**print(" ".join(res\_new))**

**05345：位查询**

**n,m = map(int, input().split())**

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

**for \_ in range(m):**

**op, i = input().split()**

**i = int(i)**

**if op == 'C':**

**for j in range(n):**

**a[j] += i**

**a[j] %= 65535**

**elif op == 'Q':**

**cnt = 0**

**for j in a:**

**s = bin(j)[2:][::-1]**

**if i < len(s) and s[i] == '1':**

**cnt += 1**

**print(cnt)**

**05430：表达式•表达式树•表达式求值**

**class Node:**

**def \_\_init\_\_(self, x):**

**self.value = x**

**self.left = None**

**self.right = None**

**def priority(x):**

**if x == '\*' or x == '/':**

**return 2**

**if x == '+' or x == '-':**

**return 1**

**return 0**

**def infix\_trans(infix):**

**postfix = []**

**op\_stack = []**

**for char in infix:**

**if char.isalpha():**

**postfix.append(char)**

**else:**

**if char == '(':**

**op\_stack.append(char)**

**elif char == ')':**

**while op\_stack and op\_stack[-1] != '(':**

**postfix.append(op\_stack.pop())**

**op\_stack.pop()**

**else:**

**while op\_stack and priority(op\_stack[-1]) >= priority(char) and op\_stack[-1] != '(':**

**postfix.append(op\_stack.pop())**

**op\_stack.append(char)**

**while op\_stack:**

**postfix.append(op\_stack.pop())**

**return postfix**

**def build\_tree(postfix):**

**stack = []**

**for item in postfix:**

**if item in '+-\*/':**

**node = Node(item)**

**node.right = stack.pop()**

**node.left = stack.pop()**

**else:**

**node = Node(item)**

**stack.append(node)**

**return stack[0]**

**def get\_val(expr\_tree, var\_vals):**

**if expr\_tree.value in '+-\*/':**

**operator = {'+': op.add, '-': op.sub, '\*': op.mul, '/': op.floordiv}**

**return operator[expr\_tree.value](get\_val(expr\_tree.left, var\_vals), get\_val(expr\_tree.right, var\_vals))**

**else:**

**return var\_vals[expr\_tree.value]**

**def getDepth(tree\_root):**

**#return max([self.child[i].getDepth() if self.child[i] else 0 for i in range(2)]) + 1**

**left\_depth = getDepth(tree\_root.left) if tree\_root.left else 0**

**right\_depth = getDepth(tree\_root.right) if tree\_root.right else 0**

**return max(left\_depth, right\_depth) + 1**

**def printExpressionTree(tree\_root, d): # d means total depth**

**graph = [" "\*(2\*\*d-1) + tree\_root.value + " "\*(2\*\*d-1)]**

**graph.append(" "\*(2\*\*d-2) + ("/" if tree\_root.left else " ")**

**+ " " + ("\\" if tree\_root.right else " ") + " "\*(2\*\*d-2))**

**if d == 0:**

**return tree\_root.value**

**d -= 1**

**if tree\_root.left:**

**left = printExpressionTree(tree\_root.left, d)**

**else:**

**left = [" "\*(2\*\*(d+1)-1)]\*(2\*d+1)**

**right = printExpressionTree(tree\_root.right, d) if tree\_root.right else [**

**" "\*(2\*\*(d+1)-1)]\*(2\*d+1)**

**for i in range(2\*d+1):**

**graph.append(left[i] + " " + right[i])**

**return graph**

**infix = input().strip()**

**n = int(input())**

**vars\_vals = {}**

**for i in range(n):**

**line = input().split()**

**vars\_vals[line[0]] = int(line[1])**

**postfix = infix\_trans(infix)**

**tree\_root = build\_tree(postfix)**

**print(''.join(str(x) for x in postfix))**

**expression\_value = get\_val(tree\_root, vars\_vals)**

**for line in printExpressionTree(tree\_root, getDepth(tree\_root)-1):**

**print(line.rstrip())**

**print(expression\_value)**

**05442：兔子与星空**

**import heapq**

**def prim(graph, start):**

**mst = []**

**used = set([start])**

**edges = [ (cost, start, to)**

**for to, cost in graph[start].items() ]**

**heapq.heapify(edges)**

**while edges:**

**cost, frm, to = heapq.heappop(edges)**

**if to not in used:**

**used.add(to)**

**mst.append((frm, to, cost))**

**for to\_next, cost2 in graph[to].items():**

**if to\_next not in used:**

**heapq.heappush(edges, (cost2, to, to\_next))**

**return mst**

**def solve():**

**n = int(input())**

**graph = {chr(i+65): {} for i in range(n)}**

**for i in range(n-1):**

**data = input().split()**

**star = data[0]**

**m = int(data[1])**

**for j in range(m):**

**to\_star = data[2+j\*2]**

**cost = int(data[3+j\*2])**

**graph[star][to\_star] = cost**

**graph[to\_star][star] = cost**

**mst = prim(graph, 'A')**

**print(sum(x[2] for x in mst))**

**solve()**

**05443：兔子与樱花**

**import heapq**

**from collections import defaultdict**

**p = int(input())**

**points = [input().strip() for \_ in range(p)]**

**maps = defaultdict(list)**

**for \_ in range(int(input())):**

**a, b, d = input().split()**

**d = int(d)**

**maps[a].append((b, d))**

**maps[b].append((a, d))**

**def dijkstra(src, dst):**

**INF = float('inf')**

**dist = {point: INF for point in points}**

**path = {point: "" for point in points}**

**dist[src] = 0**

**path[src] = src**

**pq = [(0, src)]**

**while pq:**

**d, u = heapq.heappop(pq)**

**if d > dist[u]:**

**continue**

**if u == dst:**

**break**

**for v, w in maps[u]:**

**nd = d + w**

**if nd < dist[v]:**

**dist[v] = nd**

**path[v] = path[u] + f"->({w})->" + v**

**heapq.heappush(pq, (nd, v))**

**return path[dst]**

**for \_ in range(int(input())):**

**s, t = input().split()**

**print(dijkstra(s, t))**

**05455：二又搜索树的层次遍历**

**class TreeNode:**

**def \_\_init\_\_(self, value):**

**self.value = value**

**self.left = None**

**self.right = None**

**def insert(node, value):**

**if node is None:**

**return TreeNode(value)**

**if value < node.value:**

**node.left = insert(node.left, value)**

**elif value > node.value:**

**node.right = insert(node.right, value)**

**return node**

**def level\_order\_traversal(root):**

**queue = [root]**

**traversal = []**

**while queue:**

**node = queue.pop(0)**

**traversal.append(node.value)**

**if node.left:**

**queue.append(node.left)**

**if node.right:**

**queue.append(node.right)**

**return traversal**

**numbers = list(map(int, input().strip().split()))**

**numbers = list(dict.fromkeys(numbers)) # remove duplicates**

**root = None**

**for number in numbers:**

**root = insert(root, number)**

**traversal = level\_order\_traversal(root)**

**print(' '.join(map(str, traversal)))**

**05467：多项式加法**

**from collections import defaultdict**

**def add(a):**

**i=0**

**while 1:**

**m,n=a[i],a[i+1]**

**if n<0:**

**break**

**res[n]+=m**

**i+=2**

**for \_ in range(int(input())):**

**res=defaultdict(int)**

**add(list(map(int,input().split())))**

**add(list(map(int,input().split())))**

**for i in sorted(res,reverse=True):**

**if res[i]!=0:**

**print(f'[ {res[i]} {i} ] ',end='')**

**print()**

**05902：双端队列**

**class Node:**

**def \_\_init\_\_(self, value=None):**

**self.value = value**

**self.next = None**

**self.prev = None**

**class MyDeque:**

**def \_\_init\_\_(self):**

**self.head = None**

**self.tail = None**

**def isEmpty(self):**

**return self.head is None**

**def append(self, value):**

**new\_node = Node(value)**

**if self.isEmpty():**

**self.head = self.tail = new\_node**

**else:**

**self.tail.next = new\_node**

**new\_node.prev = self.tail**

**self.tail = new\_node**

**def appendleft(self, value):**

**new\_node = Node(value)**

**if self.isEmpty():**

**self.head = self.tail = new\_node**

**else:**

**new\_node.next = self.head**

**self.head.prev = new\_node**

**self.head = new\_node**

**def pop(self):**

**if self.isEmpty():**

**return None**

**ret\_value = self.tail.value**

**if self.head == self.tail:**

**self.head = self.tail = None**

**else:**

**self.tail = self.tail.prev**

**self.tail.next = None**

**return ret\_value**

**def popleft(self):**

**if self.isEmpty():**

**return None**

**ret\_value = self.head.value**

**if self.head == self.tail:**

**self.head = self.tail = None**

**else:**

**self.head = self.head.next**

**self.head.prev = None**

**return ret\_value**

**def printDeque(self):**

**elements = []**

**current = self.head**

**while current:**

**elements.append(current.value)**

**current = current.next**

**return elements**

**t = int(input()) # 测试数据的组数**

**for \_ in range(t):**

**n = int(input()) # 操作次数**

**my\_deque = MyDeque()**

**for \_ in range(n):**

**parts = list(map(int, input().split()))**

**if parts[0] == 1: # 进队操作**

**my\_deque.append(parts[1])**

**elif parts[0] == 2: # 出队操作**

**if parts[1] == 0:**

**my\_deque.popleft()**

**else:**

**my\_deque.pop()**

**if my\_deque.isEmpty():**

**print("NULL")**

**else:**

**print(' '.join(map(str, my\_deque.printDeque())))**

**05907：二叉树的操作**

**def swap(x, y):**

**tree[loc[x][0]][loc[x][1]] = y**

**tree[loc[y][0]][loc[y][1]] = x**

**loc[x], loc[y] = loc[y], loc[x]**

**for \_ in range(int(input())):**

**n, m = map(int, input().split())**

**tree = {}**

**loc = [[] for \_ in range(n)]**

**for \_ in range(n):**

**a, b, c = map(int, input().split())**

**tree[a] = [b, c]**

**loc[b], loc[c] = [a, 0], [a, 1]**

**for \_ in range(m):**

**op = list(map(int, input().split()))**

**if op[0] == 1:**

**swap(op[1], op[2])**

**else:**

**cur = op[1]**

**while tree[cur][0] != -1:**

**cur = tree[cur][0]**

**print(cur)**

**06250：字符串最大跨距**

**def find(s, pat):**

**nex = [0]**

**for i, p in enumerate(pat[1:], 1):**

**tmp = nex[i-1]**

**while True:**

**if p == pat[tmp]:**

**nex.append(tmp+1)**

**break**

**elif tmp:**

**tmp = nex[tmp-1]**

**else:**

**nex.append(0)**

**break**

**j = 0**

**for i, char in enumerate(s):**

**while True:**

**if char == pat[j]:**

**j += 1**

**if j == len(pat):**

**return i**

**break**

**elif j:**

**j -= nex[j]**

**else:**

**break**

**s, p1, p2 = input().split(',')**

**try:**

**assert((ans := len(s)-find(s, p1)-find(s[::-1], p2[::-1])-2) >= 0)**

**print(ans)**

**except (TypeError, AssertionError):**

**print(-1)**

**06263：布尔表达式**

**while True:**

**try:**

**s=input()**

**except EOFError:**

**break**

**s=s.replace('V','True').replace('F','False')**

**s=s.replace('&',' and ').replace('|',' or ').replace('!',' not ')**

**if eval(s):**

**print('V')**

**else:**

**print('F')**

**06364：牛的选举**

**n, k = map(int, input().split())**

**cows = []**

**for i in range(n):**

**a, b = map(int, input().split())**

**cows.append((a, b, i + 1))**

**cows.sort(key=lambda x: x[0], reverse=True)**

**second\_round\_cows = cows[:k]**

**second\_round\_cows.sort(key=lambda x: x[1], reverse=True)**

**print(second\_round\_cows[0][2])**

**06640：倒排索引**

**from collections import defaultdict**

**def main():**

**n = int(input())**

**index = 1**

**inverted\_index = defaultdict(set) # 构建倒排索引**

**for i in range(1, n + 1):**

**parts = input().split()**

**doc\_id = i**

**num\_words = int(parts[0])**

**words = parts[1:num\_words + 1]**

**for word in words:**

**inverted\_index[word].add(doc\_id)**

**m = int(input())**

**results = []**

**for \_ in range(m):**

**query = input()**

**if query in inverted\_index:**

**results.append(" ".join(map(str, sorted(list(inverted\_index[query])))))**

**else:**

**results.append("NOT FOUND")**

**for result in results:**

**print(result)**

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

**main()**

**06646：二叉树的深度**

**ans, l, r = 1, [-1], [-1]**

**def dfs(n, count):**

**global ans, l, r**

**if l[n] != -1:**

**dfs(l[n], count + 1)**

**if r[n] != -1:**

**dfs(r[n], count + 1)**

**ans = max(ans, count)**

**n = int(input())**

**for i in range(n):**

**a, b = map(int, input().split())**

**l.append(a)**

**r.append(b)**

**dfs(1, 1)**

**print(ans)**

**06648: Sequence**

**import heapq**

**t = int(input())**

**for \_ in range(t):**

**m, n = map(int, input().split())**

**seq1 = sorted(map(int, input().split()))**

**for \_ in range(m - 1):**

**seq2 = sorted(map(int, input().split()))**

**min\_heap = [(seq1[i] + seq2[0], i, 0) for i in range(n)]**

**heapq.heapify(min\_heap)**

**result = []**

**for \_ in range(n):**

**current\_sum, i, j = heapq.heappop(min\_heap)**

**result.append(current\_sum)**

**if j + 1 < len(seq2):**

**heapq.heappush(min\_heap, (seq1[i] + seq2[j + 1], i, j + 1))**

**seq1 = result**

**print(\*seq1)**

**07161：森林的带度数层次序列存储**

**from collections import deque**

**class Node:**

**def \_\_init\_\_(self):**

**self.value=None**

**self.degree=0**

**self.childs=[]**

**def build():**

**node=Node()**

**node.value=l.pop(0)**

**node.degree=int(l.pop(0))**

**return node**

**def Tree():**

**root=build()**

**q=deque([root])**

**while q:**

**node=q.popleft()**

**for i in range(node.degree):**

**child=build()**

**node.childs.append(child)**

**q.append(child)**

**return root**

**def lastorder(tree):**

**for child in tree.childs:**

**lastorder(child)**

**print(tree.value,end=" ")**

**n=int(input())**

**for \_ in range(n):**

**l=list(input().split())**

**tree=Tree()**

**lastorder(tree)**

**07206：我是最快的马**

**def solve():**

**import sys**

**from collections import deque**

**start\_r, start\_c = map(int, input().split())**

**end\_r, end\_c = map(int, input().split())**

**M = int(input())**

**obstacles = set()**

**for \_ in range(M):**

**r, c = map(int, input().split())**

**obstacles.add((r, c))**

**MIN, MAX = 0, 10**

**moves = [ (2, 1, (1, 0)), (2, -1, (1, 0)),**

**(-2, 1, (-1, 0)), (-2, -1, (-1, 0)),**

**(1, 2, (0, 1)), (-1, 2, (0, 1)),**

**(1, -2, (0, -1)), (-1, -2, (0, -1))]**

**def in\_bounds(pos):**

**r, c = pos**

**return MIN <= r <= MAX and MIN <= c <= MAX**

**start = (start\_r, start\_c)**

**end = (end\_r, end\_c)**

**dist, ways, prev = {}, {}, {}**

**dist[start] = 0**

**ways[start] = 1**

**prev[start] = None**

**q = deque([start])**

**while q:**

**cur = q.popleft()**

**r, c = cur**

**for dx, dy, (br, bc) in moves:**

**block = (r + br, c + bc)**

**if in\_bounds(block) and block in obstacles:**

**continue**

**newPos = (r + dx, c + dy)**

**if not in\_bounds(newPos) or newPos in obstacles:**

**continue**

**nd = dist[cur] + 1**

**if newPos not in dist:**

**dist[newPos] = nd**

**ways[newPos] = ways[cur]**

**prev[newPos] = cur if ways[cur] == 1 else None**

**q.append(newPos)**

**elif nd == dist[newPos]:**

**ways[newPos] += ways[cur]**

**prev[newPos] = None**

**if end not in dist:**

**print("无解")**

**return**

**if ways[end] == 1:**

**path = []**

**cur = end**

**while cur is not None:**

**path.append(cur)**

**cur = prev[cur]**

**path.reverse()**

**route\_str = "-".join("({},{})".format(r, c) for r, c in path)**

**print(route\_str)**

**else:**

**print(str(ways[end]))**

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

**solve()**

**E07218：献给阿尔吉侬的花束07297：神奇的幻方**

**from collections import deque**

**def solve\_maze():**

**T = int(input())**

**for \_ in range(T):**

**R, C = map(int, input().split())**

**maze = [list(input().strip()) for \_ in range(R)]**

**for i in range(R):**

**for j in range(C):**

**if maze[i][j] == 'S':**

**start = (i, j)**

**if maze[i][j] == 'E':**

**end = (i, j)**

**queue = deque()**

**visited = [[False] \* C for \_ in range(R)]**

**queue.append((start[0], start[1], 0)) # (row, col, distance)**

**visited[start[0]][start[1]] = True**

**found = False**

**while queue:**

**x, y, dist = queue.popleft()**

**if (x, y) == end:**

**print(dist)**

**found = True**

**break**

**for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:**

**nx, ny = x + dx, y + dy**

**if 0 <= nx < R and 0 <= ny < C:**

**if not visited[nx][ny] and maze[nx][ny] != '#':**

**visited[nx][ny] = True**

**queue.append((nx, ny, dist + 1))**

**if not found:**

**print("oop!")**

**solve\_maze()**

**07734：虫子的生活**

**class UnionFind:**

**def \_\_init\_\_(self, size):**

**self.parent = list(range(size))**

**def find(self, x):**

**if x != self.parent[x]:**

**self.parent[x] = self.find(self.parent[x])**

**return self.parent[x]**

**def union(self, x, y):**

**rootX = self.find(x)**

**rootY = self.find(y)**

**if rootX != rootY:**

**self.parent[rootY] = rootX**

**def is\_connected(self, x, y):**

**return self.find(x) == self.find(y)**

**def solve\_bug\_life(scenarios):**

**for i in range(1, scenarios + 1):**

**n, m = map(int, input().split())**

**uf = UnionFind(2 \* n + 1)**

**suspicious = False**

**for \_ in range(m):**

**u, v = map(int, input().split())**

**if suspicious:**

**continue**

**if uf.is\_connected(u, v):**

**suspicious = True**

**uf.union(u, v + n) # 将u的一种性别与v的另一种性别关联**

**uf.union(u + n, v) # 同理**

**print(f'Scenario #{i}:')**

**print('Suspicious bugs found!' if suspicious else 'No suspicious bugs found!')**

**print()**

**scenarios = int(input())**

**solve\_bug\_life(scenarios)**

**07735：道路**

**import heapq**

**from collections import defaultdict**

**MAX\_COINS = int(input()) # 最大金币数**

**CITY\_COUNT = int(input()) # 城市数目**

**ROAD\_COUNT = int(input())**

**roads = defaultdict(list)**

**for \_ in range(ROAD\_COUNT):**

**start, end, length, money = map(int, input().split())**

**start, end = start - 1, end - 1**

**roads[start].append((end, length, money))**

**def bfs(start, end, max\_coins):**

**queue = [(0, max\_coins, start)]**

**visited = set()**

**while queue:**

**distance, coins, city = heapq.heappop(queue)**

**if city == end:**

**return distance**

**visited.add((city, coins))**

**for next\_city, road\_length, road\_money in roads[city]:**

**if coins >= road\_money:**

**new\_distance = distance + road\_length**

**if (next\_city, coins - road\_money) not in visited:**

**heapq.heappush(queue, (new\_distance, coins - road\_money, next\_city))**

**return -1**

**print(bfs(0, CITY\_COUNT - 1, MAX\_COINS))**

**07745：整数奇偶排序**

**numbers = list(map(int, input().split()))**

**odd\_numbers = [num for num in numbers if num % 2 == 1]**

**even\_numbers = [num for num in numbers if num % 2 == 0]**

**odd\_numbers.sort(reverse=True)**

**even\_numbers.sort()**

**sorted\_numbers = odd\_numbers + even\_numbers**

**print(' '.join(map(str, sorted\_numbers)))**

**07576：败方树**

**from collections import deque**

**from dataclasses import dataclass**

**@dataclass**

**class TreeNode:**

**value: int**

**min\_win: int**

**left: 'TreeNode' = None**

**right: 'TreeNode' = None**

**def build\_tree(values):**

**stack = deque(TreeNode(value, value) for value in values)**

**while len(stack) > 1:**

**left\_node = stack.popleft()**

**right\_node = stack.popleft()**

**new\_node = TreeNode(max(left\_node.min\_win, right\_node.min\_win),**

**min(left\_node.min\_win, right\_node.min\_win))**

**new\_node.left, new\_node.right = left\_node, right\_node**

**stack.append(new\_node)**

**root = TreeNode(stack[0].min\_win, stack[0].min\_win)**

**root.left = stack[0]**

**return root**

**def show(n, root):**

**stack = deque([root])**

**result = []**

**while stack:**

**if len(result) == n:**

**print(\*result)**

**return**

**current\_node = stack.popleft()**

**result.append(current\_node.value)**

**if current\_node.left:**

**stack.append(current\_node.left)**

**if current\_node.right:**

**stack.append(current\_node.right)**

**n, m = map(int, input().split())**

**initial\_values = list(map(int, input().split()))**

**root = build\_tree(initial\_values)**

**show(n, root)**

**for \_ in range(m):**

**position, value = map(int, input().split())**

**initial\_values[position] = value**

**root = build\_tree(initial\_values)**

**show(n, root)**

**08581：扩展二叉树**

**def build\_tree(preorder):**

**if not preorder or preorder[0] == '.':**

**return None, preorder[1:]**

**root = preorder[0]**

**left, preorder = build\_tree(preorder[1:])**

**right, preorder = build\_tree(preorder)**

**return (root, left, right), preorder**

**def inorder(tree):**

**if tree is None:**

**return ''**

**root, left, right = tree**

**return inorder(left) + root + inorder(right)**

**def postorder(tree):**

**if tree is None:**

**return ''**

**root, left, right = tree**

**return postorder(left) + postorder(right) + root**

**preorder = input().strip()**

**tree, \_ = build\_tree(preorder)**

**print(inorder(tree))**

**print(postorder(tree))**

**08758:2的幂次方表示**

**from bisect import \***

**n=int(input())**

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

**sorted\_list=[]**

**cnt=0**

**for num in a:**

**pos=bisect\_left(sorted\_list,num)**

**cnt+=pos**

**insort\_left(sorted\_list,num)**

**print(cnt)**

**09201:Freda的越野跑**

**from bisect import \***

**n=int(input())**

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

**sorted\_list=[]**

**cnt=0**

**for num in a:**

**pos=bisect\_left(sorted\_list,num)**

**cnt+=pos**

**insort\_left(sorted\_list,num)**

**print(cnt)**

**09202：舰队、海域出击!**

**from collections import deque,defaultdict**

**def topo\_sort(graph):**

**in\_degree={u:0 for u in range(1,n+1)}**

**for u in graph:**

**for v in graph[u]:**

**in\_degree[v]+=1**

**q=deque([u for u in in\_degree if in\_degree[u]==0])**

**topo\_order=[]**

**while q:**

**u=q.popleft()**

**topo\_order.append(u)**

**for v in graph[u]:**

**in\_degree[v]-=1**

**if in\_degree[v]==0:**

**q.append(v)**

**if len(topo\_order)!=len(graph):**

**return 'Yes'**

**return 'No'**

**for \_ in range(int(input())):**

**n,m=map(int,input().split())**

**graph=defaultdict(list)**

**for \_ in range(m):**

**u,v=map(int,input().split())**

**graph[u].append(v)**

**print(topo\_sort(graph))**

**14683：合并果子**

**import heapq**

**n = int(input())**

**l = list(map(int, input().split()))**

**heapq.heapify(l)**

**ans = 0**

**while len(l) > 1:**

**a = heapq.heappop(l)**

**b = heapq.heappop(l)**

**ans += a + b**

**heapq.heappush(l, a + b)**

**print(ans)**

**17968：整型关键字的散列映射**

**def insert\_hash\_table(keys, M):**

**table = [0.5] \* M # 用 0.5 表示空位**

**result = []**

**for key in keys:**

**index = key % M**

**i = index**

**while True:**

**if table[i] == 0.5 or table[i] == key:**

**result.append(i)**

**table[i] = key**

**break**

**i = (i + 1) % M**

**return result**

**import sys**

**input = sys.stdin.read**

**data = input().split()**

**N = int(data[0])**

**M = int(data[1])**

**keys = list(map(int, data[2:2 + N]))**

**positions = insert\_hash\_table(keys, M)**

**print(\*positions)**

**M17975：用二次探查法建立散列表**

**import sys**

**input = sys.stdin.read**

**data = input().split()**

**index = 0**

**n = int(data[index])**

**index += 1**

**m = int(data[index])**

**index += 1**

**num\_list = [int(i) for i in data[index:index+n]]**

**M18250：冰阔落1**

**def find(x):**

**if parent[x] != x:**

**parent[x] = find(parent[x])**

**return parent[x]**

**def union(x, y):**

**root\_x = find(x)**

**root\_y = find(y)**

**if root\_x != root\_y:**

**parent[root\_y] = root\_x**

**while True:**

**try:**

**n, m = map(int, input().split())**

**parent = list(range(n + 1))**

**for \_ in range(m):**

**a, b = map(int, input().split())**

**if find(a) == find(b):**

**print('Yes')**

**else:**

**print('No')**

**union(a, b)**

**unique\_parents = set(find(x) for x in range(1, n + 1))**

**ans = sorted(unique\_parents) # 输出有冰阔落的杯子编号**

**print(len(ans))**

**print(\*ans)**

**except EOFError:**

**break**

**19943：图的拉普拉斯矩阵**

**class Vertex:**

**def \_\_init\_\_(self, key):**

**self.id = key**

**self.connectedTo = {}**

**def addNeighbor(self, nbr, weight=0):**

**self.connectedTo[nbr] = weight**

**def \_\_str\_\_(self):**

**return str(self.id) + ' connectedTo: ' + str([x.id for x in self.connectedTo])**

**def getConnections(self):**

**return self.connectedTo.keys()**

**def getId(self):**

**return self.id**

**def getWeight(self, nbr):**

**return self.connectedTo[nbr]**

**class Graph:**

**def \_\_init\_\_(self):**

**self.vertList = {}**

**self.numVertices = 0**

**def addVertex(self, key):**

**self.numVertices = self.numVertices + 1**

**newVertex = Vertex(key)**

**self.vertList[key] = newVertex**

**return newVertex**

**def getVertex(self, n):**

**if n in self.vertList:**

**return self.vertList[n]**

**else:**

**return None**

**def \_\_contains\_\_(self, n):**

**return n in self.vertList**

**def addEdge(self, f, t, weight=0):**

**if f not in self.vertList:**

**nv = self.addVertex(f)**

**if t not in self.vertList:**

**nv = self.addVertex(t)**

**self.vertList[f].addNeighbor(self.vertList[t], weight)**

**def getVertices(self):**

**return self.vertList.keys()**

**def \_\_iter\_\_(self):**

**return iter(self.vertList.values())**

**def constructLaplacianMatrix(n, edges):**

**graph = Graph()**

**for i in range(n): # 添加顶点**

**graph.addVertex(i)**

**for edge in edges: # 添加边**

**a, b = edge**

**graph.addEdge(a, b)**

**graph.addEdge(b, a)**

**laplacianMatrix = [] # 构建拉普拉斯矩阵**

**for vertex in graph:**

**row = [0] \* n**

**row[vertex.getId()] = len(vertex.getConnections())**

**for neighbor in vertex.getConnections():**

**row[neighbor.getId()] = -1**

**laplacianMatrix.append(row)**

**return laplacianMatrix**

**n, m = map(int, input().split()) # 解析输入**

**edges = []**

**for i in range(m):**

**a, b = map(int, input().split())**

**edges.append((a, b))**

**laplacianMatrix = constructLaplacianMatrix(n, edges) # 构建拉普拉斯矩阵**

**for row in laplacianMatrix: # 输出结果**

**print(' '.join(map(str, row)))**

**20018：蚂蚁王国的越野跑**

**from bisect import bisect\_left**

**n=int(input())**

**v=[]**

**ans=0**

**for i in range(n):**

**p=int(input())**

**index=bisect\_left(v,p)**

**v.insert(index,p)**

**ans+=index**

**print(ans)**

**20449：是否被5整除**

**def binary\_divisible\_by\_five(binary\_string):**

**result = ''**

**num = 0**

**for bit in binary\_string:**

**num = (num \* 2 + int(bit)) % 5**

**if num == 0:**

**result += '1'**

**else:**

**result += '0'**

**return result**

**binary\_string = input().strip()**

**print(binary\_divisible\_by\_five(binary\_string))**

**20453：和为k的子数组个数**

**def subarray\_sum(nums, k):**

**count = 0**

**sums = 0**

**d = dict()**

**d[0] = 1**

**for i in range(len(nums)):**

**sums += nums[i]**

**count += d.get(sums - k, 0)**

**d[sums] = d.get(sums, 0) + 1**

**return count**

**nums = list(map(int, input().split()))**

**k = int(input().strip())**

**print(subarray\_sum(nums, k))**

**20456：统计封闭岛屿的数目**

**def closedIsland(grid):**

**rows, cols = len(grid), len(grid[0])**

**def dfs(r, c):**

**if grid[r][c] == 1:**

**return True**

**if r == 0 or r == rows - 1 or c == 0 or c == cols - 1:**

**return False**

**grid[r][c] = 1**

**up = dfs(r - 1, c)**

**down = dfs(r + 1, c)**

**left = dfs(r, c - 1)**

**right = dfs(r, c + 1)**

**return up and down and left and right**

**closed\_islands = 0**

**for r in range(1, rows - 1): # 从1开始，忽略边界**

**for c in range(1, cols - 1): # 从1开始，忽略边界**

**if grid[r][c] == 0 and dfs(r, c):**

**closed\_islands += 1**

**return closed\_islands**

**grid = []**

**for \_ in range(10):**

**row = list(map(int, input().split(',')))**

**grid.append(row)**

**print(closedIsland(grid))**

**20472：死循环的机器人**

**def is\_robot\_making\_loop(commands):**

**x, y = 0, 0**

**directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]**

**dir\_index = 0**

**for command in commands:**

**if command == 'G':**

**x += directions[dir\_index][0]**

**y += directions[dir\_index][1]**

**elif command == 'L':**

**dir\_index = (dir\_index - 1) % 4**

**elif command == 'R':**

**dir\_index = (dir\_index + 1) % 4**

**return (x == 0 and y == 0) or (dir\_index != 0)**

**commands = input().strip()**

**print(1 if is\_robot\_making\_loop(commands) else 0)**

**20576: printExp**

**class BinaryTree:**

**def \_\_init\_\_(self, root, left=None, right=None):**

**self.root = root**

**self.leftChild = left**

**self.rightChild = right**

**def getrightchild(self):**

**return self.rightChild**

**def getleftchild(self):**

**return self.leftChild**

**def getroot(self):**

**return self.root**

**def postorder(string): #中缀改后缀 Shunting yard algorightm**

**opStack = []**

**postList = []**

**inList = string.split()**

**prec = { '(': 0, 'or': 1,'and': 2,'not': 3}**

**for word in inList:**

**if word == '(':**

**opStack.append(word)**

**elif word == ')':**

**topWord = opStack.pop()**

**while topWord != '(':**

**postList.append(topWord)**

**topWord = opStack.pop()**

**elif word == 'True' or word == 'False':**

**postList.append(word)**

**else:**

**while opStack and prec[word] <= prec[opStack[-1]]:**

**postList.append(opStack.pop())**

**opStack.append(word)**

**while opStack:**

**postList.append(opStack.pop())**

**return postList**

**def buildParseTree(infix): #以后缀表达式为基础建树**

**postList = postorder(infix)**

**stack = []**

**for word in postList:**

**if word == 'not':**

**newTree = BinaryTree(word)**

**newTree.leftChild = stack.pop()**

**stack.append(newTree)**

**elif word == 'True' or word == 'False':**

**stack.append(BinaryTree(word))**

**else:**

**right = stack.pop()**

**left = stack.pop()**

**newTree = BinaryTree(word)**

**newTree.leftChild = left**

**newTree.rightChild = right**

**stack.append(newTree)**

**currentTree = stack[-1]**

**return currentTree**

**def printTree(parsetree: BinaryTree):**

**if parsetree.getroot() == 'or':**

**return printTree(parsetree.getleftchild()) + ['or'] + printTree(parsetree.getrightchild())**

**elif parsetree.getroot() == 'not':**

**return ['not'] + (['('] + printTree(parsetree.getleftchild()) + [')']**

**if parsetree.leftChild.getroot() not in ['True', 'False']**

**else printTree(parsetree.getleftchild()) )**

**elif parsetree.getroot() == 'and':**

**leftpart = (['('] + printTree(parsetree.getleftchild()) + [')']**

**if parsetree.leftChild.getroot() == 'or'**

**else printTree(parsetree.getleftchild()))**

**rightpart = (['('] + printTree(parsetree.getrightchild()) + [')']**

**if parsetree.rightChild.getroot() == 'or'**

**else printTree(parsetree.getrightchild()))**

**return leftpart + ['and'] + rightpart**

**else:**

**return [str(parsetree.getroot())]**

**def main():**

**infix = input()**

**Tree = buildParseTree(infix)**

**print(' '.join(printTree(Tree)))**

**main()**

**20625:1跟0数量相等的子字串**

**def count\_balanced\_substrings(s):**

**curr\_count = 1**

**prev\_count = 0**

**result = 0**

**for i in range(1, len(s)):**

**if s[i] == s[i - 1]:**

**curr\_count += 1**

**else:**

**result += min(curr\_count, prev\_count)**

**prev\_count = curr\_count**

**curr\_count = 1**

**result += min(curr\_count, prev\_count)**

**return result**

**print(count\_balanced\_substrings(input()))**

**20626：对子数列做XOR运算**

**import sys**

**input = sys.stdin.readline**

**V = [int(i) for i in input().split()]**

**preV = [0]\*(len(V)+1)**

**for i in range(len(V)):**

**preV[i+1] = preV[i] ^ V[i]**

**results = []**

**for i in range(10000):**

**L, R = map(int, input().split())**

**results.append(str(preV[R+1] ^ preV[L]))**

**sys.stdout.write('\n'.join(results) + '\n')**

**20644：统计全为1的正方形子矩阵**

**m, n = map(int, input().split())**

**mat = [[int(k) for k in input()] for i in range(m)]**

**dp = [[0 for j in range(n+1)] for i in range(m+1)]**

**for i in range(m):**

**for j in range(n):**

**if mat[i][j]:**

**dp[i+1][j+1] = min(dp[i][j], dp[i][j+1], dp[i+1][j])+1**

**print(sum(dp[i][j] for j in range(n+1) for i in range(m+1)))**

**20650：最长的公共子序列的长度**

**def longest\_common\_subsequence(s1, s2):**

**dp = [[0 for \_ in range(len(s2)+1)] for \_ in range(len(s1)+1)]**

**for i in range(len(s1)):**

**for j in range(len(s2)):**

**if s1[i] == s2[j]:**

**dp[i+1][j+1] = dp[i][j] + 1**

**else:**

**dp[i+1][j+1] = max(dp[i+1][j], dp[i][j+1])**

**return dp[len(s1)][len(s2)]**

**s1 = input()**

**s2 = input()**

**print(longest\_common\_subsequence(s1, s2))**

**20742：泰波拿契數**

**def tribonacci(n):**

**if n == 0:**

**return 0**

**elif n <= 2:**

**return 1**

**trib = [0, 1, 1] + [0] \* (n - 2)**

**for i in range(3, n + 1):**

**trib[i] = trib[i - 1] + trib[i - 2] + trib[i - 3]**

**return trib[n]**

**n = int(input())**

**print(tribonacci(n))**

**20743：整人的提词本**

**def reverse\_parentheses(s):**

**stack = []**

**for char in s:**

**if char == ')':**

**temp = []**

**while stack and stack[-1] != '(':**

**temp.append(stack.pop())**

**if stack:**

**stack.pop()**

**stack.extend(temp)**

**else:**

**stack.append(char)**

**return ''.join(stack)**

**s = input().strip()**

**print(reverse\_parentheses(s))**

**20746：满足合法工时的最少人数**

**def min\_employees(tasks, t):**

**left, right = 1, max(tasks)**

**while left < right:**

**mid = (left + right) // 2**

**total\_hours = sum((task + mid - 1) // mid for task in tasks)**

**if total\_hours > t:**

**left = mid + 1**

**else:**

**right = mid**

**return left**

**tasks = list(map(int, input().split(',')))**

**t = int(input())**

**print(min\_employees(tasks, t))**

**21006：放苹果（盘子相同）**

**m,n=map(int,input().split())**

**dp=[[0]\*(m+1) for i in range(n+1)]**

**for i in range(1,n+1):**

**dp[i][0]=1**

**dp[1]=[1]\*(m+1)**

**for i in range(1,n+1):**

**dp[i][1]=1**

**for i in range(1,n+1):**

**for j in range(1,m+1):**

**if i>j:**

**dp[i][j]=dp[j][j]**

**else:**

**dp[i][j]=dp[i-1][j]+dp[i][j-i]**

**print(dp[n][m])**

**21515：电话线路**

**from heapq import \***

**n,p,k = map(int,input().split())**

**graph = {i:{} for i in range(1,n+1)}**

**h = 0**

**for \_ in range(p):**

**a,b,l = map(int,input().split())**

**graph[a][b] = graph[b][a] = l**

**h = max(h,l)**

**l = 0**

**def search(lim):**

**heap = [(-1,-k)]**

**heapify(heap)**

**vis = {}**

**while heap:**

**idx,free = heappop(heap)**

**idx,free = -idx,-free**

**if idx == n:**

**return 1**

**if idx not in vis or vis[idx] < free:**

**vis[idx] = free**

**else:**

**continue**

**for t,length in graph[idx].items():**

**new\_free = free**

**if length > lim:**

**if new\_free > 0:**

**new\_free -= 1**

**else:**

**continue**

**if t in vis and vis[t] > new\_free:**

**continue**

**heappush(heap,(-t,-new\_free))**

**return 0**

**while l < h:**

**if l +1 == h:**

**ans\_l,ans\_h = search(l),search(h)**

**if ans\_l == ans\_h == 0:**

**print(-1)**

**else:**

**print(l if ans\_l else h)**

**exit()**

**mid = (l+h)//2**

**if search(mid):**

**h = mid**

**else:**

**l = mid**

**22067：快速堆猪**

**a = []**

**m = []**

**while True:**

**try:**

**s = input().split()**

**if s[0] == "pop":**

**if a:**

**a.pop()**

**if m:**

**m.pop()**

**elif s[0] == "min":**

**if m:**

**print(m[-1])**

**else:**

**h = int(s[1])**

**a.append(h)**

**if not m:**

**m.append(h)**

**else:**

**k = m[-1]**

**m.append(min(k, h))**

**except EOFError:**

**break**

**22068：合法出栈序列**

**origin = input()**

**while True:**

**try:**

**outout = input()**

**stack,bank = [],list(origin)**

**l = len(origin)**

**flag = False**

**if len(outout) == l:**

**for i in range(l):**

**if bank and not stack:**

**stack.append(bank.pop(0))**

**while bank and stack[-1] != outout[i]:**

**stack.append(bank.pop(0))**

**if stack.pop() != outout[i]:**

**print('NO')**

**flag = True**

**break**

**if not flag:**

**print('YES')**

**else:**

**print('NO')**

**except EOFError:**

**break**

**22158：根据二叉树前中序序列建树**

**class TreeNode:**

**def \_\_init\_\_(self, value):**

**self.value = value**

**self.left = None**

**self.right = None**

**def build\_tree(preorder, inorder):**

**if not preorder or not inorder:**

**return None**

**root\_value = preorder[0]**

**root = TreeNode(root\_value)**

**root\_index\_inorder = inorder.index(root\_value)**

**root.left = build\_tree(preorder[1:1+root\_index\_inorder], inorder[:root\_index\_inorder])**

**root.right = build\_tree(preorder[1+root\_index\_inorder:], inorder[root\_index\_inorder+1:])**

**return root**

**def postorder\_traversal(root):**

**if root is None:**

**return ''**

**return postorder\_traversal(root.left) + postorder\_traversal(root.right) + root.value**

**while True:**

**try:**

**preorder = input().strip()**

**inorder = input().strip()**

**root = build\_tree(preorder, inorder)**

**print(postorder\_traversal(root))**

**except EOFError:**

**break**

**22161：哈夫曼编码树**

**import heapq**

**class Node:**

**def \_\_init\_\_(self, weight, char=None):**

**self.weight = weight**

**self.char = char**

**self.left = None**

**self.right = None**

**def \_\_lt\_\_(self, other):**

**if self.weight == other.weight:**

**return self.char < other.char**

**return self.weight < other.weight**

**def build\_huffman\_tree(characters):**

**heap = []**

**for char, weight in characters.items():**

**heapq.heappush(heap, Node(weight, char))**

**while len(heap) > 1:**

**left = heapq.heappop(heap)**

**right = heapq.heappop(heap)**

**merged = Node(left.weight + right.weight, min(left.char, right.char))**

**merged.left = left**

**merged.right = right**

**heapq.heappush(heap, merged)**

**return heap[0]**

**def encode\_huffman\_tree(root):**

**codes = {}**

**def traverse(node, code):**

**if node.left is None and node.right is None:**

**codes[node.char] = code**

**else:**

**traverse(node.left, code + '0')**

**traverse(node.right, code + '1')**

**traverse(root, '')**

**return codes**

**def huffman\_encoding(codes, string):**

**encoded = ''**

**for char in string:**

**encoded += codes[char]**

**return encoded**

**def huffman\_decoding(root, encoded\_string):**

**decoded = ''**

**node = root**

**for bit in encoded\_string:**

**if bit == '0':**

**node = node.left**

**else:**

**node = node.right**

**if node.left is None and node.right is None:**

**decoded += node.char**

**node = root**

**return decoded**

**n = int(input())**

**characters = {}**

**for \_ in range(n):**

**char, weight = input().split()**

**characters[char] = int(weight)**

**huffman\_tree = build\_huffman\_tree(characters)**

**codes = encode\_huffman\_tree(huffman\_tree)**

**strings = []**

**while True:**

**try:**

**line = input()**

**strings.append(line)**

**except EOFError:**

**break**

**results = []**

**#print(strings)**

**for string in strings:**

**if string[0] in ('0','1'):**

**results.append(huffman\_decoding(huffman\_tree, string))**

**else:**

**results.append(huffman\_encoding(codes, string))**

**for result in results:**

**print(result)**

**22275：二叉搜索树的遍历**

**def post\_order(pre\_order):**

**if not pre\_order:**

**return []**

**root = pre\_order[0]**

**left\_subtree = [x for x in pre\_order if x < root]**

**right\_subtree = [x for x in pre\_order if x > root]**

**return post\_order(left\_subtree) + post\_order(right\_subtree) + [root]**

**n = int(input())**

**pre\_order = list(map(int, input().split()))**

**print(' '.join(map(str, post\_order(pre\_order))))**

**22359: Goldbach Conjecture**

**from math import sqrt**

**n=10000**

**ls,x,y=[True]\*(n+1),2,int(sqrt(n))+1**

**while x<y:**

**if ls[x]==True:**

**for i in range(x\*2,n+1,x):**

**ls[i]=False**

**x+=1**

**ls=set([i for i in range(2,n+1) if ls[i]==True])**

**n=int(input())**

**for i in ls:**

**if (n-i) in ls:**

**print(i,n-i)**

**break**

**22485：升空的焰火，从侧面看**

**n = int(input())**

**tree = [0]**

**for i in range(n):**

**tree.append(list(map(int, input().split())))**

**stack = [1]**

**ans = []**

**while stack:**

**ans.append(str(stack[-1]))**

**temp = []**

**for x in stack:**

**if tree[x][0] != -1:**

**temp.append(tree[x][0])**

**if tree[x][1] != -1:**

**temp.append(tree[x][1])**

**stack = temp**

**print(" ".join(ans))**

**T22508：最小奖金方案**

**import sys**

**from collections import defaultdict, deque**

**def min\_bonus(n, m, matches):**

**graph = defaultdict(list)**

**indegree = [0] \* n**

**for a, b in matches:**

**graph[b].append(a) # a > b，所以 b 是 a 的前驱**

**indegree[a] += 1**

**bonus = [100] \* n**

**queue = deque([i for i in range(n) if indegree[i] == 0])**

**while queue:**

**curr = queue.popleft()**

**for neighbor in graph[curr]:**

**if bonus[neighbor] <= bonus[curr]:**

**bonus[neighbor] = bonus[curr] + 1**

**indegree[neighbor] -= 1**

**if indegree[neighbor] == 0:**

**queue.append(neighbor)**

**return sum(bonus)**

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

**input = sys.stdin.read**

**data = input().split()**

**n = int(data[0])**

**m = int(data[1])**

**matches = []**

**idx = 2**

**for \_ in range(m):**

**a = int(data[idx])**

**b = int(data[idx+1])**

**matches.append((a, b))**

**idx += 2**

**result = min\_bonus(n, m, matches)**

**print(result)**

**22509：解方程**

**import heapq**

**def dijkstra(graph, start):**

**N = len(graph)**

**dist = [float('inf')] \* N**

**dist[start] = 0**

**queue = [(0, start)]**

**while queue:**

**v\_dist, v = heapq.heappop(queue)**

**if v\_dist != dist[v]:**

**continue**

**for w, w\_dist in graph[v]:**

**if dist[v] + w\_dist < dist[w]:**

**dist[w] = dist[v] + w\_dist**

**heapq.heappush(queue, (dist[w], w))**

**return dist**

**N, M = map(int, input().split())**

**N += 2**

**graph = [[] for \_ in range(N)]**

**time\_to\_defeat = [0, 0] + [int(input()) for \_ in range(N-2)]**

**for \_ in range(M):**

**u, v, t = map(int, input().split())**

**graph[u].append((v, t + time\_to\_defeat[v]))**

**graph[v].append((u, t + time\_to\_defeat[u]))**

**distances = dijkstra(graph, 0)**

**print(distances[1])**

**22510：皮卡丘的冒险**

**import heapq**

**def dijkstra(graph, start):**

**N = len(graph)**

**dist = [float('inf')] \* N**

**dist[start] = 0**

**queue = [(0, start)]**

**while queue:**

**v\_dist, v = heapq.heappop(queue)**

**if v\_dist != dist[v]:**

**continue**

**for w, w\_dist in graph[v]:**

**if dist[v] + w\_dist < dist[w]:**

**dist[w] = dist[v] + w\_dist**

**heapq.heappush(queue, (dist[w], w))**

**return dist**

**N, M = map(int, input().split())**

**N += 2**

**graph = [[] for \_ in range(N)]**

**time\_to\_defeat = [0, 0] + [int(input()) for \_ in range(N-2)]**

**for \_ in range(M):**

**u, v, t = map(int, input().split())**

**graph[u].append((v, t + time\_to\_defeat[v]))**

**graph[v].append((u, t + time\_to\_defeat[u]))**

**distances = dijkstra(graph, 0)**

**print(distances[1])**

**M22528：厚道的调分方法**

**grade = [float(x) for x in input().split()]**

**le = len(grade)**

**grade.sort()**

**targ = grade[int(le \* 0.4)]**

**left = 0**

**right = 1000000000 + 1**

**ans = 0**

**while left < right:**

**mid = (left + right) // 2**

**gd = targ \* mid / 1000000000 + 1.1 \*\* (targ \* mid / 1000000000)**

**if gd >= 85:**

**ans = mid**

**right = mid**

**else:**

**left = mid + 1**

**print(left)**

**22636：修仙之路**

**from functools import lru\_cache**

**@lru\_cache(maxsize=None)**

**def dfs(x,y):**

**ans=0**

**for dx,dy in dir:**

**nx,ny=x+dx,y+dy**

**if 0<=nx<m and 0<=ny<n and h[nx][ny]<h[x][y]:**

**ans=max(ans,dfs(nx,ny)+1)**

**return ans**

**m,n=map(int,input().split())**

**h=[list(map(int,input().split())) for \_ in range(m)]**

**dir=[(0,1),(1,0),(-1,0),(0,-1)]**

**res=0**

**for i in range(m):**

**for j in range(n):**

**res=max(res,dfs(i,j))**

**print(res+1)**

**22642：括号生成**

**def add(n, left, right, string):**

**if left == n and right == n:**

**print(string)**

**return**

**if left < n:**

**add(n, left+1, right, string+'(')**

**if right < left:**

**add(n, left, right+1, string+')')**

**n = int(input())**

**add(n, 0, 0, '')**

**23451：交互四则运算计算器\_带错误表达式版**

**class stack():**

**def \_\_init\_\_(self):**

**self.val=[]**

**def isempty(self):**

**return len(self.val)==0**

**def push(self,item):**

**self.val.append(item)**

**def top(self):**

**return self.val[-1]**

**def pop(self):**

**del self.val[-1]**

**def operatorcheck():**

**for i in range(len(exp)):**

**if exp[i] not in ch:**

**return 0**

**return 1**

**def bracketcheck():**

**bracket=stack()**

**for i in range(len(exp)):**

**if exp[i]=='(':**

**bracket.push('(')**

**if exp[i]==')':**

**if bracket.isempty():**

**return 0**

**else:**

**bracket.pop()**

**if bracket.isempty():**

**return 1**

**else:**

**return 0**

**def onlybracket():**

**for i in range(len(exp)):**

**if exp[i]!='(' and exp[i]!=')':**

**return 0**

**return 1**

**def cut():**

**i=0**

**while i<=len(exp)-1:**

**if exp[i]=='\*' or exp[i]=='/' or exp[i]=='(' or exp[i]==')':**

**expression.append(exp[i])**

**i+=1**

**continue**

**if exp[i]=='+' or exp[i]=='-':**

**if i==0 or exp[i-1] not in ch[5:]:**

**temp=''+exp[i]**

**i+=1**

**while i<=len(exp)-1 and exp[i] in ch[6:]:**

**temp=temp+exp[i]**

**i+=1**

**expression.append(float(temp))**

**continue**

**else:**

**expression.append(exp[i])**

**i+=1**

**continue**

**if exp[i] in ch[6:]:**

**temp=''**

**while i<=len(exp)-1 and exp[i] in ch[6:]:**

**temp=temp+exp[i]**

**i+=1**

**expression.append(float(temp))**

**continue**

**def value(s,x,y):**

**if s=='+':**

**return x+y**

**if s=='\*':**

**return x\*y**

**if s=='-':**

**return x-y**

**if s=='/':**

**return x/y**

**def calc():**

**operator=stack()**

**operand=stack()**

**for i in range(len(expression)):**

**if expression[i] not in ch[0:6]:**

**operand.push(expression[i])**

**elif expression[i]=='(':**

**operator.push('(')**

**elif expression[i]==')':**

**while operator.top()!='(':**

**b=operand.top()**

**operand.pop()**

**a=operand.top()**

**operand.pop()**

**operand.push(value(operator.top(),a,b))**

**operator.pop()**

**operator.pop()**

**elif expression[i] in ch[0:4]:**

**while not operator.isempty() and prior[operator.top()]>=prior[expression[i]]:**

**b=operand.top()**

**operand.pop()**

**a=operand.top()**

**operand.pop()**

**operand.push(value(operator.top(),a,b))**

**operator.pop()**

**operator.push(expression[i])**

**while not operator.isempty():**

**b=operand.top()**

**operand.pop()**

**a=operand.top()**

**operand.pop()**

**operand.push(value(operator.top(),a,b))**

**operator.pop()**

**print('{:.3f}'.format(operand.top()))**

**ch=['+','-','\*','/','(',')','.','0','1','2','3','4','5','6','7','8','9']**

**prior={'\*':3,'/':3,'+':2,'-':2,'(':1}**

**while True:**

**s=list(map(str,input().split()))**

**if s==["quit"]:**

**break**

**if len(s)==0:**

**print("No expression.")**

**continue**

**exp=""**

**for i in range(len(s)):**

**exp=exp+s[i]**

**if operatorcheck()==False:**

**print("Unknown operator.")**

**continue**

**if bracketcheck()==False:**

**print("Unmatched bracket.")**

**continue**

**if onlybracket()==True:**

**print("No expression.")**

**continue**

**expression=[]**

**try:**

**cut()**

**calc()**

**except:**

**print("Not implemented.")**

**continue**

**23563：多项式时间复杂度**

**def toggle(bit):**

**return '0' if bit == '1' else '1'**

**def flip(lock, i):**

**if i > 0:**

**lock[i-1] = toggle(lock[i-1])**

**lock[i] = toggle(lock[i])**

**if i + 1 < len(lock):**

**lock[i+1] = toggle(lock[i+1])**

**def main():**

**s = input()**

**fin = input()**

**n = len(s)**

**ans = float('inf')**

**for press\_first in [False, True]:**

**tmp = 0**

**lock = list(s)**

**if press\_first:**

**flip(lock, 0)**

**tmp += 1**

**for i in range(1, n):**

**if lock[i-1] != fin[i-1]:**

**flip(lock, i)**

**tmp += 1**

**if lock[n-1] == fin[n-1]:**

**ans = min(ans, tmp)**

**if ans == float('inf'):**

**print("impossible")**

**else:**

**print(ans)**

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

**main()**

**23568：幸福的寒假生活**

**def toggle(bit):**

**return '0' if bit == '1' else '1'**

**def flip(lock, i):**

**if i > 0:**

**lock[i-1] = toggle(lock[i-1])**

**lock[i] = toggle(lock[i])**

**if i + 1 < len(lock):**

**lock[i+1] = toggle(lock[i+1])**

**def main():**

**s = input()**

**fin = input()**

**n = len(s)**

**ans = float('inf')**

**for press\_first in [False, True]:**

**tmp = 0**

**lock = list(s)**

**if press\_first:**

**flip(lock, 0)**

**tmp += 1**

**for i in range(1, n):**

**if lock[i-1] != fin[i-1]:**

**flip(lock, i)**

**tmp += 1**

**if lock[n-1] == fin[n-1]:**

**ans = min(ans, tmp)**

**if ans == float('inf'):**

**print("impossible")**

**else:**

**print(ans)**

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

**main()**

**23570：特珠密码锁**

**def toggle(bit):**

**return '0' if bit == '1' else '1'**

**def flip(lock, i):**

**if i > 0:**

**lock[i-1] = toggle(lock[i-1])**

**lock[i] = toggle(lock[i])**

**if i + 1 < len(lock):**

**lock[i+1] = toggle(lock[i+1])**

**def main():**

**s = input()**

**fin = input()**

**n = len(s)**

**ans = float('inf')**

**for press\_first in [False, True]:**

**tmp = 0**

**lock = list(s)**

**if press\_first:**

**flip(lock, 0)**

**tmp += 1**

**for i in range(1, n):**

**if lock[i-1] != fin[i-1]:**

**flip(lock, i)**

**tmp += 1**

**if lock[n-1] == fin[n-1]:**

**ans = min(ans, tmp)**

**if ans == float('inf'):**

**print("impossible")**

**else:**

**print(ans)**

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

**main()**

**23660:7的倍数取法有多少种**

**def count\_combinations(numbers, index, current\_sum, count):**

**if index >= len(numbers):**

**if current\_sum % 7 == 0:**

**return count + 1**

**else:**

**return count**

**count = count\_combinations(numbers, index + 1, current\_sum + numbers[index], count)**

**count = count\_combinations(numbers, index + 1, current\_sum, count)**

**return count**

**t = int(input())**

**for \_ in range(t):**

**data = list(map(int, input().split()))**

**n = data[0]**

**numbers = data[1:]**

**result = count\_combinations(numbers, 0, 0, 0)**

**print(result)**

**24375：小木棍**

**def dfs(rem\_sticks,rem\_len,target):**

**if rem\_sticks==0 and rem\_len==0:**

**return True**

**if rem\_len==0:**

**rem\_len=target**

**for i in range(n):**

**if not used[i] and lens[i]<=rem\_len:**

**used[i]=True**

**if dfs(rem\_sticks-1,rem\_len-lens[i],target):**

**return True**

**else:**

**used[i]=False**

**if lens[i]==rem\_len or rem\_len==target:**

**return False**

**return False**

**while True:**

**n=int(input())**

**if n==0:**

**break**

**lens=list(map(int,input().split()))**

**lens.sort(reverse=True)**

**total\_len=sum(lens)**

**for l in range(lens[0],total\_len//2+1):**

**if total\_len%l!=0:**

**continue**

**used=[False]\*n**

**if dfs(n,l,l):**

**print(l)**

**break**

**else:**

**print(total\_len)**

**24588：后序表达式求值**

**def compute(stack, operator):**

**op1 = stack.pop()**

**op2 = stack.pop()**

**if operator == '+':**

**return op2 + op1**

**elif operator == '-':**

**return op2 - op1**

**elif operator == '\*':**

**return op2 \* op1**

**elif operator == '/':**

**return op2 / op1**

**def post\_eva(formula):**

**comp = '+-\*/'**

**wordlist = formula.split()**

**opStack = []**

**for word in wordlist:**

**if word not in comp:**

**opStack.append(float(word))**

**else:**

**op = compute(opStack, word)**

**opStack.append(op)**

**return opStack[0]**

**n = int(input())**

**for \_ in range(n):**

**result = post\_eva(input())**

**print(f"{result:.2f}")**

**24591：中序表达式转后序表达式**

**def inp(s):**

**#s=input().strip()**

**import re**

**s=re.split(r'([\(\)\+\-\\*\/])',s)**

**s=[item for item in s if item.strip()]**

**return s**

**exp = "(3)\*((3+4)\*(2+3.5)/(4+5)) "**

**print(inp(exp))**

**M24637：宝藏二叉树**

**class Solution:**

**def rob(self, values):**

**from functools import lru\_cache**

**n = len(values)**

**def dfs(i):**

**if i > n:**

**return 0, 0 # (rob, not\_rob)**

**left = 2 \* i**

**right = 2 \* i + 1**

**l\_rob, l\_not\_rob = dfs(left)**

**r\_rob, r\_not\_rob = dfs(right)**

**rob\_i = values[i - 1] + l\_not\_rob + r\_not\_rob**

**not\_rob\_i = max(l\_rob, l\_not\_rob) + max(r\_rob, r\_not\_rob)**

**return rob\_i, not\_rob\_i**

**return max(dfs(1)) # 根节点编号为1**

**sol = Solution()**

**n = int(input())**

**values = list(map(int, input().split()))**

**print(sol.rob(values))**

**24676：共同富裕**

**from itertools import product**

**def right\_shift(row, shift):**

**return row[-shift:] + row[:-shift]**

**def calculate\_max\_column\_sum(matrix):**

**n = len(matrix)**

**column\_sums = [0] \* n**

**for row in matrix:**

**for i, val in enumerate(row):**

**column\_sums[i] += val**

**return max(column\_sums)**

**def find\_min\_max\_column\_sum(n, original\_matrix):**

**min\_max\_sum = float('inf')**

**all\_shifts = list(product(range(n), repeat=n))**

**for shifts in all\_shifts:**

**shifted\_matrix = [**

**right\_shift(original\_matrix[i], shifts[i]) for i in range(n)]**

**max\_column\_sum = calculate\_max\_column\_sum(shifted\_matrix)**

**min\_max\_sum = min(min\_max\_sum, max\_column\_sum)**

**return min\_max\_sum**

**results = []**

**while True:**

**n = int(input())**

**if n == 0:**

**break**

**original\_matrix = [list(map(int, input().split())) for \_ in range(n)]**

**result = find\_min\_max\_column\_sum(n, original\_matrix)**

**results.append(result)**

**for result in results:**

**print(result)**

**24677：安全位置**

**def safe\_locations(s, parts, depth=0):**

**if depth == 4:**

**if not s and all(0 <= int(part) <= 500 and**

**(part == '0' or not part.startswith('0')) for part in parts):**

**return 1**

**return 0**

**return sum(safe\_locations(s[i:], parts + [s[:i]], depth + 1)**

**for i in range(1, len(s) + 1))**

**s = input().strip()**

**print(safe\_locations(s, []))**

**24678：任性买房**

**def min\_houses\_to\_buy(W, n, prices):**

**min\_length = n + 1 # 初始化为最大长度+1，表示不可能的情况**

**current\_sum = 0 # 当前窗口的价格总和**

**left = 0 # 窗口的左边界**

**for right in range(n):**

**current\_sum += prices[right] # 扩展窗口的右边界**

**while current\_sum >= W and left <= right:**

**min\_length = min(min\_length, right - left + 1)**

**current\_sum -= prices[left] # 缩小窗口的左边界**

**left += 1**

**return min\_length if min\_length <= n else 0**

**W, n = map(int, input().split())**

**prices = list(map(int, input().split()))**

**print(min\_houses\_to\_buy(W, n, prices))**

**24684：直播计票**

**from collections import defaultdict**

**votes = list(map(int, input().split()))**

**vote\_counts = defaultdict(int)**

**for vote in votes:**

**vote\_counts[vote] += 1**

**max\_votes = max(vote\_counts.values())**

**winners = sorted([item for item in vote\_counts.items() if item[1] == max\_votes])**

**print(' '.join(str(winner[0]) for winner in winners))**

**24686：树的重量**

**k, n = [int(x) for x in input().split()]**

**f, g, dep = [], [], []**

**tot = (1 << k) - 1**

**for \_ in range(tot+1):**

**f.append(0)**

**g.append(0)**

**dep.append(0)**

**for i in range(tot, 0, -1):**

**dep[i] = 1 if i \* 2 > tot else dep[i \* 2] + 1**

**for \_ in range(n):**

**a = [int(x) for x in input().split()]**

**if len(a) == 2:**

**u = a[1]**

**s = f[1]**

**while u != 1:**

**s += f[u]**

**u >>= 1**

**ans = s \* ((1 << dep[a[1]]) - 1) + g[a[1]]**

**print(ans)**

**elif len(a) == 3:**

**u = a[1]**

**w = a[2] \* ((1 << dep[u]) - 1)**

**f[u] += a[2]**

**while u != 1:**

**u >>= 1**

**g[u] += w**

**24687：封锁管控**

**def min\_population\_flow(n, m, populations):**

**# Initialize the prefix sum array for fast range sum computation**

**prefix\_sum = [0] \* (n + 1)**

**for i in range(1, n + 1):**

**prefix\_sum[i] = prefix\_sum[i - 1] + populations[i - 1]**

**# Initialize the DP table**

**dp = [[float('inf')] \* (m + 1) for \_ in range(n + 1)]**

**# Base case: with 0 control points, the flow index is just the sum of all populations times their district count**

**for i in range(1, n + 1):**

**dp[i][0] = prefix\_sum[i] \* i**

**for i in range(1, n + 1):**

**for j in range(1, min(i, m) + 1):**

**for k in range(j-1, i):**

**dp[i][j] = min(dp[i][j], dp[k][j-1] + (prefix\_sum[i] - prefix\_sum[k]) \* (i - k))**

**n, m = map(int, input().split())**

**populations = list(map(int, input().split()))**

**print(min\_population\_flow(n, m, populations))**

**24729：括号嵌套树**

**def parse\_tree(s):**

**stack = []**

**node = None**

**for char in s:**

**if char.isalpha(): # 如果是字母，创建新节点**

**node = {'value': char, 'children': []}**

**if stack:**

**stack[-1]['children'].append(node)**

**elif char == '(': # 遇到左括号，当前节点可能会有子节点**

**if node:**

**stack.append(node) # 把当前节点推入栈中**

**node = None**

**elif char == ')': # 遇到右括号，子节点列表结束**

**if stack:**

**node = stack.pop() # 弹出当前节点**

**return node # 根节点**

**def preorder(node):**

**output = [node['value']]**

**for child in node['children']:**

**output.extend(preorder(child))**

**return ''.join(output)**

**def postorder(node):**

**output = []**

**for child in node['children']:**

**output.extend(postorder(child))**

**output.append(node['value'])**

**return ''.join(output)**

**def main():**

**s = input().strip()**

**s = ''.join(s.split()) # 去掉所有空白字符**

**root = parse\_tree(s) # 解析整棵树**

**if root:**

**print(preorder(root)) # 输出前序遍历序列**

**print(postorder(root)) # 输出后序遍历序列**

**else:**

**print("input tree string error!")**

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

**main()**

**24750：根据二叉树中后序序列建树**

**def build\_tree(inorder, postorder):**

**if not inorder or not postorder:**

**return []**

**root\_val = postorder[-1]**

**root\_index = inorder.index(root\_val)**

**left\_inorder = inorder[:root\_index]**

**right\_inorder = inorder[root\_index + 1:]**

**left\_postorder = postorder[:len(left\_inorder)]**

**right\_postorder = postorder[len(left\_inorder):-1]**

**root = [root\_val]**

**root.extend(build\_tree(left\_inorder, left\_postorder))**

**root.extend(build\_tree(right\_inorder, right\_postorder))**

**return root**

**def main():**

**inorder = input().strip()**

**postorder = input().strip()**

**preorder = build\_tree(inorder, postorder)**

**print(''.join(preorder))**

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

**main()**

**25140：根据后序表达式建立队列表达式**

**from collections import deque**

**class TreeNode:**

**def \_\_init\_\_(self, value):**

**self.value = value**

**self.left = None**

**self.right = None**

**def build\_tree(postfix):**

**stack = []**

**for char in postfix:**

**node = TreeNode(char)**

**if char.isupper():**

**node.right = stack.pop()**

**node.left = stack.pop()**

**stack.append(node)**

**return stack[0]**

**def level\_order\_traversal(root):**

**dq = [root]**

**traversal = []**

**while dq:**

**node = dq.pop(0)**

**traversal.append(node.value)**

**if node.left:**

**dq.append(node.left)**

**if node.right:**

**dq.append(node.right)**

**return traversal**

**n = int(input().strip())**

**for \_ in range(n):**

**postfix = input().strip()**

**root = build\_tree(postfix)**

**queue\_expression = level\_order\_traversal(root)[::-1]**

**print(''.join(queue\_expression))**

**25145：猜二叉树（按层次遍历）**

**from collections import deque**

**class Node:**

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

**self.data = data**

**self.left = None**

**self.right = None**

**def build\_tree(inorder, postorder):**

**if inorder:**

**root = Node(postorder.pop())**

**root\_index = inorder.index(root.data)**

**root.right = build\_tree(inorder[root\_index+1:], postorder)**

**root.left = build\_tree(inorder[:root\_index], postorder)**

**return root**

**def level\_order\_traversal(root):**

**if root is None:**

**return []**

**result = []**

**queue = deque([root])**

**while queue:**

**node = queue.popleft()**

**result.append(node.data)**

**if node.left:**

**queue.append(node.left)**

**if node.right:**

**queue.append(node.right)**

**return result**

**n = int(input())**

**for \_ in range(n):**

**inorder = list(input().strip())**

**postorder = list(input().strip())**

**root = build\_tree(inorder, postorder)**

**print(''.join(level\_order\_traversal(root)))**

**25655：核酸检测统计**

**from collections import defaultdict**

**n = int(input()) # 学生数量**

**m = int(input()) # 核酸检测信息数量**

**student\_info = [list(map(int, input().split())) for \_ in range(n)]**

**test\_info = [list(map(int, input().split())) for \_ in range(m)]**

**test\_record = defaultdict(list)**

**for day, student\_id in test\_info:**

**test\_record[student\_id].append(day)**

**late\_count = 0**

**department\_uncompletion = defaultdict(int)**

**department\_total\_students = defaultdict(int)**

**for student in student\_info:**

**student\_id, department = student**

**sign = False**

**a = sorted(test\_record[student\_id])**

**if a[0] != 1 or max(a) < 7:**

**sign = True**

**for i in range(len(a)-1):**

**if a[i+1] - a[i] > 3:**

**sign = True**

**break**

**if sign:**

**late\_count += 1**

**department\_uncompletion[department] += 1**

**department\_total\_students[department] += 1**

**department\_ratio = {}**

**for department in department\_uncompletion.keys():**

**ratio = department\_uncompletion[department] / department\_total\_students[department]**

**department\_ratio[department] = ratio**

**worst\_department = max(department\_ratio, key=department\_ratio.get)**

**print(late\_count)**

**print(worst\_department)**

**25815：回文字符串**

**S = list(input())**

**n = len(S)**

**dp = [[0 for \_ in range(n)] for \_ in range(n)]**

**for length in range(1,n):**

**for i in range(n-length):**

**j = i+length**

**if S[i] == S[j]:**

**dp[i][j] = dp[i+1][j-1]**

**else:**

**dp[i][j] = min(dp[i+1][j],dp[i][j-1],dp[i+1][j-1])+1**

**print(dp[0][-1])**

**26495：素数和**

**import itertools**

**def is\_prime(n):**

**if n < 2:**

**return False**

**for i in range(2, int(n\*\*0.5) + 1):**

**if n % i == 0:**

**return False**

**return True**

**def count\_prime\_sums(nums, k):**

**count = 0**

**for combination in itertools.combinations(nums, k):**

**if is\_prime(sum(combination)):**

**count += 1**

**return count**

**n, k = map(int, input().split())**

**nums = list(map(int, input().split()))**

**print(count\_prime\_sums(nums, k))**

**26518：最大无环图扩展**

**from collections import deque**

**def max\_dag(n, edges):**

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

**indegree = [0]\*n**

**for a, b in edges:**

**adj[a].append(b)**

**indegree[b] += 1**

**topo\_order = []**

**queue = deque([i for i in range(n) if indegree[i] == 0])**

**while queue:**

**node = queue.popleft()**

**topo\_order.append(node)**

**for neighbor in adj[node]:**

**indegree[neighbor] -= 1**

**if indegree[neighbor] == 0:**

**queue.append(neighbor)**

**added\_edges = 0**

**for i in range(n):**

**for j in range(i+1, n):**

**if topo\_order[j] not in adj[topo\_order[i]]:**

**added\_edges += 1**

**return added\_edges**

**n, m = map(int, input().split())**

**edges = [list(map(int, input().split())) for \_ in range(m)]**

**print(max\_dag(n, edges))**

**26572：多余的括号**

**import re**

**while True:**

**try:**

**s = re.split(r"(\D)", input())**

**except EOFError:**

**break**

**pf = eval(''.join(s))**

**parenthesis = []**

**stack = []**

**for i, token in enumerate(s):**

**if token == '(':**

**stack.append(len(parenthesis))**

**parenthesis.append([i])**

**elif token == ')':**

**parenthesis[stack.pop()].append(i)**

**for l, r in parenthesis:**

**s[l] = ''**

**s[r] = ''**

**if pf != eval(''.join(s)):**

**s[l] = '('**

**s[r] = ')'**

**print(''.join(s))**

**26573：康托集的图像表示**

**def print\_cantor\_set(n):**

**def cantor(start, end, level):**

**if level == 0:**

**for i in range(start, end):**

**cantor\_set[i] = '\*' # Mark the segment as occupied**

**else:**

**segment\_length = (end - start) // 3**

**# Recursively mark the first third and the last third**

**cantor(start, start + segment\_length, level - 1)**

**cantor(end - segment\_length, end, level - 1)**

**cantor\_set = ['-' for \_ in range(3 \*\* n)]**

**cantor(0, 3 \*\* n, n)**

**return ''.join(cantor\_set)**

**n = int(input())**

**print(print\_cantor\_set(n))**

**27205：护林员盖房子 加强版**

**def maximalRectangle(matrix) -> int:**

**if (rows := len(matrix)) == 0:**

**return 0**

**cols = len(matrix[0])**

**height = [0 for \_ in range(cols + 1)]**

**res = 0**

**for i in range(rows): # 遍历以哪一层作为底层**

**stack = [-1]**

**for j in range(cols + 1):**

**# 计算j位置的高度，如果遇到1则置为0，否则递增**

**h = 0 if j == cols or matrix[i][j] == '1' else height[j] + 1**

**height[j] = h**

**# 单调栈维护长度**

**while len(stack) > 1 and h < height[stack[-1]]:**

**res = max(res, (j - stack[-2] - 1) \* height[stack[-1]])**

**stack.pop()**

**stack.append(j)**

**return res**

**rows, \_ = map(int, input().split())**

**a = [input().split() for \_ in range(rows)]**

**print(maximalRectangle(a))**

**27256：当前以列中位数**

**import sys**

**import heapq**

**from collections import deque, defaultdict**

**input = sys.stdin.readline**

**class DualHeap:**

**def \_\_init\_\_(self):**

**self.small = []**

**self.large = []**

**self.delayed = defaultdict(int)**

**self.small\_size = 0**

**self.large\_size = 0**

**def prune(self, heap):**

**if heap is self.small:**

**while heap and self.delayed[-heap[0]] > 0:**

**num = -heapq.heappop(heap)**

**self.delayed[num] -= 1**

**else:**

**while heap and self.delayed[heap[0]] > 0:**

**num = heapq.heappop(heap)**

**self.delayed[num] -= 1**

**def balance(self):**

**if self.small\_size > self.large\_size + 1:**

**self.prune(self.small)**

**num = -heapq.heappop(self.small)**

**self.small\_size -= 1**

**heapq.heappush(self.large, num)**

**self.large\_size += 1**

**elif self.small\_size < self.large\_size:**

**self.prune(self.large)**

**num = heapq.heappop(self.large)**

**self.large\_size -= 1**

**heapq.heappush(self.small, -num)**

**self.small\_size += 1**

**def add(self, num):**

**if not self.small or num <= -self.small[0]:**

**heapq.heappush(self.small, -num)**

**self.small\_size += 1**

**else:**

**heapq.heappush(self.large, num)**

**self.large\_size += 1**

**self.balance()**

**def remove(self, num):**

**self.delayed[num] += 1**

**if self.small and num <= -self.small[0]:**

**self.small\_size -= 1**

**if num == -self.small[0]:**

**self.prune(self.small)**

**else:**

**self.large\_size -= 1**

**if self.large and num == self.large[0]:**

**self.prune(self.large)**

**self.balance()**

**def median(self):**

**self.prune(self.small)**

**self.prune(self.large)**

**total = self.small\_size + self.large\_size**

**if total % 2 == 1:**

**return -self.small[0]**

**else:**

**return (-self.small[0] + self.large[0]) / 2**

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

**n = int(input())**

**dh = DualHeap()**

**q = deque()**

**results = []**

**for \_ in range(n):**

**parts = input().split()**

**op = parts[0]**

**if op == 'add':**

**x = int(parts[1])**

**dh.add(x)**

**q.append(x)**

**elif op == 'del':**

**x = q.popleft()**

**dh.remove(x)**

**elif op == 'query':**

**med = dh.median()**

**if med == int(med):**

**results.append(str(int(med)))**

**else:**

**results.append(str(med))**

**print("\n".join(results))**

**27625:AVL树至少有几个结点**

**from functools import lru\_cache**

**@lru\_cache(maxsize=None)**

**def avl\_min\_nodes(n):**

**if n == 0:**

**return 0**

**elif n == 1:**

**return 1**

**else:**

**return avl\_min\_nodes(n-1) + avl\_min\_nodes(n-2) + 1**

**n = int(input())**

**min\_nodes = avl\_min\_nodes(n)**

**print(min\_nodes)**

**27626:AVL树最多有几层**

**from functools import lru\_cache**

**@lru\_cache(maxsize=None)**

**def min\_nodes(h):**

**if h == 0: return 0**

**if h == 1: return 1**

**return min\_nodes(h-1) + min\_nodes(h-2) + 1**

**def max\_height(n):**

**h = 0**

**while min\_nodes(h) <= n:**

**h += 1**

**return h - 1**

**n = int(input())**

**print(max\_height(n))**

**27635：判断无向图是否连通有无回路（同23163)**

**n,m=list(map(int,input().split()))**

**edge=[[]for \_ in range(n)]**

**for \_ in range(m):**

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

**edge[a].append(b)**

**edge[b].append(a)**

**cnt,ok=set(),0**

**def dfs(x,y):**

**global cnt,ok**

**cnt.add(x)**

**for i in edge[x]:**

**if i not in cnt:dfs(i,x)**

**elif y!=i:ok=1**

**for i in range(n):**

**cnt.clear()**

**dfs(i,-1)**

**if len(cnt)==n:break**

**if ok:break**

**print("connected:"+("yes"if len(cnt)==n else "no")+'\n'+"loop:"+('yes'if ok else 'no'))**

**27637：括号嵌套二叉树**

**class TreeNode:**

**def \_\_init\_\_(self, value):**

**self.value = value**

**self.left = None**

**self.right = None**

**def parse\_tree(s):**

**if s == '\*': # 处理空树**

**return None**

**if '(' not in s: # 只有单个根节点**

**return TreeNode(s)**

**root\_value = s[0] # 根节点值**

**subtrees = s[2:-1] # 去掉根节点和外层括号**

**stack = []**

**comma\_index = None**

**for i, char in enumerate(subtrees):**

**if char == '(':**

**stack.append(char)**

**elif char == ')':**

**stack.pop()**

**elif char == ',' and not stack:**

**comma\_index = i**

**break**

**left\_subtree = subtrees[:comma\_index] if comma\_index is not None else subtrees**

**right\_subtree = subtrees[comma\_index + 1:] if comma\_index is not None else None**

**root = TreeNode(root\_value)**

**root.left = parse\_tree(left\_subtree) # 解析左子树**

**root.right = parse\_tree(right\_subtree) if right\_subtree else None**

**return root**

**def preorder\_traversal(root):**

**return root.value + preorder\_traversal(root.left) + preorder\_traversal(root.right) if root else ""**

**def inorder\_traversal(root):**

**return inorder\_traversal(root.left) + root.value + inorder\_traversal(root.right) if root else ""**

**n = int(input().strip())**

**results = []**

**for \_ in range(n):**

**tree\_string = input().strip().replace(" ", "") # 去掉可能的空格**

**tree = parse\_tree(tree\_string)**

**results.append(preorder\_traversal(tree))**

**results.append(inorder\_traversal(tree))**

**print("\n".join(results)) # 按格式输出**

**27638：求二叉树的高度和叶子数目**

**class TreeNode:**

**def \_\_init\_\_(self):**

**self.left = None**

**self.right = None**

**def tree\_height(node):**

**if node is None:**

**return -1 # 根据定义，空树高度为-1**

**return max(tree\_height(node.left), tree\_height(node.right)) + 1**

**def count\_leaves(node):**

**if node is None:**

**return 0**

**if node.left is None and node.right is None:**

**return 1**

**return count\_leaves(node.left) + count\_leaves(node.right)**

**n = int(input()) # 读取节点数量**

**nodes = [TreeNode() for \_ in range(n)]**

**has\_parent = [False] \* n # 用来标记节点是否有父节点**

**for i in range(n):**

**left\_index, right\_index = map(int, input().split())**

**if left\_index != -1:**

**nodes[i].left = nodes[left\_index]**

**has\_parent[left\_index] = True**

**if right\_index != -1:**

**#print(right\_index)**

**nodes[i].right = nodes[right\_index]**

**has\_parent[right\_index] = True**

**root\_index = has\_parent.index(False)**

**root = nodes[root\_index]**

**height = tree\_height(root)**

**leaves = count\_leaves(root)**

**print(f"{height} {leaves}"**

**27653:Fraction类**

**def gcd(a, b):**

**while b != 0:**

**a, b = b, a % b**

**return a**

**class Fraction:**

**def \_\_init\_\_(self, numerator, denominator):**

**self.numerator = numerator**

**self.denominator = denominator**

**self.simplify()**

**def simplify(self):**

**common = gcd(self.numerator, self.denominator)**

**self.numerator //= common**

**self.denominator //= common**

**def \_\_add\_\_(self, other):**

**numerator = self.numerator \* other.denominator + self.denominator \* other.numerator**

**denominator = self.denominator \* other.denominator**

**return Fraction(numerator, denominator)**

**def \_\_str\_\_(self):**

**return f"{self.numerator}/{self.denominator}"**

**a, b, c, d = map(int, input().split())**

**x = Fraction(a, b)**

**y = Fraction(c, d)**

**print(x+y)**

**27862：博弈树分析与最优策略确定：获得最大收益的抉择路径**

**from collections import defaultdict**

**def f(x,layer):**

**if x in leaf:**

**return leaf[x]**

**l=dic[x]**

**for u in l:**

**if u in vis:**

**l.remove(u)**

**vis.update(l)**

**t=[]**

**for u in l:**

**t.append(f(u,layer+1))**

**if layer%2==1:**

**t.sort()**

**return t[-1]**

**else:**

**t.sort(key=lambda x:x[1])**

**return t[-1]**

**n=int(input())**

**vis=set([1])**

**dic=defaultdict(list)**

**for \_ in range(n-1):**

**a,b=map(int,input().split())**

**dic[a].append(b)**

**dic[b].append(a)**

**leaf={}**

**k=int(input())**

**for \_ in range(k):**

**a,b,c=map(int,input().split())**

**leaf[a]=(b,c)**

**print(\*f(1,1))**

**27880：繁忙的厦门**

**import sys**

**class UnionFind:**

**def \_\_init\_\_(self, n):**

**self.parent = list(range(n))**

**self.rank = [0] \* n**

**def find(self, x):**

**if self.parent[x] != x:**

**self.parent[x] = self.find(self.parent[x])**

**return self.parent[x]**

**def union(self, x, y):**

**px, py = self.find(x), self.find(y)**

**if px != py:**

**if self.rank[px] > self.rank[py]:**

**self.parent[py] = px**

**else:**

**self.parent[px] = py**

**if self.rank[px] == self.rank[py]:**

**self.rank[py] += 1**

**def kruskal(n, edges):**

**uf = UnionFind(n)**

**edges.sort(key=lambda x: x[2])**

**mst, max\_edge = 0, 0**

**for u, v, w in edges:**

**if uf.find(u) != uf.find(v):**

**uf.union(u, v)**

**mst += 1**

**max\_edge = max(max\_edge, w)**

**if mst == n - 1:**

**break**

**return mst, max\_edge**

**def main():**

**n, m = map(int, sys.stdin.readline().split())**

**edges = []**

**for \_ in range(m):**

**u, v, c = map(int, sys.stdin.readline().split())**

**edges.append((u - 1, v - 1, c))**

**mst, max\_edge = kruskal(n, edges)**

**print(f"{mst} {max\_edge}")**

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

**main()**

**27925：小組以列**

**from collections import deque**

**t = int(input())**

**groups = {}**

**member\_to\_group = {}**

**for \_ in range(t):**

**members = list(map(int, input().split()))**

**group\_id = members[0] # Assuming the first member's ID represents the group ID**

**groups[group\_id] = deque()**

**for member in members:**

**member\_to\_group[member] = group\_id**

**queue = deque()**

**queue\_set = set()**

**while True:**

**command = input().split()**

**if command[0] == 'STOP':**

**break**

**elif command[0] == 'ENQUEUE':**

**x = int(command[1])**

**group = member\_to\_group.get(x, None)**

**# Create a new group if it's a new member not in the initial list**

**if group is None:**

**group = x**

**groups[group] = deque([x])**

**member\_to\_group[x] = group**

**else:**

**groups[group].append(x)**

**if group not in queue\_set:**

**queue.append(group)**

**queue\_set.add(group)**

**elif command[0] == 'DEQUEUE':**

**if queue:**

**group = queue[0]**

**x = groups[group].popleft()**

**print(x)**

**if not groups[group]:**

**queue.popleft()**

**queue\_set.remove(group)**

**27928：遍历树**

**from collections import defaultdict**

**n = int(input())**

**tree = defaultdict(list)**

**parents = []**

**children = []**

**for i in range(n):**

**t = list(map(int, input().split()))**

**parents.append(t[0])**

**if len(t) > 1:**

**ch = t[1::]**

**children.extend(ch)**

**tree[t[0]].extend(ch)**

**def traversal(node):**

**seq = sorted(tree[node] + [node])**

**for x in seq:**

**if x == node:**

**print(node)**

**else:**

**traversal(x)**

**traversal((set(parents) - set(children)).pop())**

**27932: Less or Equal**

**n, k = map(int, input().split())**

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

**a.sort()**

**if k == 0:**

**x = 1 if a[0] > 1 else -1**

**elif k == n:**

**x = a[-1]**

**else:**

**x = a[k-1] if a[k-1] < a[k] else -1**

**print(x)**

**27947：动态中位数**

**import heapq**

**def dynamic\_median(nums):**

**min\_heap = [] # 存储较大的一半元素，使用最小堆**

**max\_heap = [] # 存储较小的一半元素，使用最大堆**

**median = []**

**for i, num in enumerate(nums):**

**if not max\_heap or num <= -max\_heap[0]:**

**heapq.heappush(max\_heap, -num)**

**else:**

**heapq.heappush(min\_heap, num)**

**if len(max\_heap) - len(min\_heap) > 1:**

**heapq.heappush(min\_heap, -heapq.heappop(max\_heap))**

**elif len(min\_heap) > len(max\_heap):**

**heapq.heappush(max\_heap, -heapq.heappop(min\_heap))**

**if i % 2 == 0:**

**median.append(-max\_heap[0])**

**return median**

**T = int(input())**

**for \_ in range(T):**

**#M = int(input())**

**nums = list(map(int, input().split()))**

**median = dynamic\_median(nums)**

**print(len(median))**

**print(\*median)**

**27948:FBI树**

**def construct\_FBI\_tree(s):**

**if '0' in s and '1' in s:**

**node\_type = 'F'**

**elif '1' in s:**

**node\_type = 'I'**

**else:**

**node\_type = 'B'**

**if len(s) > 1: # 如果字符串长度大于1，则继续分割**

**mid = len(s) // 2**

**left\_tree = construct\_FBI\_tree(s[:mid])**

**right\_tree = construct\_FBI\_tree(s[mid:])**

**return left\_tree + right\_tree + node\_type**

**else:**

**return node\_type**

**N = int(input())**

**s = input()**

**print(construct\_FBI\_tree(s))**

**27951：机器翻译**

**from collections import deque**

**M, N = map(int, input().split())**

**words = list(map(int, input().split()))**

**memory = deque()**

**lookups = 0**

**for word in words:**

**if word not in memory:**

**if len(memory) == M:**

**memory.popleft()**

**memory.append(word)**

**lookups += 1**

**print(lookups)**

**28046：词梯**

**from collections import defaultdict,deque**

**buckets=defaultdict(list)**

**for \_ in range(int(input())):**

**word=input()**

**for k in range(4):**

**buckets[word[:k]+' '+word[k+1:]].append(word)**

**x,y=input().split()**

**father={x:x}**

**q=deque([x])**

**while q:**

**word=q.popleft()**

**if word==y:break**

**for k in range(4):**

**for i in buckets[word[:k]+' '+word[k+1:]]:**

**if i not in father:**

**q.append(i)**

**father[i]=word**

**if word==y:**

**ans=[y]**

**while y!=x:**

**y=father[y]**

**ans.append(y)**

**print(' '.join(reversed(ans)))**

**else:print('NO')**

**28050：骑士周游**

**n = int(input()) # 输入棋盘的大小 n x n**

**p, q = map(int, input().split()) # 输入骑士的起始位置 (p, q)**

**if n < 5:**

**print("fail") # 如果棋盘大小小于5x5，直接判断为失败**

**else:**

**if n % 2 == 1: # 如果棋盘大小是奇数**

**if (p + q) % 2 == 0:**

**print("success") # 如果起始位置的坐标和为偶数，可以成功**

**else:**

**print("fail") # 否则失败**

**else:**

**print("success") # 如果棋盘大小是偶数，总是可以成功**

**28170：算度**

**def dfs(x,y):**

**graph[x][y] = "-"**

**for dx,dy in [(1,0),(-1,0),(0,1),(0,-1)]:**

**if 0<=x+dx<10 and 0<=y+dy<10 and graph[x+dx][y+dy] == ".":**

**dfs(x+dx,y+dy)**

**graph = []**

**result = 0**

**for i in range(10):**

**graph.append(list(input()))**

**for i in range(10):**

**for j in range(10):**

**if graph[i][j] == ".":**

**result += 1**

**dfs(i,j)**

**print(result)**

**28190：奶牛排队**

**from bisect import bisect\_right as bl**

**lis,q1,q2,ans=[int(input())for \_ in range(int(input()))],[-1],[-1],0**

**for i in range(len(lis)):**

**while len(q1)>1 and lis[q1[-1]]>=lis[i]:q1.pop()**

**while len(q2)>1 and lis[q2[-1]]<lis[i]:q2.pop()**

**id=bl(q1,q2[-1])**

**if id<len(q1):ans=max(ans,i-q1[id]+1)**

**q1.append(i)**

**q2.append(i)**

**print(ans)**

**28203：【模板】单调栈**

**n = int(input())**

**ans = [0 for \_ in range(n)]**

**l = list(map(int, input().split()))**

**stack = []**

**i = 0**

**while i < n:**

**while stack and l[i] > l[stack[-1]]:**

**ans[stack.pop()] = i + 1**

**stack.append(i)**

**i += 1**

**print(\*ans)**

**29468.实现散列表**

**def is\_prime(num):**

**if num < 2:**

**return False**

**for i in range(2, int(num\*\*0.5) + 1):**

**if num % i == 0:**

**return False**

**return True**

**def next\_prime(n):**

**while not is\_prime(n):**

**n += 1**

**return n**

**def quadratic\_probing\_insert(table, key, table\_size):**

**hash\_val = key % table\_size**

**if table[hash\_val] == key:**

**return hash\_val**

**elif table[hash\_val] is None:**

**table[hash\_val] = key**

**return hash\_val**

**else:**

**i = 1**

**while i < table\_size:**

**new\_pos = (hash\_val + i \* i) % table\_size**

**if table[new\_pos] == key:**

**return new\_pos**

**elif table[new\_pos] is None:**

**table[new\_pos] = key**

**return new\_pos**

**i += 1**

**return -1 # 插入失败**

**import sys**

**input = sys.stdin.read().splitlines()**

**N = int(input[0])**

**numbers = list(map(int, input[1].split()))**

**table\_size = next\_prime(N)**

**hash\_table = [None] \* table\_size**

**result = []**

**for num in numbers:**

**pos = quadratic\_probing\_insert(hash\_table, num, table\_size)**

**if pos == -1:**

**result.append("-")**

**else:**

**result.append(str(pos))**

**print(" ".join(result))**