```
# import sys 中的 sys.exit(), 退出程序。
# a[-i] 倒数第i个
# A = list(map(lambda x: int(x) % b, input().split())) 生成一个对输入元素取模的列表
# print("{:.nf}".format(x)) print('%.nf' % x) 输出n位小数
# print("%d:%s->%s" % (n, a, b)) 格式化输出
# students = sorted(range(1, n + 1), key=lambda x: experiment_times[x - 1]) 根据列表中的值增序对学生的序号排序
# pairs = [i[1:-1] for i in input().split()] 特殊输入处理
# from collections import deque deque.popleft() deque.pop() deque.append() deque.appendleft() deque.extend() deque.extendleft() d.rotate(2) 向右循环2次
# enumerate(list, start=0) 返回他的索引和索引对应的值
# heapq heapq.heappush(list, item) heapify(heap)建立大、小根堆 heappop(heap) 返回堆顶的最小值 heappushpop(heap,item)
# team_member = {person: i for i, team in teams.items() for person in team} 建立成员对应小组的队列
# 集合比list快
# sorted(moves, key=lambda move: count_valid_moves(move[0], move[1]))
# tuple 元组
# from collections import defaultdict 对字典默认排序是对键排序
# bisect.bisect_left bisect.insort_left
```

```
# T-primes
def is_prime(n):
    prime = [True] * (n + 1)
    prime[0] = prime[1] = False
    for i in range(2, int(n ** 0.5) + 1):
        if prime[i]:
            for j in range(i * i, n + 1, i):
                prime[j] = False
    return prime
```

```python
# 拦截导弹
n = int(input())
lst = list(map(int, input().split()))
dp = [0] * 30
for i in range(n):
    dp[i] = 1
    for j in range(i):
        if lst[j] >= lst[i]:
            dp[i] = max(dp[i], dp[j] + 1)
ans = dp[0]
for i in range(n):
    ans = max(ans, dp[i])
print(ans)
```

```python
# 模型整理
from collections import defaultdict
n = int(input())
d = defaultdict(list)
for _ in range(n):
    name, para = input().split('-')
    if para[-1]=='M':
        d[name].append((para, float(para[:-1])/1000) )
    else:
        d[name].append((para, float(para[:-1])))
sd = sorted(d)
for k in sd:
    paras = sorted(d[k],key=lambda x: x[1])
    value = ', '.join([i[0] for i in paras])
    print(f'{k}: {value}')
```

```python
# 后序表达式求值
#小数改良版，波兰表达式则须逆序以后再求解
def doMath(op, op1, op2):
    if op == '+': return op1 + op2
```

```
    elif op == '-': return op1 - op2
    elif op == '*': return op1 * op2
    else: return op1 / op2
def f(s):
    stack = []
    for i in s:
        if i in '+-*/':
            y = stack.pop(); x = stack.pop(); z = doMath(i, x, y)
            stack.append(z)
        elif float(i).is_integer(): #判断是否是整数类型的字符
            stack.append(int(i)) #isalpha判断是否是英文单词
        elif i not in '+-*/':
            stack.append(float(i))
    return stack[0]
```

```
# 中序表达式转后序
def f(s):
    prec = {}; prec["*"] = 3; prec["/"] = 3; prec["+"] = 2; prec["-"] =
2; prec["("] = 1 #我们用一个字典来保存运算符的优先级来用于比较，“（”的优先级最
小，保证运算符可以压进栈。
    stack = []
    res = []
    for i in s:
        if i not in '+-*/()':
            res.append(i)
        elif i == '(':
            stack.append(i)
        elif i == ')':
            j = stack.pop()
            while j != '(':
                res.append(j)
                j = stack.pop()
        else:
            while stack and prec[stack[-1]] >= prec[i]:
                res.append(stack.pop())
```

```
                stack.append(i)
        while stack:
            res.append(stack.pop())
        return res
    # 后序表达式转中序，遇到一个运算符就取出栈中两个数，转化为一个数加上括号放入栈中
```

通过观察不难发现我们在后序表达式中每次遇到“+”，“-”，“*”，“/”时都要处理对应的前2个数，例如2 3 *

处理为（2 * 3）此时 （2 * 3）就变成了一个数 可以供后面的符号处理。

这样一来我们的思路变清晰了：

即每次遇到“+”，“-”，“*”，“/”符号 就对前2个数作处理，处理后将得到的数压入栈中

如我们遇到的是数则直接压入栈中即可

```
# 合法出栈序列
def is_valid_pop_sequence(origin, output):
    if len(origin) != len(output):
        return False
    stack = []
    bank = list(origin)
    for char in output:
        while (not stack or stack[-1] != char) and bank:
            stack.append(bank.pop(0))
        if not stack or stack[-1] != char:
            return False
        stack.pop()
    return True
origin = input().strip()
while True:
    try:
        output = input().strip()
        if is_valid_pop_sequence(origin, output):
            print('YES')
        else:
            print('NO')
```

```
    except EOFError:
        break
```

```
# 归并排序
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
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_merged=merge(left,right)
    return merged,inv_left+inv_right+inv_merged
  # 求逆序对
  def merge_sort(i, j):
    if j <= i:
        return 0
    mid = (i + j) >> 1
    t = merge_sort(i, mid) + merge_sort(mid + 1, j)
    temp = ls[i: j + 1]
    mid -= i
```

```
        l, r = 0, mid + 1
        for idx in range(i, j + 1):
            if l > mid:
                ls[idx] = temp[r]
                r += 1
            elif r > j - i:
                ls[idx] = temp[l]
                l += 1
            elif temp[l] <= temp[r]:
                ls[idx] = temp[l]
                l += 1
            else:
                ls[idx] = temp[r]
                r += 1
                t += mid - l + 1
        return t
```

```
# 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): #eval函数执行字符串的运行结果
        print('V')
    else:
        print('F')
```

```
# 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 = TreeNode(char)
            if stack:  # 如果栈不为空，把节点作为子节点加入到栈顶节点的子节点列表中
                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
```

```
# 文件结构图
from sys import exit
class dir:
    def __init__(self, dname):
        self.name = dname
        self.dirs = []
        self.files = []
    def getGraph(self):
        g = [self.name]
        for d in self.dirs:
            subg = d.getGraph()
            g.extend(["|     " + s for s in subg])
        for f in sorted(self.files):
            g.append(f)
        return g
n = 0
while True:
    n += 1
    stack = [dir("ROOT")]
    while (s := input()) != "*":
        if s == "#": exit(0)
```

```
        if s[0] == 'f':
            stack[-1].files.append(s)
        elif s[0] == 'd':
            stack.append(dir(s))
            stack[-2].dirs.append(stack[-1])
        else:
            stack.pop()
    print(f"DATA SET {n}:")
    print(*stack[0].getGraph(), sep='\n')
    print()
```

```
# 后序表达式建立表达式树（队列表达式）
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):
    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
```

```
# 二叉搜索树（中序即为顺序遍历）
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]

def buildTree(node, value):#建立平衡二叉树
    if node is None:
        return TreeNode(value)
    if value > node.value:
        node.right = buildTree(node.right, value)
    elif value < node.value:
        node.left = buildTree(node.left, value)
    return node
```

```
# 并查集
class DisjointSet:
    def __init__(self, n):
        self.parent = [i for i in range(n + 1)]
        self.rank = [0] * (n + 1)
    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):
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x == root_y:
            return
        if self.rank[root_x] < self.rank[root_y]:
            self.parent[root_x] = root_y
        elif self.rank[root_x] > self.rank[root_y]:
            self.parent[root_y] = root_x
```

```
        else:
            self.parent[root_y] = root_x
            self.rank[root_x] += 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
```

```
# FBI树（建树与遍历的结合）
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:
        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
```

```
# 小组队列
from collections import deque
t = int(input())
teams = {i: deque(map(int, input().split())) for i in range(t)}
team_member = {person: i for i, team in teams.items() for person in
team}
```

```
queue = deque()
group_queue ={i: deque() for i in range(t)}
while True:
    com = input().split()
    if com[0] == 'STOP':
        break
    elif com[0] == 'ENQUEUE':
        person = int(com[1])
        if person in team_member:
            i = team_member[person]
            group_queue[i].append(person)
            if i not in queue:
                queue.append(i)
        else:
            t += 1
            group_queue[t] = deque([person])
            queue.append(t)
    elif com[0] == 'DEQUEUE':
        group = queue[0]
        print(group_queue[group].popleft())
        if not group_queue[group]:
            queue.popleft()
```

```
# 遍历树（字典代替树）
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())
```

```
# 树的镜面映射
n = int(input())
l = list(input().split())
stack = []
for i in l:
    if i[1] == '0':
        stack.append(i[0])
        stack.append('(')
    if i[1] == '1':
        stack.append(i[0])
        stack.append(')')
level, dic = 0, {}
for j in stack:
    if j == '(':
        level += 1
    elif j == ')':
        level -= 1
    else:
        if j != '$':
            if level not in dic:
                dic[level] = [j]
            else:
                dic[level].append(j)
a, ans = 0, []
while a in dic:
```

```
        ans.extend(reversed(dic[a]))
        a += 1
    print(' '.join(ans))
```

```
    # 树的转换求高度
    s = input()
    h = a = b = 0
    H = [0] * len(s)
    for c in s:
        if c == 'd':
            h += 1
            H[h]= H[h-1]+1
            a = max(a, h)
            b = max(b, H[h])
        else:
            h -= 1
            H[h] += 1
    print('%d => %d' % (a, b))
```

```
    # 补齐二叉树的递归建树
    def build_tree(lst):
        if not lst:
            return None
        value = lst.pop()
        if value == '.':
            return None
        root = BinaryTreeNode(value)
        root.left = build_tree(lst)
        root.right = build_tree(lst)
        return root
```

```
    # 八皇后
    def dfs(s):
        for j in range(1, 9):
```

```
        for i in range(len(s)):
            if str(j) == s[i] or abs(j - int(s[i])) == abs(len(s) - i):
                break
        else:
            if len(s) == 7:
                res.append(s + str(j))
            else:
                dfs(s + str(j))
res = []
dfs('')
N = int(input())
for _ in range(N):
    num = int(input())
    print(res[num - 1])
```

```
# 交换二叉树两个节点
def swap(nodes, x, y):
    for node in nodes:
        if node.left and node.left.val in[x,y]:
            node.left = nodes[y] if node.left.val == x else nodes[x]
        if node.right and node.right.val in[x,y]:
            node.right = nodes[y] if node.right.val == x else nodes[x]
```

```
# 无向图是否连通有回路
def is_connected(graph, n):
    visited = [False] * n
    stack = [0]
    visited[0] = True
    while stack:
        node = stack.pop()
        for neighbor in graph[node]:
            if not visited[neighbor]:
                stack.append(neighbor)
                visited[neighbor] = True
    return all(visited)
```

```python
def has_cycle(graph, n):
    def dfs(node, visited, parent):
        visited[node] = True
        for neighbor in graph[node]:
            if not visited[neighbor]:
                if dfs(neighbor, visited, node):
                    return True
            elif parent != neighbor:
                return True
        return False
    visited = [False] * n
    for node in range(n):
        if not visited[node]:
            if dfs(node, visited, -1):
                return True
    return False
  def dfs(x, road): #有向
    vis[x] = 1
    flag = 0
    for nei in M[x]:
        if vis[nei] == 0 and M[nei]:
            flag = dfs(nei, road + [nei])
        elif nei in road:
            return 1
        if flag == 1:
            return 1
    return 0
```

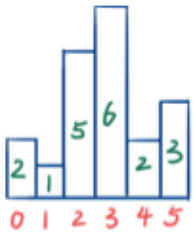
```python
# 单调栈
n = int(input())
a = list(map(int, input().split()))
stack = []
res = [0] * n
for i in range(n - 1, -1, -1):
    while stack and a[i] >= a[stack[-1]]:
```

```
            stack.pop()
        if stack and a[i] < a[stack[-1]]:
            res[i] = stack[-1] + 1
        stack.append(i)
    print(*res)
    #接雨水
    def trap(height):
        if not height:
            return 0
        n = len(height)
        water = 0
        stack = []
        for i in range(n):
            while stack and height[i] > height[stack[-1]]:
                top = stack.pop()
                if not stack:
                    break
                distance = i - stack[-1] - 1
                bounded_height = min(height[i], height[stack[-1]]) -
    height[top]
                water += distance * bounded_height
            stack.append(i)
        return water
    n = int(input())
    height = list(map(int, input().split()))
    print(trap(height))
     #矩形
```

过程示例

注：下面的图是来自：逗比克星

数组



栈初始化 [-1]

步骤	栈	当前最大矩阵面积	注释
0	[-1, 0]	0	0入栈
1	[-1]	2	高度1比2矮，0出栈，以 2 为顶的最大矩形面积为2* (1- (-1) - 1) = 2
	[-1, 1]	2	1入栈
2	[-1, 1, 2]	2	高度5比1高，2入栈
3	[-1, 1, 2, 3]	2	高度6比5高，3入栈
4	[-1, 1, 2]	6	高度2比6矮，3出栈，以 6 为顶的最大矩形面积为6* (4-2-1) = 6
	[-1, 1]	10	高度2比5矮，2出栈，以 5 为顶的最大矩形面积为5* (4-1-1) = 10
	[-1, 1, 4]	10	高度2比1高，4入栈
5	[-1, 1, 4, 5]	10	高度3比2高，5入栈

```python
# 二分法
def check(x) #查看二分法的答案是否满足
n, m = map(int, input().split())
out = [int(input()) for _ in range(n)]
left, right, ans = max(out), sum(out), 0
while left <= right:
    mid = (left + right) // 2
    if check(mid):
        ans = mid
        right = mid - 1
    else:
        left = mid + 1
print(ans)

import bisect
n = int(input())
l = list(map(int, input().split()))
```

```
ll = []
ans = 0
for v in l:
    x = len(ll) - bisect.bisect_right(ll, -v)
    ans += x
    bisect.insort_right(ll, -v)
print(ans)
```

```
# 拓扑排序
v,a=map(int,input().split())
node=["v"+str(i) for i in range(v+1)]
dic1={i:0 for i in node}
dic2={i:[] for i in node}
for _ in range(a):
    f,t=map(int,input().split())
    dic1[node[t]]+=1
    dic2[node[f]].append(node[t])
vis=set()
cnt=0
ans=[]
while cnt<v:
    for i in range(1,v+1):
        if dic1[node[i]]==0 and node[i] not in vis:
            vis.add(node[i])
            ans.append(node[i])
            cnt+=1
            for nodes in dic2[node[i]]:
                dic1[nodes]-=1
            break
print(*ans)
```

```
# 左儿子右兄弟
from collections import defaultdict
n=int(input())
nodes=[(x,int(i)) for x,i in input().split()]
```

```
output=defaultdict(list)
t=0
last=0
ma=0
for x,i in nodes:
    if last == 1:
        t-=1
    else:
        t+=1
    if x != '$':
        output[t].append(x)
        ma=max(ma,t)
    last=i
print(' '.join([' '.join(output[i][::-1])for i in range(1,ma+1)]))
```

```
# 滑动窗口
n = len(nums)
q = collections.deque()
for i in range(k):
    while q and nums[i] >= nums[q[-1]]:
        q.pop()
    q.append(i)
ans = [nums[q[0]]]
for i in range(k, n):
    while q and nums[i] >= nums[q[-1]]:
        q.pop()
    q.append(i)
    while q[0] <= i - k:
        q.popleft()
    ans.append(nums[q[0]])
return ans
```

```
# 文本二叉树
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:
                parent = tree[stack[-1]]
                if not parent.lchild:
                    parent.lchild = node
                else:
                    parent.rchild = node
            stack.append(len(tree) - 1)
        yield tree[0]
```

```
# 字典树
class TrieNode:
    def __init__(self):
        self.child={}
class Trie:
    def __init__(self):
        self.root = TrieNode()
    def insert(self, nums):
        curnode = self.root
        for x in nums:
            if x not in curnode.child:
                curnode.child[x] = TrieNode()
            curnode=curnode.child[x]
    def search(self, num):
```

```
        curnode = self.root
        for x in num:
            if x not in curnode.child:
                return 0
            curnode = curnode.child[x]
        return 1
```

```
# Kosaraju's算法（有向图连通域）
def dfs1(graph, node, visited, stack):  # 第一个深度优先搜索函数，用于遍历图并将节点按完成时间压入栈中
    visited[node] = True  # 标记当前节点为已访问
    for neighbor in graph[node]:  # 遍历当前节点的邻居节点
        if not visited[neighbor]:  # 如果邻居节点未被访问过
            dfs1(graph, neighbor, visited, stack)  # 递归调用深度优先搜索函数
    stack.append(node)  # 将当前节点压入栈中，记录完成时间
def dfs2(graph, node, visited, component):  # 第二个深度优先搜索函数，用于在转置后的图上查找强连通分量
    visited[node] = True  # 标记当前节点为已访问
    component.append(node)  # 将当前节点添加到当前强连通分量中
    for neighbor in graph[node]:  # 遍历当前节点的邻居节点
        if not visited[neighbor]:  # 如果邻居节点未被访问过
            dfs2(graph, neighbor, visited, component)  # 递归调用深度优先搜索函数
def kosaraju(graph):  # Kosaraju's 算法函数
    # Step 1: 执行第一次深度优先搜索以获取完成时间
    stack = []  # 用于存储节点的栈
    visited = [False] * len(graph)  # 记录节点是否被访问过的列表
    for node in range(len(graph)):  # 遍历所有节点
        if not visited[node]:  # 如果节点未被访问过
            dfs1(graph, node, visited, stack)  # 调用第一个深度优先搜索函数
    # Step 2: 转置图
    transposed_graph = [[] for _ in range(len(graph))]  # 创建一个转置后的图
    for node in range(len(graph)):  # 遍历原图中的所有节点
```

```
        for neighbor in graph[node]:  # 遍历每个节点的邻居节点
            transposed_graph[neighbor].append(node)  # 将原图中的边反向添加到转置图中
    # Step 3: 在转置后的图上执行第二次深度优先搜索以找到强连通分量
    visited = [False] * len(graph)  # 重新初始化节点是否被访问过的列表
    sccs = []  # 存储强连通分量的列表
    while stack:  # 当栈不为空时循环
        node = stack.pop()  # 从栈中弹出一个节点
        if not visited[node]:  # 如果节点未被访问过
            scc = []  # 创建一个新的强连通分量列表
            dfs2(transposed_graph, node, visited, scc)  # 在转置图上执行深度优先搜索
            sccs.append(scc)  # 将找到的强连通分量添加到结果列表中
    return sccs  # 返回所有强连通分量的列表
```

```
# prim
from heapq import heappop, heappush
def prim(matrix):
    ans=0
    pq,visited=[(0,0)],[False for _ in range(N)]
    while pq:
        c,cur=heappop(pq)
        if visited[cur]:continue
        visited[cur]=True
        ans+=c
        for i in range(N):
            if not visited[i] and matrix[cur][i]!=0:
                heappush(pq,(matrix[cur][i],i))
    return ans
while True:
    try:
        N=int(input())
        matrix=[list(map(int,input().split())) for _ in range(N)]
        print(prim(matrix))
    except:break
```

```python
# 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
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))
```

```python
# dij
import heapq
```

```
def dijkstra(N, G, start):
    INF = float('inf')
    dist = [INF] * (N + 1) # 存储源点到各个节点的最短距离
    dist[start] = 0 # 源点到自身的距离为0
    pq = [(0, start)] # 使用优先队列，存储节点的最短距离
    while pq:
        d, node = heapq.heappop(pq) # 弹出当前最短距离的节点
        if d > dist[node]: # 如果该节点已经被更新过了，则跳过
            continue
       for neighbor, weight in G[node]: # 遍历当前节点的所有邻居节点
          new_dist = dist[node] + weight # 计算经当前节点到达邻居节点的距离
             if new_dist < dist[neighbor]: # 如果新距离小于已知最短距离，则更新最短距离
                dist[neighbor] = new_dist
                heapq.heappush(pq, (new_dist, neighbor)) # 将邻居节点加入优先队列
    return dist
N, M = map(int, input().split())
G = [[] for _ in range(N + 1)] # 图的邻接表表示
for _ in range(M):
    s, e, w = map(int, input().split())
    G[s].append((e, w))
start_node = 1 # 源点
shortest_distances = dijkstra(N, G, start_node) # 计算源点到各个节点的最短距离
print(shortest_distances[-1])
# 兔子与樱花
import heapq
def prim(graph):
    # 初始化最小生成树的顶点集合和边集合
  mst = set()
  edges = []
  visited = set()
  total_weight = 0
    # 随机选择一个起始顶点
  start_vertex = list(graph.keys())[0]
```

```
    # 将起始顶点加入最小生成树的顶点集合中
    mst.add(start_vertex)
    visited.add(start_vertex)
    # 将起始顶点的所有边加入边集合中
    for neighbor, weight in graph[start_vertex]:
      heapq.heappush(edges, (weight, start_vertex, neighbor))
    # 循环直到所有顶点都加入最小生成树为止
    while len(mst) < len(graph):
      # 从边集合中选取权重最小的边
      weight, u, v = heapq.heappop(edges)
      # 如果边的目标顶点已经在最小生成树中，则跳过
      if v in visited:
        continue
      # 将目标顶点加入最小生成树的顶点集合中
      mst.add(v)
      visited.add(v)
      total_weight += weight
      # 将目标顶点的所有边加入边集合中
      for neighbor, weight in graph[v]:
        if neighbor not in visited:
          heapq.heappush(edges, (weight, v, neighbor))
    return total_weight
n = int(input())
graph = {}
for _ in range(n - 1):
    alist = list(input().split())
    if alist[0] not in graph.keys():
    graph[alist[0]] = []
  for i in range(1, int(alist[1]) + 1):
    if alist[2 * i] not in graph.keys():
      graph[alist[2 * i]] = []
    graph[alist[0]].append((alist[2 * i], int(alist[2 * i + 1])))
    graph[alist[2 * i]].append((alist[0], int(alist[2 * i + 1])))
print(prim(graph))
# 走山路
from heapq import heappop, heappush
```

```
    def bfs(x1, y1):
        q = [(0, x1, y1)]
        v = set()
        while q:
            t, x, y = heappop(q)
            v.add((x, y))
            if x == x2 and y == y2:
                return t
            for dx, dy in dir:
                nx, ny = x+dx, y+dy
                if 0 <= nx < m and 0 <= ny < n and ma[nx][ny] != '#' and
    (nx, ny) not in v:
                    nt = t+abs(int(ma[nx][ny])-int(ma[x][y]))
                    heappush(q, (nt, nx, ny))
        return 'NO'
    m, n, p = map(int, input().split())
    ma = [list(input().split()) for _ in range(m)]
    dir = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    for _ in range(p):
        x1, y1, x2, y2 = map(int, input().split())
        if ma[x1][y1] == '#' or ma[x2][y2] == '#':
            print('NO')
            continue
        print(bfs(x1, y1))
    # 道路
    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))
pq = [(0,0,0)]
dists = [float('inf')] * n
dists[0] = 0
spend = 0
result = dijkstra(g)
print(result)
```

Dijkstra

```
def dijkstra(graph: List[List[int]], n: int, k: int):
  # 稀疏图用邻接表graph, 其中graph[node]存储以(nb, w)形式存储邻居和边权
  expanded = [False for _ in range(n)]
  curDist = [float('inf') for _ in range(n)]
  curDist[k] = 0

  h = [(0, k)]
  while h:
    # 1. 找到没有扩展过的点中到起点距离最短的点node
    node = heappop(h)
    if not expanded[node]: # 确保不重复扩展
      # 2. 扩展
      for nb, w in graph[node]:
        # 扩展意味着搜索树中nb(或暂时成为, 取决于搜索树类型)node的子节点
        newDist = curDist[node] + w
        if newDist < curDist[nb]: # 剪枝
          curDist[nb] = newDist
          heappush(h, (newDist, nb))
      # 3. 扩展完标记该点
      expanded[node] = True
```

记录路径

```python
import heapq

def dijkstra(adjacency, start):
    # 初始化，将其余所有顶点到起始点的距离都设为inf（无穷大）
    distances = {vertex: float('inf') for vertex in adjacency}
    # 初始化，所有点的前一步都是None
    previous = {vertex: None for vertex in adjacency}
    # 起点到自身的距离为0
    distances[start] = 0
    # 优先队列
```

```python
    pq = [(0, start)]

    while pq:
        # 取出优先队列中，目前距离最小的
        current_distance, current_vertex = heapq.heappop(pq)
```

```
        # 剪枝，如果优先队列里保存的距离大于目前更新后的距离，则可以跳过
        if current_distance > distances[current_vertex]:
            continue

        # 对当前节点的所有邻居，如果距离更优，将他们放入优先队列中
        for neighbor, weight in adjacency[current_vertex].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                # 这一步用来记录每个节点的前一步
                previous[neighbor] = current_vertex
                heapq.heappush(pq, (distance, neighbor))

    return distances, previous

def shortest_path_to(adjacency, start, end):
    # 逐步访问每个节点上一步
    distances, previous = dijkstra(adjacency, start)
    path = []
    current = end
    while previous[current] is not None:
        path.insert(0, current)
        current = previous[current]
    path.insert(0, start)
    return path, distances[end]
```

```python
#Read the input data
P = int(input())
places = {input().strip() for _ in range(P)}

Q = int(input())
graph = {place: {} for place in places}
for _ in range(Q):
    src, dest, dist = input().split()
    dist = int(dist)
    graph[src][dest] = dist
    graph[dest][src] = dist  # Assuming the graph is bidirectional

R = int(input())
requests = [input().split() for _ in range(R)]
```

```python
#Process each request
for start, end in requests:
    if start == end:
        print(start)
        continue

    path, total_dist = shortest_path_to(graph, start, end)
    output = ""
    for i in range(len(path) - 1):
        output += f"{path[i]}->({graph[path[i]][path[i+1]]})->"
    output += f"{end}"
    print(output)
```

Floyd

```python
def floyd_warshall(graph):
    n = len(graph)
    dist = [[float('inf')] * n for _ in range(n)]

    for i in range(n):
        for j in range(n):
            if i == j:
                dist[i][j] = 0
            elif j in graph[i]:
                dist[i][j] = graph[i][j]

    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

    return dist
```

Prim

```python
def prim(graph: List[List[int]], n: int):
  # 稠密图用邻接矩阵graph, 其中存边权, 无边存无穷大
  curDist = [float('inf') for _ in range(n)] # 点到当前树的最小距离, 是边权
  inMST = [False for _ in range(n)] # 标记是否已加入到MST中
```

```python
  totalWeight = 0

  for _ in range(n): # 每次加一个点一条边到树中(第一次只加点不加边)
    # 1. 通过枚举点找到连接树和树外一点的最短边
    minNode = None
    for node in range(n):
      if not inMST[node] and (minNode is None
          \ or curDist[node] < curDist[minNode]):
        minNode = nodei

    # 2. 把最短边及其连接的树外点加入到MST中(第一次循环只加点不加边)
    if i != 0:  # 当然也可将起点的 curDist 初始化为 0, 则此处无需判断
```

```
        totalWeight += curDist[minNode]
        # 如果这条最短边为inf，就代表该树外点与树中任一点都不连通，即原图是不连通的
      inMST[minNode] = True

      # 3. 更新树外节点到树的最小距离
      for nb in graph[minNode]:
        curDist[nb] = min(curDist[nb], graph[minNode][nb])

    return totalWeight
```

```
from heapq import *

while True:
    n = int(input())
    if n == 0:
        break
    trucks = [input() for _ in range(n)]
    trucks.sort()

    sd = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        for j in range(i + 1, n):
            sd[i][j] = sd[j][i] = sum(a != b for a, b in zip(trucks[i],
trucks[j]))
```

```
# Prim
v = [False for _ in range(n)]
dis = [float('inf') for _ in range(n)]
dis[0] = 0
q = [(0, 0)]
total_weight = 0
while q:
```

```
    weight, node = heappop(q)
    if v[node]:
        continue
    v[node] = True
    total_weight += weight
    for nb in range(n):
        if nb != node and not v[nb] and dis[nb] > sd[nb][node]:
            dis[nb] = sd[nb][node]
            heappush(q, (dis[nb], nb))

print(f'The highest possible quality is 1/{total_weight}.')
```

拓扑排序

DFS

```
    def dfs(num):
        v[num] = 1
        for neighbour in g[num]:
            if v[neighbour] == 0 and dfs(neighbour):
                return True
            if v[neighbour] == 1:
                return True
        v[num] = 2
        ans.append(num) # 最后要反转
        return False
```

Kahn

```python
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]:
```

```python
            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
```

```python
class Node:
    def __init__(self, val):
        self.val = val
        self.children = []
def build(n, nodes):
    tree = [Node(i) for i in range(n+1)]
    for a, b in nodes:
        tree[a].children.append(tree[b])
    return tree[1]
def cal(root):
    if root.children:
        if root.val == 1:
            cnt = 1
            for child in root.children:
```

```
                cnt += cal(child)
            return cnt//2
        cnt = 1
        for child in root.children:
            cnt += cal(child)
        if cnt % 2:
            return cnt-1
        else:
            return cnt
    else:
        return 1
n = int(input())
nodes = [list(map(int, input().split())) for _ in range(n-1)]
root = build(n, nodes)
print(cal(root))
```