pylint: skip-file

重要的库

1.二分查找 bisect

```
import bisect
bisect_right(list, num, lo=0, hi=len(list)) #返回大于x的第一个下标
bisect_left(list, num, lo=0, hi=len(list)) #返回大于等于x的第一个下标
insort_right(list, num, lo=0, hi=len(list)) #如果大小相同插入右边的位置
insort_left(list, num, lo=0, hi=len(list)) #如果大小相同插入左边的位置
```

2.deque

```
from collections import deque
d = deque()
d.appendleft('a')
d.popleft()
d.extendleft("123")
```

3.heapq

```
import heapq
a = []   #创建一个空堆
heapq.heappush(a,18)
a = [1, 5, 20, 18, 10, 200]
heapq.heapify(a)
heapq.heappop(a) #从堆中弹出并返回最小的值
heapq.nlargest(2,a,key=lambda x:x[1]) #返回最大的n个
a[0] #返回最小的1个
heappushpop(heap,item) #先push后Pop
heapreplace(heap.item) #先pop后push，更快
heapq.merge() #合并多个堆然后输出
```

4. OrderedDict

```
from collections import OrderedDict
unique_list = list(OrderedDict.fromkeys(my_list)) #可以去重并且保持顺序
```

5.Counter

```
from collections import Counter
list1 = ["a", "a", "a", "b", "c", "c", "f", "g", "g", "g", "f"]
dic = Counter(list1)
print(dic)
#结果:次数是从高到低的
#Counter({'a': 3, 'g': 3, 'c': 2, 'f': 2, 'b': 1})

print(dict(dic))
#结果:按字母顺序排序的
#{'a': 3, 'b': 1, 'c': 2, 'f': 2, 'g': 3}
```

6.defaultdict

```
from collections import defaultdict
class node():
    pass
d = defaultdict(node())
#第一个参数为default_factory属性提供初始值，默认为None。会为一个不存在的键提供默认值如int，从而避免KeyError异常
```

7. itertools

```
import itertools
# 1. product
for i in itertools.product([1, 2], [3, 4]):
    print(i)
# (1, 3) (1, 4) (2, 3) (2, 4)
# 2. permutations
for i in itertools.permutations([1, 2, 3], 2):
    print(i)
# (1, 2) (1, 3) (2, 1) (2, 3)
```

知识点和代码

1.类的重构

```
__le__  #小于等于
__eq__ #深相等—根据值来判断相等
```

2.栈

调度场算法
中缀表达式转换为后缀表达式
以下是 Shunting Yard 算法的基本步骤：

1.初始化运算符栈和输出栈为空。
2.从左到右遍历中缀表达式的每个符号。
-如果是操作数（数字），则将其添加到输出栈。
-如果是左括号，则将其推入运算符栈。
-如果是运算符：
----如果运算符的优先级大于运算符栈顶的运算符，或者运算符栈顶是左括号，则将当前运算符推入运算符栈。
----否则，将运算符栈顶的运算符弹出并添加到输出栈中，直到满足上述条件（或者运算符栈为空）。
----将当前运算符推入运算符栈。
-如果是右括号，则将运算符栈顶的运算符弹出并添加到输出栈中，直到遇到左括号。将左括号弹出但不添加到输出栈中。
3.如果还有剩余的运算符在运算符栈中，将它们依次弹出并添加到输出栈中。

```
def infix_to_postfix(expression):
    precedence = {'+':1, '-':1, '*':2, '/':2}
    stack = []
    postfix = []
    number = ''

    for char in expression:
        if char.isnumeric() or char == '.':
            number += char
        else:
            if number:
                num = float(number)
                postfix.append(int(num) if num.is_integer() else num)
                number = ''
            if char in '+-*/':
                while stack and stack[-1] in '+-*/' and precedence[char] <=
precedence[stack[-1]]:
                    postfix.append(stack.pop())
                stack.append(char)
            elif char == '(':
                stack.append(char)
            elif char == ')':
                while stack and stack[-1] != '(':
                    postfix.append(stack.pop())
                stack.pop()

    if number:
        num = float(number)
        postfix.append(int(num) if num.is_integer() else num)

    while stack:
        postfix.append(stack.pop())

    return ' '.join(str(x) for x in postfix)
```

```
n = int(input())
for _ in range(n):
    expression = input()
    print(infix_to_postfix(expression))
```

表达式求值

```
eval('1+2*3')
```

3.树

哈夫曼编码实现
要构建一个最优的哈夫曼编码树，首先需要对给定的字符及其权值进行排序。然后，通过重复合并权值最小的两个节点（或子树），直到所有节点都合并为一棵树为止。
二叉搜索树

```
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)))
```

AVL树：

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
        self.height = 1

class AVL:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if not self.root:
            self.root = Node(value)
        else:
            self.root = self._insert(value, self.root)

    def _insert(self, value, node):
        if not node:
            return Node(value)
        elif value < node.value:
            node.left = self._insert(value, node.left)
        else:
            node.right = self._insert(value, node.right)

        node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))

        balance = self._get_balance(node)

        if balance > 1:
            if value < node.left.value: # 树形是 LL
                return self._rotate_right(node)
            else:   # 树形是 LR
                node.left = self._rotate_left(node.left)
                return self._rotate_right(node)

        if balance < -1:
            if value > node.right.value:    # 树形是 RR
                return self._rotate_left(node)
            else:   # 树形是 RL
                node.right = self._rotate_right(node.right)
                return self._rotate_left(node)

        return node

    def _get_height(self, node):
```

```
        if not node:
            return 0
        return node.height

    def _get_balance(self, node):
        if not node:
            return 0
        return self._get_height(node.left) - self._get_height(node.right)

    def _rotate_left(self, z):
        y = z.right
        T2 = y.left
        y.left = z
        z.right = T2
        z.height = 1 + max(self._get_height(z.left), self._get_height(z.right))
        y.height = 1 + max(self._get_height(y.left), self._get_height(y.right))
        return y

    def _rotate_right(self, y):
        x = y.left
        T2 = x.right
        x.right = y
        y.left = T2
        y.height = 1 + max(self._get_height(y.left), self._get_height(y.right))
        x.height = 1 + max(self._get_height(x.left), self._get_height(x.right))
        return x

    def preorder(self):
        return self._preorder(self.root)

    def _preorder(self, node):
        if not node:
            return []
        return [node.value] + self._preorder(node.left) +
self._preorder(node.right)

n = int(input().strip())
sequence = list(map(int, input().strip().split()))

avl = AVL()
for value in sequence:
    avl.insert(value)

print(' '.join(map(str, avl.preorder())))
```

4.并查集

```
def find(x):
    if parent[x] == x:
        return x
    else:
```

```
        parent[x] = find(parent[x])
        return find(parent[x])

def union(x, y):
    parent[find(x)] = find(y)
```

5.图

通常的dijkstra算法

```
import heapq
import sys

class Vertex:
    def __init__(self, key):
        self.id = key
        self.connectedTo = {}
        self.distance = sys.maxsize
        self.pred = None

    def addNeighbor(self, nbr, weight=0):
        self.connectedTo[nbr] = weight

    def getConnections(self):
        return self.connectedTo.keys()

    def getWeight(self, nbr):
        return self.connectedTo[nbr]

    def __lt__(self, other):
        return self.distance < other.distance

class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

    def addVertex(self, key):
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        self.numVertices += 1
        return newVertex

    def getVertex(self, n):
        return self.vertList.get(n)

    def addEdge(self, f, t, cost=0):
        if f not in self.vertList:
            self.addVertex(f)
        if t not in self.vertList:
            self.addVertex(t)
```

```
        self.vertList[f].addNeighbor(self.vertList[t], cost)

def dijkstra(graph, start):
    pq = []
    start.distance = 0
    heapq.heappush(pq, (0, start))
    visited = set()

    while pq:
        currentDist, currentVert = heapq.heappop(pq)     # 当一个顶点的最短路径确定后
（也就是这个顶点
                                                         # 从优先队列中被弹出时），它
的最短路径不会再改变。
        if currentVert in visited:
            continue
        visited.add(currentVert)

        for nextVert in currentVert.getConnections():
            newDist = currentDist + currentVert.getWeight(nextVert)
            if newDist < nextVert.distance:
                nextVert.distance = newDist
                nextVert.pred = currentVert
                heapq.heappush(pq, (newDist, nextVert))

# 创建图和边
g = Graph()
g.addEdge('A', 'B', 4)
g.addEdge('A', 'C', 2)
g.addEdge('C', 'B', 1)
g.addEdge('B', 'D', 2)
g.addEdge('C', 'D', 5)
g.addEdge('D', 'E', 3)
g.addEdge('E', 'F', 1)
g.addEdge('D', 'F', 6)

# 执行 Dijkstra 算法
print("Shortest Path Tree:")
dijkstra(g, g.getVertex('A'))

# 输出最短路径树的顶点及其距离
for vertex in g.vertList.values():
    print(f"Vertex: {vertex.id}, Distance: {vertex.distance}")

# 输出最短路径到每个顶点
def printPath(vert):
    if vert.pred:
        printPath(vert.pred)
        print(" -> ", end="")
    print(vert.id, end="")

print("\nPaths from Start Vertex 'A':")
for vertex in g.vertList.values():
    print(f"Path to {vertex.id}: ", end="")
    printPath(vertex)
```

```
    print(", Distance: ", vertex.distance)

"""
Shortest Path Tree:
Vertex: A, Distance: 0
Vertex: B, Distance: 3
Vertex: C, Distance: 2
Vertex: D, Distance: 5
Vertex: E, Distance: 8
Vertex: F, Distance: 9

Paths from Start Vertex 'A':
Path to A: A, Distance:  0
Path to B: A -> C -> B, Distance:  3
Path to C: A -> C, Distance:  2
Path to D: A -> C -> B -> D, Distance:  5
Path to E: A -> C -> B -> D -> E, Distance:  8
Path to F: A -> C -> B -> D -> E -> F, Distance:  9
"""
```

有些题目，直接使用邻接表表示图，代码更简短

```
# 03424: Candies
# http://cs101.openjudge.cn/practice/03424/
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])  # 输出结果
```

或者：

```
import heapq

def dijkstra(N, G, start):
    pq = [(0, start, k)]  # 使用优先队列，存储节点的最短距离
    while pq:
        d, node, currentfee = heapq.heappop(pq)  # 弹出当前最短距离的节点
        if node == n:
            return d
        for neighbor, weight, fee in G[node]:  # 遍历当前节点的所有邻居节点
            new_dist = d + weight  # 计算经当前节点到达邻居节点的距离
            newFee = currentfee - fee
            if newFee >= 0:
                heapq.heappush(pq, (new_dist, neighbor, newFee))  # 将邻居节点加入
优先队列
    return -1

k = int(input())
n = int(input())
r = int(input())
G = [[] for _ in range(n + 1)]
for i in range(r):
    s, d, l, t = map(int,input().split())
    G[s].append((d, l, t))
print(dijkstra(n, G, 1))
```

通常的Prim实现

```
import sys
import heapq

class Vertex:
    def __init__(self, key):
        self.id = key
        self.connectedTo = {}
        self.distance = sys.maxsize
        self.pred = None

    def addNeighbor(self, nbr, weight=0):
        self.connectedTo[nbr] = weight

    def getConnections(self):
        return self.connectedTo.keys()

    def getWeight(self, nbr):
        return self.connectedTo[nbr]
```

```
    def __lt__(self, other):
        return self.distance < other.distance

class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

    def addVertex(self, key):
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        self.numVertices += 1
        return newVertex

    def getVertex(self, n):
        return self.vertList.get(n)

    def addEdge(self, f, t, cost=0):
        if f not in self.vertList:
            self.addVertex(f)
        if t not in self.vertList:
            self.addVertex(t)
        self.vertList[f].addNeighbor(self.vertList[t], cost)
        self.vertList[t].addNeighbor(self.vertList[f], cost)

def prim(graph, start):
    pq = []
    start.distance = 0
    heapq.heappush(pq, (0, start))
    visited = set()

    while pq:
        currentDist, currentVert = heapq.heappop(pq)
        if currentVert in visited:
            continue
        visited.add(currentVert)

        for nextVert in currentVert.getConnections():
            weight = currentVert.getWeight(nextVert)
            if nextVert not in visited and weight < nextVert.distance:
                nextVert.distance = weight
                nextVert.pred = currentVert
                heapq.heappush(pq, (weight, nextVert))

# 创建图和边
g = Graph()
g.addEdge('A', 'B', 4)
g.addEdge('A', 'C', 3)
g.addEdge('C', 'B', 1)
g.addEdge('C', 'D', 2)
g.addEdge('D', 'B', 5)
g.addEdge('D', 'E', 6)

# 执行 Prim 算法
```

```
print("Minimum Spanning Tree:")
prim(g, g.getVertex('A'))

# 输出最小生成树的边
for vertex in g.vertList.values():
    if vertex.pred:
        print(f"{vertex.pred.id} -> {vertex.id} Weight:{vertex.distance}")

"""
Minimum Spanning Tree:
C -> B Weight:1
A -> C Weight:3
C -> D Weight:2
D -> E Weight:6
"""
```

kruskal算法

```
class DisjointSet:
    def __init__(self, num_vertices):
        self.parent = list(range(num_vertices))
        self.rank = [0] * num_vertices

    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:
            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_x] = root_y
                self.rank[root_y] += 1


def kruskal(graph):
    num_vertices = len(graph)
    edges = []

    # 构建边集
    for i in range(num_vertices):
        for j in range(i + 1, num_vertices):
            if graph[i][j] != 0:
                edges.append((i, j, graph[i][j]))
```

```
    # 按照权重排序
    edges.sort(key=lambda x: x[2])

    # 初始化并查集
    disjoint_set = DisjointSet(num_vertices)

    # 构建最小生成树的边集
    minimum_spanning_tree = []

    for edge in edges:
        u, v, weight = edge
        if disjoint_set.find(u) != disjoint_set.find(v):
            disjoint_set.union(u, v)
            minimum_spanning_tree.append((u, v, weight))

    return minimum_spanning_tree
```

拓扑排序

Kahn算法的基本思想是通过不断地移除图中的入度为0的顶点，并将其添加到拓扑排序的结果中，直到图中所有的顶点都被移除。具体步骤如下：

初始化一个队列，用于存储当前入度为0的顶点。
遍历图中的所有顶点，计算每个顶点的入度，并将入度为0的顶点加入到队列中。
不断地从队列中弹出顶点，并将其加入到拓扑排序的结果中。同时，遍历该顶点的邻居，并将其入度减1。如果某个邻居的入度减为0，则将其加入到队列中。
重复步骤3，直到队列为空。

```
from collections import deque, defaultdict

def topological_sort(graph):
    indegree = defaultdict(int)
    result = []
    queue = deque()

    # 计算每个顶点的入度
    for u in graph:
        for v in graph[u]:
            indegree[v] += 1

    # 将入度为 0 的顶点加入队列
    for u in graph:
        if indegree[u] == 0:
            queue.append(u)

    # 执行拓扑排序
    while queue:
        u = queue.popleft()
        result.append(u)
```

```
            for v in graph[u]:
                indegree[v] -= 1
                if indegree[v] == 0:
                    queue.append(v)

        # 检查是否存在环
        if len(result) == len(graph):
            return result
        else:
            return None

    # 示例调用代码
    graph = {
        'A': ['B', 'C'],
        'B': ['C', 'D'],
        'C': ['E'],
        'D': ['F'],
        'E': ['F'],
        'F': []
    }

    sorted_vertices = topological_sort(graph)
    if sorted_vertices:
        print("Topological sort order:", sorted_vertices)
    else:
        print("The graph contains a cycle.")

    # Output:
    # Topological sort order: ['A', 'B', 'C', 'D', 'E', 'F']
```

6.dp

采药

```
t,m=map(int,input().split())
kusakusuri=[]
dp=[[0]*(t+1) for i in range(m)]
for i in range(m):
    kusakusuri.append(list(map(int,input().split())))
for j in range(t+1):
    if kusakusuri[0][0]<=j:
            dp[0][j]=kusakusuri[0][1]
for i in range(1,m):
    for j in range(1,t+1):
        if kusakusuri[i][0]>j:
            dp[i][j]=dp[i-1][j]
        else:
            dp[i][j]=max(dp[i-1][j],dp[i-1][j-kusakusuri[i][0]]+kusakusuri[i][1])
print(dp[m-1][t])
```

7.单调栈

模板：单调栈：定义函数 f(i) 代表数列中第 i 个元素之后第一个大于 ai 的元素的下标

```
n = int(input())
s = list(map(int, input().split()))
stack = []

f = [0]*n
for i in range(n):
    while stack and s[stack[-1]] < s[i]:
        f[stack.pop()] = i + 1
    stack.append(i)

print(*f)
```

奶牛排队：
奶牛在熊大妈的带领下排成了一条直队。显然，不同的奶牛身高不一定相同.....
现在，奶牛们想知道，如果找出一些连续的奶牛，要求最左边的奶牛 A 是最矮的，最右边的 B 是最高的，且 B 高于 A 奶牛。中间如果存在奶牛，则身高不能和 A,B 奶牛相同。问这样的奶牛最多会有多少头？
从左到右给出奶牛的身高，请告诉它们符合条件的最多的奶牛数（答案可能是 0,2，但不会是 1）。

```
n = int(input())
stack = []
small = [(n)]*n
big = [-1]*n
s = [0]*n
for i in range(n):
    s[i] = int(input())
for i in range(n):
    while stack and s[stack[-1]] >= s[i]:
        small[stack.pop()] = i
    stack.append(i)
stack = []
for i in range(n-1,-1,-1):
    while stack and s[stack[-1]] <= s[i]:
        big[stack.pop()] = i
    stack.append(i)
cow = 0
for i in range(n):
#    for j in range(i+1,small[i]):
#        if big[j] < i:
#            cow = max(cow, j - i + 1)  # 这样写答案是对的，但是需要优化否则超时
    for j in range(small[i]-1,i,-1):
        if big[j] < i:
            cow = max(cow, j - i + 1)
            break
print(cow)
```

8.最长上升子序列

dp：最长下降子序列

```
k=int(input())
s=list(map(int,input().split()))
dp=[1 for i in range(k)]
for i in range(k):
    for j in range(i):
        if s[i]<=s[j]:
            dp[i]=max(dp[j]+1,dp[i])
maxi=max(dp)
print(maxi)
```

跳高：也是最长下降子序列

```
from bisect import bisect_left
n = int(input())
k = list(map(int,input().split()))
k = reversed(k)
stack = []
for i in k:
    if stack:
        if i > stack[-1]:
            stack.append(i)
        else:
            pos = bisect_left(stack, i)
            stack[pos] = i
    else:
        stack.append(i)
print(len(stack))
```

9.字典树

电话号码 给你一些电话号码，请判断它们是否是一致的，即是否有某个电话是另一个电话的前缀。比如：

Emergency 911
Alice 97 625 999
Bob 91 12 54 26

在这个例子中，我们不可能拨通Bob的电话，因为Emergency的电话是它的前缀，当拨打Bob的电话时会先接通Emergency，所以这些电话号码不是一致的。
每个测试数据，如果是一致的输出“YES”，如果不是输出“NO”

```
import collections

class Node:
```

```
    def __init__(self):
        self.child = collections.defaultdict(Node)
        self.leaf = True
    def addchild(self, name):
        newnode = Node()
        self.child[name] = newnode
        node_list.append(newnode)

def convert(l, node):
    if not l:
        return
    if not(l[0] in node.child):
        node.addchild(l[0])
        node.leaf = False
    convert(l[1:], node.child[l[0]])
t = int(input())
for ii in range(t):
    n = int(input())
    root = Node()
    node_list = []
    count = 0
    for i in range(n):
        s = list(input())
        convert(s, root)
    for k in node_list:
        if k.leaf:
            count += 1
    if count == n:
        print('YES')
    else:
        print('NO')
```