Cheetsheet for the final exam

语法工具

埃氏筛法，得到质数表

```
def judge(number):
    nlist = list(range(1,number+1))
    nlist[0] = 0
    k = 2
    while k * k <= number:
        if nlist[k-1] != 0:
            for i in range(2*k,number+1,k):
                nlist[i-1] = 0
        k += 1
    result = []
    for num in nlist:
        if num != 0:
            result.append(num)
    return result
```

二分查找（bisect)

python中直接用bisect包,进行二分查找

```
import bisect ##导入bisect包

##bisect是一个排序模块，操作对象必须为排好序的列表。
##bisect操作并不改变列表中的元素，仅仅是确认插入元素的位置
##与之对应的insort
lst = [1,3,5,7,9]
s = int(input())
bisect.bisect_left(lst, x, [lo=0, hi=len(a)])   ##[]中表示插入位置的上界和下届
##改成right同理

## 测试序列 a2
>>> a2 = [1, 3, 3, 4, 7]  # 元素从小到大排列，有重复，不等距

# 限定查找范围：[lo=1, hi=3]
>>> bisect.bisect_left(a2, 0, 1, 3)  # 与 x=0 右侧最近的元素是 1，其位置 index=0，但下限 lo=1，故只能返回位置 index=1
1
3
##如果说 bisect.bisect_left() 是为了在序列 a 中 查找 元素 x 的插入点 (左侧)，那么 bisect.insort_left() 就是在找到插入点的基础上，真正地将元素 x 插入序列 a，从而改变序列 a 同时保持元素顺序。
>>> a12 = [5, 6, 7, 8, 9]
>>> bisect.insort_left(a12, 5.5)
>>> a12
```

```
[5, 5.5, 6, 7, 8, 9]
```

##

math

gcd包，计算最大公因式

```
from math import gcd
x = gcd(15,20,25)
print(x)
## 5
```

math.pow(m,n)　计算`m`的`n`次幂。

math.log(m,n)　计算以`n`为底的`m`的对数。

eval

eval() 是 python 中功能非常强大的一个函数
将字符串当成有效的表达式来求值，并返回计算结果
所谓表达式就是：eval 这个函数会把里面的字符串参数的引号去掉，把中间的内容当成Python的代码，eval 函数会执行这段代码并且返回执行结果
也可以这样来理解：eval() 函数就是实现 list、dict、tuple、与str 之间的转化
————————————————————

```
result = eval("1 + 1")
print(result)  # 2

result = eval("'+' * 5")
print(result)  # +++++

# 3. 将字符串转换成列表
a = "[1, 2, 3, 4]"
result = type(eval(a))
print(result)  # <class 'list'>

input_number = input("请输入一个加减乘除运算公式：")
print(eval(input_number))
## 1*2 +3
## 5
```

for _ in sorted(dic.keys()): ##将字典按keys排序后

float('inf') 表示正无穷

##注意break只退一层循环

print(*lst) ##把列表中元素顺序输出

int(str,n) 将字符串`str`转换为`n`进制的整数。

for key,value in dict.items()　　遍历字典的键值对。

for index,value in enumerate(list)　枚举列表，提供元素及其索引。

dic.setdefault(key,[]).append(value) 常用在字典中加入元素的方式（如果没有值就建空表，有值就直接添加）

dict.get(key,default)　从字典中获取键对应的值，如果键不存在，则返回默认值`default`。

list(zip(a,b))　将两个列表元素一一配对，生成元组的列表。

1. `str.lstrip() / str.rstrip()`: 移除字符串左侧/右侧的空白字符。
2. `str.find(sub)`: 返回子字符串`sub`在字符串中首次出现的索引，如果未找到，则返回-1。
3. `str.replace(old, new)`: 将字符串中的`old`子字符串替换为`new`。
4. `str.isalpha() / str.isdigit() / str.isalnum()`: 检查字符串是否全部由字母/数字/字母和数字组成。
5. .`str.title()`：每个单词首字母大写。

算法(主要是图)

一、最小生成树（MST）

将图转化为树

要求1.不能有环路存在（节点数为N，合法的边数为N-1）

　　2.权值和最小

1.krustal算法

Kruskal算法是一种用于解决最小生成树（Minimum Spanning Tree，简称MST）问题的贪心算法。给定一个连通的带权无向图，Kruskal算法可以找到一个包含所有顶点的最小生成树，即包含所有顶点且边权重之和最小的树。

以下是Kruskal算法的基本步骤：

1. 将图中的所有边按照权重从小到大进行排序。（队列）
2. 初始化一个空的边集，用于存储最小生成树的边。
3. 重复以下步骤，直到边集中的边数等于顶点数减一或者所有边都已经考虑完毕：（用并查集判断新加入的边是否合法）
    - 选择排序后的边集中权重最小的边。
    - 如果选择的边不会导致形成环路（即加入该边后，两个顶点不在同一个连通分量中），则将该边加入最小生成树的边集中。
4. 返回最小生成树的边集作为结果。

Kruskal算法的核心思想是通过不断选择权重最小的边，并判断是否会形成环路来构建最小生成树。算法开始时，每个顶点都是一个独立的连通分量，随着边的不断加入，不同的连通分量逐渐合并为一个连通分量，直到最终形成最小生成树。

实现Kruskal算法时，一种常用的数据结构是并查集（Disjoint Set）。并查集可以高效地判断两个顶点是否在同一个连通分量中，并将不同的连通分量合并。

```
##class DisjointSet:
''''''

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
```

2.Prim算法

可以从任何一个节点开始，找距离已选节点最近的那个点。然后将连接该边和该点的权值加入进去，作为最小生成树的一条边。

重复这样操作，直到所有节点都进入

更适用于稠密图

```
# 01258: Agri-Net
# http://cs101.openjudge.cn/practice/01258/
from heapq import heappop, heappush, heapify

def prim(graph, start_node):
    mst = set()
    visited = set([start_node])
    edges = [
        (cost, start_node, to)
```

```
        for to, cost in graph[start_node].items()
    ]
    heapify(edges)

    while edges:
        cost, frm, to = heappop(edges)
        if to not in visited:
            visited.add(to)
            mst.add((frm, to, cost))
            for to_next, cost2 in graph[to].items():
                if to_next not in visited:
                    heappush(edges, (cost2, to, to_next))

    return mst
```

二、最小路径

1.Dijkstra 算法（从某一点到其他所有点的最短路径）

Dijkstra算法：Dijkstra算法用于解决单源最短路径问题，即从给定源节点到图中所有其他节点的最短路径。算法的基本思想是通过不断扩展离源节点最近的节点来逐步确定最短路径。具体步骤如下：

- 初始化一个距离数组，用于记录源节点到所有其他节点的最短距离。初始时，==源节点的距离为0，其他节点的距离为无穷大==。
- 选择一个未访问的节点中距离最小的节点作为当前节点。
- 更新当前节点的邻居节点的距离，如果通过当前节点到达邻居节点的路径比已知最短路径更短，则==更新最短路径==。
- 标记当前节点为==已访问==。
- 重复上述步骤，直到所有节点都被访问或者所有节点的最短路径都被确定。

Dijkstra算法的时间复杂度为O(V^2)，其中V是图中的节点数。当使用优先队列（如最小堆）来选择距离最小的节点时，可以将时间复杂度优化到O((V+E)logV)，其中E是图中的边数。

Dijkstra.py 程序在 https://github.com/GMyhf/2024spring-cs201/tree/main/code

```
# 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
```

2. Bellman-Ford算法

Bellman-Ford算法：Bellman-Ford算法用于解决单源最短路径问题，与Dijkstra算法不同，它可以处理带有负权边的图。算法的基本思想是通过松弛操作逐步更新节点的最短路径估计值，直到收敛到最终结果。具体步骤如下：

- 初始化一个距离数组，用于记录源节点到所有其他节点的最短距离。初始时，源节点的距离为0，其他节点的距离为无穷大。
- 进行V-1次循环（V是图中的节点数），每次循环对所有边进行松弛操作。如果从节点u到节点v的路径经过节点u的距离加上边(u, v)的权重比当前已知的从源节点到节点v的最短路径更短，则更新最短路径。
- 检查是否存在负权回路。如果在V-1次循环后，仍然可以通过松弛操作更新最短路径，则说明存在负权回路，因此无法确定最短路径。

Bellman-Ford算法的时间复杂度为O(V*E)，其中V是图中的节点数，E是图中的边数。

```
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []

    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])

    def bellman_ford(self, src):
        # 初始化距离数组，表示从源点到各个顶点的最短距离
        dist = [float('inf')] * self.V
        dist[src] = 0
```

```
        # 迭代 V-1 次，每次更新所有边
        for _ in range(self.V - 1):
            for u, v, w in self.graph:
                if dist[u] != float('inf') and dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w

        # 检测负权环
        for u, v, w in self.graph:
            if dist[u] != float('inf') and dist[u] + w < dist[v]:
                return "Graph contains negative weight cycle"

        return dist

# 测试代码
g = Graph(5)
g.add_edge(0, 1, -1)
g.add_edge(0, 2, 4)
g.add_edge(1, 2, 3)
g.add_edge(1, 3, 2)
g.add_edge(1, 4, 2)
g.add_edge(3, 2, 5)
g.add_edge(3, 1, 1)
g.add_edge(4, 3, -3)

src = 0
distances = g.bellman_ford(src)
print("最短路径距离：")
for i in range(len(distances)):
    print(f"从源点 {src} 到顶点 {i} 的最短距离为：{distances[i]}")
```

3 多源最短路径Floyd-Warshall算法

求解所有顶点之间的最短路径可以使用**Floyd-Warshall算法**，它是一种多源最短路径算法。Floyd-Warshall算法可以**在有向图或无向图中找到任意两个顶点之间的最短路径。**

算法的基本思想是通过一个二维数组来存储任意两个顶点之间的最短距离。初始时，这个数组包含图中各个顶点之间的直接边的权重，对于不直接相连的顶点，权重为无穷大。然后，通过迭代更新这个数组，逐步求得所有顶点之间的最短路径。

具体步骤如下：

1. 初始化一个二维数组`dist`，用于存储任意两个顶点之间的最短距离。初始时，`dist[i][j]`表示顶点i到顶点j的直接边的权重，如果i和j不直接相连，则权重为无穷大。
2. 对于每个顶点k，在更新`dist`数组时，考虑顶点k作为中间节点的情况。遍历所有的顶点对(i, j)，如果通过顶点k可以使得从顶点i到顶点j的路径变短，则更新`dist[i][j]`为更小的值。

   `dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])`
3. 重复进行上述步骤，对于每个顶点作为中间节点，进行迭代更新`dist`数组。最终，`dist`数组中存储的就是所有顶点之间的最短路径。

Floyd-Warshall算法的时间复杂度为O(V^3)，其中V是图中的顶点数。它适用于解决稠密图（边数较多）的最短路径问题，并且可以处理负权边和负权回路。

以下是一个使用Floyd-Warshall算法求解所有顶点之间最短路径的示例代码：

```
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
```

在上述代码中，`graph`是一个字典，用于表示图的邻接关系。它的键表示起始顶点，值表示一个字典，其中键表示终点顶点，值表示对应边的权重。

你可以将你的图表示为一个邻接矩阵或邻接表，并将其作为参数传递给`floyd_warshall`函数。函数将返回一个二维数组，其中`dist[i][j]`表示从顶点i到顶点j的最短路径长度。

三、拓扑排序

== 每次找入度为0的点

1.1、无向图
使用拓扑排序可以判断一个无向图中是否存在环，具体步骤如下：

求出图中所有结点的度。
将所有度 <= 1 的结点入队。（独立结点的度为 0）
当队列不空时，弹出队首元素，把与队首元素相邻节点的度减一。如果相邻节点的度变为一，则将相邻结点入队。
循环结束时判断已经访问的结点数是否等于 n。等于 n 说明全部结点都被访问过，无环；反之，则有环。
1.2、有向图
使用拓扑排序判断无向图和有向图中是否存在环的区别在于：

在判断无向图中是否存在环时，是将所有度 <= 1 的结点入队；
在判断有向图中是否存在环时，是将所有入度 = 0 的结点入队。

```
from collections import defaultdict

def topological_sort(graph):
    indegree = defaultdict(int)
```

```
    result = []
    queue = []
    # 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 range(1,node+1): ##这里node表示节点总数
        if indegree[u] == 0:
            queue.append(u)
    # 执行拓扑排序
    while queue:
        u = queue.pop(0)
        result.append(u)
        if u in graph: ##如果不在graph里，说明到了一个终点
            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) == node: ##有环时None，表示无法排序
        return True
    else:
        return False
```

在上述代码中，`graph` 是一个字典，用于表示有向图的邻接关系。它的键表示顶点，值表示一个列表，表示从该顶点出发的边所连接的顶点。

你可以将你的有向图表示为一个邻接矩阵或邻接表，并将其作为参数传递给 `topological_sort` 函数。如果存在拓扑排序，函数将返回一个列表，按照拓扑排序的顺序包含所有顶点。如果图中存在环，函数将返回 `None`，表示无法进行拓扑排序。

四：其他问题

1.最小路径

关键在于队列，保证不同方向的进度是相同的，这样的话能保证找到的第一个结果一定是最好的结果

```
from collections import deque

def bfs(m, n, grid):
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    visited = [[False] * n for _ in range(m)]
    start = (0, 0)
    queue = deque([(start, 0)])  # 起点和步数入队
    visited[start[0]][start[1]] = True

    while queue:
        current, steps = queue.popleft()
        x, y = current
```

```
        if grid[x][y] == 1:  # 到达藏宝点
            return steps

        for dx, dy in directions:
            nx, ny = x + dx, y + dy

            if 0 <= nx < m and 0 <= ny < n and grid[nx][ny] != 2 and not visited[nx][ny]:
                visited[nx][ny] = True
                queue.append(((nx, ny), steps + 1))

    return -1  # 无法到达藏宝点

m, n = map(int, input().split())
grid = []

for _ in range(m):
    row = list(map(int, input().split()))
    grid.append(row)

result = bfs(m, n, grid)

if result == -1:
    print("NO")
else:
    print(result)
```

http://cs101.openjudge.cn/2024sp_routine/04001/

抓住那头牛：

注意：一定要在队列加入前就将该元素加入到visited表里（这里用的是data），否则会超时

```
start,target = map(int,input().split())
M = abs(target - start)
data = set()
data.add(start)
queue = [(start,0)]
while queue:
    pos,count = queue.pop(0)
    if pos == target:
        print(count)
        exit()

     ##data.add(i) 如果在这里才加入到visited中，就会超时

    for i in [pos - 1,pos + 1,pos * 2]:
        if i in data or count + 1 > M or i < 0 or i > 100000:
            continue
        queue.append((i,count + 1))
        data.add(i)
```

2.并查集

并查集是一种用于维护集合（组）的数据结构，它通常用于解决一些离线查询、动态连通性和图论等相关问题。

其中最常见的应用场景是解决图论中的连通性问题，例如判断图中两个节点是否连通、查找图的连通分量、判断图是否为一棵树等等。并查集可以快速地合并两个节点所在的集合，以及查询两个节点是否属于同一个集合，从而有效地判断图的连通性。

并查集还可以用于解决一些离线查询问题，例如静态连通性查询和==最小生成树问题==，以及一些==动态连通性问题==，例如支持动态加边和删边的连通性问题。

```
class disj_set:
    def __init__(self,n):
        self.rank = [1 for i in range(n)]
        self.parent = [i for i in range(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):
        x_root = self.find(x)
        y_root = self.find(y)

        if x_root == y_root:
            return

        if self.rank[x_root] > self.rank[y_root]:
            self.parent[y_root] = x_root
        elif self.rank[y_root] < self.rank[x_root]:
            self.parent[x_root] = y_root
        else:
            self.parent[y_root] = x_root
            self.rank[x_root] += 1


##计算父亲节点个数
count = 0
for x in range(1,n+1):
    if D.parent[x-1] == x - 1:
        count += 1
```

3.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()))
```

4.dp

最大上升子序列

```
input()
b = [int(x) for x in input().split()]

n = len(b)
dp = [0]*n

for i in range(n):
    dp[i] = b[i]
    for j in range(i):
        if b[j]<b[i]:
            dp[i] = max(dp[j]+b[i], dp[i])

print(max(dp))
```

5.递归

必须有一个明确的结束条件。
每次进入更深一层递归时，问题规模（计算量）相比上次递归都应有所减少。
递归效率不高，递归层次过多会导致栈溢出（在计算机中，函数调用是通过栈（stack）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出

== 每个函数的操作，分为平行的操作和深度的操作。

def solve_n_queens(n):
  solutions = [] # 存储所有解决方案的列表
  queens = [-1] * n # 存储每一行皇后所在的列数

回溯递归（八皇后问题）

```
def backtrack(row):
    if row == n:  # 找到一个合法解决方案
        solutions.append(queens.copy())
    else:
        for col in range(n):
            if is_valid(row, col):  # 检查当前位置是否合法
                queens[row] = col  # 在当前行放置皇后
                backtrack(row + 1)  # 递归处理下一行
                queens[row] = -1  # 回溯，撤销当前行的选择

def is_valid(row, col):
    for r in range(row):
        if queens[r] == col or abs(row - r) == abs(col - queens[r]):
            return False
    return True

  backtrack(0)  # 从第一行开始回溯

  return solutions
def get_queen_string(b):
    solutions = solve_n_queens(8)
    if b > len(solutions):
        return None
    queen_string = ''.join(str(col + 1) for col in solutions[b - 1])
    return queen_string

test_cases = int(input())  # 输入的测试数据组数
for _ in range(test_cases):
    b = int(input())  # 输入的 b 值
    queen_string = get_queen_string(b)
    print(queen_string)
```

五、树

层级 Level：
从根节点开始到达一个节点的路径，所包含的==边的数量==，称为这个节点的层级。
如图 D 的层级为 2，根节点的层级为 0。

有时候，题目中会给出概念定义，如：

高度 Height：树中所有节点的==最大层级==称为树的高度。

二叉树深度：从根结点到叶结点依次经过的结点（含根、叶结点）形成树的一条路径，==最长路径的节点个数==为树的深度

2.按形态分类（主要是二叉树）

（1）完全二叉树——第n-1层全满，最后一层按顺序排列

（2）满二叉树——二叉树的最下面一层元素全部满就是满二叉树

（3）avl树——平衡因子，左右子树高度差不超过1

= =这块一定要弄懂左右旋的概念，理解什么时候左旋什么时候右旋，以及操作的具体过程！

（4）二叉查找树(二叉排序\搜索树)

= =特点：没有相同键值的节点。

= =若左子树不空，那么其所有子孙都比根节点小。

= =若右子树不空，那么其所有子孙都比根节点大。

= =左右子树也分别为二叉排序树。

（5）哈夫曼树——哈夫曼树是一种针对权值的二叉树。一般为了减少计算机运算速度，将权重大的放在最前面

```

```

树的遍历

前序遍历
在前序遍历中，先访问根节点，然后递归地前序遍历左子树，最后递归地前序遍历右子树。

中序遍历
在中序遍历中，先递归地中序遍历左子树，然后访问根节点，最后递归地中序遍历右子树。

后序遍历
在后序遍历中，先递归地后序遍历左子树，然后递归地后序遍历右子树，最后访问根节点。

```
def preorder(root): ##前序遍历
    if root is None:
        return []
    result = [root.value]
    result.extend(preorder(root.left))
    result.extend(preorder(root.right))
    return result

##改变result的位置便可以把前序变成中序，变成后序

def behindorder(root): ##后序遍历
    if root is None:
        return []
    result = []
    result.extend(behindorder(root.left))
    result.extend(behindorder(root.right))
    result.append(root.value)
    return result
```

哈弗曼编码

这段代码首先定义了一个 `Node` 类来表示哈夫曼树的节点。然后，使用最小堆来构建哈夫曼树，每次==从堆中取出两个频率最小的节点==进行合并，直到堆中只剩下一个节点，即哈夫曼树的根节点。接着，使用递归方法计算哈夫曼树的带权外部路径长度（weighted external path length）。最后，输出计算得到的带权外部路径长度。

哈夫曼编码树

http://cs101.openjudge.cn/practice/22161/

```
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) #note: 合并后，char 字段默认值是空
        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.char:
            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.char:
            decoded += node.char
            node = root
    return decoded

# 读取输入
n = int(input())
characters = {}
for _ in range(n):
    char, weight = input().split()
    characters[char] = int(weight)

#string = input().strip()
#encoded_string = input().strip()

# 构建哈夫曼编码树
huffman_tree = build_huffman_tree(characters)

# 编码和解码
codes = encode_huffman_tree(huffman_tree)

strings = []
while True:
    try:
        line = input()
        if line:
            strings.append(line)
        else:
            break
    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)
```

二叉堆实现（最小堆）

1.每次插入元素从==最后插入==，然后进行调整堆的操作

2.每次删除元素从堆顶找元素，找到元素后将该元素和最后一个元素换位，然后进行==重排操作==

（这里要求的是删除最小元素，即堆顶元素）

http://cs101.openjudge.cn/practice/04078/

```
class tree_node:
    def __init__(self):
        self.heap = []

    def parent(self, i):
        return (i - 1) // 2

    def left_child(self, i):
        return 2 * i + 1

    def right_child(self, i):
        return 2 * i + 2

    def swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

    def insert(self, item):
        self.heap.append(item)
        self.heapify_up(len(self.heap) - 1)

    def delete(self):
        if len(self.heap) == 0:
            raise IndexError("Heap is empty")

        self.swap(0, len(self.heap) - 1)
        min_value = self.heap.pop()
        self.heapify_down(0)
        return min_value

    def heapify_up(self, i):
        while i > 0 and self.heap[i] < self.heap[self.parent(i)]:
            self.swap(i, self.parent(i))
            i = self.parent(i)

    def heapify_down(self, i):
```

```
        min_index = i
        left = self.left_child(i)
        right = self.right_child(i)

        if left < len(self.heap) and self.heap[left] < self.heap[min_index]:
            min_index = left

        if right < len(self.heap) and self.heap[right] < self.heap[min_index]:
            min_index = right

        if i != min_index:
            self.swap(i, min_index)
            self.heapify_down(min_index)
n = int(input())
lst = tree_node()
for _ in range(n):
    s = input()
    if s[0] == '1':

        lst.insert(int(s[2:]))
    if s[0] == '2':
        print(lst.delete())
```

二叉搜索树的层次遍历

http://cs101.openjudge.cn/practice/05455/

```
class tree_node:
    def __init__(self,value):
        self.value = value
        self.left = None
        self.right = None
def insert_tree(node,value):
     if node is None:
         return tree_node(value)
     if node.value > value:  ##如果比根节点小
         node.left = insert_tree(node.left,value) ##和根节点的左节点继续比较
     elif node.value < value:
         node.right = insert_tree(node.right,value)
     return node
def recursion(node):
    deque = [node]
    result = []
    while deque:
        x = deque.pop(0)
        if x.left is not None:
            deque.append(x.left)
        if x.right is not None:
            deque.append(x.right)
        result.append(x.value)
    return result
```

```
lst = list(map(int,input().split()))
node = None
data = set()
for i in lst:
    if i not in data:
        data.add(i)
        node = insert_tree(node,i)
print(*recursion(node))
```

一些技巧&算法

堆

堆是一种特殊的树形数据结构，其中每个节点的值都小于或等于（最小堆）或大于或等于（最大堆）其子节点的值。堆分为最小堆和最大堆两种类型，其中：

- 最小堆： 父节点的值小于或等于其子节点的值。
- 最大堆： 父节点的值大于或等于其子节点的值。
  堆常用于实现优先队列和堆排序等算法。

== 看到一直要用min，max的基本都要用堆

```
import heapq
x = [1,2,3,5,7]

heapq.heapify(x)
###将列表转换为堆。

heapq.heappushpop(heap, item)
##将 item 放入堆中，然后弹出并返回 heap 的最小元素。该组合操作比先调用 heappush() 再调用 heappop() 运行起来更有效率

heapq.heapreplace(heap, item)
##弹出并返回最小的元素，并且添加一个新元素item

heapq.heappop(heap,item)
heapq.heappush(heap,item)
```

懒删除

懒删除就是，我表面上删除一个元素，实际上没有从堆里拿出来。

而是当我访问堆，要pop堆顶的时候，检查一下这个元素被没被删过

http://cs101.openjudge.cn/2024sp_routine/22067/

```
import heapq
stack = []
Min = []
min_pop = set()
while True:
    try:
        s = input()
        if s == 'pop':
            if stack:
                min_pop.add(stack.pop())
        elif s == 'min':
            while Min and Min[0] in min_pop:
                min_pop.remove(heapq.heappop(Min))
            if Min:
                print(Min[0])
        elif 'push' in s:
            stack.append(int(list(s.split())[-1]))
            heapq.heappush(Min,stack[-1])
    except EOFError:
        break
```

求中位数

构建==大根堆和小根堆==（这里大根堆用相反数构建，保证输入的数据都是恒正的）

因为只需要求中位数，所以只要注意一下两个堆元素之差不能大于1

```
import heapq

n = int(input())

def insert(num):
    if len(Min) == 0 or num > Min[0]:
        heapq.heappush(Min, num)
    else:
        heapq.heappush(Max,-num)
    if len(Min) - len(Max) > 1:
        heapq.heappush(Max,-heapq.heappop(Min))
    elif len(Max) - len(Min) > 1:
        heapq.heappush(Min, -heapq.heappop(Max))
for _ in range(n):
    result = []
    count = 0
    Min = []
    Max = []
    lst = list(map(int,input().split()))
    for num in range(len(lst)):
        count += 1
        insert(lst[num])
        if count % 2 == 1:
            if len(Min) > len(Max):
```

```
                result.append(Min[0])
            else:
                result.append(-Max[0])
    print(len(result))
    print(*result)
```

栈

单调栈

= =单调栈，可以找到第一个比当前节点高的节点的位置

模版：

```
n = int(input())
lst = list(int(i) for i in input().split())
pos = [0 for i in range(n)] ##初始化位置
judge = lst[0]
stack = []
for i in range(n-1,-1,-1):
    while stack and lst[stack[-1]] <= lst[i]:
        stack.pop()
    if stack:
        pos[i] = stack[-1] + 1
    stack.append(i)
print(*pos)
```

中序表达式转后序表达式

```
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))
```

Dilworth定理:

Dilworth定理表明

任何一个有限偏序集的最长反链(即最长下降子序列)的长度等于将该偏序集划分为尽量少的链(即上升子序列)的最小数量。

跳高：http://cs101.openjudge.cn/2024sp_routine/28389/

因此，计算序列的最长下降子序列长度，即可得出最少需要多少台测试仪。