数算复习

1.图

1.1 图的表示方法

1.1.1 邻接矩阵

每一行和每一列都表示图中的一个顶点。第v行和第w列交叉的格子中的值表示从顶点v到顶点w的边的权重。如果两个顶点被一条边连接起来，就称它们是相邻的。

http://cs101.openjudge.cn/practice/27928/遍历树

http://cs101.openjudge.cn/practice/04089/前缀树

```
n, m = map(int, input().split())
adjacency_matrix = [[0]*n for _ in range(n)]
for _ in range(m):
    u, v = map(int, input().split())
    adjacency_matrix[u][v] = 1
    adjacency_matrix[v][u] = 1

for row in adjacency_matrix:
    print(' '.join(map(str, row)))
```

1.1.2 邻接表

在邻接表实现中，我们为图对象的所有顶点保存一个主列表，同时为每一个顶点对象都维护一个列表，其中记录了与它相连的顶点。在对Vertex类的实现中，我们使用字典（而不是列表），字典的键是顶点，值是权重

```
n, m = map(int, input().split())
adjacency_list = [[] for _ in range(n)]
for _ in range(m):
    u, v = map(int, input().split())
    adjacency_list[u].append(v)
    adjacency_list[v].append(u)

for i in range(n):
    num = len(adjacency_list[i])
    if num == 0:
        print(f"{i}({num})")
    else:
        print(f"{i}({num})", ' '.join(map(str, adjacency_list[i])))
```

1.1.3 图的实现

在Python中，通过字典可以轻松地实现邻接表。要创建两个类：Graph类存储包含所有顶点的主列表，Vertex类表示图中的每一个顶点。
Vertex使用字典neighbors来记录与其相连的顶点，以及每一条边的权重。其构造方法简单地初始化key（它通常是一个字符串），以及字典neighbors。set_edge方法添加从一个顶点到另一个的连接。

get_neighbors方法返回邻接表中的所有顶点。get_neighbor方法返回从当前顶点到以参数传入的顶点之间的边的权重。

Graph类的实现，其中包含一个将顶点名映射到顶点对象的字典。Graph类也提供了向图中添加顶点和连接不同顶点的方法。get_vertices方法返回图中所有顶点的名字。此外，还实现了`__iter__`方法，使遍历图中的所有顶点对象更加方便。总之，这两个方法使我们能够根据顶点名或者顶点对象本身遍历图中的所有顶点。

```
class Vartex:
    def __init__(self,key):
        self.key=key
        self.neighbors={}
class Graph:
    def __init__(self):
        self.vertices={}
    def set_vertex(self,key):
        self.vertices[key]=Vartex(key)
    def get_vertex(self,key):
        return self.vertices.get(key,None)
    def __contains__(self,key):
        return key in self.vertices
    def add_edge(self,fv,tv,weight=0):
        if fv not in self:
            self.set_vertex(fv)
        if tv not in self:
            self.set_vertex(tv)
        self.get_vertex(fv).neighbors[self.get_vertex(tv)]=weight
    def get_vertices(self):
        return self.vertices.keys()
    def __iter__(self):
        return iter(self.vertices.values())
```

1.2 图算法

1.2.1基本图算法

1.2.1.1 宽度优先搜索(bfs)(与计概相同)

http://cs101.openjudge.cn/practice/28046/ 词梯问题

```
from collections import defaultdict,deque
n=int(input())
words=[input() for i in range(n)]
graph=defaultdict(list)
tongs=defaultdict(set)
for word in words:
    for i in range(4):
        tong=f"{word[:i]}_{word[i+1:]}"
        tongs[tong].add(word)
for w in tongs.values():
    for word1 in w:
        for word2 in w-{word1}:
            graph[word1].append(word2)
queue=deque()
visit=set()
```

```
ws,we=input().split()
queue.append((ws,[ws]))
visit.add(ws)
while queue:
    word,path=queue.popleft()
    if word==we:
        print(' '.join(path))
        break
    for neibour in graph[word]:
        if neibour not in visit:
            visit.add(neibour)
            queue.append((neibour,path+[neibour]))
else:
    print('NO')
```

时间复杂度：BFS算法的时间复杂度取决于图的顶点数和边数。在最坏情况下，每个节点和边都会被访问一次，因此时间复杂度为O(V + E)，其中V是顶点数，E是边数。

空间复杂度：在BFS算法中，使用了一个队列来存储待访问的节点，以及一个集合来存储已经访问过的节点。因此，空间复杂度取决于队列的大小和集合的大小。在最坏情况下，队列的大小可以达到图的顶点数，集合的大小也可以达到图的顶点数。因此，空间复杂度为O(V)，其中V是顶点数。

1.2.1.2 深度优先搜索(dfs)(与计概相同)

递归实现的DFS算法的时间复杂度为O(V + E)，空间复杂度为O(V)

http://cs101.openjudge.cn/practice/28050/ 骑士周游问题

```
from collections import defaultdict
d=[(2,-1),(2,1),(1,-2),(1,2),(-2,-1),(-2,1),(-1,-2),(-1,2)]
graph=defaultdict(list)
def trans(row,col):
    return row*n+col
def fneibour(row,col):
    neibour=[]
    for dx,dy in d:
        if 0<=row+dx<n and 0<=col+dy<n:
            neibour.append([row+dx,col+dy])
    return neibour
def cou(dot):
    c=0
    for nei in graph[dot]:
        if visit[nei]:
            c+=1
    return c
def sneibour(dot):
    neibour=[nei for nei in graph[dot] if visit[nei]]
    neibour.sort(key=lambda x:cou(x))
    return neibour
def dfs(count,dot):
    if count==n**2-1:
        return True
    visit[dot]=False
    neibour = sneibour(dot)
    for nei in neibour:
        if dfs(count+1,nei):
```

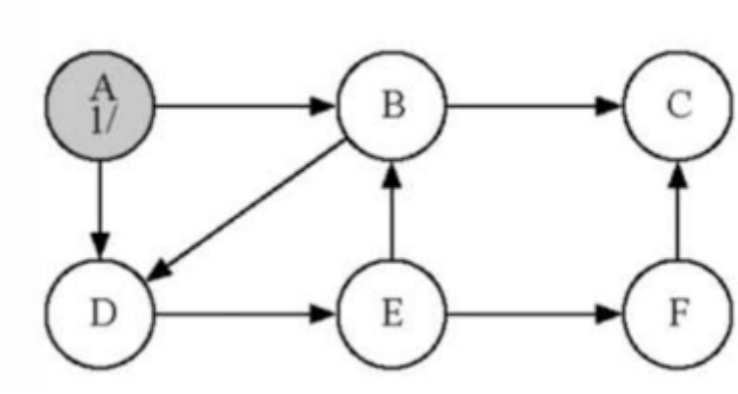
```
            return True
    else:
        visit[dot]=True
        return False
n=int(input())
sr,sc=map(int,input().split())
visit=[True]*(n**2)
for row in range(n):
    for col in range(n):
        for nr,nc in fneibour(row, col):
            graph[trans(row,col)].append(trans(nr,nc))

print(['fail','success'][dfs(0,trans(sr,sc))])
```

1.2.1.3 深度优先搜索森林

深度优先森林:创建多棵深度优先搜索树并给出起始时间和终止时间。

例：给一个有向图求出每个点的起始时间和终止时间



例如：

```
输入：
6 8
0 1
0 3
1 2
1 3
3 4
4 1
4 5
5 2
输出：
0 1 12
1 2 11
2 3 4
3 5 10
4 6 9
5 7 8
```

```
from collections import defaultdict
graph=defaultdict(list)
```

```
n,m=map(int,input().split())
for i in range(m):
    vs,ve=map(int,input().split())
    graph[vs].append(ve)
start=[-1]*n
end=[-1]*n
time=0
def dfs(vert):
    global time
    time=time+1
    start[vert]=time
    for nbr in graph[vert]:
        if start[nbr]==-1:
            dfs(nbr)
    time=time+1
    end[vert]=time
for i in range(n):
    if start[i]==-1:
        dfs(i)
for i in range(n):
    print(i,start[i],end[i])
```

可以用于接下来的拓展图算法之中

1.2.2 拓展图算法

1.2.2.1 拓扑排序

拓扑排序根据有向无环图生成一个包含所有顶点的线性序列，使得如果图G中有一条边为(v, w)，那么顶点v排在顶点w之前。在很多应用中，有向无环图被用于表明事件优先级。制作松饼只是其中一个例子，其他例子还包括软件项目调度、优化数据库查询的优先级表，以及矩阵相乘。

拓扑排序是对深度优先搜索的一种简单而强大的改进，其算法如下：

1.DFS：

(1) 对图`g`调用`dfs(g)`。之所以调用深度优先搜索函数，是因为要计算每一个顶点的结束时间。
(2) 基于结束时间，将顶点按照递减顺序存储在列表中。
(3) 将有序列表作为拓扑排序的结果返回。

就是深度优先搜索森林对结束时间排序后输出

缺点：无法识别有环图

2.Karn算法 / BFS：

Kahn算法是基于广度优先搜索（BFS）的一种拓扑排序算法。

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：

1. 初始化一个队列，用于存储当前入度为0的顶点。
2. 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。
3. 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。
4. 重复步骤3，直到队列为空。

Kahn算法的时间复杂度为O(V + E)，其中V是顶点数，E是边数。它是一种简单而高效的拓扑排序算法，在有向无环图（DAG）中广泛应用。

如果 `result` 列表的长度等于图中顶点的数量，则拓扑排序成功，返回结果列表 `result`；否则，图中存在环，无法进行拓扑排序。

https://sunnywhy.com/sfbj/10/3/382 有向图判环

示例程序：

```
from collections import defaultdict,deque
n,m=map(int,input().split())
graph=defaultdict(list)
indegree=[0]*n
for i in range(m):
    vs,ve=map(int,input().split())
    graph[vs].append(ve)
    indegree[ve]+=1
l=[i for i in range(n) if indegree[i]==0]
quene=deque(l)
ans=[]
while quene:
    vert=quene.popleft()
    ans.append(quene)
    for nbr in graph[vert]:
        indegree[nbr]-=1
        if indegree[nbr]==0:
            quene.append(nbr)
print(['Yes','No'][len(ans)==n])
```

1.2.2.2 强连通单元（SCCs）

通过一种叫作强连通单元的图算法，可以找出图中高度连通的顶点簇。对于图G，强连通单元C为最大的顶点子集$C \subset V$，其中对于每一对顶点$v, w \in C$，都有一条从v到w的路径和一条从w到v的路径。

Kosaraju / 2 DFS：

利用深度优先搜索，我们可以再次创建强大高效的算法。在学习强连通单元算法之前，还要再看一个定义。图G的转置图被定义为G^T，其中所有的边都与图G的边反向。这意味着，如果在图G中有一条由A到B的边，那么在G^T中就会有一条由B到A的边

Kosaraju算法的核心思想就是两次深度优先搜索（DFS）。

1. **第一次DFS**：在第一次DFS中，我们对图进行标准的深度优先搜索，但是在此过程中，我们记录下顶点完成搜索的顺序。这一步的目的是为了找出每个顶点的完成时间（即结束时间）。
2. **反向图**：接下来，我们对原图取反，即将所有的边方向反转，得到反向图。
3. **第二次DFS**：在第二次DFS中，我们按照第一步中记录的顶点完成时间的逆序，对反向图进行DFS。这样，我们将找出反向图中的强连通分量。

Kosaraju算法的关键在于第二次DFS的顺序，它保证了在DFS的过程中，我们能够优先访问到整个图中的强连通分量。因此，Kosaraju算法的时间复杂度为O(V + E)，其中V是顶点数，E是边数。

以下是Kosaraju算法的Python实现：

```
from collections import defaultdict
def dfs1(vert):
    visit[vert]=True
    for nbr in graph[vert]:
        if not visit[nbr]:
            dfs1(nbr)
    stack.append(vert)
def Reverse(graph):
    graph2 = defaultdict(list)
    for vert in graph.keys():
        for nbr in graph[vert]:
            graph2[nbr].append(vert)
    return graph2
def dfs2(vert):
    visit[vert] = True
    for nbr in graph2[vert]:
        if not visit[nbr]:
            dfs2(nbr)
    scc.append(vert)
graph=defaultdict(list)
n,m=map(int,input().split())
for i in range(m):
    vs,ve=map(int,input().split())
    graph[vs].append(ve)
stack=[]
visit=[False]*n
for vert in range(n):
    if not visit[vert]:
        dfs1(vert)
graph2=Reverse(graph)
visit=[False]*n
sccs=[]
while stack:
    vert=stack.pop()
    if not visit[vert]:
        scc=[]
        dfs2(vert)
        sccs.append(scc)
print(*sccs,sep='\n')
```

1.2.2.3 最短路径

1.dijkstra算法(计概学过)

非常重要的一点是，Dijkstra算法只适用于边的权重均为正的情况。如果图2中有一条边的权重为负，那么Dijkstra算法永远不会退出。

总的时间复杂度为$O((V+E)\log V)$

示例 https://leetcode.cn/problems/network-delay-time/description/网络延迟时间

```
from collections import defaultdict
import heapq
graph=defaultdict(dict)
for vs,ve,w in times:
    graph[vs][ve]=w
```

```
h=[(0,k)]
ltime=[0]+[20000]*n
ltime[k]=0
heapq.heapify(h)
while h:
    time,vert=heapq.heappop(h)
    if ltime[vert]<time:
        continue
    for nbr in graph[vert].keys():
        nt=time+graph[vert][nbr]
        if nt<ltime[nbr]:
            ltime[nbr]=nt
            heapq.heappush(h,(nt,nbr))
ans=max(ltime)
return ans if ans<20000 else -1
```

2.*bellman-ford 算法

思想：动态规划 + 松弛思想

每次迭代尝试通过已知的最短路径更新其他路径（松弛）

最多只需进行 V-1 次迭代，因为最短路径最多经过 V-1 个顶点

第 V 次检测是否还能更新，用于发现负权环

代码实现：

```
n,m,k=map(int,input().split())
edge=[list(map(int,input().split())) for i in range(m)]
dist=[float('inf')]*n
dist[k]=0
for _ in range(n-1):
    for [vs,ve,w] in edge:
        if dist[vs]!=float('inf') and dist[vs]+w<dist[ve]:
            dist[ve]=dist[vs]+w
for vs,ve,w in edge:
    if dist[vs]!=float('inf') and dist[vs]+w<dist[ve]:
        print('有负权环')
        break
else:
    print(dist)
```

Bellman-Ford算法的时间复杂度为$O(VE)$，其中V是图中的节点数，E是图中的边数，适合边稀疏图（边数远小于V^2）。

示例 https://leetcode.cn/problems/cheapest-flights-within-k-stops/ K站中转内最便宜的航班

```
dist=[float('inf') for i in range(n)]
dist[src]=0
ans=float('inf')
for _ in range(k+1):
    dist1=[float('inf')]*n
    for vs,ve,w in flights:
        dist1[ve]=min(dist1[ve],dist[vs]+w)
    dist=dist1
    ans=min(ans,dist[dst])
return ans if ans!=float('inf') else -1
```

偶遇dp，拼尽全力无法战胜

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。

思想：动态规划 + 三重循环

状态定义：`dist[i][j]` 表示 i 到 j 的最短路径长度

转移方程：

`dist[i][j]=min(dist[i][j], dist[i][k]+dist[k][j])`

表示是否通过中间点 k 能让路径更短

最终得出任意两点之间的最短路径

用python实现

```
from collections import defaultdict
n,m=map(int,input().split())
graph=defaultdict(dict)
for _ in range(m):
    vs,ve,w=map(int,input().split())
    graph[vs][ve]=w
dist=[[float('inf')]*n for i in range(n)]
for i in range(n):
    dist[i][i]=0
    for j in graph[i].keys():
        dist[i][j]=graph[i][j]
print(*dist,sep='\n')
for k in range(n):
    for i in range(n):
        for j in range(n):
            dist[i][j]=min(dist[i][k]+dist[k][j],dist[i][j])
print(*dist,sep='\n')
```

需要求出路经时：

http://cs101.openjudge.cn/practice/05443/ 兔子与樱花

```
inf=float('inf')
p=int(input())
l=[]
dic={}
for i in range(p):
    space=input()
    l.append(space)
    dic[space]=i

q=int(input())
graph=[[inf]*p for i in range(p)]
next=[[-1]*p for i in range(p)]
for i in range(q):
    svs,sve,w=input().split()
    vs,ve=dic[svs],dic[sve]
    if graph[vs][ve]>int(w):
        graph[vs][ve]=graph[ve][vs]=int(w)
        next[vs][ve]=ve
        next[ve][vs]=vs
for i in range(p):
    graph[i][i]=0

for k in range(p):
    for i in range(p):
        for j in range(p):
            dist=graph[i][k] + graph[k][j]
            if graph[i][j]>dist:
                graph[i][j]=dist
                next[i][j]=next[i][k]
def find(i,j):
    if i==j:
        return l[i]
    ans=l[i]
    while next[i][j]!=j:
        sep=next[i][j]
        ans+=f"->({graph[i][sep]})->{l[sep]}"
        i=sep
    ans+=f"->({graph[i][j]})->{l[j]}"
    return ans
r=int(input())
for _ in range(r):
    svs,sve=input().split()
    vs,ve=dic[svs],dic[sve]
    print(find(vs,ve))
```

1.2.2.3 最小生成树

最小生成树的正式定义如下：对于图G=(V, E)，最小生成树T是E的无环子集，并且连接V 中的所有顶点，并且T中边集合的权重之和最小。

1.prim算法

直观地说就是，n个点的树至少要n-1条边，每一步都取一条老边和一条新边，并且取满足此要求代价最小的。

由于每一步都选择代价最小的下一步，因此Prim算法属于一种“贪婪算法”。在这个问题中，代价最小的下一步是选择权重最小的边

时间复杂度为O(|E| log |V|)

示例：https://leetcode.cn/problems/min-cost-to-connect-all-points/连接所有点的最小费用

```
from heapq import heappush,heapify,heappop
class Solution(object):
    def minCostConnectPoints(self, points):
        """
        :type points: List[List[int]]
        :rtype: int
        """
        n=len(points)
        if n<=1:
            return 0
        graph={i:[] for i in range(n)}
        for i in range(n):
            for j in range(i+1,n):
                xi,yi=points[i]
                xj,yj=points[j]
                l=abs(xi-xj)+abs(yi-yj)
                graph[i].append((l,j))
                graph[j].append((l,i))
        visit=[False]*n
        visit[0]=True
        h=graph[0]
        heapify(h)
        ans=0
        cnt=1
        while h and cnt<n:
            cost,vert=heappop(h)
            if not visit[vert]:
                visit[vert]=True
                ans+=cost
                cnt+=1
                for cn,nbr in graph[vert]:
                    if not visit[nbr]:
                        heappush(h,(cn,nbr))
        return ans
```

2.Kruskal算法

Kruskal算法是一种用于解决最小生成树（MST）问题的贪心算法。它通过不断选择具有最小权重的边，并确保选择的边不形成环，最终构建出一个包含所有顶点的最小生成树。

在Kruskal算法中，通常会使用并查集来维护图中顶点的连通性信息。当选择一条边时，通过并查集判断该边的两个端点是否属于同一个连通分量，以避免形成环。

并查集是一种数据结构，用于管理元素的不相交集合。它通常支持两种操作：查找（Find）和合并（Union）。查找操作用于确定某个元素属于哪个集合，合并操作用于将两个集合合并为一个集合。

在Kruskal算法中，使用并查集来快速判断两个顶点是否属于同一个连通分量。当遍历边并选择加入最小生成树时，可以通过并查集来检查该边的两个端点是否已经在同一个连通分量中，以避免形成环。

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。
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：
    - 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算法的时间复杂度为 O(ElogE)，其中 E 是边的数量，适用稀疏图

代码实现：

```
class disjointset:
    def __init__(self,num):
        self.parent=[i for i in range(num)]
        self.rank=[0]*num
    def find(self,i):
        if self.parent[i]!=i:
            self.parent[i]=self.find(self.parent[i])
        return self.parent[i]
    def union(self,i,j):
        rooti=self.find(i)
        rootj=self.find(j)
        if rooti!=rootj:
            if self.rank[i]<self.rank[j]:
                self.parent[rooti]=rootj
            if self.rank[j]<self.rank[i]:
                self.parent[rootj]=rooti
            else:
                self.parent[rooti]=rootj
                self.rank[j]+=1
n,m=map(int,input().split())
edges=[list(map(int,input().split())) for _ in range(m)]
edges.sort(key=lambda x:x[2])
djset=disjointset(n)
ans=[]
for u,v,w in edges:
    if djset.find(u)!=djset.find(v):
        djset.union(u,v)
        ans.append((u,v,w))
print(*ans)
```

示例：https://leetcode.cn/problems/min-cost-to-connect-all-points/连接所有点的最小费用

```
class disjointset:
    def __init__(self,num):
        self.parent=[i for i in range(num)]
        self.rank=[0]*num
```

```
    def find(self,i):
        if self.parent[i]!=i:
            self.parent[i]=self.find(self.parent[i])
        return self.parent[i]
    def union(self,i,j):
        rooti=self.find(i)
        rootj=self.find(j)
        if rooti==rootj:
            return False

        if self.rank[i]<self.rank[j]:
            self.parent[rooti]=rootj
        if self.rank[j]<self.rank[i]:
            self.parent[rootj]=rooti
        else:
            self.parent[rooti]=rootj
            self.rank[j]+=1
        return True
class Solution(object):
    def minCostConnectPoints(self, points):
        """
        :type points: List[List[int]]
        :rtype: int
        """
        n=len(points)
        if n<=1:
            return 0
        edges=[]
        for i in range(n):
            for j in range(i+1,n):
                xi,yi=points[i]
                xj,yj=points[j]
                l=abs(xi-xj)+abs(yi-yj)
                edges.append([i,j,l])
        edges.sort(key=lambda x:x[2])
        djset=disjointset(n)
        ans=0
        num=0
        for u,v,w in edges:
            if djset.union(u,v):
                ans+=w
                num+=1
            if num==n:
                break
        return ans
```

2.一些杂的代码

2.1 Counter

```
from collections import Counter
a = [12, 3, 4, 3, 5, 11, 12, 6, 7]
x=Counter(a)
for i in x.keys():
```

```
      print(i, ":", x[i])
x_keys = list(x.keys()) #[12, 3, 4, 5, 11, 6, 7]
x_values = list(x.values()) #[2, 2, 1, 1, 1, 1, 1]
for i in x.elements():
    print ( i, end = " ") #[12,12,3,3,4,5,11,6,7]
c=Counter('12131233435212312555555555')
cc=sorted(c.items(),key=lambda x:x[1],reverse=True)
#[('5', 9), ('1', 5), ('2', 5), ('3', 5), ('4', 1)]
```

2.2 cmp_to_key

```
from functools import cmp_to_key
def compar(a,b):
    if a>b:
        return 1#大的在后
    if a<b:
        return -1#小的在前
    else:
        return 0#返回零不变位置
l=[1,5,2,4,6,7,6]
l.sort(key=cmp_to_key(compar))
print(l)#[1,2,4,5,6,6,7]
```

2.3 保留小数

```
number = 3.14159
formatted_number = "{:.2f}".format(number)
print(formatted_number)  # 输出: 3.14
```

2.4 全排列

```
from itertools import permutations
# Get all permutations of [1, 2, 3]
perm = permutations([1, 2, 3])
# Get all permutations of length 2
perm2 = permutations([1, 2, 3], 2)
# Print the obtained permutations
for i in list(perm):
    print (i)
```

2.5 二分查找

```
def bisect_right(a,x):
    hi=len(a)
    lo=0
    while lo<hi:
        mid=(lo+hi)//2
        if x<a[mid]:
            hi=mid
        else:
            lo=mid+1
    return lo
```

```
def bisect_left(a,x):
    hi=len(a)
    lo=0
    while lo<hi:
        mid=(lo+hi)//2
        if x<=a[mid]:
            hi=mid
        else:
            lo=mid+1
    return lo
a=[1,2,2,2,3,4]
x=2
print(bisect_left(a,x),bisect_right(a,x))#1,4
直接引库
import bisect
bisect.bisect(a,x)
bisect.bisect_left(a,x)
bisect_insort(a,x)(直接利用insect(lo,x))
bisect.bisect_left(a,x)
```

binary search + greedy二分:

河中跳房子

```
L,n,m = map(int,input().split())
rock = [0]
for i in range(n):
    rock.append(int(input()))
rock.append(L)

def check(x):
    num = 0
    now = 0
    for i in range(1, n+2):
        if rock[i] - now < x:
            num += 1
        else:
            now = rock[i]

    if num > m:
        return True
    else:
        return False
lo, hi = 0, L+1
ans = -1
while lo < hi:
    mid = (lo + hi) // 2

    if check(mid):
        hi = mid
    else:               # 返回False，有可能是num==m
        ans = mid       # 如果num==m, mid就是答案
        lo = mid + 1

#print(lo-1)
```

```
print(ans)
```

2.6 滑动窗口

滑动窗口是一个队列，比如例题中的 abcabcbb，进入这个队列（窗口）为 abc 满足题目要求，当再进入 a，队列变成了 abca，这时候不满足要求。所以，我们要移动这个队列！如何移动？我们只要把队列的左边的元素移出就了，直到满足题目要求！一直维持这样的队列，找出队列出现最长的长度时候！时间复杂度：O(n)

```
d={}
ma=0
start=0
for i,ele in enumerate(s):
    if ele in d and d[ele]>=start:
        start=d[ele]+1
    d[ele]=i
    ma=max(ma,i-start+1)
return ma
```

2,7 单调栈

即人为控制栈内元素单调，找某侧最近一个比其大的值，使用单调栈维持栈内元素递减；找某侧最近一个比其小的值使用单调栈，维持栈内元素递增 ....

```
stack=[]
water=0
n=len(height)
for i in range(n):
    while stack and height[stack[-1]]<height[i]:
        top=stack.pop()
        if not stack:
            break
        d=i-stack[-1]-1
        h=min(height[stack[-1]],height[i])-height[top]
        water+=d*h
    stack.append(i)
return water
```

滑动窗口最大值（长度为k的滑动窗口，给出每次窗口中最大值）：同样是维护递减双端序列

```
x from collections import dequere
s=[]
de=deque()
if not nums or k==0:
    return []
if k==1:
    return nums
for i in range(len(nums)):
    if de and de[0]<i-k+1:
        de.popleft()
    while de and nums[de[-1]]<nums[i]:
```

```
        de.pop()
    de.append(i)
    if i>=k-1:
        res.append(nums[de[0]])return res
```

2.8 KMP

```
"""
compute_lps 函数用于计算模式字符串的LPS表。LPS表是一个数组，
其中的每个元素表示模式字符串中当前位置之前的子串的最长前缀后缀的长度。
该函数使用了两个指针 length 和 i，从模式字符串的第二个字符开始遍历。
"""
def compute_lps(pattern):
    """
    计算pattern字符串的最长前缀后缀（Longest Proper Prefix which is also Suffix）表
    :param pattern: 模式字符串
    :return: lps表
    """

    m = len(pattern)
    lps = [0] * m  # 初始化lps数组
    length = 0  # 当前最长前后缀长度
    for i in range(1, m):  # 注意i从1开始，lps[0]永远是0
        while length > 0 and pattern[i] != pattern[length]:
            length = lps[length - 1]  # 回退到上一个有效前后缀长度
        if pattern[i] == pattern[length]:
            length += 1
        lps[i] = length

    return lps

def kmp_search(text, pattern):
    n = len(text)
    m = len(pattern)
    if m == 0:
        return 0
    lps = compute_lps(pattern)
    matches = []

    # 在 text 中查找 pattern
    j = 0  # 模式串指针
    for i in range(n):  # 主串指针
        while j > 0 and text[i] != pattern[j]:
            j = lps[j - 1]  # 模式串回退
        if text[i] == pattern[j]:
            j += 1
        if j == m:
            matches.append(i - j + 1)  # 匹配成功
            j = lps[j - 1]  # 查找下一个匹配

    return matches


text = "ABABABABCABABABABCABABABABC"
pattern = "ABABCABAB"
```

```
index = kmp_search(text, pattern)
print("pos matched: ", index)
# pos matched:  [4, 13]
```

对于某一字符串 $S[1 \sim i]$，在它的lps[i]的候选值中，若存在某一 next[i]使得：

注意这个i是从1开始的，写代码通常从0开始。

$$i \mod (i - lps[i]) = 0$$

那么：

- $S[1 \sim (i - lps[i])]$ 是 $S[1 \sim i]$ 的**最小循环元**（最小周期子串）；
- $K = \frac{i}{i - lps[i]}$ 是这个循环元在 $S[1 \sim i]$ 中出现的次数。

2.9 十大排序

包括：冒泡排序（Bubble Sort），插入排序（Insertion Sort），选择排序（Selection Sort），希尔排序（Shell Sort），归并排序（Merge Sort），快速排序（Quick Sort），堆排序（Heap Sort），计数排序（Counting Sort），桶排序（Bucket Sort），基数排序（Radix Sort）

2.9.1 冒泡

```
def bubbleSort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if (swapped == False):
            break
```

2.9.2 选择排序

```
A = [64, 25, 12, 22, 11]
for i in range(len(A)):
 min_idx = i
 for j in range(i + 1, len(A)):
     if A[min_idx] > A[j]:
         min_idx = j
 A[i], A[min_idx] = A[min_idx], A[i]
print(' '.join(map(str, A)))
```

2.9.3 快排

```
def quicksort(arr, left, right):
    if left < right:
        partition_pos = partition(arr, left, right)
        quicksort(arr, left, partition_pos - 1)
        quicksort(arr, partition_pos + 1, right)
def partition(arr, left, right):
    i = left
```

```
    j = right - 1
    pivot = arr[right]
    while i <= j:
        while i <= right and arr[i] < pivot:
            i += 1
        while j >= left and arr[j] >= pivot:
            j -= 1
        if i < j:
            arr[i], arr[j] = arr[j], arr[i]
    if arr[i] > pivot:
        arr[i], arr[right] = arr[right], arr[i]
    return i
arr = [22, 11, 88, 66, 55, 77, 33, 44]
quicksort(arr, 0, len(arr) - 1)
print(arr)
```

2.9.4 归并排序（稳定还nlogn）

```
def mergeSort(arr):
    if len(arr) > 1:
        mid = len(arr)//2
        L = arr[:mid]
        R = arr[mid:]
        mergeSort(L) # Sorting the first half
        mergeSort(R) # Sorting the second half
        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1
        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

2.9.5 插入排序

```
def insertion_sort(arr):
    n = len(arr)
    for i in range(1, n):
        key = arr[i]  # 取出未排序部分的第一个元素
        j = i - 1
        # 将 key 插入到已排序部分的正确位置
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]  # 向后移动元素
            j -= 1
        arr[j + 1] = key  # 插入 key
```

2.9.6 希尔排序

```
def shellSort(arr, n):
    gap = n // 2
    while gap > 0:
        j = gap
        while j < n:
            i = j - gap  # This will keep help in maintain gap value
            while i >= 0:
                if arr[i + gap] > arr[i]:
                    break
                else:
                    arr[i + gap], arr[i] = arr[i], arr[i + gap]
                i = i - gap  # To check left side also
            j += 1
        gap = gap // 2
```

2.10 递归优化

1.增加递归深度限制

```
import sys
sys.setrecursionlimit(1 << 30) # 将递归深度限制设置为 2^30
```

2.缓存中间结果

建个列表or字典调用或者直接内置函数缓存中间结果

```
from functools import lru_cache
@lru_cache(maxsize=None)
```

3.基础数据结构

3.1栈

栈（有时被称为“压入栈”）是一种有序的项目集合，其中新项目的添加和现有项目的移除总是在同一端进行。这一端通常被称为“顶端”。与顶端相对的另一端被称为“基底”。

栈的基底是重要的，因为存储在栈中靠近基底的项目代表了在栈中最久的项目。最近添加的项目是准备首先被移除的那个。这种排序原则有时被称为**LIFO（后进先出）**。它提供了一种基于项目在集合中停留时间长短的排序。较新的项目接近顶端，而较旧的项目接近基底。

3.1.1 匹配括号

有多种括号：

```
def par_checker(symbol_string):
    s = [] # Stack()
    balanced = True
    index = 0
    while index < len(symbol_string) and balanced:
        symbol = symbol_string[index]
        if symbol in "([{":
            s.append(symbol) # push(symbol)
        else:
            top = s.pop()
            if not matches(top, symbol):
                balanced = False
        index += 1
        #if balanced and s.is_empty():
        if balanced and not s:
            return True
        else:
            return False
def matches(open, close):
    opens = "([{"
    closes = ")]}"
    return opens.index(open) == closes.index(close)
print(par_checker('{{}}[]]'))
```

例http://cs101.openjudge.cn/practice/03704括号匹配问题

```
lines = []
while True:
    try:
        lines.append(input())
    except EOFError:
        break
for s in lines:
    l=[]
    mark=[]
    for i in range(len(s)):
        if s[i]=='(':
            l.append(i)
            mark+=' '
        elif s[i]==')':
            if not l:
                mark+='?'
            else:
                l.pop()
                mark+=' '
        else:
            mark+=' '
```

```
    while l:
        ind=l.pop()
        mark[ind]='$'
    print(s)
    print(''.join(mark))
```

3.1.2 中序、前序和后序表达式(含调度场算法)

A + B * C这种类型的表示法被称为**中缀**表示法，因为操作符位于它所操作的两个操作数之间

前缀表达式要求所有操作符都在其工作的两个操作数之前，而后缀则要求其操作符在其对应的操作数之后

A + B * C 可以写成 (A + (B * C))，每对括号也标明了一个操作数对的开始和结束，其中间是相应的操作符。如果将操作符移动到其对应的右括号位置，并移除匹配的左括号，就得到了后缀表达式。因此，为了将一个表达式（无论多么复杂)转换为前缀或后缀表示法，首先使用运算顺序完全加上括号。然后，根据你想要得到前缀还是后缀表示法，将括号内的操作符移动到左括号或右括号的位置。

中序转后序：调度场算法

1. 创建一个名为 `opstack` 的空栈用于存放操作符，并创建一个空列表用于输出。
2. 使用字符串方法 `split` 将输入的中缀字符串转换成列表。
3. 从左到右扫描标记列表：如果标记是操作数，则将其附加到输出列表的末尾。

   如果标记是左括号，则将其压入 `opstack` 栈中。

   如果标记是右括号，则从 `opstack` 中弹出元素直到移除相应的左括号为止，并将每个操作符附加到输出列表的末尾。

   如果标记是操作符 *, /, + 或 -，则将其压入 `opstack` 栈中。但是，首先移除已在 `opstack` 上且具有更高或相同优先级的所有操作符，并将它们附加到输出列表的末尾。
4. 当输入表达式被完全处理后，检查 `opstack`。栈上任何剩余的操作符都可以被移除并附加到输出列表的末尾。

例：中序表达式转后序表达式http://cs101.openjudge.cn/practice/24591/

```
n=int(input())
value={'(':1,'+':2,'-':2,'*':3,'/':3}
for _ in range(n):
    put=input()
    stack=[]
    out=[]
    number=''
    for s in put:
        if s.isnumeric() or s=='.':
            number+=s
        else:
            if number:
                num=float(number)
                out.append(int(num) if num.is_integer() else num)
                number=''
            if s=='(':
                stack.append(s)
            elif s==')':
                while stack and stack[-1]!='(':
                    out.append(stack.pop())
```

```
                stack.pop()
            else:
                while stack and value[stack[-1]]>=value[s]:
                    out.append(stack.pop())
                stack.append(s)
    if number:
        num = float(number)
        out.append(int(num) if num.is_integer() else num)
    while stack:
        out.append(stack.pop())
    print(*out,sep=' ')
```

3.1.3 后序表达式求值

在读取后缀表达式的过程中，每当你遇到一个操作数，就将其压入栈中。一旦遇到操作符，就从栈中弹出适当数量的操作数（对于二元操作符来说是两个），执行相应的计算，并将结果压回到栈中。这个过程会一直持续到表达式的末尾，最终栈顶元素就是整个表达式的计算结果

例 后序表达式求值http://cs101.openjudge.cn/practice/24588/

```
def doMath(op, op1, op2):
    if op == "*":
        return op1 * op2
    elif op == "/":
        return op1 / op2
    elif op == "+":
        return op1 + op2
    else:
        return op1 - op2
for _ in range(int(input())):
    put=input().split()
    stack=[]
    for token in put:
        if token in '+-*/':
            a=stack.pop()
            b=stack.pop()
            stack.append(doMath(token,b,a))
        else:
            stack.append(float(token))
    print(f"{stack[-1]:.2f}")
```

3.2队列

队列是一种有序的项目集合，其中新项目的添加发生在一端，这端被称为“尾部”，而现有项目的移除则发生在另一端，通常称为“前端”。当元素进入队列时，它从尾部开始，向前端移动，直到成为下一个要被移除的元素为止。

队列中最新增加的项目必须在集合的末尾等待。在集合中最久的项目位于前端。这种排序原则有时被称为**FIFO（先进先出）**，也称为“先来先服务”。

3.3线性表/线性结构

是一种逻辑结构，描述了元素按线性顺序排列的规则。常见的线性表存储方式有**数组**和**链表**，它们在不同场景下具有各自的优势和劣势。

数组是一种连续存储结构，它将线性表的元素按照一定的顺序依次存储在内存中的连续地址空间上。数组需要预先分配一定的内存空间，每个元素占用相同大小的内存空间，并可以通过索引来进行快速访问和操作元素。访问元素的时间复杂度为$O(1)$，因为可以直接计算元素的内存地址。然而，插入和删除元素的时间复杂度较高，平均为$O(n)$，因为需要移动其他元素来保持连续存储的特性。

链表是一种存储结构，它是线性表的链式存储方式。链表通过节点的相互链接来实现元素的存储。每个节点包含元素本身以及指向下一个节点的指针。链表的插入和删除操作非常高效，时间复杂度为$O(1)$，因为只需要调整节点的指针。然而，访问元素的时间复杂度较高，平均为$O(n)$，因为必须从头节点开始遍历链表直到找到目标元素。

3.3.1顺序表/数组/列表

关于线性表的时间复杂度：

生成、求表中元素个数、表尾添加/删除元素、返回/修改对应下标元素，均为$O(1)$；

而查找、删除、插入元素，均为$O(n)$。

线性表的优缺点：

优点：1、无须为表中元素之间的逻辑关系而增加额外的存储空间；

2、可以快速的存取表中任一位置的元素。

缺点：1、插入和删除操作需要移动大量元素；

2、当线性表长度较大时，难以确定存储空间的容量；

3、造成存储空间的“碎片”。

3.3.2 链表

在链表中，每个节点都包含两部分：

1. 数据元素（或数据项）：这是节点存储的实际数据。可以是任何数据类型，例如整数、字符串、对象等。
2. 指针（或引用）：该指针指向链表中的下一个节点（或前一个节点）。它们用于建立节点之间的连接关系，从而形成链表的结构。

根据指针的类型和连接方式，链表可以分为不同类型，包括：

1. 单向链表：每个节点只有一个指针，指向下一个节点。链表的头部指针指向第一个节点，而最后一个节点的指针为空（指向 `None`）。
2. 双向链表：每个节点有两个指针，一个指向前一个节点，一个指向后一个节点。双向链表可以从头部或尾部开始遍历，并且可以在任意位置插入或删除节点。
3. 循环链表：最后一个节点的指针指向链表的头部，形成一个环形结构。循环链表可以从任意节点开始遍历，并且可以无限地循环下去。

3.3.2.1 单向链表

单链表的特点是每个节点只有一个指针，指向下一个节点。因此，它是单向的，只能从头到尾遍历。

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None
class LinkedList:
    def __init__(self):
        self.head = None
    def insert(self, value):
        new_node = Node(value)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
    def delete(self, value):
        if self.head is None:
            return
        if self.head.value == value:
            self.head = self.head.next
        else:
            current = self.head
            while current.next:
                if current.next.value == value:
                    current.next = current.next.next
                    break
                current = current.next
    def display(self):
        current = self.head
        while current:
            print(current.value, end=" ")
            current = current.next
        print()
```

单链表的应用

动态内存管理：链表可以灵活地分配内存空间，特别适用于内存空间不固定的场景。

实现队列和栈：链表能够有效地支持栈（LIFO）和队列（FIFO）的实现，因为其在插入和删除操作上有优势。

动态集合管理：对于集合操作（如动态插入和删除元素）非常高效。

3.3.2.2 双向链表

双向链表（Doubly Linked List）是一种数据结构，其中每个节点不仅包含指向下一个节点的指针（`next`），还包含指向前一个节点的指针（`prev`）。这样，双向链表能够在两端进行遍历：从头到尾和从尾到头。

```
class Node:
    def __init__(self, data):
```

```
        self.data = data  # 节点数据
        self.next = None  # 指向下一个节点
        self.prev = None  # 指向前一个节点
class DoublyLinkedList:
    def __init__(self):
        self.head = None  # 链表头部
        self.tail = None  # 链表尾部
    def append(self, data):
        new_node = Node(data)
        if not self.head:  # 如果链表为空
            self.head = new_node
            self.tail = new_node
        else:
            self.tail.next = new_node
            new_node.prev = self.tail
            self.tail = new_node
    def prepend(self, data):
        new_node = Node(data)
        if not self.head:  # 如果链表为空
            self.head = new_node
            self.tail = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node
    def delete(self, node):
        if not self.head:  # 链表为空
            return
        if node == self.head:  # 删除头部节点
            self.head = node.next
            if self.head:  # 如果链表非空
                self.head.prev = None
        elif node == self.tail:  # 删除尾部节点
            self.tail = node.prev
            if self.tail:  # 如果链表非空
                self.tail.next = None
        else:  # 删除中间节点
            node.prev.next = node.next
            node.next.prev = node.prev
        node = None  # 删除节点
    def print_list(self):
        current = self.head
        while current:
            print(current.data, end=" <-> ")
            current = current.next
        print("None")
    def print_reverse(self):
        current = self.tail
        while current:
            print(current.data, end=" <-> ")
            current = current.prev
        print("None")
```

双链表的应用

双向遍历：由于双链表可以从头到尾或从尾到头遍历，因此在某些需要双向遍历的数据结构（如浏览器历史记录、操作系统任务调度等）中非常有用。

实现双端队列（Deque）：双链表非常适合用于双端队列的实现，可以在队头和队尾都进行快速的插入和删除。

内存管理和垃圾回收：双链表用于管理动态内存块，常见于操作系统的内存管理和垃圾回收机制中

https://leetcode.cn/problems/design-browser-history/设计浏览器历史记录

```
class ListNode:
    def __init__(self, url: str):
        self.url = url
        self.prev = None
        self.next = None
class BrowserHistory:
    def __init__(self, homepage: str):
        self.current = ListNode(homepage)
    def visit(self, url: str) -> None:
        new_node = ListNode(url)
        self.current.next = new_node
        new_node.prev = self.current
        self.current = new_node
    def back(self, steps: int) -> str:
        while steps > 0 and self.current.prev is not None:
            self.current = self.current.prev
            steps -= 1
        return self.current.url
    def forward(self, steps: int) -> str:
        while steps > 0 and self.current.next is not None:
            self.current = self.current.next
            steps -= 1
        return self.current.url
```

3.3.2.3 循环链表

```
class CircleLinkList:
    class Node:
        def __init__(self, data, next=None):
            self.data = data
            self.next = next
    def __init__(self):
        self.tail = None  # 尾指针，指向最后一个节点
        self.size = 0  # 链表大小
    def is_empty(self):
        """检查链表是否为空"""
        return self.size == 0
    def pushFront(self, data):
        """在链表头部插入元素"""
        nd = CircleLinkList.Node(data)
        if self.is_empty():
            self.tail = nd
            nd.next = self.tail  # 自己指向自己形成环
        else:
            nd.next = self.tail.next  # 新节点指向当前头节点
            self.tail.next = nd  # 当前尾节点指向新节点
```

```
        self.size += 1
    def pushBack(self, data):
        """在链表尾部插入元素"""
        nd = CircleLinkList.Node(data)
        if self.is_empty():
            self.tail = nd
            nd.next = self.tail  # 自己指向自己形成环
        else:
            nd.next = self.tail.next  # 新节点指向当前头节点
            self.tail.next = nd  # 当前尾节点指向新节点
            self.tail = nd  # 更新尾指针
        self.size += 1
    def popFront(self):
        """移除并返回链表头部元素"""
        if self.is_empty():
            return None
        else:
            old_head = self.tail.next
            if self.size == 1:
                self.tail = None  # 如果只有一个元素，更新尾指针为None
            else:
                self.tail.next = old_head.next  # 跳过旧头节点
            self.size -= 1
            return old_head.data
    def popBack(self):
        """移除并返回链表尾部元素"""
        if self.is_empty():
            return None
        elif self.size == 1:
            data = self.tail.data
            self.tail = None
            self.size -= 1
            return data
        else:
            prev = self.tail
            while prev.next != self.tail:  # 找到倒数第二个节点
                prev = prev.next
            data = self.tail.data
            prev.next = self.tail.next  # 跳过尾节点
            self.tail = prev  # 更新尾指针
            self.size -= 1
            return data
    def printList(self):
        """打印链表中的所有元素"""
        if self.is_empty():
            print('Empty!')
        else:
            ptr = self.tail.next
            while True:
                print(ptr.data, end=', ' if ptr != self.tail else '\n')
                if ptr == self.tail:
                    break
                ptr = ptr.next
```

3.3.2.4 常见链表的操作

3.3.2.4.1 链表反转

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
def reverse_linked_list(head: ListNode) -> ListNode:
    prev = None
    curr = head
    while curr is not None:
        next_node = curr.next  # 暂存当前节点的下一个节点
        curr.next = prev       # 将当前节点的下一个节点指向前一个节点
        prev = curr            # 前一个节点变为当前节点
        curr = next_node       # 当前节点变更为原先的下一个节点
    return prev
```

3.3.2.4.2 合并两个排序的链表

```
def merge_sorted_lists(l1, l2):
    dummy = Node(0)
    tail = dummy
    while l1 and l2:
        if l1.data < l2.data:
            tail.next = l1
            l1 = l1.next
        else:
            tail.next = l2
            l2 = l2.next
        tail = tail.next
    if l1:
        tail.next = l1
    else:
        tail.next = l2
    return dummy.next
```

3.3.2.4.3 查找链表的中间节点

```
def find_middle_node(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow
```

3.4 双端队列

与栈和队列不同的是，双端队列的限制很少。双端队列是与队列类似的有序集合。它有一前、一后两端，元素在其中保持自己的位置。与队列不同的是，双端队列对在哪一端添加和移除元素没有任何限制。新元素既可以被添加到前端，也可以被添加到后端。同理，已有的元素也能从任意一端移除。

解决回文的妙法：

回文数字http://cs101.openjudge.cn/practice/04067/

```python
from collections import deque
def is_palindrome(num):
    num_str = str(num)
    num_deque = deque(num_str)
    while len(num_deque) > 1:
        if num_deque.popleft() != num_deque.pop():
            return "NO"
    return "YES"
while True:
    try:
        num = int(input())
        print(is_palindrome(num))
    except EOFError:
        break
```

4 树

4.1术语

节点 Node：节点是树的基础部分。
每个节点具有名称，或“键值”。节点还可以保存额外数据项，数据项根据不同的应用而变。

边 Edge：边是组成树的另一个基础部分。
每条边恰好连接两个节点，表示节点之间具有关联，边具有出入方向；
每个节点（除根节点）恰有一条来自另一节点的入边；
每个节点可以有零条/一条/多条连到其它节点的出边。如果加限制不能有 “多条边”，这里树结构就特殊化为线性表

根节 Root: 树中唯一没有入边的节点。

路径 Path：由边依次连接在一起的有序节点列表。比如，哺乳纲→食肉目→猫科→猫属→家猫就是一条路径。

子节点 Children：入边均来自于同一个节点的若干节点，称为这个节点的子节点。

父节点 Parent：一个节点是其所有出边连接节点的父节点。

兄弟节点 Sibling：具有同一父节点的节点之间为兄弟节点。

子树 Subtree：一个节点和其所有子孙节点，以及相关边的集合。

叶节点 Leaf Node：没有子节点的节点称为叶节点。

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。

高度 Height：树中所有节点的最大层级称为树的高度

对于只有一个节点的树来说，高度为0，深度为0。如果是空树，高度、深度都是 -1.

深度：通常是指从根节点到某个节点的边数。对于空树，深度没有意义，也可以定义为 -1。

树 Tree:

定义一：树由节点及连接节点的边构成。树有以下属性：
有一个根节点；
除根节点外，其他每个节点都与其唯一的父节点相连；
从根节点到其他每个节点都有且仅有一条路径；

如果每个节点最多有两个子节点，我们就称这样的树为二叉树。

定义二：一棵树要么为空，要么由一个根节点和零棵或多棵子树构成，子树本身也是一棵树。每棵子树的根节点通过一条边连到父树的根节点。

二叉树（Binary Tree）

二叉树是树的一种特殊形式，每个节点最多有**两个子节点**：

满二叉树（Full Binary Tree）：所有非叶子节点都有两个子节点。

完全二叉树（Complete Binary Tree）：只有最后一层可以不满，并且节点从左到右排列。

平衡二叉树（Balanced Binary Tree）：左右子树的高度差不超过 1，如 AVL 树。

二叉搜索树（Binary Search Tree，BST）：对于任意节点，左子树的所有节点值小于该节点值，右子树的所有节点值大于该节点值。

4.2 n阶多叉树的表示方法

普通树（Generic trees）是由若干节点组成的集合，其中每个节点是一个数据结构，包含记录和一个指向其子节点的引用列表（不允许重复引用）。与链表不同，每个节点存储了多个节点的地址。每个节点存储其子节点的地址，而第一个节点的地址则存储在一个名为根（root）的独立指针中。

我们可以使用**动态数组**来存储子节点的地址。它既可以随机访问任意子节点的地址，其大小（容量）也没有固定的限制。

```
class Node:
    def __init__(self,data):
        self.data=data
        self.children=[]
```

高效方法：长子-兄弟表示法

在“长子 / 下一个兄弟”表示法中，采取的步骤如下：

1. 将同一父节点的所有子节点（即兄弟节点）从左到右链接起来。
2. 移除父节点到所有子节点的链接，只保留到第一个孩子的链接。

由于子节点之间已经建立了链接，因此不需要从父节点到所有子节点的额外链接。这种表示法允许我们通过从父节点的第一个孩子开始，遍历所有的元素。

```
class Node:
    def __init__(self, data):
        self.data = data
        self.firstChild = None
        self.nextSibling = None
```

可视为二叉树 – 由于我们能够将任何通用树转换为二叉树表示形式，因此可以将所有使用“第一个孩子/下一个兄弟”表示法的通用树视为二叉树。我们只需使用 `firstChild`（第一个孩子）和 `nextSibling`（下一个兄弟），而不用传统的左指针和右指针。

4.3 二叉树的递归算法

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

4.3.1 遍历

前序遍历

```
def preorder_traversal(root):
    if root:
        print(root.val)  # 访问根节点
        preorder_traversal(root.left)  # 递归遍历左子树
        preorder_traversal(root.right)  # 递归遍历右子树
```

中序遍历

```
def inorder_traversal(root):
    if root:
        inorder_traversal(root.left)  # 递归遍历左子树
        print(root.val)  # 访问根节点
        inorder_traversal(root.right)  # 递归遍历右子树
```

颜色填充法

```
class Solution:
    def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        white, gray = 0, 1
        res = []
        stack = [(white, root)]
        while stack:
            color, node = stack.pop()
            if node is None: continue
            if color == white:
                stack.append((white, node.right))
                stack.append((gray, node))
                stack.append((white, node.left))
            else:
                res.append(node.val)
        return res
```

后序遍历

```
def postorder_traversal(root):
    if root:
        postorder_traversal(root.left)  # 递归遍历左子树
        postorder_traversal(root.right)  # 递归遍历右子树
        print(root.val)  # 访问根节点
```

层序遍历：按层从左到右依次遍历

```
class TreeNode(object):
     def __init__(self, val=0, left=None, right=None):
         self.val = val
         self.left = left
         self.right = right
class Solution(object):
    def levelOrder(self, root):
        if not root:
            return []
        queue=deque([root])
        ans=[]
        while queue:
            size=len(queue)
            level=[]
            for _ in range(size):
                node=queue.popleft()
                level.append(node.val)
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)
            ans.append(level)
        return ans
```

4.3.2 求树的高度/深度

```
def tree_height(root):
    if not root:  # 空树的高度为 0
        return 0
    left_height = tree_height(root.left)  # 左子树的高度
    right_height = tree_height(root.right)  # 右子树的高度
    return max(left_height, right_height) + 1  # 树的高度是左右子树最大高度加 1
```

例 二叉树的深度http://cs101.openjudge.cn/practice/06646/

需要根据输入建树，知道树根位置

```
class Node():
    def __init__(self,val=0,left=None,right=None):
        self.val=val
        self.left=left
        self.right=right
def build(nodes):
    n=len(nodes)
    tree=[None]+[Node(i) for i in range(1,n+1)]
    for i,(left,right) in enumerate(nodes,start=1):
        if left!=-1:
            tree[i].left=tree[left]
        if right!=-1:
            tree[i].right=tree[right]
    return tree[1]
def depth(root):
```

```
    if not root:
        return 0
    leftdepth=depth(root.left)
    rightdepth=depth(root.right)
    return max(leftdepth,rightdepth)+1
n=int(input())
nodes=[list(map(int,input().split())) for _ in range(n)]
root=build(nodes)
print(depth(root))
```

求二叉树的高度和叶子数目 http://cs101.openjudge.cn/practice/27638/

根据输入建树，找到树根。

```
class Node():
    def __init__(self):
        self.left=None
        self.right=None
def build(nodes):
    hasparent=[False]*n
    tree=[Node() for _ in range(n)]
    for i,(left,right) in enumerate(nodes):
        if left!=-1:
            tree[i].left=tree[left]
            hasparent[left]=True
        if right!=-1:
            tree[i].right=tree[right]
            hasparent[right]=True
    i=hasparent.index(False)
    root=tree[i]
    return root
def depth(root):
    if not root:
        return -1
    return max(depth(root.left),depth(root.right))+1
def leavecount(node):
    if not node:
        return 0
    if not node.left and not node.right:
        return 1
    return(leavecount(node.left)+leavecount(node.right))
n=int(input())
nodes=[list(map(int,input().split())) for _ in range(n)]
root=build(nodes)
print(depth(root),leavecount(root))
```

4.3.3 判断两棵树是否相同

```
def is_same_tree(p, q):
    if not p and not q:
        return True
    if not p or not q:
        return False
    return (p.val == q.val and
            is_same_tree(p.left, q.left) and
            is_same_tree(p.right, q.right))
```

例 对称二叉树 https://leetcode.cn/problems/symmetric-tree/

```
class Solution(object):
    def isSymmetric(self, root):
        """
        :type root: Optional[TreeNode]
        :rtype: bool
        """
        if not root:
            return True
        def mirror(leftnode,rightnode):
            if not leftnode and not rightnode:
                return True
            if not leftnode or not rightnode:
                return False
            return leftnode.val==rightnode.val and
mirror(leftnode.left,rightnode.right) and mirror(leftnode.right,rightnode.left)
        return mirror(root.left,root.right)
```

4.3.4 反转二叉树

```
def invert_tree(root):
    if root:
        root.left, root.right = invert_tree(root.right), invert_tree(root.left)
    return root
```

4.3.5 寻找二叉搜索树中的最小值/最大值

```
def find_min(root):
    if not root.left:
        return root.val
    return find_min(root.left)

def find_max(root):
    if not root.right:
        return root.val
    return find_max(root.right)
```

例 二叉搜索树中第K小的元素https://leetcode.cn/problems/kth-smallest-element-in-a-bst/

```
class Solution(object):
    def kthSmallest(self, root, k):
```

```
        """
        :type root: Optional[TreeNode]
        :type k: int
        :rtype: int
        """
        ans=[]
        def find(node):
            if not node or len(ans)==k:
                return
            find(node.left)
            if len(ans)<k:
                ans.append(node.val)
            find(node.right)
        find(root)
        return ans[-1]
```

4.3.6 判断是否为平衡二叉树

```
class TreeNode():
    def __init__(self,left=None,right=None):
        self.left=left
        self.right=right
def is_balanced(root):
    def check_height(node):
        if not node:
            return 0
        left=check_height(node.left)
        if left==-1:
            return -1
        right = check_height(node.right)
        if right == -1:
            return -1
        if abs(left-right)>1:
            return -1
        return max(left,right)+1
    return check_height(root)!=-1
```

4.4 树的表示方法(各种奇怪的建树题)

4.4.1 嵌套括号表示法

Nested parentheses representation 是一种表示树结构的方法，通过括号的嵌套来表示树的层次关系。

先将根结点放入一对圆括号中，然后把它的子树按由左而右的顺序放入括号中，而对子树也采用同样方法处理：同层子树与它的根结点用圆括号括起来，同层子树之间用逗号隔开，最后用闭括号括起来。例如下图可写成如下形式
$(a(b,c,d,e))$

例 括号嵌套二叉树 http://cs101.openjudge.cn/practice/27637

篇幅受限只展示如何把字符串变成树

```
def parse_tree(s):
    if s=='*':
        return None
```

```
    if '(' not in s:
        return TreeNode(s)
    rootval=s[0]
    root=TreeNode(rootval)
    child=s[2:-1]
    stack=[]
    dot=-1
    for i,token in enumerate(child):
        if token=='(':
            stack.append('(')
        elif token==')':
            stack.pop()
        elif token==',' and not stack:
            dot=i
            break
    root.left=parse_tree(child[:dot])
    root.right=parse_tree(child[dot+1:])
    return root
```

4.4.2 拓展二叉树

由于先序、中序和后序序列中的任一个都不能唯一确定一棵二叉树，所以对二叉树做如下处理，将二叉树的空结点用补齐，如图所示。我们把这样处理后的二叉树称为原二叉树的扩展二叉树，扩展二叉树的先序和后序序列能唯一确定其二叉树，如ABD..EF..G..C..

给出拓展二叉树前序序列建树的方法：

```
def build_tree(s, index):
    if s[index]=='.':
        return None,index+1
    root=Node(s[index])
    index+=1
    root.left,index=build_tree(s,index)
    root.right,index=build_tree(s,index)
    return root,index
```

4.4.3 邻接表示法

邻接表表示法（Adjacency List Representation）是一种常见的树的表示方法，特别适用于表示稀疏树（树中节点的度数相对较小）。在邻接表表示法中，使用一个数组来存储树的节点，数组中的每个元素对应一个节点。对于每个节点，使用链表或数组等数据结构来存储它的子节点。

将多叉树转为邻接表：

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []
def print_tree_adjacency_list(root):
    adjacency_list = {}
    def build_adjacency_list(node):
        adjacency_list[node.value] = [child.value for child in node.children]
        for child in node.children:
            build_adjacency_list(child)
    build_adjacency_list(root)
    for node, children in adjacency_list.items():
        print(f"{node}: {children}")
```

括号嵌套树http://cs101.openjudge.cn/practice/24729/

```
class Node():
    def __init__(self,val=0):
        self.val=val
        self.children=[]
def build(s):
    node=None
    stack=[]
    for token in s:
        if token.isalpha():
            node=Node(token)
            if stack:
                stack[-1].children.append(node)
        if token=='(':
            if node:
                stack.append(node)
                node=None
        if token==')':
            node=stack.pop()
    return node
def preorder(root):
    ans=[root.val]
    for child in root.children:
        ans.extend(preorder(child))
    return ans
def postorder(root):
    ans=[]
    for child in root.children:
        ans.extend(postorder(child))
    ans.append(root.val)
    return ans
s=input()
root=build(s)
print(''.join(preorder(root)))
print(''.join(postorder(root)))
```

4.4.4 前中序建树

```
class Solution(object):
    def buildTree(self, preorder, inorder):
        if not preorder and not inorder:
            return None
        root=TreeNode(preorder[0])
        i=inorder.index(root.val)
        root.left=self.buildTree(preorder[1:i+1],inorder[:i])
        root.right=self.buildTree(preorder[i+1:],inorder[i+1:])
        return root
```

4.5 各种奇怪的树

4.5.1解析树

(1) 如果当前标记是(，就为当前节点添加一个左子节点，并下沉至该子节点；
(2) 如果当前标记在列表`['+', '-', '/', '*']`中，就将当前节点的值设为当前标记对应的运算符；为当前节点添加一个右子节点，并下沉至该子节点；
(3) 如果当前标记是数字，就将当前节点的值设为这个数并返回至父节点；
(4) 如果当前标记是)，就跳到当前节点的父节点。

```
def buildParseTree(fpexp):
    fplist = fpexp.split()
    pStack = Stack()
    eTree = BinaryTree('')
    pStack.push(eTree)
    currentTree = eTree
    for i in fplist:
        if i == '(':
            currentTree.insertLeft('')
            pStack.push(currentTree)
            currentTree = currentTree.getLeftChild()
        elif i not in '+-*/)':
            currentTree.setRootVal(int(i))
            parent = pStack.pop()
            currentTree = parent
        elif i in '+-*/':
            currentTree.setRootVal(i)
            currentTree.insertRight('')
            pStack.push(currentTree)
            currentTree = currentTree.getRightChild()
        elif i == ')':
            currentTree = pStack.pop()
        else:
            raise ValueError("Unknown Operator: " + i)
    return eTree
```

计算方法：

```
import operator
def evaluate(parseTree):
    opers = {'+':operator.add, '-':operator.sub, '*':operator.mul,
'/':operator.truediv}
    leftC = parseTree.getLeftChild()
    rightC = parseTree.getRightChild()
    if leftC and rightC:
        fn = opers[parseTree.getRootVal()]
        return fn(evaluate(leftC),evaluate(rightC))
    else:
        return parseTree.getRootVal()
```

后序转编码树：

```
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]
```

编码树转中序只需中序遍历（有多余括号）

```
def printexp(tree):
    sVal = ""
    if tree:
        sVal = '(' + printexp(tree.getLeftChild())
        sVal = sVal + str(tree.getRootVal())
        sVal = sVal + printexp(tree.getRightChild()) + ')'
    return sVal
print(printexp(pt))
# (((7)+3)*((5)-2))
```

对于布尔算符计算的解析树转中序表达式可去多余括号：

1. 首先，检查树的根节点的值。根据值的不同，函数会执行不同的操作。
2. 如果根节点的值为"or"，函数会递归地调用自身来处理左子树和右子树，然后将结果合并，并在两个结果之间插入"or"。
3. 如果根节点的值为"not"，函数会递归地调用自身来处理左子树。如果左子树的根节点的值不是"True"或"False"，则会在左子树的结果周围添加括号。
4. 如果根节点的值为"and"，函数会递归地调用自身来处理左子树和右子树。如果左子树或右子树的根节点的值为"or"，则会在相应子树的结果周围添加括号。
5. 如果根节点的值为"True"或"False"，函数会直接返回一个包含该值的列表。
6. 最后，函数会将生成的字符串列表合并为一个字符串，并返回。

```
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())]
```

4.5.2 霍夫曼编码Huffman code

在文本压缩中，我们要对字符进行编码。在编码中没有任何码字是另一个码字的前缀。这样的编码被称为**前缀编码prefix code**

霍夫曼编码算法从字符串X中每个独特的d个字符开始，每个字符都是单节点二叉树的根节点。算法以一系列的轮次进行。在每一轮中，算法将具有最小频率的两棵二叉树合并为一棵二叉树。此过程重复进行，直到只剩下一棵树为止。

可以证明这样的贪心算法能使哈夫曼树的带权外部路径长度（各字符出现频率乘上编码长度之和）最小

```
from heapq import heappush,heappop,heapify
class Node():
    def __init__(self,val,fre):
        self.val=val
        self.fre=fre
        self.left=None
        self.right=None
    def __lt__(self,other):
        return self.fre<other.fre
def haffman(frequance):
    tree=[Node(i,frequance[i]) for i in range(n)]
    heapify(tree)
    while len(tree)!=1:
        a=heappop(tree)
        b=heappop(tree)
        node=Node(0,a.fre+b.fre)
        node.left=a
        node.right=b
        heappush(tree,node)
    return tree[0]
def cal(node,depth):
```

```
    if not node.left and not node.right:
        return node.fre*depth
    return cal(node.left,depth+1)+cal(node.right,depth+1)
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_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.val
            node = root
    return decoded
```

如果只需要求最小带权外部路径长度：

```
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))
```

4.5.3 二叉堆

在实现二叉堆时，我们通过创建一棵**完全二叉树complete binary tree**来维持树的平衡。在完全二叉树中，除了最底层，其他每一层的节点都是满的。

完全二叉树的另一个有趣之处在于，可以用一个列表来表示它，而不需要采用“列表之列表”或“节点与引用”表示法。由于树是完全的，因此对于在列表中处于位置 p 的节点来说，它的左子节点正好处于位置 2p+1；同理，右子节点处于位置 2p+2。若要找到树中任意节点的父节点，只需使用 Python 的整数除法即可。给定列表中位置 n 处的节点，其父节点的位置就是 (n-1)//2。

我们用来存储堆元素的方法依赖于堆的有序性。 堆的有序性是指：对于堆中任意元素 x 及其父元素 p，p 都不大于 x。图 2 也展示出完全二叉树具备堆的有序性。

直接给出代码

```
class BinaryHeap():
    def __init__(self):
        self.l=[]
    def up(self,i):
        while (i-1)//2>=0:
            parent = (i - 1) // 2
            if self.l[i]<self.l[parent]:
                self.l[i],self.l[parent]=self.l[parent],self.l[i]
            i=parent
    def insert(self,num):
        self.l.append(num)
        self.up(len(self.l)-1)
    def down(self,i):
        n=len(self.l)
        while 2*i+1<=n-1:
            smchild = self.sm_child(i)
            if self.l[i]>self.l[smchild]:
                self.l[i], self.l[smchild] = self.l[smchild], self.l[i]
            else:
                break
            i=smchild
    def sm_child(self,i):
        if 2*i+2 >= len(self.l):
            return 2*i+1
        if self.l[2*i+1]<self.l[2*i+2]:
            return 2*i+1
        else:
            return 2*i+2
    def delete(self):
        self.l[0],self.l[-1]=self.l[-1],self.l[0]
        result=self.l.pop()
        self.down(0)
        return result
    def heapify(self,List):
        self.l=List
        n=len(List)
        i=n//2-1
        while i>=0:
            self.down(i)
            i-=1
```

堆排序（在最大堆的前提下）

```
def heap_sort(lst):
    size = len(lst)
    build_heap(lst) #最大堆
    for i in range(0, size)[::-1]:
        lst[0], lst[i] = lst[i], lst[0]
        adjust_heap(lst, 0, i) #相当于down(self,i,size)
```

4.5.4 二叉搜索树

二叉搜索树依赖于这样一个性质：小于父节点的键都在左子树中，大于父节点的键则都在右子树中。我们称这个性质为二叉搜索性。

二叉搜索树的遍历http://cs101.openjudge.cn/practice/22275/

给出一棵二叉搜索树的前序遍历，求它的后序遍历

```
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]

n = int(input())
pre_order = list(map(int, input().split()))
print(' '.join(map(str, post_order(pre_order))))
```

构建二叉搜索树：

```
def build(l):
    root=None
    for val in l:
        insert(root,val)
    return root
def insert(root,val):
    if not root:
        return Node(val)
    if root.val>val:
        root.left=insect(root.left,val)
    if root.val<val:
        root.right=insert(root.right,val)
    return root
```

二叉搜索树实现快排：

1. 选择数组中的一个元素作为基准。
2. 创建一个空的二叉搜索树。
3. 将数组中的其他元素逐个插入二叉搜索树中。
4. 按照二叉搜索树的中序遍历（左子树、根节点、右子树）得到排序后的结果。

4.5.5 并查集

一种存储非重叠或不相交子集的数据结构被称为**不相交集合数据结构**（Disjoint Set Data Structure）。不相交集合数据结构支持以下操作：

- 向不相交集合中**添加新集合**。
- 使用**Union（合并）**操作将不相交集合合并为一个集合。
- 使用**Find（查找）**操作找到不相交集合的代表元素。
- 检查两个集合是否为不相交。

使用到的数据结构包括：

数组： 一个名为**Parent[]**的整数数组。如果我们处理的是**N**项，数组的第i个元素代表第i项。更具体地说，Parent[]数组的第i个元素是第i项的父节点。这些关系创建了一个或多个虚拟树。

树： 它是一个**不相交集**。如果两个元素位于同一棵树中，则它们位于同一个**不相交集**中。每棵树的根节点（或最顶部的节点）被称为该集合的**代表**。每个集合总是有一个唯一的代表。识别代表的一个简单规则是，如果'i'是某个集合的代表，则**Parent[i] = i**。如果'i'不是他所在集合的代表，则可以通过沿树向上遍历直到找到代表为止。

按秩合并：

```
class unionset():
    def __init__(self,n):
        self.rank=[1]*n
        self.parent=[i for i in range(n)]
    def find(self,i):
        if self.parent[i]!=i:
            self.parent[i]=self.find(self.parent[i])
        return self.parent[i]
    def union(self,x,y):
        rootx=self.find(x)
        rooty=self.find(y)
        if rootx==rooty:
            return
        if self.rank[rootx]<self.rank[rooty]:
            self.parent[rootx]=rooty
        if self.rank[rooty]<self.rank[rootx]:
            self.parent[rooty]=rootx
        else:
            self.parent[rooty] = rootx
            self.rank[rooty]+=1
```

还有按size合并，太过雷同故不列出

4.5.6 前缀树

字典树（前缀树，Trie）：字典树是一种树形数据结构，用于高效地存储和检索字符串数据集中的键。如果你使用嵌套的字典来表示字典树，其中每个字典代表一个节点，键表示路径上的字符，而值表示子节点，那么就构成了字典树。例如：

电话号码，http://cs101.openjudge.cn/practice/04089/

```
class Node():
    def __init__(self):
        self.child={}
class trie():
    def __init__(self):
        self.root=Node()
    def insert(self,num):
        curr=self.root
        for token in num:
            if token not in curr.child:
                curr.child[token]=Node()
            curr=curr.child[token]
    def find(self,num):
```

```
        curr=self.root
        for token in num:
            if token not in curr.child:
                return 0
            curr=curr.child[token]
        return 1
for _ in range(int(input())):
    s=0
    nums=[]
    for _ in range(int(input())):
        nums.append(input())
    nums.sort(reverse=True)
    t=trie()
    for num in nums:
        s+=t.find(num)
        t.insert(num)
    print(['NO','YES'][s==0])
```