常用结构与方法

1.并查集:

列表实现

```
def find(x):
    if p[x]!=x:
        p[x]=find(p[x])
    return p[x]

def union(x,y):
    rootx,rooty=find(x),find(y)
    if rootx!=rooty:
        p[rootx]=p[rooty]

#单纯并查
p=list(range(n+1))
unique_parents = set(find(x) for x in range(1, n + 1))  最后收取数据

#反向事件  用x+n储存x的反向事件，查询时直接find（x+n）
p=list(range(2*(n+1))
if tag=="Different":
    union(x,y+n)
    union(y,x+n)
```

集合实现:

```
class UnionFind:
    def __init__(self,n):
        self.p=list(range(n))
        self.h=[0]*n
    def find(self,x):
        if self.p[x]!=x:
            self.p[x]=self.find(self.p[x])
        return self.p[x]
    def union(self,x,y):
        rootx=self.find(x)
        rooty=self.find(y)
        if rootx!=rooty:
            if self.h[rootx]<self.h[rooty]:
                self.p[rootx]=rooty
            elif self.h[rootx]>self.h[rooty]:
                self.p[rooty]=rootx
            else:
                self.p[rooty]=rootx
                self.h[rootx]+=1
```

2.堆:

```
from heapq import *

heappush(heap, item)
heappop(heap) 弹出最小，可接收
heap[0] 访问最小，可接收
heapify(lst) 建堆
heapreplace(heap, item)
heappushpop(heap, item)
```

默认为最小堆，转化为最大堆方法：全部化为负数，输出时再次加负号

3.双端队列：

```
from collections import deque
q=deque()

q.appendleft()
q.append()
q.popleft()
q.pop()
```

4.二分查找：

```
import bisect
sorted_list = [1,3,5,7,9]
position = bisect.bisect_left(sorted_list, 6) #查找元素应左插入的位置
print(position)  # 输出：3，因为6应该插入到位置3，才能保持列表的升序顺序

bisect.insort_left(sorted_list, 6) #左插入元素
print(sorted_list)  # 输出：[1, 3, 5, 6, 7, 9]，6被插入到适当的位置以保持升序顺序

sorted_list=(1,3,5,7,7,7,9)
print(bisect.bisect_left(sorted_list,7))
print(bisect.bisect_right(sorted_list,7))
```

5.扩栈：

```
import sys
sys.setrecursionlimit(1<<30)
```

counter：计数

```python
from collections import Counter
# 创建一个Counter对象
count = Counter(['apple', 'banana', 'apple', 'orange', 'banana', 'apple'])
# 输出Counter对象
print(count)  # 输出: Counter({'apple': 3, 'banana': 2, 'orange': 1})
# 访问单个元素的计数
print(count['apple'])  # 输出: 3
# 访问不存在的元素返回0
print(count['grape'])  # 输出: 0
# 添加元素
count.update(['grape', 'apple'])
print(count)  # 输出: Counter({'apple': 4, 'banana': 2, 'orange': 1, 'grape': 1})
```

permutations：全排列

```python
from itertools import permutations
# 创建一个可迭代对象的排列
perm = permutations([1, 2, 3])
# 打印所有排列
for p in perm:
    print(p)
# 输出: (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)
```

combinations：组合

```python
from itertools import combinations
# 创建一个可迭代对象的组合
comb = combinations([1, 2, 3], 2)
# 打印所有组合
for c in comb:
    print(c)
# 输出: (1, 2), (1, 3), (2, 3)
```

reduce：累次运算

```python
from functools import reduce
# 使用reduce计算列表元素的乘积
product = reduce(lambda x, y: x * y, [1, 2, 3, 4])
print(product)  # 输出: 24
```

product：笛卡尔积

```python
from itertools import product
# 创建两个可迭代对象的笛卡尔积
prod = product([1, 2], ['a', 'b'])
# 打印所有笛卡尔积对
for p in prod:
    print(p)
# 输出: (1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')
```

Stack模板：

1.单调栈：

```
leftb=[-1]*n
rightb=[n for i in range(n)]
stack=[]
for i in range(n):
    while stack and heights[stack[-1]]<heights[i]:
        stack.pop()
    if stack:
        leftb[i]=stack[-1]
    stack.append(i)

stack=[]
for i in range(n-1,-1,-1):
    while stack and  heights[stack[-1]]>heights[i]:
        stack.pop()
    if stack:
        rightb[i]=stack[-1]
    stack.append(i)
```

2.中序转后续表达式：

```
def infixToPostfix(row):
    precedence={"+":1,"-":1,"*":2,"/":2}
    stack=[];ans=[];number=""
    for i in row:
        if i.isnumeric() or i==".":
            number+=i
        else:
            if number:
                nownum=number
                number=""
                ans.append(nownum)
            if i=="(":
                stack.append(i)
            elif i in "+-*/":
                while stack and stack[-1] in "+-*/" and precedence[i]
<=precedence[stack[-1]]:
                    ans.append(stack.pop())
                stack.append(i)
            elif i==")":
                while stack and stack[-1]!="(":
                    ans.append(stack.pop())
                stack.pop()
    if number:
        ans.append(number)
    while stack:
        ans.append(stack.pop())
    return (" ".join(ans))
```

3.逆波兰表达式求值：

```
stack=[]
for t in s:
    if t in '+-*/':
        b,a=stack.pop(),stack.pop()
        stack.append(str(eval(a+t+b)))
    else:
        stack.append(t)
print(f'{float(stack[0]):.6f}')
```

最大全0子矩阵:

```
for row in ma:
    stack=[]
    for i in range(n):
        h[i]=h[i]+1 if row[i]==0 else 0
        while stack and h[stack[-1]]>h[i]:
            y=h[stack.pop()]
            w=i if not stack else i-stack[-1]-1
            ans=max(ans,y*w)
        stack.append(i)
    while stack:
        y=h[stack.pop()]
        w=n if not stack else n-stack[-1]-1
        ans=max(ans,y*w)
print(ans)
```

求逆序对数:

```
from bisect import *
a=[]
rev=0
for _ in range(n):
    num=int(input())
    rev+=bisect_left(a,num)
    insort_left(a,num)
ans=n*(n-1)//2-rev
```

```
def merge_sort(a):
    if len(a)<=1:
        return a,0
    mid=len(a)//2
    l,l_cnt=merge_sort(a[:mid])
    r,r_cnt=merge_sort(a[mid:])
    merged,merge_cnt=merge(l,r)
    return merged,l_cnt+r_cnt+merge_cnt
def merge(l,r):
    merged=[]
    l_idx,r_idx=0,0
    inverse_cnt=0
    while l_idx<len(l) and r_idx<len(r):
        if l[l_idx]<=r[r_idx]:
            merged.append(l[l_idx])
```

```
            l_idx+=1
        else:
            merged.append(r[r_idx])
            r_idx+=1
            inverse_cnt+=len(l)-l_idx
    merged.extend(l[l_idx:])
    merged.extend(r[r_idx:])
    return merged,inverse_cnt
```

Tree模板

1.二叉树深度：

```
def tree_depth(node):
    if node is None:
        return 0
    left_depth = tree_depth(node.left)
    right_depth = tree_depth(node.right)
    return max(left_depth, right_depth) + 1
```

2.二叉树的读取与建立：

输入为每个节点的子节点：

```
nodes = [TreeNode() for _ in range(n)]

for i in range(n):
    left_index, right_index = map(int, input().split())
    if left_index != -1:
        nodes[i].left = nodes[left_index]
    if right_index != -1:
        nodes[i].right = nodes[right_index]
```

但要注意，这里的index随题目要求而改变，即从0开始还是从1开始的问题，可能要-1

括号嵌套树的解析建立：

```
def parse_tree(s):
    stack = []
    node = None
    for char in s:
        if char.isalpha():  # 如果是字母，创建新节点
            node = TreeNode(char)
            if stack:  # 如果栈不为空，把节点作为子节点加入到栈顶节点的子节点列表中
                stack[-1].children.append(node)
        elif char == '(':  # 遇到左括号，当前节点可能会有子节点
            if node:
                stack.append(node)  # 把当前节点推入栈中
```

```
                node = None
            elif char == ')':  # 遇到右括号，子节点列表结束
                if stack:
                    node = stack.pop()  # 弹出当前节点
    return node  # 根节点
```

根据前中序得后序，根据中后序得前序：

```
def postorder(preorder,inorder):
    if not preorder:
        return ''
    root=preorder[0]
    idx=inorder.index(root)
    left=postorder(preorder[1:idx+1],inorder[:idx])
    right=postorder(preorder[idx+1:],inorder[idx+1:])
    return left+right+root
```

```
def preorder(inorder,postorder):
    if not inorder:
        return ''
    root=postorder[-1]
    idx=inorder.index(root)
    left=preorder(inorder[:idx],postorder[:idx])
    right=preorder(inorder[idx+1:],postorder[idx:-1])
    return root+left+right
```

3.二叉树叶节点计数：

```
def count_leaves(node):
    if node is None:
        return 0
    if node.left is None and node.right is None:
        return 1
    return count_leaves(node.left)+count_leaves(node.right)
```

4.树的遍历：

前/后序遍历：

```
def preorder(node):
    output = [node.value]
    for child in node.children:
        output.extend(preorder(child))
    return ''.join(output)

def preorder(node):
    if node is not None:
        return tree.value+preorder(tree.left)+preorder(tree.right)
    else:
        return ""
```

```
def postorder(node):
    output = []
    for child in node.children:
        output.extend(postorder(child))
    output.append(node.value)
    return ''.join(output)

def postorder(node):
    if node is not None:
        return postorder(node.left)+postorder(node.right)+node.value
    else:
        return ""
```

中序遍历：

```
def inorder(tree):
    if tree is not None:
        return inorder(tree.left)+tree.value+inorder(tree.right)
    else:
        return ""
```

分层遍历：（使用bfs）

```
from collections import deque

class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []

def level_Order(root):
    queue = deque()
    queue.append(root)

    while len(queue) != 0: # 这里是一个特殊的BFS,以层为单位
        n = len(queue)
        while n > 0: #一层层的输出结果
            point = queue.popleft()
            print(point.value, end=" ") # 这里的输出是一行
            queue.extend(point.children)
            n -= 1

        print()   #要加上 end的特殊语法
```

```
from collections import deque
def levelorder(root):
    if not root:
        return ""
    q=deque([root])
    res=""
    while q:
```

```
        node=q.popleft()
        res+=node.val
        if node.left:
            q.append(node.left)
        if node.right:
            q.append(node.right)
    return res
```

5.二叉搜索树的构建：

```
def insert(root,num):
    if not root:
        return Node(num)
    if num<root.val:
        root.left=insert(root.left,num)
    else:
        root.right=insert(root.right,num)
    return root
```

6.字典树的构建：

```
def insert(root,num):
    node=root
    for digit in num:
        if digit not in node.children:
            node.children[digit]=TrieNode()
        node=node.children[digit]
        node.cnt+=1
```

Graph模版

1.Dijikstra：

```
#用字典储存路径
ways=dict()
for _ in range(p):
    ways[input()]=[]
q=int(input())
for i in range(q):
    FRM,TO,CST=input().split()
    ways[FRM].append((TO,int(CST)))
    ways[TO].append((FRM,int(CST)))

#函数主体(带路径的实现)
from heapq import *
def dijkstra(frm,to):
    q=[]
    tim=0
    heappush(q,(tim,frm,[frm]))
```

```
    visited=set([frm])
    if frm==to:
        return frm,0
    while q:
        tim,x,how=heappop(q)
        if x==to:
            return "->".join(how),tim
        visited.add(x)
        for way in ways[x]:
            nx=way[0];cst=way[1]
            if nx not in visited:
                nhow=how.copy()
                nhow.append(f"({cst})")
                nhow.append(nx)
                heappush(q,(tim+cst,nx,nhow))
    return "NO"
```

注意visited是为了在无向图中防止返回，有向图不需要visited

2.BFS:

```
def bfs(graph, initial):
    visited = set()
    queue = [(initial,tim)]

    while queue:
        node = queue.pop(0)
        if node not in visited:
            visited.add(node)
            neighbours = graph[node]
            nt=tim

            for neighbour in neighbours:
                cst=costs[neighbour]
                queue.append((neighbour,cst+tim)
```

3.DFS:

```
def dfs(v):
    visited.add(v)
    total = values[v]  #以最大权值联通块为例
    for w in graph[v]:
        if w not in visited:
            total += dfs(w)
    return total
```

八皇后:

```
ans = []
def queen_dfs(A, cur=0):           #考虑放第cur行的皇后
    if cur == len(A):              #如果已经放了n个皇后，一组新的解产生了
        ans.append(''.join([str(x+1) for x in A])) #注意避免浅拷贝
```

```
        return

    for col in range(len(A)):      #将当前皇后逐一放置在不同的列，每列对应一组解
        for row in range(cur):     #逐一判定，与前面的皇后是否冲突
            #因为预先确定所有皇后一定不在同一行，所以只需要检查是否同列，或者在同一斜线上
            if A[row] == col or abs(col - A[row]) == cur - row:
                break
        else:                      #若都不冲突
            A[cur] = col           #放置新皇后，在cur行，col列
            queen_dfs(A, cur+1)    #对下一个皇后位置进行递归

queen_dfs([None]*8)
for _ in range(int(input())):
    print(ans[int(input()) - 1])
```

4.Prim：

用途：在N**2时间内实现最小生成树。

```
from heapq import *

def prim(graph, n):
    vis = [False] * n
    mh = [(0, 0)]  # (weight, vertex)
    mc = 0

    while mh:
        wei, ver = heappop(mh)

        if vis[vertex]:
            continue
        vis[ver] = True

        mc += wei
        for nei, nw in graph[ver]:
            if not vis[nei]:
                heappush(mh, (nw, nei))

    return mc if all(visited) else -1

def main():
    n, m = map(int, input().split())
    graph = [[] for _ in range(n)]

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

    mc = prim(graph, n)
    print(mc)

if __name__ == "__main__":
    main()
```

特殊函数：

字符串类：

1.前后缀判定：

```
if str1.startwith(str2):
if str1.endwith(str2):
```

2.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

```
str.find(sub)
```

3.判定类型：

```
str.isupper()#是否全为大写
str.islower()#是否全为小写
str.isdigit()#是否全为数字
str.isnumeric()#是否为整数
str.isalnum()#是否全为字母或汉字或数字
str.isalpha()#是否全为字母或汉字
```

4.将字符串中的`old`子字符串替换为`new`

```
str.replace(old, new)
```

5.移除字符串左侧/右侧的空白字符：

```
str.lstrip() / str.rstrip()
```

列表类：

2.列表计算元素出现次数：