数算cheet sheet

1.一些杂的东西

1.1Counter

```
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('1213123343521231255555555')
cc=sorted(c.items(),key=lambda x:x[1],reverse=True)
#[('5', 9), ('1', 5), ('2', 5), ('3', 5), ('4', 1)]
```

1.2 cmp_to_key

```
from functools import cmp_to_key
def compar(a,b):
    return a>b
l=[1,5,2,4,6,7,6]
l.sort(key=cmp_to_key(compar))
print(l)#[1,2,4,5,6,6,7]
```

1.3 保留小数

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

1.4 全排列

```
from itertools import permutations
perm = permutations([1, 2, 3])
perm2 = permutations([1, 2, 3], 2)
for i in list(perm):
    print (i)
```

1.5 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]: #bisect_left 改为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)
```

1.6 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
```

1.7 十大排序

1.7.1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
```

1.7.2 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)))
```

1.7.3 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)
```

1.7.4 归并排序

```
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
```

```python
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1
        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

1.7.5 插入排序

```python
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
```

1.7.6 希尔排序

```python
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
```

1.8 递归优化

1.增加递归深度限制

```python
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

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

2.基本数据结构

2.1 栈

2.1.1匹配括号

```
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('{{}}[]]'))
```

2.1.2 中序、前序和后序表达式(含调度场算法)

中序转后序：调度场算法

```
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=' ')
```

2.1.2 后续表达式求值

```
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}")
```

2.2 链表

2.2.1 单向链表

篇幅受限只给出单向链表实现

```
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:
```

```python
                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()
```

2.2.2 链表反转

```python
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
```

2.2.3 合并链表

```python
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
```

2.2.4 查找链表中间节点

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

3.树

3.1 二叉树及其递归算法

3.1.1 遍历

以中序遍历为例

```
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)
```

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
```

层序遍历

```
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
```

3.1.2 求树的高度/深度和叶子数量（含建树过程）

```
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))
```

3.1.3 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))
```

3.1.4 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
```

3.1.5 寻找二叉搜索树中第K小的元素

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

3.1.6 判断是否为平衡二叉树

```
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
```

3.4 树的表示方法(各种奇怪的建树题)

3.4.1 嵌套括号表示法

```
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
```

3.4.2 拓展二叉树

将二叉树的空结点用补齐，先序和后序序列能唯一确定其二叉树，如ABD..EF..G..C..

```
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
```

3.4.3 邻接表示法

括号转邻接

```
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
```

3.4.4 前中序建树

```python
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
```

3.5 各种奇怪的树

3.5.1 解析树

中序建树:

```python
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
```

后序建树:

```python
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]
```

计算方法:

```python
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()
```

编码树转中序只需中序遍历加括号

3.5.2 霍夫曼编码Huffman 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。

```python
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
```

3.5.3 二叉堆

```
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)
```

3.5.4 二叉搜索树

前序求后序

```
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。

3.5.5前缀树

```
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
```

4.图

4.1 图的表示方法

4.1.1 邻接矩阵

```
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
```

4.1.2 邻接表

```
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)
```

4.2 基本图算法

宽度优先搜索（以词梯为例）

```
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')
```

深度优先搜索（以骑士周游为例）

```
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))])
```

4.3 拓展图算法

4.3.1 拓扑排序

1.DFS:

（1）创建多棵深度优先搜索树并给出起始时间和终止时间。

```
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])
```

(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。

缺点：无法识别有环图

2.2.Karn算法 / BFS： O(V + E)

```
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]) #判断有向环
```

4.3.2 最短路径

1.dijkstra算法（只适用于边的权重均为正的情况）$O((V+E)\log V)$

```
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.多源最短路径Floyd-Warshall算法（用邻接矩阵表示）

`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

```
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))
```

4.3.3 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。

```python
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')
```

4.3.4最小生成树：

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O(|E| log |V|)

```
from heapq import heappush,heapify,heappop
visit=[False]*n
visit[0]=True
h=graph[0] #w,vert
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))
```

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

```
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)
```