cheatsheet-机考

基本语法

OJ特有的

global避免检查

第一行加# pylint: skip-file可以忽略检查

字符串：

```
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
```

列表：

```
“（连接符）”.join(list)

删除已知元素：list.remove(元素)

删除已知索引的元素：pop(idx)

倒序排序：list.sort(reverse=True)

指定顺序排序：list.sort(key= lambda s:排序指标（与s相关）

寻找索引：list.index(元素) 第一个元素，没有会触发ValueError

**enumerate()函数**（遍历方法：for index, A in enumerate(列表)）
```

字典：

```
{}  dict(元组)  半有序：Ordereddict()

遍历字典的键：for 元素 in dict() ； for 元素 in dict.keys()

遍历字典的值：for 元素 in dict.values()

删除键值对：del dict[键]

遍历键值对：for key,value in dict.items():
```

元组：

```
建立：(...,...,...,...)  含元组的列表：zip(a,b,c,...)
```

集合：

```
建立：set()

set.add()一个  set.update()多个

删除元素：set.remove() 或set.discard()（前者有KeyError风险，后者没有）

随机删除：set.pop()
```

各种库

```
import heapq：heapq.heappop 、heapq.heappush、heapq.heapify()

from collections import deque: ......=deque()，.......pooleft()，.......append()

全排列生成库
```

多值排序语句

```
sorted_students = sorted(students.items(), key=lambda x: (x[1], x[0]))
.sort()是浅拷贝
sorted()是深拷贝
```

浮点数小数位的保留

```
judge=48.00
print(f"{judge:.2f}")
```

栈

中缀表达式

$$A + B * (C - D) - E/F$$

后缀表达式

$$ABCD - * + EF/-$$。

shunting yard（中序转后序，要定符号优先级）

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

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

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

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

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

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

括号嵌套树

不好用递归建树，建议用栈。

括号匹配其实是栈的应用，所以这道题事实上是栈与树的结合。

坑点：单个字母

代码：

```python
class tree:
    def __init__(self,name):
        self.name=name
        self.children=[]

sample=list(input())
def buildtree(sample):
    stack=[]
    while sample:
        tempo=sample.pop(0)
        if tempo.isalpha():
            newtree = tree(tempo)

            if stack:
                stack[-1].children.append(newtree)
        elif tempo=="(":
            stack.append(newtree)
        elif tempo==")":
            ulta=stack.pop()
    return ulta

def preorder(root):
    stack=[]
    stack.append(root.name)
    if root.children:
        for i in root.children:
            stack.extend(preorder(i))
    return stack

def postorder(root):
    stack=[]
    if root.children:
        for i in root.children:
            stack.extend(postorder(i))
    stack.append(root.name)
    return stack
```

单调栈（快速堆猪）

```
a = []
m = []

while True:
    try:
        s = input().split()
        if s[0] == "pop":
            if a:
                a.pop()
                if m:
                    m.pop()
        elif s[0] == "min":
            if m:
                print(m[-1])
        else:
            h = int(s[1])
            a.append(h)
            if not m:
                m.append(h)
            else:
                k = m[-1]
                m.append(min(k, h))
    except EOFError:
        break
```

单调栈模板

时间复杂度O(n)

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

ans=[0]*n
stack=[]

for idx in range(n):
    if not stack or sample[idx]<=sample[stack[-1]]:
        stack.append(idx)
    else:
        while stack and sample[stack[-1]]<sample[idx]:
            ans[stack.pop()]=idx+1
        stack.append(idx)
print(*ans)
```

队列

约瑟夫问题2.0:

大胆用insert，pop就行，注意人数变化以及弹出位置的变化

```
flag=1

while flag==1:
    pp = []
    n,p,m=(int(x) for x in input().split())
    if n==0:
        flag=0
```

```
            break
        else:
            alist=[]
            alist=list(range(1,n+1))
            while len(alist)>=1:
                if len(alist) >= p:
                    for u in range(1,m):
                        alist.insert(len(alist)-1,alist.pop(0))
                    pp.append(alist.pop(p-1))
                else:
                    for u in range(1,m+1):
                        alist.insert(len(alist) - 1, alist.pop(0))
                    pp.append(alist.pop())

            print(",".join(map(str,pp)))
```

BFS（队列的运用）

词梯

一种比较新颖的bfs

利用了桶的思想

```
from collections import defaultdict
dic=defaultdict(list)
n,lis=int(input()),[]
for i in range(n):
    lis.append(input())
for word in lis:
    for i in range(len(word)):
        bucket=word[:i]+'_'+word[i+1:]
        dic[bucket].append(word)
def bfs(start,end,dic):
    queue=[(start,[start])]
    visited=[start]
    while queue:
        currentword,currentpath=queue.pop(0)
        if currentword==end:
            return ' '.join(currentpath)
        for i in range(len(currentword)):
            bucket=currentword[:i]+'_'+currentword[i+1:]
            for nbr in dic[bucket]:
                if nbr not in visited:
                    visited.append(nbr)
                    newpath=currentpath+[nbr]
                    queue.append((nbr,newpath))
    return 'NO'
```

pots

```
def bfs(A, B, C):
    start = (0, 0)
    visited = set()
    visited.add(start)
    queue = [(start, [])]

    while queue:
        (a, b), actions = queue.pop(0)

        if a == C or b == C:
            return actions

        next_states = [(A, b), (a, B), (0, b), (a, 0), (min(a + b, A),\
                max(0, a + b - A)), (max(0, a + b - B), min(a + b, B))]

        for i in next_states:
            if i not in visited:
                visited.add(i)
                new_actions = actions + [get_action(a, b, i)]
                queue.append((i, new_actions))

    return ["impossible"]


def get_action(a, b, next_state):
    if next_state == (A, b):
        return "FILL(1)"
    elif next_state == (a, B):
        return "FILL(2)"
    elif next_state == (0, b):
        return "DROP(1)"
    elif next_state == (a, 0):
        return "DROP(2)"
    elif next_state == (min(a + b, A), max(0, a + b - A)):
        return "POUR(2,1)"
    else:
        return "POUR(1,2)"


A, B, C = map(int, input().split())
solution = bfs(A, B, C)

if solution == ["impossible"]:
    print(solution[0])
else:
    print(len(solution))
    for i in solution:
        print(i)
```

DFS（通常是一种递归）

正常版本

```
graph=[]
for i in range(10):
    line=list(input())
    graph.append(line)
cnt=0
dire=[[-1,0],[1,0],[0,1],[0,-1]]
def dfs(ini_x,ini_y):
    graph[ini_x][ini_y]="-"
    for i in dire:
        newx,newy=ini_x+i[0],ini_y+i[1]
        if 0<=newx<10 and 0<=newy<10 and graph[newx][newy]==".":
            dfs(newx,newy)

for line in range(10):
    for row in range(10):
        if graph[line][row]==".":
            dfs(line,row)
            cnt+=1
```

八皇后与回溯

```
condition=[None for  i in range(8)]#储存+1后的版本
def issafe(condition,col,row):#棋盘状况,真是列（1~8）,当前行
    #先检查同列
    for i in range(8):
        if condition[i]==col:
            return False
    #检查左上角
    i=col-1#新列
    j=row-1#新行
    while 1<=i<9 and 0<=j<8:
        if condition[j]==i:
            return False
        i-=1
        j-=1
    #检查右上角
    i=col+1#新列
    j=row-1#新行
    while 1 <= i < 9 and 0 <= j < 8:
        if condition[j] == i:
            return False
        i += 1
        j -= 1
    return True#确认无误就可以放下
```

```
ans=[]
def queen(condition,row):#目前填的行数
    if row==7 :
        for m in range(1,9):
            if issafe(condition,m,7):
                condition[7]=m
                ans.append("".join(map(str,condition)))
                condition[7]=None
                break
    elif row<7:
        for ii in range(1,9):
            if issafe(condition,ii,row):
                condition[row]=ii
                queen(condition,row+1)#递归下一行
            #回溯
                condition[row]=None
n=int(input())
queen(condition,0)
ans.sort()
for p in range(n):
    g=int(input())-1
    print(ans[g])
```

树

n个结点组成的二叉树形态数目：B[n] = C[n,2n] / (n+1)

交换操作，命名了4个类，包括左父亲与右父亲

前序遍历（等价于先序遍历）：中左右

后序遍历：左右中

中序遍历：中左右

层次遍历：用队列即可

求树的高度与叶子数目

```
class tree():
    def __init__(self):
        self.left=None
        self.right=None

n=int(input())#结点个数0~n-1
arr=[tree() for _ in range(n)]#建立树
yezi=0
for i in range(n):
    x,y=map(int,input().split())
    arr[i].left=x
    arr[i].right=y
    if x==-1 and y==-1:
        yezi+=1
```

```
def count(tree):
    if tree.left!=-1 and tree.right!=-1:
        return 1+max(count(arr[tree.left]),count(arr[tree.right]))
    elif tree.left==-1 and tree.right!=-1:
        return 1+count(arr[tree.right])
    elif tree.left!=-1 and tree.right==-1:
        return 1+count(arr[tree.left])
    else:
        return 1
cnt=count(arr[0])
for m in arr[1::]:
    p=count(m)#记录叶子数
    if p>cnt:
        cnt=p
print(cnt-1,yezi)
```

面对树形转换问题，头脑清醒，确认递归终止条件即可。注意观察和debug

```
def buildleft(root):
    new=root.children[0]
    for i in range(1,len(root.children)):
        new.right=root.children[i]
        new=root.children[i]
    return root.children[0]#如何把根节点删掉

def construct(root):
    if root.children:
        root.left=buildleft(root)
        for m in root.children:
            construct(m)
    return root
```

trie数据结构（电话号码-相同前缀）

```
class trienode:
    def __init__(self):
        self .child={}
class trie:
    def __init__(self):
        self.root=trienode()
    def insert(self,str):#插入一段电话号码
        root1=self.root
        for i in str:#电话号码中的每个数字
            if i not in root1.child:
                root1.child[i]=trienode()#字典的value值对应一个新字典
            root1=root1.child[i]
    def search(self,num):#num是目标电话号码
        root2=self.root
        for i in num:
            if i not in root2.child:
                return 0
            root2=root2.child[i]
        return 1
```

```python
t=int(input())

for g in range(t):
    datas=[]
    n=int(input())
    for m in range(n):
        tempo=input()
        datas.append(tempo)
    trie1=trie()
    datas.sort(key=lambda x :-len(x))#?
    cnt=0
    for g in datas:
        cnt+=trie1.search(g)
        trie1.insert(g)
    if cnt!=0:
        print("NO")
    else:
        print("YES")
```

图

无向图的联通与有环判断

dfs版本:

```python
n,m=map(int,input().split())
lis=[[] for i in range(n)]
flag=1
for i in range(m):
    a, b = map(int, input().split())
    lis[a].append(b)
    lis[b].append(a)
vis=set()
def dfs(x,pre):
    global cnt,flag
    vis.add(x)
    for i in lis[x]:
        if i not in vis:
            dfs(i,x)
        elif i in vis and i!=pre:
            flag=0
dfs(0,None)
if len(vis)==n:
    print("connected:yes")
else:
    print("connected:no")
if flag==0:
    print("loop:yes")
else:
    print("loop:no")
```

并查集版本:

```
class unionandfind:
    def __init__(self,n):
        self.fathers=[i for i in range(n)]
    def find(self,a):
        if self.fathers[a]!=a:
            self.fathers[a]=self.find(self.fathers[a])
        return self.fathers[a]
    def union(self,a,b):
        a_fa=self.find(a)
        b_fa=self.find(b)
        if a_fa!=b_fa:
            self.fathers[a_fa]=b_fa
            return False
        else:
            return True

n,m=map(int,input().split())
uf=unionandfind(n)

flag=1
for u in range(m):
    a,b=map(int,input().split())
    if  uf.union(a,b):
        flag=0

uf.fathers=[uf.find(i) for i in uf.fathers]
uf.fathers=set(uf.fathers)
if len(uf.fathers)>1:
    print("connected:no")
else:
    print("connected:yes")
if flag==1:
    print("loop:no")
else:
    print("loop:yes")
```

有向图的联通与成环判断

联通靠dfs or bfs， 成环靠拓扑排序

拓扑排序

```
def judge(info,indgree,N):#info 字典 储藏邻接表信息，indgree 列表，储藏入度信息
    flag=1
    vis=set()
    while flag==1:
        flag=0
        for t in range(N):
            if indgree[t]==0:
                vis.add(t)
                flag=1
                for m in info[t]:
                    indgree[m]-=1
                indgree[t]=-1
    return len(vis)

T=int(input())
for i in range(T):
    N,M=map(int,input().split())
    info={v:[] for v in range(N) }
    indgree=[0]*N
    for c in range(M):
        x,y=map(int,input().split())
        x,y=x-1,y-1
        indgree[y]+=1
        info[x].append(y)
    new=judge(info,indgree,N)
    if new<N:
        print("Yes")
    else:
        print("No")
```

最小生成树

极小连通分量

例子：等价于破坏环中最大的那条边（瞎破坏）

krustal 算法

需要的库：import heapq

tips:

heapq.heapify会有一定的时间复杂度，次数越少越好

heapify的时间复杂度O（n）

heappush、heappop时间复杂度O（logn）

示例：团结不用排序就是力量

```
import heapq
n,m=map(int,input().split())
class unionandfind:
    def __init__(self,n):
        self.fathers=[i  for i in range(n)]
    def find(self,a):
        if self.fathers[a]!=a:
            self.fathers[a]=self.find(self.fathers[a])
        return self.fathers[a]
    def union(self,a,b):
        a_fa=self.find(a)
        b_fa=self.find(b)
        if a_fa==b_fa:
            return False
        else:
            self.fathers[a_fa]=b_fa
            return True
edges=[]
for i in range(m):
    a,b,c=map(float,input().split())
    edges.append((c,int(a),int(b)))
uf=unionandfind(n)
def krustal(edges):
    vis=set()
    spend=0
    ans=[]
    heapq.heapify(edges)
    while edges:
        tempo=heapq.heappop(edges)
        spot1=tempo[1]
        spot2=tempo[2]
        money=tempo[0]
        if uf.union(spot1,spot2):
            spend+=money

            newans=[spot1,spot2]
            newans.sort()
            ans.append(newans)
            panding=[uf.find(x) for x in range(n)]
            panding=set(panding)
            if len(panding)==1:
                return spend,ans
    return -1,None

judge,final=krustal(edges)
if judge==-1:
    print("NOT CONNECTED")
else:
    print(f"{judge:.2f}")

    final.sort(key=lambda x: (x[0] ,x[1] ))
    for i in final:
        print(*i)
```

prim算法

```
from collections import defaultdict
from heapq import *

def prim(vertexs, edges,start='D'):
    adjacent_dict = defaultdict(list) # 注意：defaultdict(list)必须以list做为变量
    for weight,v1, v2 in edges:
        adjacent_dict[v1].append((weight, v1, v2))
        adjacent_dict[v2].append((weight, v2, v1))
    '''
    经过上述操作，将图转化为以下邻接表形式：
    {'A': [(7, 'A', 'B'), (5, 'A', 'D')],
     'C': [(8, 'C', 'B'), (5, 'C', 'E')],
     'B': [(7, 'B', 'A'), (8, 'B', 'C'), (9, 'B', 'D'), (7, 'B', 'E')],
     'E': [(7, 'E', 'B'), (5, 'E', 'C'), (15, 'E', 'D'), (8, 'E', 'F'), (9, 'E',
'G')],
     'D': [(5, 'D', 'A'), (9, 'D', 'B'), (15, 'D', 'E'), (6, 'D', 'F')],
     'G': [(9, 'G', 'E'), (11, 'G', 'F')],
     'F': [(6, 'F', 'D'), (8, 'F', 'E'), (11, 'F', 'G')]})
    '''
    minu_tree = []  # 存储最小生成树结果
    visited = [start] # 存储访问过的顶点，注意指定起始点
    adjacent_vertexs_edges = adjacent_dict[start]
    heapify(adjacent_vertexs_edges) # 转化为小顶堆，便于找到权重最小的边
    while adjacent_vertexs_edges:
        weight, v1, v2 = heappop(adjacent_vertexs_edges) # 权重最小的边，并同时从堆中
删除。
        if v2 not in visited:
            visited.append(v2)  # 在used中有第一选定的点'A'，上面得到了距离A点最近的
点'D',举例是5。将'd'追加到used中
            minu_tree.append((weight, v1, v2))
            # 再找与d相邻的点，如果没有在heap中，则应用heappush压入堆内，以加入排序行列
            for next_edge in adjacent_dict[v2]: # 找到v2相邻的边
                if next_edge[2] not in visited: # 如果v2还未被访问过，就加入堆中
                    heappush(adjacent_vertexs_edges, next_edge)
    return minu_tree
```

迪杰斯特拉：找有权图的最短路径

oop类型（记录路径+class类型）

```
import heapq

def dijkstra(adjacency, start):
    distances = {vertex: float('infinity') for vertex in adjacency}
    previous = {vertex: None for vertex in adjacency}
    distances[start] = 0
    pq = [(0, start)]

    while pq:
        current_distance, current_vertex = heapq.heappop(pq)
        if current_distance > distances[current_vertex]:
            continue

        for neighbor, weight in adjacency[current_vertex].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                previous[neighbor] = current_vertex
                heapq.heappush(pq, (distance, neighbor))

    return distances, previous

def shortest_path_to(adjacency, start, end):
    distances, previous = dijkstra(adjacency, start)
    path = []
    current = end
    while previous[current] is not None:
        path.insert(0, current)
        current = previous[current]
    path.insert(0, start)
    return path, distances[end]

# Read the input data
P = int(input())
places = {input().strip() for _ in range(P)}

Q = int(input())
graph = {place: {} for place in places}
for _ in range(Q):
    src, dest, dist = input().split()
    dist = int(dist)
    graph[src][dest] = dist
    graph[dest][src] = dist  # Assuming the graph is bidirectional

R = int(input())
requests = [input().split() for _ in range(R)]

# Process each request
for start, end in requests:
    if start == end:
        print(start)
        continue

    path, total_dist = shortest_path_to(graph, start, end)
    output = ""
    for i in range(len(path) - 1):
        output += f"{path[i]}->({graph[path[i]][path[i+1]]})->"
    output += f"{end}"
    print(output)
```

两点多条路径的迪杰斯特拉（小心字典的覆盖，邻接表类型）

```
import heapq
n,m=map(int,input().split())
dormeat=[int(x) for x in input().split()]
dorm={i: [2001]*n for i in range(n)}
for i in range(m):
    u,v,w=map(int,input().split())
    u,v=u-1,v-1
    if 2*w<dorm[u][v]:
        dorm[u][v]=2*w
        dorm[v][u]=2*w
def bfs(a):
    global n
    ans=[]
    queue=[(0,a)]
    vis=set()
    while queue:
        if len(vis)==n:
            break
        tempo=heapq.heappop(queue)
        newdorm=tempo[1]
        newdis=tempo[0]
        ans.append(newdis+dormeat[newdorm])
        vis.add(newdorm)
        for i in range(n):  # 宿舍楼号遍历
            r = dorm[newdorm][i]
            if r != 2001 and i not in vis:
                dis1 = newdis + r
                heapq.heappush(queue, (dis1, i))

    ans.sort()
    return ans[0]
for u in range(n):
    print(bfs(u))
```

判断有向图的强连通

判断有向图是否强连通： Kosaraju's 算法函数

核心：两次dfs

作用：用于查找有向图中强连通分量的算法。强连通分量是指在有向图中，任意两个节点都可以相互到达的一组节点。

```
def dfs1(graph, node, visited, stack):
    # 第一个深度优先搜索函数，用于遍历图并将节点按完成时间压入栈中
    visited[node] = True  # 标记当前节点为已访问
    for neighbor in graph[node]:  # 遍历当前节点的邻居节点
        if not visited[neighbor]:  # 如果邻居节点未被访问过
            dfs1(graph, neighbor, visited, stack)  # 递归调用深度优先搜索函数
    stack.append(node)  # 将当前节点压入栈中，记录完成时间

def dfs2(graph, node, visited, component):
    # 第二个深度优先搜索函数，用于在转置后的图上查找强连通分量
    visited[node] = True  # 标记当前节点为已访问
    component.append(node)  # 将当前节点添加到当前强连通分量中
    for neighbor in graph[node]:  # 遍历当前节点的邻居节点
        if not visited[neighbor]:  # 如果邻居节点未被访问过
            dfs2(graph, neighbor, visited, component)  # 递归调用深度优先搜索函数

def kosaraju(graph):
    # Kosaraju's 算法函数
    # Step 1: 执行第一次深度优先搜索以获取完成时间
    stack = []  # 用于存储节点的栈
    visited = [False] * len(graph)  # 记录节点是否被访问过的列表
    for node in range(len(graph)):  # 遍历所有节点
        if not visited[node]:  # 如果节点未被访问过
            dfs1(graph, node, visited, stack)  # 调用第一个深度优先搜索函数
    # Step 2: 转置图
    transposed_graph = [[] for _ in range(len(graph))]  # 创建一个转置后的图
    for node in range(len(graph)):  # 遍历原图中的所有节点
        for neighbor in graph[node]:  # 遍历每个节点的邻居节点
            transposed_graph[neighbor].append(node)  # 将原图中的边反向添加到转置图中

    # Step 3: 在转置后的图上执行第二次深度优先搜索以找到强连通分量
    visited = [False] * len(graph)  # 重新初始化节点是否被访问过的列表
    sccs = []  # 存储强连通分量的列表
    while stack:  # 当栈不为空时循环
        node = stack.pop()  # 从栈中弹出一个节点
        if not visited[node]:  # 如果节点未被访问过
            scc = []  # 创建一个新的强连通分量列表
            dfs2(transposed_graph, node, visited, scc)  # 在转置图上执行深度优先搜索
            sccs.append(scc)  # 将找到的强连通分量添加到结果列表中
    return sccs  # 返回所有强连通分量的列表

# 例子
graph = [[1], [2, 4], [3, 5], [0, 6], [5], [4], [7], [5, 6]]  # 给定的有向图，第i个列表表示第i个元素与列表中的这些元素相连，比如节点0的相邻节点为节点1，节点1的相邻节点为节点2和节点4
sccs = kosaraju(graph)  # 使用Kosaraju's 算法查找强连通分量
print("Strongly Connected Components:")
for scc in sccs:  # 遍历并打印所有强连通分量
    print(scc)

"""
输出结果：
Strongly Connected Components:
[0, 3, 2, 1]
[6, 7]
[5, 4]
其中每个列表都表示一个有向图中的强连通分量。强连通分量是指其中任意两个顶点之间都存在双向路径的顶点集合。
"""
```

进制转换与ASCII码

chr：把数字变字符

ord：把字符变数字

n(m进制)转其他

转八：oct(int(n,m))

转十：int(n,m)

转十六：hex(int(n,m))

转二：bin(n,m)

ASCⅡ码

ASCII值	控制字符	ASCII值	控制字符	ASCII值	控制字符	ASCII值	控制字符
0	NUL	32	(space)	64	@	96	、
1	SOH	33	!	65	A	97	a
2	STX	34	”	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k