Cheat_Sheet

2024 Summer Compiled by 武昱达

排序算法

归并排序（可用于求逆序对数）

```
# start--mid 和 mid+1--end 都是sorted list
def Merge(a,start,mid,end):
    tmp=[]
    l=start
    r=mid+1
    while l<=mid and r<=end:
        if a[l]<=a[r]:
            tmp.append(a[l])
            l+=1
        else:
            tmp.append(a[r])
            r+=1
    # 以下至少有一个extend了空列表
    tmp.extend(a[l:mid+1])
    tmp.extend(a[r:end+1])
    for i in range(start,end+1):
        a[i]= tmp[i-start]

# 二分
def MergeSort(a,start,end):
    if start==end:
        return

    mid=(start+end)//2
    MergeSort(a,start,mid)
    MergeSort(a,mid+1,end)
    Merge(a,start,mid,end)

a=[8,5,6,4,3,7,10,2]
MergeSort(a,0,7)
print(a)
```

快速排序

```
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):
    # 最右端元素作为基准元素，i,j是两个指针，通过两个元素的交换实现
    # 基准元素左右分别小于、大于他本身。
    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)

# [11, 22, 33, 44, 55, 66, 77, 88]
```

单调栈

奶牛排队，寻找i右侧第一个小于i的索引

题目要求：N为数组长度，hi为数组元素，求最长满足条件子序列，子序列的左边界是该子序列的严格最小值，右边界是该子序列的严格最大值。

```
N,res=int(input()),0
hi=[int(input()) for _ in range(N)]
# left[i]是i左边第一个不小于他的元素的索引，right[i]是i右边第一个不大于他的元素的索引。
# 容易知道，对于指定的i，如果i作为右端点，left[i]是左端点的一个上界，反之同理。
left,right=[-1 for _ in range(N)],[N for _ in range(N)]
stack1,stack2=[],[]

for i in range(N-1,-1,-1):
    while stack1 and hi[stack1[-1]]>hi[i]:
        stack1.pop()
    if stack1:right[i]=stack1[-1]
    stack1.append(i)

for i in range(N):
    while stack2 and hi[stack2[-1]]<hi[i]:
        stack2.pop()
    if stack2:left[i]=stack2[-1]
    stack2.append(i)

for i in range(N):
    for j in range(right[i]-1,i,-1):
        if left[j]<i:
            res=max(j-i+1,res)
            break

print(res)
```

后序表达式求值

从左侧先后弹出两个数字a,b一个算符@，计算a@b，再放回左边。

```
样例输入
    3
    5 3.4 +
    5 3.4 + 6 /
    5 3.4 + 6 * 3 +
样例输出
    8.40
    1.40
    53.40
```

```
def cal(a,b,operate):
    if operate=="+":return a+b
    if operate=="-":return a-b
    if operate=="*":return a*b
    if operate=="/":return a/b

from collections import deque
n,operators=int(input()),("+",'-','*','/')
raw=[deque(map(str,input().split())) for _ in range(n)]

for deq in raw:
    tmp_deq=deque()
    while len(deq)>=1:
        if deq[0] not in operators:
            tmp_deq.append(float(deq.popleft()))
        else:
            b=tmp_deq.pop()
            a=tmp_deq.pop()
            operate=deq.popleft()
            deq.appendleft(cal(a,b,operate))
    print('{:.2f}'.format(tmp_deq[0]))
```

中缀转后缀

以下是 Shunting Yard 算法的基本步骤：

1. 初始化运算符栈和输出栈为空。
2. 从左到右遍历中缀表达式的每个符号。
    - 如果是操作数（数字），则将其添加到输出栈。
    - 如果是左括号，则将其推入运算符栈。
    - 如果是运算符：
        - 如果运算符的优先级大于运算符栈顶的运算符，或者运算符栈顶是左括号，则将当前运算符推入运算符栈。
        - 否则，将运算符栈顶的运算符弹出并添加到输出栈中，直到满足上述条件（或者运算符栈为空）。
        - 将当前运算符推入运算符栈。

- 如果是右括号，则将运算符栈顶的运算符弹出并添加到输出栈中，直到遇到左括号。将左括号弹出但不添加到输出栈中。

3. 如果还有剩余的运算符在运算符栈中，将它们依次弹出并添加到输出栈中。
4. 输出栈中的元素就是转换后的后缀表达式。

样例输入

```
3
7+8.3
3+4.5*(7+2)
(3)*((3+4)*(2+3.5)/(4+5))
```

样例输出

```
7 8.3 +
3 4.5 7 2 + * +
3 3 4 + 2 3.5 + * 4 5 + / *
```

```python
def infix_to_postfix(expression):
    def get_precedence(op):
        precedences = {'+': 1, '-': 1, '*': 2, '/': 2}
        return precedences[op] if op in precedences else 0

    def is_operator(c):
        return c in "+-*/"

    def is_number(c):
        return c.isdigit() or c == '.'

    output = []
    stack = []
    number_buffer = []

    def flush_number_buffer():
        if number_buffer:
            output.append(''.join(number_buffer))
            number_buffer.clear()

    # 主体部分
    for c in expression:
        if is_number(c):
            number_buffer.append(c)
        elif c == '(':
            flush_number_buffer()
            stack.append(c)
        elif c == ')':
            flush_number_buffer()
            while stack and stack[-1] != '(':
                output.append(stack.pop())
            stack.pop()  # popping '('
        elif is_operator(c):
            flush_number_buffer()
            while stack and get_precedence(c) <= get_precedence(stack[-1]):
                output.append(stack.pop())
            stack.append(c)

    flush_number_buffer()
```

```
    while stack:
        output.append(stack.pop())

    return ' '.join(output)

# Read number of expressions
n = int(input())
# Read each expression and convert it
for _ in range(n):
    infix_expr = input()
    postfix_expr = infix_to_postfix(infix_expr)
    print(postfix_expr)
```

Shunting Yard算法

中缀转后缀

```
operators=['+','-','*','/']
cals=['(',')']
# 预处理数据的部分已省略。
def pre_to_post(lst):
    s_op,s_out=[],[]
    while lst:
        tmp=lst.pop(0)
        if tmp not in operators and tmp not in cals:
            s_out.append(tmp)
            continue

        if tmp=="(":
            s_op.append(tmp)
            continue

        if tmp==")":
            while (a:=s_op.pop())!="(":
                s_out.append(a)

        if tmp in operators:
            if not s_op:
                s_op.append(tmp)
                continue
            if is_prior(tmp,s_op[-1]) or s_op[-1]=="(":
                s_op.append(tmp)
                continue
            while (not (is_prior(tmp,s_op[-1]) or s_op[-1]=="(")
                or not s_op):
                s_out.append(s_op.pop())
            s_op.append(tmp)
            continue

    while len(s_op)!=0:
        tmp=s_op.pop()
        if tmp in operators:
            s_out.append(tmp)

    return " ".join(s_out)
```

```
def is_prior(A,B):
    if (A=="*" or A=="/") and (B=="+" or B=="-"):
        return True
    return False

def input_to_lst(x):
    tmp=list(x)

for i in range(int(input())):
    print(pre_to_post(expProcessor(input())))
```

建树

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

Huffman算法

哈夫曼编码树

- **描述**

  构造一个具有n个外部节点的扩充二叉树，每个外部节点Ki有一个Wi对应，作为该外部节点的权。使得这个扩充二叉树的叶节点带权外部路径长度总和最小：$ Min(W1 * L1 + W2 * L2 + W3 * L3 + ... + Wn * Ln) $

  $ Wi $:每个节点的权值。$ Li $:根节点到第$i$个外部叶子节点的距离。编程计算最小外部路径长度总和。

- **输入**

  第一行输入一个整数n，外部节点的个数。第二行输入n个整数，代表各个外部节点的权值。
  2<=N<=100

- **输出**

  输出最小外部路径长度总和。

- **样例输入**

  `4 1 1 3 5`

- **样例输出**

  `17`

```
import heapq
class HuffmanTreeNode:
    def __init__(self,weight,char=None):
        self.weight=weight
        self.char=char
        self.left=None
        self.right=None

    def __lt__(self,other):
```

```
        return self.weight<other.weight

def BuildHuffmanTree(characters):
    heap=[HuffmanTreeNode(weight,char) for char,weight in characters.items()]
    heapq.heapify(heap)
    while len(heap)>1:
        left=heapq.heappop(heap)
        right=heapq.heappop(heap)
        merged=HuffmanTreeNode(left.weight+right.weight,None)
        merged.left=left
        merged.right=right
        heapq.heappush(heap,merged)
    root=heapq.heappop(heap)
    return root

def enpaths_huffman_tree(root):
    # 字典形如(idx,weight):path
    paths={}
    def traverse(node,path):
        if node.char:
            paths[(node.char,node.weight)]=path
        else:
            traverse(node.left,path+1)
            traverse(node.right,path+1)
    traverse(root,0)
    return paths

def min_weighted_path(paths):
    return sum(tup[1]*path for tup,path in paths.items())

n,characters=int(input()),{}
raw=list(map(int,input().split()))
for char,weight in enumerate(raw):
    characters[str(char)]=weight
root=BuildHuffmanTree(characters)
paths=enpaths_huffman_tree(root)
print(min_weighted_path(paths))
```

并查集

发现它，抓住它

```
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
```

```
        rootX = self.find(x)
        rootY = self.find(y)
        if rootX != rootY:
            if self.rank[rootX] > self.rank[rootY]:
                self.parent[rootY] = rootX
            elif self.rank[rootX] < self.rank[rootY]:
                self.parent[rootX] = rootY
            else:
                self.parent[rootY] = rootX
                self.rank[rootX] += 1

def solve():
    n, m = map(int, input().split())
    uf = UnionFind(2 * n)  # 初始化并查集，每个案件对应两个节点,一个是本身，另一个是其对立案件。
    for _ in range(m):
        operation, a, b = input().split()
        a, b = int(a) - 1, int(b) - 1
        if operation == "D":
            uf.union(a, b + n)  # a与b的对立案件合并
            uf.union(a + n, b)  # a的对立案件与b合并
        else:  # "A"
            if uf.find(a) == uf.find(b) or uf.find(a + n) == uf.find(b + n):
                print("In the same gang.")
            elif uf.find(a) == uf.find(b + n) or uf.find(a + n) == uf.find(b):
                print("In different gangs.")
            else:
                print("Not sure yet.")

T = int(input())
for _ in range(T):
    solve()
```

食物链

```
class DisjointSet:
    def __init__(self, n):
        #设[1,n] 区间表示同类，[n+1,2*n]表示x吃的动物，[2*n+1,3*n]表示吃x的动物。
        self.parent = [i for i in range(3 * n + 1)] # 每个动物有三种可能的类型，用 3 * n 来表示每种类型的并查集
        self.rank = [0] * (3 * n + 1)

    def find(self, u):
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])
        return self.parent[u]

    def union(self, u, v):
        pu, pv = self.find(u), self.find(v)
        if pu == pv:
            return False
        if self.rank[pu] > self.rank[pv]:
            self.parent[pv] = pu
```

```
        elif self.rank[pu] < self.rank[pv]:
            self.parent[pu] = pv
        else:
            self.parent[pv] = pu
            self.rank[pu] += 1
        return True


def is_valid(n,statements):
    dsu = DisjointSet(n)

    false_count = 0
    for d, x, y in statements:
        if x>n or y>n:
            false_count += 1
            continue

        if d == 1:  # 同类
            if dsu.find(x)==dsu.find(y+n) or dsu.find(x)==dsu.find(y+2*n):  # 不是同类
                false_count += 1
            else:
                dsu.union(x, y)
                dsu.union(x + n, y + n)
                dsu.union(x + 2 * n, y + 2 * n)

        else:  # X吃Y
            if dsu.find(x) == dsu.find(y) or dsu.find(x + 2*n) == dsu.find(y):
                false_count += 1
            else: #[1,n] 区间表示同类，[n+1,2*n]表示x吃的动物，[2*n+1,3*n]表示吃x的动物
                dsu.union(x + n, y)
                dsu.union(x, y + 2 * n)
                dsu.union(x + 2 * n, y + n)

    return false_count


if __name__ == "__main__":
    N, K = map(int, input().split())
    statements = []
    for _ in range(K):
        D, X, Y = map(int, input().split())
        statements.append((D, X, Y))
    result = is_valid(N,statements)
    print(result)
```

Prim算法

步骤：

1. 起点入堆。
2. 堆顶元素出堆（排序依据是到该元素的开销），如已访问过，continue；否则标记为visited。
3. 访问该节点相邻节点，（访问开销（排序依据），相邻节点）入堆。

4. 相邻节点前驱设置为当前节点（如需）。
5. 当前节点入树

全部精要在于：每次走出下一步的开销都是当前最小的。

Agri-net

题目：用邻接矩阵给出图，求最小生成树路径权值和。

```
4
0 4 9 21
4 0 8 17
9 8 0 16
21 17 16 0
        # 注意这一步continue很关键，因为一个节点会同时很多存在于pq中（这是由出队标记决定的）
        # 如果不设计这一步continue，则会重复加路径长。
```

```
from heapq import heappop, heappush
def prim(matrix):
    ans=0
    pq,visited=[(0,0)],[False for _ in range(N)]
    while pq:
        c,cur=heappop(pq)
        if visited[cur]:continue
        visited[cur]=True
        ans+=c
        for i in range(N):
            if not visited[i] and matrix[cur][i]!=0:
                heappush(pq,(matrix[cur][i],i))
    return ans

while True:
    try:
        N=int(input())
        matrix=[list(map(int,input().split())) for _ in range(N)]
        print(prim(matrix))
    except:break
```

Kruskal算法（能写Prim建议写Prim）

Agri-net

```
class DisJointSet:
    def __init__(self,num_vertices):
        self.parent=list(range(num_vertices))
        self.rank=[0 for _ in range(num_vertices)]

    def find(self,x):
        if self.parent[x]!=x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self,x,y):
        root_x=self.find(x)
```

```
        root_y=self.find(y)
        if root_x!=root_y:
            if self.rank[root_x]<self.rank[root_y]:
                self.parent[root_x]=root_y
            elif self.rank[root_x]>self.rank[root_y]:
                self.parent[root_y]=root_x
            else:
                self.parent[root_x]=root_y
                self.rank[root_y]+=1

# graph是邻接表
def kruskal(graph:list):
    res,edges,dsj=[],[],DisJointSet(len(graph))
    for i in range(len(graph)):
        for j in range(i+1,len(graph)):
            if graph[i][j]!=0:
                edges.append((i,j,graph[i][j]))

    for i in sorted(edges,key=lambda x:x[2]):
        u,v,weight=i
        if dsj.find(u)!=dsj.find(v):
            dsj.union(u,v)
            res.append((u,v,weight))
    return res

while True:
    try:
        n=int(input())
        graph=[list(map(int,input().split())) for _ in range(n)]
        res=kruskal(graph)
        print(sum(i[2] for i in res))
    except EOFError:break
```

Kahn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。

拓扑排序

题目：给出一个图的结构，输出其拓扑排序序列，要求在同等条件下，编号小的顶点在前。

题解中graph是邻接表，形如graph[1]=[2,3,4]，由于本题要求顺序，因此不用队列而用优先队列。

```
from collections import defaultdict
from heapq import heappush,heappop
def Kahn(graph):
    q,ans=[],[]
    in_degree=defaultdict(int)
    for lst in graph.values():
        for vert in lst:
            in_degree[vert]+=1

    for vert in graph.keys():
        if vert not in in_degree or in_degree[vert]==0:
            heappush(q,vert)

    while q:
        vertex=heappop(q)
        ans.append('v'+str(vertex))
        for neighbor in graph[vertex]:
            in_degree[neighbor]-=1
            if in_degree[neighbor]==0:
                heappush(q,neighbor)
    return ans

v,a=map(int,input().split())
graph={}
for _ in range(a):
    f,t=map(int,input().split())
    if f not in graph:graph[f]=[]
    if t not in graph:graph[t]=[]
    graph[f].append(t)

for i in range(1,v+1):
    if i not in graph:graph[i]=[]

res=Kahn(graph)
print(*res)
```

Dijkstra算法

道路（更推荐第二种剪枝写法）

N个以 1 ... N 标号的城市通过单向的道路相连。每条道路包含两个参数：道路的长度和需要为该路付的通行费（以金币的数目来表示）。Bob从1到N。他希望能够尽可能快的到那，但是他囊中羞涩。我们希望能够帮助Bob找到从1到N最短的路径，前提是他能够付的起通行费。输出结果应该只包括一行，即从城市1到城市N所需要的最小的路径长度（花费不能超过K个金币）。如果这样的路径不存在，结果应该输出-1。

S：起点；D：终点；L：道路长；T：通行费。

```
from heapq import heappop,heappush
from collections import defaultdict
K,N,R=int(input()),int(input()),int(input())
graph=defaultdict(list)
for i in range(R):
```

```
    S,D,L,T=map(int,input().split())
    graph[S].append((D,L,T))
def Dijkstra(graph):
    global K,N,R
    q,ans=[],[]
    heappush(q,(0,0,1,0))
    while q:
        l,cost,cur,step=heappop(q)
        if cur==N:return l
        for next,nl,nc in graph[cur]:
            # 剪枝：如果步数不少于N：意味着一定走了回头路，减掉。
            if cost+nc<=K and step+1<N:
                heappush(q,(l+nl,cost+nc,next,step+1))
    return -1
print(Dijkstra(graph))
```

```
from heapq import heappop,heappush
from collections import defaultdict
K,N,R=int(input()),int(input()),int(input())
graph=defaultdict(list)
for i in range(R):
    S,D,L,T=map(int,input().split())
    graph[S].append((D,L,T))

def Dijkstra(graph):
    global K,N,R
    q,ans=[],[]
    min_cost={i:float('inf') for i in range(1,N+1)}
    heappush(q,(0,0,1))
    while q:
        l,cost,cur=heappop(q)
        min_cost[cur]=min(min_cost[cur],cost)
        if cur==N:return l
        for next,nl,nc in graph[cur]:
            # 剪枝1：只有花费小于等于K才能入堆。
            # 剪枝2：只有到达下一个节点的花费比上次更小时才能入堆（否则路程长花费大，无意义）。
            if cost+nc<=K and nc+cost<min_cost[next]:
                heappush(q,(l+nl,cost+nc,next))
    return -1
print(Dijkstra(graph))
```

Kosaraju算法

```
def dfs1(graph, node, visited, stack):
    visited[node] = True
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs1(graph, neighbor, visited, stack)
    stack.append(node)

def dfs2(graph, node, visited, component):
    visited[node] = True
```

```
    component.append(node)
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs2(graph, neighbor, visited, component)

def kosaraju(graph):
    # Step 1: Perform first DFS to get finishing times
    stack = []
    visited = [False] * len(graph)
    for node in range(len(graph)):
        if not visited[node]:
            dfs1(graph, node, visited, stack)

    # Step 2: Transpose the graph
    transposed_graph = [[] for _ in range(len(graph))]
    for node in range(len(graph)):
        for neighbor in graph[node]:
            transposed_graph[neighbor].append(node)

    # Step 3: Perform second DFS on the transposed graph to find SCCs
    visited = [False] * len(graph)
    sccs = []
    while stack:
        node = stack.pop()
        if not visited[node]:
            scc = []
            dfs2(transposed_graph, node, visited, scc)
            sccs.append(scc)
    return sccs

# Example
graph = [[1], [2, 4], [3, 5], [0, 6], [5], [4], [7], [5, 6]]
sccs = kosaraju(graph)
print("Strongly Connected Components:")
for scc in sccs:
    print(scc)

"""
Strongly Connected Components:
[0, 3, 2, 1]
[6, 7]
[5, 4]

"""
```

无向图判断连通和成环

判断无向图是否连通有无回路

```
from collections import defaultdict,deque
# graph是邻接表{1:[2,3,4]}
def is_connected(graph,n):
    dq=deque()
    dq.append(0)
    visited=set()
```

```
    visited.add(0)
    while dq:
        cur_vert=dq.popleft()
        for next_vert in graph[cur_vert]:
            if next_vert not in visited:
                dq.append(next_vert)
                visited.add(next_vert)
    return len(visited)==n

def is_loop(graph):
    global_visited=set()
    for vertex in graph:
        if vertex not in global_visited:
            # 以下是一个BFS函数。
            local_visited={}
            dq=deque()
            dq.append((vertex,0))
            local_visited[vertex]=0
            global_visited.add(vertex)
            while dq:
                cur_vert,steps=dq.popleft()
                for next_vert in graph[cur_vert]:
                    if next_vert in local_visited:
                        if local_visited[next_vert]>=steps:
                            return True
                    else:
                        dq.append((next_vert,steps+1))
                        local_visited[next_vert]=steps+1
                        global_visited.add(next_vert)
    return False

n,m=map(int,input().split())
graph=defaultdict(list)
for _ in range(m):
    a,b=map(int,input().split())
    graph[a].append(b)
    graph[b].append(a)
print('connected:yes' if is_connected(graph,n) else 'connected:no')
print('loop:yes' if is_loop(graph) else 'loop:no')
```

二分查找算法

月度开销

```
n,m=map(int,input().split())
expend=[int(input()) for i in range(n)]
def check(x):
    # 判断x作为最大月度开销是否可以实现，如果可以实现，则说明不够小或刚好符合题意。
    # 看m个方案是否可行。
    nums,s=1,0
    for i in range(n):
        if expend[i]+s>x:
            s=expend[i]
            nums+=1     # 求和大于设定的最大月度开销，则应该插入挡板，分份+1
        else:s+=expend[i]
```

```
    return nums>m   # if nums>m return True,else return False

lo,hi,res=max(expend),sum(expend)+1,1
while lo<hi:
    mid=(lo+hi)//2
    if check(mid):lo=mid+1
    else:res,hi=mid,mid

print(res)
```

其他注意事项（Debug）

1. 写Dijkstra采用“出堆标记”是肯定正确的，尽管入堆标记可能更快。
2. 抄代码的时候注意缩进。
3. 注意要把题目数据全部接收，即使程序进行一半时已经得出结果。
4. 注意字符串和int类型，尤其不要犯‘1’==1这种错误。
5. 注意对于某一个类的实例，不要重复定义。