Cheating Paper

2300011406 陈睿阳

1. 排序算法

1.1快排

```python
#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)

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

1.2归并排序

```python
def mergeSort(arr):
    if len(arr) > 1:
        mid = len(arr)//2

        L = arr[:mid]   # Dividing the array elements
        R = arr[mid:] # Into 2 halves

        mergeSort(L) # Sorting the first half
        mergeSort(R) # Sorting the second half

        i = j = k = 0
        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] <= R[j]:
```

```python
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1


if __name__ == '__main__':
    arr = [12, 11, 13, 5, 6, 7]
    mergeSort(arr)
    print(' '.join(map(str, arr)))
# Output: 5 6 7 11 12 13
```

2. 树

2.1解析树

```python
class Stack(object):
    def __init__(self):
        self.items = []
        self.stack_size = 0

    def isEmpty(self):
        return self.stack_size == 0

    def push(self, new_item):
        self.items.append(new_item)
        self.stack_size += 1

    def pop(self):
        self.stack_size -= 1
        return self.items.pop()

    def peek(self):
        return self.items[self.stack_size - 1]

    def size(self):
        return self.stack_size


class BinaryTree:
    def __init__(self, rootObj):
        self.key = rootObj
```

```
        self.leftChild = None
        self.rightChild = None

    def insertLeft(self, newNode):
        if self.leftChild == None:
            self.leftChild = BinaryTree(newNode)
        else:  # 已经存在左子节点。此时，插入一个节点，并将已有的左子节点降一层。
            t = BinaryTree(newNode)
            t.leftChild = self.leftChild
            self.leftChild = t

    def insertRight(self, newNode):
        if self.rightChild == None:
            self.rightChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.rightChild = self.rightChild
            self.rightChild = t

    def getRightChild(self):
        return self.rightChild

    def getLeftChild(self):
        return self.leftChild

    def setRootVal(self, obj):
        self.key = obj

    def getRootVal(self):
        return self.key

    def traversal(self, method="preorder"):
        if method == "preorder":
            print(self.key, end=" ")
        if self.leftChild != None:
            self.leftChild.traversal(method)
        if method == "inorder":
            print(self.key, end=" ")
        if self.rightChild != None:
            self.rightChild.traversal(method)
        if method == "postorder":
            print(self.key, end=" ")


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


exp = "( ( 7 + 3 ) * ( 5 - 2 ) )"
pt = buildParseTree(exp)
for mode in ["preorder", "postorder", "inorder"]:
    pt.traversal(mode)
    print()

"""
* + 7 3 - 5 2
7 3 + 5 2 - *
7 + 3 * 5 - 2
"""

# 代码清单6-10
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()

print(evaluate(pt))
# 30

#代码清单6-14 后序求值
def postordereval(tree):
    opers = {'+':operator.add, '-':operator.sub,
             '*':operator.mul, '/':operator.truediv}
    res1 = None
    res2 = None
    if tree:
        res1 = postordereval(tree.getLeftChild())
        res2 = postordereval(tree.getRightChild())
        if res1 and res2:
            return opers[tree.getRootVal()](res1,res2)
        else:
            return tree.getRootVal()
```

```
print(postordereval(pt))
# 30

#代码清单6-16 中序还原完全括号表达式
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))
```

题目：前中序建树

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

def build_tree(preorder, inorder):
    if not preorder or not inorder:
        return None
    root_value = preorder[0]
    root = TreeNode(root_value)
    root_index_inorder = inorder.index(root_value)
    root.left = build_tree(preorder[1:1+root_index_inorder], 
inorder[:root_index_inorder])
    root.right = build_tree(preorder[1+root_index_inorder:], 
inorder[root_index_inorder+1:])
    return root

def postorder_traversal(root):
    if root is None:
        return ''
    return postorder_traversal(root.left) + postorder_traversal(root.right) + 
root.value

while True:
    try:
        preorder = input().strip()
        inorder = input().strip()
        root = build_tree(preorder, inorder)
        print(postorder_traversal(root))
    except EOFError:
        break
```

2.2二叉堆

```
class BinHeap:
    def __init__(self):
        self.heapList = [0]
```

```python
        self.currentSize = 0

    def percUp(self, i):
        while i // 2 > 0:
            if self.heapList[i] < self.heapList[i // 2]:
                tmp = self.heapList[i // 2]
                self.heapList[i // 2] = self.heapList[i]
                self.heapList[i] = tmp
            i = i // 2

    def insert(self, k):
        self.heapList.append(k)
        self.currentSize = self.currentSize + 1
        self.percUp(self.currentSize)

    def percDown(self, i):
        while (i * 2) <= self.currentSize:
            mc = self.minChild(i)
            if self.heapList[i] > self.heapList[mc]:
                tmp = self.heapList[i]
                self.heapList[i] = self.heapList[mc]
                self.heapList[mc] = tmp
            i = mc

    def minChild(self, i):
        if i * 2 + 1 > self.currentSize:
            return i * 2
        else:
            if self.heapList[i * 2] < self.heapList[i * 2 + 1]:
                return i * 2
            else:
                return i * 2 + 1

    def delMin(self):
        retval = self.heapList[1]
        self.heapList[1] = self.heapList[self.currentSize]
        self.currentSize = self.currentSize - 1
        self.heapList.pop()
        self.percDown(1)
        return retval

    def buildHeap(self, alist):
        i = len(alist) // 2
        self.currentSize = len(alist)
        self.heapList = [0] + alist[:]
        while (i > 0):
            print(f'i = {i}, {self.heapList}')
            self.percDown(i)
            i = i - 1
        print(f'i = {i}, {self.heapList}')



bh = BinHeap()
bh.buildHeap([9, 5, 6, 2, 3])
"""
i = 2, [0, 9, 5, 6, 2, 3]
i = 1, [0, 9, 2, 6, 5, 3]
```

```
i = 0, [0, 2, 3, 6, 5, 9]
"""

for _ in range(bh.currentSize):
    print(bh.delMin())
"""
2
3
5
6
9
"""
```

2.3AVL

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
        self.height = 1

class AVL:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if not self.root:
            self.root = Node(value)
        else:
            self.root = self._insert(value, self.root)

    def _insert(self, value, node):
        if not node:
            return Node(value)
        elif value < node.value:
            node.left = self._insert(value, node.left)
        else:
            node.right = self._insert(value, node.right)

        node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))

        balance = self._get_balance(node)

        if balance > 1:
            if value < node.left.value: # 树形是 LL
                return self._rotate_right(node)
            else:   # 树形是 LR
                node.left = self._rotate_left(node.left)
                return self._rotate_right(node)

        if balance < -1:
            if value > node.right.value:    # 树形是 RR
                return self._rotate_left(node)
            else:   # 树形是 RL
                node.right = self._rotate_right(node.right)
```

```
                return self._rotate_left(node)

        return node

    def _get_height(self, node):
        if not node:
            return 0
        return node.height

    def _get_balance(self, node):
        if not node:
            return 0
        return self._get_height(node.left) - self._get_height(node.right)

    def _rotate_left(self, z):
        y = z.right
        T2 = y.left
        y.left = z
        z.right = T2
        z.height = 1 + max(self._get_height(z.left), self._get_height(z.right))
        y.height = 1 + max(self._get_height(y.left), self._get_height(y.right))
        return y

    def _rotate_right(self, y):
        x = y.left
        T2 = x.right
        x.right = y
        y.left = T2
        y.height = 1 + max(self._get_height(y.left), self._get_height(y.right))
        x.height = 1 + max(self._get_height(x.left), self._get_height(x.right))
        return x

    def preorder(self):
        return self._preorder(self.root)

    def _preorder(self, node):
        if not node:
            return []
        return [node.value] + self._preorder(node.left) +
self._preorder(node.right)
    def delete(self, value):
        self.root = self._delete(value, self.root)

    def _delete(self, value, node):
        if not node:
            return node

        if value < node.value:
            node.left = self._delete(value, node.left)
        elif value > node.value:
            node.right = self._delete(value, node.right)
        else:
            if not node.left:
                temp = node.right
                node = None
                return temp
            elif not node.right:
                temp = node.left
```

```
                node = None
                return temp

            temp = self._min_value_node(node.right)
            node.value = temp.value
            node.right = self._delete(temp.value, node.right)

        if not node:
            return node

        node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))

        balance = self._get_balance(node)

        # Rebalance the tree
        if balance > 1:
            if self._get_balance(node.left) >= 0:
                return self._rotate_right(node)
            else:
                node.left = self._rotate_left(node.left)
                return self._rotate_right(node)

        if balance < -1:
            if self._get_balance(node.right) <= 0:
                return self._rotate_left(node)
            else:
                node.right = self._rotate_right(node.right)
                return self._rotate_left(node)

        return node

    def _min_value_node(self, node):
        current = node
        while current.left:
            current = current.left
        return current
n = int(input().strip())
sequence = list(map(int, input().strip().split()))

avl = AVL()
for value in sequence:
    avl.insert(value)

print(' '.join(map(str, avl.preorder())))
```

2.4并查集

```
def find(i):
    if parent[i] != i:
        return find(parent[i])
    else:
        return i

def union(i,j):
    parent[find(i)] = find(j)
#以下为优化的合并
```

```python
def join(x,y):
    fx,fy=find(x),find(y)
    if fx!=fy:
        if h[fx]<h[fy]:
            a[fx]=fy
        else:
            a[fy]=fx
            if h[fx]==h[fy]:
                h[fx]+=1
```

3.图

3.1拓扑排序

```python
from collections import deque, defaultdict

def topological_sort(graph):
    indegree = defaultdict(int)
    result = []
    queue = deque()

    # 计算每个顶点的入度
    for u in graph:
        for v in graph[u]:
            indegree[v] += 1

    # 将入度为 0 的顶点加入队列
    for u in graph:
        if indegree[u] == 0:
            queue.append(u)

    # 执行拓扑排序
    while queue:
        u = queue.popleft()
        result.append(u)

        for v in graph[u]:
            indegree[v] -= 1
            if indegree[v] == 0:
                queue.append(v)

    # 检查是否存在环
    if len(result) == len(graph):
        return result
    else:
        return None

# 示例调用代码
graph = {
    'A': ['B', 'C'],
    'B': ['C', 'D'],
    'C': ['E'],
    'D': ['F'],
    'E': ['F'],
    'F': []
}
```

```
sorted_vertices = topological_sort(graph)
if sorted_vertices:
    print("Topological sort order:", sorted_vertices)
else:
    print("The graph contains a cycle.")

# Output:
# Topological sort order: ['A', 'B', 'C', 'D', 'E', 'F']
```

3.2强连通单元

```
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]

"""
```

3.3最短路径

```
#使用heapq代替queue进行bfs即可
```

3.4最小生成树

```
#参照兔子与星空
'''第一行只包含一个表示星星个数的数n，n不大于26，并且这n个星星是由大写字母表里的前n个字母表示。接下来的n-1行是由字母表的前n-1个字母开头。最后一个星星表示的字母不用输入。对于每一行，以每个星星表示的字母开头，然后后面跟着一个数字，表示有多少条边可以从这个星星到后面字母表中的星星。如果k是大于0，表示该行后面会表示k条边的k个数据。每条边的数据是由表示连接到另一端星星的字母和该边的权值组成。权值是正整数的并且小于100。该行的所有数据字段分隔单一空白。该星星网络将始终连接所有的星星。该星星网络将永远不会超过75条边。没有任何一个星星会有超过15条的边连接到其他星星（之前或之后的字母）。'''
import heapq
n = int(input())
graph = {}
for i in range(n):
    graph[chr(ord('A') + i)] = {}
for _ in range(n - 1):
    l = list(input().split())
    head = l[0]
    if int(l[1]) > 0:
        for i in range(int(l[1])):
            graph[head][l[2 + 2 * i]] = int(l[3 + 2 * i])
            graph[l[2 + 2 * i]][head] = int(l[3 + 2 * i])

def prim(start):
    q = []
    heapq.heapify(q)
    for vtx in graph[start]:
        heapq.heappush(q,(graph[start][vtx], vtx))
    dis = 0
    tree = set(start)
    while len(tree) < n:
        d,vtx = heapq.heappop(q)
        if not vtx in tree:
            dis += d
            tree.add(vtx)
            for next_vtx in graph[vtx]:
                if not next_vtx in tree:
                    heapq.heappush(q,(graph[vtx][next_vtx],next_vtx))
    return dis

ans = float('inf')
for i in graph:
    ans = prim(i)
print(ans)
```

4.部分题目

4.1并查集题目 — 食物链

```
'''动物王国中有三类动物A,B,C，这三类动物的食物链构成了有趣的环形。A吃B， B吃C，C吃A。
现有N个动物，以1－N编号。每个动物都是A,B,C中的一种，但是我们并不知道它到底是哪一种。
有人用两种说法对这N个动物所构成的食物链关系进行描述：
第一种说法是"1 X Y"，表示X和Y是同类。
第二种说法是"2 X Y"，表示X吃Y。
此人对N个动物，用上述两种说法，一句接一句地说出K句话，这K句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。
1） 当前的话与前面的某些真的话冲突，就是假话；
2） 当前的话中X或Y比N大，就是假话；
3） 当前的话表示X吃X，就是假话。
你的任务是根据给定的N（1 <= N <= 50,000）和K句话（0 <= K <= 100,000），输出假话的总数。
输入
第一行是两个整数N和K，以一个空格分隔。
以下K行每行是三个正整数 D，X，Y，两数之间用一个空格隔开，其中D表示说法的种类。
若D=1，则表示X和Y是同类。
若D=2，则表示X吃Y。'''
def join(x,y):
    fx,fy=find(x),find(y)
    if fx!=fy:
        if h[fx]<h[fy]:
            a[fx]=fy
        else:
            a[fy]=fx
            if h[fx]==h[fy]:
                h[fx]+=1
def find(x):
    if a[x]!=x:
        a[x]=find(a[x])
    return a[x]

n,k=map(int,input().split())
s=0
a=[i for i in range(3*n)]
h=[0]*(3*n)
for _ in range(k):
    d,x,y=map(int,input().split())
    if x>n or y>n:
        s+=1
        continue
    if x==y:
        if d==2:
            s+=1
        continue
    x,y=x-1,y-1
    if d==1:
        if find(x)==find(y+n) or find(y)==find(x+n):
            s+=1
            continue
        join(x,y),join(x+n,y+n),join(x+2*n,y+2*n)
    if d==2:
        if find(x)==find(y) or find(x)==find(y+n):
            s+=1
            continue
```

```
        join(x+n,y)
        join(x+2*n,y+n)
        join(x,y+2*n)
print(s)
```