大作业

数据结构的本质是数据之间逻辑的体现，各个数据结构的特性最终应当落实在算法实践当中

栈与队列

栈和队列的特性分别是：一个先进后出，一个先进先出，与之具体结合的算法便是dfs-stack和bfs-queue了

dfs中使用stack存储当前可以抵达的节点，并通过先进后出实现“每走一个step，优先关注当前这一step接下来的可行节点”，从而实现深度优先；bfs中使用queue存储当前可以抵达的节点，并通过后进后出实现“每走一个step，优先关注当前这一step同一水平的其他step，直到当前无同一水平的step”

具体实现关键步骤如下：

```
def dfs(move, target, stack, visited, result):
    while stack:
        cur, value = stack.pop() # 深度优先
        if cur == target:
            return result
        result += value # 视实际需求而定
        visited.append(cur) # 添加访问节点
        for m in move:
            add_m, add_v = m
            if not cur + m ...  and not cur in visited: # 做越界判断，也即剪枝
                stack.append((cur + m, add_v)) # 添加可行节点
    return None
# 特别地，对于有回溯的dfs而言，visited可以设置为长度为节点数的列表，并通过设值为0/1进行回溯/访问操作

from collections import deque
queue = deque()
depth = 0 # 如有判断层次的需要
def bfs(move, target, queue, visited, result):
    global depth # 如有判断层次的需要
    while queue:
        cur, value, cur_depth = queue.popleft() # 广度优先
        if not cur_depth == depth:
            depth = cur_depth # 如有判断层次的需要，此步进行相应操作
            pass
        if cur == target:
            return result
        result += value # 视实际需求而定
        visited.append(cur) # 添加访问节点
        for m in move:
            add_m, add_v = m
            if not cur + m ...  and not cur in visited: # 做越界判断，也即剪枝
                queue.append((cur + m, add_v, cur_depth + 1)) # 添加可行节点
    return None
```

通过以上分析，可以想到：如果对queue和stack的规则进行改变，如依据优先级进行节点弹出，就可以实现不同的算法，在之后图的一些算法当中可以见到。

二叉树

树的特性在于多分，且父节点与子节点之间具有同类性。因此，对父节点的操作本身就可以适用于子节点，因此便实现了：将对父节点的处理分解为对其子节点的分别处理，与递归的思想相一致（本质上也是使用递归进行求解）

最典型的莫过于树的遍历：

```
def pre_loop(tree_node): # 前序遍历，以二叉树为例
    if tree_node == None:
        return ''
    result = tree_node.val + pre_loop(tree_node.left) + pre_loop(tree_node.right)
    return result
```

因此，树的最大难点在于如何建树，即如何依据所给的数据搭建所需的树，随后再依据递归算法对问题进行解决。

特别地，对于树的层次遍历，利用队列queue解决显然是最合适的，因为更多地关注兄弟节点

嵌套括号表示法

```
# 嵌套括号表示法对应的字符串具有以下特性：对于节点，如果出现符合节点条件的值（或尚未出现节点终止标识），则更新当前节点为新的父节点，并将新节点作为前一节点的对应子树，直至该新节点处理完后再回归至前一节点。
# 以A(B(E),C(F,G),D(H(I)))为例，将A作为节点，遇到(时说明可能存在子节点，故将当前节点保留，以(...)内数据作为子节点；遇到B，将B更新为当前节点，并将B作为A的子节点；直到处理完B节点，即遇到)，则将B更新为A作为父节点。
# 不难发现，其中具有明显的“先进后出”性质，故使用栈作为解决结构

def parse_tree(s): # 以节点为字母为例
    stack = []
    node = None
    for char in s:
        if char.isalpha():  # 如果是字母，创建新节点
            if node and stack:
                stack[-1].children.append(node) # 如果栈不为空，把节点加入到栈顶节点的子节点列表中
            node = TreeNode(char)
        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.parent = stack[-1]
                stack[-1].children.append(node) # 同上
                node = stack.pop()  # 更新当前节点
    return node  # 根节点
```

扩展n叉树

扩展树与括号嵌套表示法无二，本质上仍应利用“先进后出”的性质，只是缺少了判断有无子节点的依据，但由于除了点节点外每个节点都有两个子节点，故也可以作为弹出节点的依据

```
def 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)
            stack.append(node) # 直接添加新节点，因为不存在节点截止的判断条件，故放于后头判
断
        elif char == '.':
            node = None
            # 如若待加入节点满足子节点数为1，则将当前节点加入栈顶节点的子节点，并将栈顶节点弹出
            while stack and len(stack[-1].children) == n-1:
                stack[-1].children.append(node)
                node = stack.pop()
            if stack: # 如若还存在节点，说明当前节点应当为栈顶节点的第1位节点（也就是左子节
点）
                stack[-1].children.append(node)
    return node
```

可以看见，树的构建过程本质上是对输入信息逐步分析的过程，抽象而言即为从根开始搭建各种节点，类似于培养一颗小树，有根之后才有各个枝节，有了枝节才有各个叶片。只是一般的搭建往往是按类似于dfs的思路，走完一根枝条再看下一根，这也与实际构建过程中使用到了stack相一致

同时，在搭建过程中，节点的变更是极为重要的。要么根据输入信息的截止信号进行变更（如嵌套括号中的右括号），要么根据优先级进行变更（一般以父节点优先级高于子节点），此时优先级应当为节点的自身属性

以优先级作为节点进行变更的，最典型的莫过于堆的建立

```
## 最小堆的列表实现
class heap_list(object):
    def __init__(self, List):
        self.heap = sorted(List) # 排序后该列表必然满足堆的性质
        self.length = len(List)

    def getidx(self, relation, idx): ## 获取列表第idx位元素的父节点和左右子节点的索引
        if relation == 'parent':
            return (idx-1)//2
        elif relation == 'left':
            return 2*idx + 1
        elif relation == 'right':
            return 2*idx + 2

    def minchild(self, idx): ## 返回值最小的子节点，如果没有则返回None
        if 2*idx + 1 >= self.length:
            return None
        elif 2*idx + 2 >= self.length:
            return 2*idx + 1
        else:
```

```
            return [2*idx + 1, 2*idx + 2][self.heap[2*idx + 1] > self.heap[2*idx
+ 2]]

    def up(self, idx): ## 将列表第idx位元素进行上浮
        while idx > 0 and self.heap[idx] < self.heap[self.getidx('parent', idx)]:
            parent_idx = self.getidx('parent', idx) # 记录父节点属性
            self.heap[idx], self.heap[parent_idx] = self.heap[parent_idx],
self.heap[idx] # 交换节点
            idx = parent_idx # idx上移至父节点所在处

    def down(self, idx): ## 将列表第idx位元素进行下沉
        child = self.minchild(idx)
        if not child == None:
            if self.heap[child] < self.heap[idx]:
                self.heap[idx], self.heap[child] = self.heap[child],
self.heap[idx] # 交换节点
            self.down(child) # 继续下沉


    def popmin(self): ## 弹出列表最小值
        if self.length == 0:
            return None
        else:
            Min = self.heap[0]
            self.length -= 1
            self.heap[0] = self.heap[-1] # 将列表最后一位替补至首位，以进行更新
            self.heap.pop() # 末项已经替补至首位，故应当删除该项
            self.down(0) # 对替补至首位的末项进行下沉
            return Min

    def push(self, val): ## 添加新数据
        self.heap.append(val)
        self.length += 1
        self.up(self.length-1)
```

可以看见，堆的搭建本质上是利用了节点本身的值作为优先级，较小值作为父节点。一般直接使用heapq库即可

除了从根开始搭建树之外，也有从底部开始搭建树的思路，如哈夫曼编码树

```
## 哈夫曼编码树
import heapq

class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    def __lt__(self, other):
        return self.freq < other.freq # 给予节点之间的比较标准，这为堆的搭建做了铺垫

def huffman_encoding(char_freq):
```

```
    heap = [Node(char, freq) for char, freq in char_freq.items()] # 为每个字符和权重构建节点
    heapq.heapify(heap) # 进行堆排序，其中以节点的freq属性作为优先级，较小值在前

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(None, left.freq + right.freq) # 合并之后父节点对应的字符值是空
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged) # 将所得新节点加入待组装列表中

    return heap[0]
```

并查集

将输入信息分为多个类别，并在输入过程中对类别进行更新。实际上就是对每个输入节点附加了上行节点和下行节点的属性，可以认为是树的一种。

```
# 维护一个数组parent，用于记录与当前索引所在类的根节点，father最终存储的是索引值。
# 对于每个节点，我们规定一个属性为秩，在合并时将秩较小的集合的根节点指向秩较大的集合的根节点，这样做可以避免形成一个非常不平衡的树（如形成环等）。
class DisjointSet:
    def __init__(self, size):
        self.parent = [i for i in range(size)]
        self.rank = [0] * size  # 秩数组

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # 路径压缩
        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  # 如果秩相同，合并后根节点的秩加1
# 如果类别之间有对立或其他关系，可以创建每个类别对应的对立节点（即把father扩展为2n、3n等），并注意更新
# 以OJ上发现它、找到它为例:
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
    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.")
```

（对于并查集，我自认为尚未完全理解，主要是对于“每个元素的父节点无法及时更新”这一点，不太明白路径压缩和按秩合并是如何解决这个问题的。但大体的逻辑思路是理解的）

图

图是树的普遍形式，各个节点之间不再是单纯的单向父子关系而比较复杂，因此递归在图当中失去了较强的统治力。而深搜和广搜则大放异彩。除了基本的dfs和bfs之外，图中还有其他各式各样的算法，但无一例外都是对节点优先级的再定义

Dijkstra

用于解决：在有向图G=(V,E)中，假设每条边E[i]的长度为w[i]，找到由顶点V到其余各点的最短值

简单的思路即为bfs，只是元素优先级的判断发生了变化：变为V到达该节点所需最小权重值。因此我们需要计算这个值并与节点属性一同存进queue中，然后进行弹出。具体的判断排序可以通过堆来实现

```
# OJ 道路
from heapq import *

def find(graph, start, end, max_cost):
    queue = [(0, 0, start)]
    heapify(queue) # 使用堆，将queue中的元素设为元组
    # 元组第一位即为优先级的对应元素，这样利用堆的性质，我们可以实现对节点or边的优先排序
    while queue:
         # 此处优先级为长度，故将长度放在第一位以满足堆的性质
        cur_length, cur_cost, cur_city = heappop(queue)
        if cur_city == end:
            return cur_length
        for nei in graph[cur_city]:
            for value in graph[cur_city][nei]: # 这里对应的move就是指向临近节点
                add_length, add_cost = value
                if cur_cost + add_cost > max_cost:
                    continue
                else:
                    heappush(queue, (cur_length + add_length, cur_cost +
add_cost, nei))
    return -1
```

Prim

最小生成树实现，即给定无向图G=(V,E)，求使各个边权重和最小的G的子集

为了实现最小生成树，我们首先关注节点属性。对于每个节点，我们记录通往该节点的最小权重值，初始化为Inf，并在选取节点的过程中进行更新，最终遍历完所有节点的权重值表即为所求。在实际实现中，可以以列表当中的权重值为优先级进行排序储存，从而在形式上与Dijkstra相一致

```python
# OJ Truck History
from heapq import *

def prim(codes):
    result, cost = 0, [float('inf')]*len(codes) # 使用cost存储链接各个节点的最小权重边
    queue, visited = [(0,0)], [0]*len(codes)
    heapify(queue)
    cost[0] = 0
    while queue:
        cur_cost, cur = heappop(queue) # 使用cost作为优先级判断
        if visited[cur]:
            continue
        result += cur_cost
        visited[cur] = 1 # 设置访问，但后续不设置回溯的原因是：当前cur必然是结果V中的一部分
        for idx in range(len(codes)): # 实际遍历使用的当前节点的邻居，该题目所有节点彼此为邻居，故遍历全部
            if not visited[idx] and not idx == cur:
                new_cost = distance(codes[cur], codes[idx])
                if cost[idx] > new_cost: # 更新对应的cost并进行剪枝
                    cost[idx] = new_cost
                    heappush(queue, (cost[idx], idx))
    return result

def distance(code1, code2):
    result = 0
    for idx in range(len(code1)):
        result += not code1[idx] == code2[idx]
    return result
```

拓扑排序

通过拓扑排序的定义即可得到思路：记录每个节点的入度，把入度为0的节点进行输出即可，与Dijkstra算法相似。一般情况下，实际需求可能依附于节点属性进行输出，因此“入度为0”可作为一个条件（or门票）使节点入堆，本质上也是一种剪枝

```python
from heapq import *

N, E = map(int, input().split())
in_degrees = [0]*(N+1) # 节点入度存储
edges = {i+1:[] for i in range(N)}
for _ in range(E):
    fr, to = map(int, input().split())
    edges[fr].append(to)
    in_degrees[to] += 1
queue = []
result = []
for idx in range(1,N+1):
    if in_degrees[idx] == 0: # 更新队列
        queue.append(idx)
heapify(queue)
while queue:
    cur = heappop(queue) # 本质上是用入度和节点属性作为了优先级，只是入度充当了“门票”的角色
    result.append(str(cur))
    for n in edges[cur]:
```

```
            in_degrees[n] -= 1
            if in_degrees[n] == 0:
                heappush(queue, n)
print('v' + ' v'.join(result))
```

算法

归并排序

```
def merge_sort(lists):
    # 递归结束条件
    if len(lists) <= 1:
        return lists

    # 分治进行递归
    middle = len(lists)//2
    left = merge_sort(lists[:middle])
    right = merge_sort(lists[middle:])

    # 将两个有序数组进行合并
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        # 逐步比较left中元素和right中元素，将较小值放入到result中
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    # 将未被扫描到的说明都是最大的，而且这种“最大”已经是排好序了的，所以可以直接追加到result后面，
    if i == len(left):
        result.extend(right[j:])
    else:
        result.extend(left[i:])

    return result

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

单调栈

```
# 单调栈，本质上是维护一个栈底到栈顶的单调性（示例为单调递增），并返回元素的弹出位置idx（为了维护递增性）。简单理解就是返回元素A向右（按入栈顺序）数第一个大于（依据判断）该元素A的元素B的下标。
def humd(num):
    stack = []
    result = [0]*len(num) # 这里的0是未找到B时的默认输出
    for idx in range(len(num)): # 入栈顺序
        while stack and num[stack[-1]] < num[idx]: # 判断依据
            result[stack.pop()] = idx + 1 # 返回元素B下表
        stack.append(idx)
    return result

n = int(input())
num = list(map(int, input().split()))
print(*humd(num))
```

二分算法

本质是分治，关键在于判断。对于每个题目，mid的计算方式基本相同，但判断的不同决定了二分是如何在该题目解决中关键作用

```
# 月度开销
# 与一般题目不同，这道题目存在两层判断：一层是对价值总和的判断，若>mid则更新并进入下一层；另一层是对所需总段数的判断，若>limit则返回False，说明mid偏小。所以其实不难理解）
def check(x, values, limit):
    num, total = 1, 0
    idx = 0
    while idx < len(values) and num <= limit:
        if total + values[idx] > x: # 如果超出当前指定限额x，则更新total并计数+1
            total = values[idx]
            num += 1
        else:
            total += values[idx]
        idx += 1
    return num <= limit

def binary(values, limit):
    result = 1
    left, right = max(values), sum(values)
    while left < right:
        mid = (left + right)//2
        if check(mid, values, limit): # 说明mid值作为限额偏大
            result = mid
            right = mid
        else:
            left = mid + 1 # 说明mid值作为限额偏小
    return result

N, M = map(int, input().split())
values = []
for _ in range(N):
```

```
    values.append(int(input()))
print(binary(values, M))
```

也可以用自带的库：bisect

KMP

```
# 获取字符串的next数据，代表着当前idx之前，首尾相同字符串的最大长度
# 此处的末尾始终表征着索引为idx
def get_next(s):
    result = [0] # 第一个为0
    temp = 0 # 记录首位字符串长度
    idx = 1
    while idx < len(s):
        if s[idx] == s[temp]: # 如果idx与temp相同，则说明末尾可以在前一位的基础上延续
            temp += 1
            result.append(temp) # 将idx位的next值更新为temp
            idx += 1
        elif temp:
            temp = result[temp-1] # 如若不同，则考虑剪短已有首位字符串，长度由首位对应的
next值决定
            # 此处为temp-1：temp对应的是长度，-1之后才为对应的索引
        else:
            result.append(0) # 如果temp为0，说明已经剪到了开头仍没有重复的，则确定idx位
next值为0
            idx += 1
    return result
```

All last

这篇大作业的review只是一点我在这一学期关于某些常见数据结构和算法的理解，并没有完全覆盖原有的知识点，虽说比不上大佬们的见解，但实际写出来莫名有点感动，对于部分算法也有了更深的感悟。这一个学期忙忙碌碌，但在闲着的时候永远手痒痒想AC那么几道题，虽然往往是以“没思路”和WA大败而归。对于数算这门课的投入，一开始还挺多，但到后来确实因为其他课程的原因，投入精力没有之前计概那么多了，不过还是会在周末挤出几个小时逛逛OJ和洛谷的（加上作业似乎也有接近十个小时了）。这也就导致到最后复习也不完全，对于并查集理解不是很深刻，以至于考试最后两道题发挥失误没做出来，达到了底线AC4。加之那天早上英语pre，中午某门课提复习重点，上完找不到打印cheating cheet的地方慌忙赶到考试现场，心态确实不怎么好。笔试也是堪堪拿到了92，总的看下来似乎达不到优秀，真的很自责很伤心，但毕竟也是自己的原因。

当然，真的很感谢闫老师和各位群友，从当初计概激发了我对编程的兴趣，到如今数算让我深刻感受到了何为数据结构，何为算法本质。时至今日仍会在AC或搞懂某个算法时感到欣悦，仍会因为WA和TLE感到困惑，仍会debug到忘记时间、忘记吃饭（但不会忘记睡觉）。之前计概可以说是编程的入门，那么这学期的数算便是我对算法、对数据结构的全新且深刻的认知。今后还是会在OJ和洛谷等平台上逛逛，毕竟谁能拒绝得了深入了解算法以及AC的快乐呢？

最后，再次感谢闫老师和群里的各位大佬！祝大家暑假愉快！绩点高高！