数算期末cheating paper

栈、队列与链表

1.单调栈

```
n = int(input())
stack=[] #单调栈，存储奶牛的高度
ans = 0
for _ in range(n):
    height = int(input())
    while stack and stack[-1]<= height :
        stack.pop()
    ans += len(stack)
    stack.append(height)
print(ans)

class Solution:
    def trap(self, height: List[int]) -> int:
        ans = 0
        stack = list()
        n = len(height)
        for i, h in enumerate(height):
            while stack and h > height[stack[-1]]:
                top = stack.pop()
                if not stack:
                    break
                left = stack[-1]
                currWidth = i - left - 1
                currHeight = min(height[left], height[i]) - height[top]
                ans += currWidth * currHeight
            stack.append(i)
        return ans
```

```
def mid_to_back(arr):
    precedence={"+":1,"-":1,"*":2,"/":2}
    output=[]
    operation=[]
    i=0
    while i<len(arr):
        if arr[i].isdigit():
            num=""
            while i < len(arr) and (arr[i].isdigit() or arr[i]=="."):
                num+=arr[i]
                i+=1
            output.append(num)
            continue
        elif arr[i] == "(":
            operation.append(arr[i])
            i+=1
```

```
        elif arr[i] == ")":
            while operation and operation[-1]!="(":
                output.append(operation.pop())
            operation.pop()
            i+=1
        else:
            while operation and operation[-1]!="(" and precedence.get(arr[i],0)
<=precedence.get(operation[-1],0):
                output.append(operation.pop())
            operation.append(arr[i])
            i+=1
    while operation:
        output.append(operation.pop())
    return " ".join(output)
n=int(input())
for _ in range(n):
    arr=input()
    print(mid_to_back(arr))
```

2.链表

快慢指针，在判断回文时有用

```
slow,fast=head,head
while fast.next and fast.next.next:
    slow=slow.next
    fast=fast.next.next
#slow此时是中偏左位置
slow=slow.next
```

```
def fan(head):
    pre,cur=None,head
    while cur:
        tmp=cur.next
        cur.next=pre
        pre=cur
        cur=tmp
    return pre

def fan(head):
    pre,nt=None,None
    while head!=None:
        nt=head.next
        head.next=pre
        head.last=nt#last表示上一个
        pre=head
        head=nt
    return pre

def detectCycle(head):
    if head is None or head.next is None or head.next.next is None:
```

```
        return None
    slow=head.next
    fast=head.next.next
    while slow!=fast:
        if fast.next is None or fast.next.next is None:
            return None
        slow=slow.next
        fast=fast.next.next
    fast=head
    while slow!=fast:
        slow=slow.next
        fast=fast.next
    return slow
```

树相关

1.各种建树

括号嵌套树：主要要关注的是栈在建树中的使用方法。栈最主要的用处就是处理这种一点一点建树的问题，相关例题还有文字生成树

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []
def parse_tree(s):
    stack=[]
    node=None
    for char in s:
        if char.isalpha():
            node=TreeNode(char)
            if stack:
                stack[-1].children.append(node)
        elif char=="(":
            if node:
                stack.append(node)
                node=None
        elif char==")":
            if stack:
                node=stack.pop()
    return node
def preorder(node):
    output=[node.value]
    for child in node.children:
        output.extend(preorder(child))
    return ''.join(output)
def postorder(node):
    output=[]
    for child in node.children:
        output.extend(postorder(child))
```

```
    output.append(node.value)
    return ''.join(output)
def main():
    s=input().strip()
    s="".join(s.split())
    root=parse_tree(s)
    if root:
        print(preorder(root))
        print(postorder(root))
if __name__ == '__main__':
    main()
```

文本二叉树

```
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None
n=int(input())
for _ in range(n):
    tree=[]
    while True:
        info=input()
        if info=='0':
            break
        k=info.count('-')
        tree.append((k,info[-1]))
    def buildTree(lst,index):
        layer,node=lst[index][0],TreeNode(lst[index][1])
        i=index+1
        if i < len(lst) and lst[i][0] == layer+1:
            if lst[i][1] == "*":
                node.left=None
                i+=1
            else:
                node.left,i= buildTree(lst,i)
         #对于这种先左后右的一定要把左边建完，更新好index之后再去管右边
        if i < len(lst) and lst[i][0] == layer+1:
            node.right,i=buildTree(lst,i)
        return node,i
    root,_ = buildTree(tree,0)
    def preorder(node):
        if node:
            print(node.val, end='')
            preorder(node.left)
            preorder(node.right)
    def inorder(node):
        if node:
            inorder(node.left)
            print(node.val, end='')
```

```
            inorder(node.right)
    def postorder(node):
        if node:
            postorder(node.left)
            postorder(node.right)
            print(node.val, end='')
    preorder(root)
    print()
    postorder(root)
    print()
    inorder(root)
    print()
    print()
```

模版类：前序+中序

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

def build_from_pre_in(preorder, inorder):
    if not preorder:
        return None
    root_val = preorder[0]
    root = TreeNode(root_val)
    idx = inorder.index(root_val)
    root.left = build_from_pre_in(preorder[1:1+idx], inorder[:idx])
    root.right = build_from_pre_in(preorder[1+idx:], inorder[idx+1:])
    return root
```

中+后

```
def build_from_in_post(inorder, postorder):
    if not postorder:
        return None
    root_val = postorder[-1]
    root = TreeNode(root_val)
    idx = inorder.index(root_val)
    root.left = build_from_in_post(inorder[:idx], postorder[:idx])
    root.right = build_from_in_post(inorder[idx+1:], postorder[idx:-1])
    return root
```

2.各种遍历（其实就是递归思想在树中的运用）

```
def preorder(node):
    if node:
        print(node.val, end=' ')
        preorder(node.left)
        preorder(node.right)

def inorder(node):
    if node:
        inorder(node.left)
        print(node.val, end=' ')
        inorder(node.right)

def postorder(node):
    if node:
        postorder(node.left)
        postorder(node.right)
        print(node.val, end=' ')
```

```
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') ->
'TreeNode':
        if not root or root == p or root == q:
            return root
        left=self.lowestCommonAncestor(root.left,p,q)
        right=self.lowestCommonAncestor(root.right,p,q)
        if not left: return right
        if not right: return left
        return root
```

3.BTS（二叉搜索树相关）

最重要的关于二叉搜索树的一点是二叉搜索树的中序遍历是sort的

```
class Solution:
    def inorder(self, node):
        if not node: return []
        return self.inorder(node.left) + [node.val] + self.inorder(node.right)
    def balanceBST(self, root: TreeNode) -> TreeNode:
        nums = self.inorder(root)
        def dfs(start, end):
            if start == end: return TreeNode(nums[start])
            if start > end: return None
            mid = (start + end) // 2
            root = TreeNode(nums[mid])
            root.left = dfs(start, mid - 1)
            root.right = dfs(mid + 1, end)
            return root
```

```
        return dfs(0, len(nums) - 1)
```

4.字典树（Trie）

```
class TrieNode:
    def __init__(self):
        self.child={}


class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, nums):
        curnode = self.root
        for x in nums:
            if x not in curnode.child:
                curnode.child[x] = TrieNode()
            curnode=curnode.child[x]

    def search(self, num):
        curnode = self.root
        for x in num:
            if x not in curnode.child:
                return 0
            curnode = curnode.child[x]
        return 1


t = int(input())
p = []
for _ in range(t):
    n = int(input())
    nums = []
    for _ in range(n):
        nums.append(str(input()))
    nums.sort(reverse=True)
    s = 0
    trie = Trie()
    for num in nums:
        s += trie.search(num)
        trie.insert(num)
    if s > 0:
        print('NO')
    else:
        print('YES')
```

图相关

1.递归/BFS/DFS

n皇后，最好的递归debug模版

```
class Solution:
    def solveNQueens(self, n: int) -> List[List[str]]:
        boards=[[False] * n for _ in range(n)]
        ans=[]
        def backstrack(row,path):
            if row ==n:
                ans.append(list(path))
                return
            for col in range(n):
                if is_safe(row,col,path):
                    boards[row][col]=True
                    path.append(col)
                    backstrack(row+1,path)
                    path.pop()
                    boards[row][col] = False
        def is_safe(row,col,path):
            for i in range(row):
                    if boards[i][col] or path[i] == col or abs(row - i) == abs(col -
path[i]):
                        return False
            return True
        def ans_change(ans):
            for solution in ans:
                for i in range(len(solution)):
                    solution[i]="."*(solution[i])+"Q"+"."*(n-solution[i]-1)
            return
        backstrack(0,[])
        ans_change(ans)
        return ans
```

warnsdroff优化

```
def degree(x,y,board):
    global n,di
    cnt=0
    for dx,dy in di:
        nx,ny=x+dx,y+dy
        if 0<=nx<n and 0<=ny<n and board[nx][ny]==-1:
            cnt+=1
    return cnt
def dfs(x,y,cnt):
    global di,board,n
    if cnt==n*n:
        return True
    next_move=[]
```

```
    for dx,dy in di:
        nx,ny=x+dx,y+dy
        if 0<=nx<n and 0<=ny<n and board[nx][ny]==-1:
            next_move.append((degree(nx,ny,board),nx,ny))
    next_move.sort()
    for _,nx,ny in next_move:
        board[nx][ny]=cnt
        if dfs(nx,ny,cnt+1):
            return True
        board[nx][ny]=-1
    return False
```

2.图相关的各种模版题目

最小树

```
class Solution:
    def minCostConnectPoints(self, points: List[List[int]]) -> int:
        # arr=[]
        # n=len(points)
        # # 构建边
        # for i in range(n):
        #     x1,y1=points[i]
        #     for j in range(i+1,n):
        #         x2,y2=points[j]
        #         arr.append([i,j,abs(x2-x1)+abs(y2-y1)])
        # # 排序
        # arr.sort(key=lambda x:x[2])
        # # 并查集
        # parent=list(range(n))
        # def find(x):
        #     if x!=parent[x]:
        #         parent[x]=find(parent[x])
        #     return parent[x]
        # # 构建最小生成树
        # edge=0
        # cost=0
        # for i,j,d in arr:
        #     a,b=find(i),find(j)
        #     if a!=b:
        #         parent[b]=a
        #         edge+=1
        #         cost+=d
        #     if edge==n-1:
        #         break
        # return cost
----#下面的最好，prim是一种很不错的解法，快而且简单
        n = len(points)
        visited = [False] * n           # 哪些点已经加入生成树
        min_heap = [(0, 0)]             # (边权重, 点编号)，从点 0 开始
```

```
        total_cost = 0                  # 总费用
        num_visited = 0                 # 已加入生成树的点数

        while num_visited < n:
            cost, u = heapq.heappop(min_heap)
            if visited[u]:
                continue
            visited[u] = True
            total_cost += cost
            num_visited += 1
            for v in range(n):
                if not visited[v]:
                    dist = abs(points[u][0] - points[v][0]) + abs(points[u][1] -
points[v][1])
                    heapq.heappush(min_heap, (dist, v))

        return total_cost
```

拓扑排序

```
from collections import defaultdict
import heapq   #如果不是字典序的话就考虑用普通的deque就好
n,l=map(int,input().split())
links=defaultdict(list)
degree=defaultdict(int)
for _ in range(l):
    a,b=map(int,input().split())
    links[a].append(b)
    degree[b]+=1
tuo=[]
queue=list(u for u in range(1,n+1) if degree[u]==0)
heapq.heapify(queue)
while queue:
    u=heapq.heappop(queue)
    tuo.append(u)
    for v in links[u]:
        degree[v]-=1
        if degree[v]==0:
            heapq.heappush(queue,v)
    if len(tuo)==n:
        break
for i in tuo:
    print("v"+str(i),end=" ")
```

djsktra，这里放一道很神人的题目。。

```
import math
import heapq
from collections import defaultdict

def dist(a, b):
```

```
    """欧几里得距离（单位：米）"""
    return math.hypot(a[0] - b[0], a[1] - b[1])

# 输入起点和终点
sx, sy, ex, ey = map(int, input().split())
home = (sx, sy)
school = (ex, ey)

# 所有地铁站点 & 邻接图
stations = {home, school}
graph = defaultdict(list)

# 地铁线路建图
while True:
    try:
        line = list(map(int, input().split()))
        if not line:
            break
        stops = []
        for i in range(0, len(line) - 1, 2):
            x, y = line[i], line[i + 1]
            if x == -1 and y == -1:
                break
            stops.append((x, y))
            stations.add((x, y))
        for i in range(len(stops) - 1):
            a, b = stops[i], stops[i + 1]
            t = dist(a, b) / 40000 * 60  # 地铁速度：40 km/h = 40000 m/h
            graph[a].append((b, t))
            graph[b].append((a, t))
    except EOFError:
        break

# 给所有站点之间建步行边（起点终点也算）
station_list = list(stations)
for i in range(len(station_list)):
    for j in range(i + 1, len(station_list)):
        a, b = station_list[i], station_list[j]
        t = dist(a, b) / 10000 * 60  # 步行速度：10 km/h = 10000 m/h
        graph[a].append((b, t))
        graph[b].append((a, t))

# Dijkstra 求最短路径
heap = [(0, home)]
best = {home: 0}
#在djsktra之中很多时候不能简单的按照bfs那样用in_queue，更多的时候需要考虑一下dp类似的思想
while heap:
    t, u = heapq.heappop(heap)
    if u == school:
        print(round(t))
        break
    if t > best[u]:
```

```
            continue
        for v, cost in graph[u]:
            new_time = t + cost
            if v not in best or new_time < best[v]:
                best[v] = new_time
                heapq.heappush(heap, (new_time, v))
```

三色染色法判断图中是否存在环

```
class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
        g = [[] for _ in range(numCourses)]
        for a, b in prerequisites:
            g[b].append(a)

        colors = [0] * numCourses
        # 返回 True 表示找到了环
        def dfs(x: int) -> bool:
            colors[x] = 1  # x 正在访问中
            for y in g[x]:
                if colors[y] == 1 or colors[y] == 0 and dfs(y):
                    return True  # 找到了环
            colors[x] = 2  # x 完全访问完毕
            return False  # 没有找到环

        for i, c in enumerate(colors):
            if c == 0 and dfs(i):
                return False  # 有环
        return True  # 没有环
```

最大流问题

```
from collections import deque, defaultdict

def bfs(rGraph, s, t, parent):
    visited = set()
    queue = deque([s])
    visited.add(s)
    while queue:
        u = queue.popleft()
        for v in rGraph[u]:
            if v not in visited and rGraph[u][v] > 0:
                queue.append(v)
                visited.add(v)
                parent[v] = u
                if v == t:
                    return True
    return False

def ford_fulkerson(graph, source, sink):
    rGraph = defaultdict(lambda: defaultdict(int))
```

```
    for u in graph:
        for v in graph[u]:
            rGraph[u][v] = graph[u][v]

    parent = {}
    max_flow = 0

    while bfs(rGraph, source, sink, parent):
        path_flow = float('inf')
        v = sink
        while v != source:
            u = parent[v]
            path_flow = min(path_flow, rGraph[u][v])
            v = u

        v = sink
        while v != source:
            u = parent[v]
            rGraph[u][v] -= path_flow
            rGraph[v][u] += path_flow
            v = u

        max_flow += path_flow

    return max_flow
```

其他的模板

1.二分查找

```
L,n,M=map(int,input().split())
rock=[0]
for _ in range(n):
    D=int(input())
    rock.append(D)
rock.append(L)
def check(x):
    num=0
    now=0
    for i in range(1,n+2):
        if rock[i]-now<x:
            num+=1
        else:
            now=rock[i]
    if num>M:
        return True
    else:
        return False
le,re=0,L+1
```

```
ans=-1
while le<re:
    mid=(le+re)//2
    if check(mid):
        re=mid
    else:
        ans=mid
        le=mid+1
print(ans)
```

2.各种的动态规划

```
#1状态机dp（也是多维动态规划）
#这个的适用场景其实非常明显，就是上一个取不取会影响到下一个能不能取
#这里就要提到在树上的递归，很多时候不需要用层序遍历把他变成本题的这种形式，因为可能不是完全二叉树，所以直接用一个函数在树上遍历就可以了，重点还是先遍历子，再回推父
def max_treasure_value(n, values):
    dp = [[0, 0] for _ in range(n + 1)]
    def dfs(i):
        if i > n:
            return
        left = 2 * i
        right = 2 * i + 1
        if left <= n:
            dfs(left)
        if right <= n:
            dfs(right)
        dp[i][0] = 0
        if left <= n:
            dp[i][0] += max(dp[left][0], dp[left][1])
        if right <= n:
            dp[i][0] += max(dp[right][0], dp[right][1])
        dp[i][1] = values[i - 1]
        if left <= n:
            dp[i][1] += dp[left][0]
        if right <= n:
            dp[i][1] += dp[right][0]
    dfs(1)
    return max(dp[1][0], dp[1][1])
n = int(input())
values = list(map(int, input().split()))
print(max_treasure_value(n, values))

#2图上的dp
R,C=map(int,input().split())
maps=[list(map(int,input().split())) for i in range(R)]
dire=[(0,1),(0,-1),(1,0),(-1,0)]
dp=[[-1]*C for _ in range(R)]
def dfs(x,y,h):
    if dp[x][y]!=-1:
        return dp[x][y]
```

```
    max_path=1
    for i in range(4):
        nx,ny=x+dire[i][0],y+dire[i][1]
        if 0<=nx<R and 0<=ny<C and maps[nx][ny]<h:
                max_path=max(max_path,dfs(nx,ny,maps[nx][ny])+1)
    dp[x][y]=max_path
    return dp[x][y]
ans=0
for i in range(R):
    for j in range(C):
        ans=max(ans,dfs(i,j,maps[i][j]))
print(ans)
```

3.各种排序

```
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
    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
    # 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
```

4.并查集

并查集最重要的应该是在最后不管什么都要回到root上，这样可以有效避免各种问题

```
from collections import defaultdict
n,m=map(int,input().split())
parent=list(range(n))
def find(x):
    if x!=parent[x]:
        parent[x]=find(parent[x])
```

```
    return parent[x]
for _ in range(m):
    a,b=map(int,input().split())
    i,j=find(a-1),find(b-1)
    if i!=j:
        parent[i]=j
result=defaultdict(int)
count=0
for i in range(n):
    root=find(i)
    result[root]+=1
    if i==root:
        count+=1
print(count)
for i in range(min(n,100)):
    root=find(i)
    print(result[root],end=" ")
```

KMP

```
def kmp(s1,s2):
    n,m=len(s1),len(s2)
    x,y=0,0
    nt=nextarray(s2,m)
    while x<n and y<m:
        if s1[x]==s2[y]:
            x+=1
            y+=1
        elif y==0:
            x+=1
        else:
            y=nt[y]
    return x-y if y==m else -1
def nextarray(s,m):
    if m==1:
        return [-1]
    nt=[0]*m
    nt[0],nt[1]=-1,0
    i,cn=2,0
    while i<m:
        if s[i-1]==s[cn]:
            cn+=1
            nt[i]=cn
            i+=1
        elif cn>0:
            cn=nt[cn]
        else:
            nt[i]=0
            i+=1
    return nt
```

全排序

```
from itertools import permutations
perm=permutations([1,2,3])
```

二次探查法

```
import sys
input = sys.stdin.read
data = input().split()
index = 0
n = int(data[index])
index += 1
m = int(data[index])
index += 1
num_list = [int(i) for i in data[index:index+n]]

mylist = [0.5] * m

def generate_result():
    for num in num_list:
        pos = num % m
        current = mylist[pos]
        if current == 0.5 or current == num:
            mylist[pos] = num
            yield pos
        else:
            sign = 1
            cnt = 1
            while True:
                now = pos + sign * (cnt ** 2)
                current = mylist[now % m]
                if current == 0.5 or current == num:
                    mylist[now % m] = num
                    yield now % m
                    break
                sign *= -1
                if sign == 1:
                    cnt += 1

result = generate_result()
print(*result)
```

其他的在输入输出时需要注意的地方

```
print("%.xf" % (time/n))
#输出保留x位有效数字

print
```