DSA Cheating Paper

一、机考部分

0.计概

(1)二分查找

- 要求操作对象是**有序**序列（此处记作lst）

```
import bisect
bisect.bisect_left(lst,x)
# 使用bisect_left查找插入点，若x∈lst，返回最左侧x的索引；否则返回最左侧的使x若插入后能位于其左侧的元素的当前索引。
bisect.bisect_right(lst,x)
# 使用bisect_right查找插入点，若x∈lst，返回最右侧x的索引；否则返回最右侧的使x若插入后能位于其右侧的元素的当前索引。
bisect.insort(lst,x)
# 使用insort插入元素，返回插入后的lst
```

(2)排列组合

```
from itertools import permutations,combinations
l = [1,2,3]
print(list(permutations(l))) # 输出：[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]
print(list(combinations(l,2))) # 输出：[(1, 2), (1, 3), (2, 3)]
```

(3)enumerate

- 返回索引和元素本身的数组

```
print(list(enumerate(['a','b','c']))) # 输出：[(1, 2), (1, 3), (2, 3)]
```

1.栈

(1)括号匹配

- 需要两个列表：stack和ans
- stack里存的是元素**下标**而非元素本身
- 注意讨论栈为空的情况

03704:括号匹配问题

样例：输入)(rttyy())sss)(，输出如下
)(rttyy())sss)(
? ?$

```
s = input()
print(s)
stack = []
ans = []
for i in range(len(s)):
    if s[i] == '(':
        stack.append(i)
        ans.append(' ')
    elif s[i] == ')':
        if stack == []: # 若栈为空，右括号一定无法匹配，且不能pop
            ans.append('?')
        else:
            stack.pop()
            ans.append(' ')
    else:
        ans.append(' ')
for j in stack: # 未匹配的左括号最后单独处理
    ans[j] = '$'
print(''.join(ans))
```

(2)Shunting Yard算法（中序转后序）

步骤：

- 初始化运算符栈（stack）和输出栈（ans）为空
- 从左到右遍历中缀表达式的每个符号
  - 如果是数字，则将其加入ans
  - 如果是'('，则将其推入运算符栈
  - 如果是运算符：
    - 如果stack为空，直接将当前运算符推入stack
    - 如果运算符的优先级大于stack[-1]，或者stack[-1]=='('，则将当前运算符推入stack（先用字典定义优先级：pre={'+':1,'-':1,'*':2,'/':2}
    - 否则，将stack.pop()添加到ans中，直到满足上述条件（或者stack为空），再将当前运算符推入stack
  - 如果是')'，则将stack.pop()添加到ans中，直到遇到'('，将'('弹出但不添加到ans中
- 如果还有剩余的运算符在stack中，将它们依次弹出并添加到ans中
- ans中的元素就是转换后的后缀表达式

(3)含括号表达式求值

- 求出括号内的值后，将其压入栈

20140:今日化学论文

把连续的x个字符串s记为[xs]，输入由小写英文字母、数字和[]组成的字符串，输出原始的字符串。
样例：输入[2b[3a]c]，输出baaacbaaac

```
s = input()
stack = []
```

```
for i in range(len(s)):
    stack.append(s[i])
    if stack[-1] == '[':
        stack.pop()
        helpstack = [] # 利用辅助栈求括号内的原始字符串，记得每次用前要清空
        while stack[-1] != '[':
            helpstack.append(stack.pop())
        stack.pop()
        numstr = ''
        while helpstack[-1] in '0123456789':
            numstr += str(helpstack.pop())
        helpstack = helpstack*int(numstr)
        while helpstack != []:
            stack.append(helpstack.pop())
print(''.join(stack))
```

(4)进制转换（n→m）

- 不断//m，余数入栈，商作为下一轮被除数
- 栈中数字依次出栈（倒着输出）

02734:十进制到八进制

```
n = int(input())
stack = []
if n == 0: # 输入为0时单独讨论
    stack = ['0']
while n > 0: # 进入下一轮的条件是大于0
    stack.append(str(n%8))
    n //= 8
stack.reverse()
print(''.join(stack))
```

(5)单调栈

- 栈内元素保持单调递增/递减的顺序
- 主要用途是寻找序列中某个元素左侧/右侧第一个比它大/小的元素

28203:【模板】单调栈

给出项数为n的整数数列a1...an，定义函数f(i)代表数列中第i个元素之后第一个大于ai的元素的下标。若不存在，则f(i)=0。试求出f(1)...f(n)。

```
n = int(input())
a = list(map(int,input().split()))
stack = []
for i in range(n):
    while stack and a[stack[-1]]<a[i]: # 注意pop前要检查栈是否非空
        a[stack.pop()] = i+1 # 原地修改，较为简洁
    stack.append(i) # stack存元素下标而非元素本身
for x in stack:
    a[x] = 0
print(*a)
```

(6)用栈实现递归

- 属于DFS，类似后面的回溯法，入栈相当于递归，出栈相当于回溯

02754:八皇后

八皇后（8*8棋盘，任意两个皇后均不能共行、共列、共斜线）问题一共有92个解，要求输出第b个解。
输入的第一行是测试数据组数n，后面n行是b。

```
def queen_stack(n):
    stack = [] # 用于保存状态的栈，栈中的元素是(row,queens)的tuple
    solutions = [] # 存储所有解决方案的列表
    stack.append((0, tuple())) # 初始状态为第一行，所有列都未放置皇后
    while stack:
        now_row,pos = stack.pop() # 从栈中取出当前处理的行数和已放置的皇后位置
        if now_row == n:
            solutions.append(pos)
        else:
            for col in range(n):
                if is_valid(now_row,col,pos):
                    stack.append((now_row+1,pos+(col,))) # 将新的合法状态压入栈
    return solutions[::-1] # 由于栈的LIFO特性，得到的solutions为倒序
def is_valid(row,col,queens): # 检查当前位置是否合法
    for r,c in enumerate(queens):
        if c==col or abs(row-r)==abs(col-c):
            return False
    return True
solutions = queen_stack(8)
n = int(input())
for _ in range(n):
    b = int(input())
    queen_string = ''.join(str(col+1) for col in solutions[b-1])
    print(queen_string)
```

(7)懒删除

- 删除仅仅是标记一个元素被删除，而不是整个清除它

22067:快速堆猪

输入中，push n表示叠上一头重量是n的猪；pop表示将猪堆顶的猪赶走；min表示问现在猪堆里最轻的猪多重（需输出答案）。

```
import heapq
from collections import defaultdict
out = defaultdict(int) # defaultdict用于记录删除的元素（查找时比list、set快）
pigs_heap = [] # heap用于确定最小的元素
pigs_stack = [] # stack用于确定最后的元素
while True:
    try:
        s = input()
    except EOFError:
        break
    if s == "pop":
        if pigs_stack:
            out[pigs_stack.pop()] += 1 # 代表删除了最后一个元素
    elif s == "min":
        if pigs_stack:
            while True: # 循环，如果最小的元素已经被删除，就寻找下一个最小的
                x = heapq.heappop(pigs_heap)
                if not out[x]: # 如果最小的元素还没有被删除
                    heapq.heappush(pigs_heap,x)
                    print(x)
                    break
                out[x] -= 1
    else:
        y = int(s.split()[1])
        pigs_stack.append(y)
        heapq.heappush(pigs_heap,y)
```

2.树

(1)二叉树（基础）

根据每个节点左右子树建树

设共有n个节点，且节点的值分别为1~n，依次输入每个节点的左右子节点，若没有则输入-1

```
class Node: # 定义节点，用class实现
    def __init__(self,value):
        self.value = value
        self.left = None
        self.right = None
n = int(input())
nodes = [node(_) for _ in range(1,n+1)]
for i in range(n):
    l,r = map(int,input().split())
    if l != -1: # 一定要先判断子节点是否存在
        nodes[i].left = nodes[l]
    if r != -1:
```

```
            nodes[i].right = nodes[r]
# 这一方法中，指针只能表示相邻两层之间的关系
```

根据前中/中后序序列建树

以前中序为例

- 前提是每个节点的值不同，否则不方便用index()

```
def build_tree(preorder, inorder):
    if not preorder or not inorder: # 先判断是否为空树
        return None
    root_value = preorder[0]
    root = Node(root_value)
    root_index = inorder.index(root_value)
    root.left = build_tree(preorder[1:root_index+1], inorder[:root_index]) #递归
    root.right = build_tree(preorder[root_index_inorder+1:], inorder[root_index_inorder+1:])
    return root
```

根据扩展前/后序序列建树

以前序为例，设preorder中空的子节点用'.'表示

```
def build_tree(preorder):
    if not preorder: # 先判断是否为空树
        return None
    value = preorder.pop() # 倒序处理（若给后序，则正序处理）
    if value == '.':
        return None
    root = Node(value)
    root.left = build_tree(preorder) # 递归是树部分的关键思想
    root.right = build_tree(preorder)
    return root
```

计算深度

- 高度=深度-1（空树深度为0，高度为-1）

```
def depth(root):
    if root is None: # 先判断是否为空树
        return 0 # 若计算高度，则return -1
    else:
        left_depth = depth(root.left) # 递归
        right_depth = depth(root.right)
        return max(left_depth, right_depth)+1
```

计算叶节点数目

```
def count_leaves(root):
    if root is None: # 先判断是否为空树
        return 0
    if root.left is None and root.right is None:
        return 1
    return count_leaves(root.left)+count_leaves(root.right)
```

前/中/后序遍历

- DFS
- 特别地，BST的中序遍历就是从小到大排列
  以后序为例（前序：C→A→B，中序：A→C→B）

```
def post_order_traversal(root):
    output = []
    if root.left: # part A
        # 先判断子节点是否存在
        output.extend(post_order_traversal(root.left))
        # 是extend而不是append
    if root.right: # part B
        output.extend(post_order_traversal(root.right))
    output.append(root.value) # part C
    return output
```

层次遍历

- BFS

```
from collections import deque
def level_order_traversal(root):
    q = deque()
    q.append(root)
    output = []
    while q:
        node = q.popleft()
        output.append(node.value)
        if node.left: # 仍然是先判断子节点是否存在
            q.append(node.left)
        if node.right:
            q.append(node.right)
    return output
```

(2)Huffman编码树

- 实际做题时用heapq实现，合并操作等价于heappop出两个最小元素，取和后再heappush入堆

18164:剪绳子

每次剪断绳子时需要的开销是此段绳子的长度，输入将绳子剪成的段数n和准备剪成的各段绳子的长度，输出最小开销。

- 看作拼绳子

```
import heapq
n = int(input())
a = list(map(int,input().split()))
heapq.heapify(a)
ans = 0
for i in range(n-1):
    x = heapq.heappop(a)
    y = heapq.heappop(a)
    z = x+y
    heapq.heappush(a,z)
    ans += z
print(ans)
```

(3)BST

根据数字列表建树

- 每次从列表中取出一个数字插入BST

```
def insert(root,num):
    if root is None: # 先判断是否为空树
        return node(num)
    if num < root.value:
        root.left = insert(root.left,num)
    elif num > root.value:
        root.right = insert(root.right,num)
    return root
```

(4)多叉树

实现

- 可用class或dict（以dict为例，class见“27928:遍历树”）
  node_list为所有节点值的列表

```
tree = {i:[] for i in node_list}
# []中储存i所有子节点的值
```

前序/后序/层次遍历

- 类似二叉树，略

27928:遍历树（按大小的递归遍历）

遍历规则：遍历到每个节点（值为互不相同的正整数）时，按照该节点和所有子节点的值从小到大进行遍历。
输入的第一行为节点个数n，接下来的n行中第一个数是此节点的值，之后的数分别表示其所有子节点的值；分行输出遍历结果。

```
class Node:
    def __init__(self,value):
        self.value = value
        self.children = []
        # self.parent = None（有些题中需要，便于确定节点归属）
def traverse_print(root,nodes):
    if root.children == []: # 同理，先判断子节点是否存在
        print(root.value)
        return
    to_sort = {root.value:root} # 此处利用value来查找Node，而不是用指针（因为多叉树的指针往往只能表示相邻两层之间的关系）
    for child in root.children:
        to_sort[child] = nodes[child]
    for value in sorted(to_sort.keys()):
        if value in root.children:
            traverse_print(to_sort[value], nodes) # 递归
        else:
            print(root.value)
n = int(input())
nodes = {}
children_list = [] # 用来找根节点
for i in range(n):
    l = list(map(int,input().split()))
    nodes[l[0]] = Node(l[0])
    for child_value in l[1:]:
        nodes[l[0]].children.append(child_value)
        children_list.append(child_value)
root = nodes[[value for value in nodes.keys() if value not in children_list][0]]
traverse_print(root,nodes)
```

“左儿子右兄弟”转换

设输入为扩展二叉树的前序遍历，要转换为n叉树

```
nodes = {} # 用于存储n叉树的所有节点
def bi_to_n(node):
    if node.left:
        if node.left.value != '*':
            new_node = Node(node.left.value)
            nodes[node.left] = new_node
            nodes[node].child.append(new_node)
            new_node.parent = nodes[node]
            bi_to_n(node.left) # 递归
    if node.right:
        if node.right.value != '*':
            new_node = Node(node.right.value)
```

```
            nodes[node.right] = new_node
            nodes[node].parent.child.append(new_node)
            new_node.parent = nodes[node].parent
            bi_to_n(node.right)
```

(5)Trie

构建

```
class Node:
    def __init__(self,value):
        self.value = value
        self.children = []
def insert(root,num):
    node = root
    for digit in num:
        if digit not in node.children:
            node.children[digit] = Node()
        node=node.children[digit]
        node.cnt+=1
```

3.并查集

- 实质上也是树，元素的parent为其**父节点**，find所得元素为其所在集合（树）的**根节点**
- 有几个互不重合的集合，就有几棵独立的树

(1)列表实现parent

- 若parent[x] == y，则说明y是x所在集合的代表元素

```
parent = list(range(n+1))
# 将列表长度设为n+1是为了使元素本身与下标能够对应
```

(2)查询操作

- 目的是找到x所在集合的代表元素

```
def find(x):
    if parent[x] == x: # 如果x所在集合的parent就是x自身
        return x # 那么就用x代表这一集合
    else: # 递归，直到找到x所在集合的代表
        return find(parent[x])
```

(3)合并操作

- 目的是将y所在集合归入x所在集合

```
def union(self,x,y):
    x_rep,y_rep = find(x),find(y)
    if x_rep != y_rep:
        parent[y_rep] = x_rep
```

(4)rank优化

- rank表示代表某集合的树的深度
- 引入rank可保证合并后新树的深度最小

```
rank = [1]*n
# 以下是有rank时的合并操作
def union(self,x,y):
    x_rep,y_rep = find(x),find(y)
    if rank[x_rep] > rank[y_rep]:
        parent[y_rep] = x_rep
    elif rank[x_rep] < rank[y_rep]:
        parent[x_rep] = y_rep
    else:
        parent[y_rep] = x_rep
        rank[x_rep] += 1
```

4.图

(1)图的实现

- 通常用dict套list（有权值时为dict套dict）
- dict的键为各顶点，值为存储相应顶点所连顶点的list（或键为相应顶点所连顶点，值为相应边权值的dict）

(2)DFS

02386:Lake Counting（连通区域问题）

输入n行m列由'.'和'W'构成的矩阵，求'W'连通区域的个数

```
import sys
sys.setrecursionlimit(20000) # 防止递归爆栈
dx = [-1,-1,-1,0,0,1,1,1]
dy = [-1,0,1,-1,1,-1,0,1]
def dfs(x,y):
    field[x][y] = '.' # 标记，避免再次访问
    for i in range(8):
        nx,ny = x+dx[i],y+dy[i]
        if 0<=nx<n and 0<=ny<m and field[nx][ny]=='W': # 注意判断是否越界
            dfs(nx,ny) # DFS需递归
n,m = map(int,input().split())
field = [list(input()) for _ in range(n)]
cnt = 0
for i in range(n):
    for j in range(m):
```

```
        if field[i][j] == 'W':
            dfs(i,j)
            cnt += 1
print(cnt)
```

01321:棋盘问题（回溯法）

每组数据的第一行n(n<=8)、k表示将在一个n*n的矩阵内描述棋盘，以及摆放k个棋子；随后的n行描述了棋盘的形状，'#'表示棋盘区域，'.'表示空白区域。要求任意两个棋子不能放在棋盘中的同一行或同一列，求所有可行的摆放方案数。

- 回溯法就是“走不通就退回再走”

```
chess = [['' for _ in range(8)] for _ in range(8)]
def dfs(now_row,cnt):
    global ans
    if cnt==k:
        ans += 1
        return
    if now_row==n:
        return # 走不通就退回
    for i in range(now_row,n): # 一行一行地找，当在某一行上找到一个可放入的'#'后，就开始找下一行的'#'，如果下一行没有，就从再下一行找
        for j in range(n):
            if chess[i][j]='#' and not col_occupied[j]:
                col_occupied[j] = True
                dfs(i+1,cnt+1)
                col_occupied[j] = False # 若想在矩阵中寻找多条路径，访问完某点后要将其状态改回来
while True:
    n,k = map(int,input().split())
    if n==-1 and k==-1:
        break
    for i in range(n):
        chess[i] = list(input())
    col_occupied = [False]*8
    ans = 0
    dfs(0,0)
    print(ans)
```

(3)BFS

04115:鸣人和佐助（基于矩阵的BFS）

输入M行N列的地图（@代表鸣人，+代表佐助，*代表通路，#代表大蛇丸的手下）和鸣人初始的查克拉数量T（每一个查克拉可以打败一个大蛇丸的手下）。鸣人可以往上下左右四个方向移动，每移动一单位距离需要花费一单位时间。求鸣人追上佐助最少需要花费的时间（追不上则输出-1）。

- 本题的vis需要维护经过时的最大查克拉数t，只有t大于T值时候才能通过

```
from collections import deque
M,N,T = map(int,input().split())
graph = [list(input()) for i in range(M)]
```

```
dir = [(0,1),(1,0),(-1,0),(0,-1)]
for i in range(M): # 查找起点
    for j in range(N):
        if graph[i][j] == '@':
            start = (i,j)
def bfs(): # BFS也可以不定义函数直接写，此处是为了方便追不上时直接print(-1)
    q = deque([start+(T,0)])
    vis = [[-1]*N for i in range(M)] # 注意特殊的vis用法（维护t）
    vis[start[0]][start[1]] = T
    while q:
        x,y,t,time = q.popleft()
        time += 1
        for dx,dy in dir:
            if 0<=x+dx<M and 0<=y+dy<N: # 同样也要判断是否越界
                if graph[x+dx][y+dy]=='*' and t>vis[x+dx][y+dy]:
                    vis[x+dx][y+dy] = t
                    q.append((x+dx,y+dy,t,time))
                elif graph[x+dx][y+dy]=='#' and t>0 and t-1>vis[x+dx][y+dy]:
                    vis[x+dx][y+dy] = t-1
                    q.append((x+dx,y+dy,t-1,time))
                elif graph[x+dx][y+dy]=='+':
                    return time
    return -1
print(bfs())
```

(4)23163:判断无向图是否连通有无回路

- 注意是**无向图**
  输入第一行为顶点数n和边数m，接下来m行为u和v，表示顶点u和v之间有边。要求第一行输出“connected:yes/no"，第二行输出“loop:yes/no"。

```
n,m = map(int,input().split())
graph = [[] for _ in range(n)]
for _ in range(m):
    u,v = map(int,input().split())
    graph[u].append(v)
    graph[v].append(u)
def is_connected(graph):
    n = len(graph)
    vis = [False for _ in range(n)]
    cnt = 0
    def dfs(u):
        global cnt
        vis[u] = True
        cnt += 1
        for v in graph[u]:
            if not vis[v]:
                dfs(v)
    dfs(0)
    return cnt==n # 能从一个顶点出发搜索到其他顶点，说明连通
def has_loop(graph):
    n = len(graph)
```

```
    vis = [False for _ in range(n)]
    def dfs(u,x):
        vis[u] = True
        for v in graph[u]:
            if vis[v]==True: # 能从一个顶点出发搜索回到自身，说明有环
                if v!=x:
                    return True
            else:
                if dfs(v,u):
                    return True
        return False
    for i in range(n):
        if not vis[i]:
            if dfs(i,-1):
                return True
    return False
print('connected:yes' if is_connected(graph) else 'connected:no')
print('loop:yes' if has_loop(graph) else 'loop:no')
```

(5)拓扑排序

- 可判断**有向图**是否存在环
- 本质上是加了条件判断的BFS
  此处graph是dict套list的有向图

```
from collections import deque,defaultdict
def topological_sort(graph):
    indegree = defaultdict(int)
    order = []
    vis = set()
    for u in graph: # 统计各顶点入度
        for v in graph[u]:
            indegree[v] += 1
    q = deque()
    for u in graph:
        if indegree[u] == 0:
            q.append(u)
    while q:
        u = q.popleft()
        order.append(u)
        vis.add(u)
        for v in graph[u]:
            indegree[v] -= 1
            if indegree[v] == 0 and v not in vis:
                q.append(v)
    if len(order) == len(graph):
        return order
    else: # 说明存在环
        return None
```

(6)最短路径（Dijkstra算法）

- 本质上是元素在队列中按**总距离**排序的BFS（一般的BFS按**步数**排序）
  此处graph用dict套list表示

```
import heapq
def dijkstra(start,end):
    q = [(0,start,[start])]
    vis = set()
    while q:
        (distance,u,path) = heappop(q) # q中元素自动按distance排序，先取出distance小的
        if u in vis:
            continue
        vis.add(u)
        if u == end:
            return (distance,path) # 可以记录并返回路径
        for v in graph[u]:
            if v not in vis:
                heappush(q,(distance+graph[u][v],v,path+[v]))
```

(7)最小生成树（Prim算法）

- 本质上是元素在队列中按**某一步**距离排序的BFS
  此处graph用dict套list表示

```
import heapq
vis = [False]*n # vis可用list（因为最小生成树有且仅有n个顶点），比set快
q = [(0,0)]
ans = 0
while q:
    distance,u = heappop(q) # 贪心思想，通过堆找到下一步可以走的边中权值最小的
    if vis[u]:
        continue
    ans += distance # 对于某一顶点，最先pop出来的distance一定是最小的
    vis[u] = True
    for v in graph[u]:
        if not vis[v]:
            heappush(q,(graph[u][v],v))
print(ans) # 返回最小生成树中所有边权值（距离）之和
```