栈、队列

！调度场算法

shunting yard -->典型 中序表达式转为后序表达式 主要思想是使用两个栈（运算符栈和输出栈）来处理表达式的符号。算法按照运算符的优先级和结合性，将符号逐个处理并放置到正确的位置。最终，输出栈中的元素就是转换后的后缀表达式。

中序表达式转后序表达式

```
def infix_to_postfix(expression):
    precedence = {'+':1, '-':1, '*':2, '/':2}
    stack = []
    postfix = []
    number = ''
    for char in expression:
        if char.isnumeric() or char == '.':
            number += char
        else:
            if number:
                num = float(number)
                postfix.append(int(num) if num.is_integer() else num)
                number = ''
            if char in '+-*/':
                while stack and stack[-1] in '+-*/' and precedence[char] <=
precedence[stack[-1]]:
                    postfix.append(stack.pop())
                stack.append(char)
            elif char == '(':
                stack.append(char)
            elif char == ')':
                while stack and stack[-1] != '(':
                    postfix.append(stack.pop())
                stack.pop()
    if number:
        num = float(number)
        postfix.append(int(num) if num.is_integer() else num)
    while stack:
        postfix.append(stack.pop())
    return ' '.join(str(x) for x in postfix)
```

！重点 单调栈

八皇后

（dfs回溯实现或stack迭代实现）

dfs方法

```
def solve_n_queens(n):
    solutions = []  # 存储所有解决方案的列表
    queens = [-1] * n  # 存储每一行皇后所在的列数
    def backtrack(row):
        if row == n:  # 找到一个合法解决方案
            solutions.append(queens.copy())
        else:
            for col in range(n):
                if is_valid(row, col):  # 检查当前位置是否合法
                    queens[row] = col  # 在当前行放置皇后
                    backtrack(row + 1)  # 递归处理下一行
                    queens[row] = -1  # 回溯，撤销当前行的选择
    def is_valid(row, col):
        for r in range(row):
            if queens[r] == col or abs(row - r) == abs(col - queens[r]):
                return False
        return True
    backtrack(0)  # 从第一行开始回溯
    return solutions
def get_queen_string(b):
    solutions = solve_n_queens(8)
    if b > len(solutions):
        return None
    queen_string = ''.join(str(col + 1) for col in solutions[b - 1])
    return queen_string
```

stack迭代方法

```
def solve_n_queens(n):
    stack = []  # 用于保存状态的栈
    solutions = []  # 存储所有解决方案的列表
    stack.append((0, [-1] * n))  # 初始状态为第一行，所有列都未放置皇后
    while stack:
        row, queens = stack.pop()
        if row == n:  # 找到一个合法解决方案
            solutions.append(queens.copy())
        else:
            for col in range(n):
                if is_valid(row, col, queens):  # 检查当前位置是否合法
                    new_queens = queens.copy()
                    new_queens[row] = col  # 在当前行放置皇后
                    stack.append((row + 1, new_queens))  # 推进到下一行
    return solutions
def is_valid(row, col, queens):
```

合法出栈序列

给定一个由大小写字母和数字构成的，没有重复字符的长度不超过62的字符串x，现在要将该字符串的字符依次压入栈中，然后再全部弹出。 要求左边的字符一定比右边的字符先入栈，出栈顺序无要求。 再给定若干字符串，对每个字符串，判断其是否是可能的x中的字符的出栈序列

```
def isPopSeq(s1,s2):#判断s2是不是s1经出入栈得到的出栈序列
    stack = []
    if len(s1) != len(s2):
        return False
    else:
        L = len(s1)
        stack.append(s1[0])
        p1,p2 = 1,0
        while p1 < L:
            if len(stack) > 0 and stack[-1] == s2[p2]:
                stack.pop()
                p2 += 1
            else:
                stack.append(s1[p1])
                p1 += 1
        return "".join(stack[::-1]) == s2[p2:]
```

约瑟夫

（双端队列法，或双向链表）

埃拉托斯特尼筛法

可以先找到n可能的最大值，打表防超时，可用math.sqrt

```
def prime_sieve(n):
    sieve=[True]*(n+1)
    sieve[0]=sieve[1]=False
    for i in range(2,int(n**0.5)+1):
        if sieve[i]:
            sieve[i*i:n+1:i]=[False*len(range(i*i,n+1,i))]
    return [i for i in range(n+1) if sieve[i]]
```

汉诺塔问题

```
def hanoi(n,a,b,c):
    if n==1:
        print(f'1:{a}->{c}')
    elif n>1:
        hanoi(n-1,a,c,b)
        print(f'{n}:{a}->{c}')
        hanoi(n-1,b,a,c)
n,a,b,c=input().split()
```

```
n=int(n)
hanoi(n,a,b,c)
```

归并排序

```
def merge(left,right):
    merged=[]
    inv_count=0
    i=j=0
    while i<len(left) and j<len(right):
        if left[i]<=right[j]:
            merged.append(left[i])
            i+=1
        else:
            merged.append(right[j])
            j+=1
            inv_count+=len(left)-i
    merged+=left[i:]
    merged+=right[j:]
    return merged,inv_count
def merge_sort(lst):
    if len(lst)<=1:
        return lst,0
    middle=len(lst)//2
    left,inv_left=merge_sort(lst[:middle])
    right,inv_right=merge_sort(lst[middle:])
    merged,inv_merged=merge(left,right)
    return merged,inv_left+inv_right+inv_merged
```

递归、动规

数字三角形的记忆递归型动规程序

```
def MaxSum(i,j):
    if i == n-1:
        return D[i][j]
    if maxSum[i][j] != -1:
        return maxSum[i][j]
    x = MaxSum(i+1,j)
    y = MaxSum(i+1,j+1)
    maxSum[i][j] = max(x,y) + D[i][j]
    return maxSum[i][j]
```

递推程序 自下至上递堆

```
def main():
    for i in range(n):
        lst = list(map(int,input().split()))
        D.append(lst)
    for i in range(n):
        maxSum[n-1][i] = D[n-1][i]
    for i in range(n-2,-1,-1):
        for j in range(0,i+1):
            maxSum[i][j] = max(maxSum[i+1][j],maxSum[i+1][j+1]) + D[i][j]
```

树

根据后序表达式建立队列表达式（解析树）

```
def build_tree(postfix):
    stack = []
    for char in postfix:
        node = TreeNode(char)
        if char.isupper():
            node.right = stack.pop()
            node.left = stack.pop()
        stack.append(node)
    return stack[0]
def level_order_traversal(root):
    queue = [root]
    traversal = []
    while queue:
        node = queue.pop(0)
        traversal.append(node.value)
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
    return traversal
```

括号嵌套树

```
def parse_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)
            if stack:  # 如果栈不为空，把节点作为子节点加入到栈顶节点的子节点列表中
                stack[-1].children.append(node)
        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 = stack.pop()  # 弹出当前节点
    return node  # 根节点
```

扩展二叉树（嵌套树）

```
def build_tree(preorder):
    if not preorder or preorder[0] == '.':
        return None, preorder[1:]
    root = preorder[0]
    left, preorder = build_tree(preorder[1:])
    right, preorder = build_tree(preorder)
    return (root, left, right), preorder
```

二叉搜索树的遍历 **注意二叉搜索树的性质—中序遍历为有序表**

AVL树最多有几层

```
from functools import lru_cache
@lru_cache(maxsize=None)
def min_nodes(h):
 if h == 0: return 0
 if h == 1: return 1
 return min_nodes(h-1) + min_nodes(h-2) + 1

   def max_height(n):
    h = 0
    while min_nodes(h) <= n:
     h += 1
 return h - 1

   n = int(input())
   print(max_height(n))
```

图

环

bfs拓扑

```python
def topological_sort(graph):
    indegree = defaultdict(int)
    result = []
    queue = deque()
    for u in graph:
        for v in graph[u]:
            indegree[v] += 1
    for u in graph:
        if indegree[u] == 0:
            queue.append(u)
    while queue:
        u = queue.popleft()
        result.append(u)
        for v in graph[u]:
            indegree[v] -= 1
            if indegree[v] == 0:
                queue.append(v)
    if len(result) == len(graph):
        return result
    else:
        return None
```

dfs有向图判断环

```python
def has_cycle(n,edges):
    graph=[[] for _ in range(n)]
    for u,v in edges:
        graph[u].append(v)
    color=[0]*n
    def dfs(node):
        if color[node]==1:
            return True
        if color[node]==2:
            return False
        color[node]=1
        for neighbor in graph[node]:
            if dfs(neighbor):
                return True
        color[node]=2
        return False
```

强连通分量 Kosaraju

```python
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)#将相邻的顶点相邻存储
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):
    stack = []
    visited = [False] * len(graph)
    for node in range(len(graph)):
        if not visited[node]:
            dfs1(graph, node, visited, stack)
    transposed_graph = [[] for _ in range(len(graph))]
    for node in range(len(graph)):#取反
        for neighbor in graph[node]:
            transposed_graph[neighbor].append(node)
    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
```

最短路径

dijkstra

```
def dijkstra(graph, start):
    pq = []
    start.distance = 0
    heapq.heappush(pq, (0, start))
    visited = set()
    while pq:
        currentDist, currentVert = heapq.heappop(pq)    # 当一个顶点的最短路径确定（也就是这个顶点从优先队列中被弹出时），它的最短路径不会再改变。
        if currentVert in visited:
            continue
        visited.add(currentVert)
        for nextVert in currentVert.getConnections():
            newDist = currentDist + currentVert.getWeight(nextVert)
            if newDist < nextVert.distance:#初始为inf，松弛
                nextVert.distance = newDist
                nextVert.pred = currentVert
                heapq.heappush(pq, (newDist, nextVert))
```

ROADS 经典dijkstra

以下每行 R 通过指定由单个空字符分隔的整数 S、D、L 和 T 来描述一条道路： S 是源城市 D 是目的地城市 L为道路长度 T为通行费（以硬币数量表示） 请注意，不同的道路可能具有相同的始发城市和目的地城市。 输出的第一行也是唯一的一行应包含从城市 1 到城市 N 的最短路径的总长度，其总通行费小于或等于 K 个硬币。 如果此类路径不存在，则只应将数字 -1 写入输出。

```
import heapq
def dijkstra(graph):
    pq=[]
    heapq.heapify(pq)
    heapq.heappush(pq,(0,0,1,0))#length,cost,end,step
    while pq!=[]:
        l,c,cur,d=heapq.heappop(pq)
        if cur==n:
            return l
        for l1,c1,e1 in graph[cur]:
            if c+c1<=k and d+1<n:
                heapq.heappush(pq,(l1+l,c+c1,e1,d+1))
    return -1
k,n,r=[int(input()) for _ in range(3)]
graph={i:[] for i in range(1,n+1)}
for _ in range(r):
    s,e,l,c=map(int,input().split())
    graph[s].append([l,c,e])
print(dijkstra(graph))
```

最小生成树

prim

```
def prim(graph, start):
    pq = []
    start.distance = 0
    heapq.heappush(pq, (0, start))
    visited = set()
    while pq:
        currentDist, currentVert = heapq.heappop(pq)
        if currentVert in visited:
            continue
        visited.add(currentVert)
        for nextVert in currentVert.getConnections():
            weight = currentVert.getWeight(nextVert)
            if nextVert not in visited and weight < nextVert.distance:
                nextVert.distance = weight
                nextVert.pred = currentVert
                heapq.heappush(pq, (weight, nextVert))
```

kruskal and DisjointSet

```
class DisjointSet:
    def __init__(self, num_vertices):
        self.parent = list(range(num_vertices))
        self.rank = [0] * 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
def kruskal(graph):
    num_vertices = len(graph)
    edges = []
    for i in range(num_vertices):
        for j in range(i + 1, num_vertices):
            if graph[i][j] != 0:
                edges.append((i, j, graph[i][j]))
    # 按照权重排序
    edges.sort(key=lambda x: x[2])
    # 初始化并查集
    disjoint_set = DisjointSet(num_vertices)
    minimum_spanning_tree = []
    for edge in edges:
        u, v, weight = edge
        if disjoint_set.find(u) != disjoint_set.find(v):
            disjoint_set.union(u, v)
            minimum_spanning_tree.append((u, v, weight))
    return minimum_spanning_tree
```