排序、栈、队列

逆波兰表达式求值

```
stack=[]
for t in s:
    if t in '+-*/':
        b,a=stack.pop(),stack.pop()
        stack.append(str(eval(a+t+b)))
    else:
        stack.append(t)
print(f'{float(stack[0]):.6f}')
```

中序表达式转后序表达式

```
pre={'+':1,'-':1,'*':2,'/':2}
for _ in range(int(input())):
    expr=input()
    ans=[]; ops=[]
    for char in expr:
        if char.isdigit() or char=='.':
            ans.append(char)
        elif char=='(':
            ops.append(char)
        elif char==')':
            while ops and ops[-1]!='(':
                ans.append(ops.pop())
            ops.pop()
        else:
            while ops and ops[-1]!='(' and pre[ops[-1]]>=pre[char]:
                ans.append(ops.pop())
            ops.append(char)
    while ops:
        ans.append(ops.pop())
    print(''.join(ans))
```

最大全0子矩阵

```
for row in ma:
    stack=[]
    for i in range(n):
        h[i]=h[i]+1 if row[i]==0 else 0
        while stack and h[stack[-1]]>h[i]:
            y=h[stack.pop()]
            w=i if not stack else i-stack[-1]-1
            ans=max(ans,y*w)
        stack.append(i)
    while stack:
        y=h[stack.pop()]
        w=n if not stack else n-stack[-1]-1
        ans=max(ans,y*w)
print(ans)
```

求逆序对数

```
from bisect import *
a=[]
rev=0
for _ in range(n):
    num=int(input())
    rev+=bisect_left(a,num)
    insort_left(a,num)
ans=n*(n-1)//2-rev
```

```
def merge_sort(a):
    if len(a)<=1:
        return a,0
    mid=len(a)//2
    l,l_cnt=merge_sort(a[:mid])
    r,r_cnt=merge_sort(a[mid:])
    merged,merge_cnt=merge(l,r)
    return merged,l_cnt+r_cnt+merge_cnt
def merge(l,r):
    merged=[]
    l_idx,r_idx=0,0
    inverse_cnt=0
    while l_idx<len(l) and r_idx<len(r):
        if l[l_idx]<=r[r_idx]:
            merged.append(l[l_idx])
            l_idx+=1
        else:
            merged.append(r[r_idx])
            r_idx+=1
            inverse_cnt+=len(l)-l_idx
    merged.extend(l[l_idx:])
    merged.extend(r[r_idx:])
    return merged,inverse_cnt
```

树

根据前中序得后序、根据中后序得前序

```
def postorder(preorder,inorder):
    if not preorder:
        return ''
    root=preorder[0]
    idx=inorder.index(root)
    left=postorder(preorder[1:idx+1],inorder[:idx])
    right=postorder(preorder[idx+1:],inorder[idx+1:])
    return left+right+root
```

```
def preorder(inorder,postorder):
    if not inorder:
        return ''
    root=postorder[-1]
    idx=inorder.index(root)
    left=preorder(inorder[:idx],postorder[:idx])
    right=preorder(inorder[idx+1:],postorder[idx:-1])
    return root+left+right
```

层次遍历

```
from collections import deque
def levelorder(root):
    if not root:
        return ""
    q=deque([root])
    res=""
    while q:
        node=q.popleft()
        res+=node.val
        if node.left:
            q.append(node.left)
        if node.right:
            q.append(node.right)
    return res
```

解析括号嵌套表达式

```
def parse(s):
    node=Node(s[0])
    if len(s)==1:
        return node
    s=s[2:-1]; t=0; last=-1
    for i in range(len(s)):
        if s[i]=='(': t+=1
        elif s[i]==')': t-=1
        elif s[i]==',' and t==0:
            node.children.append(parse(s[last+1:i]))
            last=i
    node.children.append(parse(s[last+1:]))
    return node
```

二叉搜索树的构建

```
def insert(root,num):
    if not root:
        return Node(num)
    if num<root.val:
        root.left=insert(root.left,num)
    else:
        root.right=insert(root.right,num)
    return root
```

并查集

```
class UnionFind:
    def __init__(self,n):
        self.p=list(range(n))
        self.h=[0]*n
    def find(self,x):
        if self.p[x]!=x:
            self.p[x]=self.find(self.p[x])
        return self.p[x]
    def union(self,x,y):
        rootx=self.find(x)
        rooty=self.find(y)
        if rootx!=rooty:
            if self.h[rootx]<self.h[rooty]:
                self.p[rootx]=rooty
            elif self.h[rootx]>self.h[rooty]:
                self.p[rooty]=rootx
            else:
                self.p[rooty]=rootx
                self.h[rootx]+=1
```

字典树的构建

```
def insert(root,num):
    node=root
    for digit in num:
        if digit not in node.children:
            node.children[digit]=TrieNode()
        node=node.children[digit]
        node.cnt+=1
```

图

bfs

```
from collections import deque
def bfs(graph, start_node):
    queue = deque([start_node])
    visited = set()
    visited.add(start_node)
    while queue:
        current_node = queue.popleft()
        for neighbor in graph[current_node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
```

棋盘问题（回溯法）

```
def dfs(row, k):
    if k == 0:
        return 1
    if row == n:
        return 0
    count = 0
    for col in range(n):
        if board[row][col] == '#' and not col_occupied[col]:
            col_occupied[col] = True
            count += dfs(row + 1, k - 1)
            col_occupied[col] = False
    count += dfs(row + 1, k)
    return count
col_occupied = [False] * n
print(dfs(0, k))
```

dijkstra

```
# 1.使用vis集合
def dijkstra(start,end):
    heap=[(0,start,[start])]
    vis=set()
    while heap:
        (cost,u,path)=heappop(heap)
        if u in vis: continue
        vis.add(u)
        if u==end: return (cost,path)
        for v in graph[u]:
            if v not in vis:
                heappush(heap,(cost+graph[u][v],v,path+[v]))
# 2.使用dist数组
import heapq
def dijkstra(graph, start):
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    priority_queue = [(0, start)]
    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)
        if current_distance > distances[current_node]:
            continue
        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))
    return distances
```

kruskal

```
uf=UnionFind(n)
edges.sort()
ans=0
for w,u,v in edges:
    if uf.union(u,v):
        ans+=w
print(ans)
```

prim

```
vis=[0]*n
q=[(0,0)]
ans=0
while q:
    w,u=heappop(q)
    if vis[u]:
        continue
    ans+=w
    vis[u]=1
    for v in range(n):
        if not vis[v] and graph[u][v]!=-1:
            heappush(q,(graph[u][v],v))
print(ans)
```

拓扑排序

```
from collections import deque
def topo_sort(graph):
    in_degree={u:0 for u in graph}
    for u in graph:
        for v in graph[u]:
            in_degree[v]+=1
    q=deque([u for u in in_degree if in_degree[u]==0])
    topo_order=[]
    while q:
        u=q.popleft()
        topo_order.append(u)
        for v in graph[u]:
            in_degree[v]-=1
            if in_degree[v]==0:
                q.append(v)
    if len(topo_order)!=len(graph):
        return []
    return topo_order
```

工具

int(str,n) 将字符串`str`转换为`n`进制的整数。

for key,value in dict.items()　　遍历字典的键值对。

for index,value in enumerate(list)　枚举列表，提供元素及其索引。

dict.get(key,default)　从字典中获取键对应的值，如果键不存在，则返回默认值`default`。

list(zip(a,b))　将两个列表元素一一配对，生成元组的列表。

math.pow(m,n)　计算`m`的`n`次幂。

math.log(m,n)　　计算以`n`为底的`m`的对数。

lrucache

```
from functools import lru_cache
@lru_cache(maxsize=None)
```

bisect

```
import bisect
# 创建一个有序列表
sorted_list = [1, 3, 4, 4, 5, 7]
# 使用bisect_left查找插入点
position = bisect.bisect_left(sorted_list, 4)
print(position)  # 输出: 2
# 使用bisect_right查找插入点
position = bisect.bisect_right(sorted_list, 4)
print(position)  # 输出: 4
# 使用insort_left插入元素
bisect.insort_left(sorted_list, 4)
print(sorted_list)  # 输出: [1, 3, 4, 4, 4, 5, 7]
# 使用insort_right插入元素
bisect.insort_right(sorted_list, 4)
print(sorted_list)  # 输出: [1, 3, 4, 4, 4, 4, 5, 7]
```

字符串

1. `str.lstrip() / str.rstrip()`: 移除字符串左侧/右侧的空白字符。
2. `str.find(sub)`: 返回子字符串`sub`在字符串中首次出现的索引，如果未找到，则返回-1。
3. `str.replace(old, new)`: 将字符串中的`old`子字符串替换为`new`。
4. `str.startswith(prefix) / str.endswith(suffix)`: 检查字符串是否以`prefix`开头或以`suffix`结尾。
5. `str.isalpha() / str.isdigit() / str.isalnum()`: 检查字符串是否全部由字母/数字/字母和数字组成。
6. `str.title()`: 每个单词首字母大写。

counter：计数

```python
from collections import Counter
# 创建一个Counter对象
count = Counter(['apple', 'banana', 'apple', 'orange', 'banana', 'apple'])
# 输出Counter对象
print(count)  # 输出: Counter({'apple': 3, 'banana': 2, 'orange': 1})
# 访问单个元素的计数
print(count['apple'])  # 输出: 3
# 访问不存在的元素返回0
print(count['grape'])  # 输出: 0
# 添加元素
count.update(['grape', 'apple'])
print(count)  # 输出: Counter({'apple': 4, 'banana': 2, 'orange': 1, 'grape': 1})
```

permutations：全排列

```python
from itertools import permutations
# 创建一个可迭代对象的排列
perm = permutations([1, 2, 3])
# 打印所有排列
for p in perm:
    print(p)
# 输出: (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)
```

combinations：组合

```python
from itertools import combinations
# 创建一个可迭代对象的组合
comb = combinations([1, 2, 3], 2)
# 打印所有组合
for c in comb:
    print(c)
# 输出: (1, 2), (1, 3), (2, 3)
```

reduce：累次运算

```python
from functools import reduce
# 使用reduce计算列表元素的乘积
product = reduce(lambda x, y: x * y, [1, 2, 3, 4])
print(product)  # 输出: 24
```

product：笛卡尔积

```python
from itertools import product
# 创建两个可迭代对象的笛卡尔积
prod = product([1, 2], ['a', 'b'])
# 打印所有笛卡尔积对
for p in prod:
    print(p)
# 输出: (1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')
```

解题步骤

1. **理解题目**:
    - 仔细阅读题目，确保理解所有的要求和限制条件。
2. **分析数据范围**:
    - 根据给定的数据范围来选择合适的算法。例如，小数据范围可能适合使用暴力解法或递归，而大数据范围可能需要O(n)的算法。
3. **考虑不同的算法**:
    - 如果直接的方法不可行，回忆学过的知识，考虑贪心、递归、动态规划、BFS或DFS等算法。
4. **寻找突破口**:
    - 如果碰到难题，尝试从不同的角度审视问题，使用逆向思考或转换思路。
5. **估计复杂度**:
    - 在编写代码之前，估计所选算法的时间和空间复杂度。

调试技巧

1. **对于TLE**:
    - 检查是否有冗余或低效的操作。
    - 如果没有明显的效率问题，可能需要改进算法。
2. **对于RE**:
    - 检查数组越界、空指针访问、栈溢出等常见错误。
    - `min()` 和 `max()` 函数不能对空列表进行操作
3. **对于WA**:
    - 仔细检查代码逻辑，特别是循环和条件判断。
    - 确保所有初始化正确，边界条件得到处理。