CheatingPaper_All

二分查找

🔍 二分查找模板（以LeetCode 1760为例）

一、通用二分查找模板

1.1 左闭右闭区间模板

```
def binary_search(nums, target):
    left, right = 0, len(nums) - 1  # 闭区间 [left, right]

    while left <= right:  # 注意：left <= right
        mid = left + (right - left) // 2  # 防止溢出

        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            left = mid + 1  # 在右半部分查找
        else:
            right = mid - 1  # 在左半部分查找

    return -1  # 未找到
```

1.2 左闭右开区间模板

```
def binary_search(nums, target):
    left, right = 0, len(nums)  # 左闭右开区间 [left, right)

    while left < right:  # 注意：left < right
        mid = left + (right - left) // 2

        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            left = mid + 1  # 在右半部分查找
        else:
            right = mid  # 在左半部分查找，注意不是mid-1

    return -1  # 未找到
```

二、二分查找的三种变体

2.1 查找第一个等于target的元素

```
def find_first(nums, target):
    left, right = 0, len(nums) - 1
    result = -1

    while left <= right:
        mid = left + (right - left) // 2

        if nums[mid] >= target:
            if nums[mid] == target:
                result = mid
            right = mid - 1
        else:
            left = mid + 1

    return result
```

2.2 查找最后一个等于target的元素

```
def find_last(nums, target):
    left, right = 0, len(nums) - 1
    result = -1

    while left <= right:
        mid = left + (right - left) // 2

        if nums[mid] <= target:
            if nums[mid] == target:
                result = mid
            left = mid + 1
        else:
            right = mid - 1

    return result
```

2.3 查找第一个大于等于target的元素

```
def find_first_ge(nums, target):
    left, right = 0, len(nums) - 1
```

```
    result = -1

    while left <= right:
        mid = left + (right - left) //
2

        if nums[mid] >= target:
            result = mid
            right = mid - 1
        else:
            left = mid + 1

    return result
```

三、最小值最大化/最大值最小化问题模板

这类问题的特点是：

- **满足条件的最小值或满足条件的最大值**
- 答案具有**单调性**：如果x可行，那么x+1也一定可行（或反之）
- 典型问题：分石头、分书籍、袋子里最少数目的球等

3.1 通用模板

```
def binary_search_optimization(nums,
constraint):
    # 确定二分查找的边界
    left = min_bound  # 最小可能值
    right = max_bound  # 最大可能值

    # 左闭右闭区间查找
    while left <= right:
        mid = left + (right - left) //
2

        if check(mid, nums,
constraint):  # 检查mid是否可行
            # 如果可行，尝试寻找更优解
            right = mid - 1  # 对于最小
值最大化，通常是left=mid+1
            # 对于最大值最小化，通常是
right=mid-1
        else:
            # 如果不可行，调整边界
            left = mid + 1

    return left  # 或根据具体问题返回
```

3.2 检查函数模板

```
def check(mid, nums, constraint):
    """
    检查是否满足约束条件
    mid: 当前尝试的值
    nums: 原始数据
    constraint: 约束条件（如操作次数限制）
    返回: True/False
    """
    # 实现具体的检查逻辑
    # 通常是计算需要多少次操作才能满足mid的
限制
    operations = 0
    for num in nums:
        # 根据具体问题实现
        pass

    return operations <= constraint
```

四、LeetCode 1760 - 袋子里最少数目的球 完整解法

4.1 问题分析

- **目标**：最小化操作后袋子里球的最大值（开销）
- **操作**：将一个袋子分成两个正整数个球的袋子
- **限制**：最多进行maxOperations次操作
- **单调性**：如果开销x可行（能用不超过maxOperations次操作使所有袋子球数≤x），那么所有大于x的值也都可行

4.2 完整代码实现

```
class Solution:
    def minimumSize(self, nums:
List[int], maxOperations: int) -> int:
        """
        二分查找解决"最小值最大化"问题
        时间复杂度: O(n log M)，其中n是数
组长度，M是最大值
        空间复杂度: O(1)
        """

        def can_achieve(max_balls:
int) -> bool:
            """
            检查是否可以在maxOperations
```

```
次操作内
            使所有袋子的球数不超过
max_balls
            """
            operations_needed = 0

            for num in nums:
                if num > max_balls:
                    # 将一个有num个球的
袋子拆分成每个不超过max_balls的小袋子
                    # 需要的操作次数 =
拆分后的袋子数 - 1
                    # 拆分后的袋子数 =
ceil(num / max_balls)
                    # 使用整数除法计算
ceil: (num + max_balls - 1) //
max_balls
                    operations_needed
+= (num + max_balls - 1) // max_balls
- 1

                    # 如果已经超过最大操
作次数，提前返回False
                    if
operations_needed > maxOperations:
                        return False

            return operations_needed
<= maxOperations

        # 二分查找的边界：
        # 最小值至少为1（因为球数必须是正
整数）
        # 最大值不会超过原始数组中的最大值
（因为最差情况就是不操作）
        left, right = 1, max(nums)
        answer = right  # 初始化答案为最
大值

        while left <= right:
            mid = left + (right -
left) // 2

            if can_achieve(mid):
                # 如果mid可行，尝试更小的
值
                answer = mid  # 更新答
案
                right = mid - 1
            else:
                # 如果mid不可行，需要增大
mid
                left = mid + 1

        return answer
```

4.3 代码解析

检查函数 `can_achieve(max_balls)`：

```
def can_achieve(max_balls):
    operations_needed = 0
    for num in nums:
        if num > max_balls:
            # 关键公式：
            # 将num个球分成每个不超过
max_balls的小袋子
            # 需要 ceil(num /
max_balls) 个袋子
            # 操作次数 = 袋子数 - 1
            operations_needed += (num
+ max_balls - 1) // max_balls - 1
    return operations_needed <=
maxOperations
```

公式推导：

- 设每个小袋子最多放 `max_balls` 个球
- 将 `num` 个球分成每个不超过 `max_balls` 的小袋子，最少需要 `ceil(num / max_balls)` 个袋子
- 每次操作增加一个袋子，所以操作次数 = `袋子数 - 1`
- `ceil(a/b)` 的整数计算：`(a + b - 1) // b`

二分查找部分：

```
left, right = 1, max(nums)
answer = right

while left <= right:
    mid = left + (right - left) // 2

    if can_achieve(mid):
        answer = mid       # 记录当前可
行解
        right = mid - 1    # 尝试更小的
值
    else:
        left = mid + 1     # 需要更大的
值
```

循环不变式：

- `answer` 总是记录当前找到的可行解中最小的
- 搜索结束时，`answer` 就是最小的可行解

4.4 测试用例验证

```
# 示例1：nums = [9], maxOperations = 2
# 过程：
# 1. left=1, right=9, mid=5 → 需要操作：ceil(9/5)=2 → 操作1次 → 可行 → answer=5, right=4
# 2. left=1, right=4, mid=2 → 需要操作：ceil(9/2)=5 → 操作4次 → 不可行 → left=3
# 3. left=3, right=4, mid=3 → 需要操作：ceil(9/3)=3 → 操作2次 → 可行 → answer=3, right=2
# 4. left=3, right=2 → 循环结束，返回answer=3

# 示例2：nums = [2,4,8,2], maxOperations = 4
# 最终返回2
```

五、二分查找的注意事项

5.1 防止溢出

```
# 错误的写法：可能溢出
mid = (left + right) // 2

# 正确的写法
mid = left + (right - left) // 2
```

5.2 边界条件处理

```
# 1. 空数组
if not nums:
    return -1

# 2. 单元素数组
if len(nums) == 1:
    return 0 if nums[0] == target else -1
```

5.3 循环终止条件

```
# 左闭右闭区间 [left, right]
while left <= right:  # 循环继续的条件

# 左闭右开区间 [left, right)
while left < right:   # 循环继续的条件
```

六、同类问题模板扩展

6.1 最小值最大化问题（如LeetCode 1760）

```
def min_max_optimization(nums, constraint):
    def check(mid):
        # 计算达到mid需要的操作次数或成本
        cost = 0
        for num in nums:
            if num > mid:
                cost += (num + mid - 1) // mid - 1
        return cost <= constraint

    left, right = 1, max(nums)
    answer = right

    while left <= right:
        mid = left + (right - left) // 2
        if check(mid):
            answer = mid
            right = mid - 1
        else:
            left = mid + 1

    return answer
```

6.2 最大值最小化问题（如LeetCode 410）

```
def max_min_optimization(nums, k):
    def check(mid):
        # 计算将数组分成若干段，每段和不超过mid，需要多少段
        segments = 1
        current_sum = 0
        for num in nums:
            if current_sum + num > mid:
                segments += 1
                current_sum = num
```

```
            else:
                current_sum += num
        return segments <= k

    left, right = max(nums), sum(nums)
    answer = right

    while left <= right:
        mid = left + (right - left) //
2
        if check(mid):
            answer = mid
            right = mid - 1
        else:
            left = mid + 1

    return answer
```

七、总结

二分查找解题步骤：

1. **确定搜索范围**：根据问题确定left和right的初始值
2. **设计检查函数**：判断mid是否满足条件
3. **确定单调性**：验证"如果mid可行，则更大/更小的值是否也可行"
4. **选择模板**：根据问题选择最小值最大化或最大值最小化模板
5. **处理边界**：注意循环终止条件和返回值

对于LeetCode 1760这类问题：

- **关键公式**：`操作次数 = ceil(num / max_balls) - 1`
- **整数ceil计算**：`(a + b - 1) // b`
- **搜索范围**：`left = 1, right = max(nums)`
- **单调性**：如果x可行，那么所有大于x的值都可行

掌握这个模板，你可以解决一大类二分查找的优化问题！ 🎯

栈

📖 一、栈基础概念

1.1 栈的定义与特性

- **LIFO（后进先出）**：最后入栈的元素最先出栈
- **两个基本操作**：push（压栈）、pop（出栈）
- **只在一端操作**：栈顶（top）进行所有操作，栈底（bottom）不动

1.2 栈的ADT（抽象数据类型）

```
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):       # 压栈
O(1)
    def pop(self):              # 出栈
O(1)
    def peek(self):             # 查看栈
顶 O(1)
    def is_empty(self):         # 判空
O(1)
    def size(self):             # 大小
O(1)
```

🎯 二、栈的核心应用场景

2.1 括号匹配类问题

模板代码

```
def is_valid_parentheses(s: str) ->
bool:
    stack = []
    mapping = {')': '(', ']': '[',
'}': '{'}

    for char in s:
        if char in mapping.values():
# 左括号
            stack.append(char)
        elif char in mapping:
# 右括号
            if not stack or stack[-1]
!= mapping[char]:
                return False
            stack.pop()
        else:
            continue  # 其他字符

    return not stack
```

2.2 表达式求值类问题

中缀转后缀模板

```
def infix_to_postfix(infix: str) ->
str:
    precedence = {'+': 1, '-': 1, '*':
2, '/': 2}
    stack = []
    output = []

    i = 0
    while i < len(infix):
        if infix[i].isdigit():
# 处理数字
            num = ''
            while i < len(infix) and
(infix[i].isdigit() or infix[i] ==
'.'):
                num += infix[i]
                i += 1
            output.append(num)
            continue
        elif infix[i] == '(':
            stack.append(infix[i])
        elif infix[i] == ')':
            while stack and stack[-1]
!= '(':

output.append(stack.pop())
            stack.pop()  # 弹出'('
        elif infix[i] in precedence:
# 运算符
            while (stack and stack[-1]
!= '(' and

precedence[stack[-1]] >=
precedence[infix[i]]):

output.append(stack.pop())
            stack.append(infix[i])
        i += 1

    while stack:
        output.append(stack.pop())

    return ' '.join(output)
```

后缀表达式求值模板

```
def evaluate_postfix(postfix: str) ->
float:
    stack = []
    tokens = postfix.split()

    for token in tokens:
        if token.replace('.',
'').isdigit():  # 数字
            stack.append(float(token))
        else:
# 运算符
            b = stack.pop()
            a = stack.pop()
            if token == '+':
                stack.append(a + b)
            elif token == '-':
                stack.append(a - b)
            elif token == '*':
                stack.append(a * b)
            elif token == '/':
                stack.append(a / b)

    return stack[0]
```

2.3 栈在算法中的其他应用

```
# 1. 逆序输出
def reverse_string(s):
    stack = list(s)
    return ''.join(stack[::-1])

# 2. 回文判断
def is_palindrome(s):
    stack = list(s)
    return s == ''.join(stack[::-1])

# 3. 进制转换
def decimal_to_base(n, base):
    digits = "0123456789ABCDEF"
    stack = []
    while n > 0:
        stack.append(digits[n % base])
        n //= base
    return ''.join(stack[::-1]) if
stack else "0"
```

三、单调栈

3.1 单调栈基本概念

什么是单调栈?

- 栈内元素保持**单调递增**或**单调递减**
- 用于解决"下一个更大/更小元素"问题
- 时间复杂度：O(n)，每个元素入栈出栈各一次

两种单调栈类型

```
# 单调递增栈：栈底到栈顶递增
# 用途：找下一个更小元素

# 单调递减栈：栈底到栈顶递减
# 用途：找下一个更大元素
```

3.2 下一个更大元素问题

找每个元素的下一个更大元素

```
def next_greater_element(nums):
    """
    返回每个元素的下一个更大元素，没有则返回-1
    示例: [4, 3, 2, 5, 7, 1] → [5, 5, 5, 7, -1, -1]
    """
    n = len(nums)
    result = [-1] * n
    stack = []  # 存储索引，栈内元素对应的值单调递减

    for i in range(n):
        # 当前元素大于栈顶元素对应的值
        while stack and nums[i] > nums[stack[-1]]:
            idx = stack.pop()
            result[idx] = nums[i]
        stack.append(i)

    return result
```

3.4 每日温度问题

```
def daily_temperatures(temperatures):
    """
    返回需要等待多少天才能有更高温度
    示例: [73,74,75,71,69,72,76,73] → [1,1,4,2,1,1,0,0]
    """
    n = len(temperatures)
    result = [0] * n
    stack = []  # 存储索引

    for i in range(n):
        # 当前温度大于栈顶那天的温度
        while stack and temperatures[i] > temperatures[stack[-1]]:
            idx = stack.pop()
            result[idx] = i - idx  # 计算天数差
        stack.append(i)

    return result
```

3.5 柱状图最大矩形问题

方法1：暴力扩展（理解思路）

```
def largest_rectangle_area_bruteforce(heights):
    max_area = 0
    n = len(heights)

    for i in range(n):
        # 向左扩展
        left = i
        while left - 1 >= 0 and heights[left - 1] >= heights[i]:
            left -= 1
        # 向右扩展
        right = i
        while right + 1 < n and heights[right + 1] >= heights[i]:
            right += 1

        width = right - left + 1
        max_area = max(max_area, width * heights[i])

    return max_area
```

方法2：单调栈优化（推荐）

```
def largest_rectangle_area(heights):
    """
    使用单调递增栈，时间复杂度O(n)
    示例: [2,1,5,6,2,3] → 10
```

```
    """
    heights.append(0)  # 添加哨兵，确保所有柱子都能被处理
    stack = []
    max_area = 0

    for i in range(len(heights)):
        # 当当前柱子高度小于栈顶柱子高度时
        while stack and heights[i] < heights[stack[-1]]:
            h = heights[stack.pop()]
            # 左边界：栈中下一个元素（若栈空则为-1）
            left = stack[-1] if stack else -1
            width = i - left - 1
            max_area = max(max_area, h * width)
        stack.append(i)

    heights.pop()  # 移除哨兵
    return max_area
```

3.6 接雨水问题

```
def trap(height):
    """
    单调栈解法，按层计算雨水
    示例: [0,1,0,2,1,0,1,3,2,1,2,1] → 6
    """
    stack = []
    water = 0

    for i in range(len(height)):
        # 当前高度大于栈顶高度
        while stack and height[i] > height[stack[-1]]:
            bottom = stack.pop()
            if not stack:  # 栈空，没有左边界
                break
            left = stack[-1]
            # 计算当前层能接的雨水
            h = min(height[left], height[i]) - height[bottom]
            w = i - left - 1
            water += h * w
        stack.append(i)

    return water
```

3.7 问题识别特征

- 求"下一个更大/更小元素"
- 求"左边/右边第一个比当前大/小的元素"
- 区间最值问题
- 涉及"宽度×高度"的面积问题

解题步骤

```
def monotonic_stack_template(nums):
    n = len(nums)
    result = [默认值] * n  # 通常初始化为-1或0
    stack = []  # 通常存储索引

    for i in range(n):
        # 维护栈的单调性
        while stack and 比较条件(nums[i], nums[stack[-1]]):
            idx = stack.pop()
            # 根据问题更新结果
            result[idx] = 计算值
        # 当前索引入栈
        stack.append(i)

    # 处理栈中剩余元素（如果需要）
    while stack:
        idx = stack.pop()
        result[idx] = 最终值

    return result
```

四种单调栈类型

栈类型	比较条件	解决的问题
单调递减栈	`nums[i] > stack[-1]`	下一个更大元素
单调递增栈	`nums[i] < stack[-1]`	下一个更小元素
严格递减栈	`nums[i] >= stack[-1]`	左边第一个更大元素
严格递增栈	`nums[i] <= stack[-1]`	左边第一个更小元素

💡 四、机考技巧与注意事项

4.1 栈的使用技巧

1. **明确栈顶含义**：栈顶通常表示"最近待处理"或"当前基准"
2. **哨兵技巧**：在数组头尾添加哨兵元素，简化边界判断
3. **存储索引**：单调栈通常存储索引而非值，方便计算宽度
4. **画图分析**：复杂问题先画图，明确入栈出栈条件

4.2 常见错误

```
# 错误1：访问空栈
stack = []
if stack[-1]:  # IndexError!

# 正确做法
if stack and stack[-1]:

# 错误2：忘记处理剩余元素
while stack:
    # 处理栈中剩余元素

# 错误3：比较条件写反
# 递增栈：当前元素 < 栈顶时出栈
# 递减栈：当前元素 > 栈顶时出栈
```

链表

📚 数据结构与算法 Cheat Sheet

📋 目录

📚 排序算法

快速排序模板

```
def quick_sort(arr, left, right):
    if left < right:
        pivot_pos = partition(arr, left, right)
        quick_sort(arr, left, pivot_pos - 1)
        quick_sort(arr, pivot_pos + 1, right)

def partition(arr, left, right):
    pivot = arr[right]
    i = left
    for j in range(left, right):
        if arr[j] < pivot:
            arr[i], arr[j] = arr[j], arr[i]
            i += 1
    arr[i], arr[right] = arr[right], arr[i]
    return i
```

归并排序模板

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]
        merge_sort(L)
        merge_sort(R)

        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

        while i < len(L):
            arr[k] = L[i]
            i += 1; k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1; k += 1
```

排序算法对比表

算法	最好	平均	最坏	空间
冒泡排序	O(n)	O(n²)	O(n²)	O(1)
选择排序	O(n²)	O(n²)	O(n²)	O(1)
插入排序	O(n)	O(n²)	O(n²)	O(1)
快速排序	O(n log n)	O(n log n)	O(n²)	O(log n)
归并排序	O(n log n)	O(n log n)	O(n log n)	O(n)
希尔排序	O(n log n)	O(n^(4/3))	O(n^(3/2))	O(1)

📚 链表 Cheat Sheet

📋 目录

---

1. 单链表

节点定义

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val    # 节点值
        self.next = next  # 指向下一个节点
```

基础链表类实现

```
class SinglyLinkedList:
    def __init__(self):
        self.head = None  # 头节点
        self.tail = None  # 尾节点（可选，方便尾插）
        self.size = 0     # 链表长度

    # 头部插入
    def push_front(self, val):
        new_node = ListNode(val, self.head)
        self.head = new_node
        if self.size == 0:
            self.tail = new_node
        self.size += 1

    # 尾部插入
    def push_back(self, val):
        new_node = ListNode(val)
        if self.size == 0:
            self.head = self.tail = new_node
        else:
            self.tail.next = new_node
            self.tail = new_node
        self.size += 1

    # 头部删除
    def pop_front(self):
        if self.size == 0:
            raise Exception("Empty list")
        val = self.head.val
        self.head = self.head.next
        self.size -= 1
        if self.size == 0:
            self.tail = None
        return val

    # 遍历打印
    def print_list(self):
        curr = self.head
        while curr:
            print(curr.val, end=" -> ")
            curr = curr.next
        print("None")
```

---

2. 双链表

节点定义

```python
class DListNode:
    def __init__(self, key=0, val=0):
        self.key = key          # 键（可选）
        self.val = val          # 值
        self.prev = None        # 指向前一个节点
        self.next = None        # 指向后一个节点
```

双链表类实现

```python
class DoublyLinkedList:
    def __init__(self):
        self.head = None  # 头节点
        self.tail = None  # 尾节点
        self.size = 0     # 链表长度

    # 头部插入
    def push_front(self, key, val):
        new_node = DListNode(key, val)
        if self.size == 0:
            self.head = self.tail = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node
        self.size += 1

    # 尾部插入
    def push_back(self, key, val):
        new_node = DListNode(key, val)
        if self.size == 0:
            self.head = self.tail = new_node
        else:
            new_node.prev = self.tail
            self.tail.next = new_node
            self.tail = new_node
        self.size += 1

    # 删除指定节点
    def remove(self, node):
        if not node:
            return

        # 调整前驱节点的next
        if node.prev:
            node.prev.next = node.next
        else:  # node是头节点
            self.head = node.next

        # 调整后继节点的prev
        if node.next:
            node.next.prev = node.prev
        else:  # node是尾节点
            self.tail = node.prev

        node.prev = node.next = None
        self.size -= 1

    # 删除头部
    def pop_front(self):
        if self.size == 0:
            raise Exception("Empty list")
        node = self.head
        self.remove(node)
        return node

    # 删除尾部
    def pop_back(self):
        if self.size == 0:
            raise Exception("Empty list")
        node = self.tail
        self.remove(node)
        return node

    # 移动到头部（LRU缓存常用）
    def move_to_front(self, node):
        if node == self.head:
            return
        self.remove(node)
        self.push_front(node.key, node.val)

    # 正向遍历
    def print_forward(self):
        curr = self.head
        while curr:
            print(f"({curr.key}: {curr.val})", end=" <-> ")
            curr = curr.next
        print("None")

    # 反向遍历
    def print_backward(self):
        curr = self.tail
        while curr:
            print(f"({curr.key}: {curr.val})", end=" <-> ")
```

```
            curr = curr.prev
        print("None")
```

浏览器历史记录（双链表应用）

```
class BrowserHistory:
    def __init__(self, homepage: str):
        self.curr = DListNode()  # 当前页面节点
        self.curr.val = homepage

    def visit(self, url: str) -> None:
        # 清除前进历史
        if self.curr.next:
            self.curr.next = None

        new_node = DListNode()
        new_node.val = url
        new_node.prev = self.curr
        self.curr.next = new_node
        self.curr = new_node

    def back(self, steps: int) -> str:
        while steps > 0 and self.curr.prev:
            self.curr = self.curr.prev
            steps -= 1
        return self.curr.val

    def forward(self, steps: int) -> str:
        while steps > 0 and self.curr.next:
            self.curr = self.curr.next
            steps -= 1
        return self.curr.val
```

---

3. 循环链表

循环单链表

```
class CircularSinglyLinkedList:
    def __init__(self):
        self.tail = None  # 指向尾节点（尾节点.next = 头节点）
        self.size = 0

    def push_front(self, val):
        new_node = ListNode(val)
        if self.size == 0:
            new_node.next = new_node  # 自己指向自己
            self.tail = new_node
        else:
            new_node.next = self.tail.next  # 新节点指向原头节点
            self.tail.next = new_node  # 尾节点指向新节点
        self.size += 1

    def push_back(self, val):
        self.push_front(val)
        self.tail = self.tail.next  # 更新尾指针

    def pop_front(self):
        if self.size == 0:
            raise Exception("Empty list")

        head = self.tail.next
        if self.size == 1:
            self.tail = None
        else:
            self.tail.next = head.next  # 跳过头节点

        self.size -= 1
        return head.val
```

约瑟夫问题（循环链表应用）

```
def josephus_circular(n, m):
    # 创建循环链表：1->2->...->n->1
    head = ListNode(1)
    curr = head
    for i in range(2, n+1):
        curr.next = ListNode(i)
        curr = curr.next
    curr.next = head  # 形成环

    # 约瑟夫淘汰过程
    while curr.next != curr:  # 只剩一个节点
        # 数m-1个人
        for _ in range(m-1):
            curr = curr.next
        # 淘汰下一个节点
        curr.next = curr.next.next

    return curr.val  # 幸存者
```

4. 链表常用操作

1. 反转链表

```
# 迭代法
def reverse_list(head):
    prev = None
    curr = head
    while curr:
        next_node = curr.next
        curr.next = prev
        prev = curr
        curr = next_node
    return prev

# 递归法
def reverse_list_recursive(head):
    if not head or not head.next:
        return head
    new_head =
reverse_list_recursive(head.next)
    head.next.next = head
    head.next = None
    return new_head
```

2. 合并两个有序链表

```
def merge_two_lists(l1, l2):
    dummy = ListNode(-1)  # 哨兵节点
    tail = dummy

    while l1 and l2:
        if l1.val <= l2.val:
            tail.next = l1
            l1 = l1.next
        else:
            tail.next = l2
            l2 = l2.next
        tail = tail.next

    # 连接剩余部分
    tail.next = l1 if l1 else l2
    return dummy.next
```

3. 快慢指针找中间节点

```
def find_middle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow  # 奇数返回正中，偶数返回
靠后的中间节点
```

4. 判断链表是否有环

```
def has_cycle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True
    return False
```

5. 找环的入口节点

```
def detect_cycle(head):
    slow = fast = head
    has_cycle = False

    # 第一阶段：判断是否有环
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            has_cycle = True
            break

    if not has_cycle:
        return None

    # 第二阶段：找环入口
    slow = head
    while slow != fast:
        slow = slow.next
        fast = fast.next

    return slow
```

6. 删除倒数第N个节点

```
def remove_nth_from_end(head, n):
    dummy = ListNode(0, head)
    fast = slow = dummy

    # fast先走n+1步
    for _ in range(n + 1):
        fast = fast.next
```

```
    # 双指针一起走
    while fast:
        fast = fast.next
        slow = slow.next

    # 删除节点
    slow.next = slow.next.next
    return dummy.next
```

7. 链表排序（归并排序）

```
def sort_list(head):
    if not head or not head.next:
        return head

    # 找中点
    slow, fast = head, head.next
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    mid = slow.next
    slow.next = None  # 切断链表

    # 递归排序
    left = sort_list(head)
    right = sort_list(mid)

    # 合并
    return merge_two_lists(left,
right)
```

---

5. 复杂度对比

操作	单链表	双链表	循环单链表	Python List
访问	O(n)	O(n)	O(n)	O(1)
头部插入	O(1)	O(1)	O(1)	O(n)
尾部插入	O(n) / O(1)†	O(1)	O(1)	O(1)
头部删除	O(1)	O(1)	O(1)	O(n)
尾部删除	O(n)	O(1)	O(1)	O(1)
指定位置插入	O(n)	O(n)	O(n)	O(n)
指定位置删除	O(n)	O(n)	O(n)	O(n)
查找元素	O(n)	O(n)	O(n)	O(n)
内存开销	低	中	低	连续

† 如果维护尾指针，单链表尾部插入可为O(1)

选择指南

1. **需要频繁随机访问** → 数组/Python List
2. **频繁在头部插入/删除** → 单链表/双链表
3. **频繁在尾部插入/删除** → 双链表/循环链表
4. **需要双向遍历** → 双链表
5. **环形结构问题** → 循环链表
6. **缓存淘汰策略** → 双链表（LRU）

🎯 常见题型模板

题型2：链表相交/环问题

```
# 判断两个链表是否相交
def get_intersection_node(headA,
headB):
    if not headA or not headB:
        return None

    pA, pB = headA, headB
    while pA != pB:
        pA = pA.next if pA else headB
        pB = pB.next if pB else headA

    return pA
```

题型3：链表排序/重排

```
# 重排链表：L0→Ln→L1→Ln-1→...
def reorder_list(head):
    if not head or not head.next:
        return
```

```
    # 找中点
    slow, fast = head, head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    # 反转后半部分
    prev, curr = None, slow
    while curr:
        next_node = curr.next
        curr.next = prev
        prev = curr
        curr = next_node

    # 合并两个链表
    first, second = head, prev
    while second.next:
        temp1, temp2 = first.next,
second.next
        first.next = second
        second.next = temp1
        first, second = temp1, temp2
```

📝 链表调试技巧

1. 可视化链表

```
def visualize(head, name="链表"):
    print(f"{name}: ", end="")
    curr = head
    while curr:
        print(curr.val, end=" -> ")
        curr = curr.next
    print("None")
```

2. 创建测试链表

```
def create_list(values):
    if not values:
        return None
    head = ListNode(values[0])
    curr = head
    for val in values[1:]:
        curr.next = ListNode(val)
        curr = curr.next
    return head
```

3. 常见错误检查清单

- [ ] 空链表判断：`if not head`
- [ ] 单节点判断：`if not head.next`
- [ ] 边界节点处理：头节点、尾节点
- [ ] 指针修改顺序：先保存再修改
- [ ] 循环链表注意终止条件

🚀 快速参考

单链表 vs 双链表

```
# 单链表：节省内存，单向遍历
class Node:
    val + next

# 双链表：功能强大，双向遍历，支持O(1)删除
class DNode:
    val + prev + next
```

循环链表 vs 非循环链表

```
# 非循环：tail.next = None
1->2->3->None

# 循环：tail.next = head
1->2->3->back to 1
```

哨兵节点技巧

```
# 简化边界条件处理
dummy = ListNode(0, head)
# 操作完成后返回 dummy.next
```

使用提示：

1. 画图理解链表操作，特别是指针变化
2. 注意空指针异常（None判断）
3. 双链表操作要同时维护prev和next
4. 循环链表注意终止条件避免死循环
5. 复杂操作先处理一般情况，再处理边界情况

祝机考顺利！ 🎉

树

1 二叉树节点定义

```
# 基础二叉树节点
class TreeNode:
    def __init__(self, val=0,
left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

# 多叉树节点（邻接表表示）
class NaryTreeNode:
    def __init__(self, val=None,
children=None):
        self.val = val
        self.children = children if
children is not None else []
```

2 二叉树遍历（递归版）

```
def preorder(root):
    """前序遍历：根→左→右"""
    if not root: return []
    return [root.val] +
preorder(root.left) +
preorder(root.right)

def inorder(root):
    """中序遍历：左→根→右"""
    if not root: return []
    return inorder(root.left) +
[root.val] + inorder(root.right)

def postorder(root):
    """后序遍历：左→右→根"""
    if not root: return []
    return postorder(root.left) +
postorder(root.right) + [root.val]
```

4 层序遍历（BFS - 队列）

```
from collections import deque

def level_order(root):
    if not root: return []
    queue = deque([root])
    result = []
    while queue:
        level_size = len(queue)
        level = []
        for _ in range(level_size):
            node = queue.popleft()
            level.append(node.val)
            if node.left:
queue.append(node.left)
            if node.right:
queue.append(node.right)
        result.append(level)
    return result
```

5 二叉树深度与叶子计数

```
def tree_depth(root):
    """返回二叉树深度（节点数定义）"""
    if not root: return 0
    return max(tree_depth(root.left),
tree_depth(root.right)) + 1

def count_leaves(root):
    """统计叶子节点数"""
    if not root: return 0
    if not root.left and not
root.right: return 1
    return count_leaves(root.left) +
count_leaves(root.right)
```

6 根据输入建树（编号格式）

```
# 输入格式：n行，每行 left right
def build_tree_by_input():
    n = int(input())
    nodes = [TreeNode(i+1) for i in
range(n)]
    has_parent = [False] * n
    for i in range(n):
        left, right = map(int,
input().split())
        if left != -1:
            nodes[i].left =
nodes[left-1]
            has_parent[left-1] = True
```

```
            if right != -1:
                nodes[i].right =
nodes[right-1]
                has_parent[right-1] = True
    # 找根
    root_idx = has_parent.index(False)
    return nodes[root_idx]
```

7 根据遍历序列建树

```
# 前序+中序 → 建树
def build_from_pre_inorder(preorder,
inorder):
    if not preorder or not inorder:
return None
    root_val = preorder[0]
    root = TreeNode(root_val)
    idx = inorder.index(root_val)
    root.left =
build_from_pre_inorder(preorder[1:1+id
x], inorder[:idx])
    root.right =
build_from_pre_inorder(preorder[1+idx:
], inorder[idx+1:])
    return root

# 中序+后序 → 建树
def build_from_in_postorder(inorder,
postorder):
    if not inorder or not postorder:
return None
    root_val = postorder[-1]
    root = TreeNode(root_val)
    idx = inorder.index(root_val)
    root.left =
build_from_in_postorder(inorder[:idx],
postorder[:idx])
    root.right =
build_from_in_postorder(inorder[idx+1:
], postorder[idx:-1])
    return root
```

9 哈夫曼编码（优先队列）

```
import heapq

class HuffmanNode:
    def __init__(self, freq,
char=None):
        self.freq = freq
        self.char = char
        self.left = None
        self.right = None
    def __lt__(self, other):
        return self.freq < other.freq

def build_huffman_tree(char_freq):
    heap = [HuffmanNode(freq, char)
for char, freq in char_freq.items()]
    heapq.heapify(heap)
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = HuffmanNode(left.freq
+ right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)
    return heap[0] if heap else None
```

10 并查集（路径压缩 + 按秩合并）

```
class DisjointSet:
    """并查集：路径压缩 + 按秩合并"""
    def __init__(self, n):
        self.parent = list(range(n +
1))  # 1-indexed
        self.rank = [0] * (n + 1)
        self.size = [1] * (n + 1)  #
集合大小

    def find(self, x):
        """查找根节点（路径压缩）"""
        if self.parent[x] != x:
            self.parent[x] =
self.find(self.parent[x])
        return self.parent[x]

    def union_by_rank(self, x, y):
        """按秩合并"""
        x_root = self.find(x)
        y_root = self.find(y)

        if x_root == y_root:
            return
```

```
        # 按秩合并
        if self.rank[x_root] <
self.rank[y_root]:
            self.parent[x_root] =
y_root
        elif self.rank[x_root] >
self.rank[y_root]:
            self.parent[y_root] =
x_root
        else:
            self.parent[y_root] =
x_root
            self.rank[x_root] += 1

    def union_by_size(self, x, y):
        """按大小合并"""
        x_root = self.find(x)
        y_root = self.find(y)

        if x_root == y_root:
            return

        # 按大小合并
        if self.size[x_root] <
self.size[y_root]:
            self.parent[x_root] =
y_root
            self.size[y_root] +=
self.size[x_root]
        else:
            self.parent[y_root] =
x_root
            self.size[x_root] +=
self.size[y_root]

    def is_connected(self, x, y):
        """判断是否连通"""
        return self.find(x) ==
self.find(y)

    def count_sets(self, n):
        """统计集合数量"""
        roots = set()
        for i in range(1, n + 1):
            roots.add(self.find(i))
        return len(roots)

    def get_set_sizes(self, n):
        """获取每个集合的大小（降序）"""
        root_sizes = {}
        for i in range(1, n + 1):
            root = self.find(i)
            root_sizes[root] =
root_sizes.get(root, 0) + 1
        return
sorted(root_sizes.values(),
reverse=True)
```

12 前缀树

```
class TrieNode:
    def __init__(self):
        self.children = {}  # 字符 ->
子节点
        self.is_end = False  # 是否单词
结束
        self.count = 0  # 单词出现次数
（可选）

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

    def insert(self, word):
        """插入单词"""
        node = self.root
        for char in word:
            if char not in
node.children:
                node.children[char] =
TrieNode()
            node = node.children[char]
        node.is_end = True
        node.count += 1

    def search(self, word):
        """搜索完整单词"""
        node = self.root
        for char in word:
            if char not in
node.children:
                return False
            node = node.children[char]
        return node.is_end

    def starts_with(self, prefix):
        """检查前缀是否存在"""
        node = self.root
        for char in prefix:
            if char not in
node.children:
                return False
```

```
            node = node.children[char]
        return True

    def delete(self, word):
        """删除单词（简化版）"""
        if not self.search(word):
            return False

        node = self.root
        stack = []  # 记录路径

        # 找到单词路径
        for char in word:
            stack.append((node, char))
            node = node.children[char]

        # 标记非单词结尾
        node.is_end = False
        node.count = 0

        # 回溯删除无用的节点
        while stack and not
node.children and not node.is_end:
            parent, char = stack.pop()
            del parent.children[char]
            node = parent

        return True

    def get_words_with_prefix(self,
prefix):
        """获取以prefix开头的所有单词"""
        def dfs(node, path, result):
            if node.is_end:

result.append(''.join(path))
            for char, child in
node.children.items():
                path.append(char)
                dfs(child, path,
result)
                path.pop()

        # 先找到前缀节点
        node = self.root
        for char in prefix:
            if char not in
node.children:
                return []
            node = node.children[char]

        result = []
        dfs(node, list(prefix),
result)
        return result
```

13 二叉平衡树：

```
class AVLNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
        self.height = 1  # 节点高度

class AVLTree:
    def __init__(self):
        self.root = None

    def insert(self, val):
        """插入节点并保持平衡"""
        self.root =
self._insert(self.root, val)

    def _insert(self, node, val):
        if not node:
            return AVLNode(val)
        elif val < node.val:
            node.left =
self._insert(node.left, val)
        else:
            node.right =
self._insert(node.right, val)

        # 更新高度
        node.height = 1 +
max(self._get_height(node.left),

self._get_height(node.right))

        # 获取平衡因子
        balance =
self._get_balance(node)

        # 平衡修复
        # LL型 - 右旋
        if balance > 1 and val <
node.left.val:
            return
self._right_rotate(node)
        # RR型 - 左旋
        if balance < -1 and val >
node.right.val:
```

```
            return
self._left_rotate(node)
        # LR型 - 先左旋后右旋
        if balance > 1 and val >
node.left.val:
            node.left =
self._left_rotate(node.left)
            return
self._right_rotate(node)
        # RL型 - 先右旋后左旋
        if balance < -1 and val <
node.right.val:
            node.right =
self._right_rotate(node.right)
            return
self._left_rotate(node)

        return node

    def _get_height(self, node):
        """获取节点高度"""
        if not node:
            return 0
        return node.height

    def _get_balance(self, node):
        """获取平衡因子"""
        if not node:
            return 0
        return
self._get_height(node.left) -
self._get_height(node.right)

    def _left_rotate(self, z):
        """左旋"""
        y = z.right
        T2 = y.left

        # 执行旋转
        y.left = z
        z.right = T2

        # 更新高度
        z.height = 1 +
max(self._get_height(z.left),

self._get_height(z.right))
        y.height = 1 +
max(self._get_height(y.left),

self._get_height(y.right))

        return y

    def _right_rotate(self, y):
        """右旋"""
        x = y.left
        T2 = x.right

        # 执行旋转
        x.right = y
        y.left = T2

        # 更新高度
        y.height = 1 +
max(self._get_height(y.left),

self._get_height(y.right))
        x.height = 1 +
max(self._get_height(x.left),

self._get_height(x.right))

        return x

    def preorder(self):
        """前序遍历"""
        return
self._preorder(self.root)

    def _preorder(self, node):
        if not node:
            return []
        return [node.val] +
self._preorder(node.left) +
self._preorder(node.right)

    def inorder(self):
        """中序遍历（有序）"""
        return
self._inorder(self.root)

    def _inorder(self, node):
        if not node:
            return []
        return
self._inorder(node.left) + [node.val]
+ self._inorder(node.right)
```

🔥 堆 (heapq) 模块实用指南

1️⃣ heapq 基础操作

```python
import heapq

# 创建堆（最小堆）
heap = []
nums = [3, 1, 4, 1, 5, 9, 2]

# 方法1：逐个添加
for num in nums:
    heapq.heappush(heap, num)

# 方法2：批量建堆
heap = nums[:]
heapq.heapify(heap)  # O(n) 时间复杂度

# 弹出最小元素
min_val = heapq.heappop(heap)  # 弹出并返回最小元素

# 查看最小元素（不弹出）
min_val = heap[0]

# 弹出最小元素并添加新元素（高效）
new_val = 6
replaced = heapq.heapreplace(heap, new_val)  # 先pop再push

# 弹出最小元素，如果新元素更小则不push
pushed = heapq.heappushpop(heap, new_val)  # 先push再pop

# 获取前k个最大/最小元素
k = 3
largest = heapq.nlargest(k, nums)  # [9, 5, 4]
smallest = heapq.nsmallest(k, nums)  # [1, 1, 2]
```

2 最大堆实现技巧

```python
# heapq默认最小堆，实现最大堆的两种方法：

# 方法1：取负数
max_heap = []
for num in nums:
    heapq.heappush(max_heap, -num)  # 存负数
# 获取最大值
max_val = -heapq.heappop(max_heap)  # 取负数恢复

# 方法2：使用自定义类
class MaxHeapItem:
    def __init__(self, val):
        self.val = val
    def __lt__(self, other):
        return self.val > other.val  # 反向比较实现最大堆

max_heap = []
for num in nums:
    heapq.heappush(max_heap, MaxHeapItem(num))
max_val = heapq.heappop(max_heap).val
```

3 带优先级的堆（元组比较）

```python
# heapq支持元组比较，第一个元素为优先级
priority_heap = []
heapq.heappush(priority_heap, (2, 'task2'))
heapq.heappush(priority_heap, (1, 'task1'))
heapq.heappush(priority_heap, (3, 'task3'))

while priority_heap:
    priority, task = heapq.heappop(priority_heap)
    print(f"执行任务: {task}, 优先级: {priority}")
```

🎯 高频题型模板

2 堆排序与Top K问题

```python
def heap_sort(nums, reverse=False):
    """堆排序"""
    import heapq
    if reverse:
        # 降序排序
        nums = [-x for x in nums]
    heapq.heapify(nums)
    sorted_nums = []
    while nums:
        val = heapq.heappop(nums)
        sorted_nums.append(-val if reverse else val)
    return sorted_nums

def find_kth_smallest(nums, k):
```

```
    """第k小元素"""
    import heapq
    heapq.heapify(nums)
    for _ in range(k-1):
        heapq.heappop(nums)
    return heapq.heappop(nums)

def find_kth_largest(nums, k):
    """第k大元素（快速选择替代方案）"""
    import heapq
    min_heap = []
    for num in nums:
        heapq.heappush(min_heap, num)
        if len(min_heap) > k:
            heapq.heappop(min_heap)  # 保持堆大小为k
    return min_heap[0]  # 堆顶即为第k大
```

3 中位数维护（双堆法）

```
class MedianFinder:
    """实时计算数据流的中位数"""
    def __init__(self):
        self.small = []  # 最大堆，存较小一半
        self.large = []  # 最小堆，存较大一半

    def addNum(self, num):
        # 始终保持: len(small) >= len(large)
        heapq.heappush(self.small, -num)  # 最大堆用负数
        # 平衡两个堆
        heapq.heappush(self.large, -heapq.heappop(self.small))
        if len(self.small) < len(self.large):
            heapq.heappush(self.small, -heapq.heappop(self.large))

    def findMedian(self):
        if len(self.small) > len(self.large):
            return -self.small[0]
        return (-self.small[0] + self.large[0]) / 2
```

4 BST中第K小元素（中序遍历）

```
# 递归版
def kth_smallest_BST(root, k):
    """二叉搜索树中第K小元素"""
    stack = []
    curr = root
    count = 0

    while curr or stack:
        # 左子树入栈
        while curr:
            stack.append(curr)
            curr = curr.left
        # 访问节点
        curr = stack.pop()
        count += 1
        if count == k:
            return curr.val
        # 转向右子树
        curr = curr.right
    return None
```

6 二叉树序列化与反序列化

```
def serialize(root):
    """二叉树序列化为字符串"""
    if not root: return "null"
    return f"{root.val},{serialize(root.left)},{serialize(root.right)}"

def deserialize(data):
    """字符串反序列化为二叉树"""
    vals = data.split(',')
    def build():
        if not vals: return None
        val = vals.pop(0)
        if val == "null": return None
        node = TreeNode(int(val))
        node.left = build()
        node.right = build()
        return node
    return build()
```

7 二叉树的直径（最长路径）

```
def diameter_of_binary_tree(root):
    """二叉树直径（最长路径长度）"""
    diameter = 0

    def depth(node):
        nonlocal diameter
```

```
        if not node: return 0
        left = depth(node.left)
        right = depth(node.right)
        # 更新直径
        diameter = max(diameter, left
+ right)
        # 返回当前节点深度
        return max(left, right) + 1

    depth(root)
    return diameter
```

8 最近公共祖先（LCA）

```
def lowest_common_ancestor(root, p,
q):
    """二叉树的最近公共祖先"""
    if not root or root == p or root
== q:
        return root
    left =
lowest_common_ancestor(root.left, p,
q)
    right =
lowest_common_ancestor(root.right, p,
q)

    if left and right:  # p和q分别在左
右子树
        return root
    return left or right  # 返回非空的
那个
```

🌳 表达式与树 - 机考模板与题型

---

1 表达式转换（中缀 ↔ 后缀 ↔ 前缀）

1.1 中缀转后缀（Shunting Yard算法）

```
def infix_to_postfix(infix_expr):
    """中缀表达式转后缀表达式（支持+-*/和
括号）"""
    precedence = {'+': 1, '-': 1, '*':
2, '/': 2}
    output = []
    stack = []

    tokens = infix_expr.replace(' ',
'')
    i = 0
    while i < len(tokens):
        # 处理多位数和浮点数
        if tokens[i].isdigit() or
tokens[i] == '.':
            num = tokens[i]
            i += 1
            while i < len(tokens) and
(tokens[i].isdigit() or tokens[i] ==
'.'):
                num += tokens[i]
                i += 1
            output.append(num)
            continue
        elif tokens[i].isalpha():  #
变量名
            var = tokens[i]
            i += 1
            while i < len(tokens) and
tokens[i].isalpha():
                var += tokens[i]
                i += 1
            output.append(var)
            continue

        # 处理运算符和括号
        token = tokens[i]
        if token == '(':
            stack.append(token)
        elif token == ')':
            while stack and stack[-1]
!= '(':

output.append(stack.pop())
            stack.pop()  # 弹出 '('
        else:  # 运算符
            while (stack and stack[-1]
!= '(' and
                   precedence[token]
<= precedence.get(stack[-1], 0)):

output.append(stack.pop())
            stack.append(token)
        i += 1

    while stack:
```

```
        output.append(stack.pop())

    return ' '.join(output)

# 测试
# print(infix_to_postfix("3+4.5*(7+2)"))  # "3 4.5 7 2 + * +"
```

1.2 后缀转中缀（带括号）

```
def postfix_to_infix(postfix_expr):
    """后缀表达式转中缀表达式（添加必要括号）"""
    tokens = postfix_expr.split()
    stack = []

    for token in tokens:
        if token.replace('.', '').isdigit() or token.isalpha():
            stack.append(token)
        else:  # 运算符
            right = stack.pop()
            left = stack.pop()
            # 根据优先级决定是否加括号
            expr = f"({left} {token} {right})"
            stack.append(expr)

    return stack[0] if stack else ""
```

1.3 中缀转前缀

```
def infix_to_prefix(infix_expr):
    """中缀表达式转前缀表达式"""
    # 反转表达式，交换左右括号
    reversed_expr = infix_expr[::-1]
    reversed_expr = reversed_expr.replace('(', '$').replace(')', '(').replace('$', ')')

    # 转换为后缀
    postfix = infix_to_postfix(reversed_expr)

    # 反转后缀表达式得到前缀
    return ' '.join(postfix.split()[::-1])
```

2 表达式树（解析树）

2.1 节点定义与基础构建

```
class ExprTreeNode:
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

    def __str__(self):
        return str(self.value)
```

2.2 后缀表达式建树

```
def build_expr_tree_from_postfix(postfix_expr):
    """后缀表达式构建表达式树"""
    tokens = postfix_expr.split()
    stack = []

    for token in tokens:
        if token.replace('.', '').isdigit() or token.isalpha():
            # 操作数作为叶子节点
            stack.append(ExprTreeNode(token))
        else:
            # 运算符：弹出两个操作数，构建子树
            right = stack.pop()
            left = stack.pop()
            node = ExprTreeNode(token, left, right)
            stack.append(node)

    return stack[0] if stack else None
```

2.3 中缀表达式建树（带括号）

```
def build_expr_tree_from_infix(infix_expr):
    """中缀表达式构建表达式树（完全括号表达式）"""
    # 先转为后缀表达式
    postfix = infix_to_postfix(infix_expr)
```

```
    # 再用后缀表达式建树
    return
build_expr_tree_from_postfix(postfix)
```

2.4 前缀表达式建树

```
def
build_expr_tree_from_prefix(prefix_exp
r):
    """前缀表达式构建表达式树"""
    tokens = prefix_expr.split()[::-1]
# 反转方便处理
    stack = []

    for token in tokens:
        if token.replace('.',
'').isdigit() or token.isalpha():

stack.append(ExprTreeNode(token))
        else:
            # 前缀表达式：先出栈的是左孩子
            left = stack.pop()
            right = stack.pop()
            node = ExprTreeNode(token,
left, right)
            stack.append(node)

    return stack[0] if stack else None
```

---

3 表达式树遍历与计算

3.1 表达式树遍历

```
def expr_tree_preorder(root):
    """表达式树前序遍历（前缀表达式）"""
    if not root:
        return []
    return [root.value] +
expr_tree_preorder(root.left) +
expr_tree_preorder(root.right)

def expr_tree_inorder(root):
    """表达式中序遍历（中缀表达式，需要加括
号）"""
    if not root:
        return []

    left =
expr_tree_inorder(root.left)
    right =
expr_tree_inorder(root.right)

    # 如果是运算符节点，需要加括号
    if root.value in '+-*/':
        return ['('] + left +
[root.value] + right + [')']
    else:
        return [root.value]

def expr_tree_postorder(root):
    """表达式树后序遍历（后缀表达式）"""
    if not root:
        return []
    return
expr_tree_postorder(root.left) +
expr_tree_postorder(root.right) +
[root.value]
```

3.2 表达式树计算

```
def evaluate_expr_tree(root):
    """计算表达式树的值（支持整数、浮点数、
加减乘除）"""
    if not root:
        return 0

    # 叶子节点：操作数
    if root.value.replace('.',
'').isdigit():
        return float(root.value) if
'.' in root.value else int(root.value)

    # 内部节点：递归计算左右子树
    left_val =
evaluate_expr_tree(root.left)
    right_val =
evaluate_expr_tree(root.right)

    # 根据运算符计算
    if root.value == '+':
        return left_val + right_val
    elif root.value == '-':
        return left_val - right_val
    elif root.value == '*':
        return left_val * right_val
    elif root.value == '/':
        return left_val / right_val
    else:
        raise ValueError(f"未知运算符:
{root.value}")
```

```
# 优化版本：使用字典映射运算符
def
evaluate_expr_tree_optimized(root):
    """使用运算符映射的优化版本"""
    import operator
    ops = {
        '+': operator.add,
        '-': operator.sub,
        '*': operator.mul,
        '/': operator.truediv
    }

    def eval_node(node):
        if not node:
            return 0
        if node.value.replace('.',
'').isdigit():
            return float(node.value)
if '.' in node.value else
int(node.value)
        if node.value in ops:
            return ops[node.value]
(eval_node(node.left),
eval_node(node.right))
        raise ValueError(f"未知节点:
{node.value}")

    return eval_node(root)
```

6 表达式相关题目模板

6.3 布尔表达式化简（20576题）

```
def simplify_boolean_expression(expr):
    """化简布尔表达式，去除不必要的括号"""
    # 使用Python的ast模块解析布尔表达式
    import ast

    # 优先级映射
    precedence = {'or': 1, 'and': 2,
'not': 3}

    def parse_ast(node):
        """解析AST节点"""
        if isinstance(node,
ast.BoolOp):
            op = 'and' if
isinstance(node.op, ast.And) else 'or'
            children =
[parse_ast(child) for child in
node.values]
            # 按优先级决定是否加括号
            return (op, children)
        elif isinstance(node,
ast.UnaryOp) and isinstance(node.op,
ast.Not):
            return ('not',
[parse_ast(node.operand)])
        elif isinstance(node,
ast.Constant):
            return str(node.value)
        elif hasattr(ast,
'NameConstant') and isinstance(node,
ast.NameConstant):
            return str(node.value)
        return str(node)

    def format_expr(expr_tuple,
parent_prec=0):
        """格式化表达式，根据优先级加括
号"""
        if isinstance(expr_tuple,
str):
            return expr_tuple

        op, children = expr_tuple
        current_prec = precedence[op]

        if op == 'not':
            child_expr =
format_expr(children[0], current_prec)
            # 子表达式优先级低时需要加括号
            if isinstance(children[0],
tuple):
                child_op, _ =
children[0]
                if
precedence[child_op] < current_prec:
                    child_expr =
f'({child_expr})'
            return f'not {child_expr}'
        else:  # 'and' 或 'or'
            parts = []
            for child in children:
                child_expr =
format_expr(child, current_prec)
                # 子表达式优先级低时需要
加括号
                if isinstance(child,
tuple):
                    child_op, _ =
child
                    if
```

```
precedence[child_op] < current_prec:
                        child_expr =
f'({child_expr})'

parts.append(child_expr)

            result = f' {op}
'.join(parts)
            if current_prec <
parent_prec:
                result = f'({result})'
            return result

    # 解析表达式
    tree = ast.parse(expr.strip(),
mode='eval')
    parsed = parse_ast(tree.body)
    return format_expr(parsed)
```

7 综合应用模板

7.2 表达式树可视化（调试用）

```
def print_expr_tree(root, depth=0,
prefix=""):
    """打印表达式树结构（用于调试）"""
    if not root:
        return

    # 当前节点
    print("  " * depth + prefix +
str(root.value))

    # 左右子树
    if root.left or root.right:
        if root.left:
            print_expr_tree(root.left,
depth + 1, "L: ")
        else:
            print("  " * (depth + 1) +
"L: None")

        if root.right:

print_expr_tree(root.right, depth + 1,
"R: ")
        else:
            print("  " * (depth + 1) +
"R: None")
```

```
# 使用示例
# expr = "3+4*5"
# postfix = infix_to_postfix(expr)
# tree =
build_expr_tree_from_postfix(postfix)
# print_expr_tree(tree)
```

机考实用技巧

2. 常见错误避免

```
# 1. 递归深度限制
import sys
sys.setrecursionlimit(1000000)  # 设置
递归深度限制

# 2. 判断节点是否为叶子节点
def is_leaf(node):
    return node and not node.left and
not node.right

# 3. 空树处理
if not root: return []  # 而不是 return
None

# 4. BST插入重复值处理
def BST_insert(root, val):
    if not root: return TreeNode(val)
    if val <= root.val:  # 根据题目决定
如何处理重复值
        root.left =
BST_insert(root.left, val)
    else:
        root.right =
BST_insert(root.right, val)
    return root
```

3. 调试输出技巧

```
# 打印二叉树（层序遍历格式）
def print_tree(root):
    if not root: print("Empty tree")
    queue = [root]
    result = []
    while queue:
        node = queue.pop(0)
        result.append(node.val if node
else None)
        if node:
```

```
            queue.append(node.left)
            queue.append(node.right)
    print(result)

# 打印堆内容
def print_heap(heap):
    print([heapq.heappop(heap) for _
in range(len(heap))])
```

这个模板包含了树和堆的核心算法、高频题型实现以及机考实用技巧。建议你在复习时：

1. **理解每个模板的原理**，而不是死记硬背
2. **针对薄弱环节**，重点练习相关模板
3. **机考前**，快速浏览模板，记住关键函数的签名和用途
4. **考试时**，根据题目要求选择合适的模板进行修改

需要我针对某个特定题型或算法提供更详细的解释吗？

图

📘 图论机考速查手册（Python 模板）

一、图的存储与表示

1.1 邻接表（最常用）

```
# 无向图
n, m = map(int, input().split())
adj = [[] for _ in range(n)]
for _ in range(m):
    u, v = map(int, input().split())
    adj[u].append(v)
    adj[v].append(u)  # 有向图去掉这一行

# 带权图
adj = [[] for _ in range(n)]
for _ in range(m):
    u, v, w = map(int,
input().split())
    adj[u].append((v, w))
    # 无向图加上：adj[v].append((u, w))
```

1.2 邻接矩阵

```
n, m = map(int, input().split())
matrix = [[0]*n for _ in range(n)]
for _ in range(m):
    u, v = map(int, input().split())
    matrix[u][v] = 1
    # 无向图加上：matrix[v][u] = 1
```

1.3 OOP 实现（笔试专用）

```
class Vertex:
    def __init__(self, key):
        self.key = key
        self.neighbors = {}  #
{neighbor_vertex: weight}
        self.color = "white"
        self.distance = float('inf')
        self.previous = None

    def add_neighbor(self, nbr,
weight=0):
        self.neighbors[nbr] = weight

class Graph:
    def __init__(self):
        self.vertices = {}

    def add_vertex(self, key):
        self.vertices[key] =
Vertex(key)

    def add_edge(self, f, t,
weight=0):
        if f not in self.vertices:
            self.add_vertex(f)
        if t not in self.vertices:
            self.add_vertex(t)

self.vertices[f].add_neighbor(self.ver
tices[t], weight)
        # 无向图加上：
self.vertices[t].add_neighbor(self.ver
tices[f], weight)
```

二、广度优先搜索（BFS）

2.1 标准 BFS 模板（求层号/最短步数）

```
from collections import deque

def bfs(start, adj):
    n = len(adj)
    visited = [False] * n
    distance = [-1] * n  # 层号/最短距离
    parent = [-1] * n    # 记录路径

    queue = deque([start])
    visited[start] = True
    distance[start] = 0

    while queue:
        node = queue.popleft()
        for neighbor in adj[node]:
            if not visited[neighbor]:
                visited[neighbor] = True
                distance[neighbor] = distance[node] + 1
                parent[neighbor] = node
                queue.append(neighbor)

    return distance, parent

# 重建路径
def reconstruct_path(start, end, parent):
    path = []
    curr = end
    while curr != -1:
        path.append(curr)
        curr = parent[curr]
    path.reverse()
    return path if path[0] == start else []  # 确保路径有效
```

2.2 迷宫最短路径（带坐标）

```
from collections import deque

def bfs_maze(grid, start, end):
    # grid: 0可通过，1为墙
    n, m = len(grid), len(grid[0])
    directions = [(0,1),(0,-1),(1,0),(-1,0)]
    visited = [[False]*m for _ in range(n)]
    parent = [[None]*m for _ in range(n)]

    queue = deque([start])
    visited[start[0]][start[1]] = True

    while queue:
        x, y = queue.popleft()
        if (x, y) == end:
            break
        for dx, dy in directions:
            nx, ny = x+dx, y+dy
            if 0<=nx<n and 0<=ny<m and grid[nx][ny]==0 and not visited[nx][ny]:
                visited[nx][ny] = True
                parent[nx][ny] = (x, y)
                queue.append((nx, ny))

    # 重建路径
    path = []
    curr = end
    while curr:
        path.append(curr)
        curr = parent[curr[0]][curr[1]]
    path.reverse()
    return path
```

三、深度优先搜索（DFS）

3.1 递归 DFS（基础）

```
def dfs(node, adj, visited):
    visited[node] = True
    # 前序处理
    for neighbor in adj[node]:
        if not visited[neighbor]:
            dfs(neighbor, adj, visited)
    # 后序处理

# 计算连通块个数
def count_components(adj):
    n = len(adj)
    visited = [False] * n
    count = 0
```

```
    for i in range(n):
        if not visited[i]:
            dfs(i, adj, visited)
            count += 1
    return count
```

3.2 带时间戳的 DFS（拓扑排序基础）

```
time = 0
def dfs_timestamp(node, adj, visited,
disc, fin):
    global time
    time += 1
    disc[node] = time
    visited[node] = True

    for neighbor in adj[node]:
        if not visited[neighbor]:
            dfs_timestamp(neighbor,
adj, visited, disc, fin)

    time += 1
    fin[node] = time

# 初始化
n = len(adj)
visited = [False] * n
disc = [0] * n    # 发现时间
fin = [0] * n     # 结束时间
for i in range(n):
    if not visited[i]:
        dfs_timestamp(i, adj, visited,
disc, fin)
```

3.3 三色法判环（有向图）

```
# 0:未访问, 1:访问中, 2:已访问
def has_cycle(adj):
    n = len(adj)
    color = [0] * n

    def dfs(node):
        color[node] = 1  # 标记为访问中
        for neighbor in adj[node]:
            if color[neighbor] == 0:
                if dfs(neighbor):
                    return True
            elif color[neighbor] == 1:
# 遇到后向边，有环
                return True
        color[node] = 2  # 标记为已访问
        return False

    for i in range(n):
        if color[i] == 0:
            if dfs(i):
                return True
    return False
```

3.4 回溯模板（骑士周游/八皇后类）

```
def backtrack(path, current, visited,
...):
    # 终止条件
    if len(path) == target_length:
        return True  # 或存储结果

    visited[current] = True
    path.append(current)

    # 尝试所有可能的下一个位置（可排序优
化）
    for next_node in
sorted(adj[current]):  # Warnsdorff启
发式
        if not visited[next_node]:
            if backtrack(path,
next_node, visited, ...):
                return True

    # 回溯
    visited[current] = False
    path.pop()
    return False
```

四、拓扑排序

4.1 DFS 版本（基于结束时间）

```
def topological_sort_dfs(adj):
    n = len(adj)
    visited = [False] * n
    stack = []  # 存储按结束时间排序的节
点

    def dfs(node):
        visited[node] = True
        for neighbor in adj[node]:
            if not visited[neighbor]:
                dfs(neighbor)
```

```
        stack.append(node)  # 后序添加

    for i in range(n):
        if not visited[i]:
            dfs(i)

    return stack[::-1]  # 反转得到拓扑序
```

4.2 BFS 版本（Kahn算法）

```
from collections import deque

def topological_sort_bfs(adj):
    n = len(adj)
    in_degree = [0] * n

    # 计算入度
    for u in range(n):
        for v in adj[u]:
            in_degree[v] += 1

    # 初始化队列（入度为0的节点）
    queue = deque([i for i in range(n)
if in_degree[i] == 0])
    topo_order = []

    while queue:
        node = queue.popleft()
        topo_order.append(node)
        for neighbor in adj[node]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] ==
0:
                queue.append(neighbor)

    # 检查是否有环
    if len(topo_order) != n:
        print("图中有环，无法拓扑排序")
        return []

    return topo_order
```

五、最短路径算法

5.1 Dijkstra算法（无负权边）

```
import heapq

def dijkstra(adj, start, end=None):
    """
    adj: 邻接表，adj[u] = [(v, w), ...]
    start: 起点
    end: 终点（可选）
    返回：dist数组，从start到各点的最短距
离
    """
    n = len(adj)
    dist = [float('inf')] * n
    dist[start] = 0
    heap = [(0, start)]
    visited = [False] * n

    while heap:
        d, u = heapq.heappop(heap)
        if visited[u]:
            continue
        visited[u] = True

        # 如果只求到某点的距离，可以提前结
束
        if end is not None and u ==
end:
            return dist[end]

        for v, w in adj[u]:
            if not visited[v] and d +
w < dist[v]:
                dist[v] = d + w
                heapq.heappush(heap,
(dist[v], v))

    return dist

# 重建路径
def dijkstra_with_path(adj, start,
end):
    n = len(adj)
    dist = [float('inf')] * n
    dist[start] = 0
    prev = [-1] * n
    heap = [(0, start)]

    while heap:
        d, u = heapq.heappop(heap)
        if d > dist[u]:
            continue
        if u == end:
            break
        for v, w in adj[u]:
            if d + w < dist[v]:
                dist[v] = d + w
                prev[v] = u
```

```
                heapq.heappush(heap,
(dist[v], v))

    # 重建路径
    path = []
    curr = end
    while curr != -1:
        path.append(curr)
        curr = prev[curr]
    path.reverse()
    return dist[end] if dist[end] !=
float('inf') else -1, path
```

5.2 Bellman-Ford算法（可处理负权边）

```
def bellman_ford(edges, n, start):
    """
    edges: 边列表，每个元素为(u, v, w)
    n: 顶点数
    start: 起点
    返回：dist数组，如果存在负权环则返回
None
    """
    dist = [float('inf')] * n
    dist[start] = 0

    # 松弛V-1次
    for _ in range(n - 1):
        for u, v, w in edges:
            if dist[u] != float('inf')
and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w

    # 检测负权环
    for u, v, w in edges:
        if dist[u] != float('inf') and
dist[u] + w < dist[v]:
            print("存在负权环")
            return None

    return dist
```

5.3 SPFA算法（Bellman-Ford的优化）

```
from collections import deque

def spfa(adj, n, start):
    """
    adj: 邻接表，adj[u] = [(v, w), ...]
    n: 顶点数
    start: 起点
    返回：dist数组，如果存在负权环则返回
None
    """
    dist = [float('inf')] * n
    dist[start] = 0
    in_queue = [False] * n
    cnt = [0] * n  # 入队次数，用于检测负
权环

    queue = deque([start])
    in_queue[start] = True
    cnt[start] += 1

    while queue:
        u = queue.popleft()
        in_queue[u] = False

        for v, w in adj[u]:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                if not in_queue[v]:
                    queue.append(v)
                    in_queue[v] = True
                    cnt[v] += 1
                    if cnt[v] >= n:  #
存在负权环
                        print("存在负权
环")
                        return None

    return dist
```

5.4 Floyd-Warshall算法（多源最短路径）

```
def floyd_warshall(n, edges):
    """
    n: 顶点数
    edges: 边列表，每个元素为(u, v, w)
    返回：dist矩阵，dist[i][j]表示i到j的
最短距离
    """
    INF = float('inf')
    dist = [[INF] * n for _ in
range(n)]

    # 初始化
    for i in range(n):
        dist[i][i] = 0
    for u, v, w in edges:
```

```
        dist[u][v] = min(dist[u][v],
w)  # 处理重边

    # 动态规划
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][k] != INF
and dist[k][j] != INF:
                    dist[i][j] =
min(dist[i][j], dist[i][k] + dist[k]
[j])

    return dist
```

六、最小生成树

6.1 Prim算法

```
import heapq

def prim(adj):
    """
    adj: 邻接表，adj[u] = [(v, w), ...]
    返回：最小生成树的总权重，如果图不连通则
返回-1
    """
    n = len(adj)
    visited = [False] * n
    min_heap = [(0, 0)]  # (weight,
vertex)
    total_weight = 0
    edges_used = 0

    while min_heap and edges_used < n:
        weight, u =
heapq.heappop(min_heap)
        if visited[u]:
            continue
        visited[u] = True
        total_weight += weight
        edges_used += 1

        for v, w in adj[u]:
            if not visited[v]:

heapq.heappush(min_heap, (w, v))
```

```
    return total_weight if edges_used
== n else -1
```

6.2 Kruskal算法（配合并查集）

```
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    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):
        px, py = self.find(x),
self.find(y)
        if px == py:
            return False
        if self.rank[px] <
self.rank[py]:
            self.parent[px] = py
        elif self.rank[px] >
self.rank[py]:
            self.parent[py] = px
        else:
            self.parent[py] = px
            self.rank[px] += 1
        return True

def kruskal(n, edges):
    """
    n: 顶点数
    edges: 边列表，每个元素为(u, v, w)
    返回：最小生成树的总权重，如果图不连通则
返回-1
    """
    uf = UnionFind(n)
    edges.sort(key=lambda x: x[2])  #
按权重排序
    total_weight = 0
    edges_used = 0

    for u, v, w in edges:
        if uf.union(u, v):
            total_weight += w
            edges_used += 1
            if edges_used == n - 1:
                break
```

```
    return total_weight if edges_used
== n - 1 else -1
```

七、强连通分量（SCC）

7.1 Kosaraju算法

```
def kosaraju(adj):
    """
    adj: 邻接表
    返回：SCC列表，每个SCC是一个顶点列表
    """
    n = len(adj)
    visited = [False] * n
    order = []

    # 第一次DFS，记录完成时间
    def dfs1(u):
        visited[u] = True
        for v in adj[u]:
            if not visited[v]:
                dfs1(v)
        order.append(u)

    for i in range(n):
        if not visited[i]:
            dfs1(i)

    # 构建反向图
    rev_adj = [[] for _ in range(n)]
    for u in range(n):
        for v in adj[u]:
            rev_adj[v].append(u)

    # 第二次DFS，按照完成时间逆序
    visited = [False] * n
    scc_list = []

    def dfs2(u, component):
        visited[u] = True
        component.append(u)
        for v in rev_adj[u]:
            if not visited[v]:
                dfs2(v, component)

    for u in reversed(order):
        if not visited[u]:
            component = []
            dfs2(u, component)
            scc_list.append(component)

    return scc_list
```

7.2 Tarjan算法（更高效）

```
def tarjan(adj):
    n = len(adj)
    index = 0
    stack = []
    indices = [-1] * n
    low_link = [-1] * n
    on_stack = [False] * n
    scc_list = []

    def strongconnect(u):
        nonlocal index
        indices[u] = low_link[u] =
index
        index += 1
        stack.append(u)
        on_stack[u] = True

        for v in adj[u]:
            if indices[v] == -1:
                strongconnect(v)
                low_link[u] =
min(low_link[u], low_link[v])
            elif on_stack[v]:
                low_link[u] =
min(low_link[u], indices[v])

        # 如果u是SCC的根
        if low_link[u] == indices[u]:
            component = []
            while True:
                v = stack.pop()
                on_stack[v] = False
                component.append(v)
                if v == u:
                    break
            scc_list.append(component)

    for u in range(n):
        if indices[u] == -1:
            strongconnect(u)

    return scc_list
```

八、关键路径（AOE网络）

```python
from collections import deque

def critical_path(n, edges):
    """
    n: 顶点数（事件数）
    edges: 活动列表，每个元素为(u, v, w)表示从u到v的活动，耗时w
    返回：关键路径长度，关键活动列表
    """
    # 构建邻接表
    adj = [[] for _ in range(n)]
    indegree = [0] * n
    outdegree = [0] * n

    for u, v, w in edges:
        adj[u].append((v, w))
        indegree[v] += 1
        outdegree[u] += 1

    # 拓扑排序，计算最早开始时间
    ve = [0] * n  # 事件最早开始时间
    topo_order = []
    q = deque([i for i in range(n) if indegree[i] == 0])

    while q:
        u = q.popleft()
        topo_order.append(u)
        for v, w in adj[u]:
            ve[v] = max(ve[v], ve[u] + w)
            indegree[v] -= 1
            if indegree[v] == 0:
                q.append(v)

    if len(topo_order) != n:
        return -1, []  # 有环

    # 计算最晚开始时间
    vl = [float('inf')] * n
    vl[n-1] = ve[n-1]  # 假设终点是最后一个顶点

    for u in reversed(topo_order):
        for v, w in adj[u]:
            vl[u] = min(vl[u], vl[v] - w)

    # 计算关键活动
    critical_edges = []
    for u, v, w in edges:
        e = ve[u]  # 活动最早开始时间
        l = vl[v] - w  # 活动最晚开始时间
        if e == l:
            critical_edges.append((u, v, w))

    return ve[n-1], critical_edges
```

九、并查集（Union-Find）

```python
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    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):
        px, py = self.find(x), self.find(y)
        if px == py:
            return False
        # 按秩合并
        if self.rank[px] < self.rank[py]:
            self.parent[px] = py
        elif self.rank[px] > self.rank[py]:
            self.parent[py] = px
        else:
            self.parent[py] = px
            self.rank[px] += 1
        return True

    def connected(self, x, y):
        return self.find(x) == self.find(y)
```

十、实用工具函数

10.1 度数计算

```python
# 无向图度数
def calculate_degrees(edges, n):
    degrees = [0] * n
    for u, v in edges:
        degrees[u] += 1
        degrees[v] += 1
    return degrees

# 有向图入度出度
def calculate_in_out_degrees(edges,
n):
    in_deg = [0] * n
    out_deg = [0] * n
    for u, v in edges:
        out_deg[u] += 1
        in_deg[v] += 1
    return in_deg, out_deg
```

10.2 邻接表转邻接矩阵

```python
def adj_list_to_matrix(adj, n):
    matrix = [[0]*n for _ in range(n)]
    for u in range(n):
        for v in adj[u]:
            matrix[u][v] = 1
    return matrix
```

10.3 无向图环检测

```python
def has_cycle_undirected(adj):
    n = len(adj)
    visited = [False] * n

    def dfs(u, parent):
        visited[u] = True
        for v in adj[u]:
            if not visited[v]:
                if dfs(v, u):
                    return True
            elif v != parent:  # 遇到已
访问的节点且不是父节点
                return True
        return False

    for i in range(n):
        if not visited[i]:
            if dfs(i, -1):
                return True
    return False
```

📌 使用建议

1. **根据题目选择算法：**
   - 最短路径：
     - 无权图 → BFS
     - 带权图（无负权边）→ Dijkstra
     - 带权图（可能有负权边）→ Bellman-Ford或SPFA
     - 多源最短路径 → Floyd-Warshall
   - 最小生成树：
     - 稠密图 → Prim
     - 稀疏图 → Kruskal
   - 依赖关系 → 拓扑排序
   - 环检测 → 三色法DFS（有向图）或Union-Find（无向图）
   - 强连通分量 → Kosaraju或Tarjan
2. **邻接表 vs 邻接矩阵：**
   - 大多数情况用邻接表（节省空间）
   - 稠密图或频繁查边用邻接矩阵
3. **笔试注意：**
   - 掌握OOP实现（Vertex + Graph类）
   - 理解DFS回溯的状态恢复
   - 注意有向图/无向图的区别
4. **调试技巧：**
   - 打印visited/distance数组查看状态
   - 小规模数据手动画图验证
   - 使用print语句记录递归深度

祝你在机考中取得好成绩！ 🚀