线性表

1. 顺序表 (Sequential List)

Python 内置的 `list` 类型就是一种动态数组，可以看作是顺序表的一种高效实现。它支持动态扩容，并提供了丰富的操作方法。Python 的 `list` 在底层已经处理了内存管理和动态调整大小等复杂性，因此直接使用它通常是实现顺序表的最佳选择。利用`deque`双向链表可以高效地实现栈、队列等常见数据结构。

```
# 创建一个空列表 (顺序表)
my_list = []

# 添加元素 (在末尾)
my_list.append(10)
my_list.append(20)
my_list.append(30)
# 列表内容: [10, 20, 30]

# 在指定位置插入元素
my_list.insert(1, 15) # 在索引1处插入15
# 插入后: [10, 15, 20, 30]

# 访问元素
element = my_list[2]
# 索引为2的元素: 20

# 修改元素
my_list[0] = 5
# 修改后: [5, 15, 20, 30]

# 删除元素 (按值)
my_list.remove(20)
# 删除20后: [5, 15, 30]

# 删除元素 (按索引)
deleted_element = my_list.pop(1)
# 删除索引1的元素: 15, 列表变为: [5, 30]

# 获取列表长度
length = len(my_list)
# 列表长度: 2

# 遍历列表
print("遍历列表:")
for item in my_list:
    print(item)
# 输出:
# 5
# 30
```

2. 链表 (Linked List)

链表通过节点之间的指针连接来表示逻辑顺序，每个节点包含数据和指向下一个（或上一个）节点的引用。链表可以使用 Python 内置的`deque`双向链表类实现。

```
from collections import deque

# 创建deque

d = deque()
# 创建空deque
d = deque([1, 2, 3, 4])
# 从可迭代对象创建
d = deque(maxlen=5)
# 创建有最大长度限制的deque

# 加元素操作

d.append(5)
# 右侧添加：deque([1, 2, 3, 4, 5])
d.appendleft(0)
# 左侧添加：deque([0, 1, 2, 3, 4, 5])
d.extend([6, 7])
# 右侧扩展：deque([0, 1, 2, 3, 4, 5, 6, 7])
d.extendleft([-1, -2])
# 左侧扩展：deque([-2, -1, 0, 1, 2, 3, 4, 5, 6, 7])

# 删除元素操作

right_item = d.pop()
# 右侧删除，返回7
left_item = d.popleft()
# 左侧删除，返回-2
d.remove(3)
# 删除第一个匹配值3

# 访问和查询操作

first_item = d[0]
# 访问第一个元素
last_item = d[-1]
# 访问最后一个元素
length = len(d)
# 获取长度
count = d.count(1)
# 计数元素1出现的次数
index = list(d).index(2)
# 获取元素2的索引(注意：deque没有直接index方法)

# 旋转和反转操作
```

```
d = deque([1, 2, 3, 4, 5])
d.rotate(2)
# 向右旋转2步
# 旋转后：deque([4, 5, 1, 2, 3])
d.rotate(-1)
# 向左旋转1步
d.reverse()
# 反转deque

# 其他操作

d.clear()
# 清空deque
d.copy()
# 创建浅拷贝(需要Python 3.5+)
max_len = d.maxlen
# 获取最大长度(如果没有设置则为None)

# 转换为其他类型

list_from_deque = list(d)
# 转换为列表
tuple_from_deque = tuple(d)
# 转换为元组

# 迭代操作

for item in d:
    print(item)
# 迭代deque

# 检查元素存在

if 2 in d:
    print("2 exists in deque")
# 检查元素是否存在
```

三、线性结构中的常见算法

1. 双指针 (Two Pointers)

双指针是一种非常高效的技巧，通常用于在有序或部分有序的线性结构中查找元素对、反转序列、或者进行分区等操作。

- 快慢指针 (Fast and Slow Pointers):

  一个指针移动得快，另一个指针移动得慢。常用于：

  - **检测链表中的环:** 快指针一次走两步，慢指针一次走一步，如果相遇则有环。
  - **寻找链表的中点:** 快指针走到末尾时，慢指针正好在中间。
  - **滑动窗口 (Sliding Window):** 见下一条。

- 左右指针 (Left and Right Pointers):

  一个指针从序列的开头开始，另一个从末尾开始，然后向中间移动。常用于：

  - **反转数组/字符串:** 不断交换左右指针指向的元素。
  - **二分查找 (Binary Search) 的变种:** 在有序数组中查找满足特定条件的元素对 (例如，两数之和)。
  - **快速排序 (Quick Sort) 的分区操作。**

示例 (反转字符串):

```
def reverse_string(s: list[str]) -> None:
    left, right = 0, len(s) - 1
    while left < right:
        s[left], s[right] = s[right], s[left]
        left += 1
        right -= 1
```

2. 滑动窗口 (Sliding Window)

滑动窗口通常用于解决在给定数组或字符串中找到满足特定条件的**连续子序列**的问题。窗口的大小可以是固定的，也可以是可变的。

- **固定大小窗口:** 窗口大小不变，在序列上滑动。
- **可变大小窗口:** 窗口大小根据条件动态调整，通常用两个指针（窗口的左右边界）来维护。

核心思想:

1. 初始化窗口的左右边界 (通常都在序列的起始位置)。
2. 向右扩展右边界，将新元素纳入窗口。
3. 判断当前窗口内的元素是否满足题目条件。
4. 如果不满足，或者窗口大小超过限制，则向右收缩左边界，将元素移出窗口，直到满足条件为止。
5. 重复步骤 2-4，直到右边界到达序列末尾。

示例 (找到定长子数组的最大和):

```
def max_sum_subarray_fixed_size(arr: list[int], k: int) -> int:
    if not arr or k <= 0 or k > len(arr):
        return 0

    current_sum = sum(arr[:k])
    max_s = current_sum

    for i in range(k, len(arr)):
        current_sum = current_sum - arr[i-k] + arr[i] # 滑动
        max_s = max(max_s, current_sum)

    return max_s
```

3. 前缀和 (Prefix Sum)

前缀和数组 `prefix_sum[i]` 存储的是原数组从索引 `0` 到 `i` (包含 `i`) 的所有元素的和。通过预先计算前缀和，可以在 O(1) 的时间内快速查询任意子数组的和。

- `sum(array[i...j]) = prefix_sum[j] - prefix_sum[i-1]` (如果 `i > 0`)
- `sum(array[0...j]) = prefix_sum[j]`

应用场景:

- 频繁查询子数组的和。
- 解决与子数组和相关的问题，例如“和为K的子数组”。

示例 (计算前缀和):

```
def calculate_prefix_sum(arr: list[int]) -> list[int]:
    if not arr:
        return []
    prefix_sum = [0] * len(arr)
    prefix_sum[0] = arr[0]
    for i in range(1, len(arr)):
        prefix_sum[i] = prefix_sum[i-1] + arr[i]
    return prefix_sum

def range_sum_query(prefix_sum_arr: list[int], i: int, j: int) -> int:
    if i == 0:
        return prefix_sum_arr[j]
    else:
        return prefix_sum_arr[j] - prefix_sum_arr[i-1]
```

4. 差分数组 (Difference Array)

差分数组 `diff[i]` 记录的是原数组 `arr[i]` 与 `arr[i-1]` 的差值 (通常 `diff[0] = arr[0]`)。

- `diff[0] = arr[0]`
- `diff[i] = arr[i] - arr[i-1]` for `i > 0`

核心优势: 当需要对原数组的一个**区间**进行频繁的**增减**操作时，差分数组非常高效。对 `arr[L...R]` 区间内的所有元素增加 `val`，只需要修改差分数组的两个值：

- `diff[L] += val`
- `diff[R+1] -= val` (如果 `R+1` 在数组范围内)

修改完差分数组后，可以通过差分数组反向构造出新的原数组（本质上是前缀和的逆运算）。

应用场景:

- 对数组的区间进行频繁的加减操作，然后查询最终数组的某个元素或整个数组。

示例 (区间增加):

```
def update_range_with_difference_array(original_arr_len: int, updates: list[tuple[int,
int, int]]) -> list[int]:
    diff = [0] * (original_arr_len + 1) # 长度多1方便处理边界
```

```
    # 假设原数组初始全为0，或者可以先根据原数组构建初始差分数组
    # 这里为了简化，假设原数组初始为0

    for start, end, val in updates: # [start, end] 闭区间
        diff[start] += val
        if end + 1 < original_arr_len: # 注意边界
            diff[end + 1] -= val

    # 从差分数组还原结果数组 (本质是求前缀和)
    result_arr = [0] * original_arr_len
    result_arr[0] = diff[0]
    for i in range(1, original_arr_len):
        result_arr[i] = result_arr[i-1] + diff[i]

    return result_arr
```

字符串

一、字符串相关操作

- **创建字符串**
  可以通过单引号或双引号创建字符串：
  `string1 = 'Hello'`
  `string2 = "World"`
- **访问字符**
  使用索引访问特定字符：
  `first_char = string1[0] # 结果为 'H'`
  `last_char = string1[-1] # 结果为 'o'`
- **切片**
  切片允许获取字符串的部分内容：
  `substring = string1[1:4] # 结果为 'ell'`
- **字符串连接**
  使用加号 (+) 运算符连接字符串：
  `greeting = string1 + ' ' + string2 # 结果为 'Hello World'`
- **字符串重复**
  使用乘号 (*) 运算符重复字符串：
  `repeated = string1 * 3 # 结果为 'HelloHelloHello'`
- **字符串长度**
  使用 `len()` 函数获取字符串的长度：
  `length = len(string1) # 结果为 5`
- **查找和替换**
  `new_string = string1.replace('o', 'a') # 结果为 'Hella'`（替换所有）
- **字符串分割**
  使用 `split()` 方法根据指定分隔符分割字符串：
  `words = greeting.split(' ') # 结果为 ['Hello', 'World']`
- **字符串连接**
  使用 `join()` 方法连接字符串列表：
  `joined = ' '.join(words) # 结果为 'Hello World'`
- **去除空格**
  使用 `strip()` 方法去除字符串两端的空格：
  `trimmed = ' Hello '.strip() # 结果为 'Hello'`
- **大小写转换**
  使用 `upper()`、`lower()`、`capitalize()`、`title()` 方法进行大小写转换：
  `upper_case = string1.upper() # 结果为 'HELLO'`
  `lower_case = string1.lower() # 结果为 'hello'`
  `capitalized = string1.capitalize() # 结果为 'Hello'`
  `titled = string1.title() # 结果为 'Hello'`

- **字符串反转**
  使用切片反转字符串：
  `reversed_string = string1[::-1] # 结果为 'olleH'`

二、f-string

- **概述**
  f-字符串（格式化字符串字面量）是 Python 3.6 引入的一种简洁且直观的字符串格式化方法。通过在字符串前加 `f` 或 `F`，可以直接在字符串中使用大括号 `{}` 插入变量和表达式。
- **基本语法**
  使用 `f"格式字符串 {变量}"` 进行格式化。示例：
  `name = "Alice"`
  `age = 30`
  `print(f"My name is {name} and I am {age} years old.")`
- **表达式支持**
  f-字符串可以直接在 `{}` 中使用表达式。例如：
  `print(f"Next year, I will be {age + 1} years old.")`
- **数字格式化**
  支持格式说明符，可以控制小数位数、对齐和填充。

参数	描述	示例
`:d`	整数格式（十进制）	`f"{value:d}"`
`:b`	二进制格式	`f"{value:b}"`
`:o`	八进制格式	`f"{value:o}"`
`:x`	十六进制格式（小写）	`f"{value:x}"`
`:X`	十六进制格式（大写）	`f"{value:X}"`
`<`	左对齐	`f"{text:<10}"`
`>`	右对齐	`f"{text:>10}"`
`^`	居中对齐	`f"{text:^10}"`
`:e`	科学计数法（小写）	`f"{value:e}"`
`:E`	科学计数法（大写）	`f"{value:E}"`
`:f`	小数格式（小写）	`f"{value:f}"`
`:F`	小数格式（大写）	`f"{value:F}"`
`:#.#`	指定宽度和小数位数	`f"{value:10.2f}"`
`:,`	使用千分位分隔符	`f"{number:,}"`
`#`	强制显示小数点和尾随零	`f"{value:#.2f}"`

示例：

```
value = 255
number = 1234567.89
text = "Hello"

print(f"{value:d}")   # 输出 "255"
print(f"{value:b}")   # 输出 "11111111"
print(f"{value:o}")   # 输出 "377"
print(f"{value:x}")   # 输出 "ff"
print(f"{number:,.2f}")   # 输出 "1,234,567.89"
print(f"{number:<10.2f}")   # 左对齐，输出 "1234567.89 "
print(f"{number:>10.2f}")   # 右对齐，输出 "  1234567.89"
print(f"{number:^10.2f}")   # 居中对齐，输出 "1234567.89 "
```

- **对齐与填充**
  可以控制对齐方式和填充字符：
  `value = "test"`
  `print(f"|{value:<10}|") # 左对齐`
  `print(f"|{value:>10}|") # 右对齐`
  `print(f"|{value:^10}|") # 居中对齐`
- **字典和列表支持**
  可以直接格式化字典和列表元素：
  `data = {"name": "Alice", "age": 30}`
  `print(f"Name: {data['name']}, Age: {data['age']}")`
- **多行字符串**
  支持多行字符串格式化：
  `multi_line = f"""My name is {name}.`
  `I am {age} years old."""`

三、KMP算法

KMP算法通过**预处理模式串**生成`next`数组（部分匹配表），利用已匹配的信息避免主串指针回退。其核心步骤如下：

1. **预处理模式串**：构建`next`数组，记录每个位置的最长公共前后缀长度。
2. **匹配过程**：当字符不匹配时，模式串指针根据`next`数组回退，主串指针不动。

   如下给出了一个 KMP 算法的示例：

```
# KMP (Knuth-Morris-Pratt) 字符串匹配算法

def compute_lps(pattern: str) -> list[int]:
    """
    计算模式串的 LPS (Longest Proper Prefix which is also Suffix) 数组。
    LPS 数组用于记录在模式串中，每个位置的最长公共前后缀的长度。

    参数：
        pattern: 模式字符串。
```

```
    返回:
        一个列表，表示模式串的 LPS 数组。
    """
    m = len(pattern)
    lps = [0] * m  # 初始化 LPS 数组，所有元素的初始值为 0
    length = 0  # 当前最长公共前后缀的长度
    i = 1  # 从模式串的第二个字符开始遍历 (索引为 1)

    # 循环遍历模式串，计算 LPS 值
    while i < m:
        if pattern[i] == pattern[length]:
            # 如果当前字符与 length 位置的字符匹配
            length += 1  # 最长公共前后缀长度增加
            lps[i] = length  # 记录当前位置的 LPS 值
            i += 1  # 继续比较下一个字符
        else:
            # 如果当前字符与 length 位置的字符不匹配
            if length != 0:
                # 如果 length 不为 0，说明之前有匹配的前后缀
                # 我们需要回溯到上一个可能的匹配位置
                # lps[length - 1] 存储了上一个字符位置的最长公共前后缀长度
                # 这也是我们下一个尝试匹配的 length 值
                length = lps[length - 1]
            else:
                # 如果 length 为 0，说明当前字符没有匹配的前缀
                lps[i] = 0  # 当前位置的 LPS 值为 0
                i += 1  # 继续比较下一个字符
    return lps

def search(text: str, pattern: str) -> list[int]:
    """
    使用 KMP 算法在文本串中查找模式串的所有出现位置。

    参数:
        text: 文本字符串。
        pattern: 模式字符串。

    返回:
        一个列表，包含模式串在文本串中所有出现位置的起始索引。
    """
    n = len(text)
    m = len(pattern)
    if m == 0:
        return [] # 如果模式串为空，则不匹配任何内容
    if m > n:
        return [] # 如果模式串比文本串长，则不可能匹配

    lps = compute_lps(pattern) # 计算模式串的 LPS 数组
    results = [] # 用于存储匹配结果的列表

    i = 0  # 文本串的指针
```

```
    j = 0  # 模式串的指针

    # 循环遍历文本串
    while i < n:
        if pattern[j] == text[i]:
            # 如果模式串的当前字符与文本串的当前字符匹配
            i += 1
            j += 1
        else:
            # 如果字符不匹配
            if j != 0:
                # 如果模式串指针 j 不为 0，说明之前有部分匹配
                # 利用 LPS 数组，将模式串指针 j 移动到 lps[j-1] 的位置
                # 这样可以避免不必要的比较
                j = lps[j - 1]
            else:
                # 如果模式串指针 j 为 0，说明模式串的第一个字符就不匹配
                # 直接移动文本串指针 i 到下一个字符
                i += 1

        if j == m:
            # 如果模式串指针 j 等于模式串的长度 m
            # 说明模式串完整匹配成功
            results.append(i - j) # 记录匹配的起始索引 (i - j)
            # 继续查找下一个匹配项
            # 将模式串指针 j 移动到 lps[j-1] 的位置，以便查找后续可能的重叠匹配
            j = lps[j - 1]

    return results
```

二叉树

一、顺序存储实现二叉树

对于一个以`0`为起始索引的列表（数组）`tree_list`：

- 如果一个节点的索引是`i`：
  - 其左子节点的索引是 `2 * i + 1`
  - 其右子节点的索引是 `2 * i + 2`
- 如果一个子节点的索引是`i`（且`i > 0`）：
  - 其父节点的索引是 `(i - 1) // 2`

这种表示方式最适合**完全二叉树**，因为不会有空间浪费。对于非完全二叉树，列表中间可能会有很多 `None` 值来表示不存在的节点，这可能导致空间效率不高。如下是一个实现二叉树的 Python 类：

```
import collections # 用于层序遍历的队列

class BinaryTree:
    def __init__(self, max_nodes=100):
        self.nodes = [None] * max_nodes # 用列表（数组）存储树节点
        self.size = 0                   # 当前树中的节点数量

    def insert(self, value):
        # 在下一个可用位置插入新值（按层序插入）
        if self.size < len(self.nodes):
            self.nodes[self.size] = value
            self.size += 1
        else:
            print("错误：树已满，无法插入。") # 或者抛出异常

    def parent_value(self, index):
        # 返回给定索引处节点的父节点的值
        if index > 0 and index < self.size: # 根节点没有父节点，索引需有效
            parent_idx = (index - 1) // 2
            return self.nodes[parent_idx]
        return None # 无父节点或索引无效

    def left_child_value(self, index):
        # 返回给定索引处节点的左子节点的值
        left_index = 2 * index + 1
        if left_index < self.size: # 检查左子节点索引是否在当前大小范围内
            return self.nodes[left_index]
        return None # 无左子节点或索引无效

    def right_child_value(self, index):
        # 返回给定索引处节点的右子节点的值
        right_index = 2 * index + 2
        if right_index < self.size: # 检查右子节点索引是否在当前大小范围内
```

```
            return self.nodes[right_index]
        return None # 无右子节点或索引无效
```

二、链式存储实现二叉树

在这个版本中，每个 `BinaryTree` 对象本身就是一个节点，它拥有一个值（`key`）以及对左子节点和右子节点的引用。

```
import collections # 用于层序遍历

class BinaryTree:
    def __init__(self, root_obj):
        self.key = root_obj # 节点的值
        self.left_child = None # 左子节点引用
        self.right_child = None # 右子节点引用

    def insert_left(self, new_node_val):
        # 插入左子节点
        # 如果没有左子节点，则直接插入
        # 如果已有左子节点，则将新节点插入到当前节点和原子节点之间
        if self.left_child is None:
            self.left_child = BinaryTree(new_node_val)
        else:
            t = BinaryTree(new_node_val)
            t.left_child = self.left_child # 新节点的左孩子是原子节点
            self.left_child = t # 当前节点的左孩子更新为新节点

    def insert_right(self, new_node_val):
        # 插入右子节点
        # 逻辑同 insert_left
        if self.right_child is None:
            self.right_child = BinaryTree(new_node_val)
        else:
            t = BinaryTree(new_node_val)
            t.right_child = self.right_child
            self.right_child = t

    def get_right_child(self):
        # 返回右子节点对象
        return self.right_child

    def get_left_child(self):
        # 返回左子节点对象
        return self.left_child

    def set_root_val(self, obj):
        # 设置当前节点的值
        self.key = obj

    def get_root_val(self):
        # 获取当前节点的值
```

```
        return self.key

    # --- 遍历方法 ---
    # 注意：这里的遍历方法是实例方法，通常从根节点对象开始调用

    def preorder_traversal(self):
        # 返回前序遍历的节点值列表 (根-左-右)
        result = [self.key] # 先访问根节点
        if self.left_child:
            result.extend(self.left_child.preorder_traversal()) # 递归遍历左子树
        if self.right_child:
            result.extend(self.right_child.preorder_traversal()) # 递归遍历右子树
        return result

    def inorder_traversal(self):
        # 返回中序遍历的节点值列表 (左-根-右)
        result = []
        if self.left_child:
            result.extend(self.left_child.inorder_traversal()) # 递归遍历左子树
        result.append(self.key) # 访问根节点
        if self.right_child:
            result.extend(self.right_child.inorder_traversal()) # 递归遍历右子树
        return result

    def postorder_traversal(self):
        # 返回后序遍历的节点值列表 (左-右-根)
        result = []
        if self.left_child:
            result.extend(self.left_child.postorder_traversal()) # 递归遍历左子树
        if self.right_child:
            result.extend(self.right_child.postorder_traversal()) # 递归遍历右子树
        result.append(self.key) # 最后访问根节点
        return result

    def levelorder_traversal(self):
        # 返回层序遍历的节点值列表 (广度优先)
        if self is None: # 实际上，如果 self 是 None，这个方法不会被调用
            return []

        result = []
        queue = collections.deque()
        queue.append(self) # 从当前节点（通常是根节点）开始

        while queue:
            current_node = queue.popleft()
            if current_node: # 确保节点存在
                result.append(current_node.get_root_val()) # 获取节点值
                if current_node.get_left_child():
                    queue.append(current_node.get_left_child())
                if current_node.get_right_child():
                    queue.append(current_node.get_right_child())
        return result
```

三、二叉堆

二叉堆 (Binary Heap) 是一种特殊的、基于树的数据结构，具体来说是**完全二叉树**，并且满足“堆属性”：

1. **堆属性**:
   - **最小堆 (Min-Heap)**: 每个父节点的值都小于或等于其子节点的值。因此，堆顶（根节点）是整个堆中的最小值。
   - **最大堆 (Max-Heap)**: 每个父节点的值都大于或等于其子节点的值。堆顶是最大值。
2. **主要用途**:
   - **优先队列 (Priority Queue)**: 高效地获取和移除优先级最高（或最低）的元素。
   - **图算法**: 如 Dijkstra（最短路径）、Prim（最小生成树）。
   - **快速找到第 K 小/大的元素**。

在 Python 中，二叉堆可以通过标准库中的`heapq`模块实现：

```
import heapq # 导入 heapq 模块

data = [3, 1, 4, 1, 5, 9]
heapq.heapify(data)  # 原地转换为堆，时间复杂度 O(n)
print(data)  # 输出为 [1, 1, 4, 3, 5, 9]（满足堆性质）

heapq.heappush(data, 2)  # 插入元素 2，自动调整堆
print(data)  # 堆结构更新为 [1, 1, 2, 3, 5, 9, 4]

min_val = heapq.heappop(data)  # 弹出最小元素 1，自动调整堆
print(min_val)  # 输出最小元素 1
print(data)     # 堆结构更新为 [1, 3, 2, 4, 5, 9]

print(data[0])  # 输出当前最小元素 1（不修改堆）

list1 = [1, 4, 7, 10]  # 已排序
list2 = [2, 3, 6, 9]   # 已排序
merged = heapq.merge(list1, list2)  # 合并多个已堆化的列表，返回一个迭代器
print(list(merged))  # 输出: [1, 2, 3, 4, 6, 7, 9, 10]

# 其他实用函数
min_val = heapq.heapreplace(data, 6)  # 弹出原最小，插入 6
min_val = heapq.heappushpop(data, 0)  # 插入 0，弹出新最小
print(heapq.nlargest(3, data))  # 输出最大的 3 个元素（无需堆化）
print(heapq.nsmallest(3, data))  # 输出最小的 3 个元素（无需堆化）
```

值得注意的是，如果需要使用自定义的“序关系”，可以通过 Python 类中的魔法方法实现。这种自定义的关系可以在 Python 的各种内置方法中直接使用，极大提高了代码书写的简便性。

```python
class Task:
    def __init__(self, name, priority):
        self.name = name
        self.priority = priority

    # 定义对象的 < 操作符逻辑（优先级越高，值越小）
    def __lt__(self, other):
        return self.priority < other.priority

    def __eq__(self, other):
        return self.priority == other.priority
```

四、Huffman 编码树

Huffman 树，又称最优二叉树，是一种带权路径长度最短的二叉树，常用于数据压缩中的 **Huffman 编码**。它的核心思想是：**频率高的字符用短编码，频率低的字符用长编码**，从而减少整体编码长度。同样的，利用 Python 内置库`heapq`简单高效地实现 Huffman 树：

```python
import heapq

class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = self.right = None

    def __lt__(self, other):
        return self.freq < other.freq  # 优先队列按频率排序

def build_tree(freq_dict):
    heap = [Node(char, f) for char, f in freq_dict.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(None, left.freq + right.freq)
        merged.left, merged.right = left, right
        heapq.heappush(heap, merged)
    return heap[0]  # 返回根节点
```

五、二叉搜索树

由于在 Python 中没有现成能实现二叉搜索树的类，因此相关算法的编写较为复杂。然而，理解二叉搜索树的内核之后我们也能较为清晰地实现二叉搜索树的各种功能。相关内容的核心想法如下：

插入：根据值的大小递归地将新节点插入左子树或右子树。

查找：递归遍历树，找到目标值返回`True`，否则返回`False`。

删除：

- **叶子节点**：直接删除。
- **单个子节点**：用子节点替换。
- **两个子节点**：用右子树的最小值替换当前节点，并递归删除右子树的最小节点。

```
class TreeNode:
    def __init__(self, val):
        self.val = val  # 节点存储的值
        self.left = None  # 左子节点
        self.right = None # 右子节点

class BST:
    def __init__(self):
        self.root = None # 初始化一个空树

    def insert(self, val):
        # 公共方法：插入一个值
        if self.root is None:
            self.root = TreeNode(val) # 如果树为空，新节点作为根节点
        else:
            self._insert_recursive(self.root, val) # 否则，递归插入

    def _insert_recursive(self, node, val):
        # 辅助方法：递归插入
        if val < node.val: # 如果值小于当前节点的值，插入到左子树
            if node.left is None:
                node.left = TreeNode(val) # 如果左子节点为空，直接插入
            else:
                self._insert_recursive(node.left, val) # 否则，在左子树中递归插入
        else: # 如果值大于或等于当前节点的值，插入到右子树 (允许重复值在右侧，或可调整)
            if node.right is None:
                node.right = TreeNode(val) # 如果右子节点为空，直接插入
            else:
                self._insert_recursive(node.right, val) # 否则，在右子树中递归插入

    def find(self, val):
        # 公共方法：查找一个值
        return self._find_recursive(self.root, val)

    def _find_recursive(self, node, val):
        # 辅助方法：递归查找
        if node is None:
            return False # 值未找到 (到达叶子节点或树为空)
        if val == node.val:
            return True # 值已找到
        elif val < node.val:
            return self._find_recursive(node.left, val) # 在左子树中查找
        else:
            return self._find_recursive(node.right, val) # 在右子树中查找

    def delete(self, val):
        # 公共方法：删除一个值
```

```
        self.root = self._delete_recursive(self.root, val)

    def _delete_recursive(self, node, val):
        # 辅助方法：递归删除
        if node is None:
            return node # 值未找到，或树为空

        # 导航到要删除的节点
        if val < node.val:
            node.left = self._delete_recursive(node.left, val)
        elif val > node.val:
            node.right = self._delete_recursive(node.right, val)
        else:
            # 找到了要删除的节点
            # 情况 1 & 2：节点有一个子节点或没有子节点
            if node.left is None:
                return node.right # 返回右子节点（可能为 None）
            elif node.right is None:
                return node.left # 返回左子节点（可能为 None）
            # 情况 3：节点有两个子节点
            else:
                # 找到中序后继节点（右子树中的最小节点）
                min_node = self._find_min(node.right)
                node.val = min_node.val # 将中序后继节点的值复制到当前节点
                # 删除中序后继节点
                node.right = self._delete_recursive(node.right, min_node.val)
        return node

    def _find_min(self, node):
        # 辅助方法：在子树中查找具有最小值的节点
        current = node
        while current.left is not None:
            current = current.left # 一直向左走，直到找到最左边的叶子节点
        return current

    def inorder_traversal(self):
        # 公共方法：中序遍历（左-根-右）
        result = []
        self._inorder_recursive(self.root, result)
        return result

    def _inorder_recursive(self, node, result):
        # 辅助方法：递归中序遍历
        if node: # 如果节点存在
            self._inorder_recursive(node.left, result) # 递归遍历左子树
            result.append(node.val) # 将当前节点的值添加到结果中
            self._inorder_recursive(node.right, result) # 递归遍历右子树
```

树和森林

一、树和森林的存储表示

在计算机中存储树和森林结构时，有多种方法可以选择。每种方法都有其特定的优势和劣势，适用于不同的操作需求和空间效率考量。

1. 父指针表示法

树或森林组织成一个节点顺序表，其中每一节点包含父结点的下标；根节点不具有父节点，`parent`值为 -1。这种存储方法容易找到父节点及祖先，比较节省空间，但较难查找到子女和兄弟。如：

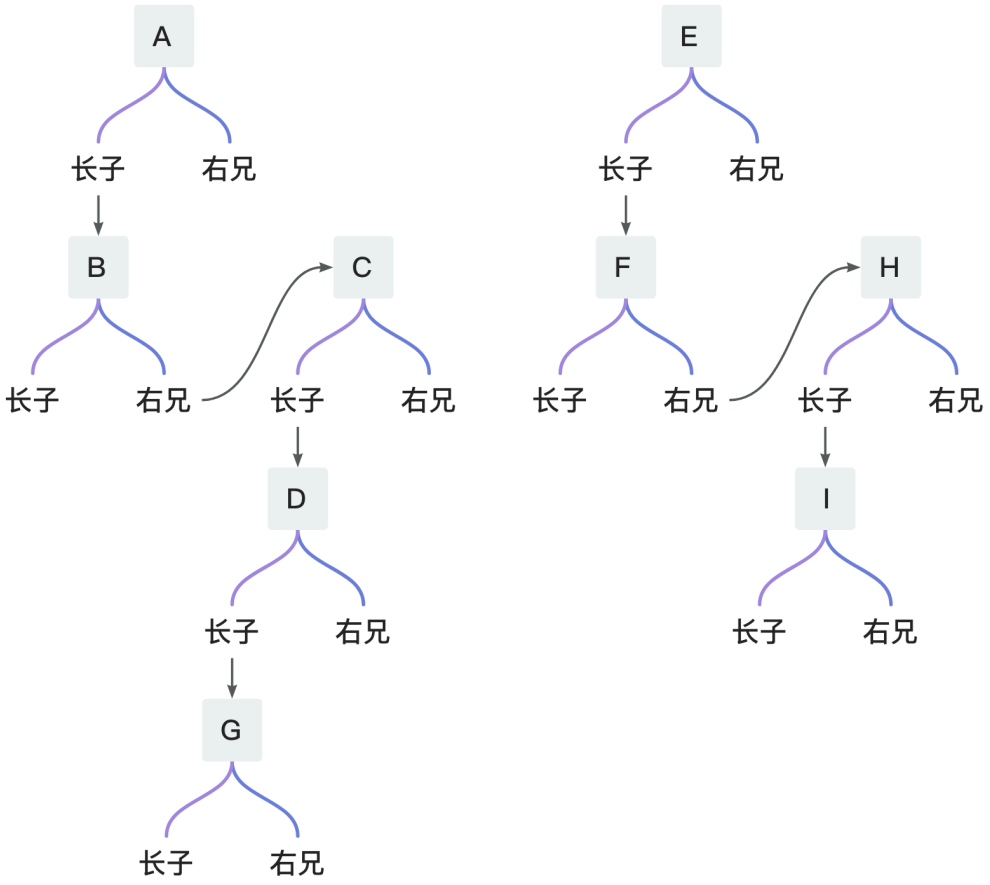
2. 子表表示法

子表表示法中整棵树或森林组织成一个节点顺序表，其中每一节点使用数组或链表记录该节点的所有子节点。这种方法容易找到子节点，但找到父节点更困难。此外，可以额外存储父节点指针作为解决的一种方法。

Node	0 (A)	1 (B)	2 (C)	3 (D)	4 (E)	5 (F)	6 (G)	7 (H)	8 (I)
Parent	-1	0	0	2	-1	4	5	4	7
Child	[1, 2]	[]	[3]	[]	[5, 7]	[6]	[]	[8]	[]

3. 长子-兄弟表示法

在长子-兄弟表示法中，每个节点存储节点值、长子指针、右侧兄弟结点指针。示意图如下：



我们可以通过 Python 中的自定义类来实现树和森林的数据结构：

```
class TreeNode:
    def __init__(self, info):
        self.info = info
        self.lchild = None
        self.rsibling = None
        self.parent = None  # 可选
```

二、并查集

Python 没有内置的并查集（Disjoint Set Union, DSU）数据结构。但是，你可以很容易地使用 Python 的内置数据类型（如字典或列表）来实现它。核心思想是维护一个父节点数组（或字典），其中每个元素指向其父节点。一个集合的代表（或根节点）是其父节点指向自身的元素。

下面是如何使用字典来实现一个基本的并查集，并包含路径压缩和按秩合并（或按大小合并）优化。

```
class DisjointSet:
    def __init__(self, size):
        # 初始化父节点数组，每个元素的父节点初始为自己
        self.parent = [i for i in range(size)]
        # 初始化秩数组（或称为“高度”/“大小”），用于合并优化，初始都为0
        self.rank = [0] * size

    def find(self, x):
        """
        查找元素x所在集合的代表元（根节点）。
        同时进行路径压缩优化。
        """
        # 如果x不是根节点（x的父节点不是它自己）
        if self.parent[x] != x:
            # 递归查找x的父节点的根节点
            # 并将x直接指向根节点（路径压缩）
            self.parent[x] = self.find(self.parent[x])
        # 返回x所在集合的根节点
        return self.parent[x]

    def union(self, x, y):
        """
        合并元素x和元素y所在的两个集合。
        使用按秩合并优化。
        """
        # 找到x和y各自所在集合的根节点
        root_x = self.find(x)
        root_y = self.find(y)

        # 如果x和y不在同一个集合中（它们的根节点不同）
        if root_x != root_y:
            # 按秩合并：将秩小的树合并到秩大的树上
            if self.rank[root_x] < self.rank[root_y]:
                # root_x的秩较小，将root_x的父节点设为root_y
```

```
                self.parent[root_x] = root_y
            elif self.rank[root_x] > self.rank[root_y]:
                # root_y的秩较小，将root_y的父节点设为root_x
                self.parent[root_y] = root_x
            else:
                # 如果两个集合的秩相同
                # 将root_x的父节点设为root_y (也可以反过来)
                self.parent[root_x] = root_y
                # 新的根节点root_y的秩增加1
                self.rank[root_y] += 1
            return True # 返回True表示成功合并
        return False # 如果已经在同一集合，返回False
```

字典与检索

在 Python 中，字典与检索可以简单地通过`dict`来实现。此外，`set`也采用了类似的方法来存储`key`。接下来是这两种数据结构的相关操作。

一、字典（`dict`）

- 字典的键必须为不可变对象。
- 删除元素
  要删除字典中的元素，可以使用`del`语句：
  `del my_dict['age']`将会删除 'age' 键及其对应的值。 还可以使用`pop()`方法来删除键并返回其值：
  `age = my_dict.pop('name')`会删除 'name' 并将其值赋给变量 age。
- 遍历字典
  可以使用`for`循环遍历字典：
  遍历所有键：`for key in my_dict`。
  遍历所有值：`for value in my_dict.values()`。
  遍历所有键值对：`for key, value in my_dict.items()`。
- 检查键是否存在
  可以使用`in`关键字检查某个键是否在字典中：
  `if 'name' in my_dict:`，如果 'name' 存在，就会执行相应的代码。
- 常用方法
  `my_dict.keys()`：返回字典中所有键的视图。
  `my_dict.values()`：返回字典中所有值的视图。
  `my_dict.items()`：返回字典中所有键值对的视图。
  `my_dict.update({'gender': 'female'})`：合并另一个字典，或更新已有键的值。
  `my_dict.clear()`：清空字典中的所有键值对。
- 字典嵌套
  字典可以包含其他字典，形成嵌套结构。例如：
- `nested_dict = {'person': {'name': 'Alice'}, 'location': 'NY'}`，可以通过`nested_dict['person']['name']`访问嵌套的值。
- 字典推导式
  字典推导式是一种快速创建字典的方式。例如：
  `{x: x**2 for x in range(5)}`会创建一个字典，其中键是数字，值是它们的平方，结果为`{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}`。
- 使用`defaultdict`
  如果需要处理缺省值，可以使用`collections.defaultdict`。例如：
  `dd = defaultdict(int)`创建一个默认值为0的字典。如果你访问一个不存在的键，它会自动创建并赋值为0。

二、集合`set`

1. 创建集合

- **使用大括号**：
  `my_set = {1, 2, 3}`
- **使用 `**set()**` 函数**：
  `my_set = set([1, 2, 3])` # 从列表创建集合

2. 集合的基本操作

- **添加元素**：
  使用 `add()` 方法向集合中添加单个元素。
  `my_set.add(4)` # 输出: {1, 2, 3, 4}
- **添加多个元素**：
  使用 `update()` 方法向集合中添加多个元素。
  `my_set.update([5, 6])` # 输出: {1, 2, 3, 4, 5, 6}
- **移除元素**：
  使用 `remove()` 方法移除指定元素。如果元素不存在，将引发 `KeyError`。
  `my_set.remove(3)` # 输出: {1, 2, 4, 5, 6}

使用 `discard()` 方法移除元素，但如果元素不存在不会引发错误。
`my_set.discard(10)` # 不会引发错误

- **弹出元素**：
  使用 `pop()` 方法随机移除并返回一个元素。
  `elem = my_set.pop()` # 随机弹出一个元素

3. 集合的其他方法

- **清空集合**：
  使用 `clear()` 方法清空集合。
  `my_set.clear()` # 输出: set()
- **查找元素**：
  使用 `in` 关键字检查元素是否在集合中。
  `is_in_set = 2 in my_set` # 输出: True 或 False
- **集合的长度**：
  使用 `len()` 函数获取集合中元素的数量。
  `length = len(my_set)` # 输出: 集合的元素个数

4. 集合的数学运算

- **并集**：
  使用 `union()` 方法或 `|` 操作符。
  `set1 = {1, 2, 3}`
  `set2 = {3, 4, 5}`
  `union_set = set1.union(set2)` # 输出: {1, 2, 3, 4, 5}
- **交集**：
  使用 `intersection()` 方法或 `&` 操作符。
  `intersection_set = set1.intersection(set2)` # 输出: {3}

- **差集**：
  使用 `difference()` 方法或 `-` 操作符。
  `difference_set = set1.difference(set2)` # 输出: {1, 2}
- **对称差集**：
  使用 `symmetric_difference()` 方法或 `^` 操作符。
  `sym_diff_set = set1.symmetric_difference(set2)` # 输出: {1, 2, 4, 5}

5. 集合的比较运算

- **判断元素是否属于集合**：
  使用 `in` 关键字。
  `is_member = 2 in my_set` # 输出: True 或 False
- **子集、真子集、超集、真超集**：
- **子集**：使用 `<=`。
  `is_subset = {1, 2} <= set1` # 输出: True
- **真子集**：使用 `<`。
  `is_proper_subset = {1, 2} < set1` # 输出: True
- **超集**：使用 `>=`。
  `is_superset = set1 >= {1, 2}` # 输出: True
- **真超集**：使用 `>`。
  `is_proper_superset = set1 > {1, 2}` # 输出: True

排序

在 Python 中，内置的排序算法既是稳定的又很高效，因此在需要排序时我们可以合理的使用它，而通过设置自定义类中的魔法方法和利用各类 Python 的特性能使算法的实现更加简洁明了。

一、Python 内置使用的排序算法

Python 通过两种主要方式提供了 Timsort 算法：

- **`list.sort()` 方法：**
  - 对列表进行**原地排序 (in-place)**，即直接修改原始列表。
  - 不返回值 (或者说返回 `None`)。
  - 示例：`my_list.sort()`
- **`sorted()` 内置函数：**
  - 可以接受任何**可迭代对象 (iterable)** 作为输入（如列表、元组、字符串、字典视图等）。
  - 返回一个新的、已排序的**列表**，原始的可迭代对象保持不变。
  - 示例：`new_list = sorted(my_iterable)`

两者都支持两个可选参数：

- `key`：指定一个接收一个参数的函数，该函数会作用于可迭代对象中的每个元素，排序将根据这些函数的返回值进行。例如，`key=str.lower` 或 `key=lambda x: x[1]`。
- `reverse`：一个布尔值。如果设置为 `True`，则列表元素将按降序排序。默认为 `False`（升序）。

对不同对象进行排序时，Python 有如下特性：数字排序时有自然的顺序，字符串或列表本身作为元素排序时依照字典序。

Python 中魔法方法之前已经提过，这里不再赘述。

```python
class Student:
    def __init__(self, name, grade):
        self.name = name
        self.grade = grade

    # 定义小于操作的行为
    def __lt__(self, other):
        # 如果成绩相同，则按姓名排序
        if self.grade == other.grade:
            return self.name < other.name
        # 否则按成绩排序
        return self.grade < other.grade
```

接下来我们将考虑各类排序算法的具体实现。

二、具体排序算法

1. 插入排序

将待排序的元素逐个插入到已经排好序的部分序列的适当位置，直到所有元素都插入完毕。就像我们打扑克牌时，一张一张地将牌插入到手中已有的有序牌中。

由于未改进的插入排序法效率不高，且只能使用插入相关算法实现的题目不多，在这里省去叙述。

2. 选择排序

每一趟从待排序的数据元素中选出最小（或最大）的一个元素，顺序放在已排好序的序列的最后（或最前），直到全部待排序的数据元素排完。

同样，直接选择排序效率较低，而堆排序算法在之前已经提到过，在这里也省去叙述。

3. 交换排序

1) 冒泡排序：

反复比较相邻的两个元素，如果顺序错误就交换，像气泡一样把大（或小）的元素“冒”到一端。非常简单；非常慢 ($O(N^2)$)，稳定。

2) 快速排序：

采用“分而治之”策略。选一个“基准”元素，将小于它的放左边，大于它的放右边，然后对左右两部分递归进行同样操作。平均情况下非常快 ($O(NlogN)$)，是实际应用中最常用的排序算法之一；最坏情况慢 ($O(N^2)$)，不稳定。

4. 基数排序

基数排序可以通过 Python 对列表排序的特性来实现，在此也不再叙述。

5. 归并排序

也是“分而治之”。先将序列递归地分成小段，直到每段只有一个元素（自然有序），然后将这些有序的小段两两“合并”(merge) 成一个更大的有序序列，直到整个序列合并完毕。以下是一个简洁明了的二路归并 Python 实现：

```
def merge_sort(arr):
    """
    归并排序函数
    :param arr: 待排序的列表
    :return: 排好序的新列表
    """
    # 基本情况：如果列表只有一个或没有元素，则它已经是有序的
    if len(arr) <= 1:
        return arr

    # 1. 分解 (Divide) 步骤
    mid = len(arr) // 2          # 找到中间点，分割列表
    left_half = arr[:mid]        # 左半部分
    right_half = arr[mid:]       # 右半部分

    # 递归地对左右两半进行排序
    left_sorted = merge_sort(left_half)
    right_sorted = merge_sort(right_half)

    # 2. 合并 (Merge) 步骤：将已排序的两半合并起来
```

```
    return merge(left_sorted, right_sorted)

def merge(left, right):
    """
    合并两个已排序的列表
    :param left: 已排序的左子列表
    :param right: 已排序的右子列表
    :return: 合并后的有序列表
    """
    merged_list = []
    left_pointer, right_pointer = 0, 0

    # 比较左右子列表的元素，按顺序放入新列表
    while left_pointer < len(left) and right_pointer < len(right):
        if left[left_pointer] <= right[right_pointer]: # 使用 <= 保证稳定性
            merged_list.append(left[left_pointer])
            left_pointer += 1
        else:
            merged_list.append(right[right_pointer])
            right_pointer += 1

    # 将其中一个子列表剩余的元素（如果有的话）追加到结果列表末尾
    # 因为两个子列表已经排序，所以剩余元素可以直接追加
    merged_list.extend(left[left_pointer:])
    merged_list.extend(right[right_pointer:])

    return merged_list
```

图及相关算法

一、图的存储方法

1. 邻接矩阵表示法

不必多说，给出一个不会引起浅拷贝错误的初始化代码：

```
n = 4
# 初始化一个 n x n 的矩阵，所有值为 0
edges = [[0] * n for _ in range(n)]
```

2. 邻接表表示法

不必多说，给出一个通过字典的简单实现：

```
graph_list = {} # 图本身

def add_edge(graph, u, v, directed=False):
    """向图中添加边 (邻接表)"""
    graph.setdefault(u, []) # 如果u不在图中，添加u
    graph.setdefault(v, []) # 如果v不在图中，添加v

    graph[u].append(v)
    if not directed:
        graph[v].append(u) # 无向图，双向添加
```

二、图的搜索与遍历

- 图的搜索：从图中给定的一个起始顶点出发，寻找一条到目标顶点的路径，称为图的搜索。
- 图的遍历：从图中某一顶点出发，按照某种方式系统地访问图中所有顶点，使得每一个顶点被访问且仅被访问一次，称为图的遍历，也称为图的周游。
- 搜索有两种基本策略：深度优先搜索（Depth First Search，DFS）以及广度优先搜索（Breadth First Search，BFS）
- 这两种策略同样可以用于图的遍历，即深度优先遍历与广度优先遍历。因此，图的搜索与遍历是联系紧密的两种操作。
- 对于各种操作型问题，可以转换为求解图中某节点到某节点的一条边。

1. 深度优先搜索

```
# 假设我们有之前定义的邻接表图和添加边的函数：
# graph_list = {}
# def add_edge(graph, u, v, directed=False): ...
# def get_neighbors(graph, u): ...

# DFS - 递归版本
```

```python
def dfs_recursive_util(graph, node, visited, traversal_order):
    visited.add(node)
    traversal_order.append(node)
    # print(f"访问: {node}") # 可以在这里处理顶点

    for neighbor in get_neighbors(graph, node):
        if neighbor not in visited:
            dfs_recursive_util(graph, neighbor, visited, traversal_order)

def dfs_recursive(graph, start_node):
    """深度优先搜索 (递归)"""
    if start_node not in graph:
        print(f"错误: 起始顶点 {start_node} 不在图中!")
        return []

    visited = set()
    traversal_order = []
    dfs_recursive_util(graph, start_node, visited, traversal_order)
    return traversal_order
```

2. 广度优先搜索

```python
from collections import deque # Python的deque可以高效地用作队列

# 假设我们有之前定义的邻接表图和添加边的函数:
# graph_list = {}
# def add_edge(graph, u, v, directed=False): ...
# def get_neighbors(graph, u): ...

def bfs(graph, start_node):
    """广度优先搜索"""
    if start_node not in graph:
        print(f"错误: 起始顶点 {start_node} 不在图中!")
        return []

    visited = set()     # 记录已访问过的顶点
    queue = deque()     # 创建一个双端队列用作FIFO队列

    visited.add(start_node)
    queue.append(start_node)

    traversal_order = [] # 记录遍历顺序

    while queue: # 当队列不为空
        current_node = queue.popleft() # 从队列头部取出顶点
        traversal_order.append(current_node)
        # print(f"访问: {current_node}") # 可以在这里处理顶点

        for neighbor in get_neighbors(graph, current_node):
            if neighbor not in visited:
                visited.add(neighbor)
```

```
                queue.append(neighbor)

    return traversal_order
```

三、最小生成树

现在，给图中的每条边都赋予一个权重 (或成本) (比如道路的长度、修建成本、通行时间等)。这样的图称为加权无向图。最小生成树 (MST) 就是这个加权无向图的一个生成树，并且它所有边的权重之和是所有可能生成树中最小的。

常见有两种经典的贪心算法来寻找最小生成树，Prim 和 Kruskal。

1. Prim 算法

```
import heapq # 重新引入 heapq 用于优先队列

def cmp_node(n1, n2):
    """辅助函数，规范化边中节点的顺序。"""
    return (n1, n2) if n1 < n2 else (n2, n1)

def get_mst(g, start_n):
    """
    参数:
    g (dict): 图的邻接表 {节点: {邻居: 权重, ...}}
    start_n (any): 起始节点

    返回:
    tuple: (MST边列表, MST总权重)
    """
    if not g:
        return [], 0
    if start_n not in g:
        raise ValueError(f"起始节点 '{start_n}' 不在图中。")

    mst_edg = []           # 存储 MST 的边
    tot_wgt = 0            # MST 的总权重
    visited = set()        # 已加入 MST 的节点集合

    # 优先队列，存储 (权重, 当前节点, 从哪个节点连接)
    # 对于起始点，第三个元素 (从哪个节点连接) 可以是它自身或None
    pq = [(0, start_n, start_n)] # (权重, 要访问的节点, 连接源节点)

    num_node = len(g)

    while pq and len(visited) < num_node:
        wgt, curr_n, prev_n = heapq.heappop(pq)

        if curr_n in visited:
            continue # 如果节点已在MST中，跳过

        visited.add(curr_n)
```

```
        # 将边加入MST (除了起始节点的第一个伪边)
        if curr_n != prev_n : # 或者检查 prev_n 是否是初始的标记值
            mst_edg.append(cmp_node(prev_n, curr_n))
            tot_wgt += wgt

        # 如果MST已包含所有节点，可以提前结束
        if len(mst_edg) == num_node -1 and num_node > 1 : # 对于多于一个节点的图
             break
        if num_node == 1 and not mst_edg: # 单节点图没有边
             break


        # 将新加入节点的所有未访问邻居的边加入优先队列
        for nbr, edge_wgt in g.get(curr_n, {}).items():
            if nbr not in visited:
                heapq.heappush(pq, (edge_wgt, nbr, curr_n))

    if len(visited) < num_node and num_node > 0:
        print(f"警告: 图可能不连通。MST 只包含 {len(visited)} 个节点，总共 {num_node} 个。")

    return mst_edg, tot_wgt
```

2. Kruskal 算法

```
# --- 并查集 (DSU) ---
par = {} # 父节点字典 (par for parent)

def mk_set(v_node):
    """创建新集合，v_node是其代表。"""
    par[v_node] = v_node

def find(v_node):
    """查找v_node所在集合的代表（带路径压缩）。"""
    if par.get(v_node) is None: # 如果节点未在并查集中初始化
        mk_set(v_node) # 则先初始化

    if par[v_node] == v_node:
        return v_node
    par[v_node] = find(par[v_node]) # 路径压缩
    return par[v_node]

def unite(v1, v2):
    """合并v1和v2所在的集合。"""
    r1 = find(v1) # r for root
    r2 = find(v2)
    if r1 != r2:
        par[r2] = r1 # 将一个集合的根指向另一个
        return True # 合并成功
    return False # 已在同一集合
```

```python
# --- Kruskal 算法 (无规范化) ---
def kruskal(g):
    """
    参数:
    g (dict): 图的邻接表 {节点: {邻居: 权重, ...}}
              假设为无向图。

    返回:
    tuple: (MST边列表, MST总权重)
           每条边格式为 (节点1, 节点2), 顺序取决于遍历。
    """
    if not g:
        return [], 0

    edges = []   # 存储所有边 (权重, 节点1, 节点2)
    nodes = set() # 存储所有节点

    # 1. 提取所有边和节点
    #    直接从邻接表添加, 如果g['A']['B']和g['B']['A']都存在, 则两者都会被加入
    for u, nbrs in g.items():
        nodes.add(u)
        for v, wgt in nbrs.items():
            nodes.add(v) # 确保所有节点都被记录
            edges.append((wgt, u, v))

    # 2. 按权重对所有边进行升序排序
    edges.sort()

    # 3. 初始化并查集
    global par # 引用全局的 par 字典
    par = {}   # 重置 par 字典
    for node_item in nodes: # 使用不同的变量名避免与外层node冲突
        mk_set(node_item)

    mst_edg = []      # 存储 MST 的边
    tot_wgt = 0       # MST 的总权重
    num_node_val = len(nodes) # 使用不同的变量名

    # 4. 遍历排序后的边
    for wgt, u, v in edges:
        # 5. 如果边的两个顶点不在同一个集合中 (即不成环)
        if find(u) != find(v):
            unite(u, v)        # 合并这两个顶点所在的集合
            mst_edg.append((u,v)) # 直接添加 (u,v)
            tot_wgt += wgt

            # 6. 当 MST 包含 V-1 条边时, 算法结束
            # (对于有效节点数大于0的情况)
            if num_node_val > 0 and len(mst_edg) == num_node_val - 1:
                break

    if num_node_val > 0 and len(mst_edg) < num_node_val - 1:
```

```
        print(f"警告：图可能不连通。MST 只包含 {len(mst_edg)} 条边，应有 {num_node_val-1} 条（若连通）。")

    return mst_edg, tot_wgt
```

四、最短路径算法

1. Dijkstra 算法

目标：

在一个带权重的图中，找到从一个指定的起始顶点（源点）到图中所有其他顶点的最短路径。这个算法计算的是路径的“长度”，即路径上各条边权重之和。注意，Dijkstra 算法不能应用于有负权重的图。

核心思想 - 贪心策略：

1. 算法维护一个集合 `S`，其中包含所有已确定最短路径的顶点。初始时，`S` 只包含源点。
2. 对于不在 `S` 中的每个顶点 `v`，算法维护一个当前已知的从源点到 `v` 的最短路径估计值 `dist[v]`。
3. 在每一步，算法从那些不在 `S` 中的顶点中，选择一个 `dist` 值最小的顶点 `u`。
4. 将 `u` 加入集合 `S` (表示从源点到 `u` 的最短路径已最终确定)。
5. 然后，对于 `u` 的每一个邻居 `v` (无论 `v` 是否在 `S` 中)，算法尝试通过 `u` 来“松弛” (relax) 到 `v` 的路径：如果从源点经由 `u` 到达 `v` 的路径比当前已知的 `dist[v]` 更短，则更新 `dist[v]`。这个过程不断重复，直到所有顶点都加入集合 `S`，或者所有可达顶点的最短路径都已找到。

下面我们给出一个具体的 Python 实现：

```
import heapq

def dijkstra(graph, start):
    """
    实现 Dijkstra 算法，查找从起始节点到图中所有其他节点的最短路径。

    Args:
        graph: 一个表示图的字典，键是节点，
               值是包含邻居和边权重的字典。
               示例：{'A': {'B': 1, 'C': 4}, 'B': {'A': 1, 'C': 2, 'D': 5}}
        start: 起始节点。

    Returns:
        一个包含两个字典的元组：
        - dists: 从起始节点到所有其他节点的最短距离。
        - prev: 从起始节点出发的最短路径中的前一个节点。
    """
    dists = {node: float('inf') for node in graph}
    dists[start] = 0
    prev = {node: None for node in graph}
    pq = [(0, start)]  # 优先队列：(距离, 节点)

    while pq:
        curr_d, curr_n = heapq.heappop(pq)
```

```
        # 如果已经找到更短的路径，则跳过
        if curr_d > dists[curr_n]:
            continue

        for neibr, weight in graph[curr_n].items():
            dist = curr_d + weight
            if dist < dists[neibr]:
                dists[neibr] = dist
                prev[neibr] = curr_n
                heapq.heappush(pq, (dist, neibr))

    return dists, prev
```

2. Floyd算法

Floyd-Warshall 算法是一种用于**在加权有向图中查找所有节点对之间的最短路径**的动态规划算法。这意味着，对于图中的任意两个节点 `i` 和 `j`，该算法都能计算出从 `i` 到 `j` 的最短距离。

核心思想：

Floyd-Warshall 算法的核心思想是**逐步允许使用更多的中间节点**来优化路径。

它基于以下观察： 从节点 `i` 到节点 `j` 的最短路径，要么**不经过**某个中间节点 `k`，要么**经过**中间节点 `k`。如果最短路径不经过节点 `k`，那么它就是之前（即在允许经过节点 `k` 之前）找到的 `i` 到 `j` 的最短路径。如果最短路径经过节点 `k`，那么这条路径可以分解为从 `i` 到 `k` 的最短路径，加上从 `k` 到 `j` 的最短路径。

算法会迭代地考虑图中的每一个节点作为潜在的中间节点。

```
INF = float('inf') # 代表无穷大，表示两点之间不可达

def floyd(graph):
    """
    实现 Floyd-Warshall 算法，计算所有节点对之间的最短路径。

    Args:
        graph: 一个邻接矩阵 (列表的列表) 表示图。
               graph[i][j] 是从节点 i 到节点 j 的边的权重。
               如果节点 i 和 j之间没有直接边，则为 INF。
               对角线元素 graph[i][i] 应为 0。

    Returns:
        一个包含所有节点对之间最短路径的邻接矩阵 (dist)。
    """
    nodes_n = len(graph)
    dist = [row[:] for row in graph] # 复制图以存储距离

    # k 是中间节点
    for k_node in range(nodes_n):
        # i 是起始节点
        for i_node in range(nodes_n):
            # j 是结束节点
```

```
            for j_node in range(nodes_n):
                if dist[i_node][k_node] != INF and \
                   dist[k_node][j_node] != INF and \
                   dist[i_node][k_node] + dist[k_node][j_node] < dist[i_node][j_node]:
                    dist[i_node][j_node] = dist[i_node][k_node] + dist[k_node][j_node]
    return dist
```

五、拓扑排序和关键路径

1. 拓扑排序

拓扑排序用于有向无环图 (DAG)，它将所有节点排成一个线性序列，使得对于任何从节点 `u` 到节点 `v` 的有向边，节点 `u` 在序列中都出现在节点 `v` 之前。

```
from collections import deque

def topo_srt(nodes, graph):
    """
    实现拓扑排序算法 (Kahn's algorithm)。

    Args:
        nodes: 节点数量 (通常是从 0 到 nodes-1)。
        graph: 一个邻接表表示的有向图。
               例如: {0: [1, 2], 1: [3], 2: [3], 3: []}
               表示从节点0可以到1和2，从1可以到3，从2可以到3。

    Returns:
        一个包含拓扑排序结果的列表。
        如果图中存在环，则返回一个空列表或错误信息。
    """
    in_deg = [0] * nodes  # 存储每个节点的入度
    adj = [[] for _ in range(nodes)] # 邻接表

    # 构建邻接表并计算每个节点的入度
    # graph 的键是源节点，值是目标节点列表
    for u_node in graph:
        for v_node in graph[u_node]:
            adj[u_node].append(v_node)
            in_deg[v_node] += 1

    queue = deque()
    # 将所有入度为0的节点加入队列
    for i in range(nodes):
        if in_deg[i] == 0:
            queue.append(i)

    topo_ord = [] # 存储拓扑排序结果
    count_v = 0   # 计数已访问的节点

    while queue:
        u_node = queue.popleft()
```

```
            topo_ord.append(u_node)
            count_v += 1

            # 对于 u_node 的每个邻居 v_node，减少其入度
            # 如果 v_node 的入度变为0，则将其加入队列
            for v_node in adj[u_node]:
                in_deg[v_node] -= 1
                if in_deg[v_node] == 0:
                    queue.append(v_node)

    if count_v != nodes:
        print("图中存在环，无法进行拓扑排序。")
        return []  # 或者可以抛出异常
    else:
        return topo_ord
```

此外，我们也可利用后序遍历实现拓扑排序：

```
def topo_srt_dfs(nodes, graph):
    """
    使用深度优先搜索 (DFS) 实现拓扑排序。
    结果是后序遍历的逆序。

    Args:
        nodes: 节点数量 (通常是从 0 到 nodes-1)。
        graph: 一个邻接表表示的有向图。
               例如: {0: [1, 2], 1: [3], 2: [3], 3: []}

    Returns:
        一个包含拓扑排序结果的列表。
        如果图中存在环，则返回一个空列表。
    """
    adj = [[] for _ in range(nodes)] # 邻接表
    for u_node in graph:
        for v_node in graph[u_node]:
            adj[u_node].append(v_node)

    visited = [False] * nodes  # 标记节点是否已被访问
    on_path = [False] * nodes  # 标记节点是否在当前DFS递归路径上 (用于检测环)
    post_ord = []              # 存储后序遍历的结果 (反向即为拓扑排序)
    has_cyc = False            # 标记是否检测到环

    def dfs_util(u_node):
        nonlocal has_cyc # 允许修改外部函数的 has_cyc 变量
        visited[u_node] = True
        on_path[u_node] = True

        for v_node in adj[u_node]:
            if has_cyc: # 如果已经检测到环，提前返回
                return
            if on_path[v_node]: # 如果邻居在当前路径上，则有环
```

```
                has_cyc = True
                return
            if not visited[v_node]:
                dfs_util(v_node)
                if has_cyc: # 如果递归调用检测到环，提前返回
                    return

        on_path[u_node] = False # 将节点移出当前路径
        post_ord.append(u_node) # 后序遍历：在所有子节点处理完毕后添加

    for i_node in range(nodes):
        if has_cyc: # 如果已经检测到环，则无需继续
            break
        if not visited[i_node]:
            dfs_util(i_node)

    if has_cyc:
        print("图中存在环，无法进行拓扑排序。")
        return []
    else:
        # 后序遍历的结果是 [..., child, parent]
        # 拓扑排序需要 [parent, child, ...]，所以需要反转
        return post_ord[::-1]
```

2. 关键路径

关键路径算法 (Critical Path Method, CPM) 是一个项目管理中用来确定项目总工期以及识别哪些任务（活动）是“关键”的（即它们的延迟会直接导致整个项目延迟）的算法。实现关键路径算法通常涉及以下步骤：

1. **表示活动和依赖关系**：将项目表示为一个有向无环图 (DAG)，其中节点代表活动，边代表依赖关系。每个活动都有一个持续时间。
2. **拓扑排序**：对活动进行拓扑排序。
3. 计算最早开始时间 (ES) 和最早完成时间 (EF)：
    - 对于每个活动，其 ES 是其所有前置活动的最晚 EF。
    - EF = ES + 活动持续时间。
    - 按拓扑顺序计算。
4. 计算最晚开始时间 (LS) 和最晚完成时间 (LF)：
    - 项目总工期是所有活动中最大的 EF。
    - 对于每个活动，其 LF 是其所有后继活动的最小 LS。
    - LF = LS + 活动持续时间 (或者 LS = LF - 活动持续时间)。
    - 按逆拓扑顺序计算。
5. 计算浮动时间 (Slack 或 Float)：
    - Slack = LF - EF (或 LS - ES)。
6. **识别关键路径**：浮动时间为零的活动构成了关键路径。

其他：Python 补充

一、`math`模块

1. `math`模块简介

`math`模块是Python的标准库之一，提供了基础数学运算的函数和常量，适用于科学计算和数学应用。

2. 常用常量

- `**math.pi**`：圆周率π，值约为3.14159。
- `**math.e**`：自然对数的底数e，值约为2.71828。

3. 基本数学运算函数

1) 取整与舍入

- `**math.floor(x)**`：返回不大于x的最大整数（向下取整）。
- `**math.ceil(x)**`：返回不小于x的最小整数（向上取整）。
- `**math.trunc(x)**`：返回x的整数部分（去掉小数部分）。

2) 幂运算

- `**math.pow(x, y)**`：返回x的y次方，结果为浮点数。
- `**math.sqrt(x)**`：返回x的平方根。
- `**math.exp(x)**`：返回e的x次方。

3) 对数运算

- `**math.log(x[, base])**`：返回x的自然对数。如果提供`base`，则返回以`base`为底的对数。
- `**math.log10(x)**`：返回x的以10为底的对数。
- `**math.log2(x)**`：返回x的以2为底的对数。

4) 其他数学函数

- `**math.factorial(x)**`：返回x的阶乘（x必须为非负整数）。
- `**math.gcd(a, b)**`：返回a和b的最大公约数。

二、常见 Python 异常及解决办法

1. `SyntaxError` (语法错误)

- **意思**：代码不符合 Python 的语法规范。解释器在运行代码之前进行语法检查时发现错误。
- **常见原因**：
  - 拼写错误（例如 `whlie` 而不是 `while`）。
  - 缺少冒号（例如 `if x > 0` 后面忘记加 `:`）。

    - 括号、引号不匹配。
    - 缩进错误（虽然通常表现为 `IndentationError`，但有时也会导致语法混乱）。
    - 使用了 Python 的关键字作为变量名。
- **解决方法**：
    - 仔细检查报错信息中指示的行及其附近的代码。
    - 注意 Python 提示的 `^` 符号，它通常指向语法错误发生的位置。
    - 使用代码编辑器的语法高亮和检查功能。

2. `NameError` (名称错误)

- **意思**：尝试访问一个未被定义的变量或函数名。
- **常见原因**：
    - 变量或函数名拼写错误。
    - 变量在使用前未被赋值。
    - 函数在使用前未被定义。
    - 作用域问题：试图在定义该变量/函数的作用域之外访问它。
- **解决方法**：
    - 检查变量或函数名的拼写。
    - 确保在使用变量前已经给它赋了初值。
    - 确保函数在使用前已经被 `def` 定义。
    - 检查变量或函数的作用域。

3. `TypeError` (类型错误)

- **意思**：对一个不支持该操作的数据类型执行了某种操作。
- **常见原因**：
    - 不同类型的数据进行运算 (例如，字符串和数字相加 `'hello' + 5`)。
    - 调用函数时传递了错误类型的参数。
    - 对非可调用对象使用 `()` (例如，`my_list()` 如果 `my_list` 是一个列表)。
    - 对不支持索引或切片的对象使用索引或切片 (例如，对一个数字 `123[0]`)。
- **解决方法**：
    - 使用 `type()` 函数检查涉及变量的数据类型。
    - 确保操作符两边的数据类型是兼容的，或者进行必要的类型转换 (例如 `str(5)` 将数字转为字符串)。
    - 检查函数调用时传递的参数类型是否符合函数定义。

4. `ValueError` (值错误)

- **意思**：传递给函数的参数类型正确，但值不合适或不在预期范围内。
- **常见原因**：

    - `int()` 函数试图转换一个非数字字符串 (例如 `int('abc')`)。
    - `math.sqrt()` 接收一个负数。
    - 序列解包时，值的数量与变量数量不匹配 (例如 `a, b = [1, 2, 3]`)。
- **解决方法**:
    - 检查传递给函数的值是否在函数要求的有效范围内。
    - 在进行类型转换前，验证输入数据的格式是否正确。
    - 使用 `try-except ValueError` 来捕获并处理可能的值错误。

5. `IndexError` (索引错误)

- **意思**：试图访问序列（如列表、元组、字符串）中不存在的索引。
- **常见原因**:
    - 索引超出了序列的范围 (例如，列表 `L = [1, 2, 3]`，访问 `L[3]` 或 `L[-4]`)。
    - 对空序列进行索引。
- **解决方法**:
    - 确保索引值在 `0` 到 `len(sequence) - 1` (对于正索引) 或 `-len(sequence)` 到 `-1` (对于负索引) 之间。
    - 在访问索引前，检查序列是否为空或长度是否足够。

6. `KeyError` (键错误)

- **意思**：试图访问字典中不存在的键。
- **常见原因**:
    - 字典中没有这个键名。
    - 键名拼写错误。
- **解决方法**:
    - 在访问字典的键之前，使用 `in` 操作符检查键是否存在 (例如 `if key in my_dict:`)。
    - 使用字典的 `get()` 方法，它允许你为不存在的键提供一个默认值 (例如 `my_dict.get(key, "default_value")`)。