

尚硅谷大模型技术之数据结构与算法

(作者: 尚硅谷研究院)

版本: V0.9.0

第 1 章 刷题地址

<https://leetcode.cn/studyplan/top-interview-150/>

第 2 章 数据结构与算法基础

2.1 数据结构基础

2.1.1 什么是数据结构

数据结构是为了高效访问数据而设计出的一种数据的组织和存储方式。更具体的说，一个数据结构包含一个数据元素的集合、数据元素之间的关系以及访问和操作数据的方法。

像前面我们接触到的 list、set、dict、tuple 其实已经是一种 python 封装的高级数据结构了，里面封装了对基本数据类型数据的存储以及组织方式。

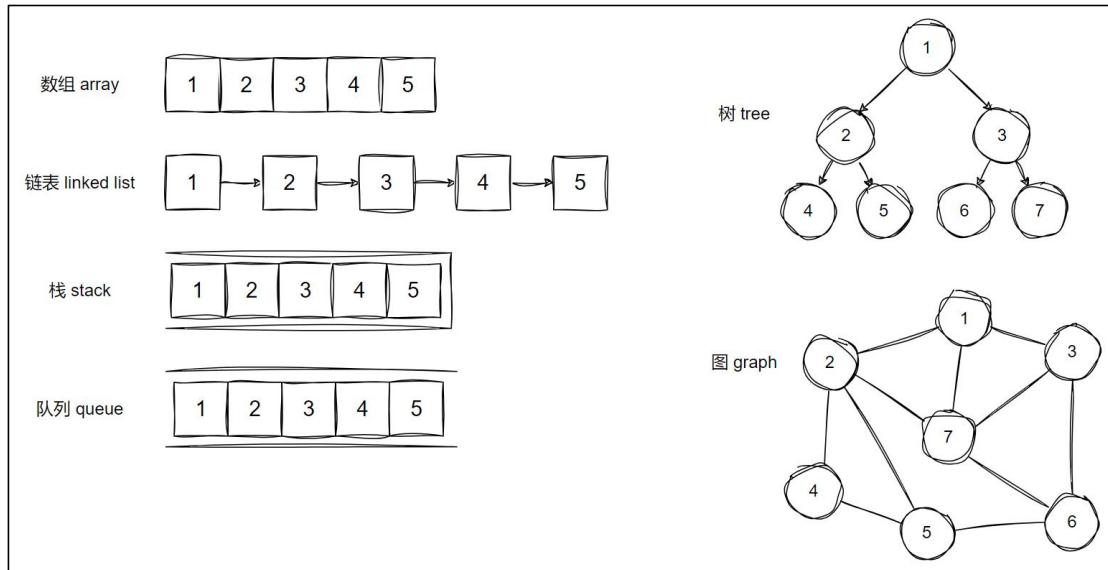
2.1.2 数据结构的分类

1) 逻辑结构

数据的逻辑结构反映了数据元素之间的逻辑关系。逻辑结构可分为线性和非线性两大类。

线性数据结构中的元素之间是一对一的顺序关系，在一对一的顺序关系中，除了第一个元素没有前驱元素，最后一个元素没有后继元素外，其余每个元素都有且仅有一个直接前驱元素和一个直接后继元素。这种关系使得数据元素可以按照一个线性的顺序依次排列，就像排成一列的队伍，每个成员前后都有明确的相邻成员（队首和队尾除外）如数组、链表、栈、队列等。

非线性数据结构中的元素之间具有多个对应关系，如树、图等。

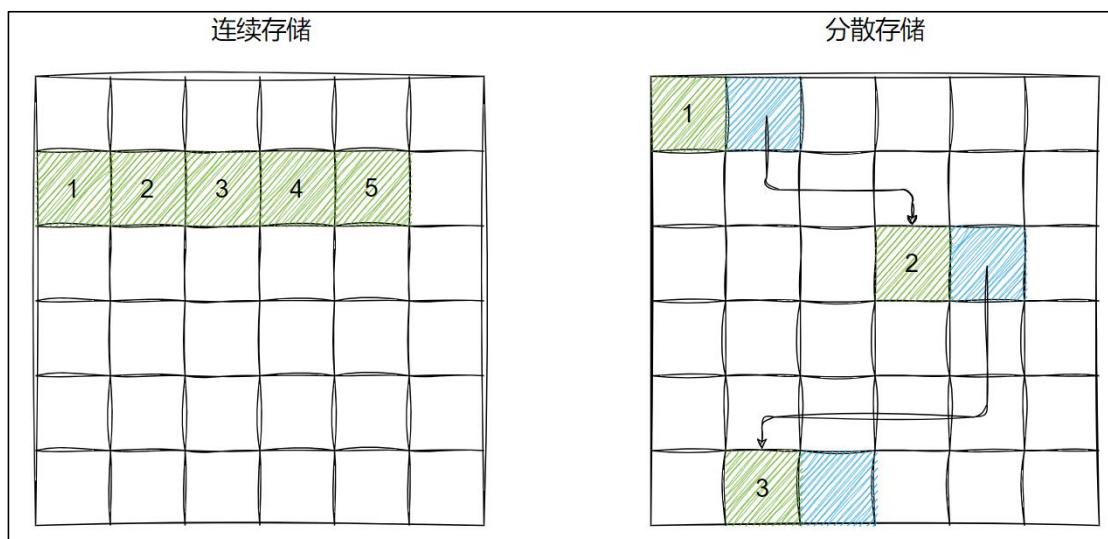


2) 物理结构

数据的物理结构反映了数据在计算机内存中的存储结构。一般而言，数据结构针对的是内存中的数据。内存由许多存储单元组成，每个存储单元可以存储一个固定大小的数据块，通常以字节（Byte）为单位。每个存储单元都有一个唯一的地址，操作系统正是根据这一地址去访问内存中的数据的。我们讨论的数据结构中的数据元素就保存在这一个个的内存单元中。数据在内存中的存储结构可分为连续存储（数组）与分散存储（链表）。

连续存储借助数据之间的相对位置来表示数据元素之间的逻辑关系。

分散存储借助指示数据位置的指针来表示数据元素之间的逻辑关系。



所有数据结构都是基于数组、链表或二者的组合实现的。例如，栈和队列既可以使用数组实现，也可以使用链表实现；而哈希表的实现可能同时包含数组和链表。

2.2 算法基础

2.2.1 什么是算法

算法是一个用于解决特定问题的有限指令序列（计算机可以执行的操作）。通俗的理解就是可以解决特定问题的方法。

算法的五大特性：

- 输入：算法具有 0 个或多个输入。
- 输出：算法至少有 1 个输出。
- 有穷性：算法在有限的步骤之后会自动结束而不会无限循环，并且每一个步骤可以在可接受的时间内完成。
- 确定性：算法中的每一步都有确定的含义，不会出现二义性。
- 可行性：算法的每一步都是清楚且可行的，能让用户用纸笔计算而求出答案。

2.2.2 算法的分类

按照应用目的分类：搜索算法（深度优先搜索、广度优先搜索等）、排序算法（冒泡排序、插入排序、选择排序、快速排序、归并排序等）、最优化算法等。

按照实现策略分类：暴力法、增量法、分治算法、动态规划算法、贪心算法等。

2.2.3 时间复杂度

1) 时间复杂度定义

时间复杂度用来描述运行一个算法所需时间资源的多少。为了屏蔽计算机硬件差异对评估结果的影响，我们不使用算法运行的绝对运行时间，而是使用运行算法时执行的基本指令数量来衡量其所需的时间资源。

常见的计算机基本指令有：

- 赋值指令：用于为变量赋值。
- 算术运算指令：包括加法、减法、乘法和除法等基本的数值运算。
- 逻辑运算指令：包括与、或、非等逻辑操作。
- 比较指令：用于比较两个值的大小或相等性。

我们假定计算机执行算法每一个基本操作的时间是固定的一个时间单位，对于不同的机器环境而言，确切的单位时间是不同的，但是对于算法进行多少个基本操作（即花费多少时间单位）在规模数量级上却是相同的。那么一个算法的所需时间 T 就可以表达为一个与输入规模 n 相关的函数： $T(n)$ 。

例如下面这个求和的算法，其所需时间 $T(n) = 5n+5$ 。其中 n 表示算法的输入规模，这里代表数组长度。

```
def sum(nums):
    sum_num = 0 # 1 次赋值操作
    i = -1 # 1 次赋值操作
    while (i := i + 1) < len(nums): # n+1 次加法运算 + n+1 次赋值操作 + n+1
        sum_num += nums[i] # n 次加法运算 + n 次赋值操作
    return sum_num
```

而下面这个寻找最大值的算法，其所需时间由于条件判断而变得不确定，若输入数组中的最大值位于首位，那么 $T(n) = 4n+1$ ；如果最大值位于末位，那么 $T(n) = 5n$ 。

```
def find_max(nums):
    max_num = nums[0] # 1 次赋值操作
    i = 0 # 1 次赋值操作
    while (i := i + 1) < len(nums): # n 次加法运算 + n 次赋值操作 + n 次比较
        if nums[i] > max_num: # n-1 次比较运算
            max_num = nums[i] # 0~(n-1)次赋值操作
    return max_num
```

我们注意到上述算法的运行时间还取决于具体数据，而不仅仅是问题规模。对于这种算法，我们把它们的执行情况分为：

- 最优时间复杂度，即算法完成工作最少需要多少基本操作。
- 最坏时间复杂度，即算法完成工作最多需要多少基本操作。
- 平均时间复杂度，即算法完成工作平均需要多少基本操作。

对于最优时间复杂度，其价值不大，因为它没有提供什么有用信息，其反映的只是最乐观最理想的情况，没有参考价值；对于最坏时间复杂度，提供了一种保证，表明算法在此种程度的基本操作中一定能完成工作；对于平均时间复杂度，是对算法的一个全面评价，因此它完整全面的反映了这个算法的性质。但另一方面，这种衡量并没有保证，不是每个计算都能在这个基本操作内完成。此外，“平均”还依赖于所选的实例，以及这些实例出现的概率。因此，我们主要关注算法的最坏情况，亦即最坏时间复杂度。因此，上述寻找最大值的算法所需时间为 $T(n) = 5n$ 。

一般情况下，我们并不关心输入规模较小时算法的用时多少，并且其实上述的 $T(n)$ 函数也并不是绝对的精确。所以我们通常只关注输入规模无限增大时，算法用时的增长趋势。也就是比较 $T(n)$ 函数在 n 趋近于无穷大时的增长速度。

到目前为止，我们就已经能够对时间复杂度下一个准确的定义了：

- 时间复杂度统计的是算法运行时执行的基本指令数，而非绝对运行时间。
- 时间复杂度体现的是算法基本指令数随输入规模 n 增大时的变化趋势。

2) 时间复杂度的大 O 表示法

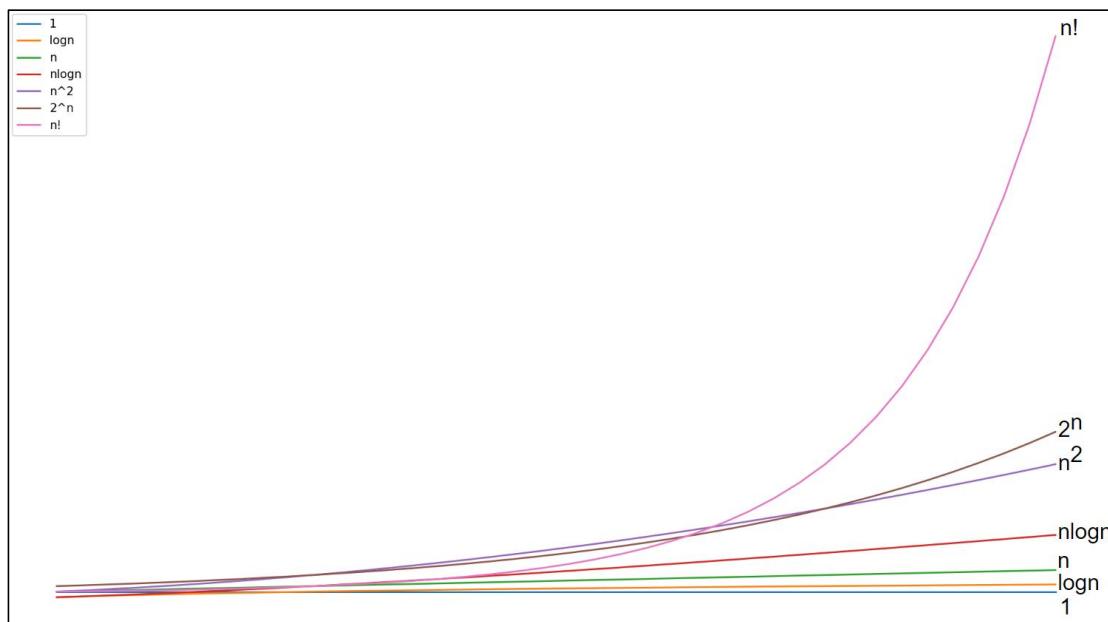
由于时间复杂度只需要关注输入规模增大时，算法用时的增长趋势。而输入规模无限增大时， $T(n)$ 函数中的低阶项和常数项，以及高阶项的常数系数，对于增长趋势的影响就会变得微乎其微，因此我们可以将其忽略掉，然后用最高阶项来表示 $T(n)$ 函数的增长趋势。

例如我们现在有两个算法，分别是算法 A： $T(n) = 5n+5$ ，算法 B： $T(n) = 2n^2+2n-2$ ，按照上述思路，算法 A 的时间复杂度可以表示为 $O(n)$ ，算法 B 的时间复杂度可以表示为 $O(n^2)$ 。

3) 常见的时间复杂度

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!) < O(n^n)$$

注意：表示时间复杂度时经常将 $\log_2 n$ 简写为 $\log n$ 。



下面给出几个常见的时间复杂度案例：

(1) $O(1)$

两数求和：

```
def add(a, b):
    return a + b
```

如果算法的执行时间不随着问题规模 n 的增加而增长，即使算法中有上千条语句，其执行时间也不过是一个较大的常数。此类算法的时间复杂度是 $O(1)$ 。

(2) $O(\log n)$

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

二分查找：

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1
```

对数阶反映了“每轮缩减到一半”的情况。

(3) O(n)

求数组中所有元素的和：

```
def sum(nums):
    sum_num = 0
    for num in nums:
        sum_num += num
    return sum_num
```

(4) O(nlogn)

归并排序：

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2

    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
```

```

        result.append(right[j])
        j += 1

    result.extend(left[i:])
    result.extend(right[j:])

    return result

```

(5) $O(n^2)$

冒泡排序：

```

def bubble_sort(nums):
    for i in range(len(nums) - 1):
        for j in range(len(nums) - 1 - i):
            if nums[j] > nums[j + 1]:
                nums[j], nums[j + 1] = nums[j + 1], nums[j]

```

(6) $O(2^n)$

枚举全部子集：

```

def subsets(nums):
    """使用位运算来生成所有子集"""
    n = len(nums)
    result = []
    for i in range(1 << n):
        subset = []
        for j in range(n):
            if i & (1 << j):
                subset.append(nums[j])
        result.append(subset)
    return result

```

(7) $O(n!)$

全排列：

```

def permute(nums):
    result = []

    if len(nums) == 1:
        return [nums]

    for i in range(len(nums)):
        remaining = nums[:i] + nums[i + 1 :]
        for perm in permute(remaining):
            result.append([nums[i]] + perm)

    return result

```

2.2.4 空间复杂度

1) 空间复杂度定义

空间复杂度用来描述一个算法运行所需内存空间的多少。同时间复杂度类似，空间复杂度并不代表算法运行时所使用的绝对内存空间，它描述的是算法使用的内存空间随输入规模变大时的变化趋势。

程序运行时占用内存空间的主要有以下内容：

- 指令：程序编译后的二进制指令。
- 数据：程序声明的变量、常量、对象等内容。

我们在计算空间复杂度时，一般只需关注数据所占用的内存空间即可。

2) 常见的空间复杂度

(1) O(1)

常数阶常见于数量与输入数据大小无关的常量、变量、对象。

需要注意的是，在循环中初始化变量或调用函数而占用的内存，在进入下一循环后就会被释放，因此不会累积占用空间，空间复杂度仍为 O(1)。

原地反转数组：

```
def reverse_array(arr):  
    left, right = -1, len(arr)  
    while (left := left + 1) < (right := right - 1):  
        arr[left], arr[right] = arr[right], arr[left]
```

(2) O(n)

判断数组中是否有重复元素：

```
def has_duplicates(arr):  
    seen = set()  
    for num in arr:  
        if num in seen:  
            return True  
        seen.add(num)  
    return False
```

(3) O(n²)

矩阵转置：

```
def transpose(matrix):  
    n = len(matrix)  
    result = [[0] * n for _ in range(n)]
```

```

for i in range(n):
    for j in range(n):
        result[j][i] = matrix[i][j]

return result

```

第 3 章 常用数据结构

抽象数据类型（Abstract Data Type，简称 ADT）是计算机科学中一个重要的概念，它是对数据的一种抽象描述，关注数据的逻辑特性和操作，而不涉及具体的实现细节。

抽象数据类型通常由以下两部分组成：

- 数据对象：描述了该数据类型所包含的数据元素以及它们之间的逻辑关系。例如，在一个栈的抽象数据类型中，数据对象是一系列按后进先出（LIFO）原则组织的元素。
- 操作集合：定义了对数据对象可以执行的操作。对于栈来说，常见的操作包括入栈（push）、出栈（pop）、查看栈顶元素（peek）等。

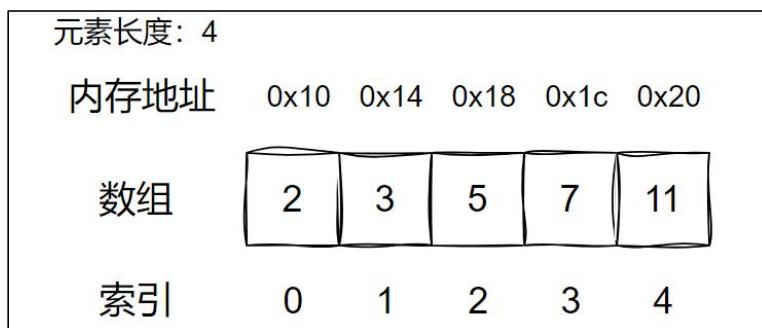
抽象数据类型与数据结构的关系

- 抽象数据类型：强调的是数据的逻辑特性和操作的功能，是一种抽象的概念，不涉及具体的实现细节。它是从用户的角度来描述数据和操作的。
- 数据结构：是抽象数据类型的 **具体实现**，它关注的是数据在计算机内存中的存储方式和操作的具体实现算法。例如，栈这种抽象数据类型可以用数组或链表等数据结构来实现。

3.1 数组

3.1.1 数组的概述

数组是一种线性数据结构，将相同类型的元素顺序地存储在连续的内存空间中，每个元素都有一个索引。



由于数组元素在内存中是连续存储的，所以只要知道数组的起始位置，以及数组元素的类型（单个元素的长度），就可以根据索引计算出任意元素的位置。

数组在创建时需要指定长度，并且数组一旦创建，长度就无法改变，如果需要扩容，只能创建一个更大的数组，再将原数据拷贝到新数组。并且由于数组的连续性，插入和删除数据可能需要移动其他元素。

在 Python 中，并没有像其他一些编程语言（如 C、Java）那样严格意义上的“数组”概念，但有多种数据结构可以用来模拟数组的功能，最常用的是列表（list），另外还有 array 模块的数组和 numpy 库的 ndarray。

通过 Python 的 list 列表实现一个动态数组，它内部存储的实际上是对象的引用（指针），而不是对象本身。每个引用指向内存中存储实际对象的位置。

3.1.2 数组的功能定义

| 方法 | 说明 |
|----------------------------|--------------|
| size() | 返回数组中元素个数 |
| is_empty() | 判断数组是否为空 |
| insert(index, item) | 在指定位置插入元素 |
| append(item) | 在末尾插入元素 |
| remove(index) | 删除指定位置的元素 |
| set(index, item) | 修改指定位置的元素 |
| get(index) | 获取指定位置的元素 |
| find(item) | 查找数组中某个元素的位置 |
| for_each(func) | 遍历数组 |

3.1.3 数组的创建

实现一个动态数组。

```
class Array:
    def __init__(self):
        """初始化数组"""
        self.__capacity = 8
        self.__size = 0
        self.__items = [0] * 8

    def __str__(self):
        """打印数组"""
        arr_str = "["
        for i in range(self.__size):
            arr_str += str(self.__items[i])
            if i < self.__size - 1:
                arr_str += ", "
        arr_str += "]"
        return arr_str
```

```

        arr_str += str(self.__items[i])
        if i < self.__size - 1:
            arr_str += ", "
    arr_str += "]"
    return arr_str

@property
def size(self):
    """获取数组元素个数"""
    return self.__size

def is_empty(self):
    """判断数组是否为空"""
    return self.__size == 0

```

3.1.4 数组扩容

当数组容量占满后，我们可以创建一个新的数组，容量为之前数组的 2 倍，并将之前数组的元素拷贝到新数组中。

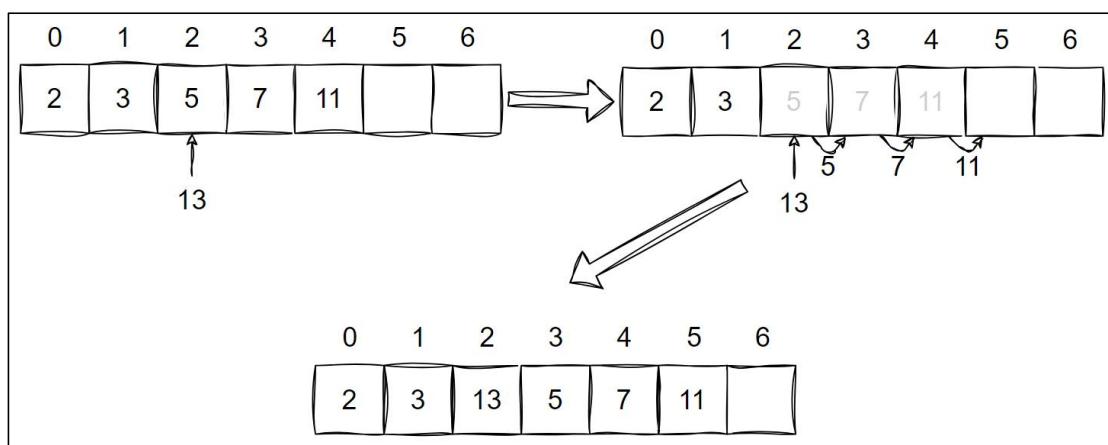
```

def __grow(self):
    """数组扩容"""
    self.new__items = [0] * self.__capacity * 2
    for i in range(self.__size):
        self.new__items[i] = self.__items[i]
    self.__items = self.new__items
    self.__capacity *= 2

```

3.1.5 插入元素

在中间插入元素时，将指定位置及其之后的元素全部向后移动一个位置，并将指定位置改为新的元素。



```
def insert(self, index, item):
```

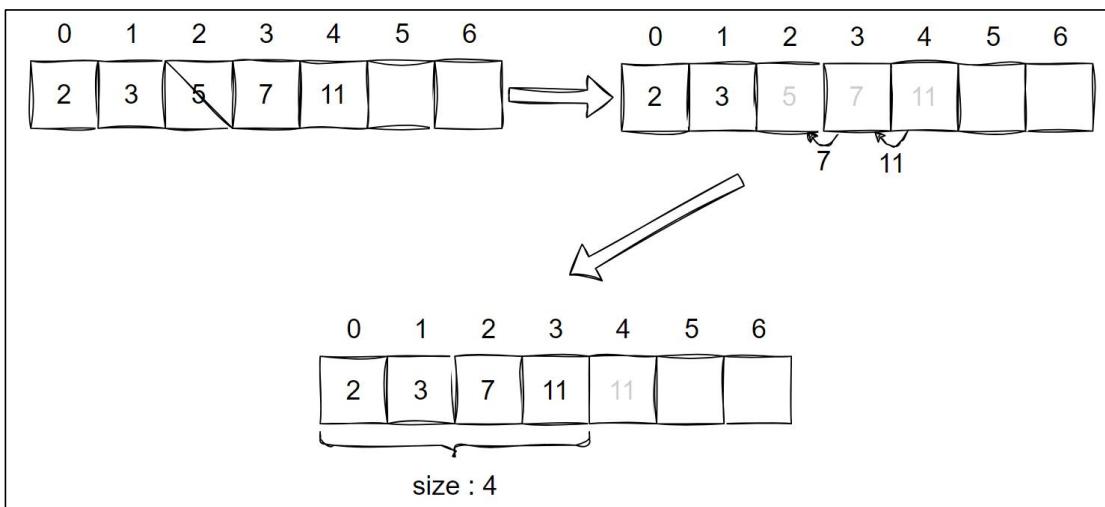
```
"""插入元素"""
if index < 0 or index > self._size:
    raise IndexError
if self._size == self._capacity:
    self._grow()
for i in range(self._size, index, -1):
    self._items[i] = self._items[i - 1]
self._items[index] = item
self._size += 1
```

在末尾插入元素时，使用 insert() 并将 index 设置为数组长度。

```
def append(self, item):
    """末尾插入元素"""
    self.insert(self._size, item)
```

3.1.6 删除元素

删除数组中指定位置的元素时，将该位置之后的所有元素向前移动一个位置。



```
def remove(self, index):
    """删除元素"""
    if index < 0 or index >= self._size:
        raise IndexError
    for i in range(index, self._size - 1):
        self._items[i] = self._items[i + 1]
    self._size -= 1
```

3.1.7 修改元素

```
def set(self, index, item):
    """修改元素"""
    if index < 0 or index >= self._size:
        raise IndexError
    self._items[index] = item
```

3.1.8 访问元素

```
def get(self, index):
    """访问元素"""
    if index < 0 or index >= self.__size:
        raise IndexError
    return self.__items[index]
```

3.1.9 查找元素

```
def find(self, item):
    """查找元素"""
    for i in range(self.__size):
        if self.__items[i] == item:
            return i
    return -1
```

3.1.10 遍历数组

```
def for_each(self, func):
    """遍历数组"""
    for i in range(self.__size):
        func(self.__items[i])
```

3.1.11 完整代码

```
class Array:
    def __init__(self):
        """初始化数组"""
        self.__capacity = 8
        self.__size = 0
        self.__items = [0] * 8

    def __str__(self):
        """打印数组"""
        arr_str = "["
        for i in range(self.__size):
            arr_str += str(self.__items[i])
            if i < self.__size - 1:
                arr_str += ","
        arr_str += "]"
        return arr_str

    @property
    def size(self):
        """获取数组元素个数"""
        return self.__size
```

```
def is_empty(self):
    """判断数组是否为空"""
    return self.__size == 0

def __grow(self):
    """数组扩容"""
    self.new__items = [0] * self.__capacity * 2
    for i in range(self.__size):
        self.new__items[i] = self.__items[i]
    self.__items = self.new__items
    self.__capacity *= 2

def insert(self, index, item):
    """插入元素"""
    if index < 0 or index > self.__size:
        raise IndexError
    if self.__size == self.__capacity:
        self.__grow()
    for i in range(self.__size, index, -1):
        self.__items[i] = self.__items[i - 1]
    self.__items[index] = item
    self.__size += 1

def append(self, item):
    """末尾插入元素"""
    self.insert(self.__size, item)

def remove(self, index):
    """删除元素"""
    if index < 0 or index >= self.__size:
        raise IndexError
    for i in range(index, self.__size - 1):
        self.__items[i] = self.__items[i + 1]
    self.__size -= 1

def set(self, index, item):
    """修改元素"""
    if index < 0 or index >= self.__size:
        raise IndexError
    self.__items[index] = item

def get(self, index):
```

```
"""访问元素"""
if index < 0 or index >= self.__size:
    raise IndexError
return self.__items[index]

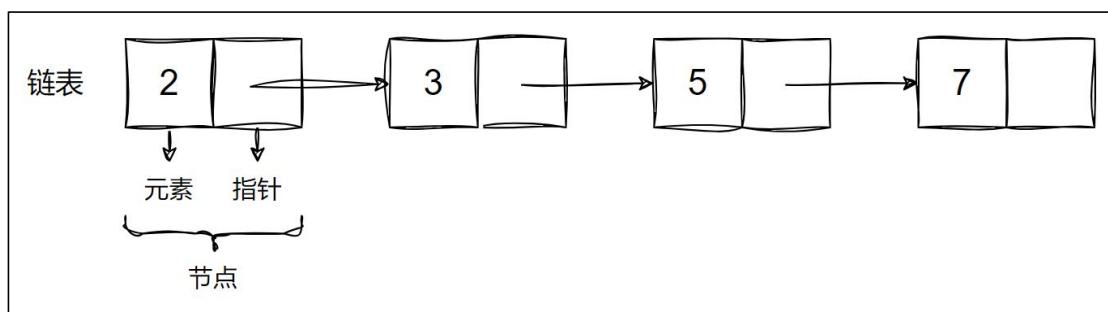
def find(self, item):
    """查找元素"""
    for i in range(self.__size):
        if self.__items[i] == item:
            return i
    return -1

def for_each(self, func):
    """遍历数组"""
    for i in range(self.__size):
        func(self.__items[i])
```

3.2 链表

3.2.1 链表的概述

链表（Linked List）是一个线性结构，由一系列节点（Node）组成，每个节点包含一个数据元素和一个指向下一节点的指针（Pointer）。所有节点通过指针相连，形成一个链式结构。通常我们将链表中的第一个节点称为头结点，并将头结点的位置作为整个链表的位置标识。与数组不同，链表中每个节点分散的存储在内存中，每个节点都保存了当前节点的数据和下一节点的地址（指针）。



由于链表中节点通过指针相连，插入和删除节点只需要修改指针的指向即可，而不需要像数组那样移动数据。且链表不需要像数组那样预先指定大小，而是可以随时动态的增长或缩小。由于链表使用分散存储的方式，因而无需使用大段连续的内存空间。

由于链表中的节点不是连续存储的，无法像数组一样根据索引直接计算出每个节点的地址。必须从头节点开始遍历链表，直到找到目标节点，这导致了链表的随机访问效率较低。

链表的每个节点都需要存储指向下一个节点的指针，这会占用额外的存储空间。相比于数组，链表需要更多的内存空间来存储相同数量的数据元素。

常见的链表包括三种：

- **单向链表**: 单向链表的节点包含值和指向下一节点的引用。我们将首个节点称为头节点，将最后一个节点称为尾节点，尾节点指向空 `None`。
- **环形链表**: 将单向链表的尾节点指向头节点（首尾相接），则得到一个环形链表。在环形链表中，任意节点都可以视作头节点。
- **双向链表**: 双向链表记录了两个方向的引用，同时包含指向后继节点（下一个节点）和前驱节点（上一个节点）的引用。

3.2.2 链表的功能定义

| 方法 | 说明 |
|----------------------------------|--------------|
| <code>size()</code> | 返回链表中元素个数 |
| <code>is_empty()</code> | 判断链表是否为空 |
| <code>insert(index, item)</code> | 在指定位置插入元素 |
| <code>append(item)</code> | 在末尾插入元素 |
| <code>remove(index)</code> | 删除指定位置的元素 |
| <code>set(index, item)</code> | 修改指定位置的元素 |
| <code>get(index)</code> | 获取指定位置的元素 |
| <code>find(item)</code> | 查找链表中某个元素的位置 |
| <code>for_each(func)</code> | 遍历链表 |

3.2.3 链表的创建

实现一个单向链表。

```
class Node:  
    def __init__(self, data, next=None):  
        self.data = data  
        self.next = next  
  
class LinkedList:  
    def __init__(self):  
        """初始化链表""""
```

```

        self.__head = None
        self.__size = 0

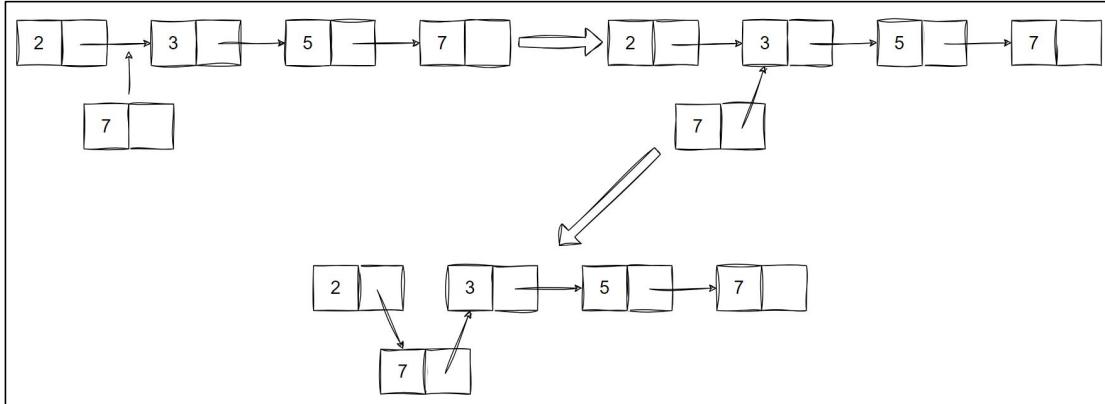
    def __str__(self):
        """打印链表"""
        result = []
        current = self.__head
        while current:
            result.append(str(current.data))
            current = current.next
        return " -> ".join(result)

    @property
    def size(self):
        """获取链表元素个数"""
        return self.__size

    def is_empty(self):
        """判断链表是否为空"""
        return self.__size == 0

```

3.2.4 插入元素



```

def insert(self, index, item):
    """插入元素"""
    if index < 0 or index > self.__size:
        raise IndexError
    if index == 0:
        # 插入到头部, 需要新建一个节点, 然后让新节点的 next 指向原来的 head,
        # 然后让 head 指向新节点
        self.__head = Node(item, self.__head)
    else:
        # 插入到中间, 先找到 index-1 位置的节点
        node = self.__head
        for i in range(index - 1):

```

```

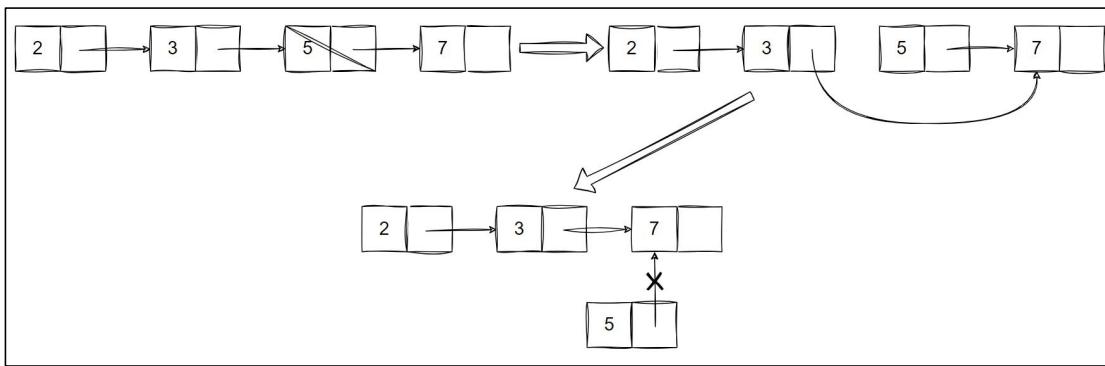
        node = node.next
        # 新节点的 next 指向 index 位置的节点，然后让 index-1 位置的节点的
        # next 指向新节点
        node.next = Node(item, node.next)
        self.__size += 1
    
```

在末尾插入元素时，使用 insert() 并将 index 设置为链表长度。

```

def append(self, item):
    """末尾插入元素"""
    self.insert(self.__size, item)
    
```

3.2.5 删除元素



```

def remove(self, index):
    """删除元素"""
    if index < 0 or index >= self.__size:
        raise IndexError
    if index == 0:
        self.__head = self.__head.next
    else:
        # 找到 index-1 位置的节点，然后让 index-1 位置的节点的 next 指向 index
        # 位置的节点的 next
        node = self.__head
        for i in range(index - 1):
            node = node.next
        node.next = node.next.next
    self.__size -= 1
    
```

3.2.6 修改元素

```

def set(self, index, item):
    """修改元素"""
    if index < 0 or index >= self.__size:
        raise IndexError
    node = self.__head
    for i in range(index):
        node = node.next
    
```

```
node.data = item
```

3.2.7 访问元素

```
def get(self, index):
    """访问元素"""
    if index < 0 or index >= self.__size:
        raise IndexError
    node = self.__head
    for i in range(index):
        node = node.next
    return node.data
```

3.2.8 查找元素

```
def find(self, item):
    """查找元素"""
    node = self.__head
    while node:
        if node.data == item:
            return True
        node = node.next
    return False
```

3.2.9 遍历链表

```
def for_each(self, func):
    """遍历链表"""
    node = self.__head
    while node:
        func(node)
        node = node.next
```

3.2.10 完整代码

```
class Node:
    def __init__(self, data, next=None):
        self.data = data
        self.next = next


class LinkedList:
    def __init__(self):
        """初始化链表"""
        self.__head = None
        self.__size = 0

    def __str__(self):
```

```
"""打印链表"""
result = []
current = self.__head
while current:
    result.append(str(current.data))
    current = current.next
return " -> ".join(result)

@property
def size(self):
    """获取链表元素个数"""
    return self.__size

def is_empty(self):
    """判断链表是否为空"""
    return self.__size == 0

def insert(self, index, item):
    """插入元素"""
    if index < 0 or index > self.__size:
        raise IndexError
    if index == 0:
        # 插入到头部, 需要新建一个节点, 然后让新节点的 next 指向原来的 head,
        然后让 head 指向新节点
        self.__head = Node(item, self.__head)
    else:
        # 插入到中间, 先找到 index-1 位置的节点
        node = self.__head
        for i in range(index - 1):
            node = node.next
        # 新节点的 next 指向 index 位置的节点, 然后让 index-1 位置的节点的
        next 指向新节点
        node.next = Node(item, node.next)
    self.__size += 1

def append(self, item):
    """末尾插入元素"""
    self.insert(self.__size, item)

def remove(self, index):
    """删除元素"""
    if index < 0 or index >= self.__size:
        raise IndexError
    if index == 0:
```

```
        self.__head = self.__head.next
    else:
        # 找到 index-1 位置的节点, 然后让 index-1 位置的节点的 next 指向 index
        # 位置的节点的 next
        node = self.__head
        for i in range(index - 1):
            node = node.next
        node.next = node.next.next
        self.__size -= 1

    def set(self, index, item):
        """修改元素"""
        if index < 0 or index >= self.__size:
            raise IndexError
        node = self.__head
        for i in range(index):
            node = node.next
        node.data = item

    def get(self, index):
        """访问元素"""
        if index < 0 or index >= self.__size:
            raise IndexError
        node = self.__head
        for i in range(index):
            node = node.next
        return node.data

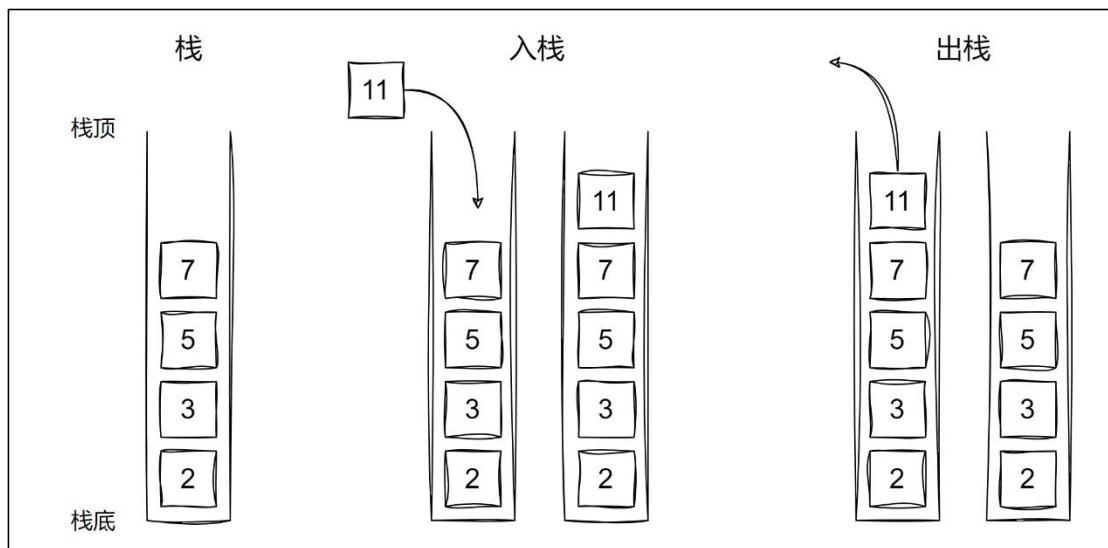
    def find(self, item):
        """查找元素"""
        node = self.__head
        while node:
            if node.data == item:
                return True
            node = node.next
        return False

    def for_each(self, func):
        """遍历链表"""
        node = self.__head
        while node:
            func(node)
            node = node.next
```

3.3 栈

3.3.1 栈的概述

栈 (Stack) 是一个线性结构，其维护了一个有序的数据列表，列表的一端称为栈顶 (top)，另一端称为栈底 (bottom)。栈对数据的操作有明确限定，插入元素只能从栈顶进行，删除元素也只能从栈顶开始逐个进行，通常将插入元素称为入栈 (push)，删除元素称为出栈 (pop)。正是由于上述规定，栈保证了后进先出的原则 (LIFO, Last-In-First-Out)。



栈的底层实现既可以选择数组也可以选择链表，只要能保证后进先出的原则即可。

3.3.2 栈的功能定义

| 方法 | 说明 |
|-------------------------|------------------|
| <code>size()</code> | 返回栈中元素个数 |
| <code>is_empty()</code> | 判断栈是否为空 |
| <code>push(item)</code> | 将新元素压入栈中 |
| <code>pop()</code> | 获取栈顶元素，并将栈顶元素弹出栈 |
| <code>peek()</code> | 获取栈顶元素，但不弹出栈 |

3.3.3 栈的实现

使用动态数组实现一个栈。

```
class Stack:
    def __init__(self):
        """初始化栈"""

```

```
        self.__size = 0
        self.__items = []

    @property
    def size(self):
        """获取栈元素个数"""
        return self.__size

    def is_empty(self):
        """判断栈是否为空"""
        return self.__size == 0

    def push(self, item):
        """入栈"""
        self.__items.append(item)
        self.__size += 1

    def pop(self):
        """出栈"""
        if self.is_empty():
            raise Exception("栈为空")
        item = self.__items[self.__size - 1]
        del self.__items[self.__size - 1]
        self.__size -= 1
        return item

    def peek(self):
        """访问栈顶元素"""
        if self.is_empty():
            raise Exception("栈为空")
        return self.__items[self.__size - 1]
```

3.3.4 栈的应用

1) 有效括号

力扣 20 题 <https://leetcode.cn/problems/valid-parentheses/>

(1) 题目描述

给定一个只包括 “(” , “)” , “[” , “]” , “{” , “}” 的字符串 s, 判断字符串是否有效。

有效字符串需满足：

- 左括号必须用相同类型的右括号闭合。

- 左括号必须以正确的顺序闭合。
- 每个右括号都有一个对应的相同类型的左括号。

(2) 示例

示例 1:

输入: s = "()"

输出: true

示例 2:

输入: s = "O[]{}"

输出: true

示例 3:

输入: s = "()"

输出: false

示例 4:

输入: s = "(())"

输出: true

(3) 思路分析

遇到左括号则入栈，遇到右括号则出栈一个左括号与之匹配，如果能够匹配则继续，如果匹配失败或者栈为空则返回 False。

(4) 代码实现

```
class Solution:  
    def isValid(self, s):  
        stack = []  
        for i in s:  
            match i:  
                case "(" | "[" | "{":  
                    stack.append(i)  
                case ")":  
                    if (not stack) or (stack.pop() != "("):  
                        return False  
                case "]":  
                    if (not stack) or (stack.pop() != "["):  
                        return False  
                case "}":  
                    if (not stack) or (stack.pop() != "{"):
```

```

        return False
    return True if not stack else False

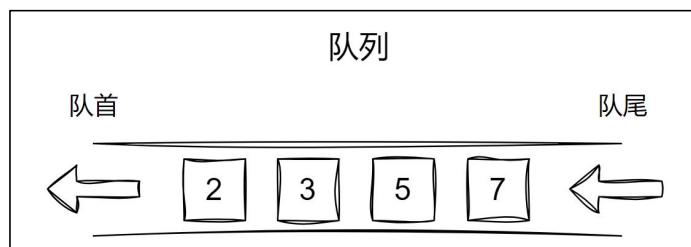
if __name__ == "__main__":
    solution = Solution()
    s = "()[]{}"
    print(s, solution.isValid(s))
    s = "([]"
    print(s, solution.isValid(s))
    s = "([)]"
    print(s, solution.isValid(s))
    s = "{[}]"
    print(s, solution.isValid(s))

```

3.4 队列

3.4.1 队列的概述

队列（Queue）也是一个线性结构，其同样维护了一个有序的数据列表，队列的一端称为队首，另一端称为队尾。队列也对数据操作做出了明确限定，插入元素只能从队尾进行，删除元素只能从队首进行，通常将插入操作称为入队（enqueue），将删除操作称为出队（dequeue）。也正是由于上述限制，队列保证了先进先出（FIFO，First-In-First-Out）的原则。



队列的底层实现既可以选择数组也可以选择链表，只要能保证先进先出的原则即可。

常见的队列包括两种：

- 单向队列：只能从一端插入数据，从另一端删除数据，遵循先进先出。
- 双向队列：在队列的两端都可以进行插入和删除操作。

3.4.2 队列的功能定义

| 方法 | 说明 |
|-------------------|-----------|
| size() | 返回队列中元素个数 |
| is_empty() | 判断队列是否为空 |

| | |
|-------------------|---------|
| push(item) | 向队尾添加元素 |
| pop() | 从队首取出元素 |
| peek() | 访问队首元素 |

3.4.3 队列的实现

使用链表实现一个单向队列。

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class Queue:
    def __init__(self):
        """初始化队列"""
        self.__head = None
        self.__tail = None
        self.__size = 0

    @property
    def size(self):
        """获取队列元素个数"""
        return self.__size

    def is_empty(self):
        """判断队列是否为空"""
        return self.__size == 0

    def push(self, data):
        """入队"""
        node = Node(data)
        if self.is_empty():
            self.__head = node
            self.__tail = node
        else:
            self.__tail.next = node
            self.__tail = node
        self.__size += 1

    def pop(self):
        """出队"""
        if self.is_empty():
            return None
        data = self.__head.data
        self.__head = self.__head.next
        self.__size -= 1
        return data
```

```
        raise Exception("队列为空")
    data = self.__head.data
    self.__head = self.__head.next
    self.__size -= 1
    return data

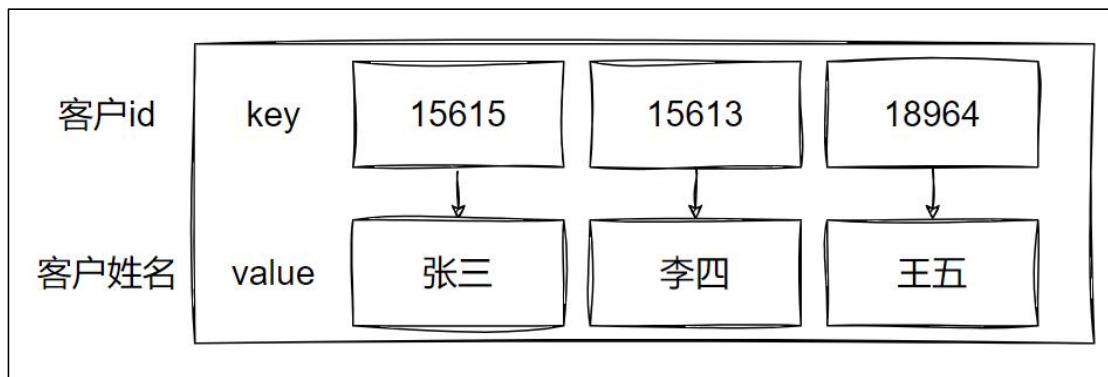
def peek(self):
    """访问队首元素"""
    if self.is_empty():
        raise Exception("队列为空")
    return self.__head.data
```

3.5 哈希表

3.5.1 哈希表的概述

哈希表（Hash Table，也叫散列表），由一系列键值对（key-value pairs）组成，并且可以通过键（key）查找对应的值（value）。哈希表通过建立 key 与 value 之间的映射，实现高效的查询，我们向哈希表中输入一个 key，可以在 O(1) 的时间内获取对应的 value。

例如通过客户 id 获取客户姓名：



哈希表常见的一个操作是根据 key 来查找 value，考虑到数组查询效率最高，选择基于数组实现哈希表。利用哈希函数计算 key 的哈希值，然后将哈希值映射到数组索引。在实现过程中我们可能会遇到如下问题：

- 如何将一个个 key 映射到数组的索引？
- 如果多个 key 映射到数组同一个索引怎么办？
- 数组长度是固定的，如果后续元素过多，大于数组长度怎么办？

1) 哈希函数

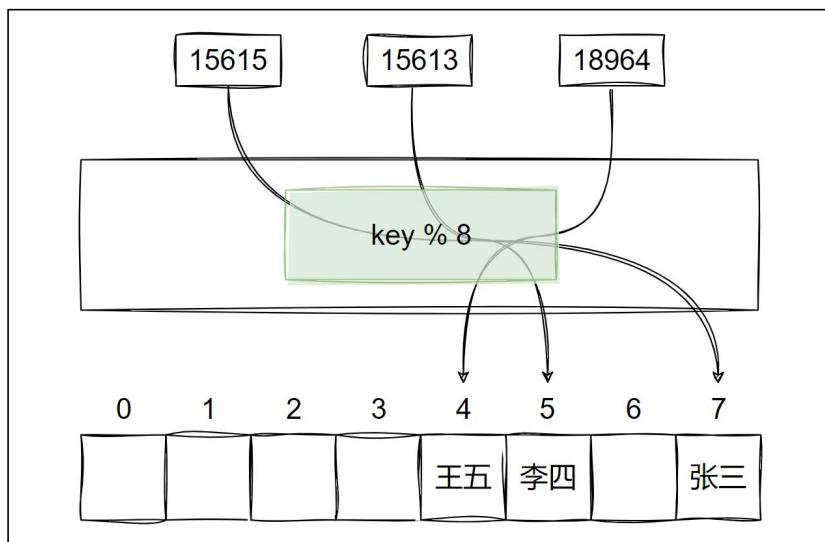
哈希表的核心组件是哈希函数。该函数将 key 转换为一个数组索引。哈希函数的目标是尽量均匀地将所有可能的 key 分布到表的不同位置，以减少冲突的发生。

哈希函数的执行步骤分为两步：

- 通过某种哈希算法计算出 key 的哈希值。
- 哈希值对数组长度取余，获取 key 对应的数组索引。

`index = hash(key) % capacity`

例如我们使用一个简单的哈希算法 $\text{hash}(\text{key})=\text{key}$ 将客户 id 映射到一个长度为 8 的数组的索引，即 $\text{index} = \text{key} \% 8$ 。



常见的哈希算法：

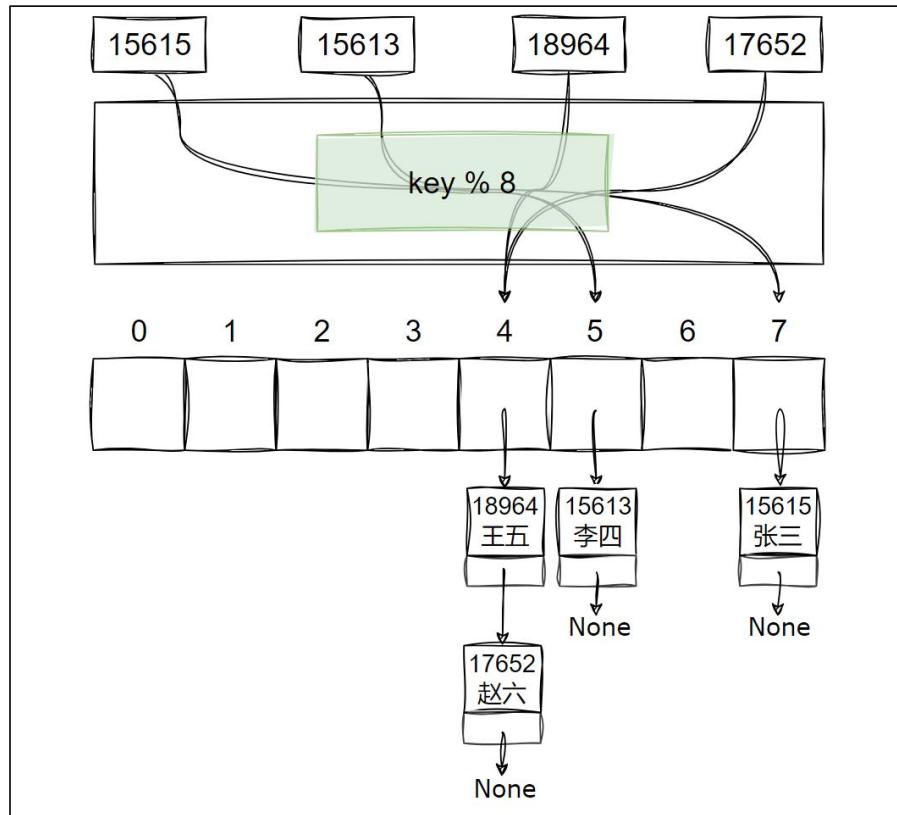
- 通用哈希算法：除法哈希、乘法哈希、MurmurHash、CityHash。
- 加密哈希算法：MD5（已被成功攻击）、SHA-1（已被成功攻击）、SHA-2、SHA-3。
- 文件完整性检查算法：Adler-32、CRC32。

2) 哈希冲突

哈希函数可能会将不同的键值映射到同一个索引位置，这就是所谓的哈希冲突。处理冲突的方式有多种，最常见的两种是链式法（Chaining）和开放寻址法（Open Addressing）。

(1) 链式法

将发生碰撞的每个键值对作为一个节点（Node）组成一个链表（Linked List），然后将链表的头节点保存在数组的目标位置中。这样一来，向字典中写入数据时，若发现数组的目标位置已有数据，那么就将当前的键值对作为一个节点插入链表；从字典中读取数据时，则从数组的目标位置获取链表，并进行遍历，直到找到目标数据。



(2) 开放寻址法

当发生冲突时根据某种探查策略寻找下一个空槽位。常见的探查策略包括：

- 线性探查（Linear Probing）：如果当前位置已经被占用，就探查下一个位置。
- 二次探查（Quadratic Probing）：以平方的步长进行探查。
- 双重哈希（Double Hashing）：使用另一个哈希函数来计算新的索引。

3) 负载因子

负载因子（Load Factor）是哈希表中元素个数与表的大小的比率。当负载因子过高时，可能需要进行扩容操作，以保持操作的效率。

较小的负载因子可以减少冲突的可能性，较大的负载因子可以提高哈希表的内存利用率。通常情况下负载因子在 0.7~0.8 是一个比较好的选择。

3.5.2 哈希表的功能定义

| 方法 | 说明 |
|------------------------|-------------|
| size() | 返回哈希表中键值对个数 |
| is_empty() | 判断哈希表是否为空 |
| put(key, value) | 向哈希表插入键值对 |

| | |
|-----------------------|---------------|
| remove(key) | 从哈希表中根据键删除键值对 |
| get(key) | 从哈希表中根据键获取值 |
| for_each(func) | 遍历哈希表中的键值对 |

3.5.3 哈希表的实现

```

class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.next = None


class HashTable:

    def __init__(self):
        """初始化哈希表"""
        self.__capacity = 8 # 数组长度
        self.__size = 0 # 键值对个数
        self.__load_factor = 0.7 # 负载因子
        self.__table = [None] * self.__capacity

    def display(self):
        """显示哈希表内容"""
        for i, node in enumerate(self.__table):
            print(f"Index {i}: ", end="")
            current = node
            while current:
                print(f"({current.key}, {current.value}) -> ", end="")
                current = current.next
            print("None")
        print()

    def __hash(self, key):
        """哈希函数，根据 key 计算索引"""
        return hash(key) % self.__capacity

    def __grow(self):
        """哈希表负载因子超过阈值时进行扩容"""
        self.__capacity = self.__capacity * 2
        self.__table, old_table = [None] * self.__capacity, self.__table
        self.__size = 0

```

```
# 将旧哈希表中的元素重新插入到新的哈希表中
for node in old_table:
    current = node
    while current:
        self.put(current.key, current.value)
        current = current.next

@property
def size(self):
    """获取哈希表键值对个数"""
    return self.__size

def is_empty(self):
    """判断哈希表是否为空"""
    return self.__size == 0

def put(self, key, value):
    """插入键值对，处理哈希冲突"""
    # 如果负载因子超过阈值则进行扩容
    if self.__size / self.__capacity > self.__load_factor:
        self.__grow()

    index = self.__hash(key)
    new_node = Node(key, value)
    # 如果当前位置为空，直接插入
    if self.__table[index] is None:
        self.__table[index] = new_node
    else:
        # 否则，发生哈希冲突，链式存储
        current = self.__table[index]
        while current and current.next:
            # 如果键已经存在，更新值
            if current.key == key:
                current.value = value
                return
            current = current.next
        # 如果键不存在，插入到链表尾部
        current.next = new_node
    self.__size += 1

def remove(self, key):
    """删除键值对"""
    index = self.__hash(key)
    current = self.__table[index]
```

```
prev = None

while current:
    if current.key == key:
        if prev:
            # 删除非头节点
            prev.next = current.next
        else:
            # 删除头节点
            self.__table[index] = current.next
        self.__size -= 1
        return True
    prev = current
    current = current.next
return False

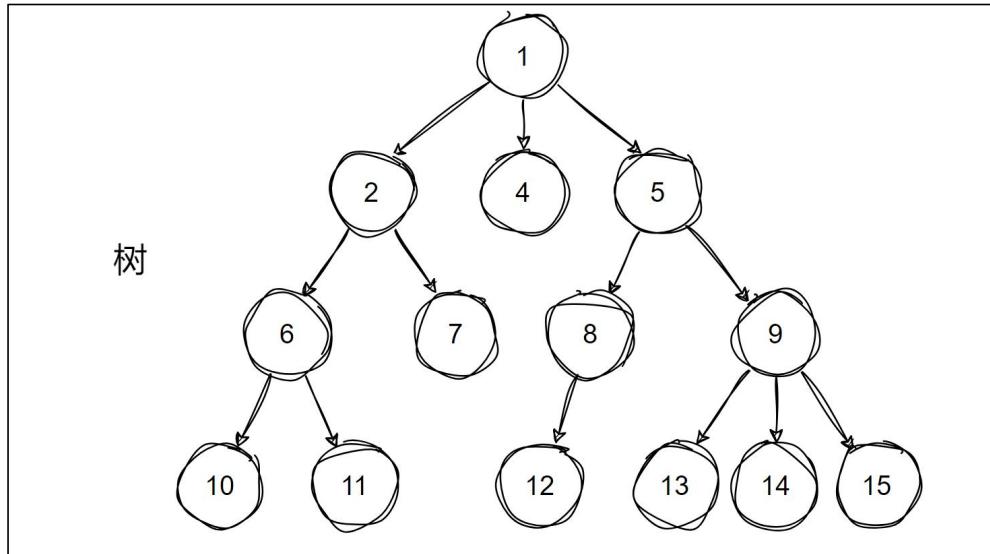
def get(self, key):
    """访问键值对"""
    index = self.__hash(key)
    current = self.__table[index]
    while current:
        if current.key == key:
            return current.value
        current = current.next
    return None

def for_each(self, func):
    """遍历哈希表"""
    for node in self.__table:
        current = node
        while current:
            func(current.key, current.value)
            current = current.next
```

3.6 树

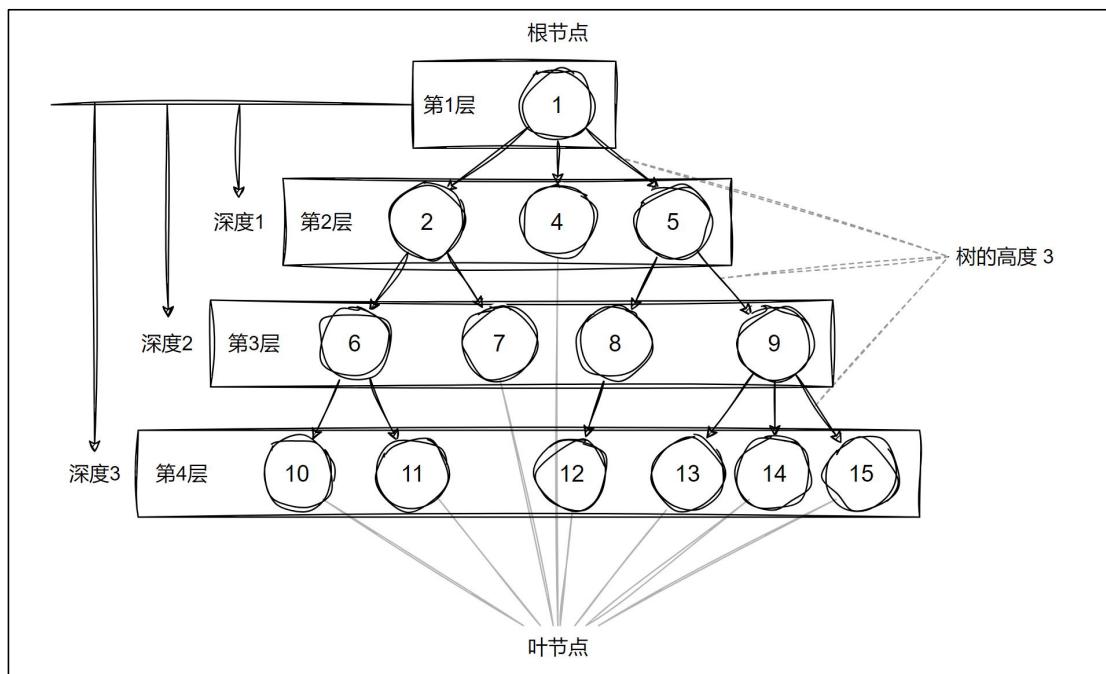
3.6.1 树的概述

树（Tree）由一系列具有层次关系的节点（Node）组成。



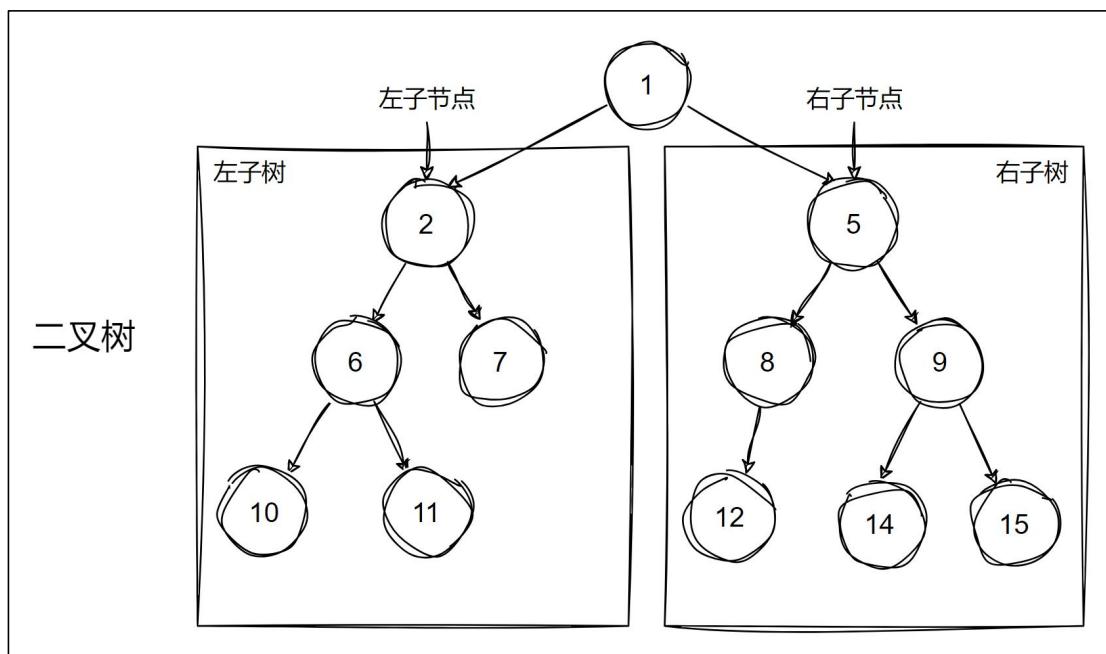
树的常见术语：

- 父节点：节点的上层节点。
- 子节点：节点的下层节点。
- 根节点：位于树的顶端，没有父节点的节点。
- 叶节点：位于树的底端，没有子节点的节点。
- 边：连接两个节点的线段。
- 节点的度：节点的子节点数量。
- 节点的层：从根开始定义起，根为第 1 层，根的子节点为第 2 层，以此类推。
- 节点的深度：从根节点到该节点所经过的边的数量，根的深度为 0。
- 节点的高度：从距离该节点最远的叶节点到该节点所经过的边的数量，所有叶节点的高度为 0。
- 树的深度（高度）：从根节点到最远叶节点所经过的边的数量。



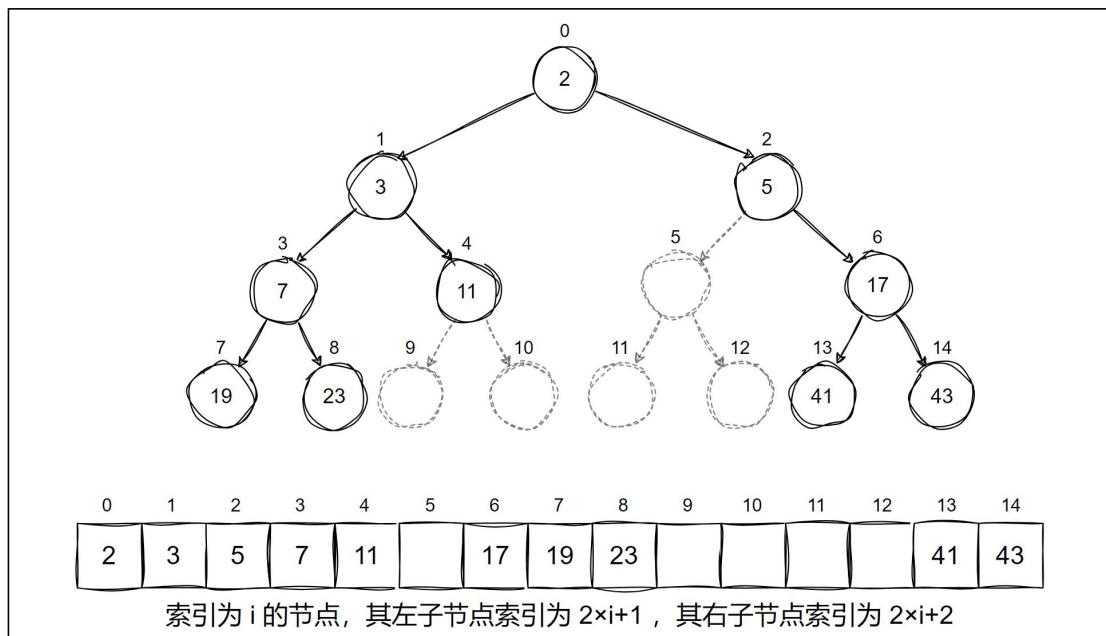
3.6.2 二叉树简介

树形结构中最具代表性的一种就是二叉树（Binary Tree）。二叉树规定，每个节点最多只能有两个子节点，两个子节点分别被称为左子节点和右子节点。以左子节点为根节点的子树被称为左子树，以右子节点为根节点的子树被称为右子树。



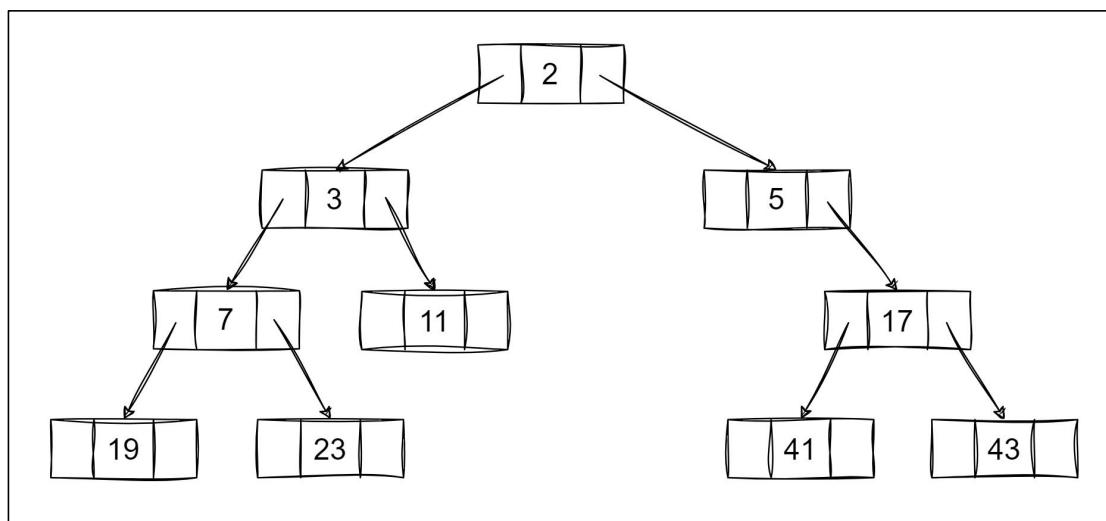
3.6.3 二叉树存储结构

1) 二叉树的数组存储



采用数组结构存储二叉树，访问与遍历速度较快。但不适合存储数据量过大的树，且增删效率较低，而且树中存在大量 None 的情况下空间利用率较低，因此不是主流方式。

2) 二叉树的链表存储

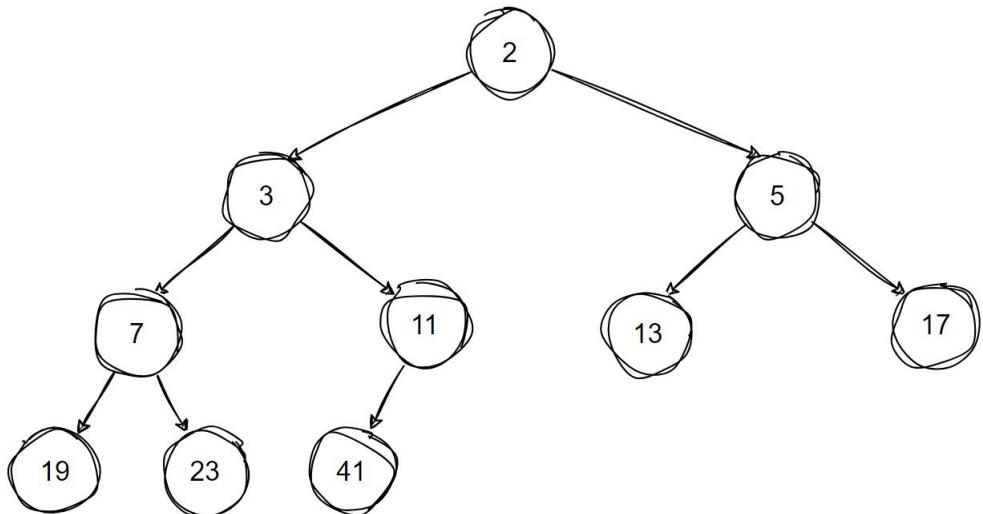


3.6.4 常见的二叉树

1) 完全二叉树

完全二叉树只有最下面一层的节点未被填满，且靠左填充。

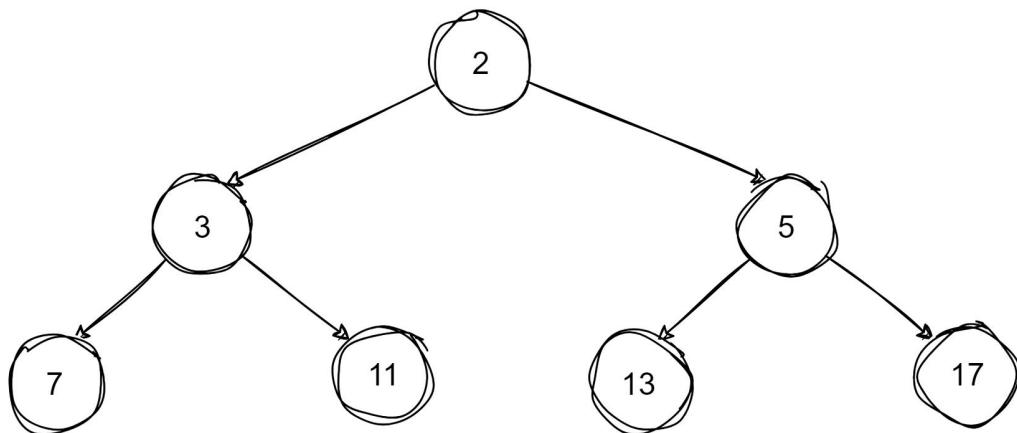
完全二叉树



2) 满二叉树

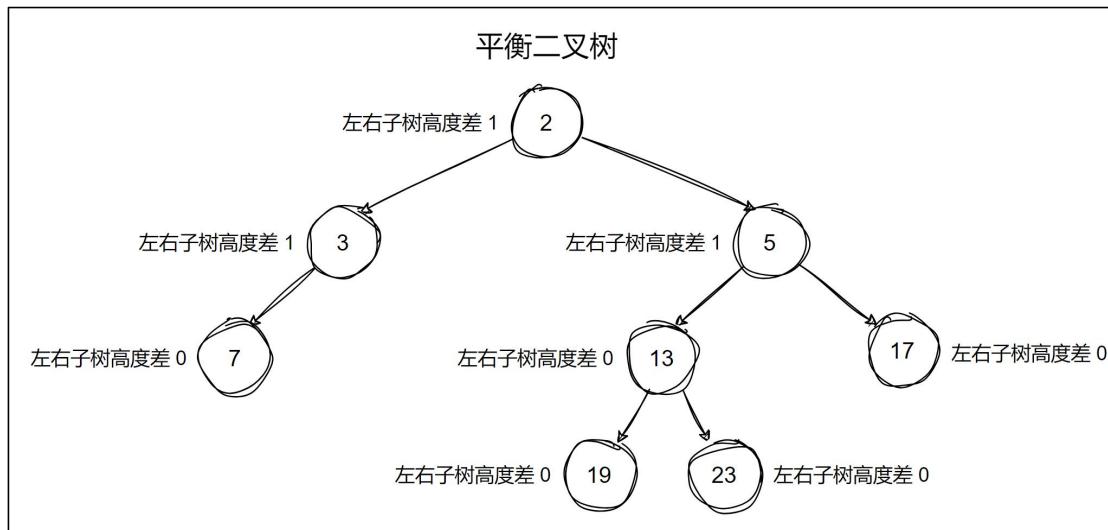
满二叉树所有层的节点都被完全填满，满二叉树也是一种完全二叉树。

满二叉树



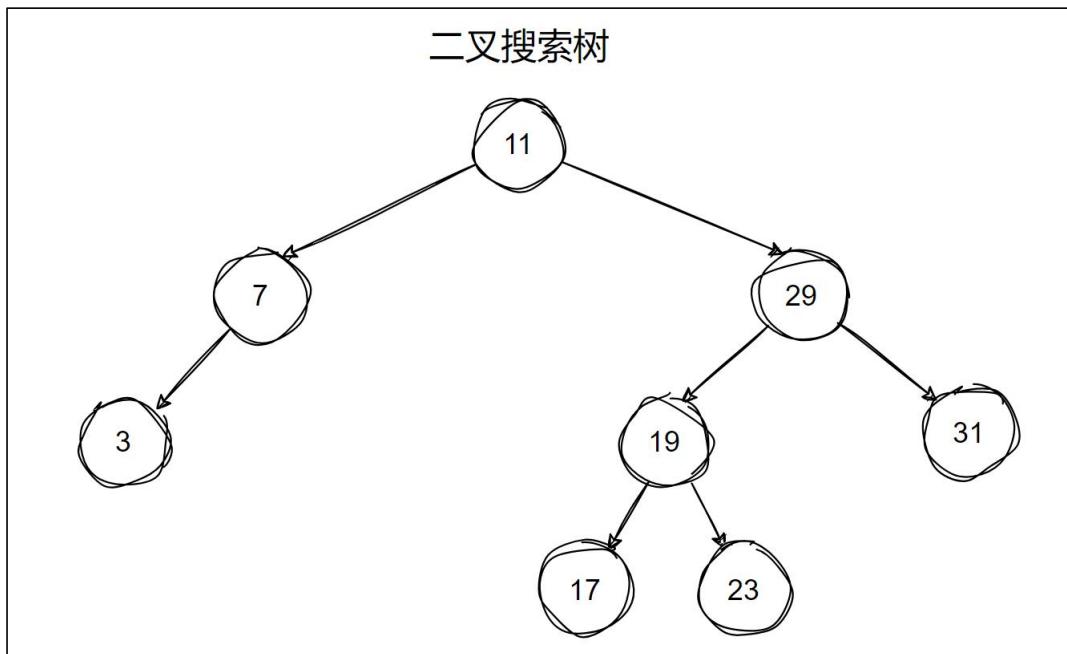
3) 平衡二叉树

平衡二叉树中任意节点的左右子树高度之差不超过 1。



4) 二叉搜索树

二叉搜索树中的每个节点的值，大于其左子树中的所有节点的值，并且小于右子树中的所有节点的值。



5) AVL 树

AVL 树是一种自平衡的二叉搜索树，插入和删除时会进行旋转操作来保证树的平衡性。

6) 红黑树

红黑树是一种特殊的二叉搜索树，除了二叉搜索树的要求外，它还具有以下特性：

- 每个节点或者是黑色，或者是红色。
- 根节点是黑色。
- 每个叶节点都是黑色。这里叶节点是指为空（None）的节点。

- 红色节点的两个子节点必须是黑色的。即从每个叶到根的所有路径上不能有两个连续的红色节点。
- 从任一个节点到其每个叶的所有路径上包含相同数目的黑色节点。

7) 堆

堆 (Heap) 是一种满足特定条件的完全二叉树，主要可分为两种类型：

- 大顶堆：每个父节点的值都大于等于其子节点的值。根节点为树中的最大值。
- 小顶堆：每个父节点的值都小于等于其子节点的值。根节点为树中的最小值。

8) 霍夫曼树

霍夫曼树又称最优二叉树，是一种带权路径长度最短的二叉树，通常用于数据压缩，它的构建基于字符出现频率的概率。

9) B 树

B 树是一种自平衡的多路查找树。虽然它不是严格意义上的二叉树，但与二叉树的结构类似。经常用于数据库、文件系统等需要磁盘访问的应用。

10) B+树

B+树是 B 树的优化版本。它通过将数据集中存储在叶子节点并通过链表连接来实现高效的范围查询，并且非叶子节点仅存储索引，提高了磁盘利用率。

3.6.5 二叉搜索树的功能定义

| 方法 | 说明 |
|------------------------------|-------------|
| size() | 返回树中节点个数 |
| is_empty() | 判断树是否为空 |
| search(item) | 查找节点是否存在 |
| add(item) | 向二叉搜索树中插入节点 |
| remove(item) | 从二叉搜索树中删除节点 |
| for_each(func, order) | 按指定方式遍历二叉树 |

3.6.6 二叉树的创建

```
from collections import deque

class Node:
    """二叉树节点"""

```

```
def __init__(self, data):
    self.data = data
    self.left = None
    self.right = None

class BinarySearchTree:
    """二叉搜索树"""

    def __init__(self):
        """初始化二叉树"""
        self.__root = None
        self.__size = 0

    def print_tree(self):
        """打印树的结构"""

        # 先得到树的层数
        def get_layer(node):
            """递归计算树的层数"""
            if node is None:
                return 0
            else:
                left_depth = get_layer(node.left)
                right_depth = get_layer(node.right)
                return max(left_depth, right_depth) + 1

        layer = get_layer(self.__root)

        # 层序遍历并打印
        queue = deque([(self.__root, 1)])
        current_level = 1
        while queue:
            node, level = queue.popleft()
            if level > current_level:
                print()
                current_level += 1
            if node:
                print(f"{node.data:^{20*layer//2**(level-1)}}, end='''")
            else:
                print(f"N:{^{20*layer//2**(level-1)}}, end='''")
            if level < layer:
                if node:
```

```

        queue.append((node.left, level + 1))
        queue.append((node.right, level + 1))
    else:
        queue.append((None, level + 1))
        queue.append((None, level + 1))
    print()

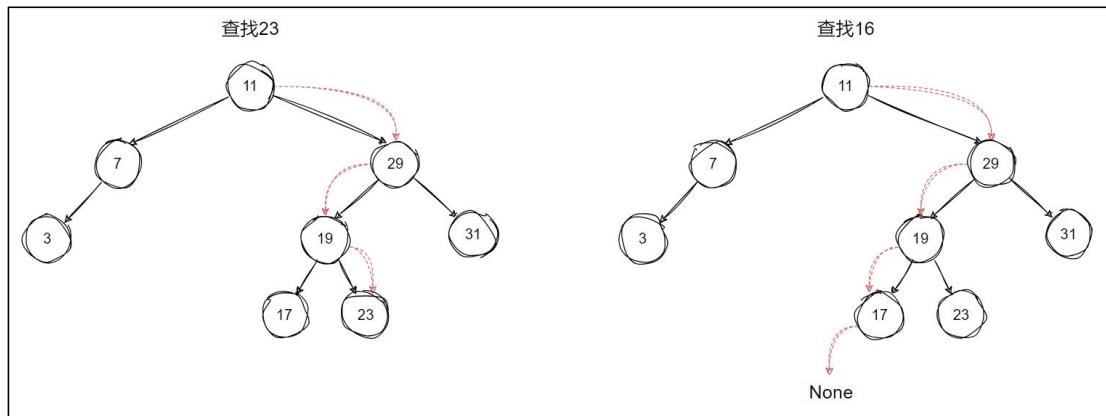
@property
def size(self):
    """返回树中节点的个数"""
    return self.__size

def is_empty(self):
    """判断树是否为空"""
    return self.__size == 0

```

3.6.7 二叉搜索树的查找操作

查找时先与当前节点比较大小，等于则找到了目标节点，小于则向左子节点查找，大于则向右子节点查找。如果查找到 None 仍未找到则说明该节点不在树中。



后续插入与删除操作也会用到查找，所以此处提供一个`_search_pos()`方法，返回查找到的节点和其父节点供后续使用。

```

def search(self, item):
    """查找节点是否存在"""
    return self.__search_pos(item)[0] is not None

def __search_pos(self, item):
    """查找节点，返回(节点,父节点)。如果节点不存在则为None，此时父节点为一个叶节点"""
    parent = None
    current = self.__root
    while current:

```

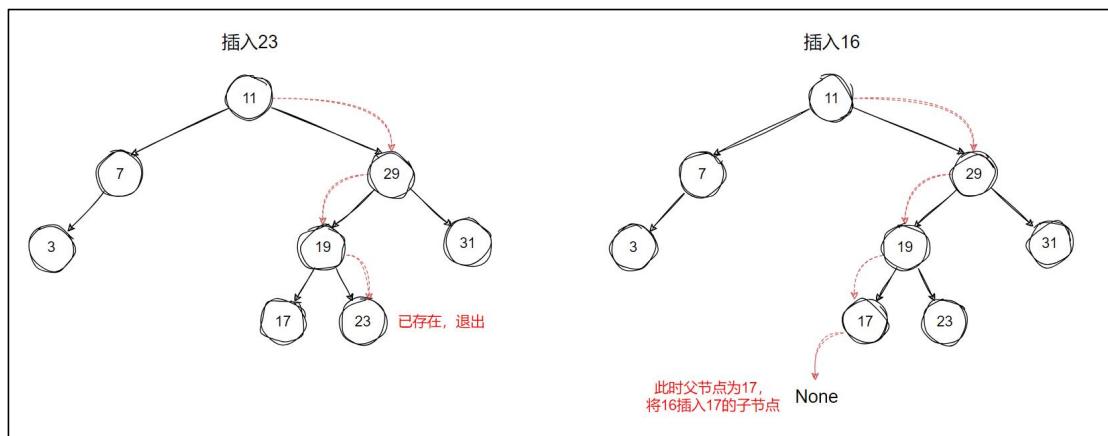
```

        if item == current.data:
            break
        parent = current
        current = current.left if item < current.data else current.right
    return current, parent

```

3.6.8 二叉搜索树的插入操作

插入时先执行查找操作，查找时保存当前节点的父节点。如果找到了节点则说明树中已有此元素，退出。如果找到了 None，此时 None 的父节点为叶节点，应将该元素插入该叶节点的子节点。



```

def add(self, item):
    """插入节点"""
    node = Node(item)
    if self.is_empty():
        self._root = node
    else:
        current, parent = self._search_pos(item)
        # 如果节点之前已存在则返回
        if current:
            return
        # 如果节点之前不存在，则插入父节点的左节点或右节点
        if parent.data > item:
            parent.left = node
        else:
            parent.right = node
    self.__size += 1

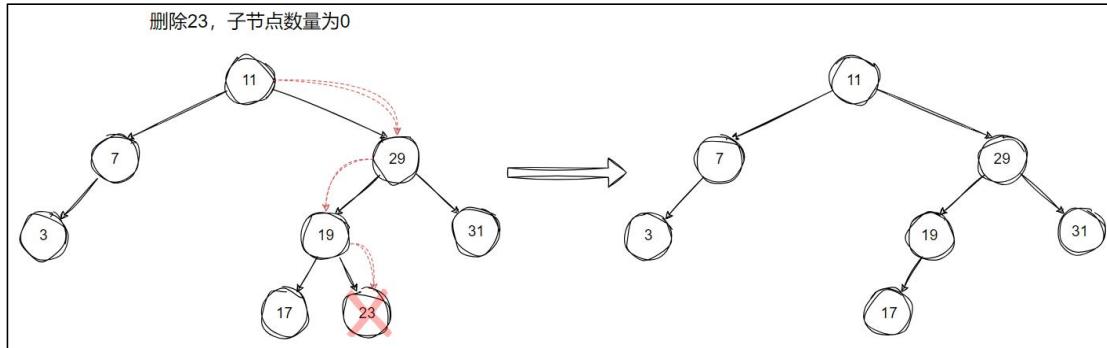
```

3.6.9 二叉搜索树的删除操作

需要保证删除节点后仍然保证二叉搜索树的性质。删除操作需要根据目标节点的子节点数量为 0、1、2 分三种情况。

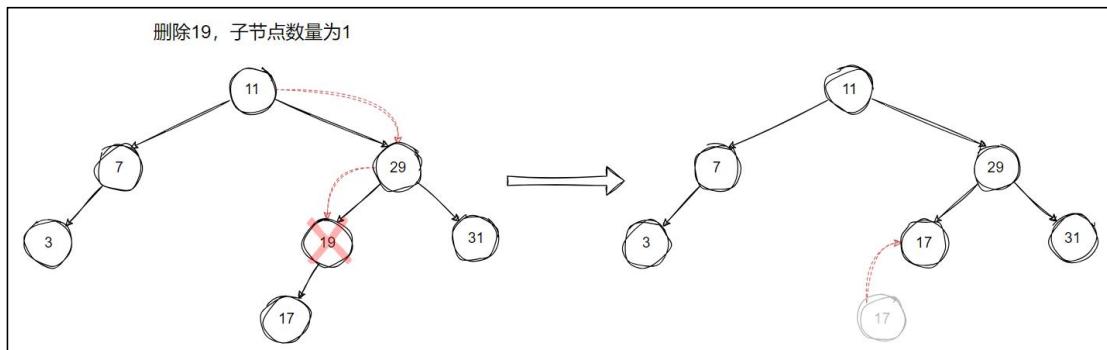
1) 目标节点的子节点数量为 0

直接删除目标节点。



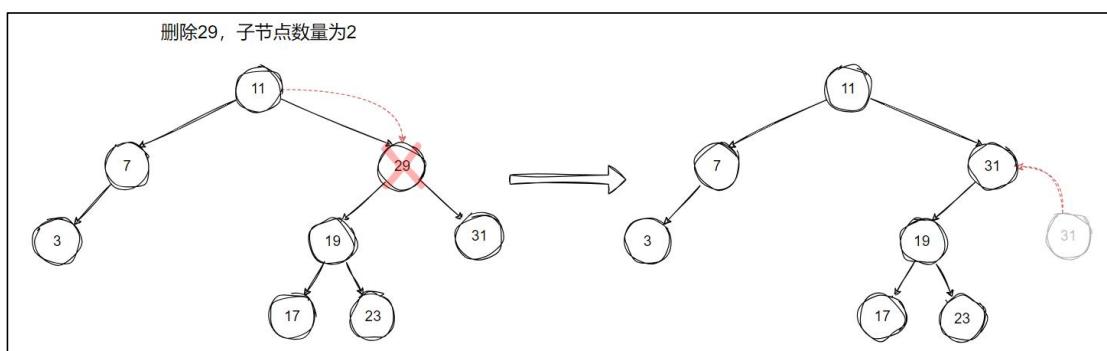
2) 目标节点的子节点数量为 1

将目标节点替换为其子节点。



3) 目标节点的子节点数量为 2

使用目标节点的右子树最小节点、或左子树最大节点替换目标节点。



4) 代码实现

```
def remove(self, item):
    """删除节点"""
    current, parent = self._search_pos(item)
    if not current:
        return
```

```
# 如果删除的是叶节点（没有子节点）
if not current.left and not current.right:
    if parent:
        if parent.left == current:
            parent.left = None
        else:
            parent.right = None
    else:
        # 如果没有父节点，说明是根节点
        self.__root = None

# 如果删除的节点只有一个子节点
elif not current.left or not current.right:
    child = current.left if current.left else current.right
    if parent:
        if parent.left == current:
            parent.left = child
        else:
            parent.right = child
    else:
        # 如果没有父节点，说明是根节点
        self.__root = child

# 如果删除的节点有两个子节点
else:
    # 找到中序后继（右子树中最小的节点）
    successor = self.__get_min(current.right)
    successor_data = successor.data
    # 删除中序后继节点
    self.remove(successor_data)
    # 用中序后继的值替代当前节点
    current.data = successor_data

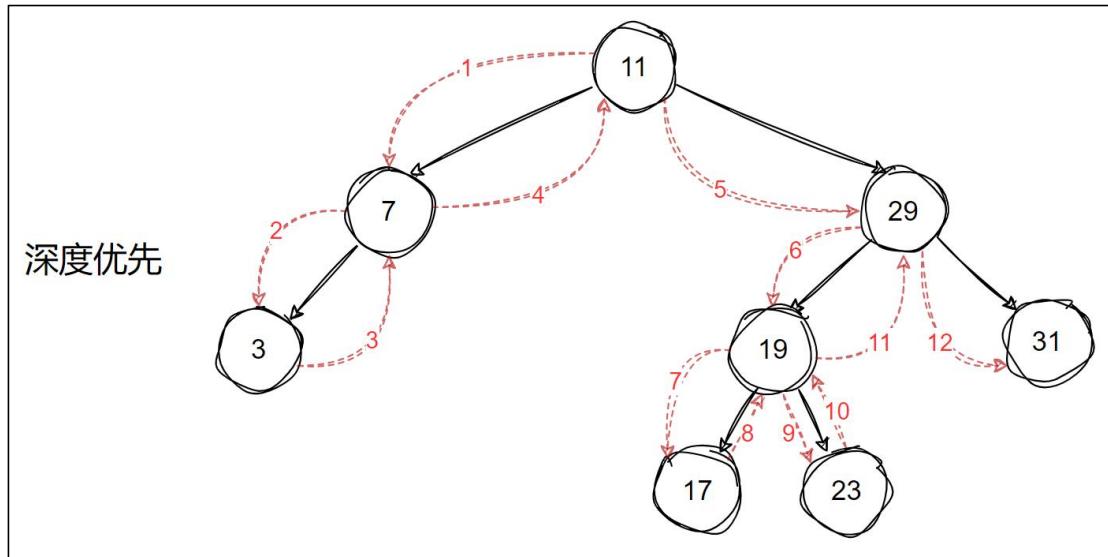
    self.__size -= 1

def __get_min(self, node):
    """找到当前子树的最小节点"""
    current = node
    while current.left:
        current = current.left
    return current
```

3.6.10 二叉树的遍历

1) 深度优先

深度优先搜索（DFS， Depth First Search）尽可能地深入每一个分支，直到不能再深入为止，然后回溯到上一个节点，继续尝试其他的分支。



(1) 前序遍历

先访问当前节点，再访问节点的左子树，再访问节点的右子树。

```
def dfs(node):
    """前序遍历"""
    if node is None:
        return
    print(node) # 访问当前节点
    dfs(node.left) # 访问节点的左子树
    dfs(node.right) # 访问节点的右子树
```

(2) 中序遍历

先访问节点的左子树，再访问当前节点，再访问节点的右子树。

二叉搜索树中序遍历的结果是有序的。

```
def dfs(node):
    """中序遍历"""
    if node is None:
        return
    dfs(node.left) # 访问节点的左子树
    print(node) # 访问当前节点
    dfs(node.right) # 访问节点的右子树
```

(3) 后续遍历

先访问节点的左子树，再访问节点的右子树，再访问当前节点。

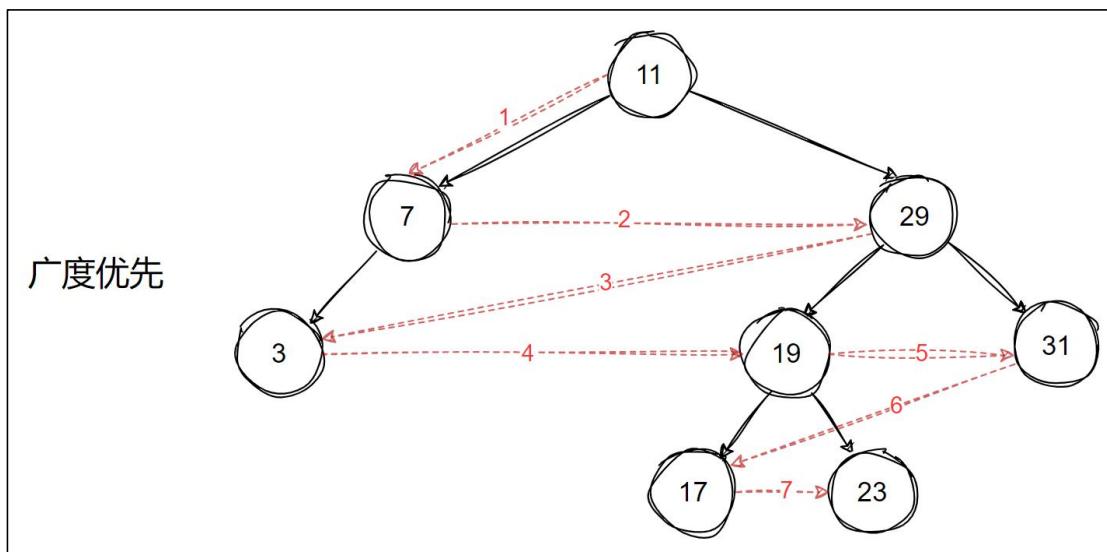
```
def dfs(node):
```

```
"""后序遍历"""
if node is None:
    return
dfs(node.left) # 访问节点的左子树
dfs(node.right) # 访问节点的右子树
print(node) # 访问当前节点
```

2) 广度优先

(1) 层序遍历

广度优先搜索 (BFS, Breadth First Search) 从起始节点开始, 首先访问该节点的所有子节点, 然后再访问子节点的子节点, 依此类推, 逐层访问节点。



广度优先搜索一般使用队列实现, 每访问一个节点, 就将该节点的子节点添加进队列中。

3) 代码实现

```
def for_each(self, func, order="inorder"):
    """遍历树, 默认中序遍历"""
    match order:
        case "inorder":
            self.__inorder_traversal(func)
        case "preorder":
            self.__preorder_traversal(func)
        case "postorder":
            self.__postorder_traversal(func)
        case "levelorder":
            self.__levelorder_traversal(func)

def __inorder_traversal(self, func):
    """深度优先搜索: 中序遍历"""

    def __traversal(node):
        if node is None:
            return
        __traversal(node.left)
        func(node)
        __traversal(node.right)
```

```
def inorder(node):
    if node:
        inorder(node.left)
        func(node.data)
        inorder(node.right)

inorder(self.__root)

def __preorder_traversal(self, func):
    """深度优先搜索：前序遍历"""

    def preorder(node):
        if node:
            func(node.data)
            preorder(node.left)
            preorder(node.right)

    preorder(self.__root)

def __postorder_traversal(self, func):
    """深度优先搜索：后序遍历"""

    def postorder(node):
        if node:
            postorder(node.left)
            postorder(node.right)
            func(node.data)

    postorder(self.__root)

def __levelorder_traversal(self, func):
    """广度优先搜索：层序遍历"""

    queue = deque()
    queue.append(self.__root)
    while queue:
        node = queue.popleft()
        func(node.data)
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
```

3.6.11 完整代码

```
from collections import deque
```

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

```
class Node:
    """二叉树节点"""

    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


class BinarySearchTree:
    """二叉搜索树"""

    def __init__(self):
        """初始化二叉树"""
        self.__root = None
        self.__size = 0

    def print_tree(self):
        """打印树的结构"""

        # 先得到树的层数
        def get_layer(node):
            """递归计算树的层数"""
            if node is None:
                return 0
            else:
                left_depth = get_layer(node.left)
                right_depth = get_layer(node.right)
                return max(left_depth, right_depth) + 1

        layer = get_layer(self.__root)
        # 层序遍历并打印
        queue = deque([(self.__root, 1)])
        current_level = 1
        while queue:
            node, level = queue.popleft()
            if level > current_level:
                print()
                current_level += 1
            if node:
                print(f"{node.data:{20*layer//2**level-1}}", end="")
            else:
```

```
        print(f"{'N':^{20*layer//2** (level-1)}}", end="")
    if level < layer:
        if node:
            queue.append((node.left, level + 1))
            queue.append((node.right, level + 1))
        else:
            queue.append((None, level + 1))
            queue.append((None, level + 1))
    print()

@property
def size(self):
    """返回树中节点的个数"""
    return self.__size

def is_empty(self):
    """判断树是否为空"""
    return self.__size == 0

def search(self, item):
    """查找节点是否存在"""
    return self.__search_pos(item)[0] is not None

def __search_pos(self, item):
    """查找节点，返回(节点,父节点)。如果节点不存在则为None，此时父节点为一个叶节点"""
    parent = None
    current = self.__root
    while current:
        if item == current.data:
            break
        parent = current
        current = current.left if item < current.data else current.right
    return current, parent

def add(self, item):
    """插入节点"""
    node = Node(item)
    if self.is_empty():
        self.__root = node
    else:
        current, parent = self.__search_pos(item)
        # 如果节点之前已存在则返回
        if current:
```

```
        return
    # 如果节点之前不存在，则插入父节点的左节点或右节点
    if parent.data > item:
        parent.left = node
    else:
        parent.right = node
    self.__size += 1

def remove(self, item):
    """删除节点"""
    current, parent = self.__search_pos(item)
    if not current:
        return

    # 如果删除的是叶节点（没有子节点）
    if not current.left and not current.right:
        if parent:
            if parent.left == current:
                parent.left = None
            else:
                parent.right = None
        else:
            # 如果没有父节点，说明是根节点
            self.__root = None

    # 如果删除的节点只有一个子节点
    elif not current.left or not current.right:
        child = current.left if current.left else current.right
        if parent:
            if parent.left == current:
                parent.left = child
            else:
                parent.right = child
        else:
            # 如果没有父节点，说明是根节点
            self.__root = child

    # 如果删除的节点有两个子节点
    else:
        # 找到中序后继（右子树中最小的节点）
        successor = self.__get_min(current.right)
        successor_data = successor.data
        # 删除中序后继节点
        self.remove(successor_data)
```

```
# 用中序后继的值替代当前节点
current.data = successor_data

self.__size -= 1

def __get_min(self, node):
    """找到当前子树的最小节点"""
    current = node
    while current.left:
        current = current.left
    return current

def for_each(self, func, order="inorder"):
    """遍历树， 默认中序遍历"""
    match order:
        case "inorder":
            self.__inorder_traversal(func)
        case "preorder":
            self.__preorder_traversal(func)
        case "postorder":
            self.__postorder_traversal(func)
        case "levelorder":
            self.__levelorder_traversal(func)

def __inorder_traversal(self, func):
    """深度优先搜索： 中序遍历"""

    def inorder(node):
        if node:
            inorder(node.left)
            func(node.data)
            inorder(node.right)

    inorder(self.__root)

def __preorder_traversal(self, func):
    """深度优先搜索： 前序遍历"""

    def preorder(node):
        if node:
            func(node.data)
            preorder(node.left)
            preorder(node.right)
```

```
preorder(self.__root)

def __postorder_traversal(self, func):
    """深度优先搜索：后序遍历"""

    def postorder(node):
        if node:
            postorder(node.left)
            postorder(node.right)
            func(node.data)

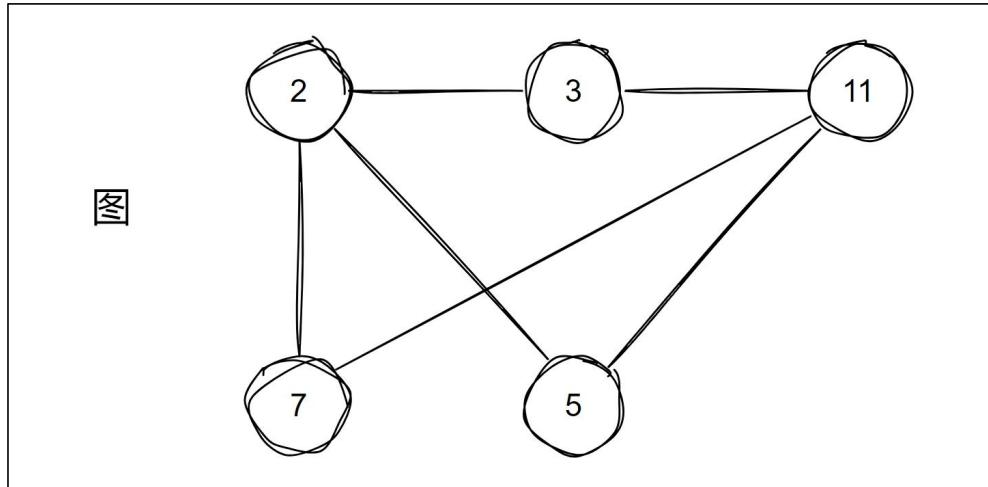
    postorder(self.__root)

def __levelorder_traversal(self, func):
    """广度优先搜索：层序遍历"""
    queue = deque()
    queue.append(self.__root)
    while queue:
        node = queue.popleft()
        func(node.data)
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
```

3.7 图

3.7.1 图的概述

前面我们学习了线性结构和树，线性结构局限于只有一个直接前驱和一个直接后继的关系，树也只能有一个直接前驱，也就是父节点，当我们需要表示多对多的关系时，就需要用到图了，图是比树更普遍的结构，可以认为树是一种特殊的图。图由节点和边组成。



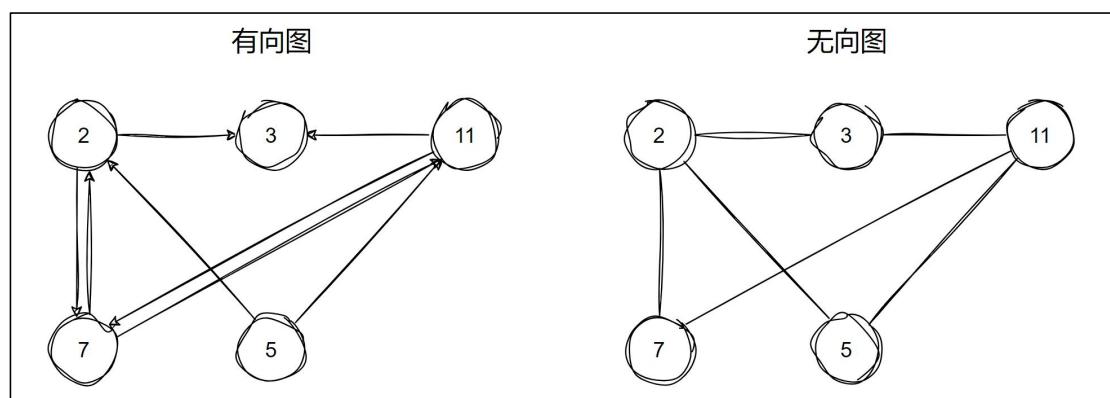
图的常见术语：

- 节点：也称为顶点，是图的基础部分。
- 边：连接两个节点，也是图的基础部分。可以是单向的，也可以是双向的。
- 权重：边可以添加“权重”变量。
- 邻接：两节点之间存在边，则称这两个节点邻接。
- 度：一个节点的边的数量。入度为指向该节点的边的数量，出度为该节点指向其他节点的边的数量。
- 路径：从一节点到另一节点所经过的边的序列。
- 环：首尾节点相同的路径。

3.7.2 图的分类

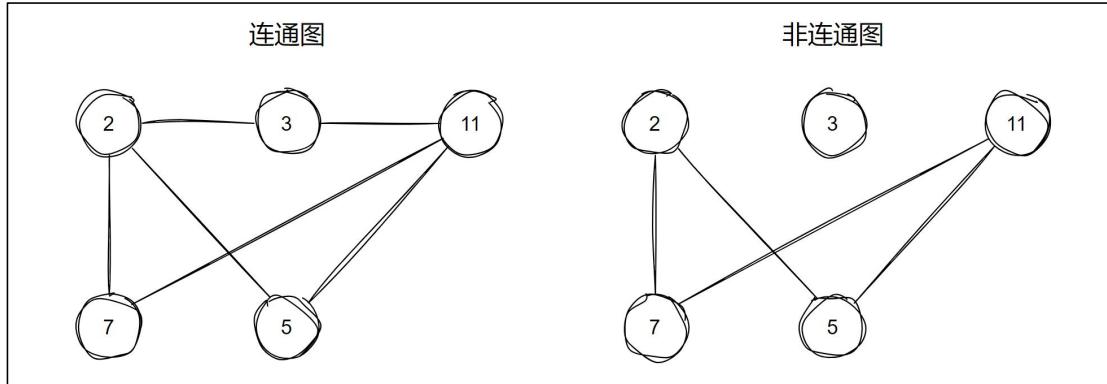
1) 有向图和无向图

- 有向图：边是单向的。
- 无向图：边是双向的。



2) 连通图和非连通图

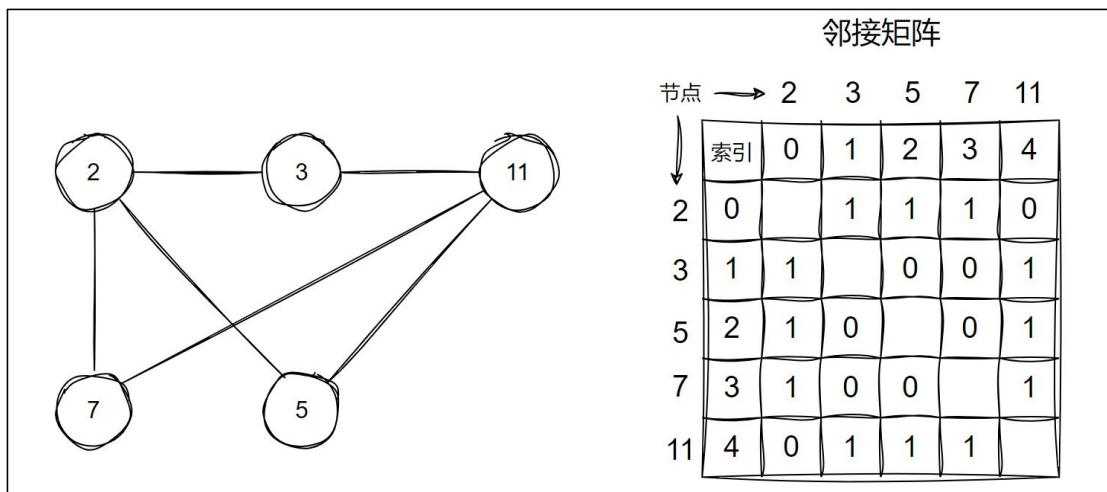
- 连通图：从某个节点出发，可以到达其余任意节点。
- 非连通图：从某个节点出发，有节点不可达。



3.7.3 图的常用表示法

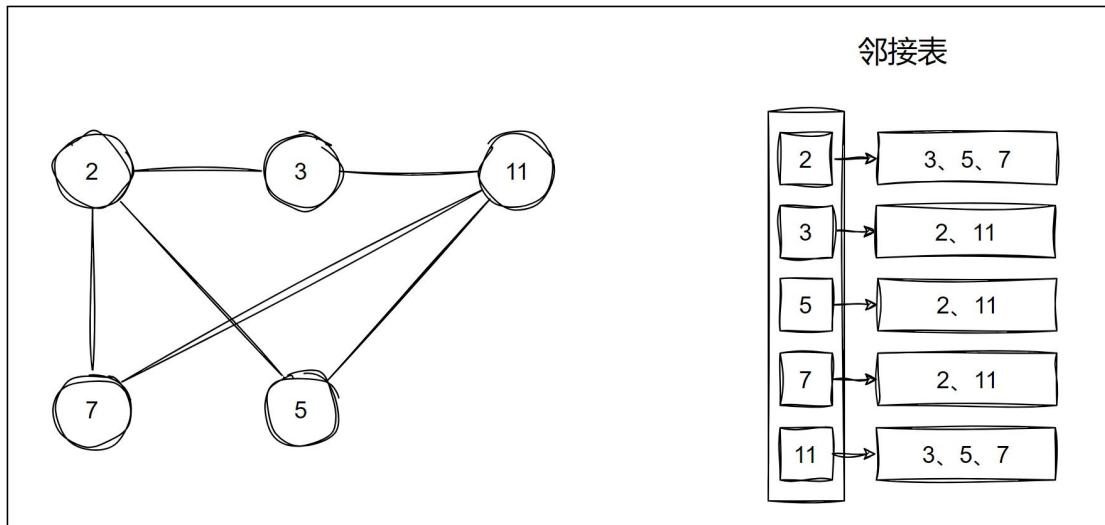
1) 邻接矩阵

邻接矩阵用一个 $n \times n$ 的矩阵来表示有 n 个节点之间的关系，矩阵的每一行（列）代表一个节点，矩阵 m 行 n 列的值代表是否存在由 m 指向 n 的边。邻接矩阵适合存储稠密图。



2) 邻接表

邻接表存储 n 个链表、列表或其他容器，每个容器存储该节点的所有邻接节点。邻接表适合存储稀疏图，空间效率高，尤其在处理边远少于节点的图时表现优越，但在进行边查找时不如邻接矩阵高效。



3.7.4 图的遍历

1) 广度优先搜索

广度优先搜索 (BFS, Breadth First Search) 从起始节点开始，首先访问该节点的所有邻接节点，然后再访问邻接节点的邻接节点，依此类推，逐层访问节点。

- 从图的起始节点开始，首先访问该节点，并标记为已访问。
- 然后依次访问所有未被访问的邻接节点，并将它们加入到队列中。
- 当队列中的节点被访问时，继续访问它的邻接节点，并将新的节点加入队列。
- 直到队列为空，表示所有节点都已被访问。

2) 深度优先搜索

深度优先搜索 (DFS, Depth First Search) 尽可能地深入到图的每一个分支，直到不能再深入为止，然后回溯到上一个节点，继续尝试其他的分支。

- 从图的一个起始节点开始，访问这个节点并标记为已访问。
- 对于每个未访问的邻接节点，递归地执行 DFS，直到没有未访问的邻接节点。
- 当回溯到一个节点时，继续访问它的其他邻接节点。

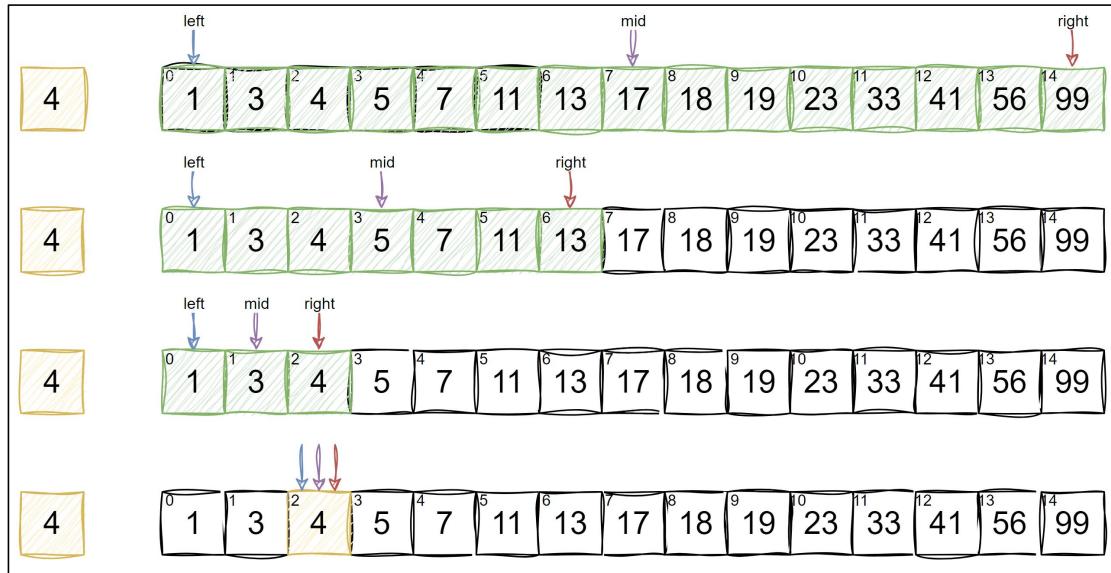
第 4 章 常用算法

4.1 查找算法

4.1.1 二分查找

1) 算法原理

二分查找又称折半查找，适用于有序列表。其利用数据的有序性，每轮缩小一半搜索范围，直至找到目标元素或搜索区间为空为止。



2) 代码实现

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            return mid # 找到目标值，返回索引
        elif arr[mid] < target:
            left = mid + 1 # 目标值在右半部分
        else:
            right = mid - 1 # 目标值在左半部分

    return -1 # 未找到目标值
```

3) 复杂度分析

(1) 时间复杂度

在循环中，区间每轮缩小一半，因此时间复杂度为 $O(\log n)$ 。

(2) 空间复杂度

使用常数大小的额外空间，空间复杂度为 $O(1)$ 。

4.1.2 查找多数元素

力扣 169 题 <https://leetcode.cn/problems/majority-element/description/>

返回数组中数量超过半数的元素，要求时间复杂度 $O(n)$ 、空间复杂度 $O(1)$ 。

示例：

- 输入： nums = [2,2,1,1,1,2,2]
- 输出： 2

1) 思路分析

为了严格符合复杂度要求，可以使用多数投票算法，多数投票算法也叫摩尔投票算法。摩尔投票算法的核心思想是对立性和抵消，它基于这样一个事实：如果一个元素在数组中出现的次数超过数组长度的一半，那么在不断消除不同元素对的过程中，这个多数元素最终会留下来。

具体来说，算法维护两个变量：一个是候选元素 candidate，另一个是该候选元素的计数 count。在遍历数组的过程中，遇到与候选元素相同的元素时，计数加 1；遇到不同的元素时，计数减 1。当计数减为 0 时，说明当前候选元素被抵消完，需要更换候选元素为当前遍历到的元素，并将计数重置为 1。

算法步骤

- **初始化：**
 - 选择数组的第一个元素作为初始候选元素 candidate。
 - 将计数 count 初始化为 1。
- **遍历数组：**
 - 从数组的第二个元素开始遍历。
 - 若当前元素与候选元素相同，count 加 1。
 - 若当前元素与候选元素不同，count 减 1。
 - 当 count 变为 0 时，将当前元素设为新的候选元素，并将 count 重置为 1。
- **返回结果：**
 - 遍历结束后，candidate 即为多数元素。

代码实现

```
def majorityElement(nums):  
    # 初始化候选元素为数组的第一个元素  
    candidate = nums[0]  
    # 初始化候选元素的票数为 1  
    count = 1  
    # 从数组的第二个元素开始遍历
```

```
for num in nums[1:]:
    if num == candidate:
        # 如果当前元素与候选元素相同，票数加 1
        count += 1
    else:
        # 如果当前元素与候选元素不同，票数减 1
        count -= 1
    if count == 0:
        # 当票数为 0 时，更新候选元素为当前元素，并将票数重置为 1
        candidate = num
        count = 1
return candidate

# 测试示例
nums = [2, 2, 1, 1, 1, 2, 2]
print(majorityElement(nums))
```

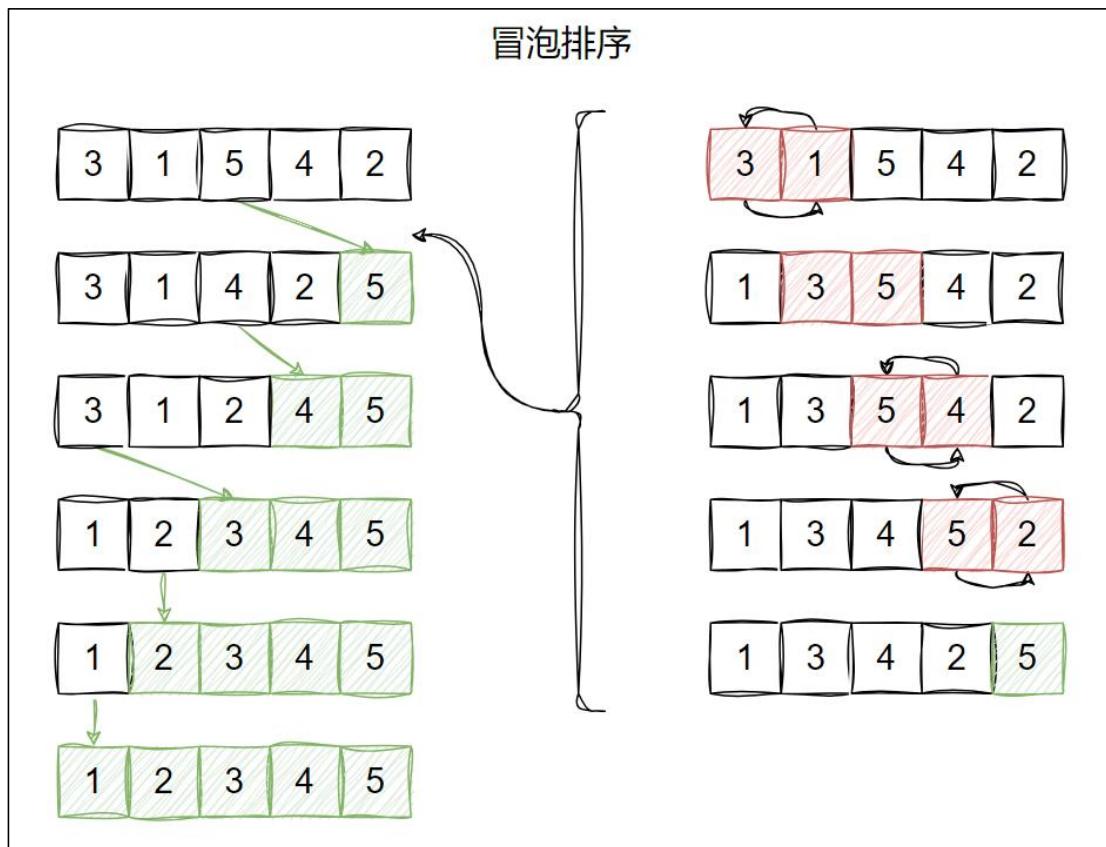
4.2 排序算法

4.2.1 冒泡排序

1) 算法原理

将待排序数组分为无序区间和有序区间两部分，无序空间在前，有序空间在后。起初，数组中的所有元素均位于无序区间。冒泡排序算法的宏观思路是，逐个将无序区间中的最大值冒出到末尾（最大值到末尾之后就相当于进入了有序区间）。直到无序区间的所有元素都冒出到有序区间。

冒出最大值的微观逻辑是两两比较并交换，从无序区间的第一个元素开始，将其与后一个相邻元素行进比较，若当前元素较大，则交换到下一个元素的位置。然后继续比较第二个元素和下一个相邻元素，直至无序区间末尾。这样一来，每经历一轮冒泡操作，都会将现有无序区间中的最大值冒出到有序区间。



2) 代码实现

```
def bubble_sort(nums):
    for i in range(len(nums) - 1):
        for j in range(len(nums) - 1 - i):
            if nums[j] > nums[j + 1]:
                nums[j], nums[j + 1] = nums[j + 1], nums[j]
```

3) 复杂度分析

(1) 时间复杂度

上述算法共执行 $(n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1 = \frac{n \times (n-1)}{2}$ 轮循环，每轮循环都执行

常量个基本指令，时间复杂度为 $O(n^2)$ 。

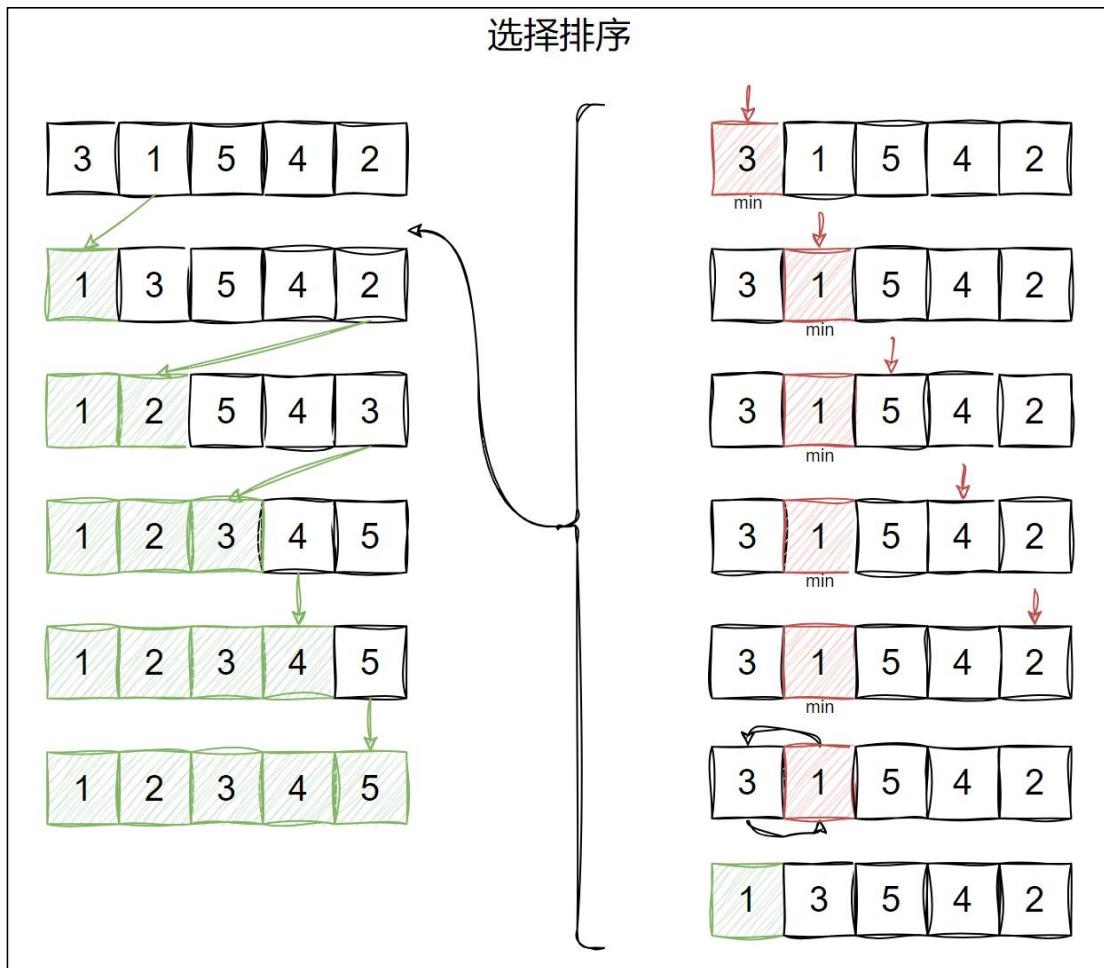
(2) 空间复杂度

就地排序，使用常数大小的额外空间，空间复杂度为 $O(1)$ 。

4.2.2 选择排序

1) 算法原理

将待排序数组分为无序区间和有序区间两部分，有序空间在前，无序空间在后。起初，所有元素均位于无序区间。选择排序算法的思路是依次在无序区间中遍历找到最小元素，然后将最小元素与无序区间中的第一位进行交换，使最小元素并入有序区间。



2) 代码实现

```
def select_sort(nums):
    for i in range(len(nums) - 1):
        min_index = i
        for j in range(i + 1, len(nums)):
            if nums[j] < nums[min_index]:
                min_index = j
        nums[i], nums[min_index] = nums[min_index], nums[i]
```

3) 复杂度分析

(1) 时间复杂度

上述算法共执行 $(n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1 = \frac{n \times (n-1)}{2}$ 轮循环，每轮循环都执行常量个基本指令，时间复杂度为 $O(n^2)$ 。

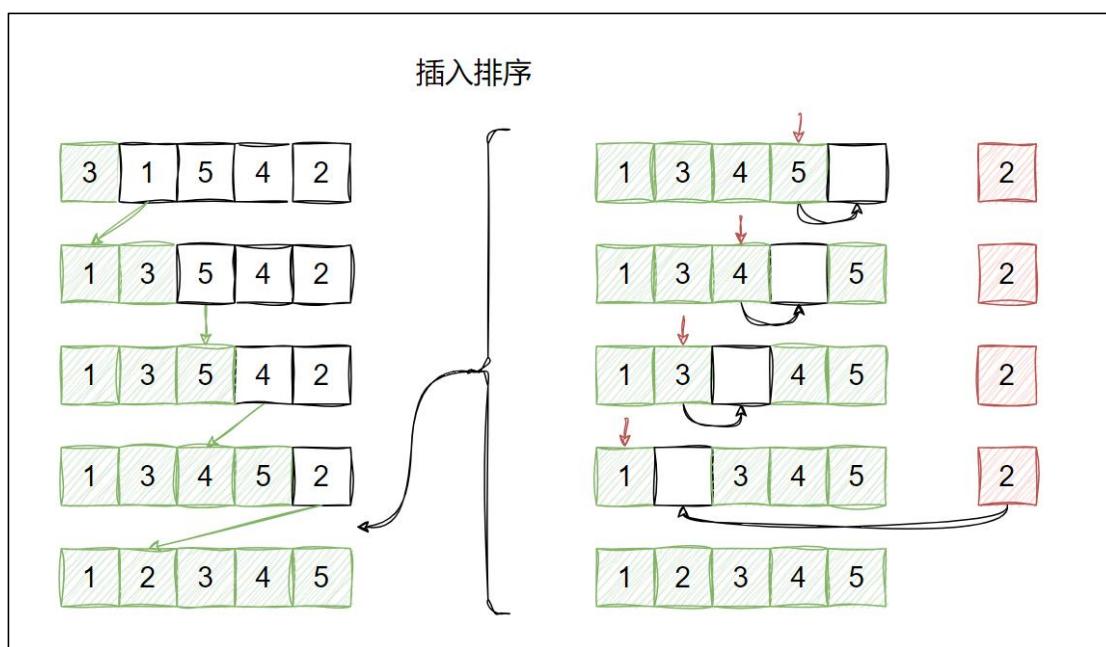
(2) 空间复杂度

就地排序，使用常数大小的额外空间，空间复杂度为 $O(1)$ 。

4.2.3 插入排序**1) 算法原理**

将待排序的数组分为无序区间和有序区间两部分，有序空间在前，无序空间在后。起初，可以认为第一个元素位于有序区间（只有一个元素，一定是有序的），后边所有元素位于无序区间。插入排序的宏观思路是依次从无序区间选择一个元素，插入到有序区间的正确位置，直到无序区间的所有元素都被插入到有序区间。

插入操作的微观逻辑是，选择无序区间的第一个元素作为待插入元素，将其保存到临时变量，然后从有序区间的最后一个元素开始比较，若大于待插入元素，则将其向后移动一位，然后继续和前一位进行比较，直到找到正确的位置，将元素插入即可。

**2) 代码实现**

```
def insert_sort(nums):
    for i in range(1, len(nums)):
        for j in range(i, 0, -1):
            if nums[j] >= nums[j - 1]:
                break
            nums[j], nums[j - 1] = nums[j - 1], nums[j]
```

3) 复杂度分析

(1) 时间复杂度

上述算法共执行 $(n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1 = \frac{n \times (n-1)}{2}$ 轮循环，每轮循环都执行常量个基本指令，时间复杂度为 $O(n^2)$ 。

(2) 空间复杂度

就地排序，使用常数大小的额外空间，空间复杂度为 $O(1)$ 。

4.2.4 归并排序

1) 算法原理

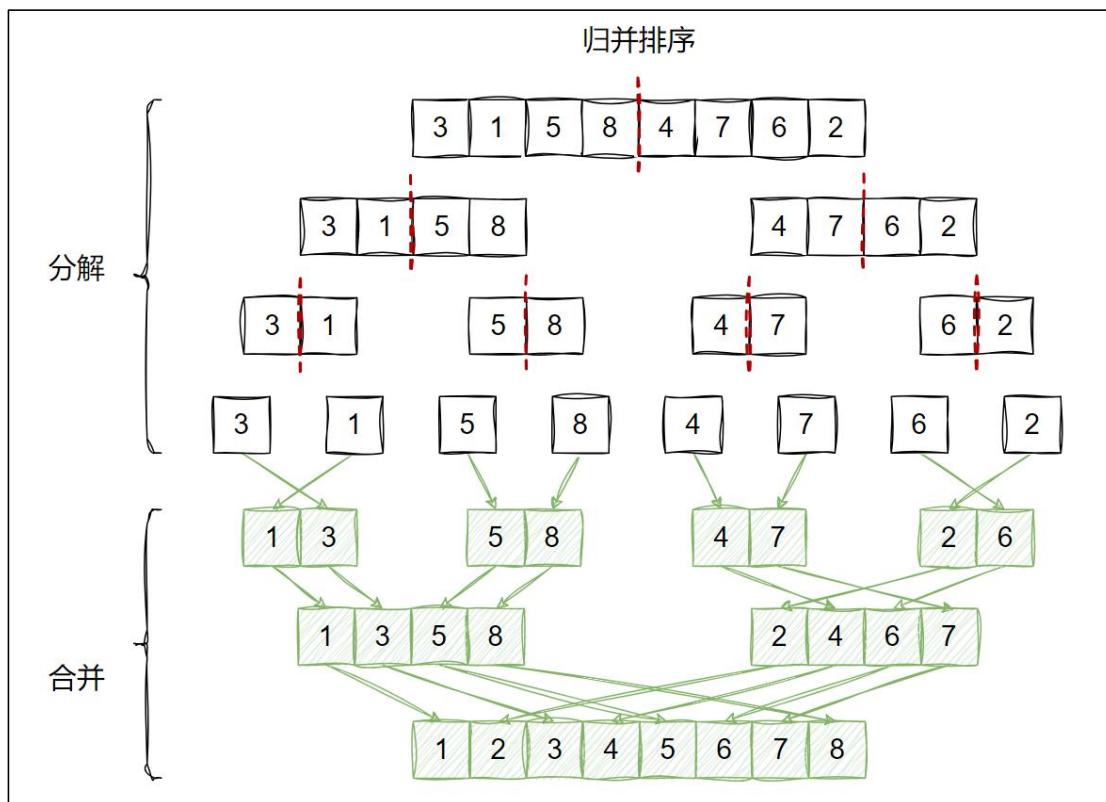
归并排序（Merge Sort）算法的核心是归并操作（Merge）。归并操作指的是将两个已经排好序的列表合并成一个有序列表的操作。

归并操作的核心逻辑十分简单：循环选取两个有序列表各自最小的元素（因为列表本身有序，所以直接取排在最前的元素即可）进行比较，每次都将两者中最小的一个移到临时数组，直到其中一个有序列表被拿空，然后将另一列表中的剩余元素，顺次放入临时数组即可。这样就能将两个有序列表合并为一个有序列表了。

归并排序算法的执行过程分为两个阶段：分解、合并。

分解阶段：不断将数组的排序问题分解为对两个子数组的排序问题，直到子数组中只包含一个元素。

合并阶段：从最小的有序数组（只包含一个数组）开始，逐层进行归并（merge）操作，将两个有序小数组合并为一个有序大数组。



2) 代码实现

```

def merge(left, right):
    """合并两个已排序的数组"""
    merged = []
    i = j = 0
    # 比较两个子数组的元素，按升序放入 merged 数组
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1
    # 将数组中剩余元素加入 merged
    merged.extend(left[i:])
    merged.extend(right[j:])
    return merged

def merge_sort(arr):
    """归并排序"""
    # 数组长度为 1 时，不再分割
    if len(arr) <= 1:

```

```

    return arr
# 分割数组
mid = len(arr) // 2
left = merge_sort(arr[:mid])
right = merge_sort(arr[mid:])
# 合并已排序的子数组
return merge(left, right)

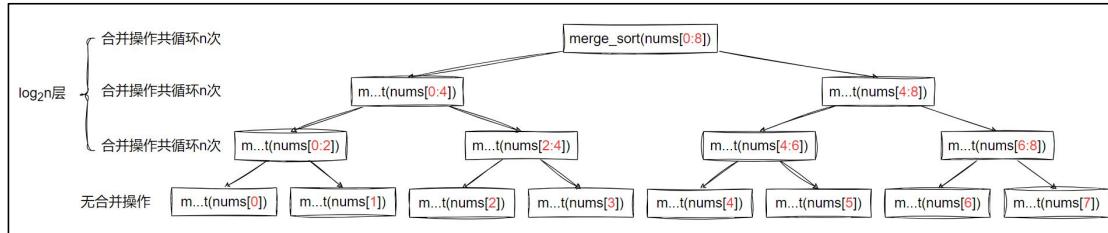
```

3) 复杂度分析

(1) 时间复杂度

该算法的时间复杂度，主要取决于合并操作的循环次数，由于该算法用到了递归，故计算循环次数时，还需要考虑递归调用的总次数。由于每次递归调用都是将数组一分为二，故递归过程可用一个二叉树进行可视化。

例如对一个长度为 8 的数组进行排序，递归调用层级如下图所示：



虽然每次递归调用 `merge_sort()` 函数，合并操作的循环次数都可能不同，但是不难发现规律，上述递归树中，每层合并操作循环的总次数为 n 次。所以只需计算出层数，便可得到循环操作的总次数。显然，二叉树的层数 $level$ 与输入数组长度 n 的关系为 $n=2^{level}$ ，因此层数 $level=\log_2 n$ 。

所以该算法循环执行的总次数为 $n \times \log_2 n$ （每层循环次数 \times 层数），每次循环操作均执行常数个指令，时间复杂度为 $O(n \log n)$ 。

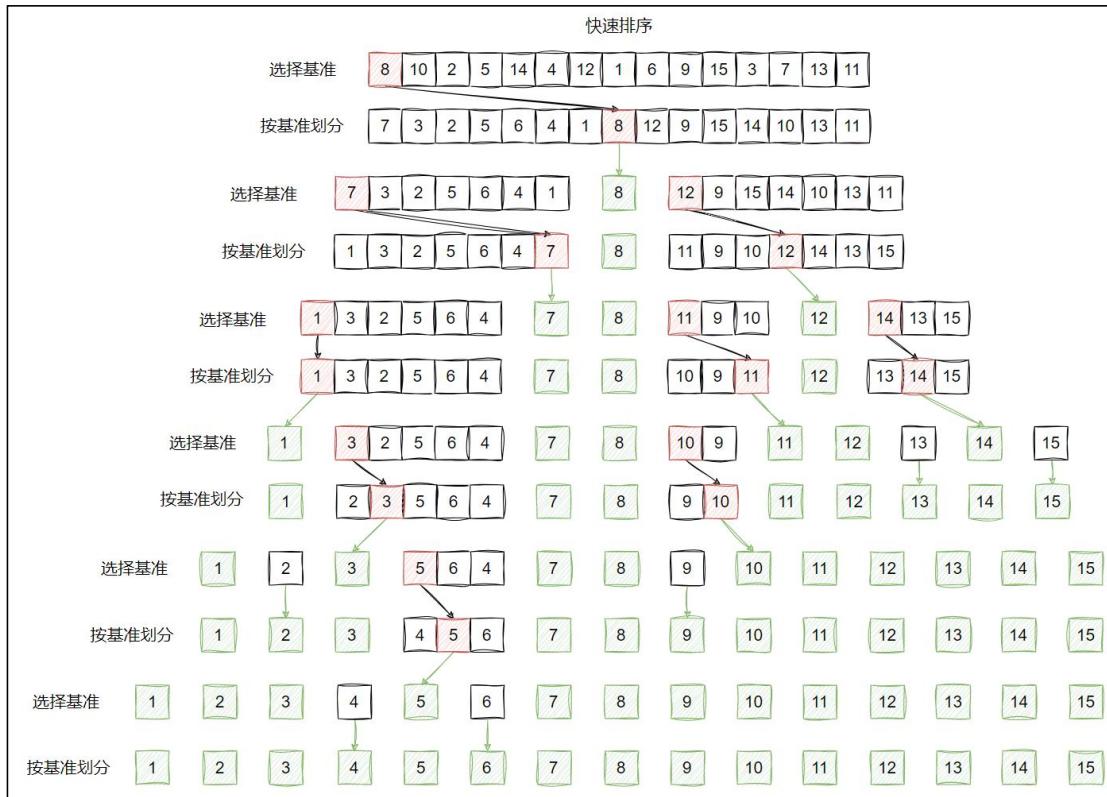
(2) 空间复杂度

由于每次合并操作都需要创建一个数组来临时存放合并结果，所以空间复杂度主要考虑临时数组占用的空间。虽然每次合并操作都会创建临时数组，但是，这些合并操作并不是同时运行的，每次合并操作结束后，临时数组的空间就会释放。也就是说，该算法在运行时，同一时刻只会有一个临时数组，所有只需考虑最大的临时数组占用的空间即可，显然临时数组的最大长度等于输入数组的长度。因此该算法的空间复杂度为 $O(n)$ 。

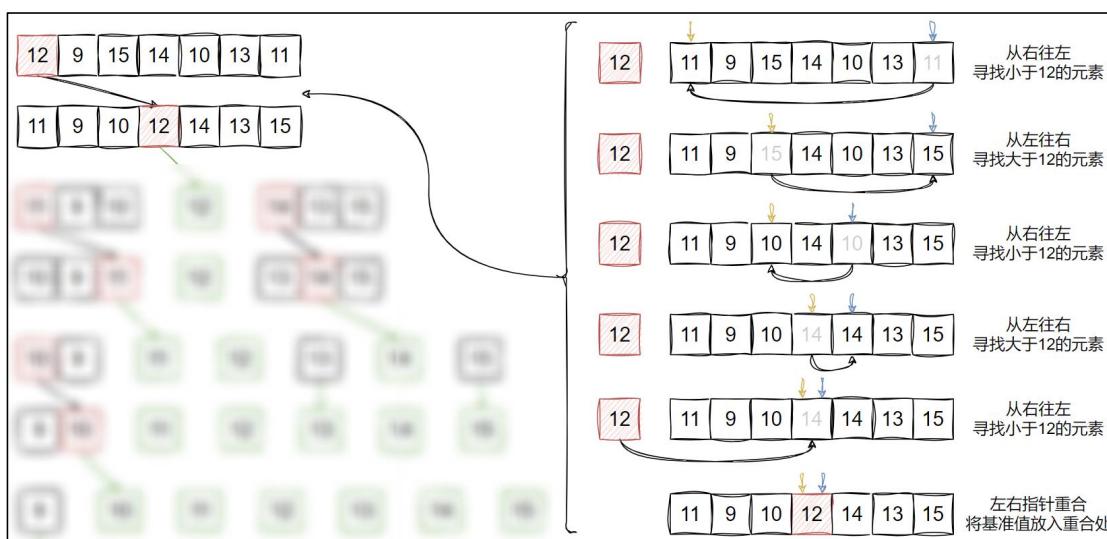
4.2.5 快速排序

1) 算法原理

快速排序算法每次从数组中挑一个元素，作为基准（pivot），然后将所有小于基准元素的元素放置于其左边，将所有大于基准元素的元素放置于其右边，完成后，就相当于完成了按基准元素的划分。同时，原来乱序的数组就也被基准点一分为二，成为两个乱序子数组。之后对两个子数组采用同样的操作划分，直到子数组只包含一个元素。



在按基准划分时，先从右到左寻找小于基准的元素，并与基准交换位置；再从左到右寻找大于基准的元素，并与基准交换位置。两者依次交替直到左右指针重合，此时将基准值放在重合处。



2) 代码实现

```

def partition(nums, left, right):
    """选择基准并按基准划分"""
    pivot = nums[left]
    while left < right:
        while left < right and nums[right] >= pivot:
            right -= 1
        nums[left] = nums[right]
        while left < right and nums[left] <= pivot:
            left += 1
        nums[right] = nums[left]
    nums[left] = pivot
    return left

def quick_sort(nums, left, right):
    """快速排序"""
    if left < right:
        mid = partition(nums, left, right)
        quick_sort(nums, left, mid - 1)
        quick_sort(nums, mid + 1, right)

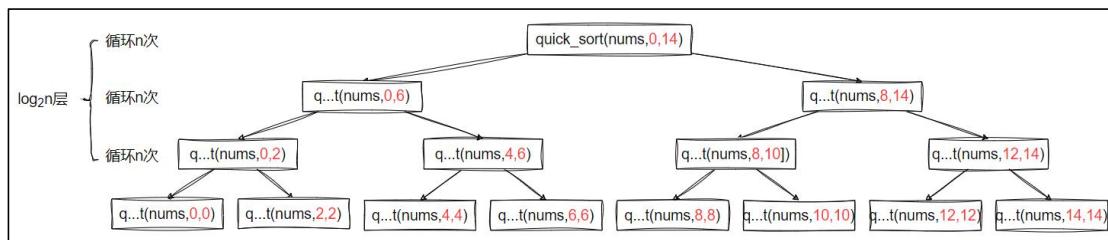
```

3) 复杂度分析

(1) 时间复杂度

由于每次递归调用时，pivot 的选择会影响到递归调用的总次数，所以该算法的时间复杂度是不固定的。下面分别分析一下最佳和最差两种情况下的时间复杂度。

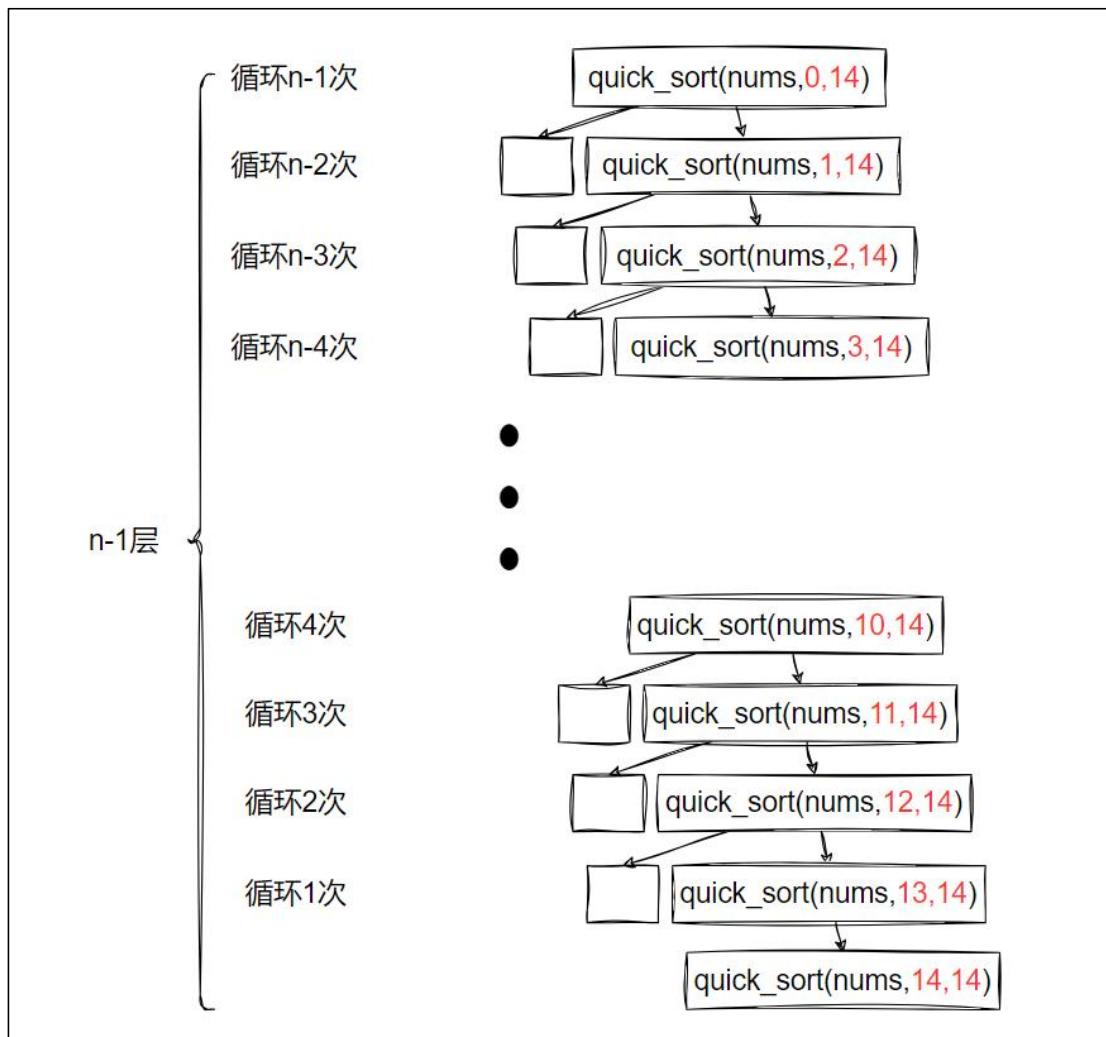
最佳情况：若每次递归调用时，选择的 pivot 恰好都是所有数据的中位数，也就是恰好能将数组均匀的一分为二。这种情况下递归调用总次数最少，时间复杂度最低。这种情况下循环总次数约为 $n \log_2 n$ ，每次循环都执行常量个基本指令，故时间复杂度为 $O(n \log n)$ 。



最差情况：若每次递归调用时，选择的 pivot 都是所有数据的最大值或者最小值，也就是将长度为 n 的数组分为长度为 0 和长度为 n-1 的两个数组。这种情况下递归调用的次数最

多，时间复杂度最高。这种情况下循环的总次数为 $(n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1 = \frac{n^2}{2}$ ，

每次循环执行常量个基本指令，故时间复杂度为 $O(n^2)$ 。



虽然快速排序算法的最差时间复杂度是 $O(n^2)$ ，但是这种情况出现的概率很低，除此之外，我们还可以通过某些手段（尽量选取接近中值的 pivot），进一步降低最差情况出现的概率，总之我们几乎可以完全避免最差情况的出现。

实际上，更为细致的分析推导与实验统计都一致地显示，在大多数情况下，快速排序算法的平均时间复杂度依然可以达到 $O(n \log n)$ 。并且由于其真实的 $T(n)$ 时间函数中，常数系数较小，所以一般情况下，其表现要优于其他算法。

(2) 空间复杂度

和时间复杂度相同，空间复杂度也不是固定的。

➤ 最佳情况：同时存在的最多的未返回的方法栈数量等于递归树的深度，每个方法栈都只保存常量个变量，所以空间复杂度为 $O(\log n)$ 。

➤ 最差情况：同时存在的最多的未返回的方法栈的数量等于 $n-1$ ，所以空间复杂度为 $O(n)$ 。

同样，大多数情况下，快速排序法的平均空间复杂度可以达到 $O(\log n)$ 。

4.2.6 堆排序

1) 算法原理

堆排序的基本思想是先将输入的数据构建成一个大顶堆，然后依次将堆顶的元素（即最大元素）移到数组的末尾，之后重新调整堆的结构，并将堆的元素个数减 1。重复这一过程，直到所有元素都被排好序。

堆排序的步骤：

① 构建大顶堆

首先将待排序的序列构建成一个大顶堆。大顶堆的特点是，堆中每个父节点的值都大于或等于其子节点的值，因此根节点是整个堆中的最大值。构建大顶堆时从最后一个非叶节点 **自底向上依次堆化**（非叶节点的计算：数组长度为 n ，最后一个非叶节点的索引为 $(n // 2) - 1$ ）。

② 交换堆顶和堆底元素

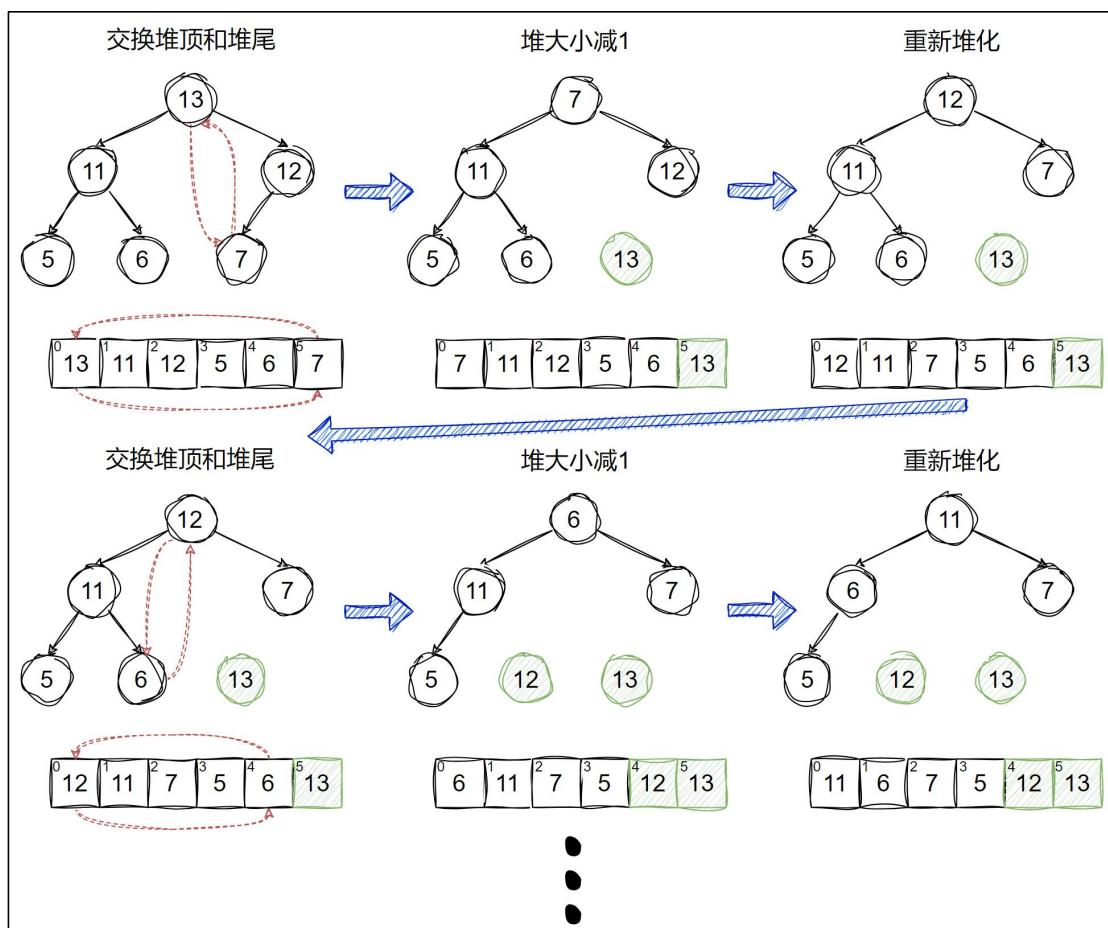
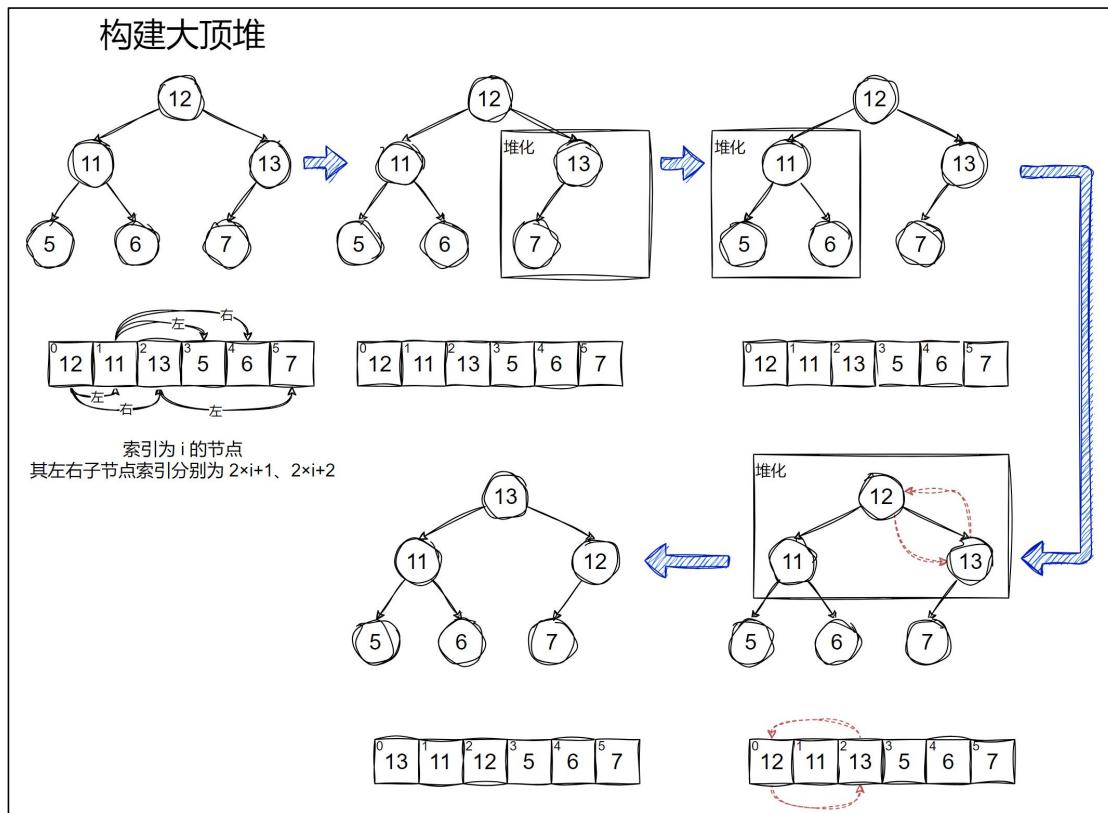
将堆顶元素（即最大元素）与当前堆的最后一个元素交换，堆的大小减 1。此时，根节点被替换为堆的最后一个元素，堆的结构被破坏，需要调整堆。

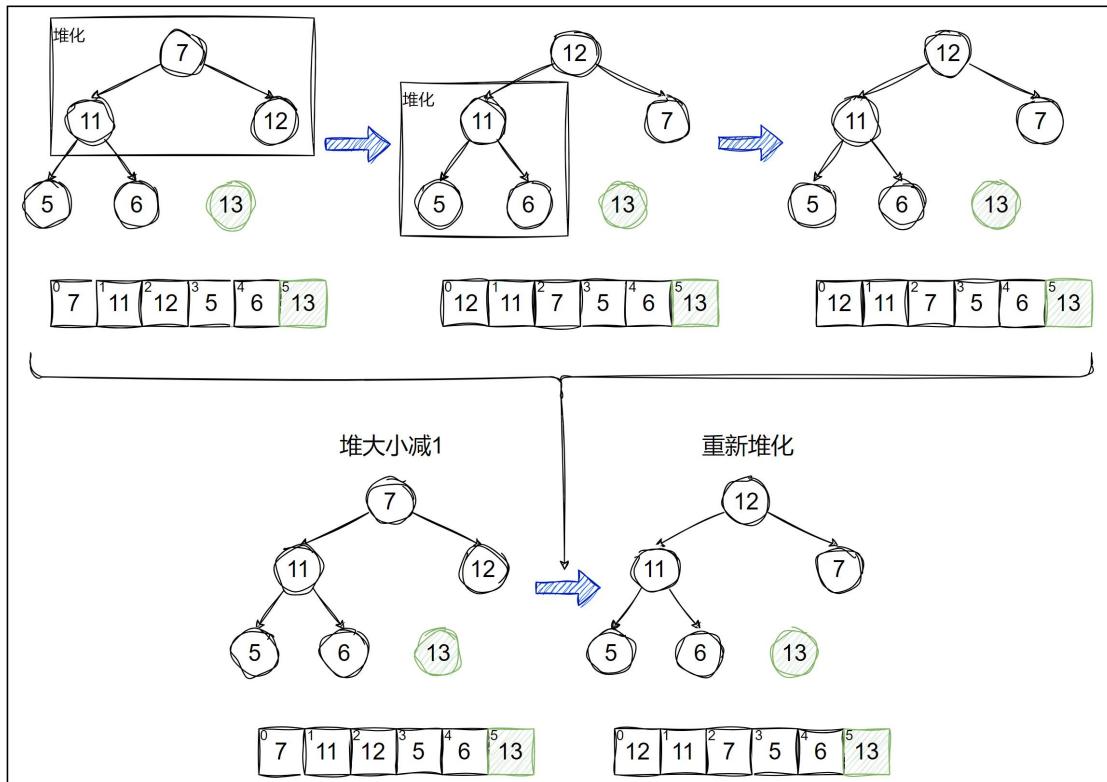
③ 重新调整堆

对堆重新进行堆化，使其满足大顶堆的性质。这样，新的堆顶元素将成为剩余元素中的最大值。**此时自顶向下堆化**。

④ 重复

重复②③，直到堆的大小为 1。





2) 代码实现

```
#arr 需要进行堆化操作的序列
#n 序列中元素的个数
#i 当前需要进行堆化操作的子树的根节点索引
def heapify(arr, n, i):
    """堆化"""
    largest = i # 最大节点指向父节点
    left = 2 * i + 1 # 左子节点
    right = 2 * i + 2 # 右子节点

    # 如果左子节点大于父节点,最大节点指向左子节点
    if left < n and arr[left] > arr[largest]:
        largest = left

    # 如果右子节点大于当前最大节点, 最大节点指向右子节点
    if right < n and arr[right] > arr[largest]:
        largest = right

    # 如果最大节点不是父节点, 则交换并递归堆化
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)
```

```
def heap_sort(arr):
    """堆排序"""
    n = len(arr)
    # 构建大顶堆
    # n//2-1 获取最后一个非叶子节点的索引
    # stop 为不包含，所以意味着循环会一直执行到索引为 0 的节点
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    # 依次将堆顶元素放在末尾，并重新堆化
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return arr

arr = [12, 11, 13, 5, 6, 7]
sorted_arr = heap_sort(arr)
print("排序后的数组:", sorted_arr)
```

3) 复杂度分析

(1) 时间复杂度

其初始构建堆时间复杂度为 $O(n)$ 。正式排序时，重建堆的时间复杂度为 $O(n \log n)$ 。所以堆排序的总体时间复杂度为 $O(n \log n)$ 。

堆排序对原始记录的排序状态不敏感，因此它无论最好、最坏和平均时间复杂度都是 $O(n \log n)$ 。但是与其他 $O(n \log n)$ 的排序算法（如归并排序、快速排序）相比，堆排序的常数因子较大，因此在某些情况下效率较低。

(2) 空间复杂度

就地排序，空间复杂度是 $O(1)$ 。

4.3 分治算法

4.3.1 概述

分治算法的基本思想为：将原问题递归的分解为若干个（通常是两个以上）规模较小、相互独立且性质相同的子问题，直到子问题足够简单，简单到可以直接求解。然后再返回结果，逐个解决上层问题。

实际上，前文提到的归并排序算法和快速排序算法都是分治思想的典型应用。

能使用分治算法解决的问题通常需要具备以下特点：

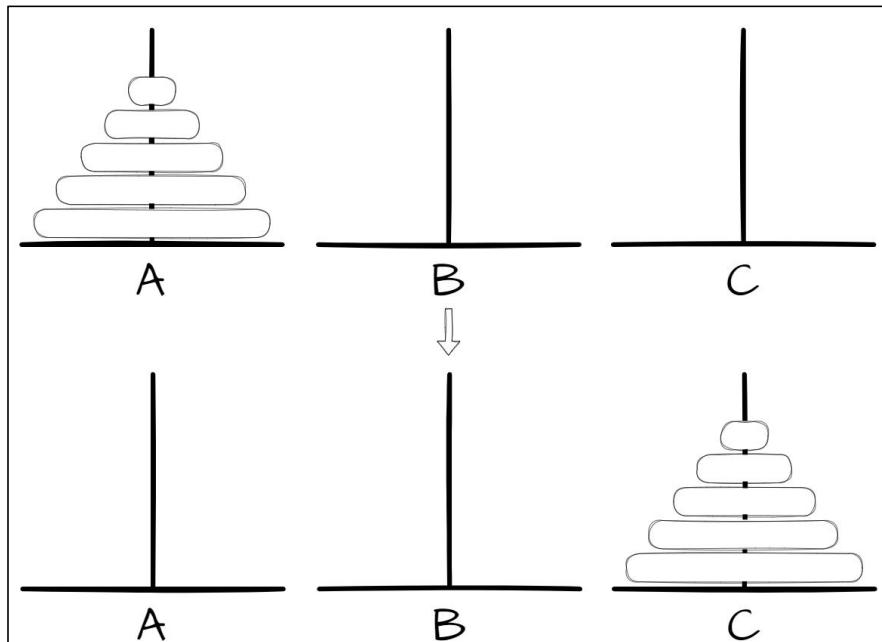
- 可分解：问题可以被划分为多个规模较小的子问题。这些子问题通常具有相同的性质，并且可以独立地解决。
- 存在基本情况：问题分解的小到一定程度后，就变得非常简单，简单到可以直接求解。
- 可合并：可以通过合并多个子问题的解，得到原问题的解。

4.3.2 案例一：汉诺塔

力扣面试题 08.06<https://leetcode.cn/problems/hanota-lcci/description/>

现有三根柱子 A、B 和 C。起始状态下，柱子 A 上套着圆盘，它们从上到下按照从小到大的顺序排列。要将所有圆盘移到柱子 C 上，并保持它们的原有顺序不变。在移动圆盘的过程中，需要遵守以下规则：

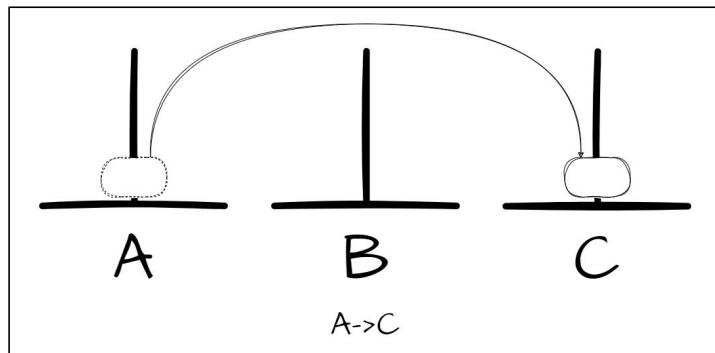
- 圆盘只能从一根柱子顶部拿出，从另一根柱子顶部放入。
- 每次只能移动一个圆盘。
- 小圆盘必须时刻位于大圆盘之上。



1) 思路分析

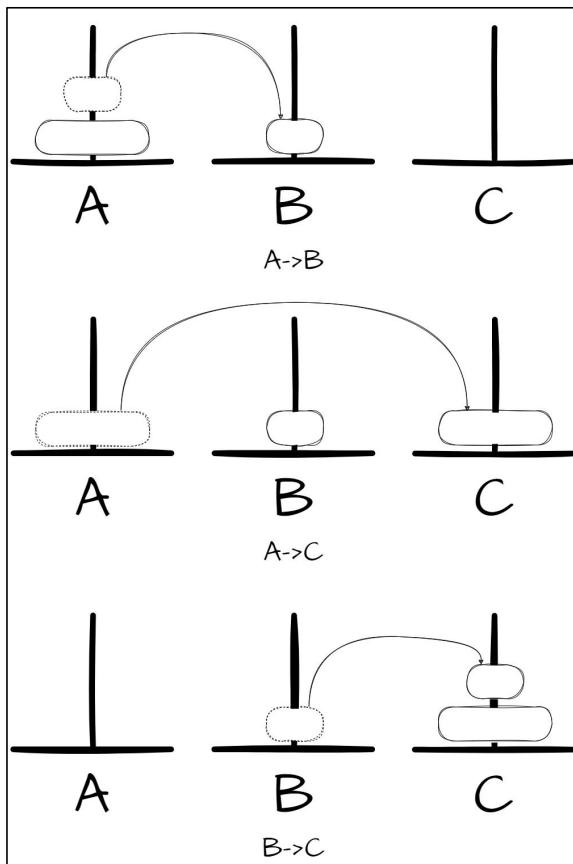
将规模为 n 的汉诺塔问题记作 $f(n)$ 。例如 $f(3)$ 代表将 3 个圆盘从 A 移动至 C。

(1) 只有 1 个圆盘



f(1), 将圆盘从 A 移动至 C。

(2) 有 2 个圆盘



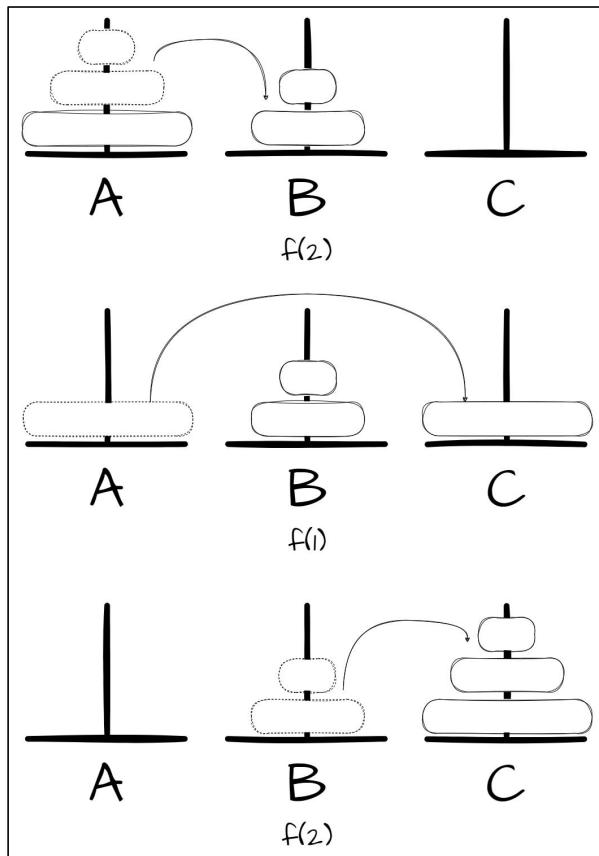
f(2), 借助 B 将两圆盘从 A 移动至 C, 分为 3 步:

使用 f(1)的方法将一个圆盘从 A 移动至 B。

使用 f(1)的方法将一个圆盘从 A 移动至 C。

使用 f(1)的方法将一个圆盘从 B 移动至 C。

(3) 有 3 个圆盘。



分为 3 步：

使用 $f(2)$ 的方法借助 C 将 2 个圆盘从 A 移动至 B。

使用 $f(1)$ 的方法将 1 个圆盘从 A 移动至 C。

使用 $f(2)$ 的方法借助 A 将 2 个圆盘从 B 移动至 C。

(4) 有 n 个圆盘

分为 3 步：

使用 $f(n-1)$ 的方法借助 C 将 $n-1$ 个圆盘从 A 移动至 B。

使用 $f(1)$ 的方法将 1 个圆盘从 A 移动至 C。

使用 $f(n-1)$ 的方法借助 A 将 $n-1$ 个圆盘从 B 移动至 C。

```
def print_abc():
    """打印 3 个柱子"""
    print("a:", a)
    print("b:", b)
    print("c:", c)
    print()

def hanota(n, source, target, buffer):
```

```
# 只有一个盘子时，直接从源柱子移动到目标柱子
if n == 1:
    target.append(source.pop())
    return

# 1. 将 n-1 个盘子从源柱子移动到缓冲柱子
hanota(n - 1, source, buffer, target)
print_abc()

# 2. 将第 n 个盘子从源柱子移动到目标柱子
hanota(1, source, target, buffer)
print_abc()

# 3. 将 n-1 个盘子从缓冲柱子移动到目标柱子
hanota(n - 1, buffer, target, source)
print_abc()

if __name__ == "__main__":
    n = 3
    a = list(range(n, 0, -1))
    b = []
    c = []
    hanota(n, a, c, b)
```

4.3.3 案例二：Karatsuba 大整数乘法算法

Karatsuba（卡拉楚巴）算法是一种高效的大整数乘法算法，关键思想是通过分治法减少了传统乘法的计算量，从而降低了乘法的时间复杂度。

现有两个大整数 A 和 B，它们的乘积 $C=A \times B$ 。在传统的朴素乘法算法中，两个 n 位数的乘积需要进行 $O(n^2)$ 次基本操作。因为每一位数字都需要与另一个数字的每一位相乘，然后再加上进位。

我们将 A、B 的高位部分和低位部分拆开，取数字长度的一半为 m ，分别表示为：

$$A=10^m \times A_1 + A_0$$

$$B=10^m \times B_1 + B_0$$

$$\text{此时可以得到: } C=A \times B=(10^m \times A_1 + A_0) \times (10^m \times B_1 + B_0)$$

$$=10^{2m} \times A_1 \times B_1 + 10^m \times (A_1 \times B_0 + A_0 \times B_1) + A_0 \times B_0$$

这个表达式由三项组成：

$A_1 \times B_1$: 高位部分的乘积。

$A_0 \times B_0$: 低位部分的乘积。

$A_1 \times B_0 + A_0 \times B_1$: 混合部分, 涉及到高位与低位的交叉乘积。

注意到, $A_1 \times B_0 + A_0 \times B_1$ 可以转换为 $(A_1 + A_0) \times (B_1 + B_0) - A_0 \times B_0 - A_1 \times B_1$, 即 $(A_1 + A_0) \times (B_1 + B_0)$ -低位部分的乘积-高位部分的乘积, 低位部分乘积和高位部分乘积可以复用, 我们只需计算 $(A_1 + A_0) \times (B_1 + B_0)$, 相较于计算 $A_1 \times B_0 + A_0 \times B_1$, 减少了 1 次乘法计算。

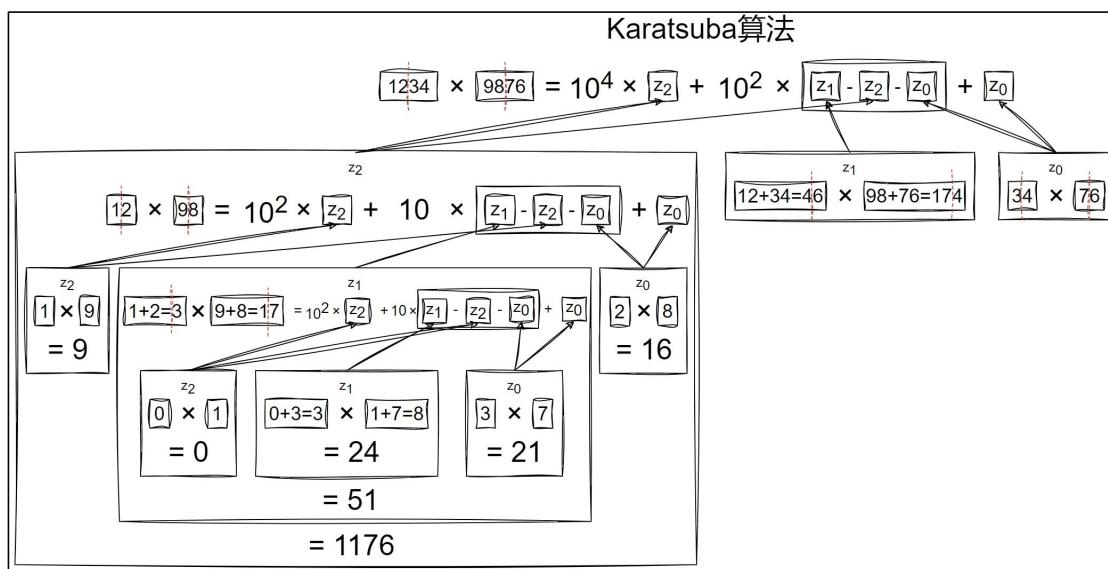
令 $z_0 = A_0 \times B_0$

令 $z_1 = (A_1 + A_0) \times (B_1 + B_0)$

令 $z_2 = A_1 \times B_1$

此时结果 $C = 10^{2m} \times z_2 + 10^m \times (z_1 - z_2 - z_0) + z_0$

这样, Karatsuba 通过减少了 1 个乘法操作, 将原本的 4 次乘法运算变成了 3 次乘法运算, 时间复杂度: $T(n)=3T(n^2)+O(n)$, $O(n^{\log 3}) \approx O(n^{1.585})$ 。



```
def karatsuba(x, y):
    """卡拉楚巴算法"""
    # 将 x 和 y 转换为字符串
    x_str, y_str = str(x), str(y)
    n = max(len(x_str), len(y_str))

    # 如果存在负数, 将其转换为正数再调用 karatsuba
    if x_str[0] == "-":
        return -karatsuba(-x, y)
    if y_str[0] == "-":
        return -karatsuba(x, -y)

    # 如果只剩 1 位则返回乘积
    if n == 1:
        return int(x) * int(y)
```

```
if n == 1:  
    return x * y  
  
# 确保数字长度一致  
x_str = x_str.zfill(n)  
y_str = y_str.zfill(n)  
  
# 计算分割点  
m = n // 2  
  
# 将数字划分为高位部分和低位部分  
high1, low1 = int(x_str[:-m]), int(x_str[-m:])  
high2, low2 = int(y_str[:-m]), int(y_str[-m:])  
  
# 递归调用 karatsuba  
z0 = karatsuba(low1, low2)  
z1 = karatsuba(low1 + high1, low2 + high2)  
z2 = karatsuba(high1, high2)  
  
return pow(10, 2 * m) * z2 + pow(10, m) * (z1 - z2 - z0) + z0
```

4.4 动态规划算法

4.4.1 概述

动态规划算法与分治法类似，也是通过将原问题拆分为若干子问题，然后递归求解子问题，最后再组合子问题的解进而得到原问题的解。不同的是，分治算法解决的问题，子问题通常相互独立。而动态规划解决的问题，子问题具有重叠现象，所谓的重叠现象，是指不同的子问题会有相同的子子问题。

对于这些重复的子问题，若不加干预，会进行多次的重复计算，效率低下。动态规划意义就是，能够保证每个重复的子问题只计算一次，其解决问题的思路就是将计算过的子问题保存起来，后续计算相同子问题时，便可直接获取结果。

动态规划算法的实现方式分为自上而下的记忆化递归和自下而上的迭代。自上而下的递归方式只需考虑原问题到子问题的递归公式，而无序考虑子问题的执行顺序；而自下而上的迭代方式，除了需要考虑递推关系，还需要确保子问题从小到大依次执行。

4.4.2 案例一：爬楼梯

力扣 70 题 <https://leetcode.cn/problems/climbing-stairs/description/>

爬有 n 个台阶的楼梯，每次可以爬 1 或 2 个台阶。有多少种不同的方法可以爬到楼顶？

更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

1) 方法一

由于每次只能上一个台阶，所以第 n 个台阶可能是从第 n-1 个台阶爬 1 阶上来的，也可能从第 n-2 个台阶爬 2 阶上来的，所以爬到第 n 阶的方法数就等于爬到第 n-1 阶的方法数加上爬到第 n-2 阶的方法数。故可以得到状态转移方程： $f(n) = f(n-1) + f(n-2)$ 。

```
def climb(n):
    if n == 1:
        return 1
    elif n == 2:
        return 2
    else:
        return climb(n - 1) + climb(n - 2)
```

2) 方法二

既然 $f(n) = f(n-1) + f(n-2)$ ，那么我们也可以自下而上从 1 阶台阶开始逐渐增加，并记录当前阶方法数和当前阶-1 的方法数。

```
def climb(n):
    pre = 1
    cur = 1
    for _ in range(1, n):
        pre, cur = cur, pre + cur
    return cur
```

4.4.3 案例二：最大的连续子数组之和

力扣 53 题 <https://leetcode.cn/problems/maximum-subarray/description/>

找出整数数组 nums 中数组之和最大的连续子数组（子数组最少包含一个元素），返回其最大和。

示例：

- 输入： nums = [-2,1,-3,4,-1,2,1,-5,4]
- 输出： 6

解释：连续子数组 [4,-1,2,1] 的和最大，为 6。

1) 思路分析

用 $f(x)$ 表示以 x 结尾的最大子数组之和，考虑处于位置 i 时，有两种选择：

- 与之前的子数组组成连续子数组，此时子数组之和 $f(i) = f(i-1) + \text{nums}[i]$ 。
- 中断连续，从头开始一个新的连续子数组，此时 $f(i) = \text{nums}[i]$ 。

得到状态转移方程： $f(i) = \max(f(i-1) + \text{nums}[i], \text{nums}[i])$ 。

持续组成连续子数组，除非连续子数组之和已经<0，此时中断连续。

```
def max_subarray(nums):
    result, f = nums[0], 0
    for i in nums:
        # 连续子数组之和若小于 0，则中断连续
        if f < 0:
            f = 0
        # 累加连续子数组之和
        f += i
        # 更新最大值
        if result < f:
            result = f
    return result
```

4.4.4 案例三：0-1 背包

0-1 背包问题是一个经典的动态规划问题。其基本描述是：给定一组物品，每个物品都有一个重量和一个价值，在背包容量有限的情况下，如何选择物品放入背包，使得背包中物品的总价值最大化，且总重量不超过背包的容量。

- 物品：有 n 个物品，每个物品 i 的重量为 $weight[i]$ 和价值 $value[i]$ 。
- 背包容量：背包可以承载的最大重量为 W 。
- 目标：选择若干物品放入背包，使得总重量不超过 W ，且总价值最大。

| 物品 <i>i</i> | 重量 <i>weight</i> | 价值 <i>value</i> |
|-------------|------------------|-----------------|
| 0 | 1 | 3 |
| 1 | 2 | 2 |
| 2 | 3 | 6 |

背包容量 5

1) 方法一

(1) 定义状态

定义一个二维数组 $dp[i][j]$ ，表示前 i 个物品中，总重量不超过 j 的情况下，能够取得的最大价值。

- i : 表示考虑第 i 个物品。
- j : 表示背包当前容量为 j 。

(2) 状态转移

对于每个物品 i , 有两个选择:

- 不选第 i 个物品: 此时最大价值就是前 $i-1$ 个物品在容量 j 下的最大价值, 即 $dp[i-1][j]$ 。
- 选第 i 个物品: 此时背包剩余容量为 $j-weight_i$, 所以最大价值为 $value_i + dp[i-1][j-weight_i]$, 前提是 $j \geq w_i$ 。

状态转移方程: $dp[i][j] = \max(dp[i-1][j], value_i + dp[i-1][j-weight_i])$

最大价值会存储在 $dp[n][W]$ 中, 其中 n 是物品的数量, W 是背包的最大容量。

(3) 实现步骤

- 初始化二维数组 dp , i 行、 $W+1$ 列, 因为要存放背包容量为 $0 \sim W$ 时的状态。
- 循环中每次增加一个可选物品。
- 每增加一个可选物品后遍历背包容量为 $0 \sim W$ 。考虑该物品是否放得下背包, 如果放得下, 将放进去后和不放进去进行比较。

| 物品 <i>i</i> | | 重量weight | 价值value | | | |
|-------------|----------|----------|---------|---|---|--|
| 0 | | 1 | 3 | | | |
| <i>i</i> | <i>j</i> | 0 | 1 | 2 | 3 | |
| 0 | 0 | 3 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | |

第一轮: 只有物品0

| 物品 <i>i</i> | | 重量weight | 价值value | | | |
|-------------|----------|----------|---------|---|---|--|
| 0 | | 1 | 3 | | | |
| <i>i</i> | <i>j</i> | 0 | 1 | 2 | 3 | |
| 0 | 0 | 3 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | |

第二轮: 物品0, 物品1

能否放得下物品1?
如果放得下, 背包中让出物品1的重量后对应的上轮状态是哪个?
 $\rightarrow dp[i-1][j-weight_i]$
该状态的价值加上物品1的价值, 与上轮当前重量不放物品1的状态的价值, 哪个更大?
 $\rightarrow dp[i][j] = \max(dp[i-1][j], value_i + dp[i-1][j-weight_i])$

放不下物品1
直接使用上一轮相同 j 的状态 放得下物品1
比较之后使用上一轮相同 j 的状态 放得下物品1
比较之后使用上一轮 $j-weight_i$ 的状态+物品1

| | | | | | | |
|----------|----------|---|---|---|---|--|
| <i>i</i> | <i>j</i> | 0 | 1 | 2 | 3 | |
| 0 | 0 | 3 | 3 | 3 | 3 | |
| 1 | 0 | 3 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | |

| | | | | | | |
|----------|----------|---|---|---|---|--|
| <i>i</i> | <i>j</i> | 0 | 1 | 2 | 3 | |
| 0 | 0 | 3 | 3 | 3 | 3 | |
| 1 | 0 | 3 | 3 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | |

| | | | | | | |
|----------|----------|---|---|---|---|--|
| <i>i</i> | <i>j</i> | 0 | 1 | 2 | 3 | |
| 0 | 0 | 3 | 3 | 3 | 3 | |
| 1 | 0 | 3 | 3 | 5 | 5 | |
| | 0 | 0 | 0 | 0 | 0 | |

| 物品i | 重量weight | 价值value |
|-----|----------|---------|
| 0 | 1 | 3 |
| 1 | 2 | 2 |
| 2 | 3 | 6 |

第三轮：物品0，物品1，物品2

能否放得下物品2？

如果放得下，背包中让出物品2的重量后对应的上轮状态是哪个？

→ $dp[i-1][j - weight_2]$

该状态的价值加上物品2的价值，与上轮当前重量不放物品2的状态的价值，哪个更大？

→ $dp[i][j] = \max(dp[i-1][j], value_2 + dp[i-1][j - weight_2])$

放不下物品2
直接使用上一轮相同 j 的状态

| i\j | 0 | 1 | 2 | 3 |
|-----|---|---|---|---|
| 0 | 0 | 3 | 3 | 3 |
| 1 | 0 | 3 | 3 | 5 |
| 2 | 0 | 3 | 0 | 0 |

放不下物品2
直接使用上一轮相同 j 的状态

| i\j | 0 | 1 | 2 | 3 |
|-----|---|---|---|---|
| 0 | 0 | 3 | 3 | 3 |
| 1 | 0 | 3 | 3 | 5 |
| 2 | 0 | 3 | 3 | 0 |

放得下物品2
比较之后使用上一轮 $j - weight_2$ 的状态+物品2

| i\j | 0 | 1 | 2 | 3 |
|-----|---|---|---|---|
| 0 | 0 | 3 | 3 | 3 |
| 1 | 0 | 3 | 3 | 5 |
| 2 | 0 | 3 | 3 | 6 |

```

def knapsack(weights, values, W):
    n = len(weights)
    # 初始化二维数组 dp, dp[i][j] 表示前 i 个物品中, 背包容量为 j 时的最大价值
    dp = [[0] * (W + 1) for _ in range(n)]

    # 每次增加一个可选物品, 增加物品后遍历一次背包重量
    for i in range(n):
        for j in range(1, W + 1):
            # 如果当前物品放的进背包, 进行比较
            if weights[i] <= j:
                dp[i][j] = max(dp[i - 1][j], values[i] + dp[i - 1][j - weights[i]])
            # 如果当前物品放不进背包, 使用上轮相同 j 的状态
            else:
                dp[i][j] = dp[i - 1][j]

            print(f"前{i + 1}个物品, 背包容量为{j}时")
            for row in range(len(dp)):
                print(dp[row])

    return dp[n - 1][W]

if __name__ == "__main__":
    weights = [1, 2, 3] # 物品的重量
    values = [3, 2, 6] # 物品的价值
    W = 3 # 背包的最大容量
    print(knapsack(weights, values, W))

```

2) 方法二

可以看到在方法一中，我们每轮遍历时只用到上 1 轮的结果，与这一轮的结果无关，因此我们只用一维数组存储上 1 轮的结果即可。

并且遍历时需要从后往前遍历，防止这轮结果将还需要用到的上轮结果覆盖。

并且不用全部遍历，遍历到背包当前重量刚好大于当前物品重量即可。

```
def knapsack(weights, values, W):
    n = len(weights)
    dp = [0] * (W + 1)
    for i in range(n):
        for j in range(W, weights[i] - 1, -1): # 从后往前遍历
            dp[j] = max(dp[j], values[i] + dp[j - weights[i]])
    return dp[W]

if __name__ == "__main__":
    weights = [1, 2, 3] # 物品的重量
    values = [3, 2, 6] # 物品的价值
    W = 3 # 背包的最大容量
    print(knapsack(weights, values, W))
```

4.4.5 案例四：完全背包

完全背包问题是 0-1 背包问题的一种扩展。与 0-1 背包不同，完全背包问题允许每个物品可以被选取多次，也就是说，物品的数量没有限制。

1) 方法一

(1) 定义状态

定义一个二维数组 $dp[i][j]$ ，表示前 i 个物品中，总重量不超过 j 的情况下，能够取得的最大价值。

- i : 表示考虑第 i 个物品。
- j : 表示背包当前容量为 j 。

(2) 状态转移

相较于 0-1 背包，仅有选择放入第 i 个物品时发生了变化。

对于每个物品 i ，有两个选择：

- 不选第 i 个物品：与 0-1 背包相同，此时最大价值就是前 $i-1$ 个物品在容量 j 下的最大价值，即 $dp[i-1][j]$ 。

➤ 选第 i 个物品：此时背包剩余容量为 $j - weight_i$ ，所以最大价值为 $value_i + dp[i][j - weight_i]$ 。

状态转移方程： $dp[i][j] = \max(dp[i-1][j], value_i + dp[i][j - weight_i])$

状态转移方程相较于 0-1 背包问题仅有一处 $i-1$ 变为了 i 。

(3) 实现步骤

- 初始化二维数组 dp ， i 行、 $W+1$ 列，因为要存放背包容量为 $0 \sim W$ 的状态。
- 循环中每次增加一个可选物品。
- 每增加一个可选物品后遍历背包容量为 $0 \sim W$ 。考虑该物品是否放得下背包，如果放得下，将放进去后和不放进去进行比较。

| 物品 <i>i</i> | 重量weight | 价值value | | | | |
|------------------|----------|---------|---|---|--|--|
| 0 | 1 | 3 | | | | |
| 第一轮：只有物品0 | | | | | | |
| $i \backslash j$ | 0 | 1 | 2 | 3 | | |
| 0 | 0 | 3 | 0 | 0 | | |
| | 0 | 0 | 0 | 0 | | |
| | 0 | 0 | 0 | 0 | | |

| | | | | | | |
|------------------|---|---|---|---|--|--|
| $i \backslash j$ | 0 | 1 | 2 | 3 | | |
| 0 | 0 | 3 | 0 | 0 | | |
| | 0 | 0 | 0 | 0 | | |
| | 0 | 0 | 0 | 0 | | |

| | | | | | | |
|------------------|---|---|---|---|--|--|
| $i \backslash j$ | 0 | 1 | 2 | 3 | | |
| 0 | 0 | 3 | 6 | 0 | | |
| | 0 | 0 | 0 | 0 | | |
| | 0 | 0 | 0 | 0 | | |

| 物品 <i>i</i> | 重量weight | 价值value | | | | | |
|---|----------|---------|-----------------------------------|---|--|-----------------------------------|--|
| 0 | 1 | 3 | | | | | |
| 1 | 2 | 7 | | | | | |
| 第二轮：物品0, 物品1 | | | | | | | |
| 能否放得下物品1？ | | | | | | | |
| 如果放得下，背包中让出物品1的重量后对应的本轮状态是哪个？ ——> $dp[i][j - weight_1]$ | | | | | | | |
| 该状态的价值加上物品1的价值，与上轮当前重量不放物品1的状态的价值，哪个更大? ——> $dp[i][j] = \max(dp[i-1][j], value_1 + dp[i][j - weight_1])$ | | | | | | | |
| 放下物品1 | | | 放得下物品1 | | | 放得下物品1 | |
| 直接使用上一轮相同 j 的状态 | | | 比较之后使用使用本轮 $j - weight_1$ 的状态+物品1 | | | 比较之后使用使用本轮 $j - weight_1$ 的状态+物品1 | |
| $i \backslash j$ | 0 | 1 | 2 | 3 | | | |
| 0 | 0 | 3 | 6 | 9 | | | |
| 1 | 0 | 3 | 0 | 0 | | | |
| | 0 | 0 | 0 | 0 | | | |

| | | | | | | |
|------------------|---|---|---|---|--|--|
| $i \backslash j$ | 0 | 1 | 2 | 3 | | |
| 0 | 0 | 3 | 6 | 9 | | |
| 1 | 0 | 3 | 7 | 0 | | |
| | 0 | 0 | 0 | 0 | | |

| | | | | | | |
|------------------|---|---|---|----|--|--|
| $i \backslash j$ | 0 | 1 | 2 | 3 | | |
| 0 | 0 | 3 | 6 | 9 | | |
| 1 | 0 | 3 | 7 | 10 | | |
| | 0 | 0 | 0 | 0 | | |

| 物品i | 重量weight | 价值value |
|-----|----------|---------|
| 0 | 1 | 3 |
| 1 | 2 | 7 |
| 2 | 3 | 11 |

第三轮：物品0，物品1，物品2

能否放得下物品2？

如果放得下，背包中让出物品2的重量后对应的本轮状态是哪个？

→ $dp[i][j-weight_i]$

该状态的价值加上物品2的价值，与上轮当前重量不放物品2的状态的价值，哪个更大？

→ $dp[i][j] = \max(dp[i-1][j], value_i + dp[i][j-weight_i])$

放不下物品2
直接使用上一轮相同 j 的状态

放不下物品2
直接使用上一轮相同 j 的状态

放得下物品2
比较之后使用本轮 $j-weight_i$ 的状态+物品2

| i\j | 0 | 1 | 2 | 3 |
|-----|---|---|---|----|
| 0 | 0 | 3 | 3 | 3 |
| 1 | 0 | 3 | 7 | 10 |
| 2 | 0 | 3 | 0 | 0 |

| i\j | 0 | 1 | 2 | 3 |
|-----|---|---|---|----|
| 0 | 0 | 3 | 3 | 3 |
| 1 | 0 | 3 | 7 | 10 |
| 2 | 0 | 3 | 7 | 0 |

| i\j | 0 | 1 | 2 | 3 |
|-----|---|---|---|----|
| 0 | 0 | 3 | 3 | 3 |
| 1 | 0 | 3 | 7 | 10 |
| 2 | 0 | 3 | 7 | 11 |

```

def knapsack(weights, values, W):
    n = len(weights)
    # 初始化二维数组 dp, dp[i][j] 表示前 i 个物品中, 背包容量为 j 时的最大价值
    dp = [[0] * (W + 1) for _ in range(n)]

    # 每次增加一个可选物品, 增加物品后遍历一次背包重量
    for i in range(n):
        for j in range(1, W + 1):
            # 如果当前物品放的进背包, 进行比较
            if weights[i] <= j:
                dp[i][j] = max(dp[i - 1][j], values[i] + dp[i][j - weights[i]])
            # 如果当前物品放不进背包, 使用上轮相同 j 的状态
            else:
                dp[i][j] = dp[i - 1][j]

        print(f"前{i + 1}个物品, 背包容量为{j}时")
        for row in range(len(dp)):
            print(dp[row])

    return dp[n - 1][W]

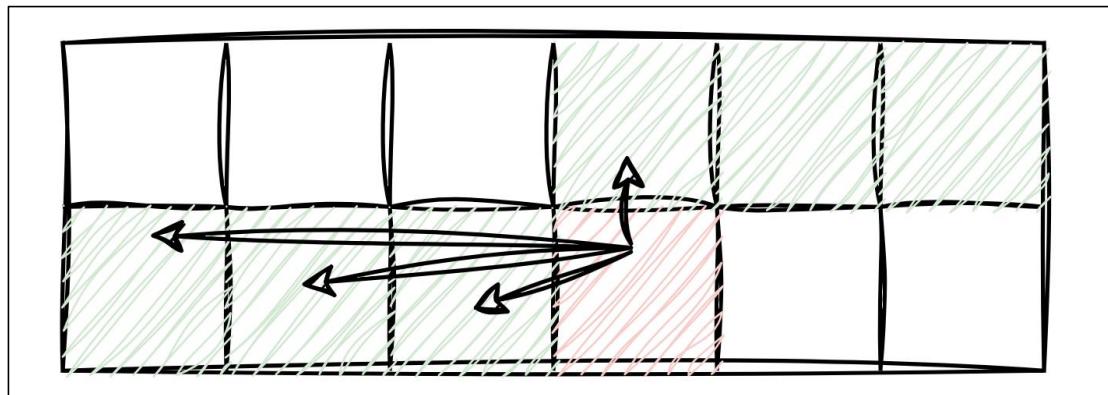
if __name__ == "__main__":
    weights = [1, 2, 3] # 物品的重量
    values = [3, 7, 11] # 物品的价值
    W = 3 # 背包的最大容量
    print(knapsack(weights, values, W))

```

2) 方法二

方法一同样可以优化为一维数组。可以看到在方法一中我们既使用了上 1 轮结果，也使用了本轮 i 之前的结果，但是两者上下没有重叠。

这时我们遍历时需要从前向后遍历，因为可能会用到本轮 i 之前的结果。



```
def knapsack(weights, values, W):
    n = len(weights)
    dp = [0] * (W + 1)

    for i in range(n):
        for j in range(weights[i], W + 1):
            dp[j] = max(dp[j], dp[j - weights[i]] + values[i])

    return dp[W]

if __name__ == "__main__":
    weights = [1, 2, 3] # 物品的重量
    values = [3, 7, 11] # 物品的价值
    W = 3 # 背包的最大容量
    print(knapsack(weights, values, W))
```

4.5 回溯算法

4.5.1 概述

回溯法是一种通过探索所有可能的解来解决问题的算法。回溯法采用试错的思想，它尝试分步的去解决一个问题。在分步解决问题的过程中，当它通过尝试发现，现有的分步答案不能得到有效的正确的解答的时候，它将取消上一步甚至是上几步的计算，再通过其它的可能的分步解答再次尝试寻找问题的答案。回溯法通常用递归方法来实现。

回溯算法的基本步骤可以总结为：

- 选择：在每个决策点选择一个候选解。

- 探索：递归地继续在下一个决策点上做选择。
- 验证：在每次选择之后，检查当前路径是否满足条件。
- 回溯：当某条路径不满足条件或无法继续时，回到上一步，尝试其他可能的选择。

4.5.2 案例一：全排列

力扣 46 题 <https://leetcode.cn/problems/permutations/>

现有一个不含重复数字的数组 `nums`，返回其所有可能的全排列。

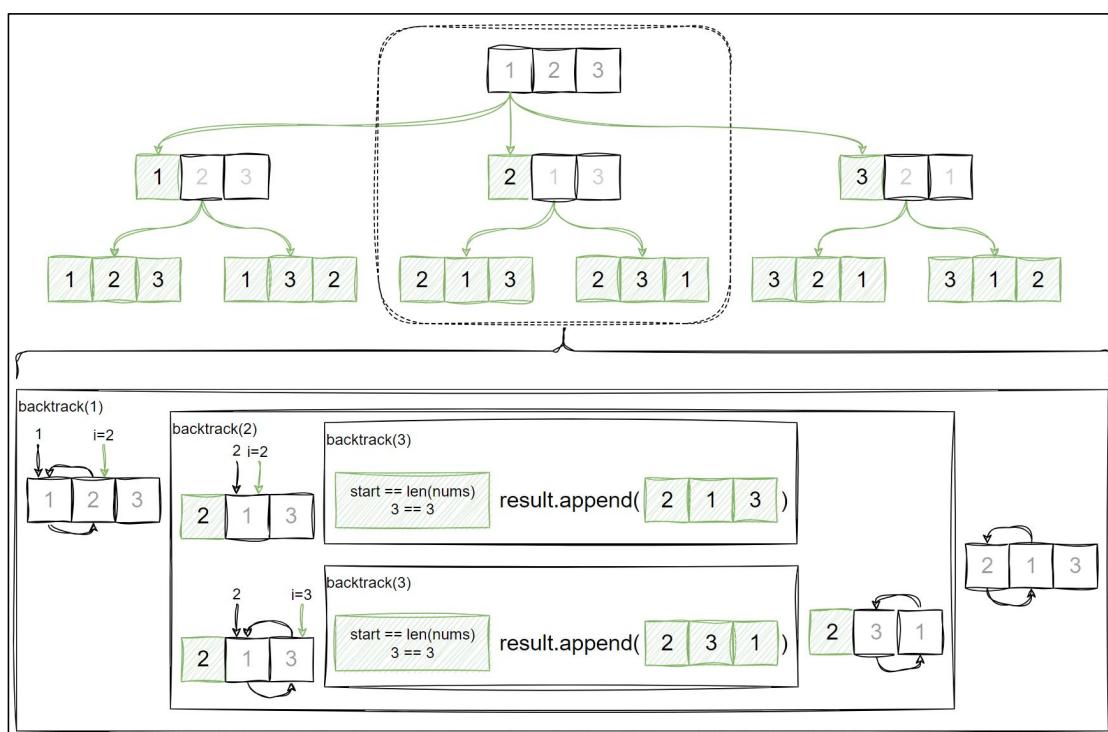
示例：

- 输入：`nums = [1,2,3]`
- 输出：`[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]`

1) 思路分析

实现步骤：

- 选择：从待排列的元素中选择一个元素作为当前位置的元素。
- 递归：将该元素固定在当前位置后，递归排列下一个元素。
- 终止条件：所有位置的元素都已确定，添加到结果中并终止递归。
- 回溯：回到上一步，撤销当前的选择，尝试下一个可能的元素。



```

def backtrack(start):
    # 到达末尾，将此时排列结果添加到最终结果中
    if start == len(nums):
        result.append(nums[:])
        return

    for i in range(start, len(nums)):
        # 选取当前位置的元素：将要选取的元素与此位置的元素互换
        if start != i:
            nums[start], nums[i] = nums[i], nums[start]
            # 递归处理下一个位置的元素
            backtrack(start + 1)
            # 回溯，恢复原始数组
            if start != i:
                nums[start], nums[i] = nums[i], nums[start]

backtrack(0)
return result

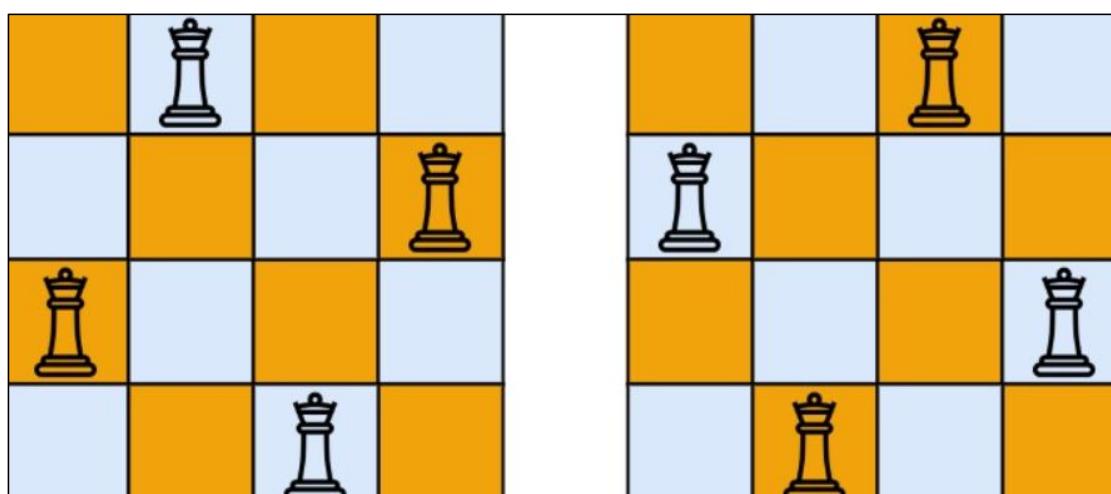
```

4.5.3 案例二：N 皇后

力扣 51 题 <https://leetcode.cn/problems/n-queens/>

按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。

n 皇后问题研究的是如何将 n 个皇后放置在 $n \times n$ 的棋盘上，并且使皇后彼此之间不能相互攻击。给一个整数 n，返回所有的解决方案。每一个方案中 'Q' 和 '!' 分别代表了皇后和空位。



➤ 输入：n = 4

➤ 输出：[[".Q..", "...Q", "Q...", "...Q"], ["..Q.", "...Q", "...Q", ".Q.."]]

解释：如上图所示，4 皇后问题存在两个不同的解法。

1) 思路分析

实现步骤:

- 选择: 从棋盘每行中选择一个列放置棋子。
- 验证合法性: 每次放置一个皇后时, 检查该位置是否与其他已经放置的皇后在列或对角线上发生冲突。
- 递归: 棋子放置位置合法则递归放置下一个棋子。
- 终止条件: 若所有棋子都放置完则终止递归。
- 回溯: 回到上一步, 尝试下一个位置。

```
def n_queens(n):  
    result = []  
    cols = set() # 记录哪些列有皇后  
    diag1 = set() # 记录哪些主对角线上有皇后  
    diag2 = set() # 记录哪些副对角线上有皇后  
  
    # 初始化棋盘  
    board = [". " for _ in range(n)] for _ in range(n)]  
  
    def backtrack(row):  
        # 如果已经放置了 n 个皇后, 说明找到一个解  
        if row == n:  
            result.append(["".join(row) for row in board])  
            return  
  
        for col in range(n):  
            # 检查当前列和对角线是否有皇后  
            if col in cols or (row - col) in diag1 or (row + col) in diag2:  
                continue # 如果有冲突, 跳过当前列  
  
            # 放置皇后  
            board[row][col] = "Q"  
            # 标记当前列和对角线  
            cols.add(col)  
            diag1.add(row - col)  
            diag2.add(row + col)  
  
            # 递归处理下一行  
            backtrack(row + 1)  
  
            # 回溯, 删除当前位置的皇后, 并清理列和对角线的标记  
            board[row][col] = ". "
```

```
cols.remove(col)
diag1.remove(row - col)
diag2.remove(row + col)

backtrack(0)
return result
```

4.6 贪心算法

4.6.1 概述

贪心算法在解决问题时，采取的是逐步选择当前状态下最优的选项（即局部最优解），并期望通过这些局部最优解得到全局最优解。在每一步中，贪心算法都会选择当前看起来最优的选择，不考虑未来的选择。并通过一系列局部最优的选择，最终达到全局最优解（虽然并非总是能得到最优解，贪心算法的有效性依赖于问题的特性）。

贪心算法的特点：

- 选择性：在每一步选择中，贪心算法根据某种启发式策略选择局部最优解。
- 不可回溯：一旦做出了选择，就不能回退或重新考虑。
- 局部最优：每一步的选择都依赖于局部最优，但并不保证整个问题能得到全局最优解。

贪心算法能够得到全局最优解的条件是问题需要满足贪心选择性质和最优子结构：

- 贪心选择性质：通过局部最优的选择可以导出全局最优解。也就是说，每一步的局部选择都不会影响后续的选择，最终可以得到全局最优。
- 最优子结构：问题的最优解包含子问题的最优解，即可以通过解决子问题来构建问题的最终解。

4.6.2 案例一：最大交换

力扣 670 题 <https://leetcode.cn/problems/maximum-swap/>

现有一个非负整数，至多可以交换一次数字中的任意两位。返回能得到的最大值。

示例：

- 输入：2736
- 输出：7236

解释：交换数字 2 和数字 7。

1) 思路分析

从右向左遍历，同时维护一个最大数的索引。

- 如果当前位置的数大于最大数，则更新最大数索引。
- 否则将当前位置的数与最大数进行交换，交换后的数更新到 result 中，再将两个位置的数交换回来将数组恢复原样。

遍历结束后，result 中的数就是能得到的最大值。

```
def maximumSwap(num):  
    result = num  
    num_list = list(str(num))  
    max_index = -1 # 最大值的索引  
    for i in range(len(num_list) - 1, -1, -1):  
        # 当前值大于最大值时，更新最大值的索引  
        if num_list[i] > num_list[max_index]:  
            max_index = i  
        # 当前值小于最大值时，交换，更新 result，再交换回来  
        else:  
            num_list[i], num_list[max_index] = num_list[max_index],  
            num_list[i]  
            result = max(result, int("".join(num_list)))  
            num_list[i], num_list[max_index] = num_list[max_index],  
            num_list[i]  
    return result
```

4.6.3 案例二：分发糖果

力扣 135 题 <https://leetcode.cn/problems/candy/>

n 个孩子站成一排。用一个整数数组 ratings 表示每个孩子的评分。按照以下要求，给这些孩子分发糖果：

- 每个孩子至少分配到 1 个糖果。
- 相邻两个孩子评分更高的孩子会获得更多的糖果。

给每个孩子分发糖果，计算并返回需要准备的最少糖果数目。

1) 方法一

每个孩子先分发 1 个糖果。

从左向右遍历，如果右边孩子比左边孩子评分高，则右边孩子糖果数量应该>左边孩子糖果数量，这时令右边孩子糖果数量为左边孩子糖果数量+1。

再从右向左遍历，如果左边孩子比右边孩子评分高，则左边孩子糖果数量应该为 max(右边孩子糖果数量+1, 自己糖果数量)。

```
def candy(ratings):
    n = len(ratings)

    # 每个孩子先分发 1 个糖果
    candy_num = [1] * len(ratings)

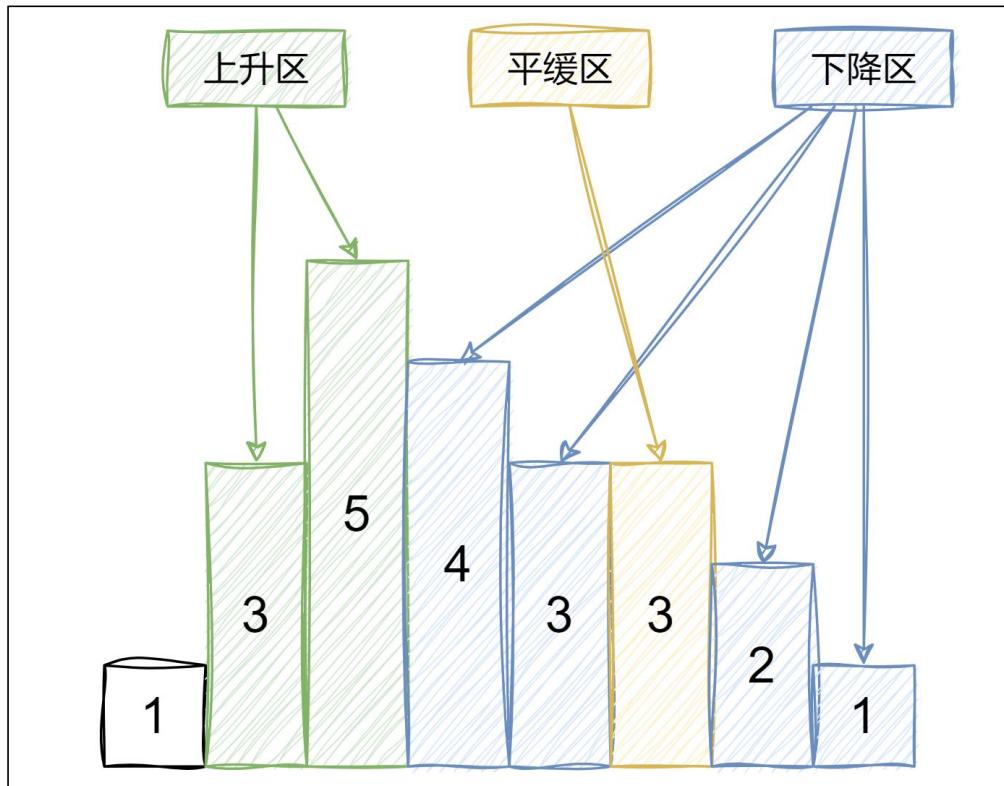
    # 从左向右遍历，如果右边评分更高，则右边孩子的糖果数改为左边孩子的糖果数+1
    for pos in range(n - 1):
        if ratings[pos] < ratings[pos + 1]:
            candy_num[pos + 1] = candy_num[pos] + 1

    # 从右向左遍历，如果左边评分更高，则左边孩子的糖果数改为右边孩子的糖果数+1
    total_candies = candy_num[-1]
    for pos in range(n - 2, -1, -1):
        if ratings[pos] > ratings[pos + 1]:
            if candy_num[pos] <= candy_num[pos + 1]:
                candy_num[pos] = candy_num[pos + 1] + 1
        total_candies += candy_num[pos]
    return total_candies
```

2) 方法二

每个孩子先分发 1 个糖果。

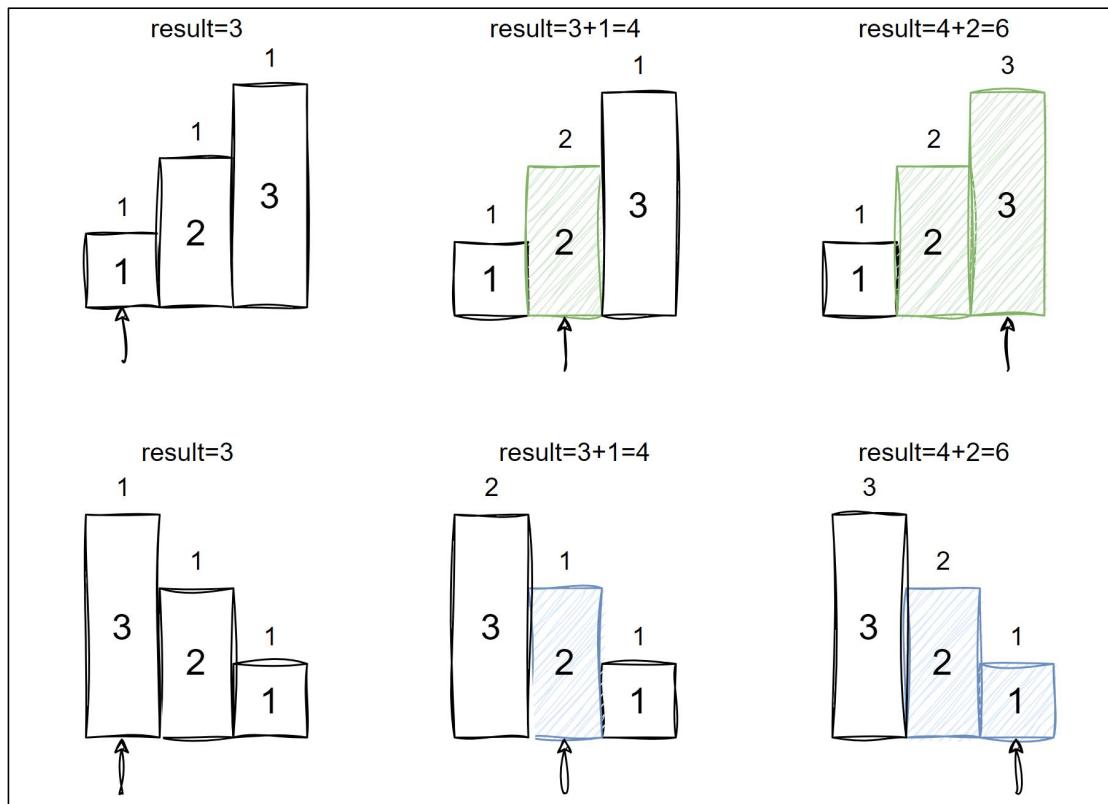
按孩子评分相较于前一个孩子评分的增加、减少、不变分成 3 个区域，分别是上升区、下降区、平缓区。



在进入不同区域时，执行不同的操作：

➤ 上升区

上升区长度+1，并清空之前的下降区长度。同时在 result 中累加糖果数。上升区或下降区长度每增加 1，result 中都需要增加与上升区或下降区长度相同数量的糖果。



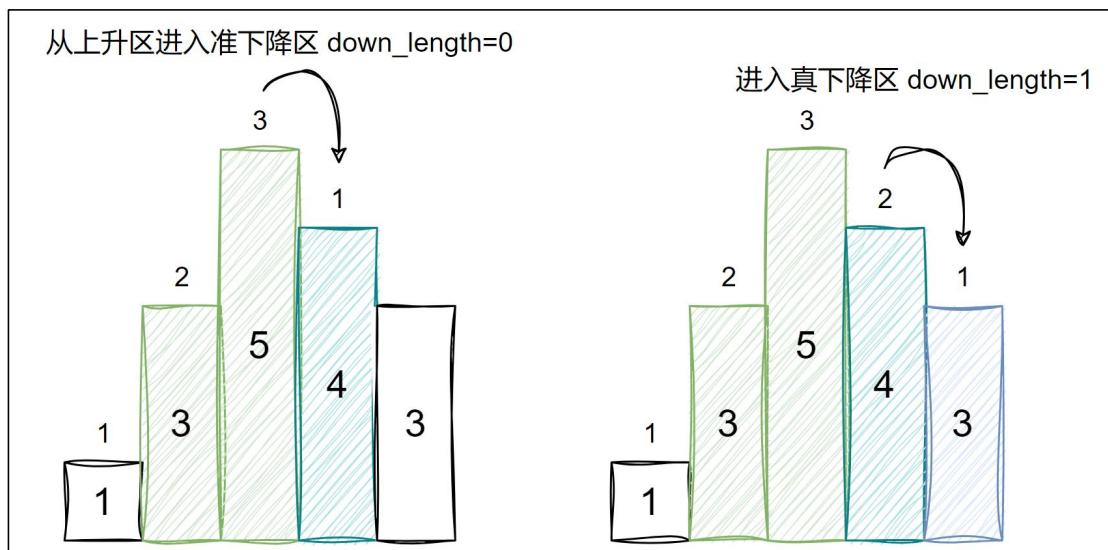
➤ 下降区

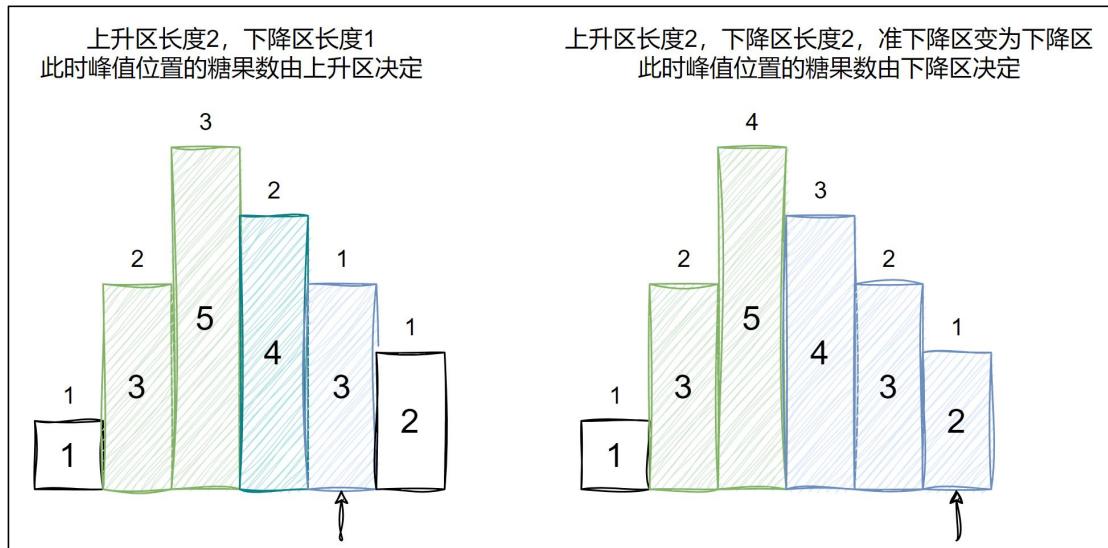
下降区存在一个特殊的准下降区。当从上升区进入下降区后，第一个进入的为准下降区。

准下降区不计入下降区长度。

当下降区长度等于上升区长度时，准下降区转变为真下降区并计入下降区长度。

下降区内清空之前的上升区长度，同时在 result 中累加糖果数。





➤ 平缓区

清空所有记录，没有其他操作。

```
def candy(ratings):
    # 每个孩子分发一个糖果
    result = len(ratings)
    # 上升区长度
    up_length = 0
    # 上升区长度的记录
    up_num = 0
    # 下降区长度
    down_length = 0
    for i in range(1, len(ratings)):
        # 如果处于上升区
        if ratings[i] > ratings[i - 1]:
            # 上升区长度+1, 清空下降区长度, 并在 result 中累加糖果数
            up_length += 1
            up_num = up_length
            down_length = 0
            result += up_length
        # 如果处于下降区
        elif ratings[i] < ratings[i - 1]:
            # 如果上升区长度为 0, 说明前一个位置已经不在上升区内, 下降区长度+1
            if up_length == 0:
                down_length += 1
            # 如果下降区长度已经等于之前上升区长度, 说明峰值的糖果数由下降区决定, 下降区长度+1
            if down_length == up_num:
                down_length += 1
```

```
# 清空上升区长度
up_length = 0
result += down_length
# 如果处于平缓区
else:
    # 清空所有记录
    up_length = 0
    down_length = 0
    up_num = 0
return result
```