

0001. 两数之和

- 标签：数组、哈希表
- 难度：简单

题目链接

- [0001. 两数之和 - 力扣](#)

题目大意

描述：给定一个整数数组 $nums$ 和一个整数目标值 $target$ 。

要求：在该数组中找出和为 $target$ 的两个整数，并输出这两个整数的下标。可以按任意顺序返回答案。

说明：

- $2 \leq nums.length \leq 10^4$ 。
- $-10^9 \leq nums[i] \leq 10^9$ 。
- $-10^9 \leq target \leq 10^9$ 。
- 只会存在一个有效答案。

示例：

- **示例 1：**

输入: $nums = [2, 7, 11, 15]$, $target = 9$

输出: $[0, 1]$

解释: 因为 $nums[0] + nums[1] == 9$ ，返回 $[0, 1]$ 。

- **示例 2：**

输入: $nums = [3, 2, 4]$, $target = 6$

输出: $[1, 2]$

解题思路

思路 1：枚举算法

1. 使用两重循环枚举数组中每一个数 $nums[i]$ 、 $nums[j]$ ，判断所有的 $nums[i] + nums[j]$ 是否等于 $target$ 。
2. 如果出现 $nums[i] + nums[j] == target$ ，则说明数组中存在和为 $target$ 的两个整数，将两个整数的下标 i 、 j 输出即可。

思路 1：代码

```
class Solution:  
    def twoSum(self, nums: List[int], target: int) -> List[int]:  
        for i in range(len(nums)):  
            for j in range(i + 1, len(nums)):  
                if i != j and nums[i] + nums[j] == target:  
                    return [i, j]  
        return []
```

思路 1：复杂度分析

- 时间复杂度： $O(n^2)$ ，其中 n 是数组 $nums$ 的元素数量。
- 空间复杂度： $O(1)$ 。

思路 2：哈希表

哈希表中键值对信息为 $target - nums[i]$: i ，其中 i 为下标。

1. 遍历数组，对于每一个数 $nums[i]$ ：
 - i. 先查找字典中是否存在 $target - nums[i]$ ，存在则输出 $target - nums[i]$ 对应的下标和当前数组的下标 i 。
 - ii. 不存在则在字典中存入 $target - nums[i]$ 的下标 i 。

思路 2：代码

```
def twoSum(self, nums: List[int], target: int) -> List[int]:  
    numDict = dict()  
    for i in range(len(nums)):  
        if target-nums[i] in numDict:  
            return numDict[target-nums[i]], i  
        numDict[nums[i]] = i  
    return [0]
```

思路 2：复杂度分析

- 时间复杂度: $O(n)$, 其中 n 是数组 $nums$ 的元素数量。
- 空间复杂度: $O(n)$ 。# 0002. 两数相加
- 标签: 递归、链表、数学
- 难度: 中等

题目链接

- [0002. 两数相加 - 力扣](#)

题目大意

描述: 给定两个非空的链表 $l1$ 和 $l2$ 。分别用来表示两个非负整数，每位数字都是按照逆序的方式存储的，每个节点存储一位数字。

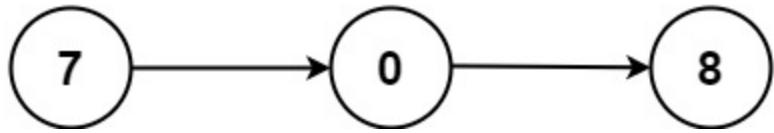
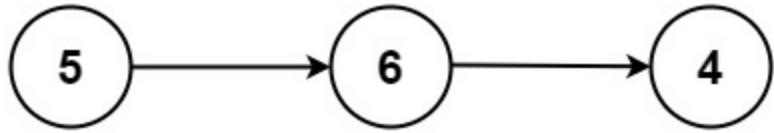
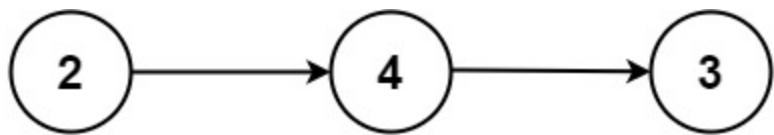
要求: 计算两个非负整数的和，并逆序返回表示和的链表。

说明:

- 每个链表中的节点数在范围 $[1, 100]$ 内。
- $0 \leq Node.val \leq 9$ 。
- 题目数据保证列表表示的数字不含前导零。

示例:

- 示例 1:



输入: `l1 = [2, 4, 3], l2 = [5, 6, 4]`

输出: `[7, 0, 8]`

解释: `342 + 465 = 807.`

- 示例 2:

输入: `l1 = [0], l2 = [0]`

输出: `[0]`

解题思路

思路 1：模拟

模拟大数加法，按位相加，将结果添加到新链表上。需要注意进位和对 10 取余。

思路 1：代码

```
class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
        head = curr = ListNode(0)
        carry = 0
        while l1 or l2 or carry:
            if l1:
                num1 = l1.val
                l1 = l1.next
            else:
                num1 = 0
            if l2:
                num2 = l2.val
                l2 = l2.next
            else:
                num2 = 0

            sum = num1 + num2 + carry
            carry = sum // 10

            curr.next = ListNode(sum % 10)
            curr = curr.next

        return head.next
```

思路 1：复杂度分析

- 时间复杂度： $O(\max(m, n))$ 。其中， m 和 n 分别是链表 $l1$ 和 $l2$ 的长度。
- 空间复杂度： $O(1)$ 。[# 0003. 无重复字符的最长子串](#)
- 标签：哈希表、字符串、滑动窗口
- 难度：中等

题目链接

- [0003. 无重复字符的最长子串 - 力扣](#)

题目大意

描述：给定一个字符串 s 。

要求：找出其中不含有重复字符的最长子串的长度。

说明：

- $0 \leq s.length \leq 5 * 10^4$ 。
- s 由英文字母、数字、符号和空格组成。

示例：

- 示例 1：

输入： $s = "abcabcbb"$

输出： 3

解释：因为无重复字符的最长子串是 "abc"，所以其长度为 3。

- 示例 2：

输入： $s = "bbbbbb"$

输出： 1

解释：因为无重复字符的最长子串是 "b"，所以其长度为 1。

解题思路

思路 1：滑动窗口（不定长度）

用滑动窗口 $window$ 来记录不重复的字符个数， $window$ 为哈希表类型。

1. 设定两个指针： $left$ 、 $right$ ，分别指向滑动窗口的左右边界，保证窗口中没有重复字符。
2. 一开始， $left$ 、 $right$ 都指向 0。
3. 向右移动 $right$ ，将最右侧字符 $s[right]$ 加入当前窗口 $window$ 中，记录该字符个数。
4. 如果该窗口中该字符的个数多于 1 个，即 $window[s[right]] > 1$ ，则不断右移 $left$ ，缩小滑动窗口长度，并更新窗口中对应字符的个数，直到 $window[s[right]] \leq 1$ 。
5. 维护更新无重复字符的最长子串长度。然后继续右移 $right$ ，直到 $right \geq len(nums)$ 结束。
6. 输出无重复字符的最长子串长度。

思路 1：代码

```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        left = 0
        right = 0
        window = dict()
        ans = 0

        while right < len(s):
            if s[right] not in window:
                window[s[right]] = 1
            else:
                window[s[right]] += 1

            while window[s[right]] > 1:
                window[s[left]] -= 1
                left += 1

            ans = max(ans, right - left + 1)
            right += 1

        return ans
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(|\Sigma|)$ 。其中 Σ 表示字符集， $|\Sigma|$ 表示字符集的大小。

0004. 寻找两个正序数组的中位数

- 标签：数组、二分查找、分治
- 难度：困难

题目链接

- [0004. 寻找两个正序数组的中位数 - 力扣](#)

题目大意

描述：给定两个正序（从小到大排序）数组 $nums1$ 、 $nums2$ 。

要求：找出并返回这两个正序数组的中位数。

说明：

- 算法的时间复杂度应该为 $O(\log(m + n))$ 。
- $nums1.length == m$ 。
- $nums2.length == n$ 。
- $0 \leq m \leq 1000$ 。
- $0 \leq n \leq 1000$ 。
- $1 \leq m + n \leq 2000$ 。
- $-10^6 \leq nums1[i], nums2[i] \leq 10^6$ 。

示例：

- **示例 1：**

输入: $nums1 = [1, 2]$, $nums2 = [3, 4]$

输出: 2.50000

解释: 合并数组 = $[1, 2, 3, 4]$ ，中位数 $(2 + 3) / 2 = 2.5$

- **示例 2：**

输入: $nums1 = [1, 2]$, $nums2 = [3, 4]$

输出: 2.50000

解释: 合并数组 = $[1, 2, 3, 4]$ ，中位数 $(2 + 3) / 2 = 2.5$

解题思路

思路 1：二分查找

单个有序数组的中位数是中间元素位置的元素。如果中间元素位置有两个元素，则为两个元素的平均数。如果是两个有序数组，则可以使用归并排序的方式将两个数组拼接为一个大的有序数组。合并后有序数组中间位置的元素，即为中位数。

当然不合并的话，我们只需找到中位数的位置即可。我们用 $n1$ 、 $n2$ 来表示数组 $nums1$ 、 $nums2$ 的长度，则合并后的大的有序数组长度为 $(n1 + n2)$ 。

我们可以发现：中位数把数组分割成了左右两部分，并且左右两部分元素个数相等。

- 如果 $(n1 + n2)$ 是奇数时，中位数是大的有序数组中第 $\lfloor \frac{(n1+n2)}{2} \rfloor + 1$ 的元素，单侧元素个数为 $\lfloor \frac{(n1+n2)}{2} \rfloor + 1$ 个（包含中位数）。
- 如果 $(n1 + n2)$ 是偶数时，中位数是第 $\lfloor \frac{(n1+n2)}{2} \rfloor$ 的元素和第 $\lfloor \frac{(n1+n2)}{2} \rfloor + 1$ 的元素的平均值，单侧元素个数为 $\lfloor \frac{(n1+n2)}{2} \rfloor$ 个。

因为是向下取整，上面两种情况综合可以写为：单侧元素个数为： $\lfloor \frac{(n1+n2+1)}{2} \rfloor$ 个。

我们用 k 来表示 $\lfloor \frac{(n1+n2+1)}{2} \rfloor$ 。现在的问题就变为了：**如何在两个有序数组中找到前 k 小的元素位置？**

如果我们从 $nums1$ 数组中取出前 $m1 (m1 \leq k)$ 个元素，那么从 $nums2$ 就需要取出前 $m2 = k - m1$ 个元素。

并且如果我们在 $nums1$ 数组中找到了合适的 $m1$ 位置，则 $m2$ 的位置也就确定了。

问题就可以进一步转换为：**如何从 $nums1$ 数组中取出前 $m1$ 个元素，使得 $nums1$ 第 $m1$ 个元素或者 $nums2$ 第 $m2 = k - m1$ 个元素为中位线位置。**

我们可以通过「二分查找」的方法，在数组 $nums1$ 中找到合适的 $m1$ 位置，具体做法如下：

1. 让 $left$ 指向 $nums1$ 的头部位置 0， $right$ 指向 $nums1$ 的尾部位置 $n1$ 。
2. 每次取中间位置作为 $m1$ ，则 $m2 = k - m1$ 。然后判断 $nums1$ 第 $m1$ 位置上元素和 $nums2$ 第 $m2 - 1$ 位置上元素之间的关系，即 $nums1[m1]$ 和 $nums2[m2 - 1]$ 的关系。
 - i. 如果 $nums1[m1] < nums2[m2 - 1]$ ，则 $nums1$ 的前 $m1$ 个元素都不可能是第 k 个元素。说明 $m1$ 取值有点小了，应该将 $m1$ 进行右移操作，即 $left = m1 + 1$ 。
 - ii. 如果 $nums1[m1] \geq nums2[m2 - 1]$ ，则说明 $m1$ 取值可能有点大了，应该将 $m1$ 进行左移。根据二分查找排除法的思路（排除一定不存在的区间，在剩下区间中继续查找），这里应取 $right = m1$ 。
3. 找到 $m1$ 的位置之后，还要根据两个数组长度和 $(n1 + n2)$ 的奇偶性，以及边界条件来计算对应的中位数。

上面之所以要判断 $nums1[m1]$ 和 $nums2[m2 - 1]$ 的关系是因为：

| 如果 $nums1[m1] < nums2[m2 - 1]$ ，则说明：

- 最多有 $m1 + m2 - 1 = k - 1$ 个元素比 $nums1[m1]$ 小，所以 $nums1[m1]$ 左侧的 $m1$ 个元素都不可能是第 k 个元素。可以将 $m1$ 左侧的元素全部排除，然后将 $m1$ 进行右移。

推理过程：

如果 $nums1[m1] < nums2[m2 - 1]$, 则:

1. $nums1[m1]$ 左侧比 $nums1[m1]$ 小的一共有 $m1$ 个元素 ($nums1[0]...nums1[m1 - 1]$ 共 $m1$ 个)。
2. $nums2$ 数组最多有 $m2 - 1$ 个元素比 $nums1[m1]$ 小 (即便是 $nums2[m2 - 1]$ 左侧所有元素都比 $nums1[m1]$ 小，也只有 $m2 - 1$ 个)。
3. 综上所述, $nums1$ 、 $nums2$ 数组中最多有 $m1 + m2 - 1 = k - 1$ 个元素比 $nums1[m1]$ 小。
4. 所以 $nums1[m1]$ 左侧的 $m1$ 个元素 ($nums1[0]...nums1[m1 - 1]$) 都不可能是第 k 个元素。可以将 $m1$ 左侧的元素全部排除，然后将 $m1$ 进行右移。

思路 1：代码

```
class Solution:
    def findMedianSortedArrays(self, nums1: List[int], nums2: List[int]) -> float:
        n1 = len(nums1)
        n2 = len(nums2)
        if n1 > n2:
            return self.findMedianSortedArrays(nums2, nums1)

        k = (n1 + n2 + 1) // 2
        left = 0
        right = n1
        while left < right:
            m1 = left + (right - left) // 2      # 在 nums1 中取前 m1 个元素
            m2 = k - m1                          # 在 nums2 中取前 m2 个元素
            if nums1[m1] < nums2[m2 - 1]:         # 说明 nums1 中所元素不够多,
                left = m1 + 1
            else:
                right = m1

        m1 = left
        m2 = k - m1

        c1 = max(float('-inf') if m1 <= 0 else nums1[m1 - 1], float('-inf') if m2 <= 0
        if (n1 + n2) % 2 == 1:
            return c1

        c2 = min(float('inf') if m1 >= n1 else nums1[m1], float('inf') if m2 >= n2 else
        return (c1 + c2) / 2
```

思路 1：复杂度分析

- 时间复杂度： $O(\log(m + n))$ 。
- 空间复杂度： $O(1)$ 。# 0005. 最长回文子串
- 标签：字符串、动态规划
- 难度：中等

题目链接

- [0005. 最长回文子串 - 力扣](#)

题目大意

描述：给定一个字符串 s 。

要求：找到 s 中最长的回文子串。

说明：

- **回文串：**如果字符串的反序与原始字符串相同，则该字符串称为回文字符串。
- $1 \leq s.length \leq 1000$ 。
- s 仅由数字和英文字母组成。

示例：

- **示例 1：**

输入: $s = "babad"$

输出: "**bab**"

解释: "**aba**" 同样是符合题意的答案。

- **示例 2：**

输入: $s = "cbbd"$

输出: "**bb**"

解题思路

思路 1：动态规划

1. 划分阶段

按照区间长度进行阶段划分。

2. 定义状态

定义状态 $dp[i][j]$ 表示为：字符串 s 在区间 $[i, j]$ 范围内是否是一个回文串。

3. 状态转移方程

- 当子串只有 1 位或 2 位的时候，如果 $s[i] == s[j]$ ，该子串为回文子串，即： $dp[i][j] = (s[i] == s[j])$ 。
- 如果子串大于 2 位，则如果 $s[i+1\dots j-1]$ 是回文串，且 $s[i] == s[j]$ ，则 $s[i\dots j]$ 也是回文串，即： $dp[i][j] = (s[i] == s[j]) \text{ and } dp[i+1][j-1]$ 。

4. 初始条件

- 初始状态下，默认字符串 s 的所有子串都不是回文串。

5. 最终结果

根据我们之前定义的状态， $dp[i][j]$ 表示为：字符串 s 在区间 $[i, j]$ 范围内是否是一个回文串。当判断完 $s[i:j]$ 是否为回文串时，同时判断并更新最长回文子串的起始位置 max_start 和最大长度 max_len 。则最终结果为 $s[max_start, max_start + max_len]$ 。

思路 1：代码

```
class Solution:
    def longestPalindrome(self, s: str) -> str:
        n = len(s)
        if n <= 1:
            return s

        dp = [[False for _ in range(n)] for _ in range(n)]
        max_start = 0
        max_len = 1

        for j in range(1, n):
            for i in range(j):
                if s[i] == s[j]:
                    if j - i <= 2:
                        dp[i][j] = True
                    else:
                        dp[i][j] = dp[i + 1][j - 1]
                if dp[i][j] and (j - i + 1) > max_len:
                    max_len = j - i + 1
                    max_start = i
        return s[max_start: max_start + max_len]
```

思路 1：复杂度分析

- 时间复杂度： $O(n^2)$ ，其中 n 是字符串的长度。
- 空间复杂度： $O(n^2)$ 。

0007. 整数反转

- 标签：数学
- 难度：中等

题目链接

- [0007. 整数反转 - 力扣](#)

题目大意

给定一个 32 位有符号整数 x ，将 x 进行反转。

解题思路

x 的范围为 $[-2^{31}, 2^{31} - 1]$ ，即 $[-2147483648, 2147483647]$ 。

反转的步骤就是让 x 不断对 10 取余，再除以 10，得到每一位的数字，同时累积结果。

注意累积结果的时候需要判断是否溢出。

当 $ans * 10 + pop > INT_MAX$ ，或者 $ans * 10 + pop < INT_MIN$ 时就会溢出。

按题设要求，无法在溢出之后对其进行判断。那么如何在进行累积操作之前判断溢出呢？

$ans * 10 + pop > INT_MAX$ 有两种情况：

1. $ans > INT_MAX / 10$ ，这种情况下，无论是否考虑 pop 进位都会溢出；
2. $ans == INT_MAX / 10$ ，这种情况下，考虑进位，如果 pop 大于 INT_MAX 的个位数，就会导致溢出。

同理 $ans * 10 + pop < INT_MIN$ 也有两种情况：

1. $ans < INT_MIN / 10$

2. $\text{ans} == \text{INT_MIN} / 10$ 且 $\text{pop} < \text{INT_MIN}$ 的个位数，就会导致溢出

代码

```
class Solution:
    def reverse(self, x: int) -> int:
        INT_MAX_10 = (1<<31)//10
        INT_MIN_10 = int((-1<<31)/10)
        INT_MAX_LAST = (1<<31) % 10
        INT_MIN_LAST = (-1<<31) % -10
        ans = 0
        while x:
            pop = x % 10 if x > 0 else x % -10
            x = x // 10 if x > 0 else int(x / 10)
            if ans > INT_MAX_10 or (ans == INT_MAX_10 and pop > INT_MAX_LAST):
                return 0
            if ans < INT_MIN_10 or (ans == INT_MIN_10 and pop < INT_MIN_LAST):
                return 0
            ans = ans*10+pop
        return ans
```

0008. 字符串转换整数 (atoi)

- 标签：字符串
- 难度：中等

题目链接

- [0008. 字符串转换整数 \(atoi\) - 力扣](#)

题目大意

描述：给定一个字符串 s 。

要求：实现一个 $\text{myAtoi}(s)$ 函数。使其能换成一个 32 位有符号整数（类似 C / C++ 中的 atoi 函数）。需要检测有效性，无法读取返回 0。

说明:

- 函数 `myAtoi(s)` 的算法如下：
 - i. 读入字符串并丢弃无用的前导空格。
 - ii. 检查下一个字符（假设还未到字符末尾）为正还是负号，读取该字符（如果有）。确定最终结果是负数还是正数。如果两者都不存在，则假定结果为正。
 - iii. 读入下一个字符，直到到达下一个非数字字符或到达输入的结尾。字符串的其余部分将被忽略。
 - iv. 将前面步骤读入的这些数字转换为整数（即，`"123" -> 123`，`"0032" -> 32`）。如果没有读入数字，则整数为 `0`。必要时更改符号（从步骤 2 开始）。
 - v. 如果整数数超过 32 位有符号整数范围 $[-2^{31}, 2^{31} - 1]$ ，需要截断这个整数，使其保持在这个范围内。具体来说，小于 -2^{31} 的整数应该被固定为 -2^{31} ，大于 $2^{31} - 1$ 的整数应该被固定为 $2^{31} - 1$ 。
 - vi. 返回整数作为最终结果。
- 本题中的空白字符只包括空格字符 ' '。
- 除前导空格或数字后的其余字符串外，请勿忽略任何其他字符。
- $0 \leq s.length \leq 200$ 。
- `s` 由英文字母（大写和小写）、数字（`0-9`）、' '、'+'、'-' 和 '.' 组成

示例:

- 示例 1：

输入： `s = "42"`

输出： `42`

解释：加粗的字符串为已经读入的字符，插入符号是当前读取的字符。

第 1 步： `"42"` （当前没有读入字符，因为没有前导空格）
 ^

第 2 步： `"42"` （当前没有读入字符，因为这里不存在 '-' 或者 '+'）
 ^

第 3 步： `"42"` （读入 `"42"`）
 ^

解析得到整数 `42`。

由于 `"42"` 在范围 `[-231, 231 - 1]` 内，最终结果为 `42`。

- 示例 2：

输入: `s = " -42"`

输出: `-42`

解释:

第 1 步: `" -42"` (读入前导空格, 但忽视掉)

^

第 2 步: `" -42"` (读入 '`-`' 字符, 所以结果应该是负数)

^

第 3 步: `" -42"` (读入 "`42`")

^

解析得到整数 `-42`。

由于 `"-42"` 在范围 `[-231, 231 - 1]` 内, 最终结果为 `-42`。

解题思路

思路 1：模拟

1. 先去除前后空格。
2. 检测正负号。
3. 读入数字，并用字符串存储数字结果。
4. 将数字字符串转为整数，并根据正负号转换整数结果。
5. 判断整数范围，并返回最终结果。

思路 1：代码

```
class Solution:
    def myAtoi(self, s: str) -> int:
        num_str = ""
        positive = True
        start = 0

        s = s.lstrip()
        if not s:
            return 0

        if s[0] == '-':
            positive = False
            start = 1
        elif s[0] == '+':
            positive = True
            start = 1
        elif not s[0].isdigit():
            return 0

        for i in range(start, len(s)):
            if s[i].isdigit():
                num_str += s[i]
            else:
                break
        if not num_str:
            return 0
        num = int(num_str)
        if not positive:
            num = -num
        return max(num, -2 ** 31)
    else:
        return min(num, 2 ** 31 - 1)
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$, 其中 n 是字符串 s 的长度。
- 空间复杂度: $O(1)$ 。

0009. 回文数

- 标签：数学
- 难度：简单

题目链接

- [0009. 回文数 - 力扣](#)

题目大意

给定整数 x ，判断 x 是否是回文数。要求不能用整数转为字符串的方式来解决这个问题。

回文数指的是正序（从左向右）和倒序（从右向左）读都是一样的整数。比如 12321。

解题思路

- 首先，负数，10 的倍数都不是回文数，可以直接排除。
- 然后将原数进行按位取余，并按位反转，若与原数完全相等，则原数为回文数。
- 其实，第二步在反转到一半的时候，就可以进行判断了。因为原数是回文数，那么在反转到中间的时候，留下的前半部分，应该与转换好的后半部分倒转过来相等。比如：1221，转换到一半，原数变为 12，转换好的数变为 12，则说明原数就是回文数。如果原数为奇数，比如：12321，转换到一半，原数变为 12，转换好的数变为 123，则应该将原数与 转换好的数对 10 取余的部分进行比较。

代码

```
class Solution:
    def isPalindrome(self, x: int) -> bool:
        if x < 0 or (x % 10 == 0 and x != 0):
            return False

        res = 0
        while x > res:
            res = res * 10 + x % 10
            x = x // 10
        return x == res or x == res // 10
```

0010. 正则表达式匹配

- 标签：递归、字符串、动态规划
- 难度：困难

题目链接

- [0010. 正则表达式匹配 - 力扣](#)

题目大意

描述：给定一个字符串 s 和一个字符模式串 p 。

要求：实现一个支持 `'.'` 和 `'*'` 的正则表达式匹配。两个字符串完全匹配才算匹配成功。如果匹配成功，则返回 `True`，否则返回 `False`。

- `'.'` 匹配任意单个字符。
- `'*'` 匹配零个或多个前面的那一个元素。

说明：

- $1 \leq s.length \leq 20$ 。
- $1 \leq p.length \leq 30$ 。
- s 只包含从 `a ~ z` 的小写字母。

- p 只包含从 $a \sim z$ 的小写字母，以及字符 $.$ 和 $*$ 。
- 保证每次出现字符 $*$ 时，前面都匹配到有效的字符。

示例：

- 示例 1：

输入： $s = "aa"$, $p = "a*"$

输出： `True`

解释：因为 $*$ 代表可以匹配零个或多个前面的那一个元素，在这里前面的元素就是 $'a'$ 。因此，字符串 $"aa"$ 可以完全匹配。

- 示例 2：

输入： $s = "aa"$, $p = "a"$

输出： `False`

解释： $"a"$ 无法匹配 $"aa"$ 整个字符串。

解题思路

思路 1：动态规划

1. 划分阶段

按照两个字符串的结尾位置进行阶段划分。

2. 定义状态

定义状态 $dp[i][j]$ 表示为：字符串 s 的前 i 个字符与字符串 p 的前 j 个字符是否匹配。

3. 状态转移方程

- 如果 $s[i - 1] == p[j - 1]$ ，则字符串 s 的第 i 个字符与字符串 p 的第 j 个字符是匹配的。此时「字符串 s 的前 i 个字符与字符串 p 的前 j 个字符是否匹配」取决于「字符串 s 的前 $i - 1$ 个字符与字符串 p 的前 $j - 1$ 个字符是否匹配」。即

$$dp[i][j] = dp[i - 1][j - 1]$$
。
- 如果 $p[j - 1] == '.'$ ，则字符串 s 的第 i 个字符与字符串 p 的第 j 个字符是匹配的（同上）。此时 $dp[i][j] = dp[i - 1][j - 1]$ 。
- 如果 $p[j - 1] == '*'$ ，则我们可以对字符 $p[j - 2]$ 进行 $0 \sim$ 若干次数的匹配。
 - 如果 $s[i - 1] != p[j - 2]$ 并且 $p[j - 2] != '.'$ ，则说明当前星号匹配不上，只能匹配 0 次（即匹配空字符串），则「字符串 s 的前 i 个字符与字符串 p 的前 j 个字符是否匹配」的结果为 `False`。
 - 如果 $s[i - 1] == p[j - 2]$ 或者 $p[j - 2] == '.'$ ，则说明当前星号匹配上，匹配 1 次，则「字符串 s 的前 i 个字符与字符串 p 的前 j 个字符是否匹配」的结果为 $dp[i][j] = dp[i - 1][j - 2]$ 。

配」取决于「字符串 s 的前 i 个字符与字符串 p 的前 $j - 2$ 个字符是否匹配」，即 $dp[i][j] = dp[i][j - 2]$ 。

- 如果 $s[i - 1] == p[j - 2]$ 或者 $p[j - 2] == \cdot\cdot\cdot$ ，则说明当前星号前面的字符 $p[j - 2]$ 可以匹配 $s[i - 1]$ 。
 - 如果匹配 0 个，则「字符串 s 的前 i 个字符与字符串 p 的前 j 个字符是否匹配」取决于「字符串 s 的前 i 个字符与字符串 p 的前 $j - 2$ 个字符是否匹配」。即 $dp[i][j] = dp[i][j - 2]$ 。
 - 如果匹配 1 个，则「字符串 s 的前 i 个字符与字符串 p 的前 j 个字符是否匹配」取决于「字符串 s 的前 i 个字符与字符串 p 的前 $j - 1$ 个字符是否匹配」。即 $dp[i][j] = dp[i][j - 1]$ 。
 - 如果匹配多个，则「字符串 s 的前 i 个字符与字符串 p 的前 j 个字符是否匹配」取决于「字符串 s 的前 $i - 1$ 个字符与字符串 p 的前 j 个字符是否匹配」。即 $dp[i][j] = dp[i - 1][j]$ 。

4. 初始条件

- 默认状态下，两个空字符串是匹配的，即 $dp[0][0] = True$ 。
- 当字符串 s 为空，字符串 p 右端有 * 时，想要匹配，则如果「空字符串」与「去掉字符串 p 右端的 * 和 * 之前的字符之后的字符串」匹配的话，则空字符串与字符串 p 匹配。也就是说如果 $p[j - 1] == \cdot\cdot\cdot$ ，则 $dp[0][j] = dp[0][j - 2]$ 。

5. 最终结果

根据我们之前定义的状态， $dp[i][j]$ 表示为：字符串 s 的前 i 个字符与字符串 p 的前 j 个字符是否匹配。则最终结果为 $dp[\text{size}_s][\text{size}_p]$ ，其实 size_s 是字符串 s 的长度， size_p 是字符串 p 的长度。

思路 1：动态规划代码

```
class Solution:
    def isMatch(self, s: str, p: str) -> bool:
        size_s, size_p = len(s), len(p)
        dp = [[False for _ in range(size_p + 1)] for _ in range(size_s + 1)]

        dp[0][0] = True
        for j in range(1, size_p + 1):
            if p[j - 1] == '*':
                dp[0][j] = dp[0][j - 2]

        for i in range(1, size_s + 1):
            for j in range(1, size_p + 1):
                if s[i - 1] == p[j - 1] or p[j - 1] == '.':
                    dp[i][j] = dp[i - 1][j - 1]
                elif p[j - 1] == '*':
                    if s[i - 1] != p[j - 2] and p[j - 2] != '.':
                        dp[i][j] = dp[i][j - 2]
                    else:
                        dp[i][j] = dp[i][j - 1] or dp[i][j - 2] or dp[i - 1][j]
                else:
                    dp[i][j] = dp[i][j - 1] or dp[i][j - 2] or dp[i - 1][j]

        return dp[size_s][size_p]
```

思路 1：复杂度分析

- **时间复杂度**: $O(mn)$, 其中 m 是字符串 s 的长度, n 是字符串 p 的长度。使用了两重循环, 外层循环遍历的时间复杂度是 $O(m)$, 内层循环遍历的时间复杂度是 $O(n)$, 所以总体的时间复杂度为 $O(mn)$ 。
- **空间复杂度**: $O(mn)$, 其中 m 是字符串 s 的长度, n 是字符串 p 的长度。使用了二维数组保存状态, 且第一维的空间复杂度为 $O(m)$, 第二维的空间复杂度为 $O(n)$, 所以总体的空间复杂度为 $O(mn)$ 。

0011. 盛最多水的容器

- 标签: 贪心、数组、双指针
- 难度: 中等

题目链接

- 0011. 盛最多水的容器 - 力扣

题目大意

描述：给定 n 个非负整数 a_1, a_2, \dots, a_n ，每个数代表坐标中的一个点 (i, a_i) 。在坐标内画 n 条垂直线，垂直线 i 的两个端点分别为 (i, a_i) 和 $(i, 0)$ 。

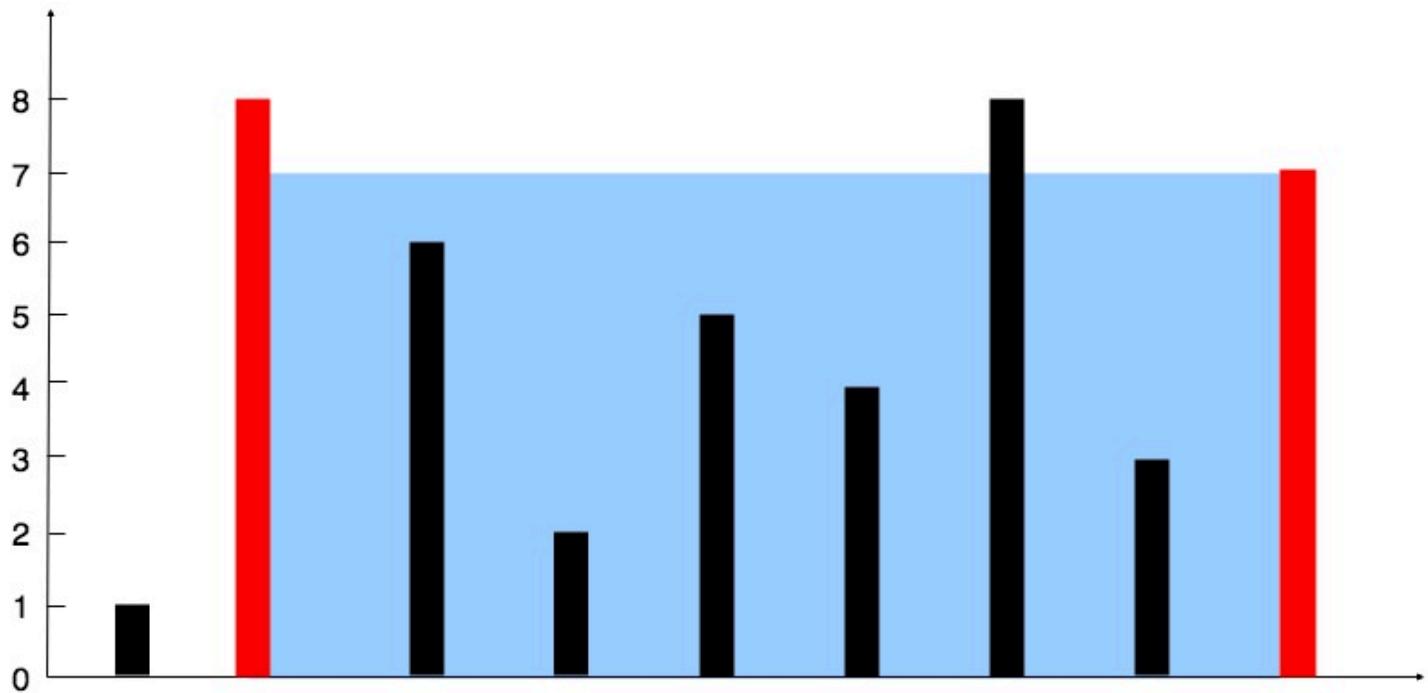
要求：找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。

说明：

- $n == height.length$ 。
- $2 \leq n \leq 10^5$ 。
- $0 \leq height[i] \leq 10^4$ 。

示例：

- **示例 1：**



输入：[1, 8, 6, 2, 5, 4, 8, 3, 7]

输出：49

解释：图中垂直线代表输入数组 [1, 8, 6, 2, 5, 4, 8, 3, 7]。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 49。

解题思路

思路 1：对撞指针

从示例中可以看出，如果确定好左右两端的直线，容纳的水量是由
左右两端直线中较低直线的高度 * 两端直线之间的距离 所决定的。所以我们应该使得 **较低直线的高度尽可能的高**，这样才能使盛水面积尽可能的大。

可以使用对撞指针求解。移动较低直线所在的指针位置，从而得到不同的高度和面积，最终获取其中最大的面积。具体做法如下：

1. 使用两个指针 `left` , `right` 。 `left` 指向数组开始位置，`right` 指向数组结束位置。
2. 计算 `left` 和 `right` 所构成的面积值，同时维护更新最大面积值。
3. 判断 `left` 和 `right` 的高度值大小。
 - i. 如果 `left` 指向的直线高度比较低，则将 `left` 指针右移。
 - ii. 如果 `right` 指向的直线高度比较低，则将 `right` 指针左移。
4. 如果遇到 `left == right` , 跳出循环，最后返回最大的面积。

思路 1：代码

```
class Solution:  
    def maxArea(self, height: List[int]) -> int:  
        left = 0  
        right = len(height) - 1  
        ans = 0  
        while left < right:  
            area = min(height[left], height[right]) * (right-left)  
            ans = max(ans, area)  
            if height[left] < height[right]:  
                left += 1  
            else:  
                right -= 1  
        return ans
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(1)$ 。

0012. 整数转罗马数字

- 标签：哈希表、数学、字符串
- 难度：中等

题目链接

- [0012. 整数转罗马数字 - 力扣](#)

题目大意

给定一个整数，将其转换为罗马数字。

罗马数字规则：

- I 代表数值 1, V 代表数值 5, X 代表数值 10, L 代表数值 50, C 代表数值 100, D 代表数值 500, M 代表数值 1000；
- 一般罗马数字较大数字在左边，较小数字在右边，此时值为两者之和，比如 $XI = X + I = 10 + 1 = 11$ 。
- 例外情况下，较小数字在左边，较大数字在右边，此时值为后者减前者之差，比如 $IX = X - I = 10 - 1 = 9$ 。

解题思路

根据规则，可以得出：

- I 代表数值 1, V 代表数值 5, X 代表数值 10, L 代表数值 50, C 代表数值 100, D 代表数值 500, M 代表数值 1000；
- CM 代表 900, CD 代表 400, XC 代表 90, XL 代表 40, IX 代表 9, IV 代表 4。

依次排序可得：

- 1000 : M、900 : CM、D : 500、400 : CD、100 : C、90 : XC、50 : L、40 : XL、10 : X、9 : IX、5 : V、4 : IV、1 : I。

使用贪心算法。每次尽量用最大的数对应的罗马字符来表示。先选择 1000，再选择 900，然后 500，等等。

代码

```
class Solution:
    def intToRoman(self, num: int) -> str:
        roman_dict = {1000:'M', 900:'CM', 500:'D', 400:'CD', 100:'C', 90:'XC', 50:'L',
        res = ""
        for key in roman_dict:
            if num // key != 0:
                res += roman_dict[key] * (num // key)
                num %= key
        return res
```

0013. 罗马数字转整数

- 标签：哈希表、数学、字符串
- 难度：简单

题目链接

- [0013. 罗马数字转整数 - 力扣](#)

题目大意

给定一个罗马数字对应的字符串，将其转换为整数。

罗马数字规则：

- I 代表数值 1, V 代表数值 5, X 代表数值 10, L 代表数值 50, C 代表数值 100, D 代表数值 500, M 代表数值 1000;
- 一般罗马数字较大数字在左边，较小数字在右边，此时值为两者之和，比如 XI = X + I = 10 + 1 = 11。
- 例外情况下，较小数字在左边，较大数字在右边，此时值为后者减前者之差，比如 IX = X - I = 10 - 1 = 9。

解题思路

用一个哈希表存储罗马数字与对应数值关系。遍历罗马数字对应的字符串，判断相邻两个数大小关系，并计算对应结果。

代码

```
class Solution:
    def romanToInt(self, s: str) -> int:
        numbers = {
            "I" : 1,
            "V" : 5,
            "X" : 10,
            "L" : 50,
            "C" : 100,
            "D" : 500,
            "M" : 1000
        }
        sum = 0
        pre_num = numbers[s[0]]
        for i in range(1, len(s)):
            cur_num = numbers[s[i]]
            if pre_num < cur_num:
                sum -= pre_num
            else:
                sum += pre_num
            pre_num = cur_num
        sum += pre_num
        return sum
```

0014. 最长公共前缀

- 标签：字典树、字符串
- 难度：简单

题目链接

- 0014. 最长公共前缀 - 力扣

题目大意

描述：给定一个字符串数组 `strs`。

要求：返回字符串数组中的最长公共前缀。如果不存在公共前缀，返回空字符串 `""`。

说明：

- $1 \leq \text{strs.length} \leq 200$ 。
- $0 \leq \text{strs}[i].length \leq 200$ 。
- `strs[i]` 仅由小写英文字母组成。

示例：

- **示例 1：**

输入: `strs = ["flower", "flow", "flight"]`

输出: `"fl"`

- **示例 2：**

输入: `strs = ["dog", "racecar", "car"]`

输出: `""`

解释: 输入不存在公共前缀。

解题思路

思路 1：纵向遍历

1. 依次遍历所有字符串的每一列，比较相同位置上的字符是否相同。
 - i. 如果相同，则继续对下一列进行比较。
 - ii. 如果不相同，则当前列字母不再属于公共前缀，直接返回当前列之前的部分。
2. 如果遍历结束，说明字符串数组中的所有字符串都相等，则可将字符串数组中的第一个字符串作为公共前缀进行返回。

思路 1：代码

```
class Solution:
    def longestCommonPrefix(self, strs: List[str]) -> str:
        if not strs:
            return ""

        length = len(strs[0])
        count = len(strs)
        for i in range(length):
            c = strs[0][i]
            for j in range(1, count):
                if len(strs[j]) == i or strs[j][i] != c:
                    return strs[0][:i]
        return strs[0]
```

思路 1：复杂度分析

- 时间复杂度: $O(m \times n)$, 其中 m 是字符串数组中的字符串的平均长度, n 是字符串的数量。
- 空间复杂度: $O(1)$ 。# 0015. 三数之和
- 标签: 数组、双指针、排序
- 难度: 中等

题目链接

- 0015. 三数之和 - 力扣

题目大意

描述: 给定一个整数数组 $nums$ 。

要求: 判断 $nums$ 中是否存在三个元素 a 、 b 、 c , 满足 $a + b + c == 0$ 。要求找出所有满足要求的不重复的三元组。

说明:

- $3 \leq nums.length \leq 3000$ 。
- $-10^5 \leq nums[i] \leq 10^5$ 。

示例:

- 示例 1:

输入: `nums = [-1, 0, 1, 2, -1, -4]`

输出: `[[-1, -1, 2], [-1, 0, 1]]`

- 示例 2:

输入: `nums = [0, 1, 1]`

输出: `[]`

解题思路

思路 1：对撞指针

直接三重遍历查找 a 、 b 、 c 的时间复杂度是: $O(n^3)$ 。我们可以通过一些操作来降低复杂度。

先将数组进行排序，以保证按顺序查找 a 、 b 、 c 时，元素值为升序，从而保证所找到的三个元素是不重复的。同时也方便下一步使用双指针减少一重遍历。时间复杂度为: $O(n \times \log n)$ 。

第一重循环遍历 a ，对于每个 a 元素，从 a 元素的下一个位置开始，使用对撞指针 $left$, $right$ 。
 $left$ 指向 a 元素的下一个位置， $right$ 指向末尾位置。先将 $left$ 右移、 $right$ 左移去除重复元素，再进行下边的判断。

1. 如果 $nums[a] + nums[left] + nums[right] == 0$ ，则得到一个解，将其加入答案数组中，并继续将 $left$ 右移， $right$ 左移；
2. 如果 $nums[a] + nums[left] + nums[right] > 0$ ，说明 $nums[right]$ 值太大，将 $right$ 向左移；
3. 如果 $nums[a] + nums[left] + nums[right] < 0$ ，说明 $nums[left]$ 值太小，将 $left$ 右移。

思路 1：代码

```
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        n = len(nums)
        nums.sort()
        ans = []

        for i in range(n):
            if i > 0 and nums[i] == nums[i - 1]:
                continue
            left = i + 1
            right = n - 1
            while left < right:
                while left < right and left > i + 1 and nums[left] == nums[left - 1]:
                    left += 1
                while left < right and right < n - 1 and nums[right + 1] == nums[right]:
                    right -= 1
                if left < right and nums[i] + nums[left] + nums[right] == 0:
                    ans.append([nums[i], nums[left], nums[right]])
                    left += 1
                    right -= 1
                elif nums[i] + nums[left] + nums[right] > 0:
                    right -= 1
                else:
                    left += 1
        return ans
```

思路 1：复杂度分析

- 时间复杂度： $O(n^2)$ 。
- 空间复杂度： $O(n)$ 。

0016. 最接近的三数之和

- 标签：数组、双指针、排序
- 难度：中等

题目链接

- 0016. 最接近的三数之和 - 力扣

题目大意

描述：给定一个整数数组 $nums$ 和一个目标值 $target$ 。

要求：从 $nums$ 中选出三个整数，使它们的和与 $target$ 最接近。返回这三个数的和。假定每组输入只存在恰好一个解。

说明：

- $3 \leq nums.length \leq 1000$ 。
- $-1000 \leq nums[i] \leq 1000$ 。
- $-10^4 \leq target \leq 10^4$ 。

示例：

- **示例 1：**

输入: $nums = [-1, 2, 1, -4]$, $target = 1$

输出: 2

解释: 与 $target$ 最接近的和是 2 ($-1 + 2 + 1 = 2$)。

- **示例 2：**

输入: $nums = [0, 0, 0]$, $target = 1$

输出: 0

解题思路

思路 1：对撞指针

直接暴力枚举三个数的时间复杂度是 $O(n^3)$ 。很明显的容易超时。考虑使用双指针减少循环内的时间复杂度。具体做法如下：

- 先对数组进行从小到大排序，使用 ans 记录最接近的三数之和。

- 遍历数组，对于数组元素 $nums[i]$ ，使用两个指针 $left$ 、 $right$ 。 $left$ 指向第 0 个元素位置， $right$ 指向第 $i - 1$ 个元素位置。
- 计算 $nums[i]$ 、 $nums[left]$ 、 $nums[right]$ 的和与 $target$ 的差值，将其与 ans 与 $target$ 的差值作比较。如果差值小，则更新 ans 。
 - 如果 $nums[i] + nums[left] + nums[right] < target$ ，则说明 $left$ 小了，应该将 $left$ 右移，继续查找。
 - 如果 $nums[i] + nums[left] + nums[right] \geq target$ ，则说明 $right$ 太大了，应该将 $right$ 左移，然后继续判断。
- 当 $left == right$ 时，区间搜索完毕，继续遍历 $nums[i + 1]$ 。
- 最后输出 ans 。

这种思路使用了两重循环，其中内层循环当 $left == right$ 时循环结束，时间复杂度为 $O(n)$ ，外层循环时间复杂度也是 $O(n)$ 。所以算法的整体时间复杂度为 $O(n^2)$ 。

思路 1：代码

```
class Solution:
    def threeSumClosest(self, nums: List[int], target: int) -> int:
        nums.sort()
        res = float('inf')
        size = len(nums)
        for i in range(2, size):
            left = 0
            right = i - 1
            while left < right:
                total = nums[left] + nums[right] + nums[i]
                if abs(total - target) < abs(res - target):
                    res = total
                if total < target:
                    left += 1
                else:
                    right -= 1
        return res
```

思路 1：复杂度分析

- 时间复杂度： $O(n^2)$ ，其中 n 为数组中元素的个数。
- 空间复杂度： $O(\log n)$ ，排序需要 $\log n$ 的栈空间。

0017. 电话号码的字母组合

- 标签：哈希表、字符串、回溯
- 难度：中等

题目链接

- [0017. 电话号码的字母组合 - 力扣](#)

题目大意

描述：给定一个只包含数字 2~9 的字符串 `digits`。给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。



要求：返回字符串 `digits` 在九宫格键盘上所能表示的所有字母组合。答案可以按「任意顺序」返回。

说明：

- $0 \leq digits.length \leq 4$ 。
- `digits[i]` 是范围 $2 \sim 9$ 的一个数字。

示例：

- 示例 1：

输入: `digits = "23"`
输出: `["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]`

- 示例 2：

输入: `digits = "2"`
输出: `["a", "b", "c"]`

解题思路

思路 1：回溯算法 + 哈希表

用哈希表保存每个数字键位对应的所有可能的字母，然后进行回溯操作。

回溯过程中，维护一个字符串 `combination`，表示当前的字母排列组合。初始字符串为空，每次取电话号码的一位数字，从哈希表中取出该数字所对应的所有字母，并将其中一个插入到 `combination` 后面，然后继续处理下一个数字，知道处理完所有数字，得到一个完整的字母排列。开始进行回退操作，遍历其余的字母排列。

思路 1：代码

```
class Solution:
    def letterCombinations(self, digits: str) -> List[str]:
        if not digits:
            return []

        phone_dict = {
            "2": "abc",
            "3": "def",
            "4": "ghi",
            "5": "jkl",
            "6": "mno",
            "7": "pqrs",
            "8": "tuv",
            "9": "wxyz"
        }

        def backtrack(combination, index):
            if index == len(digits):
                combinations.append(combination)
            else:
                digit = digits[index]
                for letter in phone_dict[digit]:
                    backtrack(combination + letter, index + 1)

        combinations = []
        backtrack('', 0)
        return combinations
```

思路 1：复杂度分析

- **时间复杂度**: $O(3^m \times 4^n)$, 其中 m 是 `digits` 中对应 3 个字母的数字个数, n 是 `digits` 中对应 4 个字母的数字个数。
- **空间复杂度**: $O(m + n)$ 。

0018. 四数之和

- 标签：数组、双指针、排序
- 难度：中等

题目链接

- [0018. 四数之和 - 力扣](#)

题目大意

描述：给定一个整数数组 $nums$ 和一个目标值 $target$ 。

要求：找出所有满足以下条件且不重复的四元组。

1. $0 \leq a, b, c, d < n$ 。
2. a 、 b 、 c 和 d 互不相同。
3. $nums[a] + nums[b] + nums[c] + nums[d] == target$ 。

说明：

- $1 \leq nums.length \leq 200$ 。
- $-10^9 \leq nums[i] \leq 10^9$ 。
- $-10^9 \leq target \leq 10^9$ 。

示例：

- **示例 1：**

输入: `nums = [1,0,-1,0,-2,2]`, `target = 0`

输出: `[[-2, -1, 1, 2], [-2, 0, 0, 2], [-1, 0, 0, 1]]`

- **示例 2：**

输入: `nums = [2,2,2,2,2]`, `target = 8`

输出: `[[2,2,2,2]]`

解题思路

思路 1：排序 + 双指针

和 [0015. 三数之和](#) 解法类似。

直接三重遍历查找 a 、 b 、 c 、 d 的时间复杂度是: $O(n^4)$ 。我们可以通过一些操作来降低复杂度。

1. 先将数组进行排序，以保证按顺序查找 a 、 b 、 c 、 d 时，元素值为升序，从而保证所找到的四个元素是不重复的。同时也方便下一步使用双指针减少一重遍历。这一步的时间复杂度为： $O(n \times \log n)$ 。
2. 两重循环遍历元素 a 、 b ，对于每个 a 元素，从 a 元素的下一个位置开始遍历元素 b 。对于元素 a 、 b ，使用双指针 $left$, $right$ 来查找 c 、 d 。 $left$ 指向 b 元素的下一个位置， $right$ 指向末尾位置。先将 $left$ 右移、 $right$ 左移去除重复元素，再进行下边的判断。
 - i. 如果 $nums[a] + nums[b] + nums[left] + nums[right] == target$, 则得到一个解，将其加入答案数组中，并继续将 $left$ 右移， $right$ 左移；
 - ii. 如果 $nums[a] + nums[b] + nums[left] + nums[right] > target$, 说明 $nums[right]$ 值太大，将 $right$ 向左移；
 - iii. 如果 $nums[a] + nums[b] + nums[left] + nums[right] < target$, 说明 $nums[left]$ 值太小，将 $left$ 右移。

思路 1：代码

```
class Solution:
    def fourSum(self, nums: List[int], target: int) -> List[List[int]]:
        n = len(nums)
        nums.sort()
        ans = []

        for i in range(n):
            if i > 0 and nums[i] == nums[i-1]:
                continue
            for j in range(i+1, n):
                if j > i+1 and nums[j] == nums[j-1]:
                    continue
                left = j + 1
                right = n - 1
                while left < right:
                    while left < right and left > j + 1 and nums[left] == nums[left - 1]:
                        left += 1
                    while left < right and right < n - 1 and nums[right + 1] == nums[right]:
                        right -= 1
                    if left < right and nums[i] + nums[j] + nums[left] + nums[right] == target:
                        ans.append([nums[i], nums[j], nums[left], nums[right]])
                    left += 1
                    right -= 1
                elif nums[i] + nums[j] + nums[left] + nums[right] > target:
                    right -= 1
                else:
                    left += 1
        return ans
```

思路 1：复杂度分析

- 时间复杂度： $O(n^3)$ ，其中 n 为数组中元素个数。
- 空间复杂度： $O(\log n)$ ，排序额外使用空间为 $\log n$ 。

0019. 删除链表的倒数第 N 个结点

- 标签：链表、双指针
- 难度：中等

题目链接

- 0019. 删除链表的倒数第 N 个结点 - 力扣

题目大意

描述：给定一个链表的头节点 `head`。

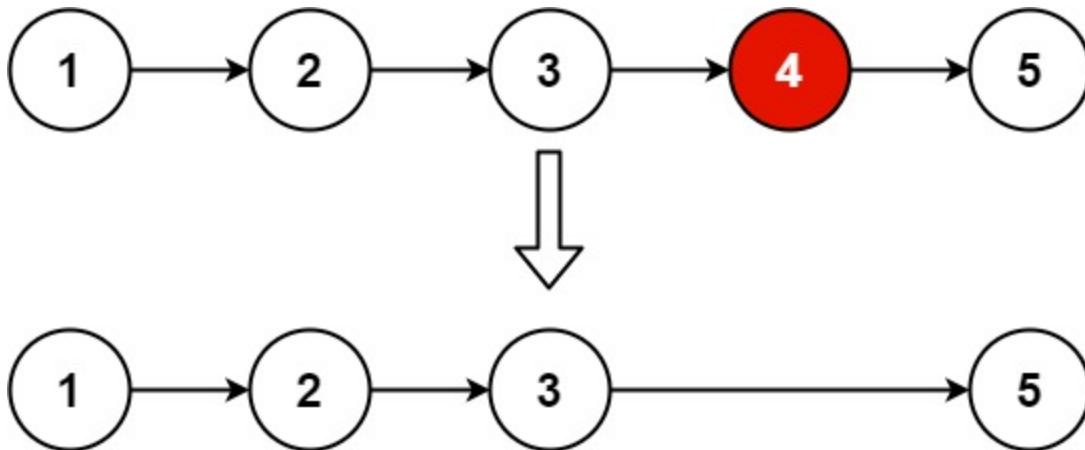
要求：删除链表的倒数第 n 个节点，并且返回链表的头节点。

说明：

- 要求使用一次遍历实现。
- 链表中结点的数目为 sz 。
- $1 \leq sz \leq 30$ 。
- $0 \leq Node.val \leq 100$ 。
- $1 \leq n \leq sz$ 。

示例：

- 示例 1：



输入：`head = [1,2,3,4,5]`, $n = 2$

输出：`[1,2,3,5]`

- 示例 2：

输入：`head = [1]`, $n = 1$

输出：`[]`

解题思路

思路 1：快慢指针

常规思路是遍历一遍链表，求出链表长度，再遍历一遍到对应位置，删除该位置上的节点。

如果用一次遍历实现的话，可以使用快慢指针。让快指针先走 n 步，然后快慢指针、慢指针再同时走，每次一步，这样等快指针遍历到链表尾部的时候，慢指针就刚好遍历到了倒数第 n 个节点位置。将该位置上的节点删除即可。

需要注意的是要删除的节点可能包含了头节点。我们可以考虑在遍历之前，新建一个头节点，让其指向原来的头节点。这样，最终如果删除的是头节点，则删除原头节点即可。返回结果的时候，可以直接返回新建头节点的下一位节点。

思路 1：代码

```
class Solution:
    def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:
        newHead = ListNode(0, head)
        fast = head
        slow = newHead
        while n:
            fast = fast.next
            n -= 1
        while fast:
            fast = fast.next
            slow = slow.next
        slow.next = slow.next.next
        return newHead.next
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(1)$ 。

0020. 有效的括号

- 标签：栈、字符串

- 难度：简单

题目链接

- 0020. 有效的括号 - 力扣

题目大意

描述：给定一个只包括 '(', ')', '{', '}', '[', ']' 的字符串 s。

要求：判断字符串 s 是否有效（即括号是否匹配）。

说明：

- 有效字符串需满足：
 - i. 左括号必须用相同类型的右括号闭合。
 - ii. 左括号必须以正确的顺序闭合。

示例：

- 示例 1：

输入：s = "()"

输出：True

- 示例 2：

输入：s = "()[]{}"

输出：True

解题思路

思路 1：栈

括号匹配是「栈」的经典应用。我们可以用栈来解决这道题。具体做法如下：

1. 先判断一下字符串的长度是否为偶数。因为括号是成对出现的，所以字符串的长度应为偶数，可以直接判断长度为奇数的字符串不匹配。如果字符串长度为奇数，则说明字符串 s 中的括号不匹

配，直接返回 `False`。

2. 使用栈 `stack` 来保存未匹配的左括号。然后依次遍历字符串 `s` 中的每一个字符。
 - i. 如果遍历到左括号时，将其入栈。
 - ii. 如果遍历到右括号时，先看栈顶元素是否是与当前右括号相同类型的左括号。
 - a. 如果是与当前右括号相同类型的左括号，则令其出栈，继续向前遍历。
 - b. 如果不是与当前右括号相同类型的左括号，则说明字符串 `s` 中的括号不匹配，直接返回 `False`。
3. 遍历完，还要再判断一下栈是否为空。
 - i. 如果栈为空，则说明字符串 `s` 中的括号匹配，返回 `True`。
 - ii. 如果栈不为空，则说明字符串 `s` 中的括号不匹配，返回 `False`。

思路 1：代码

```
class Solution:  
    def isValid(self, s: str) -> bool:  
        if len(s) % 2 == 1:  
            return False  
        stack = list()  
        for ch in s:  
            if ch == '(' or ch == '[' or ch == '{':  
                stack.append(ch)  
            elif ch == ')':  
                if len(stack) != 0 and stack[-1] == '(':  
                    stack.pop()  
                else:  
                    return False  
            elif ch == ']':  
                if len(stack) != 0 and stack[-1] == '[':  
                    stack.pop()  
                else:  
                    return False  
            elif ch == '}':  
                if len(stack) != 0 and stack[-1] == '{':  
                    stack.pop()  
                else:  
                    return False  
        if len(stack) == 0:  
            return True  
        else:  
            return False
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(n)$ 。

0021. 合并两个有序链表

- 标签：递归、链表
- 难度：简单

题目链接

- [0021. 合并两个有序链表 - 力扣](#)

题目大意

描述：给定两个升序链表的头节点 `list1` 和 `list2`。

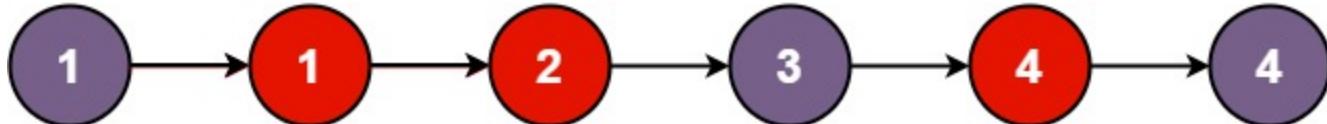
要求：将其合并为一个升序链表。

说明：

- 两个链表的节点数目范围是 $[0, 50]$ 。
- $-100 \leq \text{Node.val} \leq 100$ 。
- `list1` 和 `list2` 均按 **非递减顺序** 排列

示例：

- **示例 1：**



输入: `list1 = [1,2,4], list2 = [1,3,4]`

输出: `[1,1,2,3,4,4]`

- 示例 2:

输入: `list1 = [], list2 = []`

输出: `[]`

解题思路

思路 1：归并排序

利用归并排序的思想，具体步骤如下：

1. 使用哑节点 `dummy_head` 构造一个头节点，并使用 `curr` 指向 `dummy_head` 用于遍历。
2. 然后判断 `list1` 和 `list2` 头节点的值，将较小的头节点加入到合并后的链表中。并向后移动该链表的头节点指针。
3. 然后重复上一步操作，直到两个链表中出现链表为空的情况。
4. 将剩余链表链接到合并后的链表中。
5. 将哑节点 `dummy_head` 的下一个链节点 `dummy_head.next` 作为合并后有序链表的头节点返回。

思路 1：代码

```
class Solution:
    def mergeTwoLists(self, list1: Optional[ListNode], list2: Optional[ListNode]) -> Optional[ListNode]:
        dummy_head = ListNode(-1)

        curr = dummy_head
        while list1 and list2:
            if list1.val <= list2.val:
                curr.next = list1
                list1 = list1.next
            else:
                curr.next = list2
                list2 = list2.next
            curr = curr.next

        curr.next = list1 if list1 is not None else list2
        return dummy_head.next
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(1)$ 。

0022. 括号生成

- 标签：字符串、回溯算法
- 难度：中等

题目链接

- [0022. 括号生成 - 力扣](#)

题目大意

描述：给定一个整数 n ，代表生成括号的对数。

要求：生成所有有可能且有效的括号组合。

说明：

- $1 \leq n \leq 8$ 。

示例：

- **示例 1：**

输入：n = 3

输出：["((()))", "(()())", "(())()", "()(())", "()()()"]

- **示例 2：**

输入：n = 1

输出：["()"]

解题思路

思路 1：回溯算法

为了生成的括号组合是有效的，回溯的时候，使用一个标记变量 `symbol` 来表示是否当前组合是否成对匹配。

如果在当前组合中增加一个 (，则令 `symbol` 加 1，如果增加一个)，则令 `symbol` 减 1。

显然只有在 `symbol < n` 的时候，才能增加 (，在 `symbol > 0` 的时候，才能增加)。

如果最终生成 $2 \times n$ 的括号组合，并且 `symbol == 0`，则说明当前组合是有效的，将其加入到最终答案数组中。

下面我们根据回溯算法三步走，写出对应的回溯算法。

1. **明确所有选择：** $2 \times n$ 的括号组合中的每个位置，都可以从 (或者) 中选出。并且，只有在 `symbol < n` 的时候，才能选择 (，在 `symbol > 0` 的时候，才能选择)。

2. **明确终止条件：**

- 当遍历到决策树的叶子节点时，就终止了。即当前路径搜索到末尾时，递归终止。

3. **将决策树和终止条件翻译成代码：**

- i. 定义回溯函数：

- `backtracking(symbol, index)`: 函数的传入参数是 `symbol` (用于表示是否当前组合是否成对匹配), `index` (当前元素下标), 全局变量是 `parentheses` (用于保存所有有效的括号组合), `parenthesis` (当前括号组合),。
- `backtracking(symbol, index)` 函数代表的含义是: 递归根据 `symbol`, 在 (和) 中选择第 `index` 个元素。

ii. 书写回溯函数主体 (给出选择元素、递归搜索、撤销选择部分)。

- 从目前正在考虑元素, 到第 $2 \times n$ 个元素为止, 枚举出所有可选的元素。对于每一个可选元素:
 - 约束条件: `symbol < n` 或者 `symbol > 0`。
 - 选择元素: 将其添加到当前括号组合 `parenthesis` 中。
 - 递归搜索: 在选择该元素的情况下, 继续递归选择剩下元素。
 - 撤销选择: 将该元素从当前括号组合 `parenthesis` 中移除。

```
if symbol < n:
    parenthesis.append('(')
    backtrack(symbol + 1, index + 1)
    parenthesis.pop()
if symbol > 0:
    parenthesis.append(')')
    backtrack(symbol - 1, index + 1)
    parenthesis.pop()
```

iii. 明确递归终止条件 (给出递归终止条件, 以及递归终止时的处理方法)。

- 当遍历到决策树的叶子节点时, 就终止了。也就是当 `index == 2 * n` 时, 递归停止。
- 并且在 `symbol == 0` 时, 当前组合才是有效的, 此时将其加入到最终答案数组中。

思路 1：代码

```
class Solution:
    def generateParenthesis(self, n: int) -> List[str]:
        parentheses = []           # 存放所有括号组合
        parenthesis = []            # 存放当前括号组合
        def backtrack(symbol, index):
            if n * 2 == index:
                if symbol == 0:
                    parentheses.append("".join(parenthesis))
                else:
                    if symbol < n:
                        parenthesis.append('(')
                        backtrack(symbol + 1, index + 1)
                        parenthesis.pop()
                    if symbol > 0:
                        parenthesis.append(')')
                        backtrack(symbol - 1, index + 1)
                        parenthesis.pop()
            backtrack(0, 0)
        return parentheses
```

思路 1：复杂度分析

- 时间复杂度： $O(\frac{2^{2n}}{\sqrt{n}})$ ，其中 n 为生成括号的对数。
- 空间复杂度： $O(n)$ 。

参考资料

- 【题解】22. 括号生成 - 力扣 (Leetcode) # 0023. 合并 K 个升序链表
- 标签：链表、分治、堆（优先队列）、归并排序
- 难度：困难

题目链接

- 0023. 合并 K 个升序链表 - 力扣

题目大意

描述：给定一个链表数组，每个链表都已经按照升序排列。

要求：将所有链表合并到一个升序链表中，返回合并后的链表。

说明：

- $k == lists.length$ 。
- $0 \leq k \leq 10^4$ 。
- $0 \leq lists[i].length \leq 500$ 。
- $-10^4 \leq lists[i][j] \leq 10^4$ 。
- $lists[i]$ 按升序排列。
- $lists[i].length$ 的总和不超过 10^4 。

示例：

- **示例 1：**

输入: `lists = [[1,4,5],[1,3,4],[2,6]]`

输出: `[1,1,2,3,4,4,5,6]`

解释: 链表数组如下:

```
[  
    1->4->5,  
    1->3->4,  
    2->6  
]
```

将它们合并到一个有序链表中得到。

`1->1->2->3->4->4->5->6`

- **示例 2：**

输入: `lists = []`

输出: `[]`

解题思路

思路 1：分治算法

分而治之的思想。将链表数组不断二分，转为规模为二分之一的子问题，然后再进行归并排序。

思路 1：代码

```
class Solution:
    def merge_sort(self, lists: List[ListNode], left: int, right: int) -> ListNode:
        if left == right:
            return lists[left]
        mid = left + (right - left) // 2
        node_left = self.merge_sort(lists, left, mid)
        node_right = self.merge_sort(lists, mid + 1, right)
        return self.merge(node_left, node_right)

    def merge(self, a: ListNode, b: ListNode) -> ListNode:
        root = ListNode(-1)
        cur = root
        while a and b:
            if a.val < b.val:
                cur.next = a
                a = a.next
            else:
                cur.next = b
                b = b.next
            cur = cur.next
        if a:
            cur.next = a
        if b:
            cur.next = b
        return root.next

    def mergeKLists(self, lists: List[ListNode]) -> ListNode:
        if not lists:
            return None
        size = len(lists)
        return self.merge_sort(lists, 0, size - 1)
```

思路 1：复杂度分析

- 时间复杂度： $O(k \times n \times \log_2 k)$ 。
- 空间复杂度： $O(\log_2 k)$ 。

0024. 两两交换链表中的节点

- 标签：递归、链表
- 难度：中等

题目链接

- 0024. 两两交换链表中的节点 - 力扣

题目大意

描述：给定一个链表的头节点 `head`。

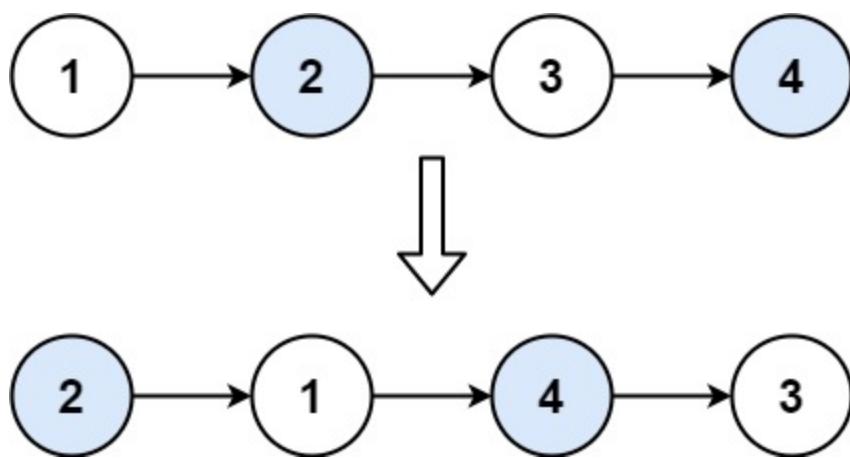
要求：按顺序将链表中每两个节点交换一下，并返回交换后的链表。

说明：

- 需要实际进行节点交换，而不是纸改变节点内部的值。
- 链表中节点的数目在范围 $[0, 100]$ 内。
- $0 \leq \text{Node.val} \leq 100$ 。

示例：

- **示例 1：**



输入：`head = [1, 2, 3, 4]`

输出：`[2, 1, 4, 3]`

- **示例 2：**

输入: head = []

输出: []

解题思路

思路 1：迭代

1. 创建一个哑节点 new_head，令 new_head.next = head。
2. 遍历链表，并判断当前链表后两位节点是否为空。如果后两个节点不为空，则使用三个指针：curr 指向当前节点，node1 指向下一个节点，node2 指向下面第二个节点。
3. 将 curr 指向 node2，node1 指向 node2 后边的节点，node2 指向 node1。则节点关系由 curr → node1 → node2 变为了 curr → node2 → node1。
4. 依次类推，最终返回哑节点连接的后一个节点。

思路 1：代码

```
class Solution:  
    def swapPairs(self, head: ListNode) -> ListNode:  
        new_head = ListNode(0)  
        new_head.next = head  
        curr = new_head  
        while curr.next and curr.next.next:  
            node1 = curr.next  
            node2 = curr.next.next  
            curr.next = node2  
            node1.next = node2.next  
            node2.next = node1  
            curr = node1  
        return new_head.next
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ ，其中 n 为链表的节点数量。
- 空间复杂度: $O(n)$ 。# 0025. K 个一组翻转链表
- 标签: 递归、链表
- 难度: 困难

题目链接

- [0025. K 个一组翻转链表 - 力扣](#)

题目大意

描述：给你链表的头节点 `head`，再给定一个正整数 `k`，`k` 的值小于或等于链表的长度。

要求：每 `k` 个节点一组进行翻转，并返回修改后的链表。如果链表节点总数不是 `k` 的整数倍，则将最后剩余的节点保持原有顺序。

说明：

- 不能只是单纯的改变节点内部的值，而是需要实际进行节点交换。
- 假设链表中的节点数目为 `n`。
- $1 \leq k \leq n \leq 5000$ 。
- $0 \leq \text{Node.val} \leq 1000$ 。
- 要求设计一个只用 $O(1)$ 额外内存空间的算法解决此问题。

示例：

- **示例 1：**

输入: `head = [1,2,3,4,5]`, `k = 2`

输出: `[2,1,4,3,5]`

解题思路

思路 1：迭代

在「[0206. 反转链表](#)」中我们可以通过迭代、递归两种方法将整个链表反转。而这道题要求以 `k` 为单位，对链表的区间进行反转。而区间反转其实就是「[0092. 反转链表 II](#)」这道题的题目要求。

本题中，我们可以以 `k` 为单位对链表进行切分，然后分别对每个区间部分进行反转。最后再返回头节点即可。

但是需要注意一点，如果需要反转的区间包含了链表的第一个节点，那么我们可以事先创建一个哑节点作为链表初始位置开始遍历，这样就能避免找不到需要反转的链表区间的前一个节点。

这道题的具体解题步骤如下：

1. 先使用哑节点 `dummy_head` 构造一个指向 `head` 的指针，避免找不到需要反转的链表区间的前一个节点。使用变量 `index` 记录当前元素的序号。
2. 使用两个指针 `cur`、`tail` 分别表示链表中待反转区间的首尾节点。初始 `cur` 赋值为 `dummy_head`，`tail` 赋值为 `dummy_head.next`，也就是 `head`。
3. 将 `tail` 向右移动，每移动一步，就令 `index` 加 1。
 - i. 当 `index % k != 0` 时，直接将 `tail` 向右移动，直到移动到当前待反转区间的结尾位置。
 - ii. 当 `index % k == 0` 时，说明 `tail` 已经移动到了当前待反转区间的结尾位置，此时调用 `cur = self.reverse(cur, tail.next)`，将待反转区间进行反转，并返回反转后区间的起始节点赋值给当前反转区间的首节点 `cur`。然后将 `tail` 移动到 `cur` 的下一个节点。
4. 最后返回新的头节点 `dummy_head.next`。

关于 `def reverse(self, head, tail):` 方法这里也说下具体步骤：

1. `head` 代表当前待反转区间的第一个节点的前一个节点，`tail` 代表当前待反转区间的最后一个节点的后一个节点。
2. 先用 `first` 保存一下待反转区间的第一个节点（反转之后为区间的尾节点），方便反转之后进行连接。
3. 我们使用两个指针 `cur` 和 `pre` 进行迭代。`pre` 指向 `cur` 前一个节点位置，即 `pre` 指向需要反转节点的前一个节点，`cur` 指向需要反转的节点。初始时，`pre` 指向待反转区间的第一个节点的前一个节点 `head`，`cur` 指向待反转区间的第一个节点，即 `pre.next`。
4. 当当前节点 `cur` 不等于 `tail` 时，将 `pre` 和 `cur` 的前后指针进行交换，指针更替顺序为：
 - i. 使用 `next` 指针保存当前节点 `cur` 的后一个节点，即 `next = cur.next`；
 - ii. 断开当前节点 `cur` 的后一节点链接，将 `cur` 的 `next` 指针指向前一节点 `pre`，即 `cur.next = pre`；
 - iii. `pre` 向前移动一步，移动到 `cur` 位置，即 `pre = cur`；
 - iv. `cur` 向前移动一步，移动到之前 `next` 指针保存的位置，即 `cur = next`。
5. 继续执行第 4 步中的 1、2、3、4 步。
6. 最后等到 `cur` 遍历到链表末尾（即 `cur == tail`）时，令「当前待反转区间的第一个节点的前一个节点」指向「反转区间后的头节点」，即 `head.next = pre`。令「待反转区间的第一个节点（反转之后为区间的尾节点）」指向「待反转分区间的最后一个节点的后一个节点」，即 `first.next = tail`。
7. 最后返回新的头节点 `dummy_head.next`。

思路 1：代码

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def reverse(self, head, tail):
        pre = head
        cur = pre.next
        first = cur
        while cur != tail:
            next = cur.next
            cur.next = pre
            pre = cur
            cur = next
        head.next = pre
        first.next = tail
        return first

    def reverseKGroup(self, head: ListNode, k: int) -> ListNode:
        dummy_head = ListNode(0)
        dummy_head.next = head
        cur = dummy_head
        tail = dummy_head.next
        index = 0
        while tail:
            index += 1
            if index % k == 0:
                cur = self.reverse(cur, tail.next)
                tail = cur.next
            else:
                tail = tail.next
        return dummy_head.next
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。其中 n 为链表的总长度。
- 空间复杂度： $O(1)$ 。

0026. 删 除有序数组中的重复项

- 标签：数组、双指针
- 难度：简单

题目链接

- 0026. 删 除有序数组中的重复项 - 力扣

题目大意

描述：给定一个有序数组 `nums`。

要求：删除数组 `nums` 中的重复元素，使每个元素只出现一次。并输出去除重复元素之后数组的长度。

说明：

- 不能使用额外的数组空间，在原地修改数组，并在使用 $O(1)$ 额外空间的条件下完成。

示例：

- **示例 1：**

输入: `nums = [1,1,2]`

输出: `2, nums = [1,2,_]`

解释: 函数应该返回新的长度 `2`，并且原数组 `nums` 的前两个元素被修改为 `1, 2`。不需要考虑数组中超出新长度的元素。

- **示例 2：**

输入: `nums = [0,0,1,1,1,2,2,3,3,4]`

输出: `5, nums = [0,1,2,3,4]`

解释: 函数应该返回新的长度 `5`，并且原数组 `nums` 的前五个元素被修改为 `0, 1, 2, 3, 4`。不需要考虑数组中超出新长度的元素。

解题思路

思路 1：快慢指针

因为数组是有序的，那么重复的元素一定会相邻。

删除重复元素，实际上就是将不重复的元素移到数组左侧。考虑使用双指针。具体算法如下：

1. 定义两个快慢指针 `slow`，`fast`。其中 `slow` 指向去除重复元素后的数组的末尾位置。`fast` 指向当前元素。
2. 令 `slow` 在后，`fast` 在前。令 `slow = 0`，`fast = 1`。
3. 比较 `slow` 位置上元素值和 `fast` 位置上元素值是否相等。
 - 如果不相等，则将 `slow` 后移一位，将 `fast` 指向位置的元素复制到 `slow` 位置上。
4. 将 `fast` 右移 1 位。
5. 重复上述 3~4 步，直到 `fast` 等于数组长度。
6. 返回 `slow + 1` 即为新数组长度。

思路 1：代码

```
class Solution:  
    def removeDuplicates(self, nums: List[int]) -> int:  
        if len(nums) <= 1:  
            return len(nums)  
  
        slow, fast = 0, 1  
  
        while (fast < len(nums)):  
            if nums[slow] != nums[fast]:  
                slow += 1  
                nums[slow] = nums[fast]  
            fast += 1  
  
        return slow + 1
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(1)$ 。

0027. 移除元素

- 标签：数组、双指针
- 难度：简单

题目链接

- 0027. 移除元素 - 力扣

题目大意

描述：给定一个数组 $nums$, 和一个值 val 。

要求：不使用额外数组空间，将数组中所有数值等于 val 值的元素移除掉，并且返回新数组的长度。

说明：

- $0 \leq nums.length \leq 100$ 。
- $0 \leq nums[i] \leq 50$ 。
- $0 \leq val \leq 100$ 。

示例：

- 示例 1：

输入: $nums = [3, 2, 2, 3]$, $val = 3$

输出: 2 , $nums = [2, 2]$

解释：函数应该返回新的长度 2 ，并且 $nums$ 中的前两个元素均为 2 。你不需要考虑数组中超出新长度后面的元素。1

- 示例 2：

输入: $nums = [0, 1, 2, 2, 3, 0, 4, 2]$, $val = 2$

输出: 5 , $nums = [0, 1, 4, 0, 3]$

解释：函数应该返回新的长度 5 ，并且 $nums$ 中的前五个元素为 $0, 1, 3, 0, 4$ 。注意这五个元素可为任意顺序。

解题思路

思路 1：快慢指针

1. 使用两个指针 $slow$, $fast$ 。 $slow$ 指向处理好的非 val 值元素数组的尾部, $fast$ 指针指向当前待处理元素。
2. 不断向右移动 $fast$ 指针，每次移动到非 val 值的元素，则将左右指针对应的数交换，交换同时将 $slow$ 右移。

- 这样就将非 val 值的元素进行前移， $slow$ 指针左边均为处理好的非 val 值元素，而从 $slow$ 指针指向的位置开始， $fast$ 指针左边都为 val 值。
- 遍历结束之后，则所有 val 值元素都移动到了右侧，且保持了非零数的相对位置。此时 $slow$ 就是新数组的长度。

思路 1：代码

```
class Solution:  
    def removeElement(self, nums: List[int], val: int) -> int:  
        slow = 0  
        fast = 0  
        while fast < len(nums):  
            if nums[fast] != val:  
                nums[slow], nums[fast] = nums[fast], nums[slow]  
                slow += 1  
            fast += 1  
        return slow
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(1)$ 。

0028. 找出字符串中第一个匹配项的下标

- 标签：双指针、字符串、字符串匹配
- 难度：中等

题目链接

- [0028. 找出字符串中第一个匹配项的下标 - 力扣](#)

题目大意

描述：给定两个字符串 $haystack$ 和 $needle$ 。

要求: 在 `haystack` 字符串中找出 `needle` 字符串出现的第一个位置 (从 `0` 开始)。如果不存在, 则返回 `-1`。

说明:

- 当 `needle` 为空字符串时, 返回 `0`。
- $1 \leq \text{haystack.length}, \text{needle.length} \leq 10^4$ 。
- `haystack` 和 `needle` 仅由小写英文字母组成。

示例:

- **示例 1:**

输入: `haystack = "hello", needle = "ll"`

输出: `2`

解释: "`sad`" 在下标 `0` 和 `6` 处匹配。第一个匹配项的下标是 `0`, 所以返回 `0`。

- **示例 2:**

输入: `haystack = "leetcode", needle = "leeto"`

输出: `-1`

解释: "`leeto`" 没有在 "`leetcode`" 中出现, 所以返回 `-1`。

解题思路

字符串匹配的经典题目。常见的字符串匹配算法有: BF (Brute Force) 算法、RK (Robin-Karp) 算法、KMP (Knuth Morris Pratt) 算法、BM (Boyer Moore) 算法、Horspool 算法、Sunday 算法等。

思路 1: BF (Brute Force) 算法

BF 算法思想: 对于给定文本串 T 与模式串 p , 从文本串的第一个字符开始与模式串 p 的第一个字符进行比较, 如果相等, 则继续逐个比较后续字符, 否则从文本串 T 的第二个字符起重新和模式串 p 进行比较。依次类推, 直到模式串 p 中每个字符依次与文本串 T 的一个连续子串相等, 则模式匹配成功。否则模式匹配失败。

BF 算法具体步骤如下:

1. 对于给定的文本串 T 与模式串 p , 求出文本串 T 的长度为 n , 模式串 p 的长度为 m 。
2. 同时遍历文本串 T 和模式串 p , 先将 $T[0]$ 与 $p[0]$ 进行比较。

- i. 如果相等，则继续比较 $T[1]$ 和 $p[1]$ 。以此类推，一直到模式串 p 的末尾 $p[m - 1]$ 为止。
 - ii. 如果不相等，则将文本串 T 移动到上次匹配开始位置的下一个字符位置，模式串 p 则回退到开始位置，再依次进行比较。
3. 当遍历完文本串 T 或者模式串 p 的时候停止搜索。

思路 1：代码

```
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        i = 0
        j = 0
        len1 = len(haystack)
        len2 = len(needle)

        while i < len1 and j < len2:
            if haystack[i] == needle[j]:
                i += 1
                j += 1
            else:
                i = i - (j - 1)
                j = 0

            if j == len2:
                return i - j
        else:
            return -1
```

思路 1：复杂度分析

- **时间复杂度**：平均时间复杂度为 $O(n + m)$ ，最坏时间复杂度为 $O(m \times n)$ 。其中文本串 T 的长度为 n ，模式串 p 的长度为 m 。
- **空间复杂度**： $O(1)$ 。

思路 2：RK (Robin Karp) 算法

RK 算法思想：对于给定文本串 T 与模式串 p ，通过滚动哈希算快速筛选出与模式串 p 不匹配的文本位置，然后在其余位置继续检查匹配项。

RK 算法具体步骤如下：

1. 对于给定的文本串 T 与模式串 p ，求出文本串 T 的长度为 n ，模式串 p 的长度为 m 。
2. 通过滚动哈希算法求出模式串 p 的哈希值 hash_p 。
3. 再通过滚动哈希算法对文本串 T 中 $n - m + 1$ 个子串分别求哈希值 hash_t 。
4. 然后逐个与模式串的哈希值比较大小。
 - i. 如果当前子串的哈希值 hash_t 与模式串的哈希值 hash_p 不同，则说明两者不匹配，则继续向后匹配。
 - ii. 如果当前子串的哈希值 hash_t 与模式串的哈希值 hash_p 相等，则验证当前子串和模式串的每个字符是否真的相等（避免哈希冲突）。
 - a. 如果当前子串和模式串的每个字符相等，则说明当前子串和模式串匹配。
 - b. 如果当前子串和模式串的每个字符不相等，则说明两者不匹配，继续向后匹配。
5. 比较到末尾，如果仍未成功匹配，则说明文本串 T 中不包含模式串 p ，方法返回 -1 。

思路 2：代码

```

class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        def rabinKarp(T: str, p: str) -> int:
            len1, len2 = len(T), len(p)

            hash_p = hash(p)
            for i in range(len1 - len2 + 1):
                hash_T = hash(T[i: i + len2])
                if hash_p != hash_T:
                    continue
                k = 0
                for j in range(len2):
                    if T[i + j] != p[j]:
                        break
                    k += 1
                if k == len2:
                    return i
            return -1
        return rabinKarp(haystack, needle)

```

思路 1：复杂度分析

- **时间复杂度：** $O(n)$ 。其中文本串 T 的长度为 n ，模式串 p 的长度为 m 。
- **空间复杂度：** $O(m)$ 。

思路 3：KMP（Knuth Morris Pratt）算法

KMP 算法思想：对于给定文本串 T 与模式串 p ，当发现文本串 T 的某个字符与模式串 p 不匹配的时候，可以利用匹配失败后的信息，尽量减少模式串与文本串的匹配次数，避免文本串位置的回退，以达到快速匹配的目的。

KMP 算法具体步骤如下：

1. 根据 next 数组的构造步骤生成「前缀表」 next 。
2. 使用两个指针 i 、 j ，其中 i 指向文本串中当前匹配的位置， j 指向模式串中当前匹配的位置。初始时， $i = 0$ ， $j = 0$ 。
3. 循环判断模式串前缀是否匹配成功，如果模式串前缀匹配不成功，将模式串进行回退，即 $j = \text{next}[j - 1]$ ，直到 $j == 0$ 时或前缀匹配成功时停止回退。
4. 如果当前模式串前缀匹配成功，则令模式串向右移动 1 位，即 $j += 1$ 。
5. 如果当前模式串 **完全** 匹配成功，则返回模式串 p 在文本串 T 中的开始位置，即 $i - j + 1$ 。
6. 如果还未完全匹配成功，则令文本串向右移动 1 位，即 $i += 1$ ，然后继续匹配。
7. 如果直到文本串遍历完也未完全匹配成功，则说明匹配失败，返回 -1 。

思路 3：代码

```
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        # KMP 匹配算法，T 为文本串，p 为模式串
        def kmp(T: str, p: str) -> int:
            n, m = len(T), len(p)

            next = generateNext(p) # 生成 next 数组

            i, j = 0, 0
            while i < n and j < m:
                if j == -1 or T[i] == p[j]:
                    i += 1
                    j += 1
                else:
                    j = next[j]
            if j == m:
                return i - j

            return -1

        # 生成 next 数组
        # next[i] 表示坏字符在模式串中最后一次出现的位置
        def generateNext(p: str):
            m = len(p)

            next = [-1 for _ in range(m)] # 初始化数组元素全部为 -1
            i, k = 0, -1
            while i < m - 1: # 生成下一个 next 元素
                if k == -1 or p[i] == p[k]:
                    i += 1
                    k += 1
                if p[i] == p[k]:
                    next[i] = next[k] # 设置 next 元素
                else:
                    next[i] = k # 退到更短相同前缀
                else:
                    k = next[k]
            return next

        return kmp(haystack, needle)
```

思路 3：复杂度分析

- **时间复杂度**: $O(n + m)$, 其中文本串 T 的长度为 n , 模式串 p 的长度为 m 。
- **空间复杂度**: $O(m)$ 。

思路 4：BM (Boyer Moore) 算法

BM 算法思想: 对于给定文本串 T 与模式串 p , 先对模式串 p 进行预处理。然后在匹配的过程中, 当发现文本串 T 的某个字符与模式串 p 不匹配的时候, 根据启发策略, 能够直接尽可能地跳过一些无法匹配的情况, 将模式串多向后滑动几位。

BM 算法具体步骤如下:

1. 计算出文本串 T 的长度为 n , 模式串 p 的长度为 m 。
2. 先对模式串 p 进行预处理, 生成坏字符位置表 bc_table 和好后缀规则后移位数表 gs_table 。
3. 将模式串 p 的头部与文本串 T 对齐, 将 i 指向文本串开始位置, 即 $i = 0$ 。 j 指向模式串末尾位置, 即 $j = m - 1$, 然后从模式串末尾位置开始进行逐位比较。
 - i. 如果文本串对应位置 $T[i + j]$ 上的字符与 $p[j]$ 相同, 则继续比较前一位字符。
 - a. 如果模式串全部匹配完毕, 则返回模式串 p 在文本串中的开始位置 i 。
 - ii. 如果文本串对应位置 $T[i + j]$ 上的字符与 $p[j]$ 不相同, 则:
 - a. 根据坏字符位置表计算出在「坏字符规则」下的移动距离 bad_move 。
 - b. 根据好后缀规则后移位数表计算出在「好后缀规则」下的移动距离 $good_move$ 。
 - c. 取两种移动距离的最大值, 然后对模式串进行移动, 即 $i += \max(bad_move, good_move)$ 。
4. 如果移动到末尾也没有找到匹配情况, 则返回 -1 。

思路 4：代码

```
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        def boyerMoore(T: str, p: str) -> int:
            n, m = len(T), len(p)

            bc_table = generateBadCharTable(p)          # 生成坏字符位置表
            gs_list = generateGoodSuffixList(p)         # 生成好后缀规则后移位数表

            i = 0
            while i <= n - m:
                j = m - 1
                while j > -1 and T[i + j] == p[j]:
                    j -= 1
                if j < 0:
                    return i
                bad_move = j - bc_table.get(T[i + j], -1)
                good_move = gs_list[j]
                i += max(bad_move, good_move)
            return -1

        # 生成坏字符位置表
        def generateBadCharTable(p: str):
            bc_table = dict()

            for i in range(len(p)):
                bc_table[p[i]] = i           # 坏字符在模式串中最后一次出现的位置
            return bc_table

        # 生成好后缀规则后移位数表
        def generateGoodSuffixList(p: str):
            m = len(p)
            gs_list = [m for _ in range(m)]
            suffix = generateSuffixArray(p)
            j = 0
            for i in range(m - 1, -1, -1):
                if suffix[i] == i + 1:
                    while j < m - 1 - i:
                        if gs_list[j] == m:
                            gs_list[j] = m - 1 - i
                        j += 1
```

```

        for i in range(m - 1):
            gs_list[m - 1 - suffix[i]] = m - 1 - i

    return gs_list

def generateSuffixArray(p: str):
    m = len(p)
    suffix = [m for _ in range(m)]
    for i in range(m - 2, -1, -1):
        start = i
        while start >= 0 and p[start] == p[m - 1 - i + start]:
            start -= 1
        suffix[i] = i - start
    return suffix

return boyerMoore(haystack, needle)

```

思路 4：复杂度分析

- **时间复杂度**: $O(n + \sigma)$, 其中文本串 T 的长度为 n , 字符集的大小是 σ 。
- **空间复杂度**: $O(m)$ 。其中模式串 p 的长度为 m 。

思路 5：Horspool 算法

Horspool 算法思想: 对于给定文本串 T 与模式串 p ，先对模式串 p 进行预处理。然后在匹配的过程中，当发现文本串 T 的某个字符与模式串 p 不匹配的时候，根据启发策略，能够尽可能的跳过一些无法匹配的情况，将模式串多向后滑动几位。

Horspool 算法具体步骤如下：

1. 计算出文本串 T 的长度为 n ，模式串 p 的长度为 m 。
2. 先对模式串 p 进行预处理，生成后移位数表 bc_table 。
3. 将模式串 p 的头部与文本串 T 对齐，将 i 指向文本串开始位置，即 $i = 0$ 。 j 指向模式串末尾位置，即 $j = m - 1$ ，然后从模式串末尾位置开始比较。
 - i. 如果文本串对应位置的字符 $T[i + j]$ 与模式串对应字符 $p[j]$ 相同，则继续比较前一位字符。
 - a. 如果模式串全部匹配完毕，则返回模式串 p 在文本串中的开始位置 i 。
 - ii. 如果文本串对应位置的字符 $T[i + j]$ 与模式串对应字符 $p[j]$ 不同，则：
 - a. 根据后移位数表 bc_table 和模式串末尾位置对应的文本串上的字符 $T[i + m - 1]$ ，计算出可移动距离 $bc_table[T[i + m - 1]]$ ，然后将模式串进行后移。
4. 如果移动到末尾也没有找到匹配情况，则返回 -1 。

思路 5：代码

```
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        def horspool(T: str, p: str) -> int:
            n, m = len(T), len(p)

            bc_table = generateBadCharTable(p)

            i = 0
            while i <= n - m:
                j = m - 1
                while j > -1 and T[i + j] == p[j]:
                    j -= 1
                if j < 0:
                    return i
                i += bc_table.get(T[i + m - 1], m)
            return -1

        # 生成后移位置表
        # bc_table[bad_char] 表示坏字符在模式串中最后一次出现的位置
        def generateBadCharTable(p: str):
            m = len(p)
            bc_table = dict()

            for i in range(m - 1):
                bc_table[p[i]] = m - i - 1      # 更新坏字符在模式串中最后一次出现的位置
            return bc_table

        return horspool(haystack, needle)
```

思路 5：复杂度分析

- 时间复杂度： $O(n)$ 。其中文本串 T 的长度为 n 。
- 空间复杂度： $O(m)$ 。其中模式串 p 的长度为 m 。

思路 6：Sunday 算法

Sunday 算法思想：对于给定文本串 T 与模式串 p ，先对模式串 p 进行预处理。然后在匹配的过程中，当发现文本串 T 的某个字符与模式串 p 不匹配的时候，根据启发策略，能够尽可能的跳过一些无法匹配的情况，将模式串多向后滑动几位。

Sunday 算法具体步骤如下：

1. 计算出文本串 T 的长度为 n ，模式串 p 的长度为 m 。
2. 先对模式串 p 进行预处理，生成后移位数表 bc_table 。
3. 将模式串 p 的头部与文本串 T 对齐，将 i 指向文本串开始位置，即 $i = 0$ 。 j 指向模式串末尾位置，即 $j = m - 1$ ，然后从模式串末尾位置开始比较。
 - i. 如果文本串对应位置的字符 $T[i + j]$ 与模式串对应字符 $p[j]$ 相同，则继续比较前一位字符。
 - a. 如果模式串全部匹配完毕，则返回模式串 p 在文本串中的开始位置 i 。
 - ii. 如果文本串对应位置的字符 $T[i + j]$ 与模式串对应字符 $p[j]$ 不同，则：
 - a. 根据后移位数表 bc_table 和模式串末尾位置对应的文本串上的字符 $T[i + m - 1]$ ，计算出可移动距离 $bc_table[T[i + m - 1]]$ ，然后将模式串进行后移。
4. 如果移动到末尾也没有找到匹配情况，则返回 -1 。

思路 6：代码

```
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        # sunday 算法, T 为文本串, p 为模式串
        def sunday(T: str, p: str) -> int:
            n, m = len(T), len(p)
            if m == 0:
                return 0

            bc_table = generateBadCharTable(p) # 生成后移位数表

            i = 0
            while i <= n - m:
                if T[i: i + m] == p:
                    return i # 匹配完成, 返回模式串 p 在文本串 T
                if i + m >= n:
                    return -1
                i += bc_table.get(T[i + m], m + 1) # 通过后移位数表, 向右进行进行快速移位
            return -1 # 匹配失败

        # 生成后移位数表
        # bc_table[bad_char] 表示遇到坏字符可以向右移动的距离
        def generateBadCharTable(p: str):
            m = len(p)
            bc_table = dict()

            for i in range(m):
                bc_table[p[i]] = m - i # 更新遇到坏字符可向右移动的距离
            return bc_table

        return sunday(haystack, needle)
```

思路 6：复杂度分析

- **时间复杂度:** $O(n)$ 。其中文本串 T 的长度为 n 。
- **空间复杂度:** $O(m)$ 。其中模式串 p 的长度为 m 。# 0029. 两数相除
- 标签: 位运算、数学
- 难度: 中等

题目链接

- 0029. 两数相除 - 力扣

题目大意

给定两个整数，被除数 dividend 和除数 divisor。要求返回两数相除的商，并且不能使用乘法、除法和取余运算。取值范围在 $[-2^{31}, 2^{31} - 1]$ 。如果结果溢出，则返回 $2^{31} - 1$ 。

解题思路

题目要求不能使用乘法、除法和取余运算。

可以把被除数和除数当做二进制，这样进行运算的时候，就可以通过移位运算来实现二进制的乘除。

- 先将除数不断左移，移位到位数大于或等于被除数。记录其移位次数 count。
- 然后再将除数右移 count 次，模拟二进制除法运算。
 - 如果当前被除数大于等于除数，则将 1 左移 count 位，即为当前位的商，并将其累加答案上。
 - 再用除数减去被除数，进行下一次运算。

代码

```
class Solution:
    def divide(self, dividend: int, divisor: int) -> int:
        MIN_INT, MAX_INT = -2147483648, 2147483647
        # 标记被除数和除数是否异号
        symbol = True if (dividend ^ divisor) < 0 else False
        # 将被除数和除数转换为正数处理
        if dividend < 0:
            dividend = -dividend
        if divisor < 0:
            divisor = -divisor

        # 除数不断左移，移位到位数大于或等于被除数
        count = 0
        while dividend >= divisor:
            count += 1
            divisor <<= 1

        # 向右移位，不断模拟二进制除法运算
        res = 0
        while count > 0:
            count -= 1
            divisor >>= 1
            if dividend >= divisor:
                res += (1 << count)
                dividend -= divisor
        if symbol:
            res = -res
        if MIN_INT <= res <= MAX_INT:
            return res
        else:
            return MAX_INT
```

0032. 最长有效括号

- 标签：栈、字符串、动态规划
- 难度：困难

题目链接

- [0032. 最长有效括号 - 力扣](#)

题目大意

描述：给定一个只包含 '(', ')' 的字符串。

要求：找出最长有效（格式正确且连续）括号子串的长度。

说明：

- $0 \leq s.length \leq 3 * 10^4$ 。
- $s[i]$ 为 '(' 或 ')'。

示例：

- **示例 1：**

输入: $s = "(\)"$

输出: 2

解释: 最长有效括号子串是 "()"

- **示例 2：**

输入: $s = ")()()$ "

输出: 4

解释: 最长有效括号子串是 "()()"

解题思路

思路 1：动态规划

1. 划分阶段

按照最长有效括号子串的结束位置进行阶段划分。

2. 定义状态

定义状态 $dp[i]$ 表示为：以字符 $s[i]$ 为结尾的最长有效括号的长度。

3. 状态转移方程

- 如果 $s[i] == '('$ ，此时以 $s[i]$ 结尾的子串不可能构成有效括号对，则 $dp[i] = 0$ 。
- 如果 $s[i] == ')'$ ，我们需要考虑 $s[i - 1]$ 来判断是否能够构成有效括号对。
 - 如果 $s[i - 1] == '('$ ，字符串形如()，此时 $s[i - 1]$ 与 $s[i]$ 为 ()，则：
 - $dp[i]$ 取决于「以字符 $s[i - 2]$ 为结尾的最长有效括号长度」 + 「 $s[i - 1]$ 与 $s[i]$ 构成的有效括号对长度 (2)」，即 $dp[i] = dp[i - 2] + 2$ 。
 - 特别地，如果 $s[i - 2]$ 不存在，即 $i - 2 < 0$ ，则 $dp[i]$ 直接取决于「 $s[i - 1]$ 与 $s[i]$ 构成的有效括号对长度 (2)」，即 $dp[i] = 2$ 。
 - 如果 $s[i - 1] == ')'$ ，字符串形如))，此时 $s[i - 1]$ 与 $s[i]$ 为))。那么以 $s[i - 1]$ 为结尾的最长有效长度为 $dp[i - 1]$ ，则我们需要看 $i - 1 - dp[i - 1]$ 位置上的字符 $s[i - 1 - dp[i - 1]]$ 是否与 $s[i]$ 匹配。
 - 如果 $s[i - 1 - dp[i - 1]] == '('$ ，则说明 $s[i - 1 - dp[i - 1]]$ 与 $s[i]$ 相匹配，此时我们需要看以 $s[i - 1 - dp[i - 1]]$ 的前一个字符 $s[i - 1 - dp[i - 2]]$ 为结尾的最长括号长度是多少，将其加上 $s[i - 1 - dp[i - 1]]$ 与 $s[i]$ ，从而构成更长的有效括号对：
 - $dp[i]$ 取决于「以字符 $s[i - 1]$ 为结尾的最长括号长度」 + 「以字符 $s[i - 1 - dp[i - 2]]$ 为结尾的最长括号长度」 + 「 $s[i - 1 - dp[i - 1]]$ 与 $s[i]$ 的长度 (2)」，即 $dp[i] = dp[i - 1] + dp[i - dp[i - 1] - 2] + 2$ 。
 - 特别地，如果 $s[i - dp[i - 1] - 2]$ 不存在，即 $i - dp[i - 1] - 2 < 0$ ，则 $dp[i]$ 直接取决于「以字符 $s[i - 1]$ 为结尾的最长括号长度」 + 「 $s[i - 1 - dp[i - 1]]$ 与 $s[i]$ 的长度 (2)」，即 $dp[i] = dp[i - 1] + 2$ 。

4. 初始条件

- 默认所有以字符 $s[i]$ 为结尾的最长有效括号的长度为 0，即 $dp[i] = 0$ 。

5. 最终结果

根据我们之前定义的状态， $dp[i]$ 表示为：以字符 $s[i]$ 为结尾的最长有效括号的长度。则最终结果为 $\max(dp[i])$ 。

思路 1：代码

```
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        dp = [0 for _ in range(len(s))]
        ans = 0
        for i in range(1, len(s)):
            if s[i] == '(':
                continue
            if s[i - 1] == ')':
                if i >= 2:
                    dp[i] = dp[i - 2] + 2
                else:
                    dp[i] = 2
            elif i - dp[i - 1] > 0 and s[i - dp[i - 1] - 1] == '(':
                if i - dp[i - 1] >= 2:
                    dp[i] = dp[i - 1] + dp[i - dp[i - 1] - 2] + 2
                else:
                    dp[i] = dp[i - 1] + 2
            ans = max(ans, dp[i])
        return ans
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ ，其中 n 为字符串长度。
- 空间复杂度： $O(n)$ 。

思路 2：栈

1. 定义一个变量 `ans` 用于维护最长有效括号的长度，初始时，`ans = 0`。
2. 定义一个栈用于判定括号对是否匹配（栈中存储的是括号的下标），栈底元素始终保持「最长有效括号子串的开始元素的前一个元素下标」。
3. 初始时，我们在栈中存储 `-1` 作为哨兵节点，表示「最长有效括号子串的开始元素的前一个元素下标为 `-1`」，即 `stack = [-1]`，
4. 然后从左至右遍历字符串。
 - i. 如果遇到左括号，即 `s[i] == '('`，则将其下标 `i` 压入栈，用于后续匹配右括号。
 - ii. 如果遇到右括号，即 `s[i] == ')'`，则将其与最近的左括号进行匹配（即栈顶元素），弹出栈顶元素，与当前右括号进行匹配。弹出之后：
 - a. 如果栈为空，则说明：

- a. 之前弹出的栈顶元素实际上是「最长有效括号子串的开始元素的前一个元素下标」，而不是左括号 (，此时无法完成合法匹配。
 - b. 将当前右括号的坐标 i 压入栈中，充当「下一个有效括号子串的开始元素前一个下标」。
 - b. 如果栈不为空，则说明：
 - a. 之前弹出的栈顶元素为左括号 (，此时可完成合法匹配。
 - b. 当前合法匹配的长度为「当前右括号的下标 i」 - 「最长有效括号子串的开始元素的前一个元素下标」。即 $i - \text{stack}[-1]$ 。
 - c. 更新最长匹配长度 ans 为 $\max(\text{ans}, i - \text{stack}[-1])$ 。
5. 遍历完输出答案 ans。

思路 2：代码

```
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        stack = [-1]
        ans = 0
        for i in range(len(s)):
            if s[i] == '(':
                stack.append(i)
            else:
                stack.pop()
                if stack:
                    ans = max(ans, i - stack[-1])
                else:
                    stack.append(i)
        return ans
```

思路 2：复杂度分析

- 时间复杂度： $O(n)$ ，其中 n 为字符串长度。
- 空间复杂度： $O(n)$ 。

参考资料

- 【题解】动态规划思路详解（C++）——32.最长有效括号
- 【题解】32. 最长有效括号 - 力扣（Leetcode）
- 【题解】【Nick~Hot一百题系列】超简单思路栈！# 0033. 搜索旋转排序数组
- 标签：数组、二分查找

- 难度：中等

题目链接

- [0033. 搜索旋转排序数组 - 力扣](#)

题目大意

描述：给定一个整数数组 $nums$ ，数组中值互不相同。给定的 $nums$ 是经过升序排列后的又进行了「旋转」操作的。再给定一个整数 $target$ 。

要求：从 $nums$ 中找到 $target$ 所在位置，如果找到，则返回对应下标，找不到则返回 -1 。

说明：

- 旋转操作：升序排列的数组 $nums$ 在预先未知的第 k 个位置进行了右移操作，变成了 $[nums[k]], nums[k + 1], \dots, nums[n - 1], \dots, nums[0], nums[1], \dots, nums[k - 1]$ 。

示例：

- **示例 1：**

输入: $nums = [4, 5, 6, 7, 0, 1, 2]$, $target = 0$

输出: 4

- **示例 2：**

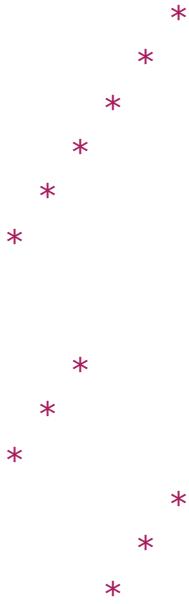
输入: $nums = [4, 5, 6, 7, 0, 1, 2]$, $target = 3$

输出: -1

解题思路

思路 1：二分查找

原本为升序排列的数组 $nums$ 经过「旋转」之后，会有两种情况，第一种就是原先的升序序列，另一种是两段升序的序列。



最直接的办法就是遍历一遍，找到目标值 $target$ 。但是还可以有更好的方法。考慮用二分查找来降低算法的时间复杂度。

我们将旋转后的数组看成左右两个升序部分：左半部分和右半部分。

有人会说第一种情况不是只有一个部分吗？其实我们可以把第一种情况中的整个数组看做是左半部分，然后右半部分为空数组。

然后创建两个指针 $left$ 、 $right$ ，分别指向数组首尾。让后计算出两个指针中间值 mid 。将 mid 与两个指针做比较，并考慮与 $target$ 的关系。

- 如果 $nums[mid] == target$, 说明找到了 $target$, 直接返回下标。
- 如果 $nums[mid] \geq nums[left]$, 则 mid 在左半部分 (因为右半部分值都比 $nums[left]$ 小)。
 - 如果 $nums[mid] \geq target$, 并且 $target \geq nums[left]$, 则 $target$ 在左半部分, 并且在 mid 左侧, 此时应将 $right$ 左移到 $mid - 1$ 位置。
 - 否则如果 $nums[mid] \leq target$, 则 $target$ 在左半部分, 并且在 mid 右侧, 此时应将 $left$ 右移到 $mid + 1$ 位置。
 - 否则如果 $nums[left] > target$, 则 $target$ 在右半部分, 应将 $left$ 移动到 $mid + 1$ 位置。
- 如果 $nums[mid] < nums[left]$, 则 mid 在右半部分 (因为右半部分值都比 $nums[left]$ 小)。
 - 如果 $nums[mid] < target$, 并且 $target \leq nums[right]$, 则 $target$ 在右半部分, 并且在 mid 右侧, 此时应将 $left$ 右移到 $mid + 1$ 位置。
 - 否则如果 $nums[mid] \geq target$, 则 $target$ 在右半部分, 并且在 mid 左侧, 此时应将 $right$ 左移到 $mid - 1$ 位置。

- 否则如果 $nums[right] < target$, 则 $target$ 在左半部分, 应将 $right$ 左移到 $mid - 1$ 位置。

思路 1：代码

```
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        left = 0
        right = len(nums) - 1
        while left <= right:
            mid = left + (right - left) // 2
            if nums[mid] == target:
                return mid

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

        return -1
```

思路 1：复杂度分析

- **时间复杂度**: $O(\log n)$ 。二分查找算法的时间复杂度为 $O(\log n)$ 。
- **空间复杂度**: $O(1)$ 。只用到了常数空间存放若干变量。

0034. 在排序数组中查找元素的第一个和最后一个位置

- 标签: 数组、二分查找
- 难度: 中等

题目链接

- 0034. 在排序数组中查找元素的第一个和最后一个位置 - 力扣

题目大意

描述：给你一个按照非递减顺序排列的整数数组 `nums`，和一个目标值 `target`。

要求：找出给定目标值在数组中的开始位置和结束位置。

说明：

- 要求使用时间复杂度为 $O(\log n)$ 的算法解决问题。

示例：

- **示例 1：**

输入: `nums = [5, 7, 7, 8, 8, 10]`, `target = 8`

输出: `[3, 4]`

- **示例 2：**

输入: `nums = [5, 7, 7, 8, 8, 10]`, `target = 6`

输出: `[-1, -1]`

解题思路

思路 1：二分查找

要求使用时间复杂度为 $O(\log n)$ 的算法解决问题，那么就需要使用「二分查找算法」了。

- 进行两次二分查找，第一次尽量向左搜索。第二次尽量向右搜索。

思路 1：代码

```
class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        ans = [-1, -1]
        n = len(nums)
        if n == 0:
            return ans

        left = 0
        right = n - 1
        while left < right:
            mid = left + (right - left) // 2
            if nums[mid] < target:
                left = mid + 1
            else:
                right = mid

        if nums[left] != target:
            return ans

        ans[0] = left

        left = 0
        right = n - 1
        while left < right:
            mid = left + (right - left + 1) // 2
            if nums[mid] > target:
                right = mid - 1
            else:
                left = mid

        if nums[left] == target:
            ans[1] = left

        return ans
```

思路 1：复杂度分析

- 时间复杂度： $O(\log_2 n)$ 。
- 空间复杂度： $O(1)$ 。# 0035. 搜索插入位置
- 标签：数组、二分查找

- 难度：简单

题目链接

- [0035. 搜索插入位置 - 力扣](#)

题目大意

描述：给定一个排好序的数组 $nums$ ，以及一个目标值 $target$ 。

要求：在数组中找到目标值，并返回下标。如果找不到，则返回目标值按顺序插入数组的位置。

说明：

- $1 \leq nums.length \leq 10^4$ 。
- $-10^4 \leq nums[i] \leq 10^4$ 。
- $nums$ 为无重复元素的升序排列数组。
- $-10^4 \leq target \leq 10^4$ 。

示例：

- **示例 1：**

输入: `nums = [1,3,5,6], target = 5`

输出: `2`

解题思路

思路 1：二分查找

设定左右节点为数组两端，即 $left = 0$ ， $right = len(nums) - 1$ ，代表待查找区间为 $[left, right]$ （左闭右闭）。

取两个节点中心位置 mid ，先比较中心位置值 $nums[mid]$ 与目标值 $target$ 的大小。

- 如果 $target == nums[mid]$ ，则当前中心位置为待插入数组的位置。
- 如果 $target > nums[mid]$ ，则将左节点设置为 $mid + 1$ ，然后继续在右区间 $[mid + 1, right]$ 搜索。

- 如果 $target < \text{nums}[\text{mid}]$, 则将右节点设置为 $\text{mid} - 1$, 然后继续在左区间 $[\text{left}, \text{mid} - 1]$ 搜索。

直到查找到目标值返回待插入数组的位置, 或者等到 $\text{left} > \text{right}$ 时停止查找, 此时 left 所在位置就是待插入数组的位置。

思路 1: 二分查找代码

```
class Solution:
    def searchInsert(self, nums: List[int], target: int) -> int:
        size = len(nums)
        left, right = 0, size - 1

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

        return left
```

思路 1: 复杂度分析

- **时间复杂度:** $O(\log n)$ 。二分查找算法的时间复杂度为 $O(\log n)$ 。
- **空间复杂度:** $O(1)$ 。只用到了常数空间存放若干变量。

0036. 有效的数独

- 标签: 数组、哈希表、矩阵
- 难度: 中等

题目链接

- [0036. 有效的数独 - 力扣](#)

题目大意

描述：给定一个数独，用 $9 * 9$ 的二维字符数组 `board` 来表示，其中，未填入的空白用 “.” 代替。

要求：判断该数独是否是一个有效的数独。

说明：

- 一个有效的数独（部分已被填充）不一定是可解的。
- 只需要根据以上规则，验证已经填入的数字是否有效即可。
- 空白格用 ‘.’ 表示。

一个有效的数独需满足：

1. 数字 1-9 在每一行只能出现一次。
2. 数字 1-9 在每一列只能出现一次。
3. 数字 1-9 在每一个以粗实线分隔的 $3 * 3$ 宫内只能出现一次。（请参考示例图）

示例：

- **示例 1：**

5	3	.	.	7
6	.	.	1	9	5	.	.	.
.	9	8	.	.	.	6	.	.
8	.	.	6	3
4	.	8	.	3	.	.	.	1
7	.	.	2	.	.	.	6	.
.	6	.	.	.	2	8	.	.
.	.	4	1	9	.	.	.	5
.	.	.	8	.	.	7	9	.

```
输入: board =
[[5,"3",".",".","7",".",".",".","."]
,[6,".",".","1","9","5",".",".","."]
,[],"9","8",".",".",".","6","."]
,[8,".",".","6",".",".",".","3"]
,[4,".",".","8",".","3",".",".","1"]
,[7,".",".","2",".",".",".","6"]
,[],"6",".",".","2","8","."]
,[],"4","1","9",".",".","5"]
,[],"8",".","7","9"]
输出: True
```

解题思路

思路 1：哈希表

判断数独有效，需要分别看每一行、每一列、每一个 3×3 的小方格是否出现了重复数字，如果都没有出现重复数字就是一个有效的数独，如果出现了重复数字则不是有效的数独。

- 用 3 个 9×9 的数组分别来表示该数字是否在所在的行，所在的列，所在的方格出现过。其中方格角标的计算用 `box[(i / 3) * 3 + (j / 3)][n]` 来表示。
- 双重循环遍历数独矩阵。如果对应位置上的数字如果已经在所在的行 / 列 / 方格出现过，则返回 `False`。
- 遍历完没有重复出现，则返回 `True`。

思路 1：代码

```
class Solution:
    def isValidSudoku(self, board: List[List[str]]) -> bool:
        rows_map = [dict() for _ in range(9)]
        cols_map = [dict() for _ in range(9)]
        boxes_map = [dict() for _ in range(9)]

        for i in range(9):
            for j in range(9):
                if board[i][j] == '.':
                    continue
                num = int(board[i][j])
                box_index = (i // 3) * 3 + j // 3
                row_num = rows_map[i].get(num, 0)
                col_num = cols_map[j].get(num, 0)
                box_num = boxes_map[box_index].get(num, 0)
                if row_num > 0 or col_num > 0 or box_num > 0:
                    return False
                rows_map[i][num] = 1
                cols_map[j][num] = 1
                boxes_map[box_index][num] = 1

        return True
```

思路 1：复杂度分析

- **时间复杂度**: $O(1)$ 。数独总共 81 个单元格，对每个单元格遍历一次，可以看做是常数级的时间复杂度。
- **空间复杂度**: $O(1)$ 。使用 81 个单位空间，可以看做是常数级的空间复杂度。

0037. 解数独

- 标签：数组、哈希表、回溯、矩阵
- 难度：困难

题目链接

- [0037. 解数独 - 力扣](#)

题目大意

描述：给定一个二维的字符数组 *board* 用来表示数独，其中数字 1 ~ 9 表示该位置已经填入了数字，‘.’ 表示该位置还没有填入数字。

要求：现在编写一个程序，通过填充空格的方式来解决数独问题，最终不用返回答案，将题目给定 *board* 修改为可行的方案即可。

说明：

- 数独解法需遵循如下规则：
 - 数字 1 ~ 9 在每一行只能出现一次。
 - 数字 1 ~ 9 在每一列只能出现一次。
 - 数字 1 ~ 9 在每一个以粗直线分隔的 3×3 宫格内只能出现一次。
- *board.length == 9*。
- *board[i].length == 9*。
- *board[i][j]* 是一位数字或者 ‘.’。
- 题目数据保证输入数独仅有一个解。

示例：

- 示例 1：

5	3	.	.	7
6	.	.	1	9	5	.	.	.
.	9	8	6	.
8	.	.	.	6	.	.	.	3
4	.	.	8	.	3	.	.	1
7	.	.	.	2	.	.	.	6
.	6	2	8	.
.	.	4	1	9	.	.	.	5
.	.	.	8	.	.	7	9	.

输入： *board* = [[“5”, “3”, “.”, “.”, “7”, “.”, “.”, “.”, “.”], [“6”, “.”, “.”, “.”, “1”, “9”, “5”, “.”, “.”, “.”]]

输出： [[“5”, “3”, “4”, “6”, “7”, “8”, “9”, “1”, “2”], [“6”, “7”, “2”, “1”, “9”, “5”, “3”, “4”, “8”], [“1”,

解释： 输入的数独如上图所示，唯一有效的解决方案如下所示：

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

解题思路

思路 1：回溯算法

对于每一行、每一列、每一个数字，都需要一重 `for` 循环来遍历，这样就是三重 `for` 循环。

对于第 i 行、第 j 列的元素来说，如果当前位置为空位，则尝试将第 k 个数字置于此处，并检验数独的有效性。如果有效，则继续遍历下一个空位，直到遍历完所有空位，得到可行方案或者遍历失败时结束。

遍历完下一个空位之后再将此位置进行回退，置为 `.`。

思路 1：代码

```
class Solution:
    def backtrack(self, board: List[List[str]]):
        for i in range(len(board)):
            for j in range(len(board[0])):
                if board[i][j] != '.':
                    continue
                for k in range(1, 10):
                    if self.isValid(i, j, k, board):
                        board[i][j] = str(k)
                        if self.backtrack(board):
                            return True
                        board[i][j] = '.'
        return False
    return True

    def isValid(self, row: int, col: int, val: int, board: List[List[str]]) -> bool:
        for i in range(0, 9):
            if board[row][i] == str(val):
                return False

        for j in range(0, 9):
            if board[j][col] == str(val):
                return False

        start_row = (row // 3) * 3
        start_col = (col // 3) * 3

        for i in range(start_row, start_row + 3):
            for j in range(start_col, start_col + 3):
                if board[i][j] == str(val):
                    return False
        return True

    def solveSudoku(self, board: List[List[str]]) -> None:
        self.backtrack(board)
        ....
        Do not return anything, modify board in-place instead.
        ....
```

思路 1：复杂度分析

- 时间复杂度： $O(9^m)$, m 为棋盘中 ‘.’ 的数量。
- 空间复杂度： $O(9^2)$ 。

0038. 外观数列

- 标签：字符串
- 难度：中等

题目链接

- [0038. 外观数列 - 力扣](#)

题目大意

给定一个正整数 n , ($1 \leq n \leq 30$), 要求输出外观数列的第 n 项。

外观数列：整数序列，数字由 1 开始，每一项都是对前一项的描述

例如：

1.	1	由 1 开始
2.	11	表示 1 个 1
3.	21	表示 2 个 1
4.	1211	表示 1 个 1, 1 个 2

解题思路

模拟题目遍历求解。

将 ans 设为 "1", 每次遍历判断相邻且相同的数字有多少个, 再将 ans 拼接上「数字个数 + 数字」。

代码

```
class Solution:
    def countAndSay(self, n: int) -> str:
        ans = "1"

        for _ in range(1, n):
            s = ""
            start = 0
            for i in range(len(ans)):
                if ans[i] != ans[start]:
                    s += str(i-start) + ans[start]
                    start = i
                s += str(len(ans)-start) + ans[start]
            ans = s
        return ans
```

0039. 组合总和

- 标签：数组、回溯
- 难度：中等

题目链接

- [0039. 组合总和 - 力扣](#)

题目大意

描述：给定一个无重复元素的正整数数组 `candidates` 和一个正整数 `target`。

要求：找出 `candidates` 中所有可以使数字和为目标数 `target` 的所有不同组合，并以列表形式返回。可以按照任意顺序返回这些组合。

说明：

- 数组 `candidates` 中的数字可以无限重复选取。
- 如果至少一个数字的被选数量不同，则两种组合是不同的。
- $1 \leq candidates.length \leq 30$ 。

- $2 \leq candidates[i] \leq 40$ 。
- `candidates` 的所有元素互不相同。
- $1 \leq target \leq 40$ 。

示例：

- 示例 1：

输入：`candidates = [2,3,6,7], target = 7`

输出：`[[2,2,3],[7]]`

解释：

`2` 和 `3` 可以形成一组候选，`2 + 2 + 3 = 7`。注意 `2` 可以使用多次。

`7` 也是一个候选，`7 = 7`。

仅有这两种组合。

- 示例 2：

输入：`candidates = [2,3,5], target = 8`

输出：`[[2,2,2,2],[2,3,3],[3,5]]`

解题思路

思路 1：回溯算法

定义回溯方法，`start_index = 1` 开始进行回溯。

- 如果 `sum > target`，则直接返回。
- 如果 `sum == target`，则将 `path` 中的元素加入到 `res` 数组中。
- 然后对 `[start_index, n]` 范围内的数进行遍历取值。
 - 如果 `sum + candidates[i] > target`，可以直接跳出循环。
 - 将和累积，即 `sum += candidates[i]`，然后将当前元素 `i` 加入 `path` 数组。
 - 递归遍历 `[start_index, n]` 上的数。
 - 加之前的和回退，即 `sum -= candidates[i]`，然后将遍历的 `i` 元素进行回退。
- 最终返回 `res` 数组。

根据回溯算法三步走，写出对应的回溯算法。

1. **明确所有选择：**一个组合每个位置上的元素都可以从剩余可选元素中选出。
2. **明确终止条件：**

- 当遍历到决策树的叶子节点时，就终止了。即当前路径搜索到末尾时，递归终止。

3. 将决策树和终止条件翻译成代码：

i. 定义回溯函数：

- `backtrack(total, start_index)`: 函数的传入参数是 `total` (当前和)、`start_index` (剩余可选元素开始位置)，全局变量是 `res` (存放所有符合条件结果的集合数组) 和 `path` (存放当前符合条件的结果)。
 - `backtrack(total, start_index)`: 函数代表的含义是：当前组合和为 `total`，递归从 `candidates` 的 `start_index` 位置开始，选择剩下的元素。

ii. 书写回溯函数主体 (给出选择元素、递归搜索、撤销选择部分)。

- 从目前正在考虑元素，到数组结束为止，枚举出所有可选的元素。对于每一个可选元素：
 - 约束条件：之前已经选择的元素不再重复选用，只能从剩余元素中选择。
 - 选择元素：将其添加到当前数组 `path` 中。
 - 递归搜索：在选择该元素的情况下，继续递归选择剩下元素。
 - 撤销选择：将该元素从当前结果数组 `path` 中移除。

```

for i in range(start_index, len(candidates)):
    if total + candidates[i] > target:
        break

    total += candidates[i]
    path.append(candidates[i])
    backtrack(total, i)
    total -= candidates[i]
    path.pop()
  
```

iii. 明确递归终止条件 (给出递归终止条件，以及递归终止时的处理方法)。

- 当不可能再出现解 (`total > target`)，或者遍历到决策树的叶子节点时 (`total == target`) 时，就终止了。
- 当遍历到决策树的叶子节点时 (`total == target`) 时，将当前结果的数组 `path` 放入答案数组 `res` 中，递归停止。

思路 1：代码

```
class Solution:
    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        res = []
        path = []
        def backtrack(total, start_index):
            if total > target:
                return

            if total == target:
                res.append(path[:])
                return

            for i in range(start_index, len(candidates)):
                if total + candidates[i] > target:
                    break

                total += candidates[i]
                path.append(candidates[i])
                backtrack(total, i)
                total -= candidates[i]
                path.pop()

        candidates.sort()
        backtrack(0, 0)
        return res
```

思路 1：复杂度分析

- **时间复杂度**: $O(2^n \times n)$, 其中 n 是数组 `candidates` 的元素个数, 2^n 指的是所有状态数。
- **空间复杂度**: $O(target)$, 递归函数需要用到栈空间, 栈空间取决于递归深度, 最坏情况下递归深度为 $O(target)$, 所以空间复杂度为 $O(target)$ 。

0040. 组合总和 II

- 标签: 数组、回溯
- 难度: 中等

题目链接

- [0040. 组合总和 II - 力扣](#)

题目大意

描述：给定一个数组 `candidates` 和一个目标数 `target`。

要求：找出 `candidates` 中所有可以使数字和为目标数 `target` 的组合。

说明：

- 数组 `candidates` 中的数字在每个组合中只能使用一次。
- $1 \leq candidates.length \leq 100$ 。
- $1 \leq candidates[i] \leq 50$ 。

示例：

- **示例 1：**

输入: `candidates = [10,1,2,7,6,1,5]`, `target = 8`,

输出:

```
[  
[1,1,6],  
[1,2,5],  
[1,7],  
[2,6]  
]
```

- **示例 2：**

输入: `candidates = [2,5,2,1,2]`, `target = 5`,

输出:

```
[  
[1,2,2],  
[5]  
]
```

解题思路

思路 1：回溯算法

跟「0039. 组合总和」不一样的地方在于本题不能有重复组合，所以关键步骤在于去重。

在回溯遍历的时候，下一层递归的 `start_index` 要从当前节点的后一位开始遍历，即 `i + 1` 位开始。而且统一递归层不能使用相同的元素，即需要增加一句判断

```
if i > start_index and candidates[i] == candidates[i - 1]: continue .
```

思路 1：代码

```
class Solution:
    res = []
    path = []
    def backtrack(self, candidates: List[int], target: int, sum: int, start_index: int):
        if sum > target:
            return
        if sum == target:
            self.res.append(self.path[:])
            return

        for i in range(start_index, len(candidates)):
            if sum + candidates[i] > target:
                break
            if i > start_index and candidates[i] == candidates[i - 1]:
                continue
            sum += candidates[i]
            self.path.append(candidates[i])
            self.backtrack(candidates, target, sum, i + 1)
            sum -= candidates[i]
            self.path.pop()

    def combinationSum2(self, candidates: List[int], target: int) -> List[List[int]]:
        self.res.clear()
        self.path.clear()
        candidates.sort()
        self.backtrack(candidates, target, 0, 0)
        return self.res
```

思路 1：复杂度分析

- **时间复杂度**: $O(2^n \times n)$, 其中 n 是数组 `candidates` 的元素个数, 2^n 指的是所有状态数。
- **空间复杂度**: $O(target)$, 递归函数需要用到栈空间, 栈空间取决于递归深度, 最坏情况下递归深度为 $O(target)$, 所以空间复杂度为 $O(target)$ 。

0041. 缺失的第一个正数

- 标签: 数组、哈希表
- 难度: 困难

题目链接

- [0041. 缺失的第一个正数 - 力扣](#)

题目大意

描述: 给定一个未排序的整数数组 `nums`。

要求: 找出其中没有出现的最小的正整数。

说明:

- $1 \leq \text{nums.length} \leq 5 * 10^5$ 。
- $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$ 。
- 要求实现时间复杂度为 $O(n)$ 并且只使用常数级别额外空间的解决方案。

示例:

- **示例 1:**

输入: `nums = [1, 2, 0]`

输出: `3`

- **示例 2:**

输入: `nums = [3, 4, -1, 1]`

输出: `2`

解题思路

思路 1：哈希表、原地哈希

如果使用普通的哈希表，我们只需要遍历一遍数组，将对应整数存入到哈希表中，再从 1 开始，依次判断对应正数是否在哈希表中即可。但是这种做法的空间复杂度为 $O(n)$ ，不满足常数级别的额外空间要求。

我们可以将当前数组视为哈希表。一个长度为 n 的数组，对应存储的元素值应该为 $[1, n + 1]$ 之间，其中还包含一个缺失的元素。

1. 我们可以遍历一遍数组，将当前元素放到其对应位置上（比如元素值为 1 的元素放到数组第 0 个位置上、元素值为 2 的元素放到数组第 1 个位置上，等等）。
2. 然后再次遍历一遍数组。遇到第一个元素值不等于下标 + 1 的元素，就是答案要求的缺失的第一个正数。
3. 如果遍历完没有在数组中找到缺失的第一个正数，则缺失的第一个正数是 $n + 1$ 。
4. 最后返回我们找到的缺失的第一个正数。

思路 1：代码

```
class Solution:  
    def firstMissingPositive(self, nums: List[int]) -> int:  
        size = len(nums)  
  
        for i in range(size):  
            while 1 <= nums[i] <= size and nums[i] != nums[nums[i] - 1]:  
                index1 = i  
                index2 = nums[i] - 1  
                nums[index1], nums[index2] = nums[index2], nums[index1]  
  
        for i in range(size):  
            if nums[i] != i + 1:  
                return i + 1  
        return size + 1
```

思路 1：复杂度分析

- **时间复杂度：** $O(n)$ ，其中 n 为数组 nums 的元素个数。
- **空间复杂度：** $O(1)$ 。

0042. 接雨水

- 标签：栈、数组、双指针、动态规划、单调栈
- 难度：困难

题目链接

- [0042. 接雨水 - 力扣](#)

题目大意

描述：给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，用数组 `height` 表示，其中 `height[i]` 表示第 i 根柱子的高度。

要求：计算按此排列的柱子，下雨之后能接多少雨水。

说明：

- $n == height.length$ 。
- $1 \leq n \leq 2 * 10^4$ 。
- $0 \leq height[i] \leq 10^5$ 。

示例：

- **示例 1：**



输入：`height = [0,1,0,2,1,0,1,3,2,1,2,1]`

输出：`6`

解释：上面是由数组 `[0,1,0,2,1,0,1,3,2,1,2,1]` 表示的高度图，在这种情况下，可以接 `6` 个单位的雨水（蓝色部分）。

- **示例 2：**

输入: height = [4,2,0,3,2,5]

输出: 9

解题思路

思路 1：单调栈

1. 遍历高度数组 height。
2. 如果当前柱体高度较小，小于等于栈顶柱体的高度，则将当前柱子高度入栈。
3. 如果当前柱体高度较大，大于栈顶柱体的高度，则一直出栈，直到当前柱体小于等于栈顶柱体的高度。
4. 假设当前柱体为 C，出栈柱体为 B，出栈之后新的栈顶柱体为 A。则说明：
 - i. 当前柱体 C 是出栈柱体 B 向右找到的第一个大于当前柱体高度的柱体，那么以出栈柱体 B 为中心，可以向右将宽度扩展到当前柱体 C。
 - ii. 新的栈顶柱体 A 是出栈柱体 B 向左找到的第一个大于当前柱体高度的柱体，那么以出栈柱体 B 为中心，可以向左将宽度扩展到当前柱体 A。
5. 出栈后，以新的栈顶柱体 A 为左边界，以当前柱体 C 为右边界，以左右边界与出栈柱体 B 的高度差为深度，计算可以接到雨水的面积。然后记录并更新累积面积。

思路 1：代码

```
class Solution:  
    def trap(self, height: List[int]) -> int:  
        ans = 0  
        stack = []  
        size = len(height)  
        for i in range(size):  
            while stack and height[i] > height[stack[-1]]:  
                cur = stack.pop(-1)  
                if stack:  
                    left = stack[-1] + 1  
                    right = i - 1  
                    high = min(height[i], height[stack[-1]]) - height[cur]  
                    ans += high * (right - left + 1)  
                else:  
                    break  
            stack.append(i)  
        return ans
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$, 其中 n 是数组 `height` 的长度。
- 空间复杂度: $O(n)$ 。# 0043. 字符串相乘
- 标签: 数学、字符串、模拟
- 难度: 中等

题目链接

- [0043. 字符串相乘 - 力扣](#)

题目大意

描述: 给定两个以字符串形式表示的非负整数 `num1` 和 `num2`。

要求: 返回 `num1` 和 `num2` 的乘积，它们的乘积也表示为字符串形式。

说明:

- 不能使用任何标准库的大数类型（比如 `BigInteger`）或直接将输入转换为整数来处理。
- $1 \leq num1.length, num2.length \leq 200$ 。
- `num1` 和 `num2` 只能由数字组成。
- `num1` 和 `num2` 都不包含任何前导零，除了数字0本身。

示例:

- **示例 1:**

输入: `num1 = "2", num2 = "3"`

输出: `"6"`

- **示例 2:**

输入: `num1 = "123", num2 = "456"`

输出: `"56088"`

解题思路

思路 1：模拟

我们可以使用数组来模拟大数乘法。长度为 `len(num1)` 的整数 `num1` 与长度为 `len(num2)` 的整数 `num2` 相乘的结果长度为 `len(num1) + len(num2) - 1` 或 `len(num1) + len(num2)`。所以我们可以使用长度为 `len(num1) + len(num2)` 的整数数组 `nums` 来存储两个整数相乘之后的结果。

整个计算流程的步骤如下：

1. 从个位数字由低位到高位开始遍历 `num1`，取得每一位数字 `digit1`。从个位数字由低位到高位开始遍历 `num2`，取得每一位数字 `digit2`。
2. 将 `digit1 * digit2` 的结果累积存储到 `nums` 对应位置 `i + j + 1` 上。
3. 计算完毕之后从 `len(num1) + len(num2) - 1` 的位置由低位到高位遍历数组 `nums`。将每个数位上大于等于 10 的数字进行进位操作，然后对该位置上的数字进行取余操作。
4. 最后判断首位是否有进位。如果首位为 0，则从第 1 个位置开始将答案数组拼接成字符串。如果首位不为 0，则从第 0 个位置开始将答案数组拼接成字符串。并返回答案字符串。

思路 1：代码

```
class Solution:
    def multiply(self, num1: str, num2: str) -> str:
        if num1 == "0" or num2 == "0":
            return "0"

        len1, len2 = len(num1), len(num2)
        nums = [0 for _ in range(len1 + len2)]

        for i in range(len1 - 1, -1, -1):
            digit1 = int(num1[i])
            for j in range(len2 - 1, -1, -1):
                digit2 = int(num2[j])
                nums[i + j + 1] += digit1 * digit2

        for i in range(len1 + len2 - 1, 0, -1):
            nums[i - 1] += nums[i] // 10
            nums[i] %= 10

        if nums[0] == 0:
            ans = "".join(str(digit) for digit in nums[1:])
        else:
            ans = "".join(str(digit) for digit in nums[:])
        return ans
```

思路 1：复杂度分析

- 时间复杂度： $O(m \times n)$ ，其中 m 和 n 分别为 nums1 和 nums2 的长度。
- 空间复杂度： $O(m + n)$ 。

0044. 通配符匹配

- 标签：贪心、递归、字符串、动态规划
- 难度：困难

题目链接

- [0044. 通配符匹配 - 力扣](#)

题目大意

描述：给定一个字符串 s 和一个字符模式串 p 。

要求：实现一个支持 ' $?$ ' 和 ' $*$ ' 的通配符匹配。两个字符串完全匹配才算匹配成功。如果匹配成功，则返回 `True`，否则返回 `False`。

- ' $?$ ' 可以匹配任何单个字符。
- ' $*$ ' 可以匹配任意字符串（包括空字符串）。

说明：

- s 可能为空，且只包含从 $a \sim z$ 的小写字母。
- p 可能为空，且只包含从 $a \sim z$ 的小写字母，以及字符 ' $?$ ' 和 ' $*$ '。

示例：

- **示例 1：**

输入: $s = "aa"$ $p = "a"$

输出: `False`

解释: " a " 无法匹配 " aa " 整个字符串。

- **示例 2：**

输入: $s = "aa"$ $p = "*"$

输出: `True`

解释: '*' 可以匹配任意字符串。

解题思路

思路 1：动态规划

1. 划分阶段

按照两个字符串的结尾位置进行阶段划分。

2. 定义状态

定义状态 $dp[i][j]$ 表示为：字符串 s 的前 i 个字符与字符串 p 的前 j 个字符是否匹配。

3. 状态转移方程

- 如果 $s[i - 1] == p[j - 1]$ ，或者 $p[j - 1] == '?'$ ，则表示字符串 s 的第 i 个字符与字符串 p 的第 j 个字符是匹配的。此时「字符串 s 的前 i 个字符与字符串 p 的前 j 个字符是否匹配」取决于「字符串 s 的前 $i - 1$ 个字符与字符串 p 的前 $j - 1$ 个字符是否匹配」。即 $dp[i][j] = dp[i - 1][j - 1]$ 。
- 如果 $p[j - 1] == '*'$ ，则字符串 p 的第 j 个字符可以对应字符串 s 中 $0 \sim$ 若干个字符。则：
 - 如果当前星号没有匹配当前第 i 个字符，则「字符串 s 的前 i 个字符与字符串 p 的前 j 个字符是否匹配」取决于「字符串 s 的前 $i - 1$ 个字符与字符串 p 的前 j 个字符是否匹配」，即 $dp[i][j] = dp[i - 1][j]$ 。
 - 如果当前星号匹配了当前第 i 个字符，则「字符串 s 的前 i 个字符与字符串 p 的前 j 个字符是否匹配」取决于「字符串 s 的前 i 个字符与字符串 p 的前 $j - 1$ 个字符是否匹配」，即 $dp[i][j] = dp[i][j - 1]$ 。
 - 这两种情况只需匹配一种，就视为匹配，所以 $dp[i][j] = dp[i - 1][j] \text{ or } dp[i][j - 1]$ 。

则动态转移方程为：

**ParseError: KaTeX parse error: Undefined control sequence: \or at position 66: ...1] == p[j - 1] \or
_p[j - 1] == '?'...**

4. 初始条件

- 默认状态下，两个空字符串是匹配的，即 $dp[0][0] = \text{True}$ 。
- 当字符串 s 为空，字符串 p 开始字符为若干个 * 时，两个字符串是匹配的，即 $p[j - 1] == '*'$ 时， $dp[0][j] = \text{True}$ 。

5. 最终结果

根据我们之前定义的状态， $dp[i][j]$ 表示为：字符串 s 的前 i 个字符与字符串 p 的前 j 个字符是否匹配。则最终结果为 $dp[\text{size}_s][\text{size}_p]$ ，其实 size_s 是字符串 s 的长度， size_p 是字符串 p 的长度。

思路 1：动态规划代码

```
class Solution:
    def isMatch(self, s: str, p: str) -> bool:
        size_s, size_p = len(s), len(p)
        dp = [[False for _ in range(size_p + 1)] for _ in range(size_s + 1)]
        dp[0][0] = True

        for j in range(1, size_p + 1):
            if p[j - 1] != '*':
                break
            dp[0][j] = True

        for i in range(1, size_s + 1):
            for j in range(1, size_p + 1):
                if s[i - 1] == p[j - 1] or p[j - 1] == '?':
                    dp[i][j] = dp[i - 1][j - 1]
                elif p[j - 1] == '*':
                    dp[i][j] = dp[i - 1][j] or dp[i][j - 1]

        return dp[size_s][size_p]
```

思路 1：复杂度分析

- **时间复杂度**: $O(mn)$, 其中 m 是字符串 s 的长度, n 是字符串 p 的长度。使用了两重循环, 外层循环遍历的时间复杂度是 $O(m)$, 内层循环遍历的时间复杂度是 $O(n)$, 所以总体的时间复杂度为 $O(mn)$ 。
- **空间复杂度**: $O(mn)$, 其中 m 是字符串 s 的长度, n 是字符串 p 的长度。使用了二维数组保存状态, 且第一维的空间复杂度为 $O(m)$, 第二位的空间复杂度为 $O(n)$, 所以总体的空间复杂度为 $O(mn)$ 。

0045. 跳跃游戏 II

- 标签: 贪心、数组、动态规划
- 难度: 中等

题目链接

- 0045. 跳跃游戏 II - 力扣

题目大意

描述：给定一个非负整数数组 `nums`，数组中每个元素代表在该位置可以跳跃的最大长度。开始位置为数组的第一个下标处。

要求：计算出到达最后一个下标处的最少的跳跃次数。假设你总能到达数组的最后一个下标处。

说明：

- $1 \leq \text{nums.length} \leq 10^4$ 。
- $0 \leq \text{nums}[i] \leq 1000$ 。

示例：

- **示例 1：**

输入: `nums = [2,3,1,1,4]`

输出: `2`

解释: 跳到最后一个位置的最小跳跃数是 `2`。从下标为 `0` 跳到下标为 `1` 的位置，跳 `1` 步，然后跳 `3` 步到达数组的

解题思路

思路 1：动态规划（超时）

1. 划分阶段

按照位置进行阶段划分。

2. 定义状态

定义状态 `dp[i]` 表示为：跳到下标 `i` 所需要的最小跳跃次数。

3. 状态转移方程

对于当前位置 `i`，如果之前的位置 `j` ($0 \leq j < i$) 能够跳到位置 `i` 需要满足：位置 `j` ($0 \leq j < i$) 加上位置 `j` 所能跳到的最远长度要大于等于 `i`，即 `j + nums[j] >= i`。

而跳到下标 i 所需要的最小跳跃次数则等于满足上述要求的位置 j 中最小跳跃次数加 1，即 $dp[i] = \min(dp[i], dp[j] + 1)$ 。

4. 初始条件

初始状态下，跳到下标 0 需要的最小跳跃次数为 0 ，即 $dp[0] = 0$ 。

5. 最终结果

根据我们之前定义的状态， $dp[i]$ 表示为：跳到下标 i 所需要的最小跳跃次数。则最终结果为 $dp[size - 1]$ 。

思路 1：动态规划（超时）代码

```
class Solution:
    def jump(self, nums: List[int]) -> int:
        size = len(nums)
        dp = [float("inf") for _ in range(size)]
        dp[0] = 0

        for i in range(1, size):
            for j in range(i):
                if j + nums[j] >= i:
                    dp[i] = min(dp[i], dp[j] + 1)

        return dp[size - 1]
```

思路 1：复杂度分析

- **时间复杂度：** $O(n^2)$ 。两重循环遍历的时间复杂度是 $O(n^2)$ ，所以总体时间复杂度为 $O(n^2)$ 。
- **空间复杂度：** $O(n)$ 。用到了一维数组保存状态，所以总体空间复杂度为 $O(n)$ 。

思路 2：动态规划 + 贪心

因为本题的数据规模为 10^4 ，而思路 1 的时间复杂度是 $O(n^2)$ ，所以就超时了。那么我们有什么方法可以优化一下，减少一下时间复杂度吗？

上文提到，在满足 $j + nums[j] \geq i$ 的情况下， $dp[i] = \min(dp[i], dp[j] + 1)$ 。

通过观察可以发现， $dp[i]$ 是单调递增的，也就是说 $dp[i - 1] \leq dp[i] \leq dp[i + 1]$ 。

举个例子，比如跳到下标 i 最少需要 5 步，即 $dp[i] = 5$ ，那么必然不可能出现少于 5 步就能跳到下标 $i + 1$ 的情况，跳到下标 $i + 1$ 至少需要 5 步或者更多步。

既然 $dp[i]$ 是单调递增的，那么在更新 $dp[i]$ 时，我们找到最早可以跳到 i 的点 j ，从该点更新 $dp[i]$ 。即找到满足 $j + nums[j] \geq i$ 的第一个 j ，使得 $dp[i] = dp[j] + 1$ 。

而查找第一个 j 的过程可以通过使用一个指针变量 j 从前向后迭代查找。

最后，将最终结果 $dp[size - 1]$ 返回即可。

思路 2：动态规划 + 贪心代码

```
class Solution:
    def jump(self, nums: List[int]) -> int:
        size = len(nums)
        dp = [float("inf") for _ in range(size)]
        dp[0] = 0

        j = 0
        for i in range(1, size):
            while j + nums[j] < i:
                j += 1
            dp[i] = dp[j] + 1

        return dp[size - 1]
```

思路 2：复杂度分析

- **时间复杂度**: $O(n)$ 。最外层循环遍历的时间复杂度是 $O(n)$ ，看似和内层循环结合遍历的时间复杂度是 $O(n^2)$ ，实际上内层循环只遍历了一遍，与外层循环遍历次数是相加关系，两者的时间复杂度和是 $O(2n)$ ， $O(2n) = O(n)$ ，所以总体时间复杂度为 $O(n)$ 。
- **空间复杂度**: $O(n)$ 。用到了一维数组保存状态，所以总体空间复杂度为 $O(n)$ 。

思路 2：贪心算法

如果第 i 个位置所能跳到的位置为 $[i + 1, i + nums[i]]$ ，则：

- 第 0 个位置所能跳到的位置就是 $[0 + 1, 0 + nums[0]]$ ，即 $[1, nums[0]]$ 。
- 第 1 个位置所能跳到的位置就是 $[1 + 1, 1 + nums[1]]$ ，即 $[2, 1 + nums[1]]$ 。
-

对于每一个位置 i 来说，所能跳到的所有位置都可以作为下一个起跳点，为了尽可能使用最少的跳跃次数，所以我们应该使得下一次起跳所能达到的位置尽可能的远。简单来说，就是每次在「可跳范围」内选择可以使下一次跳的更远的位置。这样才能获得最少跳跃次数。具体做法如下：

1. 维护几个变量：当前所能达到的最远位置 end ，下一步所能跳到的最远位置 max_pos ，最少跳跃次数 $steps$ 。
2. 遍历数组 $nums$ 的前 $\text{len}(nums) - 1$ 个元素：
3. 每次更新第 i 位置下一步所能跳到的最远位置 max_pos 。
4. 如果索引 i 到达了 end 边界，则：更新 end 为新的当前位置 max_pos ，并令步数 $steps$ 加 1。
5. 最终返回跳跃次数 $steps$ 。

思路 2：贪心算法代码

```
class Solution:  
    def jump(self, nums: List[int]) -> int:  
        end, max_pos = 0, 0  
        steps = 0  
        for i in range(len(nums) - 1):  
            max_pos = max(max_pos, nums[i] + i)  
            if i == end:  
                end = max_pos  
                steps += 1  
        return steps
```

思路 2：复杂度分析

- **时间复杂度**: $O(n)$ 。一重循环遍历的时间复杂度是 $O(n)$ ，所以总体时间复杂度为 $O(n)$ 。
- **空间复杂度**: $O(1)$ 。只用到了常数项的变量，所以总体空间复杂度为 $O(1)$ 。

参考资料

- [【题解】【宫水三叶の相信科学系列】详解「DP + 贪心 + 双指针」解法，以及该如何猜 DP 的状态定义 - 跳跃游戏 II - 力扣](#)
- [【题解】动态规划+贪心，易懂。 - 跳跃游戏 II - 力扣](#)

0046. 全排列

- 标签：数组、回溯
- 难度：中等

题目链接

- [0046. 全排列 - 力扣](#)

题目大意

描述：给定一个不含重复数字的数组 `nums`。

要求：返回其有可能的全排列。

说明：

- $1 \leq \text{nums.length} \leq 6$
- $-10 \leq \text{nums}[i] \leq 10$ 。
- `nums` 中的所有整数互不相同。

示例：

- **示例 1：**

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]]`

- **示例 2：**

输入: `nums = [0,1]`

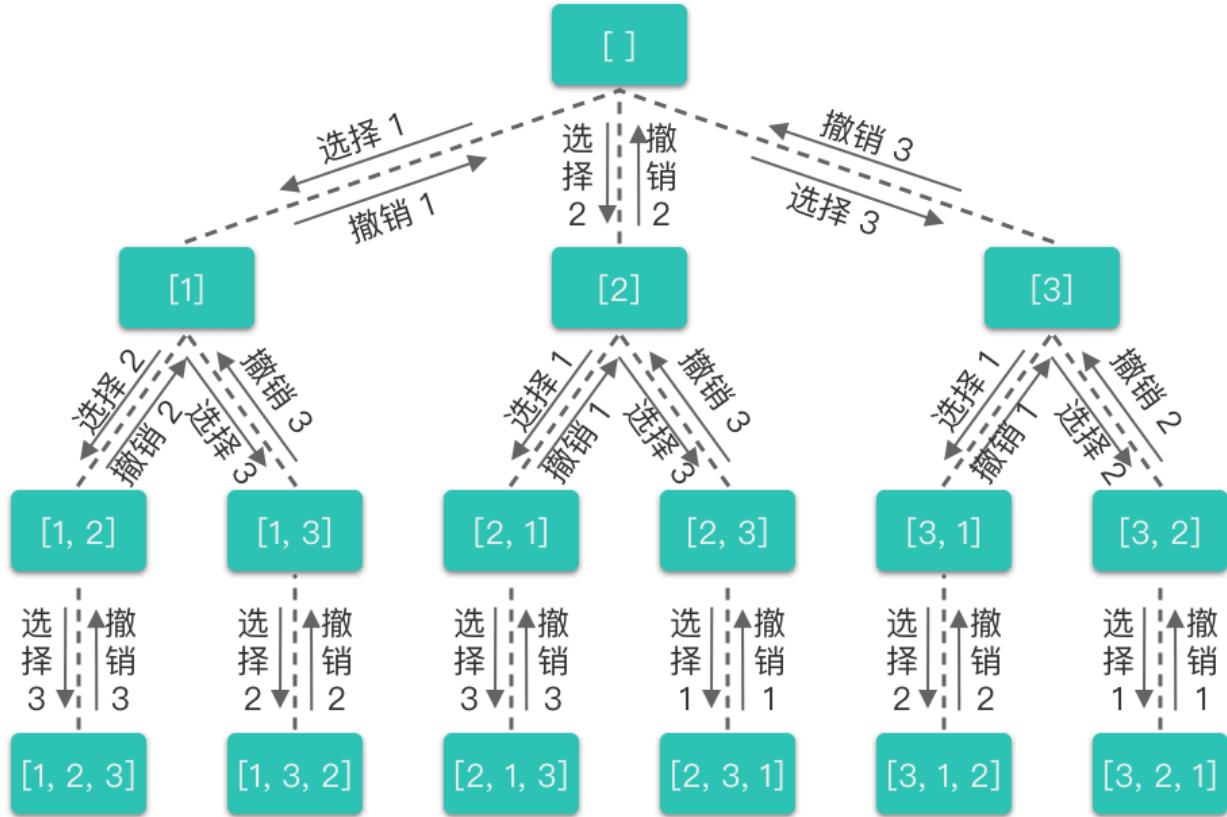
输出: `[[0,1], [1,0]]`

解题思路

思路 1：回溯算法

根据回溯算法三步走，写出对应的回溯算法。

1. 明确所有选择：全排列中每个位置上的元素都可以从剩余可选元素中选出，对此画出决策树，如下图所示。



2. 明确终止条件：

- 当遍历到决策树的叶子节点时，就终止了。即当前路径搜索到末尾时，递归终止。

3. 将决策树和终止条件翻译成代码：

- 定义回溯函数：

- backtracking(nums)：函数的传入参数是 nums（可选数组列表），全局变量是 res（存放所有符合条件结果的集合数组）和 path（存放当前符合条件的结果）。
- backtracking(nums)：函数代表的含义是：递归在 nums 中选择剩下的元素。

- 书写回溯函数主体（给出选择元素、递归搜索、撤销选择部分）。

- 从目前正在考虑元素，到数组结束为止，枚举出所有可选的元素。对于每一个可选元素：
 - 约束条件：之前已经选择的元素不再重复选用，只能从剩余元素中选择。
 - 选择元素：将其添加到当前子集数组 path 中。
 - 递归搜索：在选择该元素的情况下，继续递归选择剩下元素。

- 撤销选择：将该元素从当前结果数组 `path` 中移除。

```

for i in range(len(nums)):
    # 枚举可选元素列表
    if nums[i] not in path:
        # 从当前路径中没有出现的数字中选择
        path.append(nums[i])
        # 选择元素
        backtracking(nums)
        # 递归搜索
        path.pop()
        # 撤销选择

```

iii. 明确递归终止条件（给出递归终止条件，以及递归终止时的处理方法）。

- 当遍历到决策树的叶子节点时，就终止了。也就是存放当前结果的数组 `path` 的长度等于给定数组 `nums` 的长度（即 `len(path) == len(nums)`）时，递归停止。

思路 1：代码

```

class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        res = []      # 存放所有符合条件结果的集合
        path = []     # 存放当前符合条件的结果
        def backtracking(nums):           # nums 为选择元素列表
            if len(path) == len(nums):
                res.append(path[:])       # 说明找到了一组符合条件的结果
                path.append(nums[0])       # 将当前符合条件的结果放入集合中
                return

            for i in range(len(nums)):
                if nums[i] not in path:
                    path.append(nums[i])
                    backtracking(nums)
                    path.pop()
                    # 撤销选择

        backtracking(nums)
        return res

```

思路 1：复杂度分析

- 时间复杂度： $O(n \times n!)$ ，其中 n 为数组 `nums` 的元素个数。
- 空间复杂度： $O(n)$ 。

0047. 全排列 II

- 标签：数组、回溯
- 难度：中等

题目链接

- [0047. 全排列 II - 力扣](#)

题目大意

描述：给定一个可包含重复数字的序列 `nums`。

要求：按任意顺序返回所有不重复的全排列。

说明：

- $1 \leq \text{nums.length} \leq 8$ 。
- $-10 \leq \text{nums}[i] \leq 10$ 。

示例：

- **示例 1：**

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

- **示例 2：**

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]]`

解题思路

思路 1：回溯算法

这道题跟「[0046. 全排列](#)」不一样的地方在于增加了序列中的元素可重复这一条件。这就涉及到了如何去重。

我们可以先对数组 `nums` 进行排序，然后使用一个数组 `visited` 标记该元素在当前排列中是否被访问过。

如果未被访问过则将其加入排列中，并在访问后将该元素变为未访问状态。

然后再递归遍历下一层元素之前，增加一句语句进行判

重：`if i > 0 and nums[i] == nums[i - 1] and not visited[i - 1]: continue。`

然后再进行回溯遍历。

思路 1：代码

```
class Solution:
    res = []
    path = []
    def backtrack(self, nums: List[int], visited: List[bool]):
        if len(self.path) == len(nums):
            self.res.append(self.path[:])
            return
        for i in range(len(nums)):
            if i > 0 and nums[i] == nums[i - 1] and not visited[i - 1]:
                continue

            if not visited[i]:
                visited[i] = True
                self.path.append(nums[i])
                self.backtrack(nums, visited)
                self.path.pop()
                visited[i] = False

    def permuteUnique(self, nums: List[int]) -> List[List[int]]:
        self.res.clear()
        self.path.clear()
        nums.sort()
        visited = [False for _ in range(len(nums))]
        self.backtrack(nums, visited)
        return self.res
```

思路 1：复杂度分析

- 时间复杂度： $O(n \times n!)$ ，其中 n 为数组 `nums` 的元素个数。
- 空间复杂度： $O(n)$ 。

0048. 旋转图像

- 标签：数组、数学、矩阵
- 难度：中等

题目链接

- 0048. 旋转图像 - 力扣

题目大意

描述：给定一个 $n \times n$ 大小的二维矩阵（代表图像） $matrix$ 。

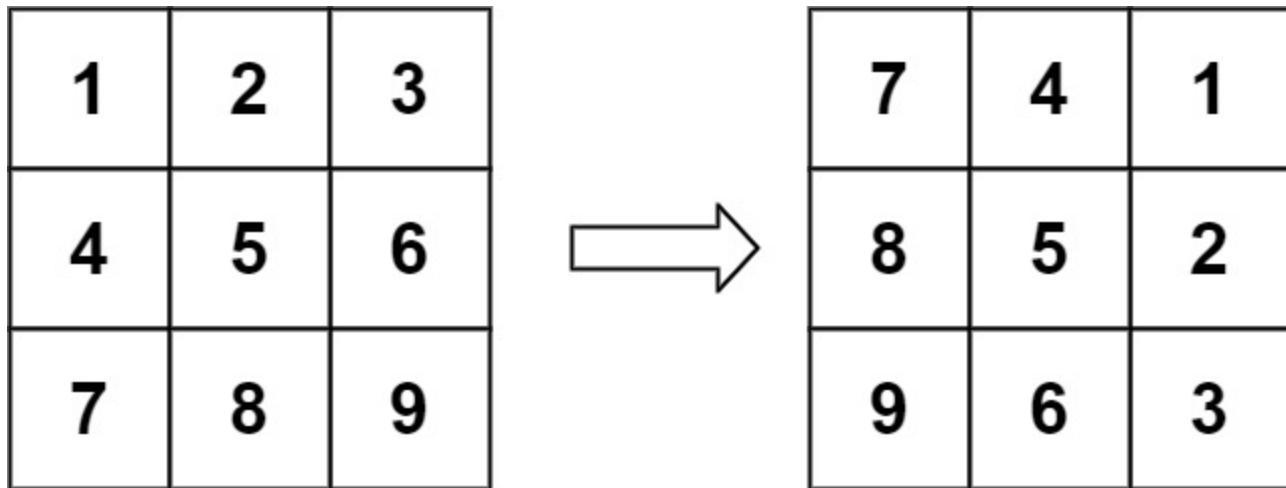
要求：将二维矩阵 $matrix$ 顺时针旋转 90° 。

说明：

- 不能使用额外的数组空间。
- $n == matrix.length == matrix[i].length$ 。
- $1 \leq n \leq 20$ 。
- $-1000 \leq matrix[i][j] \leq 1000$ 。

示例：

- 示例 1：



输入：`matrix = [[1,2,3],[4,5,6],[7,8,9]]`

输出：`[[7,4,1],[8,5,2],[9,6,3]]`

- 示例 2：

5	1	9	11
2	4	8	10
13	3	6	7
15	14	12	16

15	13	2	5
14	3	4	1
12	6	8	9
16	7	10	11

输入: `matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]`

输出: `[[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]`

解题思路

思路 1：原地旋转

如果使用额外数组空间的话，将对应元素存放到对应位置即可。如果不使用额外的数组空间，则需要观察每一个位置上的点最初位置和最终位置有什么规律。

对于矩阵中第 i 行的第 j 个元素，在旋转后，它出现在倒数第 i 列的第 j 个位置。即 $matrixnew[j][n - i - 1] = matrix[i][j]$ 。

而 $matrixnew[j][n - i - 1]$ 的点经过旋转变移到了 $matrix[n - i - 1][n - j - 1]$ 的位置。

$matrix[n - i - 1][n - j - 1]$ 位置上的点经过旋转变移到了 $matrix[n - j - 1][i]$ 的位置。

$matrix[n - j - 1][i]$ 位置上的点经过旋转变移到到了最初的 $matrix[i][j]$ 的位置。

这样就形成了一个循环，我们只需要通过一个临时变量 $temp$ 就可以将循环中的元素逐一进行交换。Python 中则可以直接使用语法直接交换。

思路 1：代码

```
class Solution:
    def rotate(self, matrix: List[List[int]]) -> None:
        n = len(matrix)

        for i in range(n // 2):
            for j in range((n + 1) // 2):
                matrix[i][j], matrix[n - j - 1][i], matrix[n - i - 1][n - j - 1], matrix[n - i - 1][n - j - 1] = matrix[n - j - 1][i], matrix[n - i - 1][n - j - 1], matrix[i][j], matrix[n - i - 1][n - j - 1]
```

思路 1：复杂度分析

- 时间复杂度： $O(n^2)$ 。
- 空间复杂度： $O(1)$ 。

思路 2：原地翻转

通过观察可以得出：原矩阵可以通过一次「水平翻转」+「主对角线翻转」得到旋转后的二维矩阵。

思路 2：代码

```
def rotate(self, matrix: List[List[int]]) -> None:
    n = len(matrix)

    for i in range(n // 2):
        for j in range(n):
            matrix[i][j], matrix[n - i - 1][j] = matrix[n - i - 1][j], matrix[i][j]

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

思路 2：复杂度分析

- 时间复杂度： $O(n^2)$ 。
- 空间复杂度： $O(1)$ 。

0049. 字母异位词分组

- 标签：数组、哈希表、字符串、排序
- 难度：中等

题目链接

- [0049. 字母异位词分组 - 力扣](#)

题目大意

给定一个字符串数组，将包含字母相同的字符串组合在一起，不需要考虑输出顺序。

解题思路

使用哈希表记录字母相同的字符串。对每一个字符串进行排序，按照 排序字符串：字母相同的字符串数组 的键值顺序进行存储。

最终将哈希表的值转换为对应数组返回结果。

代码

```
class Solution:  
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:  
        str_dict = dict()  
        res = []  
        for s in strs:  
            sort_s = str(sorted(s))  
            if sort_s in str_dict:  
                str_dict[sort_s] += [s]  
            else:  
                str_dict[sort_s] = [s]  
  
        for sort_s in str_dict:  
            res += [str_dict[sort_s]]  
        return res
```

0050. Pow(x, n)

- 标签：递归、数学
- 难度：中等

题目链接

- [0050. Pow\(x, n\) - 力扣](#)

题目大意

描述：给定浮点数 x 和整数 n 。

要求：计算 x 的 n 次方（即 x^n ）。

说明：

- $-100.0 < x < 100.0$ 。
- $-2^{31} \leq n \leq 2^{31} - 1$ 。
- n 是一个整数。
- $-10^4 \leq x^n \leq 10^4$ 。

示例：

- **示例 1：**

输入: `x = 2.00000, n = 10`

输出: `1024.00000`

- **示例 2：**

输入: `x = 2.00000, n = -2`

输出: `0.25000`

解释: $2^{-2} = 1/2^2 = 1/4 = 0.25$

解题思路

思路 1：分治算法

常规方法是直接将 x 累乘 n 次得出结果，时间复杂度为 $O(n)$ 。

我们可以利用分治算法来减少时间复杂度。

根据 n 的奇偶性，我们可以得到以下结论：

1. 如果 n 为偶数， $x^n = x^{n/2} \times x^{n/2}$ 。
2. 如果 n 为奇数， $x^n = x \times x^{(n-1)/2} \times x^{(n-1)/2}$ 。

$x^{(n/2)}$ 或 $x^{(n-1)/2}$ 又可以继续向下递归划分。

则我们可以利用低纬度的幂计算结果，来得到高纬度的幂计算结果。

这样递归求解，时间复杂度为 $O(\log n)$ ，并且递归也可以转为递推来做。

需要注意如果 n 为负数，可以转换为 $\frac{1}{x}^{-n}$ 。

思路 1：代码

```
class Solution:  
    def myPow(self, x: float, n: int) -> float:  
        if x == 0.0:  
            return 0.0  
        res = 1  
        if n < 0:  
            x = 1/x  
            n = -n  
        while n:  
            if n & 1:  
                res *= x  
            x *= x  
            n >>= 1  
        return res
```

思路 1：复杂度分析

- 时间复杂度： $O(\log n)$ 。

- 空间复杂度: $O(1)$ 。

0051. N 皇后

- 标签: 数组、回溯
- 难度: 困难

题目链接

- [0051. N 皇后 - 力扣](#)

题目大意

描述: 给定一个整数 n 。

要求: 返回所有不同的「 n 皇后问题」的解决方案。每一种解法包含一个不同的「 n 皇后问题」的棋子放置方案，该方案中的 `Q` 和 `.` 分别代表了皇后和空位。

说明:

- **n 皇后问题:** 将 n 个皇后放置在 $n * n$ 的棋盘上，并且使得皇后彼此之间不能攻击。
- **皇后彼此不能相互攻击:** 指的是任何两个皇后都不能处于同一条横线、纵线或者斜线上。
- $1 \leq n \leq 9$ 。

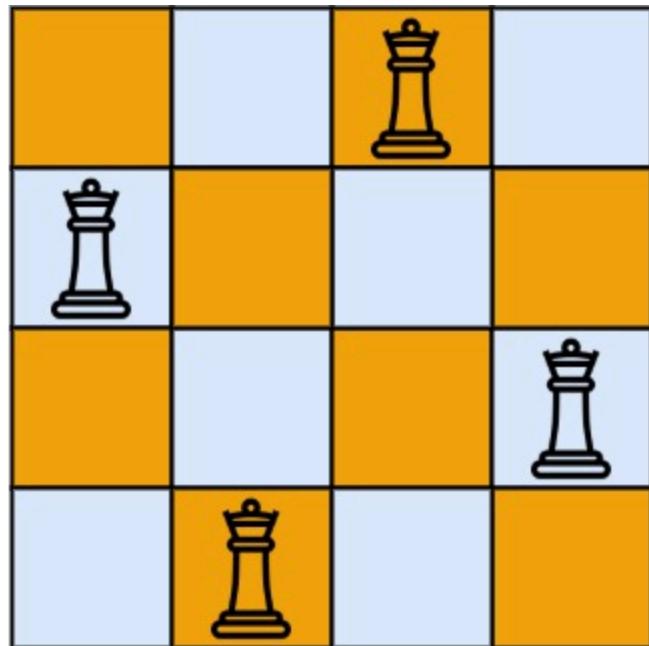
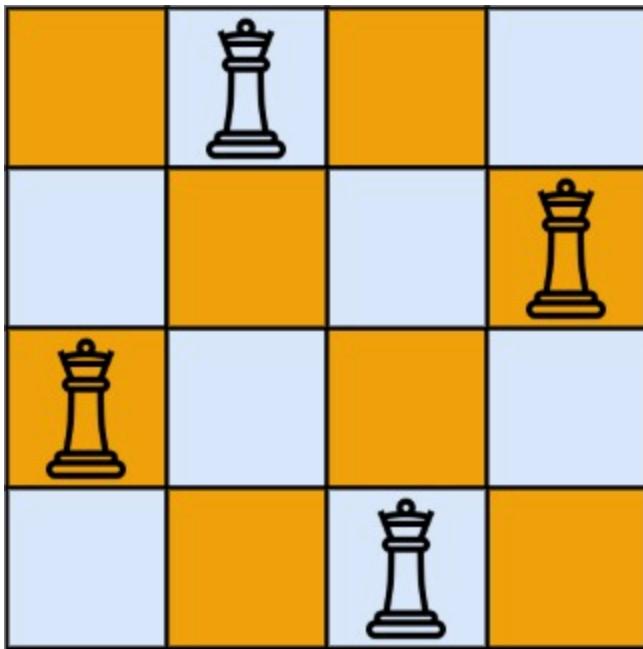
示例:

- **示例 1:**

输入: `n = 4`

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

解释: 如下图所示, `4` 皇后问题存在 `2` 个不同的解法。



解题思路

思路 1：回溯算法

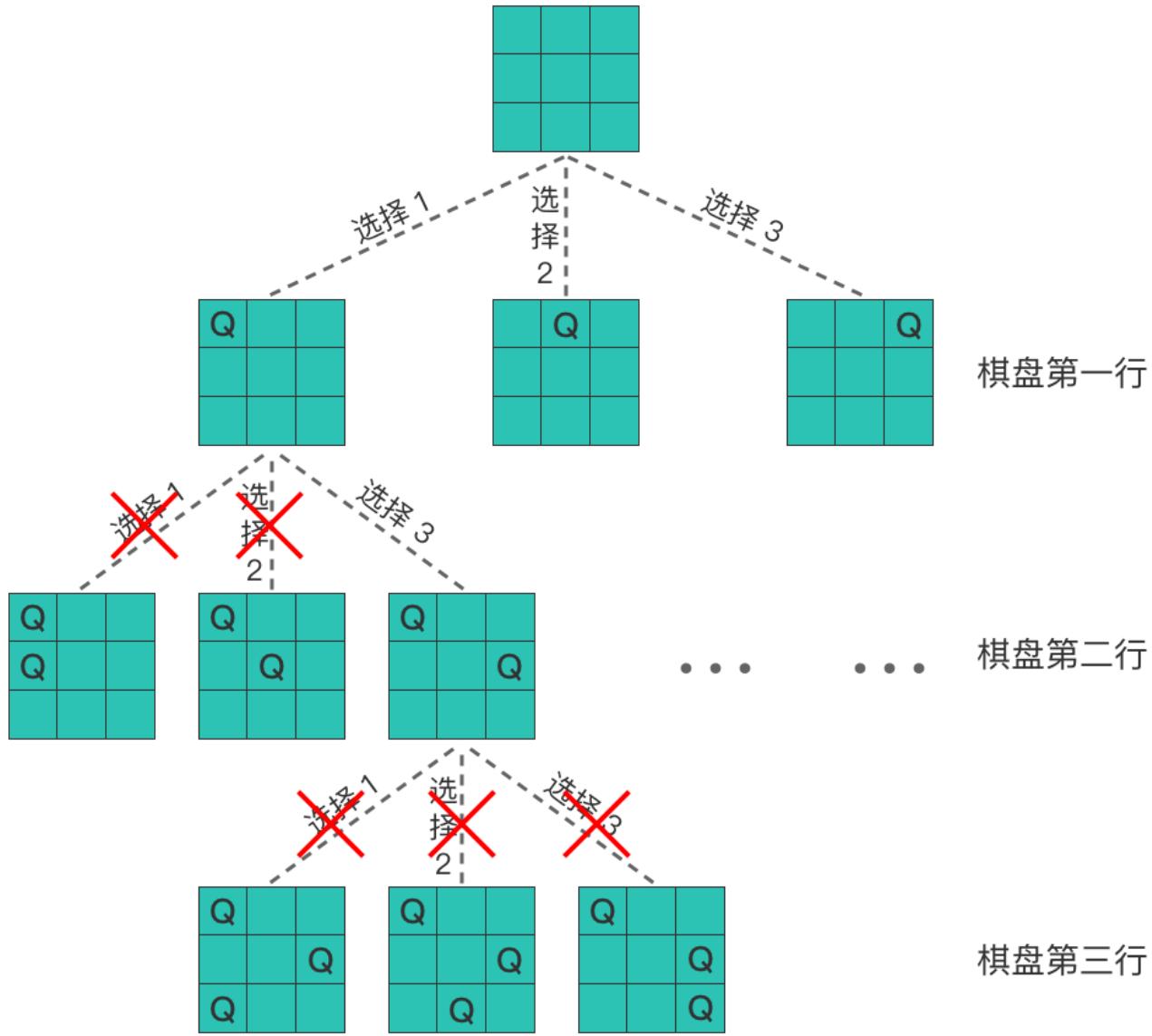
这道题是经典的回溯问题。我们可以按照行序来放置皇后，也就是先放第一行，再放第二行……一直到最后一行。

对于 $n * n$ 的棋盘来说，每一行有 n 列，也就有 n 种放法可供选择。我们可以尝试选择其中一列，查看是否与之前放置的皇后有冲突，如果没有冲突，则继续在下一行放置皇后。依次类推，直到放置完所有皇后，并且都不发生冲突时，就得到了一个合理的解。

并且在放置完之后，通过回溯的方式尝试其他可能的分支。

下面我们根据回溯算法三步走，写出对应的回溯算法。

1. 明确所有选择：根据棋盘中当前行的所有列位置上是否选择放置皇后，画出决策树，如下图所示。



2. 明确终止条件:

- 当遍历到决策树的叶子节点时，就终止了。也就是在最后一行放置完皇后时，递归终止。

3. 将决策树和终止条件翻译成代码:

i. 定义回溯函数:

- 首先我们先使用一个 $n * n$ 大小的二维矩阵 `chessboard` 来表示当前棋盘，`chessboard` 中的字符 `Q` 代表皇后，`.` 代表空位，初始都为 `.`。
- 然后定义回溯函数 `backtrack(chessboard, row)`： 函数的传入参数是 `chessboard`（棋盘数组）和 `row`（代表目前正在考虑放置第 `row` 行皇后），全局变量是 `res`（存放所有符合条件结果的集合数组）。
- `backtrack(chessboard, row)`： 函数代表的含义是：在放置好第 `row` 行皇后的情况下，递归放置剩下行的皇后。

ii. 书写回溯函数主体（给出选择元素、递归搜索、撤销选择部分）。

- 枚举出当前行所有的列。对于每一列位置：

- 约束条件：定义一个判断方法，先判断一下当前位置是否与之前棋盘上放置的皇后发生冲突，如果不发生冲突则继续放置，否则则继续向后遍历判断。
- 选择元素：选择 `row`, `col` 位置放置皇后，将其棋盘对应位置设置为 `'Q'`。
- 递归搜索：在该位置放置皇后的情况下，继续递归考虑下一行。
- 撤销选择：将棋盘上 `row`, `col` 位置设置为 `'.'`。

```

# 判断当前位置 row, col 是否与之前放置的皇后发生冲突
def isValid(self, n: int, row: int, col: int, chessboard: List[List[str]]):
    for i in range(row):
        if chessboard[i][col] == 'Q':
            return False

    i, j = row - 1, col - 1
    while i >= 0 and j >= 0:
        if chessboard[i][j] == 'Q':
            return False
        i -= 1
        j -= 1
    i, j = row - 1, col + 1
    while i >= 0 and j < n:
        if chessboard[i][j] == 'Q':
            return False
        i -= 1
        j += 1

    return True

for col in range(n):
    if self.isValid(n, row, col, chessboard):          # 枚举可放置皇后的列
        chessboard[row][col] = 'Q'                      # 如果该位置与之前放置的皇后不发生冲突
        backtrack(row + 1, chessboard)                  # 选择 row, col 位置放置皇后
        chessboard[row][col] = '.'                      # 递归放置 row + 1 行之后的皇后
                                                        # 撤销选择 row, col 位置

```

iii. 明确递归终止条件（给出递归终止条件，以及递归终止时的处理方法）。

- 当遍历到决策树的叶子节点时，就终止了。也就是在最后一行放置完皇后（即 `row == n`）时，递归停止。
- 递归停止时，将当前符合条件的棋盘转换为答案需要的形式，然后将其存入答案数组 `res` 中即可。

思路 1：代码

```
class Solution:
    res = []
    def backtrack(self, n: int, row: int, chessboard: List[List[str]]):
        if row == n:
            temp_res = []
            for temp in chessboard:
                temp_str = ''.join(temp)
                temp_res.append(temp_str)
            self.res.append(temp_res)
            return
        for col in range(n):
            if self.isValid(n, row, col, chessboard):
                chessboard[row][col] = 'Q'
                self.backtrack(n, row + 1, chessboard)
                chessboard[row][col] = '.'

    def isValid(self, n: int, row: int, col: int, chessboard: List[List[str]]):
        for i in range(row):
            if chessboard[i][col] == 'Q':
                return False

        i, j = row - 1, col - 1
        while i >= 0 and j >= 0:
            if chessboard[i][j] == 'Q':
                return False
            i -= 1
            j -= 1
        i, j = row - 1, col + 1
        while i >= 0 and j < n:
            if chessboard[i][j] == 'Q':
                return False
            i -= 1
            j += 1

    return True

    def solveNQueens(self, n: int) -> List[List[str]]:
        self.res.clear()
        chessboard = [['.' for _ in range(n)] for _ in range(n)]
```

```
    self.backtrack(n, 0, chessboard)
    return self.res
```

思路 1：复杂度分析

- **时间复杂度**: $O(n!)$, 其中 n 是皇后数量。
- **空间复杂度**: $O(n^2)$, 其中 n 是皇后数量。递归调用层数不会超过 n , 每个棋盘的空间复杂度为 $O(n^2)$, 所以空间复杂度为 $O(n^2)$ 。

0052. N 皇后 II

- 标签：回溯
- 难度：困难

题目链接

- [0052. N 皇后 II - 力扣](#)

题目大意

描述：给定一个整数 n 。

要求：返回「 n 皇后问题」不同解决方案的数量。

说明：

- **n 皇后问题**: 将 n 个皇后放置在 $n * n$ 的棋盘上，并且使得皇后彼此之间不能攻击。
- **皇后彼此不能相互攻击**: 指的是任何两个皇后都不能处于同一条横线、纵线或者斜线上。
- $1 \leq n \leq 9$ 。

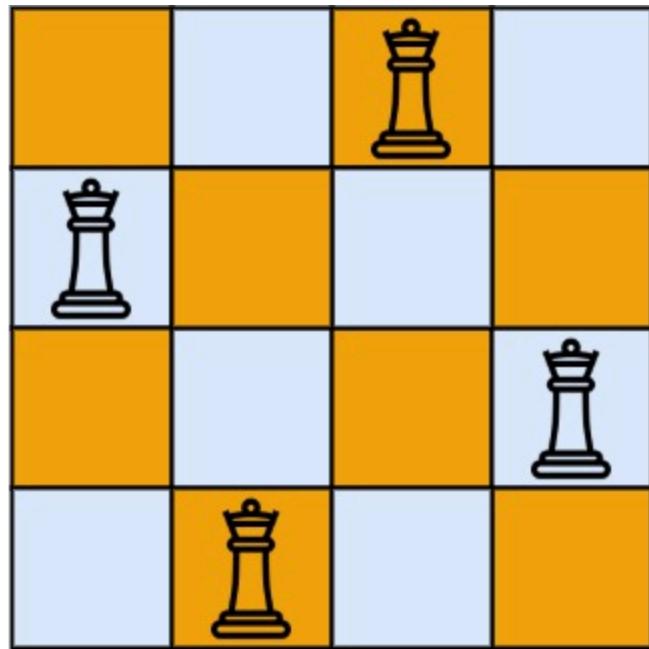
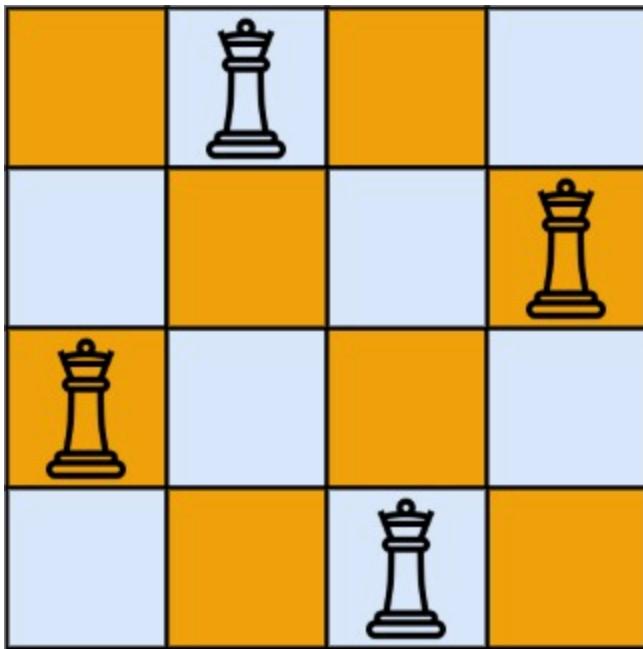
示例：

- **示例 1：**

输入: $n = 4$

输出: 2

解释: 如下图所示, 4 皇后问题存在两个不同的解法。



解题思路

思路 1：回溯算法

和「[51. N 皇后 - 力扣](#)」做法一致。区别在于「[51. N 皇后 - 力扣](#)」需要返回所有解决方案，而这道题只需要得到所有解决方案的数量即可。下面来说一下这道题的解题思路。

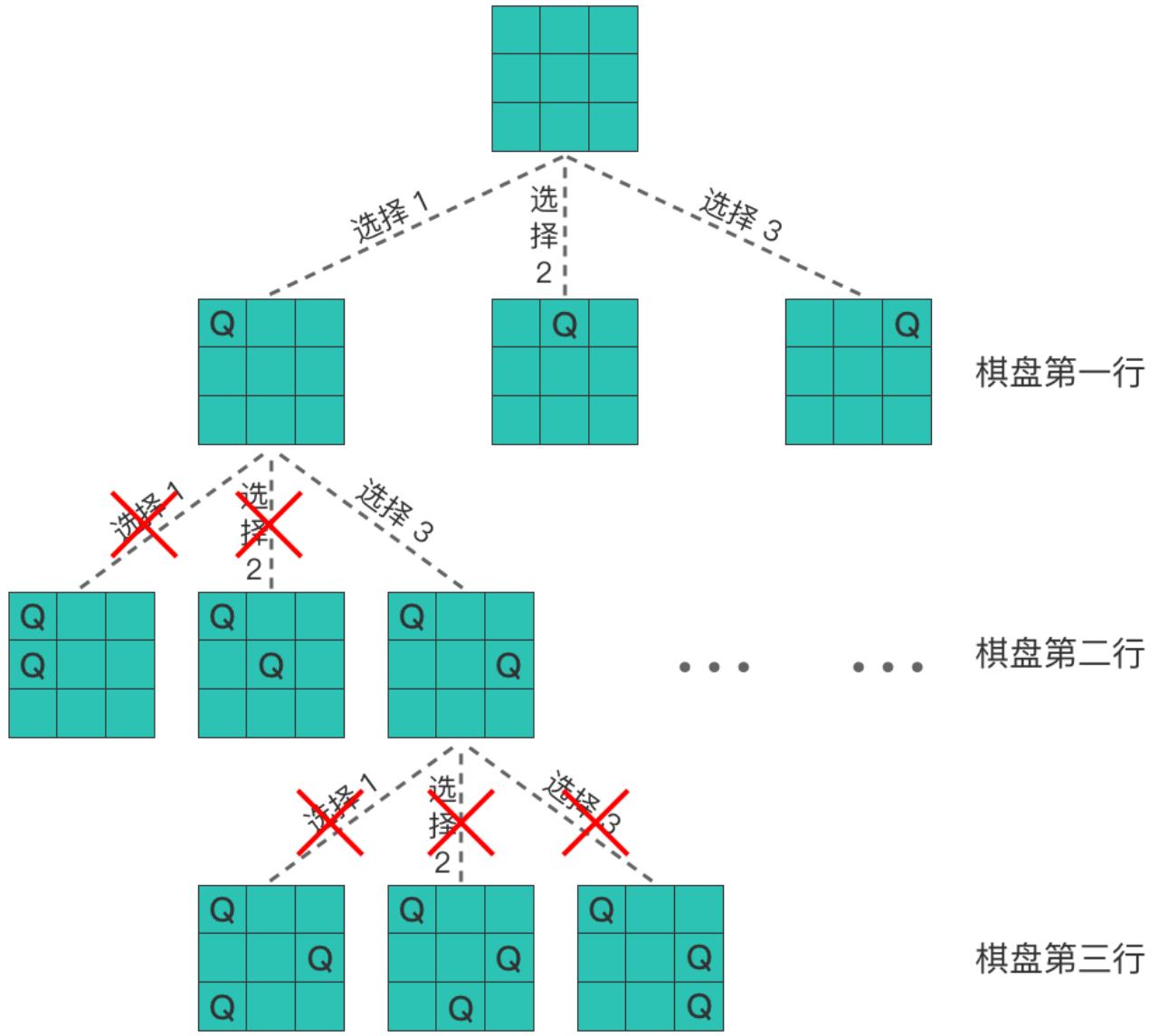
我们可以按照行序来放置皇后，也就是先放第一行，再放第二行……一直放到最后一行。

对于 $n * n$ 的棋盘来说，每一行有 n 列，也就有 n 种放法可供选择。我们可以尝试选择其中一列，查看是否与之前放置的皇后有冲突，如果没有冲突，则继续在下一行放置皇后。依次类推，直到放置完所有皇后，并且都不发生冲突时，就得到了一个合理的解。

并且在放置完之后，通过回溯的方式尝试其他可能的分支。

下面我们根据回溯算法三步走，写出对应的回溯算法。

1. 明确所有选择：根据棋盘中当前行的所有列位置上是否选择放置皇后，画出决策树，如下图所示。



2. 明确终止条件:

- 当遍历到决策树的叶子节点时，就终止了。也就是在最后一行放置完皇后时，递归终止。

3. 将决策树和终止条件翻译成代码:

i. 定义回溯函数:

- 首先我们先使用一个 $n * n$ 大小的二维矩阵 `chessboard` 来表示当前棋盘，`chessboard` 中的字符 `Q` 代表皇后，`.` 代表空位，初始都为 `.`。
- 然后定义回溯函数 `backtrack(chessboard, row)`： 函数的传入参数是 `chessboard`（棋盘数组）和 `row`（代表目前正在考虑放置第 `row` 行皇后），全局变量是 `ans`（所有可行方案的数量）。
- `backtrack(chessboard, row)`： 函数代表的含义是： 在放置好第 `row` 行皇后的情况下，递归放置剩下行的皇后。

ii. 书写回溯函数主体（给出选择元素、递归搜索、撤销选择部分）。

- 枚举出当前行所有的列。对于每一列位置：

- 约束条件：定义一个判断方法，先判断一下当前位置是否与之前棋盘上放置的皇后发生冲突，如果不发生冲突则继续放置，否则则继续向后遍历判断。
- 选择元素：选择 `row`, `col` 位置放置皇后，将其棋盘对应位置设置为 `Q`。
- 递归搜索：在该位置放置皇后的情况下，继续递归考虑下一行。
- 撤销选择：将棋盘上 `row`, `col` 位置设置为 `.`。

思路 1：代码

```
class Solution:
    # 判断当前位置 row, col 是否与之前放置的皇后发生冲突
    def isValid(self, n: int, row: int, col: int, chessboard: List[List[str]]):
        for i in range(row):
            if chessboard[i][col] == 'Q':
                return False

        i, j = row - 1, col - 1
        while i >= 0 and j >= 0:
            if chessboard[i][j] == 'Q':
                return False
            i -= 1
            j -= 1

        i, j = row - 1, col + 1
        while i >= 0 and j < n:
            if chessboard[i][j] == 'Q':
                return False
            i -= 1
            j += 1

        return True

    def totalNQueens(self, n: int) -> int:
        chessboard = [['.' for _ in range(n)] for _ in range(n)]      # 棋盘初始化

        ans = 0
        def backtrack(chessboard: List[List[str]], row: int):           # 正在
            if row == n:                                              # 遇到终止条件
                nonlocal ans
                ans += 1
                return

            for col in range(n):                                     # 枚举可放置皇后的列
                if self.isValid(n, row, col, chessboard):             # 如果该位置与之前放置的皇后不发
                    chessboard[row][col] = 'Q'                         # 选择 row, col 位置放置皇后
                    backtrack(chessboard, row + 1)                      # 递归放置 row + 1 行之后的皇后
                    chessboard[row][col] = '.'                         # 撤销选择 row, col 位置

        backtrack(chessboard, 0)
```

```
return ans
```

思路 1：复杂度分析

- **时间复杂度**: $O(n!)$, 其中 n 是皇后数量。
- **空间复杂度**: $O(n^2)$, 其中 n 是皇后数量。递归调用层数不会超过 n , 每个棋盘的空间复杂度为 $O(n^2)$, 所以空间复杂度为 $O(n^2)$ 。

0053. 最大子数组和

- 标签: 数组、分治、动态规划
- 难度: 中等

题目链接

- [0053. 最大子数组和 - 力扣](#)

题目大意

描述: 给定一个整数数组 $nums$ 。

要求: 找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

说明:

- **子数组**: 指的是数组中的一个连续部分。
- $1 \leq nums.length \leq 10^5$ 。
- $-10^4 \leq nums[i] \leq 10^4$ 。

示例:

- **示例 1:**

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。

- 示例 2：

输入: `nums = [1]`

输出: `1`

解题思路

思路 1：动态规划

1. 划分阶段

按照连续子数组的结束位置进行阶段划分。

2. 定义状态

定义状态 $dp[i]$ 为：以第 i 个数结尾的连续子数组的最大和。

3. 状态转移方程

状态 $dp[i]$ 为：以第 i 个数结尾的连续子数组的最大和。则我们可以从「第 $i - 1$ 个数结尾的连续子数组的最大和」，以及「第 i 个数的值」来讨论 $dp[i]$ 。

- 如果 $dp[i - 1] < 0$ ，则「第 $i - 1$ 个数结尾的连续子数组的最大和」 + 「第 i 个数的值」 < 「第 i 个数的值」，即： $dp[i - 1] + nums[i] < nums[i]$ 。所以，此时 $dp[i]$ 应取「第 i 个数的值」，即 $dp[i] = nums[i]$ 。
- 如果 $dp[i - 1] \geq 0$ ，则「第 $i - 1$ 个数结尾的连续子数组的最大和」 + 「第 i 个数的值」 >= 第 i 个数的值，即： $dp[i - 1] + nums[i] \geq nums[i]$ 。所以，此时 $dp[i]$ 应取「第 $i - 1$ 个数结尾的连续子数组的最大和」 + 「第 i 个数的值」，即 $dp[i] = dp[i - 1] + nums[i]$ 。

归纳一下，状态转移方程为：

$$dp[i] = \begin{cases} nums[i], & dp[i - 1] < 0 \\ dp[i - 1] + nums[i] & dp[i - 1] \geq 0 \end{cases}$$

4. 初始条件

- 第 0 个数结尾的连续子数组的最大和为 $nums[0]$ ，即 $dp[0] = nums[0]$ 。

5. 最终结果

根据状态定义， $dp[i]$ 为：以第 i 个数结尾的连续子数组的最大和。则最终结果应为所有 $dp[i]$ 的最大值，即 $\max(dp)$ 。

思路 1：代码

```
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        size = len(nums)
        dp = [0 for _ in range(size)]

        dp[0] = nums[0]
        for i in range(1, size):
            if dp[i - 1] < 0:
                dp[i] = nums[i]
            else:
                dp[i] = dp[i - 1] + nums[i]
        return max(dp)
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ ，其中 n 为数组 $nums$ 的元素个数。
- 空间复杂度： $O(n)$ 。

思路 2：动态规划 + 滚动优化

因为 $dp[i]$ 只和 $dp[i - 1]$ 和当前元素 $nums[i]$ 相关，我们也可以使用一个变量 $subMax$ 来表示以第 i 个数结尾的连续子数组的最大和。然后使用 $ansMax$ 来保存全局中最大值。

思路 2：代码

```
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        size = len(nums)
        subMax = nums[0]
        ansMax = nums[0]

        for i in range(1, size):
            if subMax < 0:
                subMax = nums[i]
            else:
                subMax += nums[i]
            ansMax = max(ansMax, subMax)
        return ansMax
```

思路 2：复杂度分析

- **时间复杂度**: $O(n)$, 其中 n 为数组 $nums$ 的元素个数。
- **空间复杂度**: $O(1)$ 。

思路 3：分治算法

我们将数组 $nums$ 根据中心位置分为左右两个子数组。则具有最大和的连续子数组可能存在以下 3 种情况：

1. 具有最大和的连续子数组在左子数组中。
2. 具有最大和的连续子数组在右子数组中。
3. 具有最大和的连续子数组跨过中心位置，一部分在左子数组中，另一部分在右子树组中。

那么我们要求出具有最大和的连续子数组的最大和，则分别对上面 3 种情况求解即可。具体步骤如下：

1. 将数组 $nums$ 根据中心位置递归分为左右两个子数组，直到所有子数组长度为 1。
2. 长度为 1 的子数组最大和肯定是数组中唯一的数，将其返回即可。
3. 求出左子数组的最大和 $leftMax$ 。
4. 求出右子树组的最大和 $rightMax$ 。
5. 求出跨过中心位置，一部分在左子数组中，另一部分在右子树组的子数组最大和 $leftTotal + rightTotal$ 。
6. 求出 3、4、5 中的最大值，即为当前数组的最大和，将其返回即可。

思路 3：代码

```
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        def max_sub_array(low, high):
            if low == high:
                return nums[low]

            mid = low + (high - low) // 2
            leftMax = max_sub_array(low, mid)
            rightMax = max_sub_array(mid + 1, high)

            total = 0
            leftTotal = -inf
            for i in range(mid, low - 1, -1):
                total += nums[i]
                leftTotal = max(leftTotal, total)

            total = 0
            rightTotal = -inf
            for i in range(mid + 1, high + 1):
                total += nums[i]
                rightTotal = max(rightTotal, total)

            return max(leftMax, rightMax, leftTotal + rightTotal)

        return max_sub_array(0, len(nums) - 1)
```

思路 3：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(\log n)$ 。# 0054. 螺旋矩阵
- 标签：数组、矩阵、模拟
- 难度：中等

题目链接

- 0054. 螺旋矩阵 - 力扣

题目大意

描述：给定一个 $m \times n$ 大小的二维矩阵 $matrix$ 。

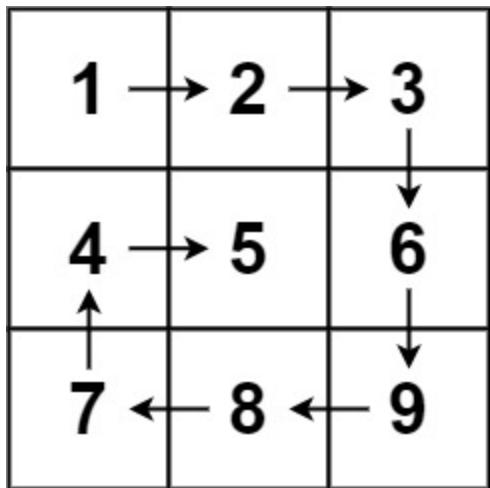
要求：按照顺时针旋转的顺序，返回矩阵中的所有元素。

说明：

- $m == matrix.length$ 。
- $n == matrix[i].length$ 。
- $1 \leq m, n \leq 10$ 。
- $-100 \leq matrix[i][j] \leq 100$ 。

示例：

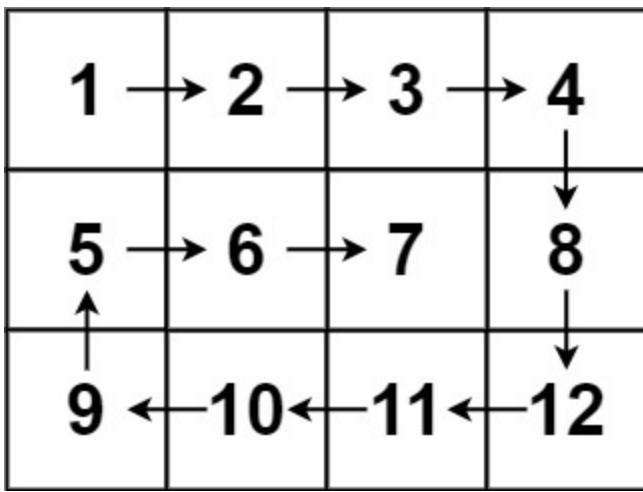
- **示例 1：**



输入： `matrix = [[1,2,3],[4,5,6],[7,8,9]]`

输出： `[1,2,3,6,9,8,7,4,5]`

- **示例 2：**



输入: `matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]`

输出: `[1,2,3,4,8,12,11,10,9,5,6,7]`

解题思路

思路 1：模拟

1. 使用数组 `ans` 存储答案。然后定义一下上、下、左、右的边界。
2. 然后按照逆时针的顺序从边界上依次访问元素。
3. 当访问完当前边界之后，要更新一下边界位置，缩小范围，方便下一轮进行访问。
4. 最后返回答案数组 `ans`。

思路 1：代码

```
class Solution:
    def spiralOrder(self, matrix: List[List[int]]) -> List[int]:
        up, down, left, right = 0, len(matrix)-1, 0, len(matrix[0])-1
        ans = []
        while True:
            for i in range(left, right + 1):
                ans.append(matrix[up][i])
            up += 1
            if up > down:
                break
            for i in range(up, down + 1):
                ans.append(matrix[i][right])
            right -= 1
            if right < left:
                break
            for i in range(right, left - 1, -1):
                ans.append(matrix[down][i])
            down -= 1
            if down < up:
                break
            for i in range(down, up - 1, -1):
                ans.append(matrix[i][left])
            left += 1
            if left > right:
                break
        return ans
```

思路 1：复杂度分析

- **时间复杂度**: $O(m \times n)$ 。其中 m 、 n 分别为二维矩阵的行数和列数。
- **空间复杂度**: $O(m \times n)$ 。如果算上答案数组的空间占用，则空间复杂度为 $O(m \times n)$ 。不算上则空间复杂度为 $O(1)$ 。

0055. 跳跃游戏

- 标签：贪心、数组、动态规划
- 难度：中等

题目链接

- [0055. 跳跃游戏 - 力扣](#)

题目大意

描述：给定一个非负整数数组 `nums`，数组中每个元素代表在该位置可以跳跃的最大长度。开始位置位于数组的第一个下标处。

要求：判断是否能够到达最后一个下标。

说明：

- $1 \leq \text{nums.length} \leq 3 \times 10^4$ 。
- $0 \leq \text{nums}[i] \leq 10^5$ 。

示例：

- **示例 1：**

输入: `nums = [2,3,1,1,4]`

输出: `true`

解释: 可以先跳 `1` 步，从下标 `0` 到达下标 `1`，然后再从下标 `1` 跳 `3` 步到达最后一个下标。

- **示例 2：**

输入: `nums = [3,2,1,0,4]`

输出: `false`

解释: 无论怎样，总会到达下标为 `3` 的位置。但该下标的最大跳跃长度是 `0`，所以永远不可能到达最后一个下标。

解题思路

思路 1：贪心算法

如果我們能通過前面的某个位置 j ，到达后面的某个位置 i ，則我們一定能到达区间 $[j, i]$ 中所有的点 ($j \leq i$)。

而前面的位置 j 肯定也是通过 j 前面的点到达的。所以我們可以通过贪心算法来計算出所能到达的最远位置。具体步骤如下：

1. 初始化能到达的最远位置 max_i 为 0。
2. 遍历数组 `nums`。
3. 如果能到达当前位置，即 $max_i \leq i$ ，并且当前位置 + 当前位置最大跳跃长度 > 能到达的最远位置，即 $i + nums[i] > max_i$ ，则更新能到达的最远位置 max_i 。
4. 遍历完数组，最后比较能到达的最远位置 max_i 和数组最远距离 `size - 1` 的关系。如果 $max_i \geq len(nums)$ ，则返回 `True`，否则返回 `False`。

思路 1：代码

```
class Solution:
    def canJump(self, nums: List[int]) -> bool:
        size = len(nums)
        max_i = 0
        for i in range(size):
            if max_i >= i and i + nums[i] > max_i:
                max_i = i + nums[i]

        return max_i >= size - 1
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ ，其中 n 是数组 `nums` 的长度。
- 空间复杂度：

思路 2：动态规划

1. 划分阶段

按照位置进行阶段划分。

2. 定义状态

定义状态 `dp[i]` 表示为：从位置 0 出发，经过 $j \leq i$ ，可以跳出的最远距离。

3. 状态转移方程

- 如果能通过 $0 \sim i - 1$ 个位置到达 i ，即 $dp[i - 1] \leq i$ ，则 $dp[i] = \max(dp[i - 1], i + nums[i])$ 。
- 如果不能通过 $0 \sim i - 1$ 个位置到达 i ，即 $dp[i - 1] < i$ ，则 $dp[i] = dp[i - 1]$ 。

4. 初始条件

初始状态下，从 0 出发，经过 0，可以跳出的最远距离为 $\text{nums}[0]$ ，即 $\text{dp}[0] = \text{nums}[0]$ 。

5. 最终结果

根据我们之前定义的状态， $\text{dp}[i]$ 表示为：从位置 0 出发，经过 $j \leq i$ ，可以跳出的最远距离。则我们需要判断 $\text{dp}[\text{size} - 1]$ 与数组最远距离 $\text{size} - 1$ 的关系。

思路 2：代码

```
class Solution:
    def canJump(self, nums: List[int]) -> bool:
        size = len(nums)
        dp = [0 for _ in range(size)]
        dp[0] = nums[0]
        for i in range(1, size):
            if i <= dp[i - 1]:
                dp[i] = max(dp[i - 1], i + nums[i])
            else:
                dp[i] = dp[i - 1]
        return dp[size - 1] >= size - 1
```

思路 2：复杂度分析

- 时间复杂度： $O(n)$ ，其中 n 是数组 nums 的长度。
- 空间复杂度： $O(n)$ 。

0056. 合并区间

- 标签：数组、排序
- 难度：中等

题目链接

- [0056. 合并区间 - 力扣](#)

题目大意

描述：给定数组 `intervals` 表示若干个区间的集合，其中单个区间为 `intervals[i] = [starti, endi]`。

要求：合并所有重叠的区间，并返回一个不重叠的区间数组，该数组需恰好覆盖输入中的所有区间。

说明：

- $1 \leq intervals.length \leq 10^4$ 。
- $intervals[i].length == 2$ 。
- $0 \leq starti \leq endi \leq 10^4$ 。

示例：

- **示例 1：**

输入: `intervals = [[1,3],[2,6],[8,10],[15,18]]`

输出: `[[1,6],[8,10],[15,18]]`

解释: 区间 `[1,3]` 和 `[2,6]` 重叠，将它们合并为 `[1,6]`.

- **示例 2：**

输入: `intervals = [[1,4],[4,5]]`

输出: `[[1,5]]`

解释: 区间 `[1,4]` 和 `[4,5]` 可被视为重叠区间。

解题思路

思路 1：排序

1. 设定一个数组 `ans` 用于表示最终不重叠的区间数组，然后对原始区间先按照区间左端点大小从小到大进行排序。
2. 遍历所有区间。
3. 先将第一个区间加入 `ans` 数组中。
4. 然后依次考虑后边的区间：
 - i. 如果第 `i` 个区间左端点在前一个区间右端点右侧，则这两个区间不会重合，直接将该区间加入 `ans` 数组中。

- ii. 否则的话，这两个区间重合，判断一下两个区间的右区间值，更新前一个区间的右区间值为较大值，然后继续考虑下一个区间，以此类推。
5. 最后返回数组 `ans`。

思路 1：代码

```
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        intervals.sort(key=lambda x: x[0])

        ans = []
        for interval in intervals:
            if not ans or ans[-1][1] < interval[0]:
                ans.append(interval)
            else:
                ans[-1][1] = max(ans[-1][1], interval[1])
        return ans
```

思路 1：复杂度分析

- 时间复杂度： $O(n \times \log_2 n)$ 。其中 n 为区间数量。
- 空间复杂度： $O(n)$ 。

0058. 最后一个单词的长度

- 标签：字符串
- 难度：简单

题目链接

- [0058. 最后一个单词的长度 - 力扣](#)

题目大意

给定一个字符串 s ，返回字符串中最后一个单词长度。

- 「单词」：指仅由字母组成、不包含任何空格字符的最大子字符串。

解题思路

从字符串末尾开始逆序遍历，先过滤掉末尾空白字符，然后统计字符数量，直到遇到空格或到达字符串开始位置。

代码

```
class Solution:
    def lengthOfLastWord(self, s: str) -> int:
        ans = 0
        for i in range(len(s)-1, -1, -1):
            if s[i] == " ":
                if ans == 0:
                    continue
                else:
                    return ans
            else:
                ans += 1
        return ans
```

0059. 螺旋矩阵 II

- 标签：数组、矩阵、模拟
- 难度：中等

题目链接

- [0059. 螺旋矩阵 II - 力扣](#)

题目大意

给你一个正整数 n 。

要求：生成一个包含 $1 \sim n^2$ 的所有元素，且元素按顺时针顺序螺旋排列的 $n \times n$ 正方形矩阵 $matrix$ 。

解题思路

思路 1：模拟

这道题跟「[54. 螺旋矩阵](#)」思路是一样的。

1. 构建一个 $n \times n$ 大小的数组 $matrix$ 存储答案。然后定义一下上、下、左、右的边界。
2. 然后按照逆时针的顺序从边界上依次给数组 $matrix$ 相应位置赋值。
3. 当访问完当前边界之后，要更新一下边界位置，缩小范围，方便下一轮进行访问。
4. 最后返回 $matrix$ 。

思路 1：代码

```
class Solution:
    def generateMatrix(self, n: int) -> List[List[int]]:
        matrix = [[0 for _ in range(n)] for _ in range(n)]
        up, down, left, right = 0, len(matrix) - 1, 0, len(matrix[0]) - 1
        index = 1
        while True:
            for i in range(left, right + 1):
                matrix[up][i] = index
                index += 1
            up += 1
            if up > down:
                break
            for i in range(up, down + 1):
                matrix[i][right] = index
                index += 1
            right -= 1
            if right < left:
                break
            for i in range(right, left - 1, -1):
                matrix[down][i] = index
                index += 1
            down -= 1
            if down < up:
                break
            for i in range(down, up - 1, -1):
                matrix[i][left] = index
                index += 1
            left += 1
            if left > right:
                break
        return matrix
```

思路 1：复杂度分析

- 时间复杂度： $O(n^2)$ 。
- 空间复杂度： $O(n^2)$ 。

0061. 旋转链表

- 标签：链表、双指针
- 难度：中等

题目链接

- [0061. 旋转链表 - 力扣](#)

题目大意

给定一个链表和整数 k ，将链表每个节点向右移动 k 个位置。

解题思路

我们可以将链表先连成环，然后将链表在指定位置断开。

先遍历一遍，求出链表节点个数 n 。注意到 k 可能很大，我们只需将链表右移 $k \% n$ 个位置即可。

第二次遍历到 $n - k \% n$ 的位置，记录下断开后新链表头节点位置，再将其断开并返回新的头节点。

代码

```
class Solution:
    def rotateRight(self, head: ListNode, k: int) -> ListNode:
        if k == 0 or not head or not head.next:
            return head
        curr = head
        count = 1
        while curr.next:
            count += 1
            curr = curr.next
        cut = count - k % count
        curr.next = head
        while cut:
            curr = curr.next
            cut -= 1
        newHead = curr.next
        curr.next = None
        return newHead
```

0062. 不同路径

- 标签：数学、动态规划、组合数学
- 难度：中等

题目链接

- [0062. 不同路径 - 力扣](#)

题目大意

描述：给定两个整数 m 和 n ，代表大小为 $m \times n$ 的棋盘，一个机器人位于棋盘左上角的位置，机器人每次只能向右、或者向下移动一步。

要求：计算出机器人从棋盘左上角到达棋盘右下角一共有多少条不同的路径。

说明：

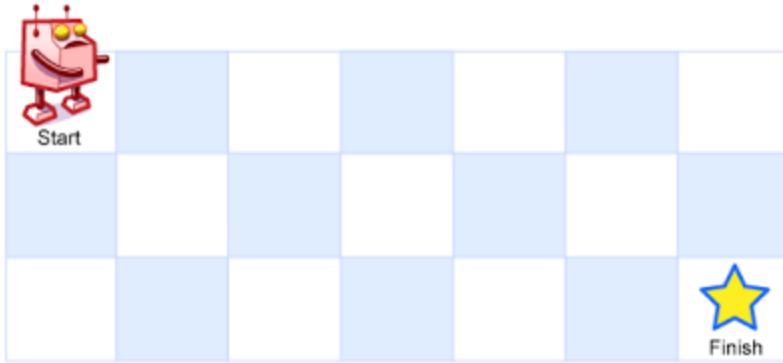
- $1 \leq m, n \leq 100$ 。
- 题目数据保证答案小于等于 2×10^9 。

示例：

- 示例 1：

输入： $m = 3, n = 7$

输出： 28



- 示例 2：

输入： $m = 3, n = 2$

输出： 3

解释：

从左上角开始，总共有 3 条路径可以到达右下角。

1. 向右 -> 向下 -> 向下
2. 向下 -> 向下 -> 向右
3. 向下 -> 向右 -> 向下

解题思路

思路 1：动态规划

1. 划分阶段

按照路径的结尾位置（行位置、列位置组成的二维坐标）进行阶段划分。

2. 定义状态

定义状态 $dp[i][j]$ 为：从左上角到达位置 (i, j) 的路径数量。

3. 状态转移方程

因为我们每次只能向右、或者向下移动一步，因此想要走到 (i, j) ，只能从 $(i - 1, j)$ 向下走一步走过来；或者从 $(i, j - 1)$ 向右走一步走过来。所以可以写出状态转移方程为： $dp[i][j] = dp[i - 1][j] + dp[i][j - 1]$ ，此时 $i > 0, j > 0$ 。

4. 初始条件

- 从左上角走到 $(0, 0)$ 只有一种方法，即 $dp[0][0] = 1$ 。
- 第一行元素只有一条路径（即只能通过前一个元素向右走得到），所以 $dp[0][j] = 1$ 。
- 同理，第一列元素只有一条路径（即只能通过前一个元素向下走得到），所以 $dp[i][0] = 1$ 。

5. 最终结果

根据状态定义，最终结果为 $dp[m - 1][n - 1]$ ，即从左上角到达右下角 $(m - 1, n - 1)$ 位置的路径数量为 $dp[m - 1][n - 1]$ 。

思路 1：动态规划代码

```
class Solution:  
    def uniquePaths(self, m: int, n: int) -> int:  
        dp = [[0 for _ in range(n)] for _ in range(m)]  
  
        for j in range(n):  
            dp[0][j] = 1  
        for i in range(m):  
            dp[i][0] = 1  
  
        for i in range(1, m):  
            for j in range(1, n):  
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1]  
  
        return dp[m - 1][n - 1]
```

思路 1：复杂度分析

- **时间复杂度：** $O(m \times n)$ 。初始条件赋值的时间复杂度为 $O(m + n)$ ，两重循环遍历的时间复杂度为 $O(m \times n)$ ，所以总体时间复杂度为 $O(m \times n)$ 。
- **空间复杂度：** $O(m \times n)$ 。用到了二维数组保存状态，所以总体空间复杂度为 $O(m \times n)$ 。因为 $dp[i][j]$ 的状态只依赖于上方值 $dp[i - 1][j]$ 和左侧值 $dp[i][j - 1]$ ，而我们在进行遍历时的顺序刚好是从上至下、从左到右。所以我们可以使用长度为 n 的一维数组来保存状态，从而将空间复杂度优化到 $O(n)$ 。# 0063. 不同路径 II

- 标签：数组、动态规划、矩阵
- 难度：中等

题目链接

- [0063. 不同路径 II - 力扣](#)

题目大意

描述：一个机器人位于一个 $m \times n$ 网格的左上角。机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角。但是网格中有障碍物，不能通过。

现在给定一个二维数组表示网格，1 代表障碍物，0 表示空位。

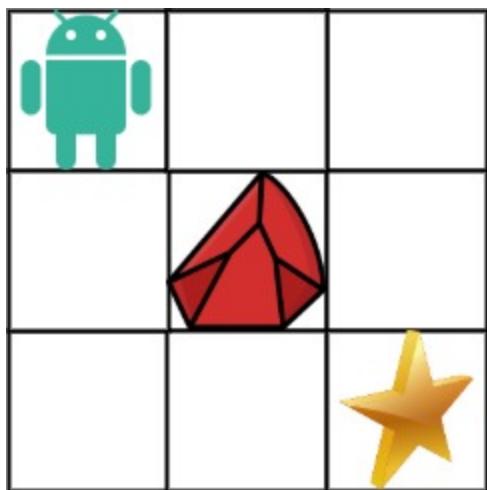
要求：计算出从左上角到右下角会有多少条不同的路径。

说明：

- $m == obstacleGrid.length$ 。
- $n == obstacleGrid[i].length$ 。
- $1 \leq m, n \leq 100$ 。
- $obstacleGrid[i][j]$ 为 0 或 1。

示例：

- **示例 1：**



输入: `obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]`

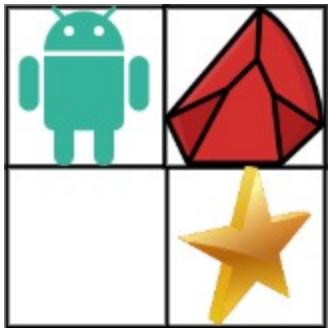
输出: `2`

解释: 3x3 网格的正中间有一个障碍物。

从左上角到右下角一共有 `2` 条不同的路径:

1. 向右 -> 向右 -> 向下 -> 向下
2. 向下 -> 向下 -> 向右 -> 向右

- 示例 2:



输入: `obstacleGrid = [[0,1],[0,0]]`

输出: `1`

解题思路

思路 1：动态规划

1. 划分阶段

按照路径的结尾位置（行位置、列位置组成的二维坐标）进行阶段划分。

2. 定义状态

定义状态 $dp[i][j]$ 表示为: 从 $(0, 0)$ 到 (i, j) 的不同路径数。

3. 状态转移方程

因为我们每次只能向右、或者向下移动一步，因此想要走到 (i, j) ，只能从 $(i - 1, j)$ 向下走一步走过来；或者从 $(i, j - 1)$ 向右走一步走过来。则状态转移方程为: $dp[i][j] = dp[i - 1][j] + dp[i][j - 1]$ ，其中 $obstacleGrid[i][j] == 0$ 。

4. 初始条件

- 对于第一行、第一列，因为只能朝一个方向走，所以 $dp[i][0] = 1$, $dp[0][j] = 1$ 。如果在第一行、第一列遇到障碍，则终止赋值，跳出循环。

5. 最终结果

根据我们之前定义的状态， $dp[i][j]$ 表示为：从 $(0, 0)$ 到 (i, j) 的不同路径数。所以最终结果为 $dp[m - 1][n - 1]$ 。

思路 1：代码

```
class Solution:
    def uniquePathsWithObstacles(self, obstacleGrid: List[List[int]]) -> int:
        m = len(obstacleGrid)
        n = len(obstacleGrid[0])
        dp = [[0 for _ in range(n)] for _ in range(m)]

        for i in range(m):
            if obstacleGrid[i][0] == 1:
                break
            dp[i][0] = 1

        for j in range(n):
            if obstacleGrid[0][j] == 1:
                break
            dp[0][j] = 1

        for i in range(1, m):
            for j in range(1, n):
                if obstacleGrid[i][j] == 1:
                    continue
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1]
        return dp[m - 1][n - 1]
```

思路 1：复杂度分析

- 时间复杂度： $O(m \times n)$ 。
- 空间复杂度： $O(m \times n)$ 。

0064. 最小路径和

- 标签：数组、动态规划、矩阵
- 难度：中等

题目链接

- 0064. 最小路径和 - 力扣

题目大意

描述：给定一个包含非负整数的 $m \times n$ 大小的网格 $grid$ 。

要求：找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

说明：

- 每次只能向下或者向右移动一步。
- $m == grid.length$ 。
- $n == grid[i].length$ 。
- $1 \leq m, n \leq 200$ 。
- $0 \leq grid[i][j] \leq 100$ 。

示例：

- **示例 1：**

1	3	1
1	5	1
4	2	1

输入： `grid = [[1,3,1],[1,5,1],[4,2,1]]`

输出： `7`

解释：因为路径 `1→3→1→1→1` 的总和最小。

- **示例 2：**

输入: `grid = [[1,2,3],[4,5,6]]`

输出: `12`

解题思路

思路 1：动态规划

1. 划分阶段

按照路径的结尾位置（行位置、列位置组成的二维坐标）进行阶段划分。

2. 定义状态

定义状态 $dp[i][j]$ 为：从左上角到达 (i, j) 位置的最小路径和。

3. 状态转移方程

当前位置 (i, j) 只能从左侧位置 $(i, j - 1)$ 或者上方位置 $(i - 1, j)$ 到达。为了使得从左上角到达 (i, j) 位置的最小路径和最小，应从 $(i, j - 1)$ 位置和 $(i - 1, j)$ 位置选择路径和最小的位置达到 (i, j) 。

即状态转移方程为： $dp[i][j] = \min(dp[i][j - 1], dp[i - 1][j]) + grid[i][j]$ 。

4. 初始条件

- 当左侧和上方是矩阵边界时（即 $i = 0, j = 0$ ）， $dp[i][j] = grid[i][j]$ 。
- 当只有左侧是矩阵边界时（即 $i \neq 0, j = 0$ ），只能从上方到达， $dp[i][j] = dp[i - 1][j] + grid[i][j]$ 。
- 当只有上方是矩阵边界时（即 $i = 0, j \neq 0$ ），只能从左侧到达， $dp[i][j] = dp[i][j - 1] + grid[i][j]$ 。

5. 最终结果

根据状态定义，最后输出 $dp[rows - 1][cols - 1]$ （即从左上角到达 $(rows - 1, cols - 1)$ 位置的最小路径和）即可。其中 $rows$ 、 $cols$ 分别为 $grid$ 的行数、列数。

思路 1：代码

```
class Solution:
    def minPathSum(self, grid: List[List[int]]) -> int:
        rows, cols = len(grid), len(grid[0])
        dp = [[0 for _ in range(cols)] for _ in range(rows)]

        dp[0][0] = grid[0][0]

        for i in range(1, rows):
            dp[i][0] = dp[i - 1][0] + grid[i][0]

        for j in range(1, cols):
            dp[0][j] = dp[0][j - 1] + grid[0][j]

        for i in range(1, rows):
            for j in range(1, cols):
                dp[i][j] = min(dp[i][j - 1], dp[i - 1][j]) + grid[i][j]

        return dp[rows - 1][cols - 1]
```

思路 1：复杂度分析

- 时间复杂度： $O(m * n)$ ，其中 m 、 n 分别为 $grid$ 的行数和列数。
- 空间复杂度： $O(m * n)$ 。

0066. 加一

- 标签：数组、数学
- 难度：简单

题目链接

- [0066. 加一 - 力扣](#)

题目大意

描述：给定一个非负整数数组，数组每一位对应整数的一位数字。

要求：计算整数加 1 后的结果。

说明：

- $1 \leq digits.length \leq 100$ 。
- $0 \leq digits[i] \leq 9$ 。

示例：

- **示例 1：**

输入: `digits = [1,2,3]`

输出: `[1,2,4]`

解释: 输入数组表示数字 `123`, 加 `1` 之后为 `124`。

- **示例 2：**

输入: `digits = [4,3,2,1]`

输出: `[4,3,2,2]`

解释: 输入数组表示数字 `4321`。

解题思路

思路 1：模拟

这道题把整个数组看成了一个整数，然后个位数加 1。问题的实质是利用数组模拟加法运算。

如果个位数不为 9 的话，直接把个位数加 1 就好。如果个位数为 9 的话，还要考虑进位。

具体步骤：

1. 数组前补 0 位。
2. 将个位数字进行加 1 计算。
 - i. 如果该位数字大于等于 10，则向后一位进 1，继续后一位判断进位。
 - ii. 如果该位数字小于 10，则跳出循环。

思路 1：代码

```
def plusOne(self, digits: List[int]) -> List[int]:  
    digits = [0] + digits  
    digits[len(digits) - 1] += 1  
    for i in range(len(digits)-1, 0, -1):  
        if digits[i] != 10:  
            break  
        else:  
            digits[i] = 0  
            digits[i - 1] += 1  
  
    if digits[0] == 0:  
        return digits[1:]  
    else:  
        return digits
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。一重循环遍历的时间复杂度为 $O(n)$ 。
- 空间复杂度： $O(1)$ 。[# 0067. 二进制求和](#)
- 标签：位运算、数学、字符串、模拟
- 难度：简单

题目链接

- [0067. 二进制求和 - 力扣](#)

题目大意

给定两个二进制数的字符串 a 、 b 。计算 a 和 b 的和，返回结果也用二进制表示。

解题思路

这道题可以直接将 a 、 b 转换为十进制数，相加后再转换为二进制数。

也可以利用位运算的一些知识，直接求和。

因为 a 、 b 为二进制的字符串，先将其转换为二进制数。

本题用到的位运算知识：

- 异或运算 $x \wedge y$ ：可以获得 $x + y$ 无进位的加法结果。
- 与运算 $x \& y$ ：对应位置为 1，说明 x 、 y 该位置上原来都为 1，则需要进位。
- 座椅运算 $x << 1$ ：将 a 对应二进制数左移 1 位。

这样，通过 $x \wedge y$ 运算，我们可以得到相加后无进位结果，再根据 $(x \& y) << 1$ ，计算进位后结果。

进行 $x \wedge y$ 和 $(x \& y) << 1$ 操作之后判断进位是否为 0，若不为 0，则继续上一步操作，直到进位为 0。

最后将其结果转为 2 进制返回。

代码

```
class Solution:  
    def addBinary(self, a: str, b: str) -> str:  
        x = int(a, 2)  
        y = int(b, 2)  
        ans = 0  
        while y:  
            carry = ((x & y) << 1)  
            x ^= y  
            y = carry  
        return bin(x)[2:]
```

0069. x 的平方根

- 标签：数学、二分查找
- 难度：简单

题目链接

- [0069. x 的平方根 - 力扣](#)

题目大意

要求：实现 `int sqrt(int x)` 函数。计算并返回 x 的平方根（只保留整数部分），其中 x 是非负整数。

说明：

- $0 \leq x \leq 2^{31} - 1$ 。

示例：

- 示例 1：

输入：`x = 4`

输出：`2`

- 示例 2：

输入：`x = 8`

输出：`2`

解释：`8` 的算术平方根是 `2.82842...`，由于返回类型是整数，小数部分将被舍去。

解题思路

思路 1：二分查找

因为求解的是 x 开方的整数部分。所以我们可以从 $0 \sim x$ 的范围进行遍历，找到 $k^2 \leq x$ 的最大结果。

为了减少算法的时间复杂度，我们使用二分查找的方法来搜索答案。

思路 1：代码

```
class Solution:
    def mySqrt(self, x: int) -> int:
        left = 0
        right = x
        ans = -1
        while left <= right:
            mid = (left + right) // 2
            if mid * mid <= x:
                ans = mid
                left = mid + 1
            else:
                right = mid - 1
        return ans
```

思路 1：复杂度分析

- **时间复杂度**: $O(\log n)$ 。二分查找算法的时间复杂度为 $O(\log n)$ 。
- **空间复杂度**: $O(1)$ 。只用到了常数空间存放若干变量。

0070. 爬楼梯

- 标签：记忆化搜索、数学、动态规划
- 难度：简单

题目链接

- [0070. 爬楼梯 - 力扣](#)

题目大意

描述：假设你正在爬楼梯。需要 n 阶你才能到达楼顶。每次你可以爬 1 或 2 个台阶。现在给定一个整数 n 。

要求：计算出有多少种不同的方法可以爬到楼顶。

说明：

- $1 \leq n \leq 45$ 。

示例：

- 示例 1：

输入：n = 2

输出：2

解释：有两种方法可以爬到楼顶。

1. 1 阶 + 1 阶
2. 2 阶

- 示例 2：

输入：n = 3

输出：3

解释：有三种方法可以爬到楼顶。

1. 1 阶 + 1 阶 + 1 阶
2. 1 阶 + 2 阶
3. 2 阶 + 1 阶

解题思路

思路 1：递归（超时）

根据我们的递推三步走策略，写出对应的递归代码。

1. 写出递推公式： $f(n) = f(n - 1) + f(n - 2)$ 。
2. 明确终止条件： $f(0) = 0, f(1) = 1$ 。
3. 翻译为递归代码：

i. 定义递归函数：climbStairs(self, n) 表示输入参数为问题的规模 n，返回结果为爬 n 阶台阶到达楼顶的方案数。

ii. 书写递归主体：return self.climbStairs(n - 1) + self.climbStairs(n - 2)。

iii. 明确递归终止条件：

- a. if n == 0: return 0
- b. if n == 1: return 1

思路 1：代码

```
class Solution:
    def climbStairs(self, n: int) -> int:
        if n == 1:
            return 1
        if n == 2:
            return 2
        return self.climbStairs(n - 1) + self.climbStairs(n - 2)
```

思路 1：复杂度分析

- 时间复杂度： $O((\frac{1+\sqrt{5}}{2})^n)$ 。
- 空间复杂度： $O(n)$ 。每次递归的空间复杂度是 $O(1)$ ，调用栈的深度为 n ，所以总的空间复杂度就是 $O(n)$ 。

思路 2：动态规划

1. 划分阶段

按照台阶的层数进行划分为 $0 \sim n$ 。

2. 定义状态

定义状态 $dp[i]$ 为：爬到第 i 阶台阶的方案数。

3. 状态转移方程

根据题目大意，每次只能爬 1 或 2 个台阶。则第 i 阶楼梯只能从第 $i - 1$ 阶向上爬 1 阶上来，或者从第 $i - 2$ 阶向上爬 2 阶上来。所以可以推出状态转移方程为 $dp[i] = dp[i - 1] + dp[i - 2]$ 。

4. 初始条件

- 第 0 层台阶方案数：可以看做 1 种方法（从 0 阶向上爬 0 阶），即 $dp[0] = 1$ 。
- 第 1 层台阶方案数：1 种方法（从 0 阶向上爬 1 阶），即 $dp[1] = 1$ 。
- 第 2 层台阶方案数：2 种方法（从 0 阶向上爬 2 阶，或者从 1 阶向上爬 1 阶）。

5. 最终结果

根据状态定义，最终结果为 $dp[n]$ ，即爬到第 n 阶台阶（即楼顶）的方案数为 $dp[n]$ 。

思路 2：代码

```
class Solution:
    def climbStairs(self, n: int) -> int:
        dp = [0 for _ in range(n + 1)]
        dp[0] = 1
        dp[1] = 1
        for i in range(2, n + 1):
            dp[i] = dp[i - 1] + dp[i - 2]

        return dp[n]
```

思路 2：复杂度分析

- **时间复杂度**: $O(n)$ 。一重循环遍历的时间复杂度为 $O(n)$ 。
- **空间复杂度**: $O(n)$ 。用到了一维数组保存状态，所以总体空间复杂度为 $O(n)$ 。因为 $dp[i]$ 的状态只依赖于 $dp[i - 1]$ 和 $dp[i - 2]$ ，所以可以使用 3 个变量来分别表示 $dp[i]$ 、 $dp[i - 1]$ 、 $dp[i - 2]$ ，从而将空间复杂度优化到 $O(1)$ 。

0072. 编辑距离

- 标签：字符串、动态规划
- 难度：困难

题目链接

- [0072. 编辑距离 - 力扣](#)

题目大意

描述：给定两个单词 $word1$ 、 $word2$ 。

对一个单词可以进行以下三种操作：

- 插入一个字符
- 删一个字符
- 替换一个字符

要求：计算出将 $word1$ 转换为 $word2$ 所使用的最少操作数。

说明：

- $0 \leq word1.length, word2.length \leq 500$ 。
- $word1$ 和 $word2$ 由小写英文字母组成。

示例：

- 示例 1：

输入: $word1 = "horse"$, $word2 = "ros"$

输出: 3

解释:

$horse \rightarrow rorse$ (将 'h' 替换为 'r')

$rorse \rightarrow rose$ (删除 'r')

$rose \rightarrow ros$ (删除 'e')

- 示例 2：

输入: $word1 = "intention"$, $word2 = "execution"$

输出: 5

解释:

$intention \rightarrow inention$ (删除 't')

$inention \rightarrow enention$ (将 'i' 替换为 'e')

$enention \rightarrow exention$ (将 'n' 替换为 'x')

$exention \rightarrow exection$ (将 'n' 替换为 'c')

$exection \rightarrow execution$ (插入 'u')

解题思路

思路 1：动态规划

1. 划分阶段

按照两个字符串的结尾位置进行阶段划分。

2. 定义状态

定义状态 $dp[i][j]$ 表示为：「以 $word1$ 中前 i 个字符组成的子字符串 $str1$ 」变为「以 $word2$ 中前 j 个字符组成的子字符串 $str2$ 」，所需要的最少操作次数。

3. 状态转移方程

1. 如果当前字符相同 ($word1[i - 1] = word2[j - 1]$)，无需插入、删除、替换。 $dp[i][j] = dp[i - 1][j - 1]$ 。
2. 如果当前字符不同 ($word1[i - 1] \neq word2[j - 1]$)， $dp[i][j]$ 取源于以下三种情况中的最小情况：
 - i. 替换 ($word1[i - 1]$ 替换为 $word2[j - 1]$)：最少操作次数依赖于「以 $word1$ 中前 $i - 1$ 个字符组成的子字符串 $str1$ 」变为「以 $word2$ 中前 $j - 1$ 个字符组成的子字符串 $str2$ 」，再加上替换的操作数 1，即： $dp[i][j] = dp[i - 1][j - 1] + 1$ 。
 - ii. 插入 ($word1$ 在第 $i - 1$ 位置上插入元素)：最少操作次数依赖于「以 $word1$ 中前 $i - 1$ 个字符组成的子字符串 $str1$ 」变为「以 $word2$ 中前 j 个字符组成的子字符串 $str2$ 」，再加上插入需要的操作数 1，即： $dp[i][j] = dp[i - 1][j] + 1$ 。
 - iii. 删除 ($word1$ 删除第 $i - 1$ 位置元素)：最少操作次数依赖于「以 $word1$ 中前 i 个字符组成的子字符串 $str1$ 」变为「以 $word2$ 中前 $j - 1$ 个字符组成的子字符串 $str2$ 」，再加上删除需要的操作数 1，即： $dp[i][j] = dp[i][j - 1] + 1$ 。

综合上述情况，状态转移方程为：

$$dp[i][j] = \begin{cases} dp[i - 1][j - 1] & word1[i - 1] = word2[j - 1] \\ \min(dp[i - 1][j - 1], dp[i - 1][j], dp[i][j - 1]) + 1 & word1[i - 1] \neq word2[j - 1] \end{cases}$$

4. 初始条件

- 当 $i = 0$ ，「以 $word1$ 中前 i 个字符组成的子字符串 $str1$ 」为空字符串，「 $str1$ 」变为「以 $word2$ 中前 j 个字符组成的子字符串 $str2$ 」时，至少需要插入 j 次，即： $dp[0][j] = j$ 。
- 当 $j = 0$ ，「以 $word2$ 中前 j 个字符组成的子字符串 $str2$ 」为空字符串，「以 $word1$ 中前 i 个字符组成的子字符串 $str1$ 」变为「 $str2$ 」时，至少需要删除 i 次，即： $dp[i][0] = i$ 。

5. 最终结果

根据状态定义，最后输出 $dp[size1][size2]$ （即 $word1$ 变为 $word2$ 所使用的最少操作数）即可。其中 $size1$ 、 $size2$ 分别为 $word1$ 、 $word2$ 的字符串长度。

思路 1：代码

```
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        size1 = len(word1)
        size2 = len(word2)
        dp = [[0 for _ in range(size2 + 1)] for _ in range(size1 + 1)]

        for i in range(size1 + 1):
            dp[i][0] = i
        for j in range(size2 + 1):
            dp[0][j] = j
        for i in range(1, size1 + 1):
            for j in range(1, size2 + 1):
                if word1[i - 1] == word2[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1]
                else:
                    dp[i][j] = min(dp[i - 1][j - 1], dp[i - 1][j], dp[i][j - 1]) + 1
        return dp[size1][size2]
```

思路 1：复杂度分析

- **时间复杂度**: $O(n \times m)$, 其中 n 、 m 分别是字符串 $word1$ 、 $word2$ 的长度。两重循环遍历的时间复杂度是 $O(n \times m)$, 所以总的时间复杂度为 $O(n \times m)$ 。
- **空间复杂度**: $O(n \times m)$ 。用到了二维数组保存状态，所以总体空间复杂度为 $O(n \times m)$ 。

0073. 矩阵置零

- 标签：数组、哈希表、矩阵
- 难度：中等

题目链接

- [0073. 矩阵置零 - 力扣](#)

题目大意

描述：给定一个 $m \times n$ 大小的矩阵 $matrix$ 。

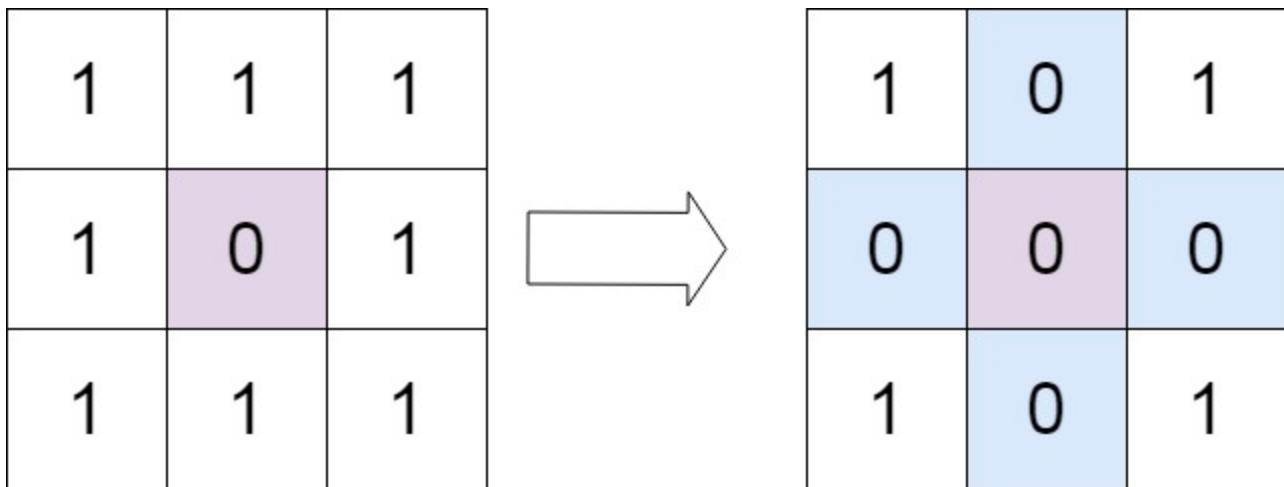
要求: 如果一个元素为 0，则将其所在行和列所有元素都置为 0。

说明:

- 请使用「原地」算法。
- $m == matrix.length$ 。
- $n == matrix[0].length$ 。
- $1 \leq m, n \leq 200$ 。
- $-2^{31} \leq matrix[i][j] \leq 2^{31} - 1$ 。
- **进阶:**
 - 一个直观的解决方案是使用 $O(m \times n)$ 的额外空间，但这并不是一个好的解决方案。
 - 一个简单的改进方案是使用 $O(m + n)$ 的额外空间，但这仍然不是最好的解决方案。
 - 你能想出一个仅使用常量空间的解决方案吗？

示例:

- **示例 1:**



输入: `matrix = [[1,1,1],[1,0,1],[1,1,1]]`

输出: `[[1,0,1],[0,0,0],[1,0,1]]`

- **示例 2:**

0	1	2	0
3	4	5	2
1	3	1	5

0	0	0	0
0	4	5	0
0	3	1	0

输入: `matrix = [[1,1,1],[1,0,1],[1,1,1]]`
 输出: `[[1,0,1],[0,0,0],[1,0,1]]`

解题思路

思路 1：使用标记变量

直观上可以使用两个数组来标记行和列出现 0 的情况，但这样空间复杂度就是 $O(m + n)$ 了，不符合题意。

考虑使用数组原本的元素进行记录出现 0 的情况。

1. 设定两个变量 `flag_row0`、`flag_col0` 来标记第一行、第一列是否出现了 0。
2. 接下来我们使用数组第一行、第一列来标记 0 的情况。
3. 对数组除第一行、第一列之外的每个元素进行遍历，如果某个元素出现 0 了，则使用数组的第一行、第一列对应位置来存储 0 的标记。
4. 再对数组除第一行、第一列之外的每个元素进行遍历，通过对第一行、第一列的标记 0 情况，进行置为 0 的操作。
5. 最后再根据 `flag_row0`、`flag_col0` 的标记情况，对第一行、第一列进行置为 0 的操作。

思路 1：代码

```
class Solution:
    def setZeroes(self, matrix: List[List[int]]) -> None:
        m = len(matrix)
        n = len(matrix[0])
        flag_col0 = False
        flag_row0 = False
        for i in range(m):
            if matrix[i][0] == 0:
                flag_col0 = True
                break

        for j in range(n):
            if matrix[0][j] == 0:
                flag_row0 = True
                break

        for i in range(1, m):
            for j in range(1, n):
                if matrix[i][j] == 0:
                    matrix[i][0] = matrix[0][j] = 0

        for i in range(1, m):
            for j in range(1, n):
                if matrix[i][0] == 0 or matrix[0][j] == 0:
                    matrix[i][j] = 0

        if flag_col0:
            for i in range(m):
                matrix[i][0] = 0

        if flag_row0:
            for j in range(n):
                matrix[0][j] = 0
```

思路 1：复杂度分析

- 时间复杂度： $O(m \times n)$ 。
- 空间复杂度： $O(1)$ 。

0074. 搜索二维矩阵

- 标签：数组、二分查找、矩阵
- 难度：中等

题目链接

- [0074. 搜索二维矩阵 - 力扣](#)

题目大意

描述：给定一个 $m \times n$ 大小的有序二维矩阵 $matrix$ 。矩阵中每行元素从左到右升序排列，每列元素从上到下升序排列。再给定一个目标值 $target$ 。

要求：判断矩阵中是否存在目标值 $target$ 。

说明：

- $m == matrix.length$ 。
- $n == matrix[i].length$ 。
- $1 \leq m, n \leq 100$ 。
- $-10^4 \leq matrix[i][j], target \leq 10^4$ 。

示例：

- 示例 1：

1	3	5	7
10	11	16	20
23	30	34	60

输入: `matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]]`, `target = 3`

输出: `True`

- 示例 2:

1	3	5	7
10	11	16	20
23	30	34	60

输入: `matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]]`, `target = 13`

输出: `False`

解题思路

思路 1：二分查找

二维矩阵是有序的，可以考虑使用二分搜索来进行查找。

1. 首先二分查找遍历对角线元素，假设对角线元素的坐标为 (row, col) 。把数组元素按对角线分为右上角部分和左下角部分。
2. 然后对于当前对角线元素右侧第 row 行、对角线元素下侧第 col 列进行二分查找。
 - i. 如果找到目标，直接返回 `True`。
 - ii. 如果找不到目标，则缩小范围，继续查找。
 - iii. 直到所有对角线元素都遍历完，依旧没找到，则返回 `False`。

思路 1：代码

```
class Solution:
    # 二分查找对角线元素
    def diagonalBinarySearch(self, matrix, diagonal, target):
        left = 0
        right = diagonal
        while left < right:
            mid = left + (right - left) // 2
            if matrix[mid][mid] < target:
                left = mid + 1
            else:
                right = mid
        return left

    def rowBinarySearch(self, matrix, begin, cols, target):
        left = begin
        right = cols
        while left < right:
            mid = left + (right - left) // 2
            if matrix[begin][mid] < target:
                left = mid + 1
            elif matrix[begin][mid] > target:
                right = mid - 1
            else:
                left = mid
                break
        return begin <= left <= cols and matrix[begin][left] == target

    def colBinarySearch(self, matrix, begin, rows, target):
        left = begin + 1
        right = rows
        while left < right:
            mid = left + (right - left) // 2
            if matrix[mid][begin] < target:
                left = mid + 1
            elif matrix[mid][begin] > target:
                right = mid - 1
            else:
                left = mid
                break
        return begin <= left <= rows and matrix[left][begin] == target
```

```

def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
    rows = len(matrix)
    if rows == 0:
        return False
    cols = len(matrix[0])
    if cols == 0:
        return False

    min_val = min(rows, cols)
    index = self.diagonalBinarySearch(matrix, min_val - 1, target)
    if matrix[index][index] == target:
        return True
    for i in range(index + 1):
        row_search = self.rowBinarySearch(matrix, i, cols - 1, target)
        col_search = self.colBinarySearch(matrix, i, rows - 1, target)
        if row_search or col_search:
            return True
    return False

```

思路 1：复杂度分析

- 时间复杂度： $O(\log m + \log n)$ ，其中 m 、 n 分别是矩阵的行数和列数。
- 空间复杂度： $O(1)$ 。

0075. 颜色分类

- 标签：数组、双指针、排序
- 难度：中等

题目链接

- [0075. 颜色分类 - 力扣](#)

题目大意

描述：给定一个数组 $nums$ ，元素值只有 0、1、2，分别代表红色、白色、蓝色。

要求：将数组进行排序，使得红色在前，白色在中间，蓝色在最后。

说明:

- 要求不使用标准库函数，同时仅用常数空间，一趟扫描解决。
- $n == \text{nums.length}$ 。
- $1 \leq n \leq 300$ 。
- $\text{nums}[i]$ 为 0、1 或 2。

示例:

- 示例 1:

输入: `nums = [2,0,2,1,1,0]`
输出: `[0,0,1,1,2,2]`

- 示例 2:

输入: `nums = [2,0,1]`
输出: `[0,1,2]`

解题思路

思路 1：双指针 + 快速排序思想

快速排序算法中的 *partition* 过程，利用双指针，将序列中比基准数 *pivot* 大的元素移动到了基准数右侧，将比基准数 *pivot* 小的元素移动到了基准数左侧。从而将序列分为了三部分：比基准数小的部分、基准数、比基准数大的部分。

这道题我们也可以借鉴快速排序算法中的 *partition* 过程，将 1 作为基准数 *pivot*，然后将序列分为三部分：0（即比 1 小的部分）、等于 1 的部分、2（即比 1 大的部分）。具体步骤如下：

1. 使用两个指针 *left*、*right*，分别指向数组的头尾。*left* 表示当前处理好红色元素的尾部，*right* 表示当前处理好蓝色的头部。
2. 再使用一个下标 *index* 遍历数组，如果遇到 $\text{nums}[\text{index}] == 0$ ，就交换 $\text{nums}[\text{index}]$ 和 $\text{nums}[\text{left}]$ ，同时将 *left* 右移。如果遇到 $\text{nums}[\text{index}] == 2$ ，就交换 $\text{nums}[\text{index}]$ 和 $\text{nums}[\text{right}]$ ，同时将 *right* 左移。
3. 直到 *index* 移动到 *right* 位置之后，停止遍历。遍历结束之后，此时 *left* 左侧都是红色，*right* 右侧都是蓝色。

注意：移动的时候需要判断 $index$ 和 $left$ 的位置，因为 $left$ 左侧是已经处理好的数组，所以需要判断 $index$ 的位置是否小于 $left$ ，小于的话，需要更新 $index$ 位置。

思路 1：代码

```
class Solution:
    def sortColors(self, nums: List[int]) -> None:
        left = 0
        right = len(nums) - 1
        index = 0
        while index <= right:
            if index < left:
                index += 1
            elif nums[index] == 0:
                nums[index], nums[left] = nums[left], nums[index]
                left += 1
            elif nums[index] == 2:
                nums[index], nums[right] = nums[right], nums[index]
                right -= 1
            else:
                index += 1
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(1)$ 。

0076. 最小覆盖子串

- 标签：哈希表、字符串、滑动窗口
- 难度：困难

题目链接

- [0076. 最小覆盖子串 - 力扣](#)

题目大意

描述：给定一个字符串 s 、一个字符串 t 。

要求：返回 s 中涵盖 t 所有字符的最小子串。如果 s 中不存在涵盖 t 所有字符的子串，则返回空字符串 ""。

说明：

- $1 \leq s.length, t.length \leq 10^5$ 。
- s 和 t 由英文字母组成。

示例：

- **示例 1：**

输入: $s = "ADOBECODEBANC"$, $t = "ABC"$

输出: "**BANC**"

- **示例 2：**

输入: $s = "a"$, $t = "a"$

输出: "**a**"

解题思路

思路 1：滑动窗口

1. $left$ 、 $right$ 表示窗口的边界，一开始都位于下标 0 处。 $need$ 用于记录短字符串需要的字符数。 $window$ 记录当前窗口内的字符数。
2. 将 $right$ 右移，直到出现了 t 中全部字符，开始右移 $left$ ，减少滑动窗口的大小，并记录下最小覆盖子串的长度和起始位置。
3. 最后输出结果。

思路 1：代码

```
import collections

class Solution:
    def minWindow(self, s: str, t: str) -> str:
        need = collections.defaultdict(int)
        window = collections.defaultdict(int)
        for ch in t:
            need[ch] += 1

        left, right = 0, 0
        valid = 0
        start = 0
        size = len(s) + 1

        while right < len(s):
            insert_ch = s[right]
            right += 1

            if insert_ch in need:
                window[insert_ch] += 1
                if window[insert_ch] == need[insert_ch]:
                    valid += 1

            while valid == len(need):
                if right - left < size:
                    start = left
                    size = right - left
                remove_ch = s[left]
                left += 1
                if remove_ch in need:
                    if window[remove_ch] == need[remove_ch]:
                        valid -= 1
                    window[remove_ch] -= 1
                if size == len(s) + 1:
                    return ''
        return s[start:start+size]
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。其中 n 是字符串 s 的长度。

- 空间复杂度: $O(|\Sigma|)$ 。 $|\Sigma|$ 是 s 和 t 的字符集大小。# 0077. 组合
- 标签: 回溯
- 难度: 中等

题目链接

- 0077. 组合 - 力扣

题目大意

给定两个整数 n 和 k ，返回范围 $[1, n]$ 中所有可能的 k 个数的组合。可以按任何顺序返回答案。

解题思路

组合问题通常可以用回溯算法来解决。定义两个数组 res 、 $path$ 。 res 用来存放最终答案， $path$ 用来存放当前符合条件的一个结果。再使用一个变量 $start_index$ 来表示从哪一个数开始遍历。

定义回溯方法， $start_index = 1$ 开始进行回溯。

- 如果 $path$ 数组的长度等于 k ，则将 $path$ 中的元素加入到 res 数组中。
- 然后对 $[start_index, n]$ 范围内的数进行遍历取值。
 - 将当前元素 i 加入 $path$ 数组。
 - 递归遍历 $[start_index, n]$ 上的数。
 - 将遍历的 i 元素进行回退。
- 最终返回 res 数组。

代码

```
class Solution:
    res = []
    path = []
    def backtrack(self, n: int, k: int, start_index: int):
        if len(self.path) == k:
            self.res.append(self.path[:])
            return
        for i in range(start_index, n - (k - len(self.path)) + 2):
            self.path.append(i)
            self.backtrack(n, k, i + 1)
            self.path.pop()

    def combine(self, n: int, k: int) -> List[List[int]]:
        self.res.clear()
        self.path.clear()
        self.backtrack(n, k, 1)
        return self.res
```

0078. 子集

- 标签：位运算、数组、回溯
- 难度：中等

题目链接

- [0078. 子集 - 力扣](#)

题目大意

描述：给定一个整数数组 `nums`，数组中的元素互不相同。

要求：返回该数组所有可能的不重复子集。可以按任意顺序返回解集。

说明：

- $1 \leq \text{nums.length} \leq 10$ 。

- $-10 \leq \text{nums}[i] \leq 10$ 。
- `nums` 中的所有元素互不相同。

示例：

- 示例 1：

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

- 示例 2：

输入：`nums = [0]`
输出：`[[], [0]]`

解题思路

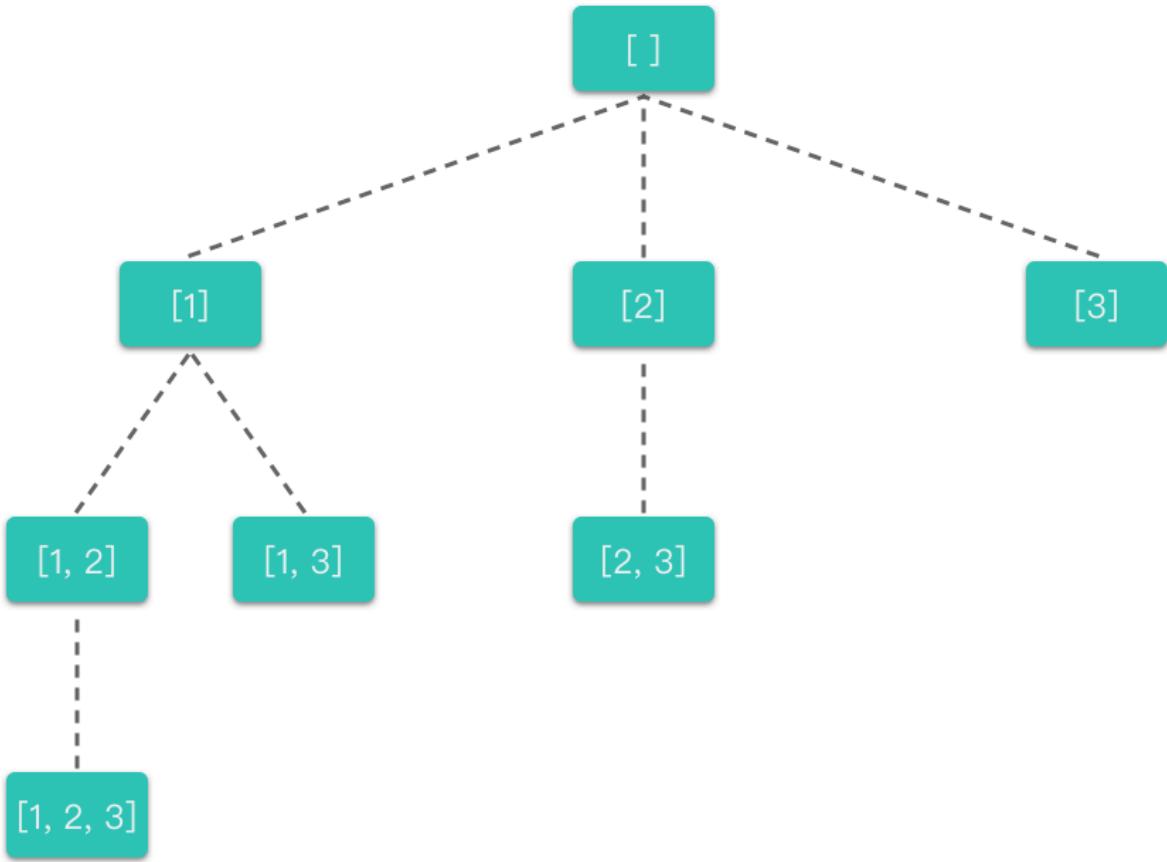
思路 1：回溯算法

数组的每个元素都有两个选择：选与不选。

我们可以通过向当前子集数组中添加可选元素来表示选择该元素。也可以在当前递归结束之后，将之前添加的元素从当前子集数组中移除（也就是回溯）来表示不选择该元素。

下面我们根据回溯算法三步走，写出对应的回溯算法。

1. **明确所有选择：**根据数组中每个位置上的元素选与不选两种选择，画出决策树，如下图所示。



2. 明确终止条件：

- 当遍历到决策树的叶子节点时，就终止了。即当前路径搜索到末尾时，递归终止。

3. 将决策树和终止条件翻译成代码：

i. 定义回溯函数：

- `backtracking(nums, index)`: 函数的传入参数是 `nums` (可选数组列表) 和 `index` (代表目前正在考虑元素是 `nums[i]`)，全局变量是 `res` (存放所有符合条件结果的集合数组) 和 `path` (存放当前符合条件的结果)。
- `backtracking(nums, index)`: 函数代表的含义是：在选择 `nums[index]` 的情况下，递归选择剩下的元素。

ii. 书写回溯函数主体 (给出选择元素、递归搜索、撤销选择部分)。

- 从目前正在考虑元素，到数组结束为止，枚举出所有可选的元素。对于每一个可选元素：
 - 约束条件：之前选过的元素不再重复选用。每次从 `index` 位置开始遍历而不是从 `0` 位置开始遍历就是为了避免重复。集合跟全排列不一样，子集中 `{1, 2}` 和 `{2, 1}` 是等价的。为了避免重复，我们之前考虑过的元素，就不再重复考虑了。
 - 选择元素：将其添加到当前子集数组 `path` 中。
 - 递归搜索：在选择该元素的情况下，继续递归考虑下一个位置上的元素。
 - 撤销选择：将该元素从当前子集数组 `path` 中移除。

```

for i in range(index, len(nums)):
    # 枚举可选元素列表
    path.append(nums[i])
    # 选择元素
    backtracking(nums, i + 1)
    # 递归搜索
    path.pop()
    # 撤销选择

```

iii. 明确递归终止条件（给出递归终止条件，以及递归终止时的处理方法）。

- 当遍历到决策树的叶子节点时，就终止了。也就是当正在考虑的元素位置到达数组末尾（即 `start >= len(nums)`）时，递归停止。
- 从决策树中也可以看出，子集需要存储的答案集合应该包含决策树上所有的节点，应该需要保存递归搜索的所有状态。所以无论是否达到终止条件，我们都应该将当前符合条件的结果放入到集合中。

思路 1：代码

```

class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        res = [] # 存放所有符合条件结果的集合
        path = [] # 存放当前符合条件的结果
        def backtracking(nums, index): # 正在考虑可选元素列表中第 index 个元素
            res.append(path[:]) # 将当前符合条件的结果放入集合中
            if index >= len(nums): # 遇到终止条件（本题）
                return

            for i in range(index, len(nums)): # 枚举可选元素列表
                path.append(nums[i]) # 选择元素
                backtracking(nums, i + 1) # 递归搜索
                path.pop() # 撤销选择

        backtracking(nums, 0)
        return res

```

思路 1：复杂度分析

- **时间复杂度**: $O(n \times 2^n)$, 其中 n 指的是数组 `nums` 的元素个数, 2^n 指的是所有状态数。每种状态需要 $O(n)$ 的时间来构造子集。
- **空间复杂度**: $O(n)$, 每种状态下构造子集需要使用 $O(n)$ 的空间。

思路 2：二进制枚举

对于一个元素个数为 n 的集合 nums 来说，每一个位置上的元素都有选取和未选取两种状态。我们可以用数字 1 来表示选取该元素，用数字 0 来表示不选取该元素。

那么我们就可以用一个长度为 n 的二进制数来表示集合 nums 或者表示 nums 的子集。其中二进制的每一位数都对应了集合中某一个元素的选取状态。对于集合中第 i 个元素（ i 从 0 开始编号）来说，二进制对应位置上的 1 代表该元素被选取， 0 代表该元素未被选取。

举个例子来说明一下，比如长度为 5 的集合 $\text{nums} = \{5, 4, 3, 2, 1\}$ ，我们可以用一个长度为 5 的二进制数来表示该集合。

比如二进制数 11111 就表示选取集合的第 0 位、第 1 位、第 2 位、第 3 位、第 4 位元素，也就是集合 $\{5, 4, 3, 2, 1\}$ ，即集合 nums 本身。如下表所示：

集合 nums 对应位置（下标）	4	3	2	1	0
二进制数对应位数	1	1	1	1	1
对应选取状态	选取	选取	选取	选取	选取

再比如二进制数 10101 就表示选取集合的第 0 位、第 2 位、第 5 位元素，也就是集合 $\{5, 3, 1\}$ 。如下表所示：

集合 nums 对应位置（下标）	4	3	2	1	0
二进制数对应位数	1	0	1	0	1
对应选取状态	选取	未选取	选取	未选取	选取

再比如二进制数 01001 就表示选取集合的第 0 位、第 3 位元素，也就是集合 $\{5, 2\}$ 。如下标所示：

集合 nums 对应位置（下标）	4	3	2	1	0
二进制数对应位数	0	1	0	0	1
对应选取状态	未选取	选取	未选取	未选取	选取

通过上面的例子我们可以得到启发：对于长度为 5 的集合 nums 来说，我们只需要从 $00000 \sim 11111$ 枚举一次（对应十进制为 $0 \sim 2^4 - 1$ ）即可得到长度为 5 的集合 s 的所有子集。

我们将上面的例子拓展到长度为 n 的集合 nums 。可以总结为：

- 对于长度为 5 的集合 `nums` 来说，只需要枚举 $0 \sim 2^n - 1$ (共 2^n 种情况)，即可得到所有的子集。

思路 2：代码

```

class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        n = len(nums)                                # n 为集合 nums 的元素个数
        sub_sets = []                                 # sub_sets 用于保存所有子集
        for i in range(1 << n):                     # 枚举 0 ~ 2^n - 1
            sub_set = []                             # sub_set 用于保存当前子集
            for j in range(n):                      # 枚举第 i 位元素
                if i >> j & 1:                      # 如果第 i 为元素对应二进制位为 1，则表示选取该元素
                    sub_set.append(nums[j])          # 将选取的元素加入到子集 sub_set 中
            sub_sets.append(sub_set)                 # 将子集 sub_set 加入到所有子集数组 sub_sets 中
        return sub_sets                               # 返回所有子集

```

思路 2：复杂度分析

- 时间复杂度：** $O(n \times 2^n)$ ，其中 n 指的是数组 `nums` 的元素个数， 2^n 指的是所有状态数。每种状态需要 $O(n)$ 的时间来构造子集。
- 空间复杂度：** $O(n)$ ，每种状态下构造子集需要使用 $O(n)$ 的空间。

0079. 单词搜索

- 标签：数组、回溯、矩阵
- 难度：中等

题目链接

- [0079. 单词搜索 - 力扣](#)

题目大意

描述：给定一个 $m \times n$ 大小的二维字符矩阵 `board` 和一个字符串单词 `word`。

要求：如果 `word` 存在于网格中，返回 `True`，否则返回 `False`。

说明:

- 单词必须按照字母顺序通过上下左右相邻的单元格字母构成。且同一个单元格内的字母不允许被重复使用。
- $m == board.length$ 。
- $n == board[i].length$ 。
- $1 \leq m, n \leq 6$ 。
- $1 \leq word.length \leq 15$ 。
- $board$ 和 $word$ 仅由大小写英文字母组成。

示例:

- 示例 1:

A	B	C	E
S	F	C	S
A	D	E	E

输入: `board = [["A", "B", "C", "E"], ["S", "F", "C", "S"], ["A", "D", "E", "E"]], word = "ABCED"`
输出: `true`

- 示例 2:

A	B	C	E
S	F	C	S
A	D	E	E

输入: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "SEE"

输出: true

解题思路

思路 1：回溯算法

使用回溯算法在二维矩阵 $board$ 中按照上下左右四个方向递归搜索。

设函数 $\text{backtrack}(i, j, \text{index})$ 表示从 $board[i][j]$ 出发，能否搜索到单词字母 $word[\text{index}]$ ，以及 index 位置之后的后缀子串。如果能搜索到，则返回 `True`，否则返回 `False`。

$\text{backtrack}(i, j, \text{index})$ 执行步骤如下：

1. 如果 $board[i][j] = word[\text{index}]$ ，而且 index 已经到达 $word$ 字符串末尾，则返回 `True`。
2. 如果 $board[i][j] = word[\text{index}]$ ，而且 index 未到达 $word$ 字符串末尾，则遍历当前位置的所有相邻位置。如果从某个相邻位置能搜索到后缀子串，则返回 `True`，否则返回 `False`。
3. 如果 $board[i][j] \neq word[\text{index}]$ ，则当前字符不匹配，返回 `False`。

思路 1：代码

```
class Solution:
    def exist(self, board: List[List[str]], word: str) -> bool:
        directs = [(0, 1), (0, -1), (1, 0), (-1, 0)]
        rows = len(board)
        if rows == 0:
            return False
        cols = len(board[0])
        visited = [[False for _ in range(cols)] for _ in range(rows)]

        def backtrack(i, j, index):
            if index == len(word) - 1:
                return board[i][j] == word[index]

            if board[i][j] == word[index]:
                visited[i][j] = True
                for direct in directs:
                    new_i = i + direct[0]
                    new_j = j + direct[1]
                    if 0 <= new_i < rows and 0 <= new_j < cols and visited[new_i][new_j]:
                        if backtrack(new_i, new_j, index + 1):
                            return True
                visited[i][j] = False
            return False

        for i in range(rows):
            for j in range(cols):
                if backtrack(i, j, 0):
                    return True
        return False
```

思路 1：复杂度分析

- **时间复杂度**: $O(m \times n \times 2^l)$, 其中 m 、 n 为二维矩阵 $board$ 的行数和列数。 l 为字符串 $word$ 的长度。
- **空间复杂度**: $O(m \times n)$ 。

0080. 删 除有序数组中的重复项 II

- 标签：数组、双指针
- 难度：中等

题目链接

- 0080. 删 除有序数组中的重复项 II - 力扣

题目大意

描述：给定一个有序数组 $nums$ 。

要求：在原数组空间基础上删除重复出现 2 次以上的元素，并返回删除后数组的新长度。

说明：

- $1 \leq nums.length \leq 3 * 10^4$ 。
- $-10^4 \leq nums[i] \leq 10^4$ 。
- $nums$ 已按升序排列。

示例：

- **示例 1：**

输入: `nums = [1,1,1,2,2,3]`

输出: `5, nums = [1,1,2,2,3]`

解释: 函数应返回新长度 `length = 5`，并且原数组的前五个元素被修改为 `1, 1, 2, 2, 3`。不需要考虑数组中

- **示例 2：**

输入: `nums = [0,0,1,1,1,2,3,3]`

输出: `7, nums = [0,0,1,1,2,3,3]`

解释: 函数应返回新长度 `length = 7`，并且原数组的前五个元素被修改为 `0, 0, 1, 1, 2, 3, 3`。不需要考

解题思路

思路 1：快慢指针

因为数组是有序的，所以重复元素必定是连续的。可以使用快慢指针来解决。具体做法如下：

1. 使用两个指针 $slow$, $fast$ 。 $slow$ 指针指向即将放置元素的位置, $fast$ 指针指向当前待处理元素。
2. 本题要求相同元素最多出现 2 次，并且 $slow - 2$ 是上上次放置了元素的位置。则应该检查 $nums[slow - 2]$ 和当前待处理元素 $nums[fast]$ 是否相同。
 - i. 如果 $nums[slow - 2] == nums[fast]$ 时, 此时必有 $nums[slow - 2] == nums[slow - 1] == nums[fast]$, 则当前 $nums[fast]$ 不保留, 直接向右移动快指针 $fast$ 。
 - ii. 如果 $nums[slow - 2] \neq nums[fast]$ 时, 则保留 $nums[fast]$ 。将 $nums[fast]$ 赋值给 $nums[slow]$, 同时将 $slow$ 右移。然后再向右移动快指针 $fast$ 。
3. 这样 $slow$ 指针左边均为处理好的数组元素, 而从 $slow$ 指针指向的位置开始, $fast$ 指针左边都为舍弃的重复元素。
4. 遍历结束之后, 此时 $slow$ 就是新数组的长度。

思路 1：代码

```
class Solution:  
    def removeDuplicates(self, nums: List[int]) -> int:  
        size = len(nums)  
        if size <= 2:  
            return size  
        slow, fast = 2, 2  
        while (fast < size):  
            if nums[slow - 2] != nums[fast]:  
                nums[slow] = nums[fast]  
                slow += 1  
            fast += 1  
        return slow
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。
- 空间复杂度: $O(1)$ 。# 0081. 搜索旋转排序数组 II
- 标签: 数组、二分查找

- 难度：中等

题目链接

- [0081. 搜索旋转排序数组 II - 力扣](#)

题目大意

描述：一个按照升序排列的整数数组 $nums$ ，在位置的某个下标 k 处进行了旋转操作。（例如： $[0, 1, 2, 5, 6, 8]$ 可能变为 $[5, 6, 8, 0, 1, 2]$ ）。

现在给定旋转后的数组 $nums$ 和一个整数 $target$ 。

要求：编写一个函数来判断给定的 $target$ 是否存在与数组中。如果存在则返回 `True`，否则返回 `False`。

说明：

- $1 \leq nums.length \leq 5000$ 。
- $-10^4 \leq nums[i] \leq 10^4$ 。
- 题目数据保证 $nums$ 在预先未知的某个下标上进行了旋转。
- $-10^4 \leq target \leq 10^4$ 。

示例：

- **示例 1：**

输入: `nums = [2,5,6,0,0,1,2]`, `target = 0`

输出: `true`

- **示例 2：**

输入: `nums = [2,5,6,0,0,1,2]`, `target = 3`

输出: `false`

解题思路

思路 1：二分查找

这道题算是「[0033. 搜索旋转排序数组](#)」的变形，只不过输出变为了判断。

原本为升序排列的数组 `nums` 经过「旋转」之后，会有两种情况，第一种就是原先的升序序列，另一种是两段升序的序列。



最直接的办法就是遍历一遍，找到目标值 `target`。但是还可以有更好的方法。考虑用二分查找来降低算法的时间复杂度。

我们将旋转后的数组看成左右两个升序部分：左半部分和右半部分。

有人会说第一种情况不是只有一个部分吗？其实我们可以把第一种情况中的整个数组看做是左半部分，然后右半部分为空数组。

然后创建两个指针 `left`、`right`，分别指向数组首尾。让后计算出两个指针中间值 `mid`。将 `mid` 与两个指针做比较，并考虑与 `target` 的关系。

- 如果 $nums[mid] > nums[left]$ ，则 `mid` 在左半部分（因为右半部分值都比 $nums[left]$ 小）。
 - 如果 $nums[mid] \geq target$ ，并且 $target \geq nums[left]$ ，则 `target` 在左半部分，并且在 `mid` 左侧，此时应将 `right` 左移到 $mid - 1$ 位置。
 - 否则如果 $nums[mid] < target$ ，则 `target` 在左半部分，并且在 `mid` 右侧，此时应将 `left` 右移到 $mid + 1$ 。

- 否则如果 $nums[left] > target$, 则 $target$ 在右半部分, 应将 $left$ 移动到 $mid + 1$ 位
置。
- 如果 $nums[mid] < nums[left]$, 则 mid 在右半部分 (因为右半部分值都比 $nums[left]$ 小)。
 - 如果 $nums[mid] < target$, 并且 $target \leq nums[right]$, 则 $target$ 在右半部分, 并且在 mid 右侧, 此时应将 $left$ 右移到 $mid + 1$ 位置。
 - 否则如果 $nums[mid] \geq target$, 则 $target$ 在右半部分, 并且在 mid 左侧, 此时应将 $right$ 左移到 $mid - 1$ 位置。
 - 否则如果 $nums[right] < target$, 则 $target$ 在左半部分, 应将 $right$ 左移到 $mid - 1$ 位置。
- 最终判断 $nums[left]$ 是否等于 $target$, 如果等于, 则返回 `True`, 否则返回 `False`。

思路 1：代码

```
class Solution:
    def search(self, nums: List[int], target: int) -> bool:
        n = len(nums)
        if n == 0:
            return False

        left = 0
        right = len(nums) - 1
        while left < right:
            mid = left + (right - left) // 2

            if nums[mid] > nums[left]:
                if nums[left] <= target and target <= nums[mid]:
                    right = mid
                else:
                    left = mid + 1
            elif nums[mid] < nums[left]:
                if nums[mid] < target and target <= nums[right]:
                    left = mid + 1
                else:
                    right = mid
            else:
                if nums[mid] == target:
                    return True
                else:
                    left = left + 1

        return nums[left] == target
```

思路 1：复杂度分析

- **时间复杂度**: $O(n)$, 其中 n 是数组 $nums$ 的长度。最坏情况下数组元素均相等且不为 $target$, 我们需要访问所有位置才能得出结果。
- **空间复杂度**: $O(1)$ 。

0082. 删除排序链表中的重复元素 II

- 标签: 链表、双指针

- 难度：中等

题目链接

- 0082. 删除排序链表中的重复元素 II - 力扣

题目大意

描述：给定一个已排序的链表的头 *head*。

要求：删除原始链表中所有重复数字的节点，只留下不同的数字。返回已排序的链表。

说明：

- 链表中节点数目在范围 $[0, 300]$ 内。
- $-100 \leq \text{Node.val} \leq 100$ 。
- 题目数据保证链表已经按升序排列。

示例：

- **示例 1：**

输入: `head = [1,2,3,3,4,4,5]`

输出: `[1,2,5]`

解题思路

思路 1：遍历

这道题的题意是需要保留所有不同数字，而重复出现的所有数字都要删除。因为给定的链表是升序排列的，所以我们要删除的重复元素在链表中的位置是连续的。所以我们可以对链表进行一次遍历，然后将连续的重复元素从链表中删除即可。具体步骤如下：

- 先使用哑节点 *dummy_head* 构造一个指向 *head* 的指针，使得可以防止从 *head* 开始就是重复元素。
- 然后使用指针 *cur* 表示链表中当前元素，从 *head* 开始遍历。
- 当指针 *cur* 的下一个元素和下下一个元素存在时：

- 如果下一个元素值和下下一个元素值相同，则我们使用指针 $temp$ 保存下一个元素，并使用 $temp$ 向后遍历，跳过所有重复元素，然后令 cur 的下一个元素指向 $temp$ 的下一个元素，继续向后遍历。
- 如果下一个元素值和下下一个元素值不同，则令 cur 向右移动一位，继续向后遍历。
- 当指针 cur 的下一个元素或者下下一个元素不存在时，说明已经遍历完，则返回哑节点 $dummy_head$ 的下一个节点作为头节点。

思路 1：代码

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def deleteDuplicates(self, head: ListNode) -> ListNode:
        dummy_head = ListNode(-1)
        dummy_head.next = head

        cur = dummy_head
        while cur.next and cur.next.next:
            if cur.next.val == cur.next.next.val:
                temp = cur.next
                while temp and temp.next and temp.val == temp.next.val:
                    temp = temp.next
                cur.next = temp.next
            else:
                cur = cur.next
        return dummy_head.next
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。其中 n 为链表长度。
- 空间复杂度： $O(1)$ 。[# 0083. 删除排序链表中的重复元素](#)
- 标签：链表
- 难度：简单

题目链接

- [0083. 删除排序链表中的重复元素 - 力扣](#)

题目大意

描述：给定一个已排序的链表的头 *head*。

要求：删除所有重复的元素，使每个元素只出现一次。返回已排序的链表。

说明：

- 链表中节点数目在范围 $[0, 300]$ 内。
- $-100 \leq Node.val \leq 100$ 。
- 题目数据保证链表已经按升序排列。

示例：

- **示例 1：**

输入: head = [1,1,2,3,3]

输出: [1,2,3]

解题思路

思路 1：遍历

- 使用指针 *curr* 遍历链表，先将 *head* 保存到 *curr* 指针。
- 判断当前元素的值和当前元素下一个节点元素值是否相等。
- 如果相等，则让当前指针指向当前指针下两个节点。
- 否则，让 *curr* 继续向后遍历。
- 遍历完之后返回头节点 *head*。

思路 1：遍历代码

```
class Solution:
    def deleteDuplicates(self, head: ListNode) -> ListNode:
        if head == None:
            return head

        curr = head
        while curr.next:
            if curr.val == curr.next.val:
                curr.next = curr.next.next
            else:
                curr = curr.next
        return head
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。其中 n 为链表长度。
- 空间复杂度： $O(1)$ 。[# 0084. 柱状图中最大的矩形](#)
- 标签：栈、数组、单调栈
- 难度：困难

题目链接

- [0084. 柱状图中最大的矩形 - 力扣](#)

题目大意

给定一个非负整数数组 `heights`，`heights[i]` 用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1。

要求：计算出在该柱状图中，能够勾勒出来的矩形的最大面积。

解题思路

思路一：枚举「宽度」。一重循环枚举所有柱子，第二重循环遍历柱子右侧的柱子，所得的宽度就是两根柱子形成区间的宽度，高度就是这段区间中的最小高度。然后计算出对应面积，记录并更新最大面积。这样下来，时间复杂度为 $O(n^2)$ 。

思路二：枚举「高度」。一重循环枚举所有柱子，以柱子高度为当前矩形高度，然后向两侧延伸，遇到小于当前矩形高度的情况就停止。然后计算当前矩形面积，记录并更新最大面积。这样下来，时间复杂度也是 $O(n^2)$ 。

思路三：利用「单调栈」减少两侧延伸的复杂度。

- 枚举所有柱子。
- 如果当前柱子高度较大，大于等于栈顶柱体的高度，则直接将当前柱体入栈。
- 如果当前柱体高度较小，小于栈顶柱体的高度，则一直出栈，直到当前柱体大于等于栈顶柱体高度。
 - 出栈后，说明当前柱体是出栈柱体向右找到的第一个小于当前柱体高度的柱体，那么就可以向右将宽度扩展到当前柱体。
 - 出栈后，说明新的栈顶柱体是出栈柱体向左找到的第一个小于新的栈顶柱体高度的柱体，那么就可以向左将宽度扩展到新的栈顶柱体。
 - 以新的栈顶柱体为左边界，当前柱体为右边界，以出栈柱体为高度。计算矩形面积，然后记录并更新最大面积。

代码

```
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        heights.append(0)
        ans = 0
        stack = []
        for i in range(len(heights)):
            while stack and heights[stack[-1]] >= heights[i]:
                cur = stack.pop(-1)
                left = stack[-1] + 1 if stack else 0
                right = i - 1
                ans = max(ans, (right - left + 1) * heights[cur])
            stack.append(i)

        return ans
```

0088. 合并两个有序数组

- 标签：数组、双指针、排序
- 难度：简单

题目链接

- 0088. 合并两个有序数组 - 力扣

题目大意

描述：给定两个有序数组 $nums1$ 、 $nums2$ 。

要求：将 $nums2$ 合并到 $nums1$ 中，使 $nums1$ 成为一个有序数组。

说明：

- 给定数组 $nums1$ 空间大小为 $m + n$ 个，其中前 m 个为 $nums1$ 的元素。 $nums2$ 空间大小为 n 。这样可以用 $nums1$ 的空间来存储最终的有序数组。
- $nums1.length == m + n$ 。
- $nums2.length == n$ 。
- $0 \leq m, n \leq 200$ 。
- $1 \leq m + n \leq 200$ 。
- $-10^9 \leq nums1[i], nums2[j] \leq 10^9$ 。

示例：

- **示例 1：**

输入: $nums1 = [1, 2, 3, \mathbf{0}, 0, 0]$, $m = 3$, $nums2 = [2, 5, 6]$, $n = 3$

输出: $[1, 2, \mathbf{2}, 3, 5, 6]$

解释: 需要合并 $[1, 2, 3]$ 和 $[2, 5, 6]$ 。

合并结果是 $[1, 2, \mathbf{2}, 3, 5, 6]$ ，其中斜体加粗标注的为 $nums1$ 中的元素。

- **示例 2：**

输入: $nums1 = [1]$, $m = 1$, $nums2 = []$, $n = 0$

输出: $[1]$

解释: 需要合并 $[1]$ 和 $[]$ 。

合并结果是 $[1]$ 。

解题思路

思路 1：快慢指针

1. 将两个指针 $index1$ 、 $index2$ 分别指向 $nums1$ 、 $nums2$ 数组的尾部，再用一个指针 $index$ 指向数组 $nums1$ 的尾部。
2. 从后向前判断当前指针下 $nums1[index1]$ 和 $nums2[index2]$ 的值大小，将较大值存入 $num1[index]$ 中，然后继续向前遍历。
3. 最后再将 $nums2$ 中剩余元素赋值到 $num1$ 前面对应位置上。

思路 1：代码

```
class Solution:  
    def merge(self, nums1: List[int], m: int, nums2: List[int], n: int) -> None:  
        index1 = m - 1  
        index2 = n - 1  
        index = m + n - 1  
        while index1 >= 0 and index2 >= 0:  
            if nums1[index1] < nums2[index2]:  
                nums1[index] = nums2[index2]  
                index2 -= 1  
            else:  
                nums1[index] = nums1[index1]  
                index1 -= 1  
            index -= 1  
  
        nums1[:index2+1] = nums2[:index2+1]
```

思路 1：复杂度分析

- 时间复杂度： $O(m + n)$ 。
- 空间复杂度： $O(m + n)$ 。

0089. 格雷编码

- 标签：位运算、数学、回溯
- 难度：中等

题目链接

- 0089. 格雷编码 - 力扣

题目大意

描述：给定一个整数 n 。

要求：返回任一有效的 n 位格雷码序列。

说明：

- **n 位格雷码序列：**是一个由 2^n 个整数组成的序列，其中：
 - 每个整数都在范围 $[0, 2^n - 1]$ 内（含 0 和 $2^n - 1$ ）。
 - 第一个整数是 0。
 - 一个整数在序列中出现不超过一次。
 - 每对相邻整数的二进制表示恰好一位不同，且第一个和最后一个整数的二进制表示恰好一位不同。
- $1 \leq n \leq 16$ 。

示例：

- **示例 1：**

输入： $n = 2$

输出： $[0, 1, 3, 2]$

解释：

$[0, 1, 3, 2]$ 的二进制表示是 $[00, 01, 11, 10]$ 。

- 00 和 01 有一位不同
- 01 和 11 有一位不同
- 11 和 10 有一位不同
- 10 和 00 有一位不同

$[0, 2, 3, 1]$ 也是一个有效的格雷码序列，其二进制表示是 $[00, 10, 11, 01]$ 。

- 00 和 10 有一位不同
- 10 和 11 有一位不同
- 11 和 01 有一位不同
- 01 和 00 有一位不同

- **示例 2：**

输入: `n = 1`

输出: `[0, 1]`

解题思路

思路 1：位运算 + 公式法

- 格雷编码生成规则：以二进制值为 0 的格雷编码作为第 0 项，第一次改变最右边的数位，第二次改变从右边数第一个为 1 的数位左边的数位，第三次跟第一次一样，改变最右边的数位，第四次跟第二次一样，改变从右边数第一个为 1 的数位左边的数位。此后，第五、六次，第七、八次 ... 都跟第一、二次一样反复进行，直到生成 2^n 个格雷编码。
- 也可以直接利用二进制转换为格雷编码公式：

某二进制数为 $B_{n-1}B_{n-2}\cdots B_2B_1B_0$

其对应的格雷码为 $G_{n-1}G_{n-2}\cdots G_2G_1G_0$

其中：最高位保留—— $G_{n-1} = B_{n-1}$

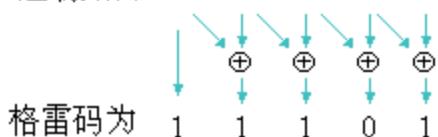
异或运算：

相同为0

相异为1

其他各位—— $G_i = B_{i+1} \oplus B_i \quad i=0, 1, 2, \dots, n-2$

例：二进制数为 1 0 1 1 0



思路 1：代码

```
class Solution:  
    def grayCode(self, n: int) -> List[int]:  
        gray = []  
        binary = 0  
        while binary < (1 << n):  
            gray.append(binary ^ binary >> 1)  
            binary += 1  
        return gray
```

思路 1：复杂度分析

- 时间复杂度： $O(2^n)$ 。
- 空间复杂度： $O(1)$ 。

0090. 子集 II

- 标签：位运算、数组、回溯
- 难度：中等

题目链接

- [0090. 子集 II - 力扣](#)

题目大意

描述：给定一个整数数组 `nums`，其中可能包含重复元素。

要求：返回该数组所有可能的子集（幂集）。

说明：

- 解集不能包含重复的子集。返回的解集中，子集可以按任意顺序排列。
- $1 \leq \text{nums.length} \leq 10$ 。
- $-10 \leq \text{nums}[i] \leq 10$ 。

示例：

- **示例 1：**

输入：`nums = [1,2,2]`

输出：`[[], [1], [1, 2], [1, 2, 2], [2], [2, 2]]`

解题思路

思路 1：回溯算法

数组的每个元素都有两个选择：选与不选。

我们可以通过向当前子集数组中添加可选元素来表示选择该元素。也可以在当前递归结束之后，将之前添加的元素从当前子集数组中移除（也就是回溯）来表示不选择该元素。

因为数组中可能包含重复元素，所以我们可以先将数组排序，然后在回溯时，判断当前元素是否和上一个元素相同，如果相同，则直接跳过，从而去除重复元素。

回溯算法解决这道题的步骤如下：

- 先对数组 `nums` 进行排序。
- 从第 `0` 个位置开始，调用 `backtrack` 方法进行深度优先搜索。
- 将当前子集数组 `sub_set` 添加到答案数组 `sub_sets` 中。
- 然后从当前位置开始，到数组结束为止，枚举出所有可选的元素。对于每一个可选元素：
 - 如果当前元素与上一个元素相同，则跳过当前生成的子集。
 - 将可选元素添加到当前子集数组 `sub_set` 中。
 - 在选择该元素的情况下，继续递归考虑下一个元素。
 - 进行回溯，撤销选择该元素。即从当前子集数组 `sub_set` 中移除之前添加的元素。

思路 1：代码

```
class Solution:
    def backtrack(self, nums, index, res, path):
        res.append(path[:])

        for i in range(index, len(nums)):
            if i > index and nums[i] == nums[i - 1]:
                continue
            path.append(nums[i])
            self.backtrack(nums, i + 1, res, path)
            path.pop()

    def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
        nums.sort()
        res, path = [], []
        self.backtrack(nums, 0, res, path)
        return res
```

思路 1：复杂度分析

- **时间复杂度**: $O(n \times 2^n)$, 其中 n 指的是数组 nums 的元素个数, 2^n 指的是所有状态数。每种状态需要 $O(n)$ 的时间来构造子集。
- **空间复杂度**: $O(n)$, 每种状态下构造子集需要使用 $O(n)$ 的空间。

思路 2：二进制枚举

对于一个元素个数为 n 的集合 nums 来说，每一个位置上的元素都有选取和未选取两种状态。我们可以用数字 1 来表示选取该元素，用数字 0 来表示不选取该元素。

那么我们就可以用一个长度为 n 的二进制数来表示集合 nums 或者表示 nums 的子集。其中二进制的每一位数都对应了集合中某一个元素的选取状态。对于集合中第 i 个元素（ i 从 0 开始编号）来说，二进制对应位置上的 1 代表该元素被选取， 0 代表该元素未被选取。

举个例子来说明一下，比如长度为 5 的集合 $\text{nums} = \{5, 4, 3, 2, 1\}$ ，我们可以用一个长度为 5 的二进制数来表示该集合。

比如二进制数 11111 就表示选取集合的第 0 位、第 1 位、第 2 位、第 3 位、第 4 位元素，也就是集合 $\{5, 4, 3, 2, 1\}$ ，即集合 nums 本身。如下表所示：

集合 <code>nums</code> 对应位置 (下标)	4	3	2	1	0
二进制数对应位数	1	1	1	1	1
对应选取状态	选取	选取	选取	选取	选取

再比如二进制数 `10101` 就表示选取集合的第 0 位、第 2 位、第 5 位元素，也就是集合 `{5, 3, 1}`。如下表所示：

集合 <code>nums</code> 对应位置 (下标)	4	3	2	1	0
二进制数对应位数	1	0	1	0	1
对应选取状态	选取	未选取	选取	未选取	选取

再比如二进制数 `01001` 就表示选取集合的第 0 位、第 3 位元素，也就是集合 `{5, 2}`。如下表所示：

集合 <code>nums</code> 对应位置 (下标)	4	3	2	1	0
二进制数对应位数	0	1	0	0	1
对应选取状态	未选取	选取	未选取	未选取	选取

通过上面的例子我们可以得到启发：对于长度为 5 的集合 `nums` 来说，我们只需要从 `00000 ~ 11111` 枚举一次（对应十进制为 $0 \sim 2^4 - 1$ ）即可得到长度为 5 的集合 `s` 的所有子集。

我们将上面的例子拓展到长度为 `n` 的集合 `nums`。可以总结为：

- 对于长度为 5 的集合 `nums` 来说，只需要枚举 $0 \sim 2^n - 1$ (共 2^n 种情况)，即可得到所有的子集。

因为数组中可能包含重复元素，所以我们可以先对数组进行排序。然后在枚举过程中，如果发现当前元素和上一个元素相同，则直接跳过当前生层的子集，从而去除重复元素。

思路 2：代码

```
class Solution:
    def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
        nums.sort()
        n = len(nums)                                # n 为集合 nums 的元素个数
        sub_sets = []                                 # sub_sets 用于保存所有子集
        for i in range(1 << n):                      # 枚举 0 ~ 2^n - 1
            sub_set = []                             # sub_set 用于保存当前子集
            flag = True                            # flag 用于判断重复元素
            for j in range(n): # 枚举第 i 位元素
                if i >> j & 1: # 如果第 i 为元素对应二进制位为 1, 则表示选取该元素
                    if j > 0 and (i >> (j - 1) & 1) == 0 and nums[j] == nums[j - 1]:
                        flag = False           # 如果出现重复元素, 则跳过当前生成的子集
                        break
                    sub_set.append(nums[j]) # 将选取的元素加入到子集 sub_set 中
            if flag:
                sub_sets.append(sub_set) # 将子集 sub_set 加入到所有子集数组 sub_sets 中
        return sub_sets                                # 返回所有子集
```

思路 2：复杂度分析

- **时间复杂度**: $O(n \times 2^n)$, 其中 n 指的是数组 `nums` 的元素个数, 2^n 指的是所有状态数。每种状态需要 $O(n)$ 的时间来构造子集。
- **空间复杂度**: $O(n)$, 每种状态下构造子集需要使用 $O(n)$ 的空间。

0091. 解码方法

- 标签: 字符串、动态规划
- 难度: 中等

题目链接

- [0091. 解码方法 - 力扣](#)

题目大意

描述: 给定一个数字字符串 s 。该字符串已经按照下面的映射关系进行了编码：

- A 映射为 1。
- B 映射为 2。
- ...
- Z 映射为 26。

基于上述映射的方法，现在对字符串 s 进行「解码」。即从数字到字母进行反向映射。比如 "11106" 可以映射为：

- "AAJF"，将消息分组为 (11106)。
- "KJF"，将消息分组为 (11106)。

要求：计算出共有多少种可能的解码方案。

说明：

- $1 \leq s.length \leq 100$ 。
- s 只包含数字，并且可能包含前导零。
- 题目数据保证答案肯定是一个 32 位的整数。

示例：

- 示例 1：

输入： $s = "226"$

输出： 3

解释：它可以解码为 "BZ" (2 26)，"VF" (22 6)，或者 "BBF" (2 2 6)。

解题思路

思路 1：动态规划

1. 划分阶段

按照字符串的结尾位置进行阶段划分。

2. 定义状态

定义状态 $dp[i]$ 表示为：字符串 s 前 i 个字符构成的字符串可能构成的翻译方案数。

3. 状态转移方程

$dp[i]$ 的来源有两种情况：

- 使用了一个字符，对 $s[i]$ 进行翻译。只要 $s[i] \neq 0$ ，就可以被翻译为 A ~ I 的某个字母，此时方案数为 $dp[i] = dp[i - 1]$ 。
- 使用了两个字符，对 $s[i - 1]$ 和 $s[i]$ 进行翻译，只有 $s[i - 1] \neq 0$ ，且 $s[i - 1]$ 和 $s[i]$ 组成的整数必须小于等于 26 才能翻译，可以翻译为 J ~ Z 中的某字母，此时方案数为 $dp[i] = dp[i - 2]$ 。

这两种情况有可能是同时存在的，也有可能都不存在。在进行转移的时候，将符合要求的方案数累加起来即可。

状态转移方程可以写为：

$$dp[i] = \begin{cases} dp[i - 1] & s[i] \neq 0 \\ dp[i - 2] & s[i - 1] \neq 0, s[i - 1 : i] \leq 26 \end{cases}$$

4. 初始条件

- 字符串为空时，只有一个翻译方案，翻译为空字符串，即 $dp[0] = 1$ 。
- 字符串只有一个字符时，需要考虑该字符是否为 0，不为 0 的话， $dp[1] = 1$ ，为 0 的话， $dp[0] = 0$ 。

5. 最终结果

根据我们之前定义的状态， $dp[i]$ 表示为：字符串 s 前 i 个字符构成的字符串可能构成的翻译方案数。则最终结果为 $dp[size]$ ， $size$ 为字符串长度。

思路 1：动态规划代码

```
class Solution:
    def numDecodings(self, s: str) -> int:
        size = len(s)
        dp = [0 for _ in range(size + 1)]
        dp[0] = 1
        for i in range(1, size + 1):
            if s[i - 1] != '0':
                dp[i] += dp[i - 1]
            if i > 1 and s[i - 2] != '0' and int(s[i - 2: i]) <= 26:
                dp[i] += dp[i - 2]
        return dp[size]
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。一重循环遍历的时间复杂度是 $O(n)$ 。

- 空间复杂度: $O(n)$ 。用到了一维数组保存状态，所以总体空间复杂度为 $O(n)$ 。

0092. 反转链表 II

- 标签：链表
- 难度：中等

题目链接

- [0092. 反转链表 II - 力扣](#)

题目大意

描述：给定单链表的头指针 `head` 和两个整数 `left` 和 `right`，其中 `left <= right`。

要求：反转从位置 `left` 到位置 `right` 的链表节点，返回反转后的链表。

说明：

- 链表中节点数目为 n 。
- $1 \leq n \leq 500$ 。
- $-500 \leq Node.val \leq 500$ 。
- $1 \leq left \leq right \leq n$ 。

示例：

- **示例 1：**

输入: `head = [1,2,3,4,5]`, `left = 2`, `right = 4`
输出: `[1,4,3,2,5]`

解题思路

在「[0206. 反转链表](#)」中我们可以通过迭代、递归两种方法将整个链表反转。这道题而这道题要求对链表的部分区间进行反转。我们同样可以通过迭代、递归两种方法将链表的部分区间进行反转。

思路 1：迭代

我们可以先遍历到需要反转的链表区间的前一个节点，然后对需要反转的链表区间进行迭代反转。最后再返回头节点即可。

但是需要注意一点，如果需要反转的区间包含了链表的第一个节点，那么我们可以事先创建一个哑节点作为链表初始位置开始遍历，这样就能避免找不到需要反转的链表区间的前一个节点。

这道题的具体解题步骤如下：

1. 先使用哑节点 `dummy_head` 构造一个指向 `head` 的指针，使得可以从 `head` 开始遍历。使用 `index` 记录当前元素的序号。
2. 我们使用一个指针 `reverse_start`，初始赋值为 `dummy_head`。然后向右逐步移动到需要反转的区间的前一个节点。
3. 然后再使用两个指针 `cur` 和 `pre` 进行迭代。`pre` 指向 `cur` 前一个节点位置，即 `pre` 指向需要反转节点的前一个节点，`cur` 指向需要反转的节点。初始时，`pre` 指向 `reverse_start`，`cur` 指向 `pre.next`。
4. 当当前节点 `cur` 不为空，且 `index` 在反转区间内时，将 `pre` 和 `cur` 的前后指针进行交换，指针更替顺序为：
 - i. 使用 `next` 指针保存当前节点 `cur` 的后一个节点，即 `next = cur.next`；
 - ii. 断开当前节点 `cur` 的后一节点链接，将 `cur` 的 `next` 指针指向前一节点 `pre`，即 `cur.next = pre`；
 - iii. `pre` 向前移动一步，移动到 `cur` 位置，即 `pre = cur`；
 - iv. `cur` 向前移动一步，移动到之前 `next` 指针保存的位置，即 `cur = next`。
 - v. 然后令 `index` 加 1。
5. 继续执行第 4 步中的 1、2、3、4、5 步。
6. 最后等到 `cur` 遍历到链表末尾（即 `cur == None`）或者遍历到需要反转区间的末尾时（即 `index > right`）时，将反转区间的头尾节点分别与之前保存的需要反转的区间的前一个节点 `reverse_start` 相连，即 `reverse_start.next.next = cur`，`reverse_start.next = pre`。
7. 最后返回新的头节点 `dummy_head.next`。

思路 1：代码

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def reverseBetween(self, head: ListNode, left: int, right: int) -> ListNode:
        index = 1
        dummy_head = ListNode(0)
        dummy_head.next = head
        pre = dummy_head

        reverse_start = dummy_head
        while reverse_start.next and index < left:
            reverse_start = reverse_start.next
            index += 1

        pre = reverse_start
        cur = pre.next
        while cur and index <= right:
            next = cur.next
            cur.next = pre
            pre = cur
            cur = next
            index += 1

        reverse_start.next.next = cur
        reverse_start.next = pre

        return dummy_head.next
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。其中 n 是链表节点个数。
- 空间复杂度： $O(1)$ 。

思路 2：递归算法

1. 翻转链表前 n 个节点

1. 当 `left == 1` 时，无论 `right` 等于多少，实际上都是将当前链表到 `right` 部分进行翻转，也就是将前 `right` 个节点进行翻转。
2. 我们可以先定义一个递归函数 `reverseN(self, head, n)`，含义为：将链表前第 n 个节点位置进行翻转。
 - i. 然后从 `head.next` 的位置开始调用递归函数，即将 `head.next` 为头节点的链表的前 $n - 1$ 个位置进行反转，并返回该链表的新头节点 `new_head`。
 - ii. 然后改变 `head`（原先头节点）和 `new_head`（新头节点）之间的指向关系，即将 `head` 指向的节点作为 `head` 下一个节点的下一个节点。
 - iii. 先保存 `head.next` 的 `next` 指针，也就是新链表前 n 个节点的尾指针，即 `last = head.next.next`。
 - iv. 将 `head.next` 的 `next` 指针先指向当前节点 `head`，即 `head.next.next = head`。
 - v. 然后让当前节点 `head` 的 `next` 指针指向 `last`，则完成了前 $n - 1$ 个位置的翻转。
3. 递归终止条件：当 $n == 1$ 时，相当于翻转第一个节点，直接返回 `head` 即可。

4. 翻转链表 `[left, right]` 上的节点。

接下来我们来翻转区间上的节点。

1. 定义递归函数 `reverseBetween(self, head, left, right)` 为
- 2.

思路 2：代码

```
class Solution:
    def reverseBetween(self, head: Optional[ListNode], left: int, right: int) -> Optional[ListNode]:
        if left == 1:
            return self.reverseN(head, right)

        head.next = self.reverseBetween(head.next, left - 1, right - 1)
        return head

    def reverseN(self, head, n):
        if n == 1:
            return head
        last = self.reverseN(head.next, n - 1)
        next = head.next.next
        head.next.next = head
        head.next = next
        return last
```

思路 2：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(n)$ 。最多需要 n 层栈空间。

参考资料

- 【题解】[动画图解：翻转链表的指定区间 - 反转链表 II - 力扣](#)
- 【题解】[【宫水三叶】一个能应用所有「链表」题里的「哨兵」技巧 - 反转链表 II - 力扣](#)

0093. 复原 IP 地址

- 标签：字符串、回溯
- 难度：中等

题目链接

- [0093. 复原 IP 地址 - 力扣](#)

题目大意

描述：给定一个只包含数字的字符串 s ，用来表示一个 IP 地址

要求：返回所有由 s 构成的有效 IP 地址，这些地址可以通过在 s 中插入 ‘.’ 来形成。不能重新排序或删除 s 中的任何数字。可以按任何顺序返回答案。

说明：

- **有效 IP 地址：**正好由四个整数（每个整数由 $0 \sim 255$ 的数构成，且不能含有前导 0），整数之间用 ‘.’ 分割。
- $1 \leq s.length \leq 20$ 。
- s 仅由数字组成。

示例：

- **示例 1：**

输入： $s = "25525511135"$
输出：["255.255.11.135", "255.255.111.35"]

- **示例 2：**

输入： $s = "0000"$
输出：["0.0.0.0"]

解题思路

思路 1：回溯算法

一个有效 IP 地址由四个整数构成，中间用 3 个点隔开。现在给定的是无分隔的整数字符串，我们可以通过在整数字符串中间的不同位置插入 3 个点来生成不同的 IP 地址。这个过程可以通过回溯算法来生成。

根据回溯算法三步走，写出对应的回溯算法。

1. **明确所有选择：**全排列中每个位置上的元素都可以从剩余可选元素中选出，对此画出决策树，如下图所示。
2. **明确终止条件：**

- 当遍历到决策树的叶子节点时，就终止了。即当前路径搜索到末尾时，递归终止。

3. 将决策树和终止条件翻译成代码：

i. 定义回溯函数：

- `backtracking(index)`：函数的传入参数是 `index`（剩余字符开始位置），全局变量是 `res`（存放所有符合条件结果的集合数组）和 `path`（存放当前符合条件的结果）。
- `backtracking(index)`：函数代表的含义是：递归从 `index` 位置开始，从剩下字符中，选择当前子段的值。

ii. 书写回溯函数主体（给出选择元素、递归搜索、撤销选择部分）。

- 从当前正在考虑的字符，到字符串结束为止，枚举出所有可作为当前子段值的字符。对于每一个子段值：
 - 约束条件：只能从 `index` 位置开始选择，并且要符合规则要求。
 - 选择元素：将其添加到当前子集数组 `path` 中。
 - 递归搜索：在选择该子段值的情况下，继续递归从剩下字符中，选择下一个子段值。
 - 撤销选择：将该子段值从当前结果数组 `path` 中移除。

```

for i in range(index, len(s)):      # 枚举可选元素列表
    sub = s[index: i + 1]
    # 如果当前值不在 0 ~ 255 之间，直接跳过
    if int(sub) > 255:
        continue
    # 如果当前值为 0，但不是单个 0 ("00...")，直接跳过
    if int(sub) == 0 and i != index:
        continue
    # 如果当前值大于 0，但是以 0 开头 ("0XX...")，直接跳过
    if int(sub) > 0 and s[index] == '0':
        continue

    path.append(sub)                  # 选择元素
    backtracking(i + 1)              # 递归搜索
    path.pop()                      # 撤销选择
  
```

iii. 明确递归终止条件（给出递归终止条件，以及递归终止时的处理方法）。

- 当遍历到决策树的叶子节点时，就终止了。也就是存放当前结果的数组 `path` 的长度等于 4，并且剩余字符开始位置为字符串结束位置（即 `len(path) == 4 and index == len(s)`）时，递归停止。
- 如果回溯过程中，切割次数大于 4（即 `len(path) > 4`），递归停止，直接返回。

思路 1：代码

```
class Solution:
    def restoreIpAddresses(self, s: str) -> List[str]:
        res = []
        path = []
        def backtracking(index):
            # 如果切割次数大于 4, 直接返回
            if len(path) > 4:
                return

            # 切割完成, 将当前结果加入答案结果数组中
            if len(path) == 4 and index == len(s):
                res.append('.'.join(path))
                return

            for i in range(index, len(s)):
                sub = s[index: i + 1]
                # 如果当前值不在 0 ~ 255 之间, 直接跳过
                if int(sub) > 255:
                    continue
                # 如果当前值为 0, 但不是单个 0 ("00..."), 直接跳过
                if int(sub) == 0 and i != index:
                    continue
                # 如果当前值大于 0, 但是以 0 开头 ("0XX..."), 直接跳过
                if int(sub) > 0 and s[index] == '0':
                    continue

                path.append(sub)
                backtracking(i + 1)
                path.pop()

        backtracking(0)
        return res
```

思路 1：复杂度分析

- **时间复杂度**: $O(3^4 \times |s|)$, 其中 $|s|$ 是字符串 s 的长度。由于 IP 地址的每一子段位数不会超过 3, 因此在递归时, 我们最多只会深入到下一层中的 3 种情况。而 IP 地址由 4 个子段构成, 所以递归的最大层数为 4 层, 则递归的时间复杂度为 $O(3^4)$ 。而每次将有效的 IP 地址添加到答案数组的时间复杂度为 $|s|$, 所以总的时间复杂度为 $3^4 \times |s|$ 。

- 空间复杂度: $O(|s|)$, 只记录除了用来存储答案数组之外的空间复杂度。

0094. 二叉树的中序遍历

- 标签: 栈、树、深度优先搜索、二叉树
- 难度: 简单

题目链接

- [0094. 二叉树的中序遍历 - 力扣](#)

题目大意

描述: 给定一个二叉树的根节点 `root`。

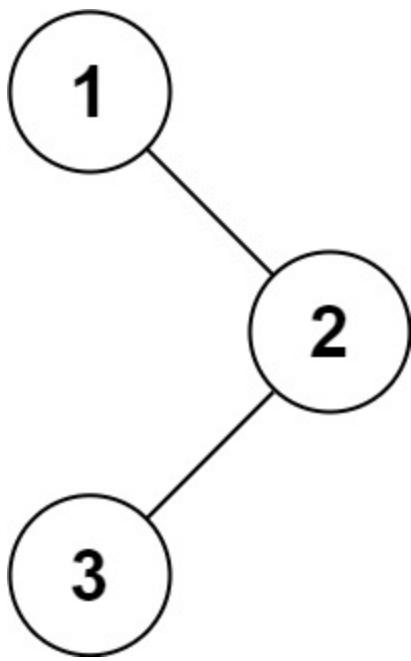
要求: 返回该二叉树的中序遍历结果。

说明:

- 树中节点数目在范围 $[0, 100]$ 内。
- $-100 \leq Node.val \leq 100$ 。

示例:

- **示例 1:**



输入: `root = [1,null,2,3]`

输出: `[1,3,2]`

- 示例 2:

输入: `root = []`

输出: `[]`

解题思路

思路 1：递归遍历

二叉树的前序遍历递归实现步骤为：

1. 判断二叉树是否为空，为空则直接返回。
2. 先访问根节点。
3. 然后递归遍历左子树。
4. 最后递归遍历右子树。

思路 1：代码

```
class Solution:
    def inorderTraversal(self, root: TreeNode) -> List[int]:
        res = []
        def inorder(root):
            if not root:
                return
            inorder(root.left)
            res.append(root.val)
            inorder(root.right)

        inorder(root)
        return res
```

思路 1：复杂度分析

- **时间复杂度**: $O(n)$ 。其中 n 是二叉树的节点数目。
- **空间复杂度**: $O(n)$ 。

思路 2：模拟栈迭代遍历

二叉树的前序遍历递归实现的过程，实际上就是调用系统栈的过程。我们也可以使用一个显式栈 `stack` 来模拟递归的过程。

前序遍历的顺序为：根节点 - 左子树 - 右子树，而根据栈的「先入后出」特点，所以入栈的顺序应该为：先放入右子树，再放入左子树。这样可以保证最终遍历顺序为前序遍历顺序。

二叉树的前序遍历显式栈实现步骤如下：

1. 判断二叉树是否为空，为空则直接返回。
2. 初始化维护一个栈，将根节点入栈。
3. 当栈不为空时：
 - i. 弹出栈顶元素 `node`，并访问该元素。
 - ii. 如果 `node` 的右子树不为空，则将 `node` 的右子树入栈。
 - iii. 如果 `node` 的左子树不为空，则将 `node` 的左子树入栈。

思路 2：代码

```
class Solution:
    def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if not root:                      # 二叉树为空直接返回
            return []

        res = []
        stack = []

        while root or stack:             # 根节点或栈不为空
            while root:
                stack.append(root)       # 将当前树的根节点入栈
                root = root.left         # 找到最左侧节点

            node = stack.pop()          # 遍历到最左侧，当前节点无左子树时，将最左侧节点弹出
            res.append(node.val)        # 访问该节点
            root = node.right          # 尝试访问该节点的右子树

        return res
```

思路 2：复杂度分析

- **时间复杂度**: $O(n)$ 。其中 n 是二叉树的节点数目。
- **空间复杂度**: $O(n)$ 。[# 0095. 不同的二叉搜索树 II](#)
- 标签: 树、二叉搜索树、动态规划、回溯、二叉树
- 难度: 中等

题目链接

- [0095. 不同的二叉搜索树 II - 力扣](#)

题目大意

描述: 给定一个整数 n 。

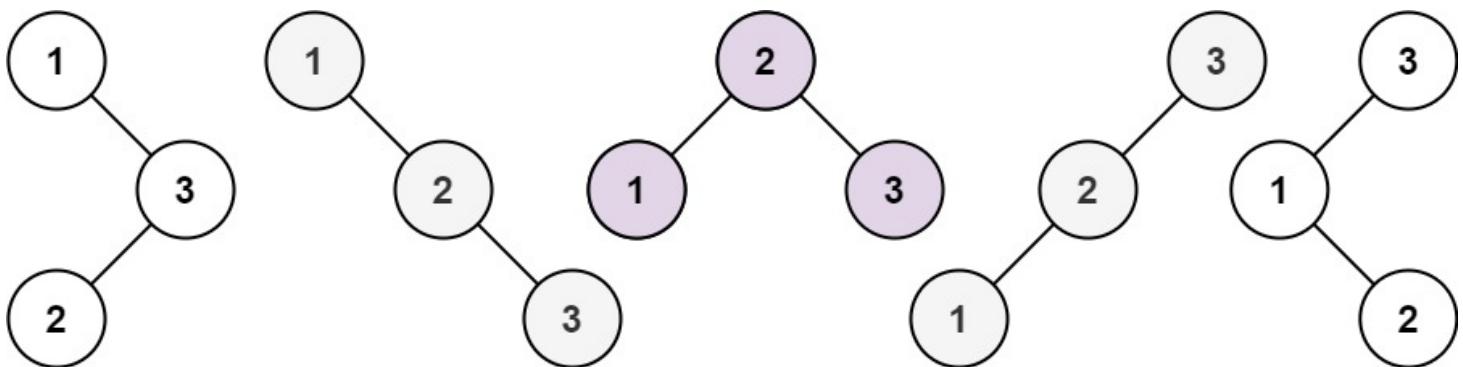
要求: 请生成返回以 1 到 n 为节点构成的「二叉搜索树」，可以按任意顺序返回答案。

说明:

- $1 \leq n \leq 8$ 。

示例：

- 示例 1：



输入： `n = 3`

输出： `[[1,null,2,null,3],[1,null,3,2],[2,1,3],[3,1,null,null,2],[3,2,null,1]]`

- 示例 2：

输入： `n = 1`

输出： `[[1]]`

解题思路

思路 1： 递归遍历

如果根节点为 i ，则左子树的节点为 $(1, 2, \dots, i - 1)$ ，右子树的节点为 $(i + 1, i + 2, \dots, n)$ 。可以递归的构建二叉树。

定义递归函数 `generateTrees(start, end)`，表示生成 $[left, \dots, right]$ 构成的所有可能的二叉搜索树。

- 如果 $start > end$ ，返回 `[None]`。
- 初始化存放所有可能二叉搜索树的数组。
- 遍历 $[left, \dots, right]$ 的每一个节点 i ，将其作为根节点。
 - 递归构建左右子树。
 - 将所有符合要求的左右子树组合起来，将其加入到存放二叉搜索树的数组中。
- 返回存放二叉搜索树的数组。

思路 1：代码

```
class Solution:
    def generateTrees(self, n: int) -> List[TreeNode]:
        if n == 0:
            return []

        def generateTrees(start, end):
            if start > end:
                return [None]
            trees = []
            for i in range(start, end+1):
                left_trees = generateTrees(start, i - 1)
                right_trees = generateTrees(i + 1, end)
                for left_tree in left_trees:
                    for right_tree in right_trees:
                        curr_tree = TreeNode(i)
                        curr_tree.left = left_tree
                        curr_tree.right = right_tree
                        trees.append(curr_tree)
            return trees
        return generateTrees(1, n)
```

思路 1：复杂度分析

- 时间复杂度: $O(C_n)$, 其中 C_n 是第 n 个卡特兰数。
- 空间复杂度: $O(C_n)$, 其中 C_n 是第 n 个卡特兰数。

0096. 不同的二叉搜索树

- 标签：树、二叉搜索树、数学、动态规划、二叉树
- 难度：中等

题目链接

- [0096. 不同的二叉搜索树 - 力扣](#)

题目大意

描述：给定一个整数 n 。

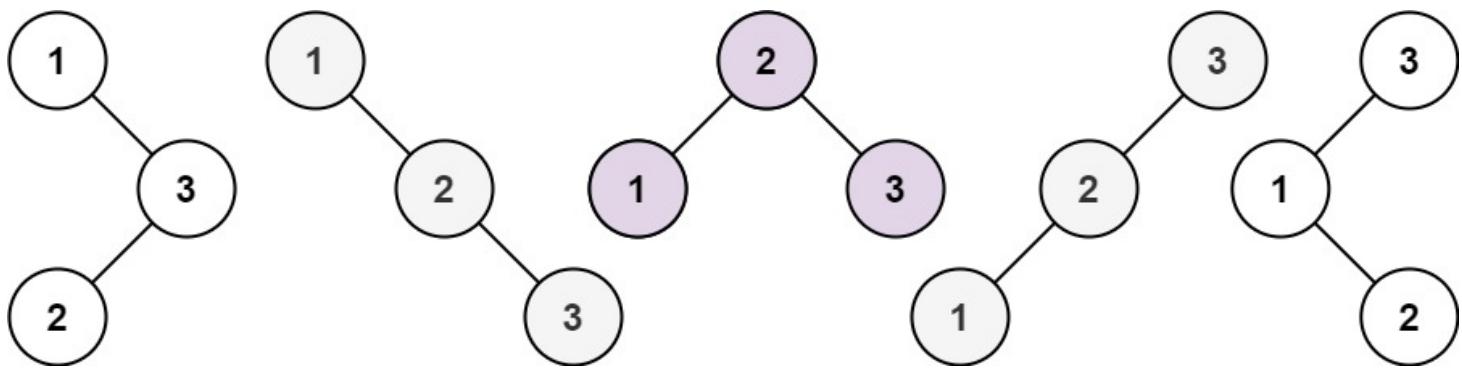
要求：求以 1 到 n 为节点构成的「二叉搜索树」有多少种？

说明：

- $1 \leq n \leq 19$ 。

示例：

- 示例 1：



输入：n = 3

输出：5

- 示例 2：

输入：n = 1

输出：1

解题思路

思路 1：动态规划

一棵搜索二叉树的左、右子树，要么也是搜索二叉树，要么就是空树。

如果定义 $f[i]$ 表示以 i 为根的二叉搜索树个数，定义 $g(i)$ 表示 i 个节点可以构成的二叉搜索树个数，则有：

- $g(i) = f(1) + f(2) + f(3) + \dots + f(i)$ 。

其中当 i 为根节点时，则用 $(1, 2, \dots, i - 1)$ 共 $i - 1$ 个节点去递归构建左子搜索二叉树，用 $(i + 1, i + 2, \dots, n)$ 共 $n - i$ 个节点去递归构建右子搜索树。则有：

- $f(i) = g(i - 1) \times g(n - i)$ 。

综合上面两个式子 $\begin{cases} g(i) = f(1) + f(2) + f(3) + \dots + f(i) \\ f(i) = g(i - 1) \times g(n - i) \end{cases}$ 可得出：

- $g(n) = g(0) \times g(n - 1) + g(1) \times g(n - 2) + \dots + g(n - 1) \times g(0)$ 。

将 n 换为 i ，可变为：

- $g(i) = g(0) \times g(i - 1) + g(1) \times g(i - 2) + \dots + g(i - 1) \times g(0)$ 。

再转换一下，可变为：

- $g(i) = \sum_{1 \leq j \leq i} \{g(j - 1) \times g(i - j)\}$ 。

则我们可以通过动态规划的方法，递推求解 $g(i)$ ，并求解出 $g(n)$ 。具体步骤如下：

1. 划分阶段

按照根节点的编号进行阶段划分。

2. 定义状态

定义状态 $dp[i]$ 表示为： i 个节点可以构成的二叉搜索树个数。

3. 状态转移方程

$$dp[i] = \sum_{1 \leq j \leq i} \{dp[j - 1] \times dp[i - j]\}$$

4. 初始条件

- 0 个节点可以构成的二叉搜索树个数为 1（空树），即 $dp[0] = 1$ 。

5. 最终结果

根据我们之前定义的状态， $dp[i]$ 表示为： i 个节点可以构成的二叉搜索树个数。。所以最终结果为 $dp[n]$ 。

思路 1：代码

```
class Solution:
    def numTrees(self, n: int) -> int:
        dp = [0 for _ in range(n + 1)]
        dp[0] = 1
        for i in range(1, n + 1):
            for j in range(1, i + 1):
                dp[i] += dp[j - 1] * dp[i - j]
        return dp[n]
```

思路 1：复杂度分析

- 时间复杂度： $O(n^2)$ 。
- 空间复杂度： $O(n)$ 。

参考资料

- [【题解】画解算法：96. 不同的二叉搜索树 - 不同的二叉搜索树](#)

0098. 验证二叉搜索树

- 标签：树、深度优先搜索、二叉搜索树、二叉树
- 难度：中等

题目链接

- [0098. 验证二叉搜索树 - 力扣](#)

题目大意

描述：给定一个二叉树的根节点 `root`。

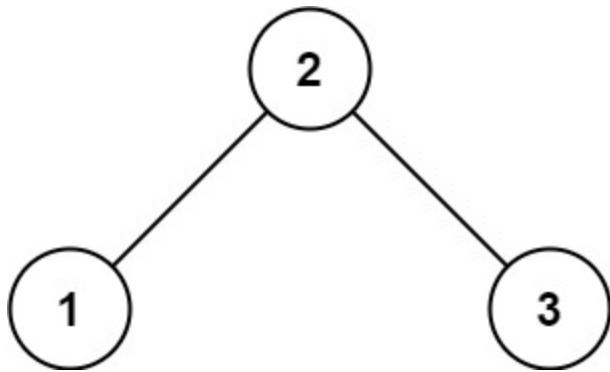
要求：判断其是否是一个有效的二叉搜索树。

说明：

- 二叉搜索树特征：
 - 节点的左子树只包含小于当前节点的数。
 - 节点的右子树只包含大于当前节点的数。
 - 所有左子树和右子树自身必须也是二叉搜索树。
- 树中节点数目范围在 $[1, 10^4]$ 内。
- $-2^{31} \leq \text{Node.val} \leq 2^{31} - 1$ 。

示例：

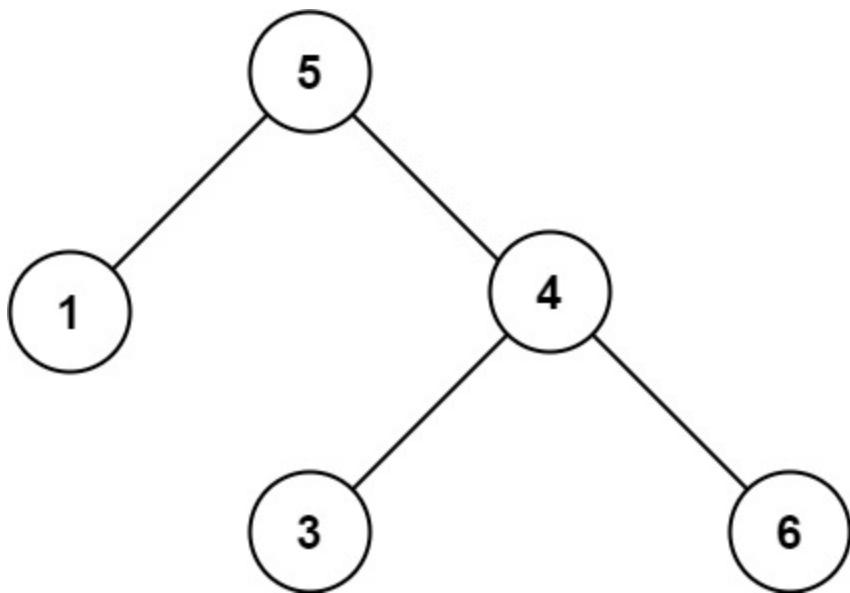
- 示例 1：



输入: `root = [2,1,3]`

输出: `true`

- 示例 2：



输入: `root = [5,1,4,null,null,3,6]`

输出: `false`

解释: 根节点的值是 `5`，但是右子节点的值是 `4`。

解题思路

思路 1：递归遍历

根据题意进行递归遍历即可。前序、中序、后序遍历都可以。

1. 以前序遍历为例，递归函数为： `preorderTraversal(root, min_v, max_v)`。
2. 前序遍历时，先判断根节点的值是否在 (\min_v, \max_v) 之间。
 - i. 如果不在则直接返回 `False`。
 - ii. 如果在区间内，则继续递归检测左右子树是否满足，都满足才是一棵二叉搜索树。
3. 当递归遍历左子树的时候，要将上界 \max_v 改为左子树的根节点值，因为左子树上所有节点的值均小于根节点的值。
4. 当递归遍历右子树的时候，要将下界 \min_v 改为右子树的根节点值，因为右子树上所有节点的值均大于根节点。

思路 1：代码

```
class Solution:  
    def isValidBST(self, root: TreeNode) -> bool:  
        def preorderTraversal(root, min_v, max_v):  
            if root == None:  
                return True  
            if root.val >= max_v or root.val <= min_v:  
                return False  
            return preorderTraversal(root.left, min_v, root.val) and preorderTraversal(  
                root.right, root.val, max_v)  
  
        return preorderTraversal(root, float('-inf'), float('inf'))
```

思路 1：复杂度分析

- **时间复杂度：** $O(n)$ ，其中 n 是二叉树的节点数目。
- **空间复杂度：** $O(n)$ 。递归函数需要用到栈空间，栈空间取决于递归深度，最坏情况下递归深度为 n ，所以空间复杂度为 $O(n)$ 。# 0100. 相同的树
- 标签：树、深度优先搜索、广度优先搜索、二叉树
- 难度：简单

题目链接

- 0100. 相同的树 - 力扣

题目大意

描述：给定两个二叉树的根节点 p 和 q 。

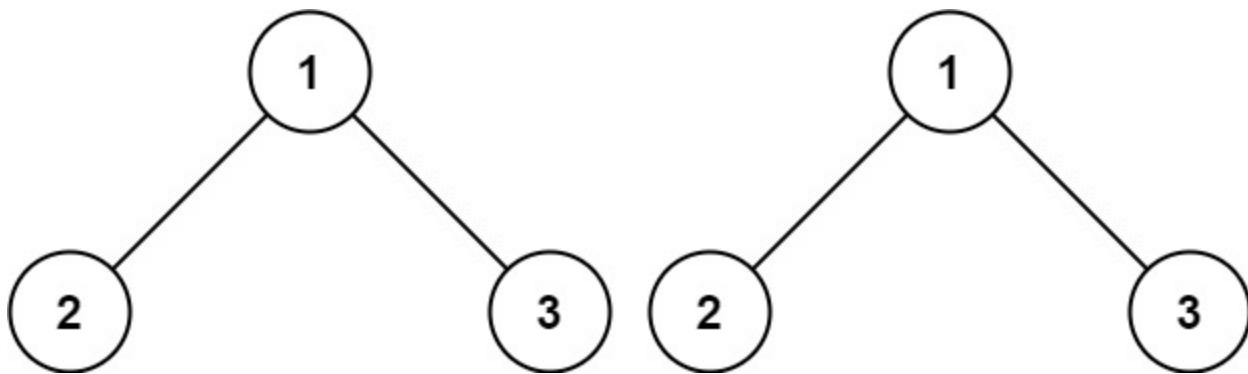
要求：判断这两棵树是否相同。

说明：

- **两棵树相同的定义：**结构上相同；节点具有相同的值。
- 两棵树上的节点数目都在范围 $[0, 100]$ 内。
- $-10^4 \leq \text{Node.val} \leq 10^4$ 。

示例：

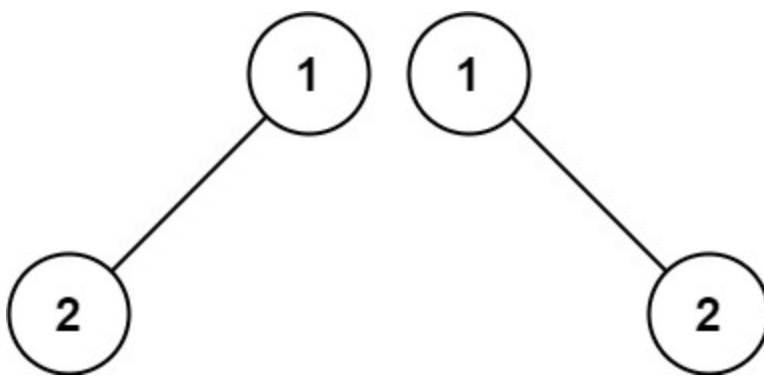
- **示例 1：**



输入： $p = [1, 2, 3]$, $q = [1, 2, 3]$

输出： True

- **示例 2：**



输入: `p = [1,2], q = [1,null,2]`

输出: `False`

解题思路

思路 1：递归

1. 先判断两棵树的根节点是否相同。
2. 然后再递归地判断左右子树是否相同。

思路 1：代码

```

class Solution:
    def isSameTree(self, p: TreeNode, q: TreeNode) -> bool:
        if not p and not q:
            return True
        if not p or not q:
            return False
        if p.val != q.val:
            return False
        return self.isSameTree(p.left, q.left) and self.isSameTree(p.right, q.right)

```

思路 1：复杂度分析

- 时间复杂度: $O(\min(m, n))$, 其中 m 、 n 分别为两棵树中的节点数量。
- 空间复杂度: $O(\min(m, n))$ 。

0101. 对称二叉树

- 标签：树、深度优先搜索、广度优先搜索、二叉树
- 难度：简单

题目链接

- [0101. 对称二叉树 - 力扣](#)

题目大意

描述：给定一个二叉树的根节点 `root`。

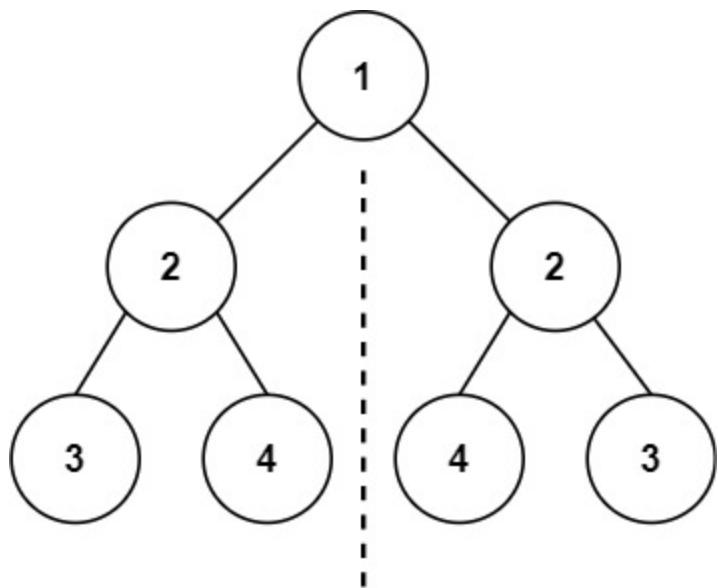
要求：判断该二叉树是否是左右对称的。

说明：

- 树中节点数目在范围 $[1, 1000]$ 内。
- $-100 \leq \text{Node.val} \leq 100$ 。

示例：

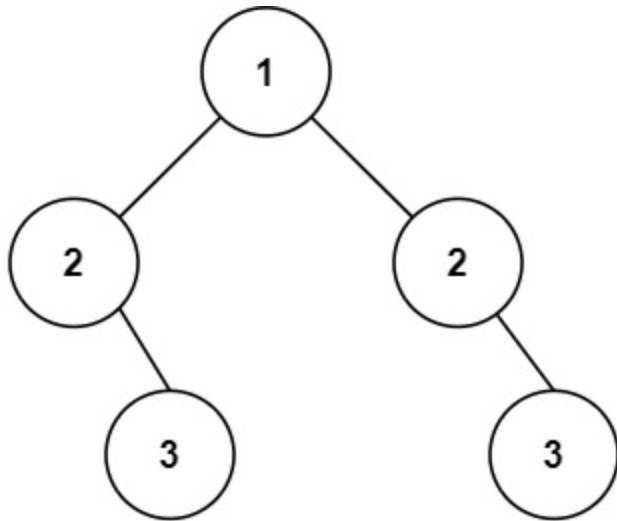
- **示例 1：**



输入: `root = [1,2,2,3,4,4,3]`

输出: `true`

- 示例 2：



输入：root = [1,2,2,null,3,null,3]

输出：false

解题思路

思路 1：递归遍历

如果一棵二叉树是对称的，那么其左子树和右子树的外侧节点的节点值应当是相等的，并且其左子树和右子树的内侧节点的节点值也应当是相等的。

那么我们可以通过递归方式，检查其左子树与右子树外侧节点和内测节点是否相等。即递归检查左子树的左子节点值与右子树的右子节点值是否相等（外侧节点值是否相等），递归检查左子树的右子节点值与右子树的左子节点值是否相等（内测节点值是否相等）。

具体步骤如下：

1. 如果当前根节点为 `None`，则直接返回 `True`。
2. 如果当前根节点不为 `None`，则调用 `check(left, right)` 方法递归检查其左右子树是否对称。
 - i. 如果左子树节点为 `None`，并且右子树节点也为 `None`，则直接返回 `True`。
 - ii. 如果左子树节点为 `None`，并且右子树节点不为 `None`，则直接返回 `False`。
 - iii. 如果左子树节点不为 `None`，并且右子树节点为 `None`，则直接返回 `False`。
 - iv. 如果左子树节点值不等于右子树节点值，则直接返回 `False`。
 - v. 如果左子树节点不为 `None`，并且右子树节点不为 `None`，并且左子树节点值等于右子树节点值，则：
 - a. 递归检测左右子树的外侧节点是否相等。

- b. 递归检测左右子树的内测节点是否相等。
- c. 如果左右子树的外侧节点、内测节点值相等，则返回 True。

思路 1：代码

```

class Solution:
    def isSymmetric(self, root: TreeNode) -> bool:
        if root == None:
            return True
        return self.check(root.left, root.right)

    def check(self, left: TreeNode, right: TreeNode):
        if left == None and right == None:
            return True
        elif left == None and right != None:
            return False
        elif left != None and right == None:
            return False
        elif left.val != right.val:
            return False

        return self.check(left.left, right.right) and self.check(left.right, right.left)

```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ ，其中 n 是二叉树的节点数目。
- 空间复杂度： $O(n)$ 。递归函数需要用到栈空间，栈空间取决于递归深度，最坏情况下递归深度为 n ，所以空间复杂度为 $O(n)$ 。

0102. 二叉树的层序遍历

- 标签：树、广度优先搜索、二叉树
- 难度：中等

题目链接

- [0102. 二叉树的层序遍历 - 力扣](#)

题目大意

描述：给定一个二叉树的根节点 `root`。

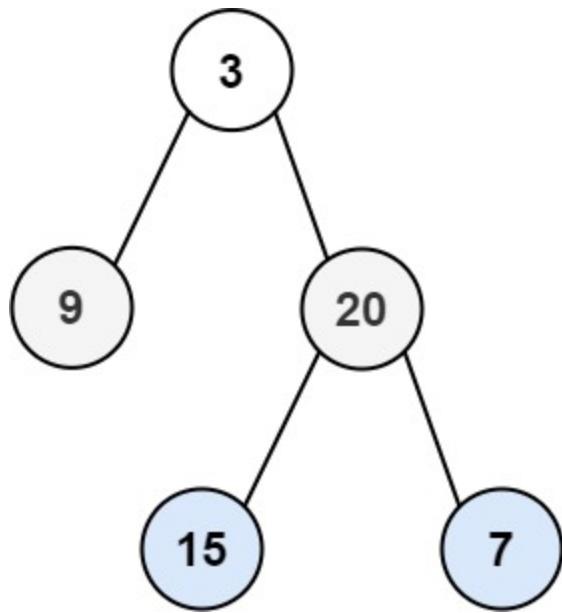
要求：返回该二叉树按照「层序遍历」得到的节点值。

说明：

- 返回结果为二维数组，每一层都要存为数组返回。

示例：

- **示例 1：**



输入: `root = [3,9,20,null,null,15,7]`

输出: `[[3],[9,20],[15,7]]`

- **示例 2：**

输入: `root = [1]`

输出: `[[1]]`

解题思路

思路 1：广度优先搜索

广度优先搜索，需要增加一些变化。普通广度优先搜索只取一个元素，变化后的广度优先搜索每次取出第 i 层上所有元素。

具体步骤如下：

1. 判断二叉树是否为空，为空则直接返回。
2. 令根节点入队。
3. 当队列不为空时，求出当前队列长度 s_i 。
4. 依次从队列中取出这 s_i 个元素，并对这 s_i 个元素依次进行访问。然后将其左右孩子节点入队，然后继续遍历下一层节点。
5. 当队列为空时，结束遍历。

思路 1：代码

```
class Solution:  
    def levelOrder(self, root: TreeNode) -> List[List[int]]:  
        if not root:  
            return []  
        queue = [root]  
        order = []  
        while queue:  
            level = []  
            size = len(queue)  
            for _ in range(size):  
                curr = queue.pop(0)  
                level.append(curr.val)  
                if curr.left:  
                    queue.append(curr.left)  
                if curr.right:  
                    queue.append(curr.right)  
            if level:  
                order.append(level)  
        return order
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。其中 n 是二叉树的节点数目。
- 空间复杂度： $O(n)$ 。[# 0103. 二叉树的锯齿形层序遍历](#)
- 标签：树、广度优先搜索、二叉树
- 难度：中等

题目链接

- [0103. 二叉树的锯齿形层序遍历 - 力扣](#)

题目大意

描述：给定一个二叉树的根节点 `root`。

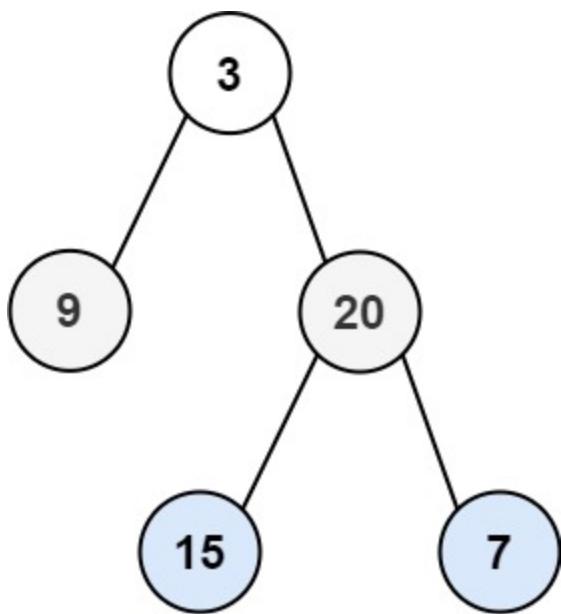
要求：返回其节点值的锯齿形层序遍历结果。

说明：

- **锯齿形层序遍历：**从左往右，再从右往左进行下一层遍历，以此类推，层与层之间交替进行。

示例：

- **示例 1：**



输入: root = [3, 9, 20, null, null, 15, 7]

输出: [[3], [20, 9], [15, 7]]

- 示例 2:

输入: root = [1]

输出: [[1]]

解题思路

思路 1：广度优先搜索

在二叉树的层序遍历的基础上需要增加一些变化。

普通广度优先搜索只取一个元素，变化后的广度优先搜索每次取出第 i 层上所有元素。

新增一个变量 `odd`，用于判断当前层数是奇数层，还是偶数层。从而判断元素遍历方向。

存储每层元素的 `level` 列表改用双端队列，如果是奇数层，则从末尾添加元素。如果是偶数层，则从头部添加元素。

具体步骤如下：

1. 使用列表 `order` 存放锯齿形层序遍历结果，使用整数 `odd` 变量用于判断奇偶层，使用双端队列 `level` 存放每层元素，使用列表 `queue` 用于进行广度优先搜索。
2. 将根节点放入入队列中，即 `queue = [root]`。
3. 当队列 `queue` 不为空时，求出当前队列长度 s_i ，并判断当前层数的奇偶性。
 - i. 如果当前层为奇数层，如果是奇数层，则从 `level` 末尾添加元素。
 - ii. 如果当前层是偶数层，则从 `level` 头部添加元素。
 - iii. 然后将当前元素的左右子节点加入队列 `queue` 中，然后继续迭代。
4. 依次从队列中取出这 s_i 个元素。
5. 将存储当前层元素的 `level` 存入答案列表 `order` 中。
6. 当队列为空时，结束。返回锯齿形层序遍历结果 `order`。

思路 1：代码

```
import collections
class Solution:
    def zigzagLevelOrder(self, root: TreeNode) -> List[List[int]]:
        if not root:
            return []
        queue = [root]
        order = []
        odd = True
        while queue:
            level = collections.deque()
            size = len(queue)
            for _ in range(size):
                curr = queue.pop(0)
                if odd:
                    level.append(curr.val)
                else:
                    level.appendleft(curr.val)
                if curr.left:
                    queue.append(curr.left)
                if curr.right:
                    queue.append(curr.right)
            if level:
                order.append(list(level))
            odd = not odd
        return order
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。其中 n 是二叉树的节点数目。
- 空间复杂度： $O(n)$ 。

0104. 二叉树的最大深度

- 标签：树、深度优先搜索、广度优先搜索、二叉树
- 难度：简单

题目链接

- 0104. 二叉树的最大深度 - 力扣

题目大意

描述：给定一个二叉树的根节点 `root`。

要求：找出该二叉树的最大深度。

说明：

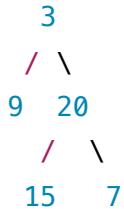
- 二叉树的深度：根节点到最远叶子节点的最长路径上的节点数。
- 叶子节点：没有子节点的节点。

示例：

- 示例 1：

输入：[`3,9,20,null,null,15,7`]

对应二叉树



输出：`3`

解释：该二叉树的最大深度为 `3`

解题思路

思路 1：递归算法

根据递归三步走策略，写出对应的递归代码。

1. 写出递推公

式：当前二叉树的最大深度 = `max`(当前二叉树左子树的最大深度，当前二叉树右子树的最大深度) + 1。

- 即：先得到左右子树的高度，在计算当前节点的高度。

2. 明确终止条件：当前二叉树为空。

3. 翻译为递归代码：

- i. 定义递归函数： `maxDepth(self, root)` 表示输入参数为二叉树的根节点 `root`，返回结果为该二叉树的最大深度。
- ii. 书写递归主
体：`return max(self.maxDepth(root.left) + self.maxDepth(root.right))`。
- iii. 明确递归终止条件：`if not root: return 0`

思路 1：代码

```
class Solution:  
    def maxDepth(self, root: Optional[TreeNode]) -> int:  
        if not root:  
            return 0  
  
        return max(self.maxDepth(root.left), self.maxDepth(root.right)) + 1
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ ，其中 n 是二叉树的节点数目。
- 空间复杂度： $O(n)$ 。递归函数需要用到栈空间，栈空间取决于递归深度，最坏情况下递归深度为 n ，所以空间复杂度为 $O(n)$ 。[# 0105. 从前序与中序遍历序列构造二叉树](#)
- 标签：树、数组、哈希表、分治、二叉树
- 难度：中等

题目链接

- [0105. 从前序与中序遍历序列构造二叉树 - 力扣](#)

题目大意

描述：给定一棵二叉树的前序遍历结果 `preorder` 和中序遍历结果 `inorder`。

要求：构造出该二叉树并返回其根节点。

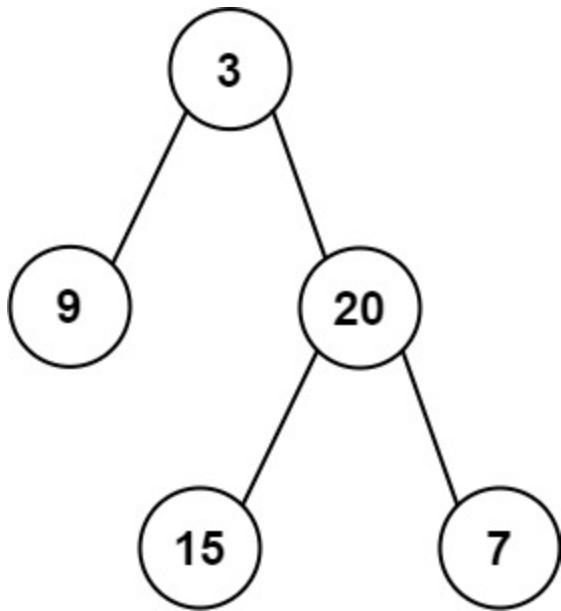
说明：

- $1 \leq preorder.length \leq 3000$ 。
- $inorder.length == preorder.length$ 。

- $-3000 \leq preorder[i], inorder[i] \leq 3000$ 。
- `preorder` 和 `inorder` 均无重复元素。
- `inorder` 均出现在 `preorder`。
- `preorder` 保证为二叉树的前序遍历序列。
- `inorder` 保证为二叉树的中序遍历序列。

示例：

- 示例 1：



输入： `preorder = [3, 9, 20, 15, 7]`, `inorder = [9, 3, 15, 20, 7]`

输出： `[3, 9, 20, null, null, 15, 7]`

- 示例 2：

输入： `preorder = [-1]`, `inorder = [-1]`

输出： `[-1]`

解题思路

思路 1：递归遍历

前序遍历的顺序是：根 -> 左 -> 右。中序遍历的顺序是：左 -> 根 -> 右。根据前序遍历的顺序，可以找到根节点位置。然后在中序遍历的结果中可以找到对应的根节点位置，就可以从根节点位置将二叉树分

割成左子树、右子树。同时能得到左右子树的节点个数。此时构建当前节点，并递归建立左右子树，在左右子树对应位置继续递归遍历进行上述步骤，直到节点为空，具体操作步骤如下：

1. 从前序遍历顺序中当前根节点的位置在 `postorder[0]`。
2. 通过在中序遍历中查找上一步根节点对应的位置 `inorder[k]`，从而将二叉树的左右子树分隔开，并得到左右子树节点的个数。
3. 从上一步得到的左右子树个数将前序遍历结果中的左右子树分开。
4. 构建当前节点，并递归建立左右子树，在左右子树对应位置继续递归遍历并执行上述三步，直到节点为空。

思路 1：代码

```
class Solution:  
    def buildTree(self, preorder: List[int], inorder: List[int]) -> TreeNode:  
        def createTree(preorder, inorder, n):  
            if n == 0:  
                return None  
            k = 0  
            while preorder[0] != inorder[k]:  
                k += 1  
            node = TreeNode(inorder[k])  
            node.left = createTree(preorder[1: k+1], inorder[0: k], k)  
            node.right = createTree(preorder[k+1:], inorder[k+1:], n-k-1)  
            return node  
        return createTree(preorder, inorder, len(inorder))
```

思路 1：复杂度分析

- **时间复杂度**: $O(n)$ ，其中 n 是二叉树的节点数目。
- **空间复杂度**: $O(n)$ 。递归函数需要用到栈空间，栈空间取决于递归深度，最坏情况下递归深度为 n ，所以空间复杂度为 $O(n)$ 。[# 0106. 从中序与后序遍历序列构造二叉树](#)
- 标签：树、数组、哈希表、分治、二叉树
- 难度：中等

题目链接

- [0106. 从中序与后序遍历序列构造二叉树 - 力扣](#)

题目大意

描述：给定一棵二叉树的中序遍历结果 `inorder` 和后序遍历结果 `postorder`。

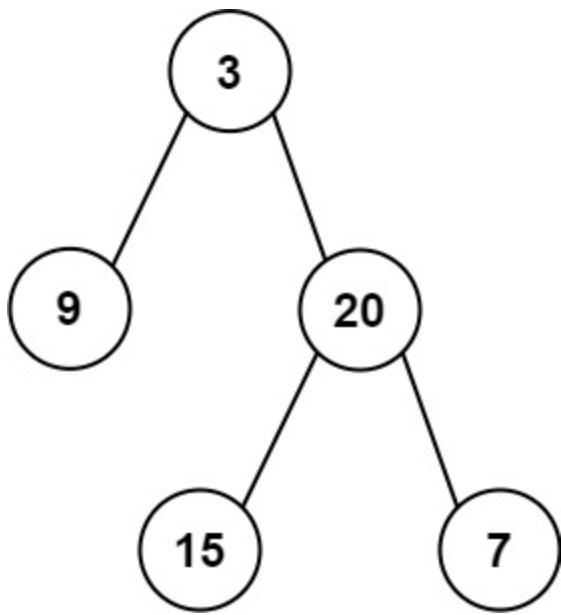
要求：构造出该二叉树并返回其根节点。

说明：

- $1 \leq \text{inorder.length} \leq 3000$ 。
- $\text{postorder.length} == \text{inorder.length}$ 。
- $-3000 \leq \text{inorder}[i], \text{postorder}[i] \leq 3000$ 。
- `inorder` 和 `postorder` 都由不同的值组成。
- `postorder` 中每一个值都在 `inorder` 中。
- `inorder` 保证是二叉树的中序遍历序列。
- `postorder` 保证是二叉树的后序遍历序列。
- `inorder` 保证为二叉树的中序遍历序列。

示例：

- 示例 1：



输入：`inorder = [9, 3, 15, 20, 7]`, `postorder = [9, 15, 7, 20, 3]`

输出：`[3, 9, 20, null, null, 15, 7]`

- 示例 2：

输入: inorder = [-1], postorder = [-1]

输出: [-1]

解题思路

思路 1：递归

中序遍历的顺序是：左 -> 根 -> 右。后序遍历的顺序是：左 -> 右 -> 根。根据后序遍历的顺序，可以找到根节点位置。然后在中序遍历的结果中可以找到对应的根节点位置，就可以从根节点位置将二叉树分割成左子树、右子树。同时能得到左右子树的节点个数。此时构建当前节点，并递归建立左右子树，在左右子树对应位置继续递归遍历进行上述步骤，直到节点为空，具体操作步骤如下：

1. 从后序遍历顺序中当前根节点的位置在 `postorder[n - 1]`。
2. 通过在中序遍历中查找上一步根节点对应的位置 `inorder[k]`，从而将二叉树的左右子树分隔开，并得到左右子树节点的个数。
3. 从上一步得到的左右子树个数将后序遍历结果中的左右子树分开。
4. 构建当前节点，并递归建立左右子树，在左右子树对应位置继续递归遍历并执行上述三步，直到节点为空。

思路 1：代码

```
class Solution:  
    def buildTree(self, inorder: List[int], postorder: List[int]) -> TreeNode:  
        def createTree(inorder, postorder, n):  
            if n == 0:  
                return None  
            k = 0  
            while postorder[n-1] != inorder[k]:  
                k += 1  
            node = TreeNode(inorder[k])  
            node.right = createTree(inorder[k+1: n], postorder[k: n-1], n-k-1)  
            node.left = createTree(inorder[0: k], postorder[0: k], k)  
            return node  
        return createTree(inorder, postorder, len(postorder))
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ ，其中 n 是二叉树的节点数目。

- **空间复杂度**: $O(n)$ 。递归函数需要用到栈空间，栈空间取决于递归深度，最坏情况下递归深度为 n ，所以空间复杂度为 $O(n)$ 。

0107. 二叉树的层序遍历 II

- 标签：树、广度优先搜索、二叉树
- 难度：中等

题目链接

- [0107. 二叉树的层序遍历 II - 力扣](#)

题目大意

描述：给定一个二叉树的根节点 $root$ 。

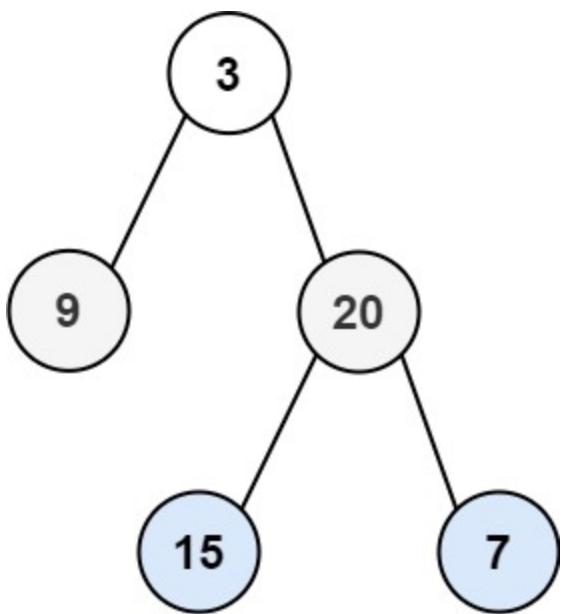
要求：返回其节点值按照「自底向上」的「层序遍历」（即按从叶子节点所在层到根节点所在的层，逐层从左向右遍历）。

说明：

- 树中节点数目在范围 $[0, 2000]$ 内。
- $-1000 \leq Node.val \leq 1000$ 。

示例：

- **示例 1：**



输入: `root = [3, 9, 20, null, null, 15, 7]`

输出: `[[15, 7], [9, 20], [3]]`

- 示例 2:

输入: `root = [1]`

输出: `[[1]]`

解题思路

思路 1：二叉树的层次遍历

先得到层次遍历的节点顺序，再将其进行反转返回即可。

其中层次遍历用到了广度优先搜索，不过需要增加一些变化。普通广度优先搜索只取一个元素，变化后的广度优先搜索每次取出第 i 层上所有元素。

具体步骤如下：

1. 根节点入队。
2. 当队列不为空时，求出当前队列长度 s_i 。
3. 依次从队列中取出这 s_i 个元素，将其左右子节点入队，然后继续迭代。
4. 当队列为空时，结束。

思路 1：代码

```
class Solution:
    def levelOrderBottom(self, root: TreeNode) -> List[List[int]]:
        if not root:
            return []
        queue = [root]
        order = []
        while queue:
            level = []
            size = len(queue)
            for _ in range(size):
                curr = queue.pop(0)
                level.append(curr.val)
                if curr.left:
                    queue.append(curr.left)
                if curr.right:
                    queue.append(curr.right)
            if level:
                order.append(level)
        return order[::-1]
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ ，其中 n 为树中节点个数。
- 空间复杂度： $O(n)$ 。

0108. 将有序数组转换为二叉搜索树

- 标签：树、二叉搜索树、数组、分治、二叉树
- 难度：简单

题目链接

- [0108. 将有序数组转换为二叉搜索树 - 力扣](#)

题目大意

描述：给定一个升序的有序数组 `nums`。

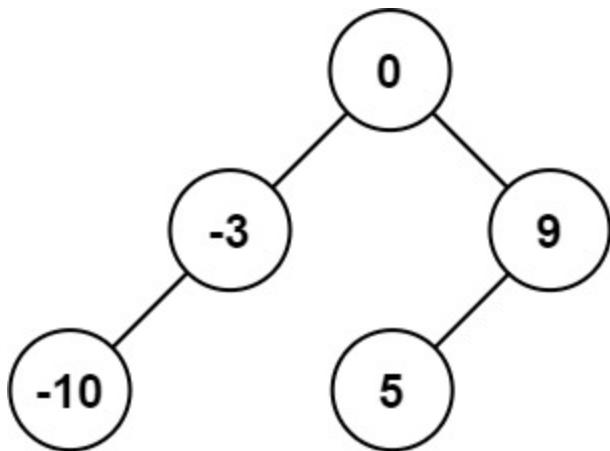
要求：将其转换为一棵高度平衡的二叉搜索树。

说明：

- $1 \leq \text{nums.length} \leq 10^4$ 。
- $-10^4 \leq \text{nums}[i] \leq 10^4$ 。
- `nums` 按严格递增顺序排列。

示例：

- **示例 1：**

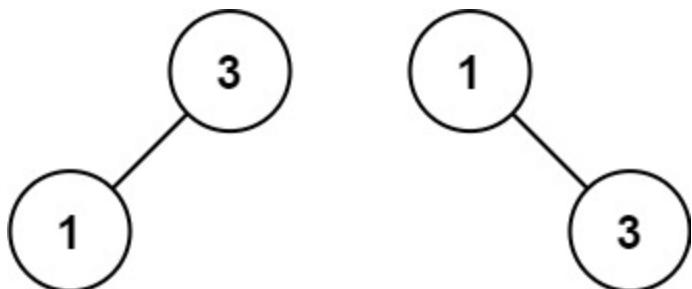


输入: `nums = [-10, -3, 0, 5, 9]`

输出: `[0, -3, 9, -10, null, 5]`

解释: `[0, -10, 5, null, -3, null, 9]` 也将被视为正确答案

- **示例 2：**



输入: `nums = [1,3]`
输出: `[3,1]`
解释: `[1,null,3]` 和 `[3,1]` 都是高度平衡二叉搜索树。

解题思路

思路 1：递归遍历

直观上，如果把数组的中间元素当做根，那么数组左侧元素都小于根节点，右侧元素都大于根节点，且左右两侧元素个数相同，或最多相差 1 个。那么构建的树高度差也不会超过 1。

所以猜想到：如果左右子树越平均，树就越平衡。这样我们就可以每次取中间元素作为当前的根节点，两侧的元素作为左右子树递归建树，左侧区间 $[L, mid - 1]$ 作为左子树，右侧区间 $[mid + 1, R]$ 作为右子树。

思路 1：代码

```
class Solution:
    def sortedArrayToBST(self, nums: List[int]) -> Optional[TreeNode]:
        def build(left, right):
            if left > right:
                return
            mid = left + (right - left) // 2
            root = TreeNode(nums[mid])
            root.left = build(left, mid - 1)
            root.right = build(mid + 1, right)
            return root
        return build(0, len(nums) - 1)
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。其中 n 是数组的长度。
- 空间复杂度: $O(n)$ 。[# 0110. 平衡二叉树](#)
- 标签: 树、深度优先搜索、二叉树
- 难度: 简单

题目链接

- 0110. 平衡二叉树 - 力扣

题目大意

描述：给定一个二叉树的根节点 `root`。

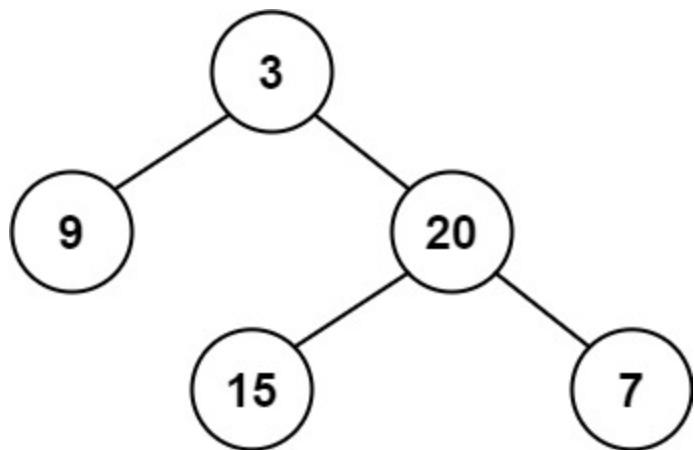
要求：判断该二叉树是否是高度平衡的二叉树。

说明：

- **高度平衡二叉树：**二叉树中每个节点的左右两个子树的高度差的绝对值不超过 1。
- 树中的节点数在范围 $[0, 5000]$ 内。
- $-10^4 \leq \text{Node.val} \leq 10^4$ 。

示例：

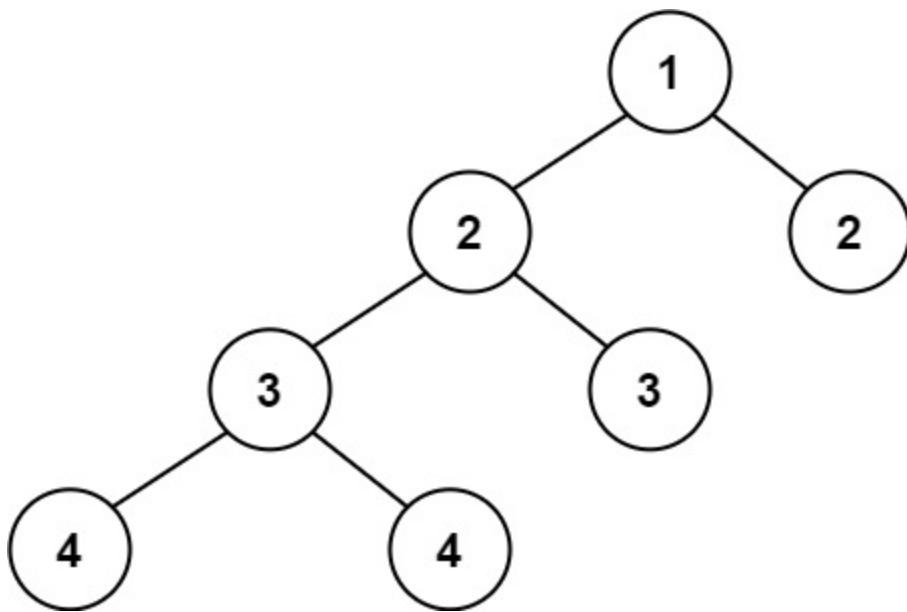
- **示例 1：**



输入: `root = [3,9,20,null,null,15,7]`

输出: `True`

- **示例 2：**



输入: `root = [1,2,2,3,3,null,null,4,4]`

输出: `False`

解题思路

思路 1：递归遍历

1. 先递归遍历左右子树，判断左右子树是否平衡，再判断以当前节点为根节点的左右子树是否平衡。
2. 如果遍历的子树是平衡的，则返回它的高度，否则返回 -1。
3. 只要出现不平衡的子树，则该二叉树一定不是平衡二叉树。

思路 1：代码

```

class Solution:
    def isBalanced(self, root: TreeNode) -> bool:
        def height(root: TreeNode) -> int:
            if root == None:
                return False
            leftHeight = height(root.left)
            rightHeight = height(root.right)
            if leftHeight == -1 or rightHeight == -1 or abs(leftHeight-rightHeight) > 1:
                return -1
            else:
                return max(leftHeight, rightHeight)+1
        return height(root) >= 0
  
```

思路 1：复杂度分析

- **时间复杂度**: $O(n)$, 其中 n 是二叉树的节点数目。
- **空间复杂度**: $O(n)$ 。递归函数需要用到栈空间, 栈空间取决于递归深度, 最坏情况下递归深度为 n , 所以空间复杂度为 $O(n)$ 。

0111. 二叉树的最小深度

- 标签: 树、深度优先搜索、广度优先搜索
- 难度: 简单

题目链接

- [0111. 二叉树的最小深度 - 力扣](#)

题目大意

描述: 给定一个二叉树的根节点 $root$ 。

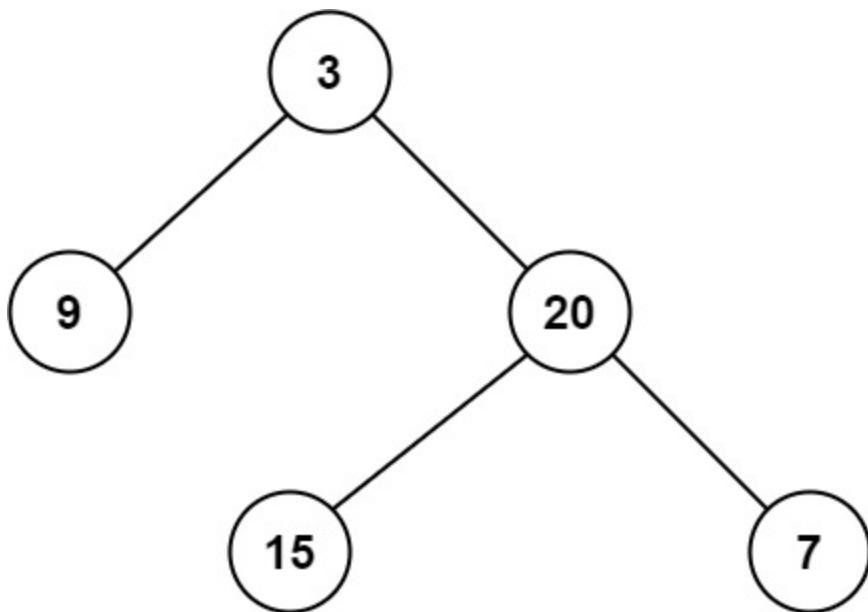
要求: 找出该二叉树的最小深度。

说明:

- **最小深度**: 从根节点到最近叶子节点的最短路径上的节点数量。
- **叶子节点**: 指没有子节点的节点。
- 树中节点数的范围在 $[0, 10^5]$ 内。
- $-1000 \leq Node.val \leq 1000$ 。

示例:

- **示例 1:**



输入: `root = [3,9,20,null,null,15,7]`

输出: `2`

- 示例 2:

输入: `root = [2,null,3,null,4,null,5,null,6]`

输出: `5`

解题思路

思路 1：深度优先搜索

深度优先搜索递归遍历左右子树，记录最小深度。

对于每一个非叶子节点，计算其左右子树的最小叶子节点深度，将较小的深度+1 即为当前节点的最小叶子节点深度。

思路 1：代码

```
class Solution:
    def minDepth(self, root: TreeNode) -> int:
        # 遍历到空节点，直接返回 0
        if root == None:
            return 0

        # 左右子树为空，说明为叶子节点 返回 1
        if root.left == None and root.right == None:
            return 1

        leftHeight = self.minDepth(root.left)
        rightHeight = self.minDepth(root.right)

        # 当前节点的左右子树的最小叶子节点深度
        min_depth = 0xfffffff
        if root.left:
            min_depth = min(leftHeight, min_depth)
        if root.right:
            min_depth = min(rightHeight, min_depth)

        # 当前节点的最小叶子节点深度
        return min_depth + 1
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ ，其中 n 是树中的节点数量。
- 空间复杂度： $O(n)$ 。

0112. 路径总和

- 标签：树、深度优先搜索
- 难度：简单

题目链接

- [0112. 路径总和 - 力扣](#)

题目大意

描述：给定一个二叉树的根节点 `root` 和一个值 `targetSum`。

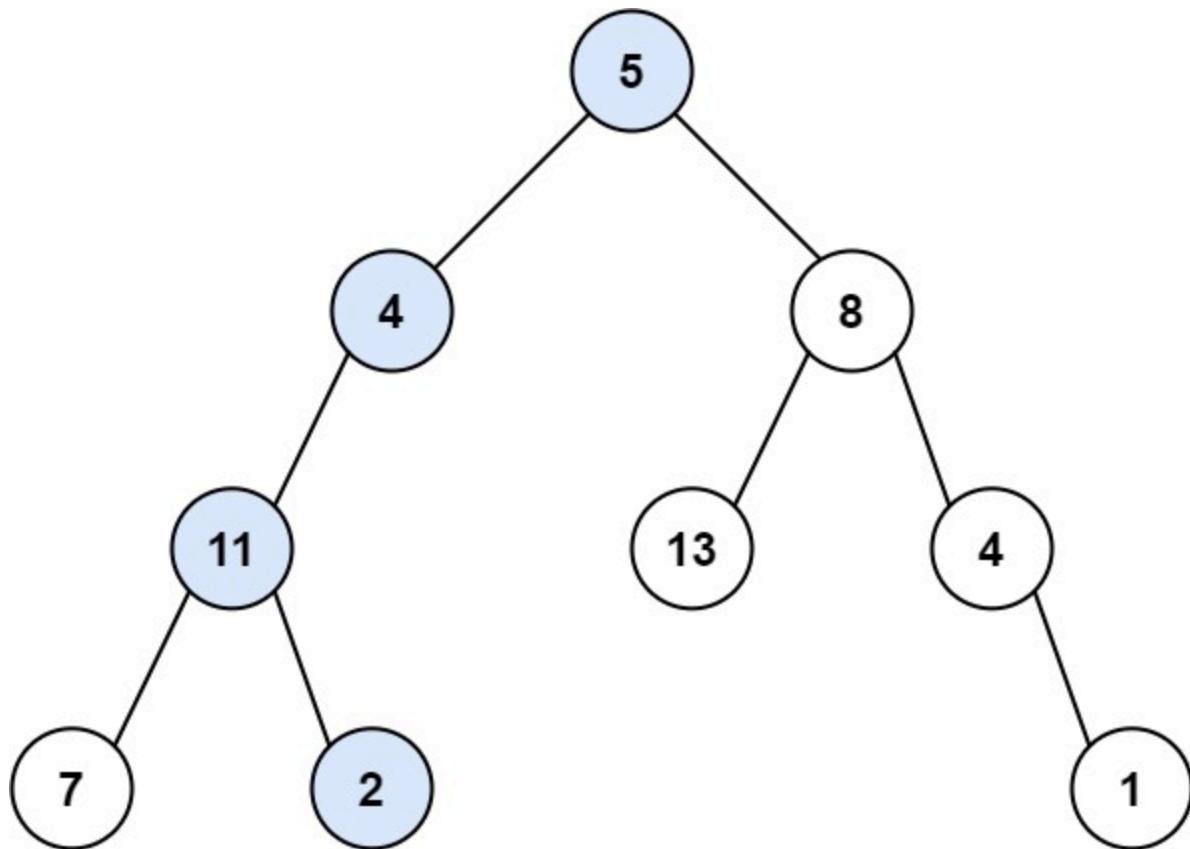
要求：判断该树中是否存在从根节点到叶子节点的路径，使得这条路径上所有节点值相加等于 `targetSum`。如果存在，返回 `True`；否则，返回 `False`。

说明：

- 树中节点的数目在范围 $[0, 5000]$ 内。
- $-1000 \leq \text{Node.val} \leq 1000$ 。
- $-1000 \leq \text{targetSum} \leq 1000$ 。

示例：

- **示例 1：**

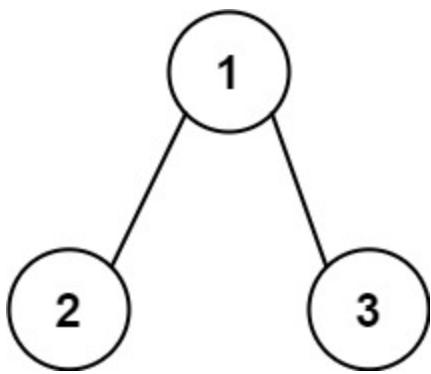


输入: `root = [5,4,8,11,null,13,4,7,2,null,null,null,1]`, `targetSum = 22`

输出: `true`

解释：等于目标和的根节点到叶节点路径如上图所示。

- **示例 2：**



输入: `root = [1,2,3], targetSum = 5`

输出: `false`

解释: 树中存在两条根节点到叶子节点的路径:

`(1 --> 2)`: 和为 `3`

`(1 --> 3)`: 和为 `4`

不存在 `sum = 5` 的根节点到叶子节点的路径。

解题思路

思路 1：递归遍历

1. 定义一个递归函数，递归函数传入当前根节点 `root`，目标节点和 `targetSum`，以及新增变量 `currSum`（表示为从根节点到当前节点的路径上所有节点值之和）。
2. 递归遍历左右子树，同时更新维护 `currSum` 值。
3. 如果当前节点为叶子节点时，判断 `currSum` 是否与 `targetSum` 相等。
 - i. 如果 `currSum` 与 `targetSum` 相等，则返回 `True`。
 - ii. 如果 `currSum` 不与 `targetSum` 相等，则返回 `False`。
4. 如果当前节点不为叶子节点，则继续递归遍历左右子树。

思路 1：代码

```
class Solution:
    def hasPathSum(self, root: TreeNode, targetSum: int) -> bool:
        return self.sum(root, targetSum, 0)

    def sum(self, root: TreeNode, targetSum: int, curSum:int) -> bool:
        if root == None:
            return False
        curSum += root.val
        if root.left == None and root.right == None:
            return curSum == targetSum
        else:
            return self.sum(root.left, targetSum, curSum) or self.sum(root.right, targetSum, curSum)
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ ，其中 n 是二叉树的节点数目。
- 空间复杂度： $O(n)$ 。递归函数需要用到栈空间，栈空间取决于递归深度，最坏情况下递归深度为 n ，所以空间复杂度为 $O(n)$ 。# 0113. 路径总和 II
- 标签：树、深度优先搜索、回溯、二叉树
- 难度：中等

题目链接

- [0113. 路径总和 II - 力扣](#)

题目大意

描述：给定一棵二叉树的根节点 `root` 和一个整数目标 `targetSum`。

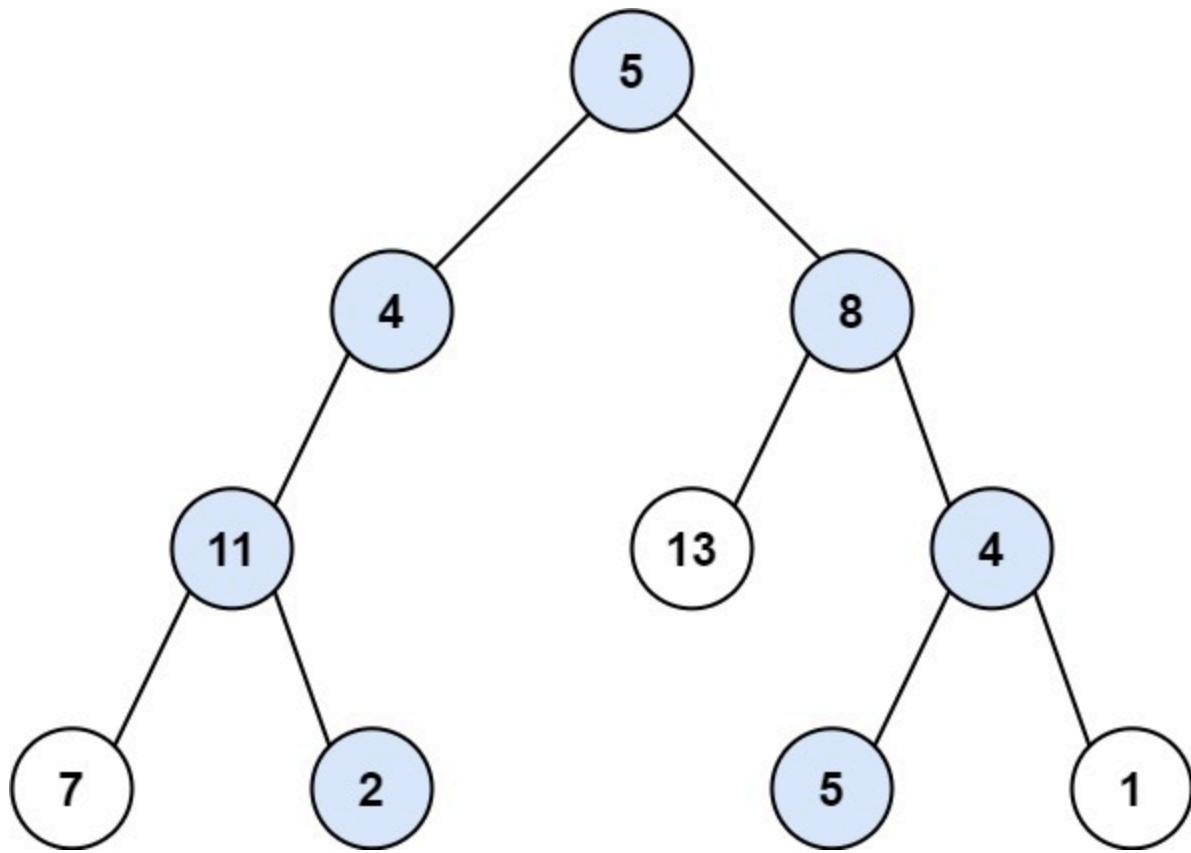
要求：找出「所有从根节点到叶子节点路径总和」等于给定目标和 `targetSum` 的路径。

说明：

- **叶子节点：**指没有子节点的节点。
- 树中节点总数在范围 $[0, 5000]$ 内。
- $-1000 \leq Node.val \leq 1000$ 。
- $-1000 \leq targetSum \leq 1000$ 。

示例：

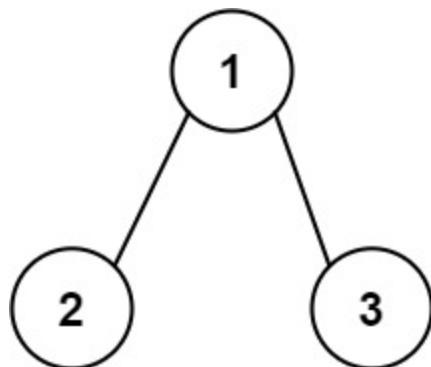
- 示例 1：



输入: root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22

输出: [[5,4,11,2],[5,8,4,5]]

- 示例 2：



输入: root = [1,2,3], targetSum = 5

输出: []

解题思路

思路 1：回溯

在回溯的同时，记录下当前路径。同时维护 `targetSum`，每遍历到一个节点，就减去该节点值。如果遇到叶子节点，并且 `targetSum == 0` 时，将当前路径加入答案数组中。然后递归遍历左右子树，并回退当前节点，继续遍历。

具体步骤如下：

1. 使用列表 `res` 存储所有路径，使用列表 `path` 存储当前路径。
2. 如果根节点为空，则直接返回。
3. 将当前节点值添加到当前路径 `path` 中。
4. `targetSum` 减去当前节点值。
5. 如果遇到叶子节点，并且 `targetSum == 0` 时，将当前路径加入答案数组中。
6. 递归遍历左子树。
7. 递归遍历右子树。
8. 回退当前节点，继续递归遍历。

思路 1：代码

```
class Solution:  
    def pathSum(self, root: TreeNode, targetSum: int) -> List[List[int]]:  
        res = []  
        path = []  
  
        def dfs(root: TreeNode, targetSum: int):  
            if not root:  
                return  
            path.append(root.val)  
            targetSum -= root.val  
            if not root.left and not root.right and targetSum == 0:  
                res.append(path[:])  
            dfs(root.left, targetSum)  
            dfs(root.right, targetSum)  
            path.pop()  
  
        dfs(root, targetSum)  
        return res
```

思路 1：复杂度分析

- **时间复杂度**: $O(n^2)$, 其中 n 是二叉树的节点数目。
- **空间复杂度**: $O(n)$ 。递归函数需要用到栈空间, 栈空间取决于递归深度, 最坏情况下递归深度为 n , 所以空间复杂度为 $O(n)$ 。

0115. 不同的子序列

- 标签: 字符串、动态规划
- 难度: 困难

题目链接

- [0115. 不同的子序列 - 力扣](#)

题目大意

描述: 给定两个字符串 s 和 t 。

要求: 计算在 s 的子序列中 t 出现的个数。

说明:

- **字符串的子序列**: 通过删除一些 (也可以不删除) 字符且不干扰剩余字符相对位置所组成的新字符串。(例如, "ACE" 是 "ABCDE" 的一个子序列, 而 "AEC" 不是)。
- $0 \leq s.length, t.length \leq 1000$ 。
- s 和 t 由英文字母组成。

示例:

- **示例 1:**

输入: $s = "rabbbit"$, $t = "rabbit"$

输出: 3

解释: 如下图所示, 有 3 种可以从 s 中得到 "rabbit" 的方案。

rabbbit
rabbbit

解题思路

思路 1：动态规划

1. 划分阶段

按照子序列的结尾位置进行阶段划分。

2. 定义状态

定义状态 $dp[i][j]$ 表示为：以第 $i - 1$ 个字符为结尾的 s 子序列中出现以第 $j - 1$ 个字符为结尾的 t 的个数。

3. 状态转移方程

双重循环遍历字符串 s 和 t ，则状态转移方程为：

- 如果 $s[i - 1] == t[j - 1]$ ，则： $dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j]$ 。即 $dp[i][j]$ 来源于两部分：
 - 使用 $s[i - 1]$ 匹配 $t[j - 1]$ ，则 $dp[i][j]$ 取源于以 $i - 2$ 为结尾的 s 子序列中出现以 $j - 2$ 为结尾的 t 的个数，即 $dp[i - 1][j - 1]$ 。
 - 不使用 $s[i - 1]$ 匹配 $t[j - 1]$ ，则 $dp[i][j]$ 取源于以 $i - 2$ 为结尾的 s 子序列中出现以 $j - 1$ 为结尾的 t 的个数，即 $dp[i - 1][j]$ 。
- 如果 $s[i - 1] != t[j - 1]$ ，那么肯定不能用 $s[i - 1]$ 匹配 $t[j - 1]$ ，则 $dp[i][j]$ 取源于 $dp[i - 1][j]$ 。

4. 初始条件

- $dp[i][0]$ 表示以 $i - 1$ 为结尾的 s 子序列中出现空字符串的个数。把 s 中的元素全删除，出现空字符串的个数就是 1，则 $dp[i][0] = 1$ 。
- $dp[0][j]$ 表示空字符串中出现以 $j - 1$ 结尾的 t 的个数，空字符串无论怎么变都不会变成 t ，则 $dp[0][j] = 0$
- $dp[0][0]$ 表示空字符串中出现空字符串的个数，这个应该是 1，即 $dp[0][0] = 1$ 。

5. 最终结果

根据我们之前定义的状态， $dp[i][j]$ 表示为：以第 $i - 1$ 个字符为结尾的 s 子序列中出现以第 $j - 1$ 个字符为结尾的 t 的个数。则最终结果为 $dp[\text{size}_s][\text{size}_t]$ ，将其返回即可。

思路 1：动态规划代码

```
class Solution:
    def numDistinct(self, s: str, t: str) -> int:
        size_s = len(s)
        size_t = len(t)
        dp = [[0 for _ in range(size_t + 1)] for _ in range(size_s + 1)]
        for i in range(size_s):
            dp[i][0] = 1
        for i in range(1, size_s + 1):
            for j in range(1, size_t + 1):
                if s[i - 1] == t[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j]
                else:
                    dp[i][j] = dp[i - 1][j]
        return dp[size_s][size_t]
```

思路 1：复杂度分析

- 时间复杂度： $O(n^2)$ 。两重循环遍历的时间复杂度是 $O(n^2)$ ，所以总的时间复杂度为 $O(n^2)$ 。
- 空间复杂度： $O(n^2)$ 。用到了二维数组保存状态，所以总体空间复杂度为 $O(n^2)$ 。

0116. 填充每个节点的下一个右侧节点指针

- 标签：树、深度优先搜索、广度优先搜索、链表、二叉树
- 难度：中等

题目链接

- [0116. 填充每个节点的下一个右侧节点指针 - 力扣](#)

题目大意

描述：给定一个完美二叉树，所有叶子节点都在同一层，每个父节点都有两个子节点。完美二叉树结构如下：

```

struct Node {
    int val;
    Node *left;
    Node *right;
    Node *next;
}

```

要求: 填充每个 `next` 指针，使得这个指针指向下一个右侧节点。如果找不到下一个右侧节点，则将 `next` 置为 `None`。

说明:

- 初始状态下，所有 `next` 指针都被设置为 `None`。
- 树中节点的数量在 $[0, 2^{12} - 1]$ 范围内。
- $-1000 \leq node.val \leq 1000$ 。
- 进阶：
 - 只能使用常量级额外空间。
 - 使用递归解题也符合要求，本题中递归程序占用的栈空间不算做额外的空间复杂度。

示例:

- **示例 1:**

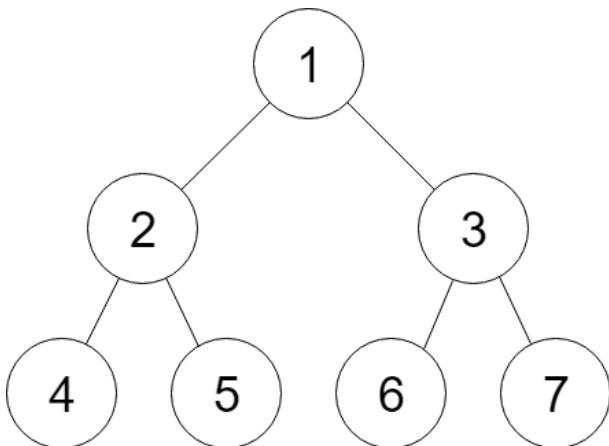


Figure A

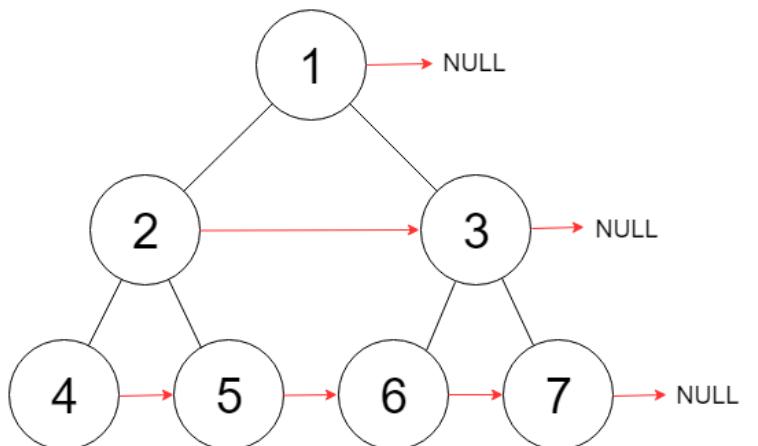


Figure B

输入: `root = [1,2,3,4,5,6,7]`

输出: `[1,#,2,3,#,4,5,6,7,#]`

解释: 给定二叉树如图 A 所示，你的函数应该填充它的每个 `next` 指针，以指向其下一个右侧节点，如图 B 所示。)

- **示例 2:**

输入: `root = []`

输出: `[]`

解题思路

思路 1：层次遍历

在层次遍历的过程中，依次取出每一层的节点，并进行连接。然后再扩展下一层节点。

思路 1：代码

```
import collections

class Solution:
    def connect(self, root: 'Node') -> 'Node':
        if not root:
            return root
        queue = collections.deque()
        queue.append(root)
        while queue:
            size = len(queue)
            for i in range(size):
                node = queue.popleft()
                if i < size - 1:
                    node.next = queue[0]

                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)
        return root
```

思路 1：复杂度分析

- **时间复杂度:** $O(n)$ ，其中 n 为树中的节点数量。
- **空间复杂度:** $O(1)$ 。

0117. 填充每个节点的下一个右侧节点指针 II

- 标签：树、深度优先搜索、广度优先搜索、链表、二叉树
- 难度：中等

题目链接

- [0117. 填充每个节点的下一个右侧节点指针 II - 力扣](#)

题目大意

描述：给定一个二叉树。二叉树结构如下：

```
struct Node {  
    int val;  
    Node *left;  
    Node *right;  
    Node *next;  
}
```

要求：填充每个 `next` 指针，使得这个指针指向下一个右侧节点。如果找不到下一个右侧节点，则将 `next` 置为 `None`。

说明：

- 初始状态下，所有 `next` 指针都被设置为 `None`。
- 树中节点的数量在 $[0, 6000]$ 范围内。
- $-100 \leq \text{Node.val} \leq 100$ 。
- 进阶：
 - 只能使用常量级额外空间。
 - 使用递归解题也符合要求，本题中递归程序占用的栈空间不算做额外的空间复杂度。

示例：

- **示例 1：**

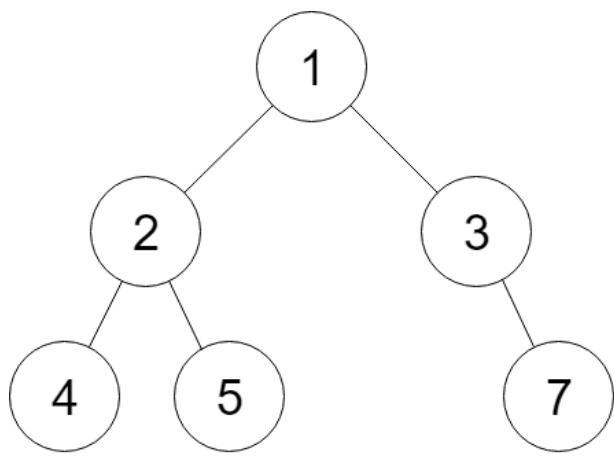


Figure A

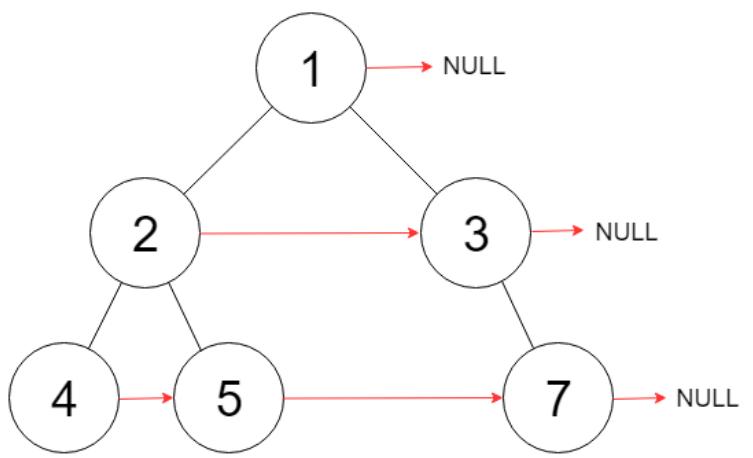


Figure B

输入: `root = [1,2,3,4,5,null,7]`

输出: `[1,#,2,3,#,4,5,7,#]`

解释: 给定二叉树如图 A 所示, 你的函数应该填充它的每个 `next` 指针, 以指向其下一个右侧节点, 如图 B 所示。)

- 示例 2:

输入: `root = []`

输出: `[]`

解题思路

思路 1：层次遍历

在层次遍历的过程中, 依次取出每一层的节点, 并进行连接。然后再扩展下一层节点。

思路 1：代码

```
import collections

class Solution:
    def connect(self, root: 'Node') -> 'Node':
        if not root:
            return root
        queue = collections.deque()
        queue.append(root)
        while queue:
            size = len(queue)
            for i in range(size):
                node = queue.popleft()
                if i < size - 1:
                    node.next = queue[0]

                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)
        return root
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ ，其中 n 为树中的节点数量。
- 空间复杂度： $O(1)$ 。

0118. 杨辉三角

- 标签：数组、动态规划
- 难度：简单

题目链接

- [0118. 杨辉三角 - 力扣](#)

题目大意

描述：给定一个整数 $numRows$ 。

要求：生成前 $numRows$ 行的杨辉三角。

说明：

- $1 \leq numRows \leq 30$ 。

示例：

- **示例 1：**

```
输入: numRows = 5
输出: [[1], [1,1], [1,2,1], [1,3,3,1], [1,4,6,4,1]]
即
[
    [1],
    [1,1],
    [1,2,1],
    [1,3,3,1],
    [1,4,6,4,1]
]
```

- **示例 2：**

```
输入: numRows = 1
输出: [[1]]
```

解题思路

思路 1：动态规划

1. 划分阶段

按照行数进行阶段划分。

2. 定义状态

定义状态 $dp[i][j]$ 为：杨辉三角第 i 行、第 j 列位置上的值。

3. 状态转移方程

根据观察，很容易得出状态转移方程为： $dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j]$ ，此时 $i > 0$, $j > 0$ 。

4. 初始条件

- 每一行第一列都为 1，即 $dp[i][0] = 1$ 。
- 每一行最后一列都为 1，即 $dp[i][i] = 1$ 。

5. 最终结果

根据题意和状态定义，我们将每行结果存入答案数组中，将其返回。

思路 1：动态规划代码

```
class Solution:
    def generate(self, numRows: int) -> List[List[int]]:
        dp = [[0] * i for i in range(1, numRows + 1)]

        for i in range(numRows):
            dp[i][0] = 1
            dp[i][i] = 1

        res = []
        for i in range(numRows):
            for j in range(i):
                if i != 0 and j != 0:
                    dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j]
            res.append(dp[i])

        return res
```

思路 1：复杂度分析

- **时间复杂度：** $O(n^2)$ 。初始条件赋值的时间复杂度为 $O(n)$ ，两重循环遍历的时间复杂度为 $O(n^2)$ ，所以总的时间复杂度为 $O(n^2)$ 。
- **空间复杂度：** $O(n^2)$ 。用到了二维数组保存状态，所以总体空间复杂度为 $O(n^2)$ 。

思路 2：动态规划 + 滚动数组优化

因为 $dp[i][j]$ 仅依赖于上一行（第 $i - 1$ 行）的 $dp[i - 1][j - 1]$ 和 $dp[i - 1][j]$ ，所以我们没必要保存所有阶段的状态，只需要保存上一阶段的所有状态和当前阶段的所有状态就可以了，这样使用两个一维数组分别保存相邻两个阶段的所有状态就可以实现了。

其实我们还可以进一步进行优化，即我们只需要使用一个一维数组保存上一阶段的所有状态。

定义 $dp[j]$ 为杨辉三角第 i 行第 j 列位置上的值。则第 $i + 1$ 行、第 j 列的值可以通过 $dp[j] + dp[j - 1]$ 所得到。

这样我们就可以对这个一维数组保存的「上一阶段的所有状态值」进行逐一计算，从而获取「当前阶段的所有状态值」。

需要注意：本题在计算的时候需要从右向左依次遍历每个元素位置，这是因为如果从左向右遍历，如果当前元素 $dp[j]$ 已经更新为当前阶段第 j 列位置的状态值之后，右侧 $dp[j + 1]$ 想要更新的话，需要的是上一阶段的状态值 $dp[j]$ ，而此时 $dp[j]$ 已经更新了，会破坏当前阶段的状态值。而如果用从右向左的顺序，则不会出现该问题。

思路 2：动态规划 + 滚动数组优化代码

```
class Solution:
    def generate(self, numRows: int) -> List[List[int]]:
        dp = [1 for _ in range(numRows + 1)]
        res = []

        for i in range(numRows):
            for j in range(i - 1, -1, -1):
                if i != 0 and j != 0:
                    dp[j] = dp[j - 1] + dp[j]
            res.append(dp[:i + 1])

        return res
```

思路 2：复杂度分析

- **时间复杂度：** $O(n^2)$ 。两重循环遍历的时间复杂度为 $O(n^2)$ 。
- **空间复杂度：** $O(n)$ 。不考虑最终返回值的空间占用，则总的空间复杂度为 $O(n)$ 。

0119. 杨辉三角 II

- 标签：数组、动态规划
- 难度：简单

题目链接

- [0119. 杨辉三角 II - 力扣](#)

题目大意

描述：给定一个非负整数 $rowIndex$ 。

要求：返回杨辉三角的第 $rowIndex$ 行。

说明：

- $0 \leq rowIndex \leq 33$ 。
- 要求使用 $O(k)$ 的空间复杂度。

示例：

- **示例 1：**

输入: `rowIndex = 3`
输出: `[1, 3, 3, 1]`

解题思路

思路 1：动态规划

因为这道题是从 0 行开始计算，则可以先将 $rowIndex$ 加 1，计算出总共的行数，即 $numRows = rowIndex + 1$ 。

1. 划分阶段

按照行数进行阶段划分。

2. 定义状态

定义状态 $dp[i][j]$ 为：杨辉三角第 i 行、第 j 列位置上的值。

3. 状态转移方程

根据观察，很容易得出状态转移方程为： $dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j]$ ，此时 $i > 0$, $j > 0$ 。

4. 初始条件

- 每一行第一列都为 1，即 $dp[i][0] = 1$ 。
- 每一行最后一列都为 1，即 $dp[i][i] = 1$ 。

5. 最终结果

根据题意和状态定义，将 dp 最后一行返回。

思路 1：代码

```
class Solution:
    defgetRow(self, rowIndex: int) -> List[int]:
        # 本题从 0 行开始计算
        numRows = rowIndex + 1

        dp = [[0] * i for i in range(1, numRows + 1)]

        for i in range(numRows):
            dp[i][0] = 1
            dp[i][i] = 1

        for i in range(numRows):
            for j in range(i):
                if i != 0 and j != 0:
                    dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j]

        return dp[-1]
```

思路 1：复杂度分析

- 时间复杂度：** $O(n^2)$ 。初始条件赋值的时间复杂度为 $O(n)$ ，两重循环遍历的时间复杂度为 $O(n^2)$ ，所以总的时间复杂度为 $O(n^2)$ 。
- 空间复杂度：** $O(n^2)$ 。用到了二维数组保存状态，所以总体空间复杂度为 $O(n^2)$ 。

思路 2：动态规划 + 滚动数组优化

因为 $dp[i][j]$ 仅依赖于上一行（第 $i - 1$ 行）的 $dp[i - 1][j - 1]$ 和 $dp[i - 1][j]$ ，所以我们没必要保存所有阶段的状态，只需要保存上一阶段的所有状态和当前阶段的所有状态就可以了，这样使用两个一维数组分别保存相邻两个阶段的所有状态就可以实现了。

其实我们还可以进一步进行优化，即我们只需要使用一个一维数组保存上一阶段的所有状态。

定义 $dp[j]$ 为杨辉三角第 i 行第 j 列位置上的值。则第 $i + 1$ 行、第 j 列的值可以通过 $dp[j] + dp[j - 1]$ 所得到。

这样我们就可以对这个一维数组保存的「上一阶段的所有状态值」进行逐一计算，从而获取「当前阶段的所有状态值」。

需要注意：本题在计算的时候需要从右向左依次遍历每个元素位置，这是因为如果从左向右遍历，如果当前元素 $dp[j]$ 已经更新为当前阶段第 j 列位置的状态值之后，右侧 $dp[j + 1]$ 想要更新的话，需要的是上一阶段的状态值 $dp[j]$ ，而此时 $dp[j]$ 已经更新了，会破坏当前阶段的状态值。而是用从左向左的顺序，则不会出现该问题。

思路 2：动态规划 + 滚动数组优化代码

```
class Solution:
    def.getRow(self, rowIndex: int) -> List[int]:
        # 本题从 0 行开始计算
        numRows = rowIndex + 1

        dp = [1 for _ in range(numRows)]

        for i in range(numRows):
            for j in range(i - 1, -1, -1):
                if i != 0 and j != 0:
                    dp[j] = dp[j - 1] + dp[j]

        return dp
```

思路 2：复杂度分析

- **时间复杂度：** $O(n^2)$ 。两重循环遍历的时间复杂度为 $O(n^2)$ 。
- **空间复杂度：** $O(n)$ 。不考虑最终返回值的空间占用，则总的空间复杂度为 $O(n)$ 。

0120. 三角形最小路径和

- 标签：数组、动态规划
- 难度：中等

题目链接

- [0120. 三角形最小路径和 - 力扣](#)

题目大意

描述：给定一个代表三角形的二维数组 triangle , triangle 共有 n 行，其中第 i 行（从 0 开始编号）包含了 $i + 1$ 个数。

我们每一步只能从当前位置移动到下一行中相邻的节点上。也就是说，如果正位于第 i 行第 j 列的节点，那么下一步可以移动到第 $i + 1$ 行第 j 列的位置上，或者第 $i + 1$ 行，第 $j + 1$ 列的位置上。

要求：找出自顶向下的最小路径和。

说明：

- $1 \leq \text{triangle.length} \leq 200$ 。
- $\text{triangle}[0].length == 1$ 。
- $\text{triangle}[i].length == \text{triangle}[i - 1].length + 1$ 。
- $-10^4 \leq \text{triangle}[i][j] \leq 10^4$ 。

示例：

- **示例 1：**

输入: `triangle = [[2], [3, 4], [6, 5, 7], [4, 1, 8, 3]]`

输出: `11`

解释: 如下面简图所示:

```
 2  
 3 4  
6 5 7  
4 1 8 3
```

自顶向下的最小路径和为 `11` (即, `2 + 3 + 5 + 1 = 11`)。

- 示例 2：

输入: `triangle = [[-10]]`

输出: `-10`

解题思路

思路 1：动态规划

1. 划分阶段

按照行数进行阶段划分。

2. 定义状态

定义状态 $dp[i][j]$ 表示为：从顶部走到第 i 行（从 0 开始编号）、第 j 列的位置时的最小路径和。

3. 状态转移方程

由于每一步只能从当前位置移动到下一行中相邻的节点上，想要移动到第 i 行、第 j 列的位置，那么上一步只能在第 $i - 1$ 行、第 $j - 1$ 列的位置上，或者在第 $i - 1$ 行、第 j 列的位置上。则状态转移方程为：

$dp[i][j] = \min(dp[i - 1][j - 1], dp[i - 1][j]) + triangle[i][j]$ 。其中 $triangle[i][j]$ 表示第 i 行、第 j 列位置上的元素值。

4. 初始条件

在第 0 行、第 j 列时，最小路径和为 $triangle[0][0]$ ，即 $dp[0][0] = triangle[0][0]$ 。

5. 最终结果

根据我们之前定义的状态， $dp[i][j]$ 表示为：从顶部走到第 i 行（从 0 开始编号）、第 j 列的位置时的最小路径和。为了计算出最小路径和，则需要再遍历一遍 $dp[size - 1]$ 行的每一列，求出最小值即为最终结果。

思路 1：动态规划代码

```
class Solution:
    def minimumTotal(self, triangle: List[List[int]]) -> int:
        size = len(triangle)
        dp = [[0 for _ in range(size)] for _ in range(size)]
        dp[0][0] = triangle[0][0]

        for i in range(1, size):
            dp[i][0] = dp[i - 1][0] + triangle[i][0]
            for j in range(1, i):
                dp[i][j] = min(dp[i - 1][j - 1], dp[i - 1][j]) + triangle[i][j]
            dp[i][i] = dp[i - 1][i - 1] + triangle[i][i]

        return min(dp[size - 1])
```

思路 1：复杂度分析

- **时间复杂度**: $O(n^2)$ 。两重循环遍历的时间复杂度是 $O(n^2)$, 最后求最小值的时间复杂度是 $O(n)$, 所以总体时间复杂度为 $O(n^2)$ 。
- **空间复杂度**: $O(n^2)$ 。用到了二维数组保存状态, 所以总体空间复杂度为 $O(n^2)$ 。

0121. 买卖股票的最佳时机

- 标签: 数组、动态规划
- 难度: 简单

题目链接

- [0121. 买卖股票的最佳时机 - 力扣](#)

题目大意

描述: 给定一个数组 `prices` , 它的第 `i` 个元素 `prices[i]` 表示一支给定股票第 `i` 天的价格。只能选择某一天买入这只股票，并选择在未来的某一个不同的日子卖出该股票。

要求: 计算出能获取的最大利润。如果你不能获取任何利润，返回 0。

说明:

- $1 \leq \text{prices.length} \leq 10^5$ 。
- $0 \leq \text{prices}[i] \leq 10^4$ 。

示例:

- 示例 1:

输入: [7, 1, 5, 3, 6, 4]

输出: 5

解释: 在第 2 天 (股票价格 = 1) 的时候买入, 在第 5 天 (股票价格 = 6) 的时候卖出, 最大利润 = 6 - 1 = 5。

注意利润不能是 7 - 1 = 6, 因为卖出价格需要大于买入价格; 同时, 你不能在买入前卖出股票。

- 示例 2:

输入: prices = [7, 6, 4, 3, 1]

输出: 0

解释: 在这种情况下, 没有交易完成, 所以最大利润为 0。

解题思路

最简单的思路当然是两重循环暴力枚举, 寻找不同天数下的最大利润。但更好的做法是进行一次遍历, 递推求解。

思路 1: 递推

1. 设置两个变量 `minprice` (用来记录买入的最小值)、`maxprofit` (用来记录可获取的最大利润)。
2. 从左到右进行遍历数组 `prices`。
3. 如果遇到当前价格比 `minprice` 还要小的, 就更新 `minprice`。
4. 如果遇到当前价格大于或者等于 `minprice`, 则判断一下以当前价格卖出的话能卖多少, 如果比 `maxprofit` 还要大, 就更新 `maxprofit`。
5. 最后输出 `maxprofit`。

思路 1：代码

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        minprice = 10010
        maxprofit = 0
        for price in prices:
            if price < minprice:
                minprice = price
            elif price - minprice > maxprofit:
                maxprofit = price - minprice
        return maxprofit
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ ，其中 n 是数组 `prices` 的元素个数。
- 空间复杂度： $O(1)$ 。

0122. 买卖股票的最佳时机 II

- 标签：贪心、数组、动态规划
- 难度：中等

题目链接

- [0122. 买卖股票的最佳时机 II - 力扣](#)

题目大意

描述：给定一个整数数组 `prices`，其中 `prices[i]` 表示某支股票第 i 天的价格。在每一天，你可以决定是否购买 / 出售股票。你在任何时候最多只能持有一股股票。你也可以先购买，然后在同一天出售。

要求：计算出能获取的最大利润。

说明：

- $1 \leq \text{prices.length} \leq 3 * 10^4$ 。

- $0 \leq prices[i] \leq 10^4$ 。

示例：

- 示例 1：

输入：prices = [7, 1, 5, 3, 6, 4]

输出：7

解释：在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润：随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，这笔交易所能获得总利润为 $4 + 3 = 7$ 。

- 示例 2：

输入：prices = [1, 2, 3, 4, 5]

输出：4

解释：在第 1 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润总利润为 4。

解题思路

思路 1：贪心算法

股票买卖获取利润主要是看差价，必然是低点买入，高点卖出才会赚钱。而要想获取最大利润，就要在跌入谷底的时候买入，在涨到波峰的时候卖出利益才会最大化。所以我们购买股票的策略变为了：

1. 连续跌的时候不买。
2. 跌到最低点买入。
3. 涨到最高点卖出。

在这种策略下，只要计算波峰和谷底的差值即可。而波峰和谷底的差值可以通过两两相减所得的差值来累加计算。

思路 1：代码

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        ans = 0
        for i in range(1, len(prices)):
            ans += max(0, prices[i]-prices[i-1])
        return ans
```

思路 1：复杂度分析

- **时间复杂度**: $O(n)$, 其中 n 是数组 `prices` 的元素个数。
- **空间复杂度**: $O(1)$ 。