

## 704. 二分查找

给定一个  $n$  个元素有序的（升序）整型数组 `nums` 和一个目标值 `target`，写一个函数搜索 `nums` 中的 `target`，如果目标值存在返回下标，否则返回 `-1`。

示例 1：

```
输入: nums = [-1,0,3,5,9,12], target = 9
输出: 4
解释: 9 出现在 nums 中并且下标为 4
```

示例 2：

```
输入: nums = [-1,0,3,5,9,12], target = 2
输出: -1
解释: 2 不在 nums 中因此返回 -1
```

提示：

1. 你可以假设 `nums` 中的所有元素是不重复的。
2.  $n$  将在  $[1, 10000]$  之间。
3. `nums` 的每个元素都将在  $[-9999, 9999]$  之间。

```
In [10]: from typing import List

class Solution:
    # 我的尝试1: for 循环所有
    def search_for_all(self, nums: List[int], target: int) -> int:
        output = -1

        # search target in nums, if target exists, then return its index
        if target in nums:
            output = 0
            for num in nums:
                if num == target:
                    return output
                else:
                    output += 1

        # Not in nums, return -1
        else:
            return output

    # 我的尝试2：利用二分查找，降低复杂度
    # 每次取查找范围的中点
    def search(self, nums: List[int], target: int) -> int:
        output = -1
        left = 0
        right = len(nums)
        if target in nums:
            while left <= right:
                mid_i = left + (right - left) // 2
                mid_num = nums[mid_i]
                if mid_num == target:
                    return mid_i
                elif mid_num > target:
                    right = mid_i - 1
                elif mid_num < target:
                    left = mid_i + 1

        # Not in nums, return -1
        else:
            return output

    # 去掉target in nums的loop
    def search(self, nums: List[int], target: int) -> int:
        left = 0
        right = len(nums) - 1 # 修正: 索引范围是 [0, len(nums) - 1]
        while left <= right:
            mid_i = left + (right - left) // 2
            mid_num = nums[mid_i]
            if mid_num == target:
                return mid_i # 找到目标值, 直接返回索引
            elif mid_num > target:
                right = mid_i - 1 # 目标值在左侧, 缩小右边界
            else:
                left = mid_i + 1 # 目标值在右侧, 缩小左边界

        return -1 # 未找到目标值, 返回 -1
```

```
In [11]: def run_tests():
    solution = Solution() # 创建类的实例

    test_cases = [
```

```

        {"nums": [-1, 0, 3, 5, 9, 12], "target": 9, "expected": 4},
        {"nums": [-1, 0, 3, 5, 9, 12], "target": 2, "expected": -1}
    ]

# 测试 search 方法
print("Testing search method:")
for i, test in enumerate(test_cases):
    nums = test["nums"]
    target = test["target"]
    expected = test["expected"]
    result = solution.search(nums, target)
    print(f"Test case {i + 1}: {'PASSED' if result == expected else 'FAILED'}")
    print(f"Input: nums={nums}, target={target}")
    print(f"Expected Output: {expected}, Actual Output: {result}\n")

# 测试 search_for_all 方法
print("Testing search_for_all method:")
for i, test in enumerate(test_cases):
    nums = test["nums"]
    target = test["target"]
    expected = test["expected"]
    result = solution.search_for_all(nums, target)
    print(f"Test case {i + 1}: {'PASSED' if result == expected else 'FAILED'}")
    print(f"Input: nums={nums}, target={target}")
    print(f"Expected Output: {expected}, Actual Output: {result}\n")

# 运行测试
run_tests()

```

```

Testing search method:
Test case 1: PASSED
Input: nums=[-1, 0, 3, 5, 9, 12], target=9
Expected Output: 4, Actual Output: 4

Test case 2: PASSED
Input: nums=[-1, 0, 3, 5, 9, 12], target=2
Expected Output: -1, Actual Output: -1

Testing search_for_all method:
Test case 1: PASSED
Input: nums=[-1, 0, 3, 5, 9, 12], target=9
Expected Output: 4, Actual Output: 4

Test case 2: PASSED
Input: nums=[-1, 0, 3, 5, 9, 12], target=2
Expected Output: -1, Actual Output: -1

```

## 703反思

二分查找法： - 直接把边界定义为两个变量 - 更新边界 - 更新的时候跳过中间元素 left/right = mid\_i +/- 1 - 考虑左闭右闭更直接

---

## 27. 移除元素

给你一个数组 `nums` 和一个值 `val`, 你需要原地移除所有数值等于 `val` 的元素。元素的顺序可能发生改变。然后返回 `nums` 中与 `val` 不同的元素的数量。

假设 `nums` 中不等于 `val` 的元素数量为 `k`, 要通过此题, 您需要执行以下操作:

更改 `nums` 数组, 使 `nums` 的前 `k` 个元素包含不等于 `val` 的元素。`nums` 的其余元素和 `nums` 的大小并不重要。返回 `k`。用户评测:

评测机将使用以下代码测试您的解决方案:

```
int[] nums = [...]; // 输入数组 int val = ...; // 要移除的值 int[] expectedNums = [...]; // 长度正确的预期答案。// 它以不等于 val 的值排序。
```

```
int k = removeElement(nums, val); // 调用你的实现
```

```
assert k == expectedNums.length; sort(nums, 0, k); // 排序 nums 的前 k 个元素 for (int i = 0; i < actualLength; i++) { assert nums[i] == expectedNums[i]; } 如果所有的断言都通过, 你的解决方案将会通过。
```

示例 1:

输入: `nums` = [3,2,2,3], `val` = 3 输出: 2, `nums` = [2,2,,] 解释: 你的函数应该返回 `k` = 2, 并且 `nums` 中的前两个元素均为 2。你在返回的 `k` 个元素之外留下了什么并不重要 (因此它们并不计入评测)。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: 你的函数应该返回 `k` = 5, 并且 `nums` 中的前五个元素为 0,0,1,3,4。注意这五个元素可以任意顺序返回。你在返回的 `k` 个元素之外留下了什么并不重要 (因此它们并不计入评测)。

提示:

`0 <= nums.length <= 100` `0 <= nums[i] <= 50` `0 <= val <= 100`

### Problem: 27. Remove Element

#### Problem Statement

Given an integer array `nums` and an integer `val`, remove all occurrences of `val` in `nums` **in-place**. The order of the elements may be changed. Then return the number of elements in `nums` which are not equal to `val`.

Consider the number of elements in `nums` which are not equal to `val` be `k`. To get accepted, you need to do the following things:

1. Change the array `nums` such that the **first `k` elements** of `nums` contain the elements which are not equal to `val`.
  2. The remaining elements of `nums` are not important, as well as the size of `nums`.
  3. Return `k`.
- 

### Custom Judge:

The judge will test your solution with the following code:

```
int[] nums = [...]; // Input array
int val = ...; // Value to remove
int[] expectedNums = [...]; // The expected answer with correct length.
                           // It is sorted with no values equaling val.

int k = removeElement(nums, val); // Calls your implementation

assert k == expectedNums.length;
sort(nums, 0, k); // Sort the first k elements of nums
for (int i = 0; i < actualLength; i++) {
    assert nums[i] == expectedNums[i];
}
```

If all assertions pass, then your solution will be accepted.

---

## Examples

### Example 1:

**Input:**

```
plaintext
nums = [3,2,2,3], val = 3
```

**Output:**

```
plaintext
2, nums = [2,2,_,_]
```

**Explanation:**

Your function should return `k = 2`, with the first two elements of `nums` being `2`. It does not matter what you leave beyond the returned `k` (hence they are underscores).

---

### Example 2:

**Input:**

```
plaintext
nums = [0,1,2,2,3,0,4,2], val = 2
```

**Output:**

```
plaintext
5, nums = [0,1,4,0,3,_,_,_]
```

**Explanation:**

Your function should return `k = 5`, with the first five elements of `nums` containing `0, 0, 1, 3, and 4`. Note that the five elements can be returned in any order.

It does not matter what you leave beyond the returned `k` (hence they are underscores).

---

## Constraints:

- $(0 \leq \text{len}(\text{nums}) \leq 100)$
  - $(0 \leq \text{len}(\text{nums})[\text{i}] \leq 50)$
  - $(0 \leq \text{len}(\text{val}) \leq 100)$
- 

```
In [13]: class Solution:
    # 遍历方法
    def removeElement(self, nums: List[int], val: int) -> int:

        current_i = 0
        k = 0
        for num in nums:
            if num != val:
                nums[k] = num #直接覆盖, 因为循环不会往回看了
```

```

        k += 1
        current_i += 1
    else:
        current_i += 1
    return k

# 快慢指针 fast and slow
# 通过一个快指针和慢指针在一个for循环下完成两个for循环的工作。

# slow 就是 k
def removeElement(self, nums: List[int], val: int) -> int:
    current_i = 0
    k = 0
    fast = 0
    slow = 0
    size = len(nums)

    while fast < size:
        if nums[fast] != val:
            # slow 不会更新那么频繁，满足 != val 才更新
            nums[slow] = nums[fast]
            slow += 1
        # 而 fast一直在更新，只要不等于
        fast += 1
    return slow

```

```

In [14]: def test_remove_element():
solution = Solution()

# 测试用例列表
test_cases = [
    {"nums": [3, 2, 2, 3], "val": 3, "expected_k": 2, "expected_nums": [2, 2]},
    {"nums": [0, 1, 2, 2, 3, 0, 4, 2], "val": 2, "expected_k": 5, "expected_nums": [0, 1, 3, 0, 4]},
    {"nums": [], "val": 1, "expected_k": 0, "expected_nums": []},
    {"nums": [4, 5], "val": 5, "expected_k": 1, "expected_nums": [4]},
    {"nums": [4, 4, 4], "val": 4, "expected_k": 0, "expected_nums": []},
]

print("Testing the traverse method:")
for i, case in enumerate(test_cases):
    nums = case["nums"][:]
    val = case["val"]
    expected_k = case["expected_k"]
    expected_nums = case["expected_nums"]

    # 调用第一种实现方法
    result_k = solution.removeElement(nums, val)
    assert result_k == expected_k, f"Test case {i+1} failed: k mismatch"
    assert sorted(nums[:result_k]) == sorted(expected_nums), f"Test case {i+1} failed: nums mismatch"
    print(f"Test case {i+1} passed!")

print("\nTesting the fast and slow pointer method:")
for i, case in enumerate(test_cases):
    nums = case["nums"][:]
    val = case["val"]
    expected_k = case["expected_k"]
    expected_nums = case["expected_nums"]

    # 调用第二种实现方法
    result_k = solution.removeElement(nums, val)
    assert result_k == expected_k, f"Test case {i+1} failed: k mismatch"
    assert sorted(nums[:result_k]) == sorted(expected_nums), f"Test case {i+1} failed: nums mismatch"
    print(f"Test case {i+1} passed!")

# 运行测试
test_remove_element()

Testing the traverse method:
Test case 1 passed!
Test case 2 passed!
Test case 3 passed!
Test case 4 passed!
Test case 5 passed!

Testing the fast and slow pointer method:
Test case 1 passed!
Test case 2 passed!
Test case 3 passed!
Test case 4 passed!
Test case 5 passed!

```

## 977.有序数组的平方

给你一个按 非递减顺序 排序的整数数组 nums，返回 每个数字的平方 组成的新数组，要求也按 非递减顺序 排序。

示例 1：

- 输入： nums = [-4,-1,0,3,10]
- 输出： [0,1,9,16,100]
- 解释： 平方后，数组变为 [16,1,0,9,100]，排序后，数组变为 [0,1,9,16,100]

示例 2：

- 输入： nums = [-7,-3,2,3,11]
- 输出： [4,9,9,49,121]

```
In [ ]: ## 暴力法, 直接平方在sort
直接平方, 复杂度 O(n)
sort: 复杂度 (nlogn)
整体 O(n + nlogn) = O(nlogn)
```

## 左右指针 + 结果指针法

```
In [2]: from typing import List

class Solution:
    def sortedSquares(self, nums: List[int]) -> List[int]:
        left_index = 0
        right_index = len(nums) - 1 # 注意用len要-1才行
        result_index = len(nums) - 1 # 从大到小
        result = [float('inf')] * len(nums)

        while left_index <= right_index:
            left_value_squared = nums[left_index] ** 2
            right_value_squared = nums[right_index] ** 2
            if left_value_squared < right_value_squared:
                result[result_index] = right_value_squared
                right_index -= 1
            elif left_value_squared == right_value_squared:
                result[result_index] = right_value_squared
                right_index -= 1
            else:
                result[result_index] = left_value_squared
                left_index += 1
            result_index -= 1

        return result
```

```
In [ ]: # 性能优化

class Solution:
    def sortedSquares(self, nums: List[int]) -> List[int]:
        # 提前判断最小值正负, 可以避免双指针的遍历过程, 从而提升性能。
        if nums[0] >= 0: # 全为非负数
            return [num ** 2 for num in nums]
        if nums[-1] <= 0: # 全为非正数
            return [num ** 2 for num in reversed(nums)]

        left_index = 0
        right_index = len(nums) - 1 # 注意用len要-1才行
        result_index = len(nums) - 1 # 从大到小

        # inf 略微浪费初始化的时间和内存。可以直接初始化为 [0] * len(nums)
        result = [0] * len(nums)

        while left_index <= right_index:
            left_value_squared = nums[left_index] ** 2
            right_value_squared = nums[right_index] ** 2
            if left_value_squared <= right_value_squared:
                result[result_index] = right_value_squared
                right_index -= 1
            else:
                result[result_index] = left_value_squared
                left_index += 1
            result_index -= 1

        return result
```

## 知识点

```
right_index = len(nums) - 1 # 注意用len要-1才行
result = [float('inf')] * len(nums) # 注意这个用法
```

```
result_index -= 1 # 不要忘了移动结果的指针  
可以分三类讨论，大于小于等于，其中等于其实放到哪类都可以，但是分类讨论容易想清楚
```

```
In [5]: def test_sortedSquares():  
    solution = Solution()  
  
    # 测试用例列表  
    test_cases = [  
        {"nums": [-4, -1, 0, 3, 10], "expected": [0, 1, 9, 16, 100]},  
        {"nums": [-7, -3, 2, 3, 11], "expected": [4, 9, 9, 49, 121]},  
        {"nums": [-1], "expected": [1]},  
        {"nums": [0, 1, 2, 3], "expected": [0, 1, 4, 9]},  
        {"nums": [], "expected": []}, # 边界情况：空数组  
        {"nums": [-5, -3, -1], "expected": [1, 9, 25]} # 全部为负数的数组  
    ]  
  
    # 运行测试  
    for i, case in enumerate(test_cases):  
        nums = case["nums"]  
        expected = case["expected"]  
        result = solution.sortedSquares(nums)  
  
        assert result == expected, f"Test case {i+1} failed: Input {nums}, Expected {expected}, Got {result}"  
        print(f"Test case {i+1} passed!")  
  
    # 运行测试  
    test_sortedSquares()  
  
Test case 1 passed!  
Test case 2 passed!  
Test case 3 passed!  
Test case 4 passed!  
Test case 5 passed!  
Test case 6 passed!
```

## 209. 长度最小的子数组

给定一个含有 n 个正整数的数组和一个正整数 target。

找出该数组中满足其总和大于等于 target 的长度最小的 子数组 [numsl, numsl+1, ..., numsr-1, numsr]， 并返回其长度。如果不存在符合条件的子数组，返回 0。

示例 1：

输入：target = 7, nums = [2,3,1,2,4,3] 输出：2 解释：子数组 [4,3] 是该条件下的长度最小的子数组。示例 2：

输入：target = 4, nums = [1,4,4] 输出：1 示例 3：

输入：target = 11, nums = [1,1,1,1,1,1] 输出：0

提示：

$1 \leq \text{target} \leq 10^9$   $1 \leq \text{nums.length} \leq 10^5$   $1 \leq \text{nums}[i] \leq 10^4$

进阶：

如果你已经实现  $O(n)$  时间复杂度的解法，请尝试设计一个  $O(n \log(n))$  时间复杂度的解法。

理解：

就相当于前指针排除了很多无用的branch target : 100

```
1, 1, 1, 100  
^ * ->
```

```
1, 1, 1, 100  
^ -> *
```

满足条件 103，开始向前移动前指针

```
1, 1, 1, 100  
^ -> *
```

```
1, 1, 1, 100  
^ ->*
```

核心的误区： 考虑太多太长数列的问题

这个数列可能很长，但是如果我们将能找到很小的区间，那么就不用再看后边的了  
所以滑动窗口可以

## 关键点总结

- 滑动窗口的本质：

- 动态调整区间  $[l, r]$  的大小，通过不断扩展和收缩窗口找到满足条件的解。

## 2. 优化搜索：

- 当窗口内的总和已经满足条件时，立即尝试收缩窗口，避免浪费时间继续扩大区间。

## 3. 边界处理：

- 如果 `min_length` 没有被更新过，说明没有满足条件的子数组，返回 `0`。

通过滑动窗口，你只需遍历一次数组，时间复杂度为  $O(n)$ ，适用于长度较大的数组。你的理解是对的，这个方法有效地避免了冗余的计算路径。

```
In [7]: class Solution:
    def minSubArrayLen(self, target: int, nums: List[int]) -> int:
        # 滑动窗口
        l = 0
        r = 0
        sum = 0
        result = float("inf")  # 记录结果list的长度

        for r in range(len(nums)):  # 注意range是从 0 到 len() - 1
            sum += nums[r]

            while sum >= target:  # 条件还不满足
                current_length = r - l + 1  # 更新list长度
                result = min(result, current_length)  # 记录最小的
                sum -= nums[l]  # 因为缩小了，要减去
                l += 1  # 向左移动左指针

            if result == float("inf"):
                return 0
            else:
                return result
```

```
In [8]: def test_minSubArrayLen():
    solution = Solution()

    # 测试用例列表
    test_cases = [
        {"target": 7, "nums": [2, 3, 1, 2, 4, 3], "expected": 2},  # 子数组 [4, 3]
        {"target": 4, "nums": [1, 4, 4], "expected": 1},  # 子数组 [4]
        {"target": 11, "nums": [1, 1, 1, 1, 1, 1, 1], "expected": 0},  # 无子数组满足条件
        {"target": 15, "nums": [1, 2, 3, 4, 5], "expected": 5},  # 子数组 [1, 2, 3, 4, 5]
        {"target": 100, "nums": [50, 50], "expected": 2},  # 子数组 [50, 50]
        {"target": 1, "nums": [], "expected": 0},  # 空数组
        {"target": 1, "nums": [1], "expected": 1},  # 单个元素满足条件
    ]

    # 运行测试
    for i, case in enumerate(test_cases):
        target = case["target"]
        nums = case["nums"]
        expected = case["expected"]

        result = solution.minSubArrayLen(target, nums)
        assert result == expected, f"Test case {i+1} failed: Input target={target}, nums={nums}, Expected {expected}, Got {result}"
        print(f"Test case {i+1} passed!")

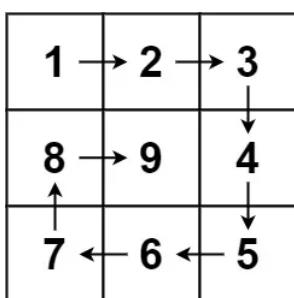
# 运行测试
test_minSubArrayLen()
```

```
Test case 1 passed!
Test case 2 passed!
Test case 3 passed!
Test case 4 passed!
Test case 5 passed!
Test case 6 passed!
Test case 7 passed!
```

## 59.螺旋矩阵II

给定一个正整数  $n$ ，生成一个包含  $1$  到  $n^2$  所有元素，且元素按顺时针顺序螺旋排列的正方形矩阵。

示例：



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

题目建议：本题关键还是在转圈的逻辑，在二分搜索中提到的区间定义，在这里又用上了。

题目链接：<https://leetcode.cn/problems/spiral-matrix-ii/>

文章讲解：<https://programmercarl.com/0059.%E8%9E%BA%E6%97%8B%E7%9F%A9%E9%98%B5II.html>

视频讲解：<https://www.bilibili.com/video/BV1SL4y1N7mV/>

```
In [60]: class Solution:
    def generateMatrix(self, n: int) -> List[List[int]]:
        # 初始化
        # 这种是浅拷贝
        # result = [[0] * n] * n
        # 这种才对
        result = [[0] * n for _ in range(n)]

        # 一定要注意这里的方向，我们最早的方向是右，
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        current_coord = (0, 0)
        visited = set()
        direction_index = 0

        for i in range(1, n*n+1):

            x, y = current_coord # 记住这种用法

            result[x][y] = i
            visited.add(current_coord)

            # 计算下一步坐标
            next_x = x + directions[direction_index][0]
            next_y = y + directions[direction_index][1]
            next_coord = (next_x, next_y)

            if not (0 <= next_x < n and 0 <= next_y < n) or next_coord in visited: # 如果越界，或已经到达，改变方向
                direction_index = (direction_index + 1) % 4
                # 重新计算改变方向后的坐标
                next_x = x + directions[direction_index][0]
                next_y = y + directions[direction_index][1]
                next_coord = (next_x, next_y)

            current_coord = next_coord

        return result
```

## 知识点：在数学和编程中，以 ((row, column)) 的顺序理解坐标

- 编程中的数组坐标表示：
  - 在二维数组或矩阵中，通常用 行优先 的表示方式：
    - 第一维（索引 (0)）表示 行 (row)。
    - 第二维（索引 (1)）表示 列 (column)。
  - 因此，((row, column)) 的顺序是 行在前，列在后。

这意味着在编程中，如果你定义方向向量，应以 ((row, column)) 的顺序理解。

- 数学常规表示（笛卡尔坐标系）：
  - 横轴为 (x)，纵轴为 (y)，通常描述二维平面上的点 ((x, y))。
  - 例如，点 ((1, 2)) 表示 (x=1) 和 (y=2)。

## 在螺旋矩阵中的表示方式

编程中，二维数组的表示一般是以行优先的方式进行，因此：

1. 矩阵表示为 ((row, column))：
  - (row) 是数组的第一维度，代表第几行。
  - (column) 是数组的第二维度，代表第几列。
2. 方向向量的意义：
  - ((0, 1))：表示在当前行保持不变，列索引加 1，即 向右移动。
  - ((1, 0))：表示行索引加 1，列保持不变，即 向下移动。
  - ((0, -1))：表示行保持不变，列索引减 1，即 向左移动。
  - ((-1, 0))：表示行索引减 1，列保持不变，即 向上移动。
3. 螺旋方向顺序：
  - 顺时针方向依次为 右、下、左、上，即：
    - 右：((0, 1))
    - 下：((1, 0))
    - 左：((0, -1))
    - 上：((-1, 0))

## 改为列优先的方式

如果你希望将列为(x), 行为(y), 即按照笛卡尔坐标系的习惯来理解方向, 可以调整方向向量, 同时在访问矩阵时也需要调换索引。

### 1. 方向向量调整:

- 右: ((1, 0)) (列增加, 行不变)
- 下: ((0, 1)) (行增加, 列不变)
- 左: ((-1, 0)) (列减少, 行不变)
- 上: ((0, -1)) (行减少, 列不变)

### 2. 矩阵访问调整:

- 访问矩阵元素时, 将列作为第一维, 行作为第二维。
- 在代码中, 需要将 `result[row][col]` 改为 `result[col][row]`。

```
In [68]: # 笛卡尔坐标

from typing import List

class Solution:
    def generateMatrix(self, n: int) -> List[List[int]]:
        # 初始化矩阵
        result = [[0] * n for _ in range(n)]

        # 按列优先顺序: 右 (列增)、下 (行增)、左 (列减)、上 (行减)
        directions = [(1, 0), (0, 1), (-1, 0), (0, -1)]
        current_coord = (0, 0) # 初始坐标 (x, y)
        direction_index = 0 # 当前方向索引

        for i in range(1, n * n + 1):
            x, y = current_coord # 当前坐标
            result[y][x] = i # 注意行列顺序与列优先一致

            # 计算下一步坐标
            next_x = x + directions[direction_index][0]
            next_y = y + directions[direction_index][1]

            # 如果越界或下一位置已填充, 切换方向
            if not (0 <= next_x < n and 0 <= next_y < n) or result[next_y][next_x] != 0:
                direction_index = (direction_index + 1) % 4 # 顺时针切换方向
                next_x = x + directions[direction_index][0]
                next_y = y + directions[direction_index][1]

            # 更新当前坐标
            current_coord = (next_x, next_y)

        return result
```

```
In [69]: def test_generateMatrix():
    solution = Solution()

    # 测试用例列表
    test_cases = [
        {"n": 3, "expected": [[1, 2, 3], [8, 9, 4], [7, 6, 5]]},
        {"n": 1, "expected": [[1]]},
        {"n": 2, "expected": [[1, 2], [4, 3]]},
        {"n": 4, "expected": [[1, 2, 3, 4], [12, 13, 14, 5], [11, 16, 15, 6], [10, 9, 8, 7]]},
    ]

    # 运行测试
    for i, case in enumerate(test_cases):
        n = case["n"]
        expected = case["expected"]

        result = solution.generateMatrix(n)
        assert result == expected, f"Test case {i+1} failed: Input n={n}, Expected {expected}, Got {result}"
        print(f"Test case {i+1} passed!")

    # 运行测试
    test_generateMatrix()
```

```
Test case 1 passed!
Test case 2 passed!
Test case 3 passed!
Test case 4 passed!
```

```
In [70]: # 方法2, 用offset跟踪, 而不是边界条件
# 按照一圈一圈的方式loop每一行每一列
# 每圈起始点都会向右下角移动(1,1)

class Solution:
    def generateMatrix_offset(self, n: int) -> List[List[int]]:
        nums = [[0] * n for _ in range(n)]
        startx, starty = 0, 0 # 起始点
        loop, mid = n // 2, n // 2 # 迭代次数、n为奇数时, 矩阵的中心点
        count = 1 # 计数

        for offset in range(1, loop + 1) : # 每循环一层偏移量加1, 偏移量从1开始
            for i in range(starty, n - offset) : # 从左至右, 左闭右开
                nums[startx][i] = count
```

```

        count += 1
    for i in range(startx, n - offset) :
        # 从上至下
        nums[i][n - offset] = count
        count += 1
    for i in range(n - offset, starty, -1) :
        # 从右至左
        nums[n - offset][i] = count
        count += 1
    for i in range(n - offset, startx, -1) :
        # 从下至上
        nums[i][starty] = count
        count += 1
    startx += 1           # 更新起始点, 关注到每圈起始点
    starty += 1

if n % 2 != 0 :          # n为奇数时, 填充中心点
    nums[mid][mid] = count
return nums

```

```

In [71]: def test_generateMatrix_offset():
solution = Solution()

# 测试用例列表
test_cases = [
    {"n": 3, "expected": [[1, 2, 3], [8, 9, 4], [7, 6, 5]]},
    {"n": 1, "expected": [[1]]},
    {"n": 2, "expected": [[1, 2], [4, 3]]},
    {"n": 4, "expected": [[1, 2, 3, 4], [12, 13, 14, 5], [11, 16, 15, 6], [10, 9, 8, 7]]},
]

# 运行测试
for i, case in enumerate(test_cases):
    n = case["n"]
    expected = case["expected"]

    result = solution.generateMatrix_offset(n)
    assert result == expected, f"Test case {i+1} failed: Input n={n}, Expected {expected}, Got {result}"
    print(f"Test case {i+1} passed!")

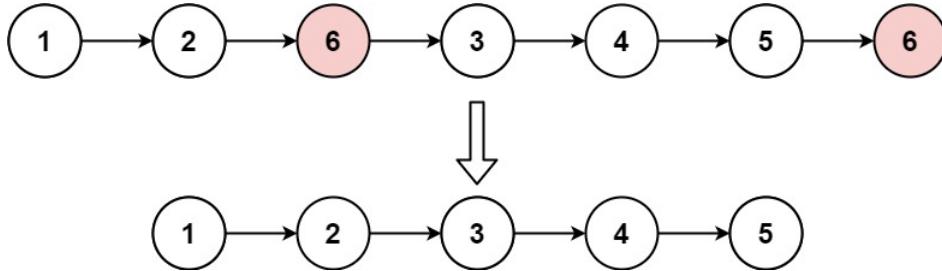
# 运行测试
test_generateMatrix_offset()

```

Test case 1 passed!  
Test case 2 passed!  
Test case 3 passed!  
Test case 4 passed!

### 203. 移除链表元素

给你一个链表的头节点 head 和一个整数 val，请你删除链表中所有满足 `Node.val == val` 的节点，并返回新的头节点。



示例 1：

输入： `head = [1,2,6,3,4,5,6], val = 6` 输出： `[1,2,3,4,5]`

示例 2：

输入： `head = [], val = 1` 输出： `[]`

示例 3：

输入： `head = [7,7,7,7], val = 7` 输出： `[]`

提示： 列表中的节点数目在范围 `[0, 104]` 内  $1 \leq \text{Node.val} \leq 50$   $0 \leq \text{val} \leq 50$

## 两种情况：

如果是头节点：删除方式会不同 头节点：删除当前节点，把 `head` 替换为下一个节点 中间节点：删除当前节点，将前一个节点的指针指向下一个节点

为了让节点的操作统一，采用虚拟头节点法

## 虚拟头节点

先 new 一个节点，作为头节点链接到 `head` 前 遍历结束，`return dummy.next`，也就是原先链表的头节点 注意如果真实的头节点符合条件，它也可能会被删除

```
In [37]: from typing import Optional

# 定义链表节点类
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

    # 用于打印链表
    def __repr__(self):
        result = []
        current = self
        while current:
            result.append(str(current.val))
            current = current.next
        return " -> ".join(result)

class Solution:
    def removeElements(self, head: Optional[ListNode], val: int) -> Optional[ListNode]:
        dummy_head = ListNode(next = head)

        current = dummy_head

        while current.next:
            if current.next.val == val:
                current.next = current.next.next
            else:
                current = current.next

        return dummy_head.next
```

```
In [38]: # 三种思路：分别对应method1,2,3
# 1. 删除节点分为两种情况，删除头节点和非头节点
# 2. 添加一个虚拟头节点，对头节点的删除操作与其他节点一样
# 3. 递归
# method1 分类讨论
class Solution1:
    def removeElements(self, head: ListNode, val: int) -> ListNode:
        while head and head.val == val:
            # 让自head起第一个值不为val的节点作为头节点
            # 退出while循环时，有两种情况
            # 1 head为空(即链表左右节点值均为val，则进入if并return)
            # 2 找到了第一个值不为val的节点(是真正的头节点)，那么之后就开始对该节点之后的非头节点的元素进行遍历处理
            head = head.next
        if head is None:
```

```

        return head
    node = head
    while node.next:
        if node.next.val == val:
            node.next = node.next.next
        else:
            node = node.next
    return head

# method2 虚拟头节点
class Solution2:
    def removeElements(self, head: ListNode, val: int) -> ListNode:
        dummy_node = ListNode(next=head)
        node = dummy_node
        while node.next:
            if node.next.val == val:
                node.next = node.next.next
            else:
                node = node.next
        return dummy_node.next

# method3 递归
class Solution3:
    def removeElements(self, head: ListNode, val: int) -> ListNode:
        if head is None:
            return head
        head.next = self.removeElements(head.next, val)
        # 利用递归快速到达链表尾端，然后从后往前判断并删除重复元素
        return head.next if head.val == val else head
        # 每次递归返回的为当前递归层的head(若其值不为val)或head.next
        # head.next及之后的链表在深层递归中已经做了删除值为val节点的处理,
        # 因此只需要判断当前递归层的head值是否为val, 从而决定head是删是留即可

```

```

In [39]: # 测试代码

# 定义链表节点类
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

    # 用于打印链表
    def __repr__(self):
        result = []
        current = self
        while current:
            result.append(str(current.val))
            current = current.next
        return ">".join(result)

# 辅助函数: 从列表生成链表
def create_linked_list(values):
    if not values: # 空列表返回 None
        return None
    head = ListNode(values[0]) # 创建头节点
    current = head
    for value in values[1:]:
        current.next = ListNode(value) # 创建新节点并链接
        current = current.next # 更新指针
    return head

# 辅助函数: 从链表生成列表
def linked_list_to_list(head):
    result = []
    while head: # 遍历链表直到 None
        result.append(head.val)
        head = head.next
    return result

# 测试函数
def test_removeElements():
    # 测试用例
    test_cases = [
        {"input": ([1, 2, 6, 3, 4, 5, 6], 6), "expected": [1, 2, 3, 4, 5]},
        {"input": ([1], 1), "expected": []},
        {"input": ([1, 1, 1, 1, 1], 1), "expected": []},
        {"input": ([1, 2, 3, 4], 5), "expected": [1, 2, 3, 4]},
        {"input": ([], 1), "expected": []},
    ]

    # 测试三种实现
    solutions = [Solution1(), Solution2(), Solution3()]

    for i, case in enumerate(test_cases):
        input_list, val = case["input"]
        expected = case["expected"]
        print(f"Test case {i + 1}: Input {input_list} with val={val}")

        for j, solution in enumerate(solutions):
            # 创建链表
            head = create_linked_list(input_list)
            # 调用方法
            result = solution.removeElements(head, val)
            result_list = linked_list_to_list(result)
            if result_list == expected:
                print(f"Solution {j + 1} passed")
            else:
                print(f"Solution {j + 1} failed")

```

```

new_head = solution.removeElements(head, val)
# 转换结果为列表
result = linked_list_to_list(new_head)

# 验证结果
assert result == expected, f"Method {j + 1} failed for test case {i + 1}: Expected {expected}, Got {result}"
print(f" Method {j + 1} passed: Output {result}")
print()

# 运行测试
test_removeElements()

Test case 1: Input [1, 2, 6, 3, 4, 5, 6] with val=6
Method 1 passed: Output [1, 2, 3, 4, 5]
Method 2 passed: Output [1, 2, 3, 4, 5]
Method 3 passed: Output [1, 2, 3, 4, 5]

Test case 2: Input [1] with val=1
Method 1 passed: Output []
Method 2 passed: Output []
Method 3 passed: Output []

Test case 3: Input [1, 1, 1, 1] with val=1
Method 1 passed: Output []
Method 2 passed: Output []
Method 3 passed: Output []

Test case 4: Input [1, 2, 3, 4] with val=5
Method 1 passed: Output [1, 2, 3, 4]
Method 2 passed: Output [1, 2, 3, 4]
Method 3 passed: Output [1, 2, 3, 4]

Test case 5: Input [] with val=1
Method 1 passed: Output []
Method 2 passed: Output []
Method 3 passed: Output []

```

```

In [40]: class MyLinkedList:
    def __init__(self, val = 0, next = None):
        self.val = val
        self.next = next

    class MyLinkedList:
        def __init__(self):
            self.dummy_head = MyLinkedList()
            self.size = 0

        def get(self, index: int) -> int:
            if index < 0 or index >= self.size:
                return -1

            else:
                current = self.dummy_head.next

                # 要从头遍历才能找到对象
                for i in range(index):
                    current = current.next
                return current.val

        def addAtHead(self, val: int) -> None:
            self.dummy_head.next = MyLinkedList(val, self.dummy_head.next)
            self.size += 1

        def addAtTail(self, val: int) -> None:
            current = self.dummy_head
            while current.next:
                current = current.next
            current.next = MyLinkedList(val)
            self.size += 1

        def addAtIndex(self, index: int, val: int) -> None:
            if index < 0 or index > self.size:
                return

            current = self.dummy_head
            for i in range(index):
                current = current.next
            current.next = MyLinkedList(val, current.next)
            self.size += 1

        def deleteAtIndex(self, index: int) -> None:
            if index < 0 or index >= self.size:
                return

            current = self.dummy_head
            for i in range(index):
                current = current.next
            current.next = current.next.next
            self.size -= 1

```

```
# Your MyLinkedList object will be instantiated and called as such:
# obj = MyLinkedList()
# param_1 = obj.get(index)
# obj.addAtHead(val)
# obj.addAtTail(val)
# obj.addAtIndex(index,val)
# obj.deleteAtIndex(index)
```

```
In [41]: def test_my_linked_list():
    # 创建一个空的链表
    linked_list = MyLinkedList()

    # 测试用例 1: 空链表的 get 操作
    assert linked_list.get(0) == -1, "Failed: Get from empty list"

    # 测试用例 2: 添加头部
    linked_list.addAtHead(1)
    assert linked_list.get(0) == 1, "Failed: Add at head"

    # 测试用例 3: 添加尾部
    linked_list.addAtTail(2)
    assert linked_list.get(1) == 2, "Failed: Add at tail"

    # 测试用例 4: 添加中间
    linked_list.addAtIndex(1, 3)
    assert linked_list.get(1) == 3, "Failed: Add at index"
    assert linked_list.get(2) == 2, "Failed: Add at index did not maintain order"

    # 测试用例 5: 删除中间
    linked_list.deleteAtIndex(1)
    assert linked_list.get(1) == 2, "Failed: Delete at index"

    # 测试用例 6: 删除头部
    linked_list.deleteAtIndex(0)
    assert linked_list.get(0) == 2, "Failed: Delete head"

    # 测试用例 7: 删除尾部
    linked_list.deleteAtIndex(0)
    assert linked_list.get(0) == -1, "Failed: Delete tail"

    # 测试用例 8: 添加到超出范围的索引
    linked_list.addAtIndex(5, 10) # 无操作
    assert linked_list.get(0) == -1, "Failed: Add at invalid index"

    # 测试用例 9: 添加到空链表的尾部
    linked_list.addAtIndex(0, 4)
    assert linked_list.get(0) == 4, "Failed: Add at valid index in empty list"

    # 测试用例 10: 删除超出范围的索引
    linked_list.deleteAtIndex(10) # 无操作
    assert linked_list.get(0) == 4, "Failed: Delete at invalid index"

    # 测试用例 11: 添加多个节点
    linked_list.addAtHead(3)
    linked_list.addAtTail(5)
    linked_list.addAtTail(6)
    assert linked_list.get(0) == 3, "Failed: Add multiple nodes at head/tail"
    assert linked_list.get(1) == 4, "Failed: Add multiple nodes at head/tail"
    assert linked_list.get(2) == 5, "Failed: Add multiple nodes at head/tail"
    assert linked_list.get(3) == 6, "Failed: Add multiple nodes at head/tail"

    print("All test cases passed!")

# 运行测试
test_my_linked_list()
```

All test cases passed!

```
In [42]: # 双链表法

# 双链表要比单链表性能好一些
# 占用空间会大一些

class MyDoubleLinkedListNode:
    def __init__(self, val=0, prev=None, next=None):
        self.val = val
        self.next = next
        self.prev = prev

class MyDoubleLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def get(self, index: int) -> int:
        if index < 0 or index >= self.size:
            return -1

        if index < self.size // 2:
            current = self.head
            for _ in range(index):
                current = current.next
```

```

# 双链可以实现从前和从后查询，加入了一个判断当前index在中点的前后逻辑
else:
    current = self.tail
    for _ in range(self.size - index - 1):
        current = current.prev

return current.val

def addAtHead(self, val: int) -> None:
    new_node = MyDoubleLinkedListNode(val, None, self.head)
    # 如果不是空list
    if self.head:
        self.head.prev = new_node
    # 如果是空的list
    else:
        self.tail = new_node
    self.head = new_node
    self.size += 1

def addAtTail(self, val: int) -> None:
    new_node = MyDoubleLinkedListNode(val, self.tail, None)
    # 还可以直接实现在尾部加node
    if self.tail:
        self.tail.next = new_node
    else:
        self.head = new_node
    self.tail = new_node
    self.size += 1

def addAtIndex(self, index: int, val: int) -> None:
    # 过滤取件
    if index < 0 or index > self.size:
        return

    # 过滤极端情况
    if index == 0:
        self.addAtHead(val)
    elif index == self.size:
        self.addAtTail(val)

    else:
        # 判断与中间的位置关系
        if index < self.size // 2:
            current = self.head
            for i in range(index - 1):
                current = current.next
        else:
            current = self.tail
            for i in range(self.size - index):
                current = current.prev
        new_node = MyDoubleLinkedListNode(val, current, current.next)
        current.next.prev = new_node
        current.next = new_node
        self.size += 1

def deleteAtIndex(self, index: int) -> None:
    if index < 0 or index >= self.size:
        return

    if index == 0:
        self.head = self.head.next
        if self.head:
            self.head.prev = None
        else:
            self.tail = None
    elif index == self.size - 1:
        self.tail = self.tail.prev
        if self.tail:
            self.tail.next = None
        else:
            self.head = None
    else:
        if index < self.size // 2:
            current = self.head
            for i in range(index):
                current = current.next
        else:
            current = self.tail
            for i in range(self.size - index - 1):
                current = current.prev
        current.next.next = current.next
        current.next.prev = current.prev
        self.size -= 1

```

```

In [43]: def test_my_double_linked_list():
    # 创建一个空的双向链表
    linked_list = MyDoubleLinkedList()

    # 测试用例 1: 空链表的 get 操作
    assert linked_list.get(0) == -1, "Failed: Get from empty list"

    # 测试用例 2: 添加头部
    linked_list.addAtHead(1)

```

```

assert linked_list.get(0) == 1, "Failed: Add at head"

# 测试用例 3: 添加尾部
linked_list.addAtTail(2)
assert linked_list.get(1) == 2, "Failed: Add at tail"

# 测试用例 4: 添加中间
linked_list.addAtIndex(1, 3)
assert linked_list.get(1) == 3, "Failed: Add at index"
assert linked_list.get(2) == 2, "Failed: Add at index did not maintain order"

# 测试用例 5: 删除中间
linked_list.deleteAtIndex(1)
assert linked_list.get(1) == 2, "Failed: Delete at index"

# 测试用例 6: 删除头部
linked_list.deleteAtIndex(0)
assert linked_list.get(0) == 2, "Failed: Delete head"

# 测试用例 7: 删除尾部
linked_list.deleteAtIndex(0)
assert linked_list.get(0) == -1, "Failed: Delete tail"

# 测试用例 8: 添加到超出范围的索引
linked_list.addAtIndex(5, 10) # 无操作
assert linked_list.get(0) == -1, "Failed: Add at invalid index"

# 测试用例 9: 添加到空链表的尾部
linked_list.addAtIndex(0, 4)
assert linked_list.get(0) == 4, "Failed: Add at valid index in empty list"

# 测试用例 10: 删除超出范围的索引
linked_list.deleteAtIndex(10) # 无操作
assert linked_list.get(0) == 4, "Failed: Delete at invalid index"

# 测试用例 11: 添加多个节点
linked_list.addAtHead(3)
linked_list.addAtTail(5)
linked_list.addAtTail(6)
assert linked_list.get(0) == 3, "Failed: Add multiple nodes at head/tail"
assert linked_list.get(1) == 4, "Failed: Add multiple nodes at head/tail"
assert linked_list.get(2) == 5, "Failed: Add multiple nodes at head/tail"
assert linked_list.get(3) == 6, "Failed: Add multiple nodes at head/tail"

print("All test cases passed!")

# 运行测试
test_my_double_linked_list()

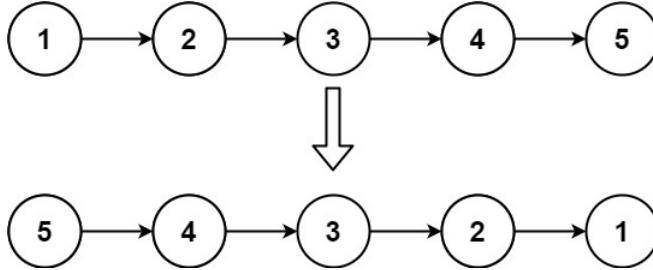
```

All test cases passed!

## 206. 反转链表

给你单链表的头节点 head，请你反转链表，并返回反转后的链表。

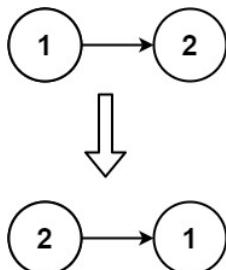
示例 1:



输入: head = [1,2,3,4,5]

输出: [5,4,3,2,1]

示例 2:



输入: head = [1,2]

输出: [2,1]

示例 3:

输入: head = []

输出: []

进阶: 链表可以选用迭代或递归方式完成反转。你能否用两种方法解决这道题?

```
In [ ]: # Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def reverseList_double_pointers(self, head: Optional[ListNode]) -> Optional[ListNode]:
        if head == []:
            return []
        # 双指针法, 一个在后一个在前
        prev = None
        current = head

        while current:
            next_node = current.next # 保存下一个节点
            current.next = prev # 反转当前节点的指针
            prev = current # 更新前一个节点
            current = next_node # 移动到下一个节点

        return prev

    # 递归法
    # 定义一个辅助函数, 然后迭代

    def reverseList_recursion(self, head: Optional[ListNode]) -> Optional[ListNode]:
        # recursive function
        def reverse(cur: ListNode, pre: ListNode) -> ListNode:
            if cur == None:
                return pre
            temp = cur.next
            cur.next = pre
            return reverse(temp, cur)

        return reverse(head, None)
```

## 复杂度分析

### 时间复杂度

- 每个节点仅被访问一次, 共  $n$  个节点。
- 时间复杂度为  $O(n)$ 。

### 空间复杂度

- 由于递归调用栈的深度为链表的长度  $n$ , 空间复杂度为  $O(n)$ 。

与迭代法相比, 递归法在空间使用上较高。

复杂度对比:

方法	时间复杂度	空间复杂度
递归法	$O(n)$	$O(n)$
迭代法	$O(n)$	$O(1)$

递归法的逻辑较直观, 但在链表很长时可能导致栈溢出问题, 因此推荐在需要优化空间使用时采用迭代法。

```
In [ ]:
```

## 24. 两两交换链表中的节点

用虚拟头结点, 这样会方便很多。

本题链表操作就比较复杂了, 建议大家先看视频, 视频里我讲解了注意事项, 为什么需要temp保存临时节点。

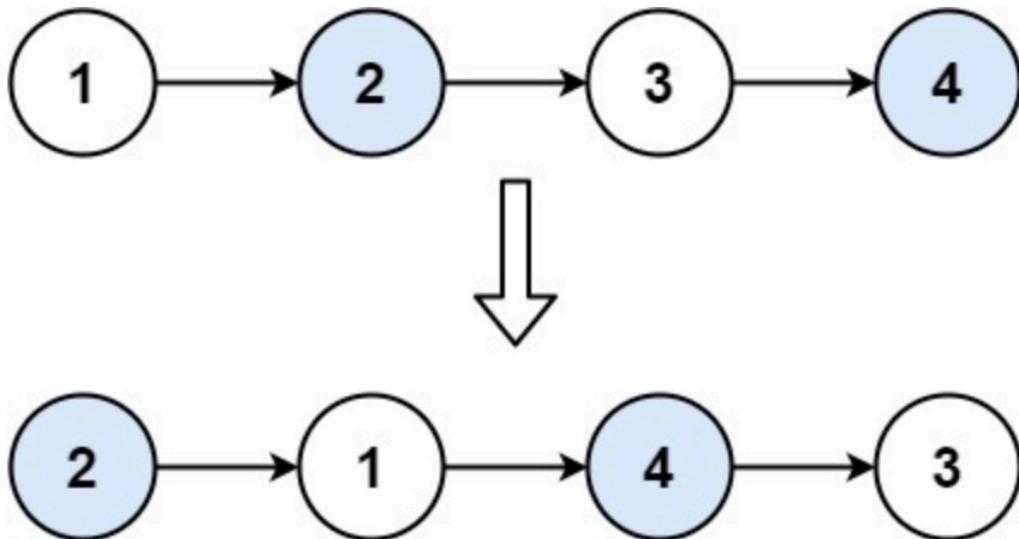
题目链接/文章讲解/视频讲解:

<https://programmercarl.com/0024.%E4%B8%A4%E4%B8%A4%E4%BA%A4%E6%8D%A2%E9%93%BE%E8%A1%A8%E4%B8%AD%E7%9A%84%E8%8A%82%E7%9C%8B/>

给你一个链表, 两两交换其中相邻的节点, 并返回交换后链表的头节点。你必须在不修改节点内部的值的情况下完成本题 (即, 只能进行节点交换)。

示例 1:

### 示例 1：



输入: head = [1,2,3,4]

输出: [2,1,4,3]

### 示例 2：

输入: head = []

输出: []

### 示例 3：

输入: head = [1]

输出: [1]

输入: head = [1,2,3,4]

输出: [2,1,4,3]

### 示例 2：

输入: head = []

输出: []

### 示例 3：

输入: head = [1]

输出: [1]

提示:

- 链表中节点的数目在范围 `[0, 100]` 内
- `0 <= Node.val <= 100`

```
In [50]: # 结束条件
# 如果链表长度为奇数，则current.next.next为空
# 如果链表长度为偶数，则current.next为空
# 如果长度为0，也满足偶数条件，current.next为空

# 一起判断就行，不需要分类讨论
```

```

# 交换逻辑
# temp_1 = current.next
# temp_3 = current.next.next

# while current.next and current.next.next:

    # current.next = current.next.next
    # current.next.next = temp_1
    # temp_1.next = temp_3

    # 最后移动current指针
    # current = current.next.next

# return dummy_head.next

# Definition for singly-linked list.
from typing import Optional
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def swapPairs_dummy_head(self, head: Optional[ListNode]) -> Optional[ListNode]:
        dummy_head = ListNode(next=head)
        current = dummy_head

        while current.next and current.next.next:
            temp_1 = current.next
            temp_3 = current.next.next.next
            current.next = current.next.next
            current.next.next = temp_1
            temp_1.next = temp_3

            # 最后移动current指针
            current = current.next.next

        return dummy_head.next

    def swapPairs_recursion(self, head: Optional[ListNode]) -> Optional[ListNode]:
        # 定义递归函数
        def swap(cur: Optional[ListNode]) -> Optional[ListNode]:
            # 递归终止条件 如果链表为空或者只有一个节点 (head 或 head.next 为 None), 直接返回当前节点。
            if not cur or not cur.next:
                return cur

            # 交换当前两个节点
            new_head = cur.next # 第二个节点作为新头
            cur.next = swap(new_head.next) # 递归处理后续节点并连接
            new_head.next = cur # 完成当前两个节点的交换

            return new_head # 返回新头节点

        return swap(head)

```

```

In [51]: # 定义链表节点类
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

# 辅助函数: 从列表创建链表
def create_linked_list(values):
    if not values:
        return None
    head = ListNode(values[0])
    current = head
    for value in values[1:]:
        current.next = ListNode(value)
        current = current.next
    return head

# 辅助函数: 从链表生成列表
def linked_list_to_list(head):
    result = []
    while head:
        result.append(head.val)
        head = head.next
    return result

# 测试代码
def test_swap_pairs():
    solution = Solution()
    test_cases = [
        {"input": [1, 2, 3, 4], "expected": [2, 1, 4, 3]},
        {"input": [], "expected": []},
        {"input": [1], "expected": [1]},
        {"input": [1, 2, 3], "expected": [2, 1, 3]},
    ]
    for i, case in enumerate(test_cases):

```

```

head = create_linked_list(case["input"])
expected = case["expected"]

# 测试虚拟头结点方法
swapped_head_dummy = solution.swapPairs_dummy_head(head)
result_dummy = linked_list_to_list(swapped_head_dummy)
assert result_dummy == expected, f"Test case {i + 1} failed for dummy head: Expected {expected}, Got {result_dummy}"
print(f"Test case {i + 1} passed for dummy head: Output {result_dummy}")

# 重新创建链表, 因为 dummy head 方法会修改原链表
head = create_linked_list(case["input"])

# 测试递归方法
swapped_head_recursion = solution.swapPairs_recursion(head)
result_recursion = linked_list_to_list(swapped_head_recursion)
assert result_recursion == expected, f"Test case {i + 1} failed for recursion: Expected {expected}, Got {result_recursion}"
print(f"Test case {i + 1} passed for recursion: Output {result_recursion}")

# 运行测试
test_swap_pairs()

```

Test case 1 passed for dummy head: Output [2, 1, 4, 3]  
Test case 1 passed for recursion: Output [2, 1, 4, 3]  
Test case 2 passed for dummy head: Output []  
Test case 2 passed for recursion: Output []  
Test case 3 passed for dummy head: Output [1]  
Test case 3 passed for recursion: Output [1]  
Test case 4 passed for dummy head: Output [2, 1, 3]  
Test case 4 passed for recursion: Output [2, 1, 3]

In [ ]: # 19. 删除链表的倒数第 N 个结点

给你一个链表，删除链表的倒数第 n 个结点，并且返回链表的头结点。

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: []

示例 3：

输入: head = [1,2], n = 1  
输出: [1]

提示：

链表中结点的数目为 sz

1 <= sz <= 30  
0 <= Node.val <= 100  
1 <= n <= sz

在链表操作中，当我们输出头节点时，通常是指从实际的头节点开始遍历整个链表，而不是只输出头节点本身的价值

In [52]: # 双指针的经典应用，如果要删除倒数第n个节点，让fast移动n步，然后让fast和slow同时移动，直到fast指向链表末尾。删掉slow所指向的节点就可以了。

```

# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next

class Solution:
    def removeNthFromEnd(self, head: Optional[ListNode], n: int) -> Optional[ListNode]:
        dummy_head = ListNode(0, head)

        slow = dummy_head
        fast = dummy_head
        for _ in range(n+1):
            fast = fast.next

        while fast:
            slow = slow.next
            fast = fast.next

        slow.next = slow.next.next
        return dummy_head.next

```

```
In [ ]: # 递归嵌套子函数写法, 但好像不是特别简洁
#
class Solution:
    def removeNthFromEnd(self, head: Optional[ListNode], n: int) -> Optional[ListNode]:
        # 定义递归函数
        def remove(node: Optional[ListNode], n: int) -> int:
            if not node: # 递归终止条件: 到达链表末尾
                return 0

            # 递归返回当前节点的序号 (从链表末尾计数)
            index_from_end = remove(node.next, n) + 1

            # 如果当前节点是倒数第 n 个节点的前一个节点, 删除它
            if index_from_end == n + 1:
                node.next = node.next.next

            return index_from_end

        # 创建虚拟头节点, 以统一删除逻辑
        dummy = ListNode(0, head)

        # 从虚拟头节点开始递归
        remove(dummy, n)

        # 返回新链表的头节点
        return dummy.next
```

## 复杂度分析

双指针法:

- 快指针的移动:
  - 快指针移动  $n + 1$  步, 时间复杂度为  $O(n)$ 。
- 快慢指针的同步移动:
  - 快指针从第  $n + 1$  个节点移动到链表末尾, 同时慢指针移动到倒数第  $n + 1$  个节点, 总步数为  $O(n)$ 。
- 删除节点的操作:
  - 修改指针 `slow.next`, 时间复杂度为  $O(1)$ 。

总时间复杂度:  $O(n)$ 。

- 指针存储:
  - 使用两个指针 `fast` 和 `slow`, 额外的空间复杂度为  $O(1)$ 。

总空间复杂度:  $O(1)$ 。

递归:

1. 时间复杂度:  $O(n)$ 
  - 递归方法需要遍历整个链表一次, 每个节点被访问一次。
2. 空间复杂度:  $O(n)$ 
  - 由于递归调用栈的深度等于链表的长度, 空间复杂度为线性。

方法	时间复杂度	空间复杂度	备注
双指针法	$O(n)$	$O(1)$	高效, 无递归, 无额外空间。
递归方法	$O(n)$	$O(n)$	递归栈的空间消耗较大。

双指针法适用于链表长度较长的场景, 因为其空间复杂度为  $O(1)$ , 没有递归栈溢出的风险, 因此是更推荐的实现方式。

## 链表相交

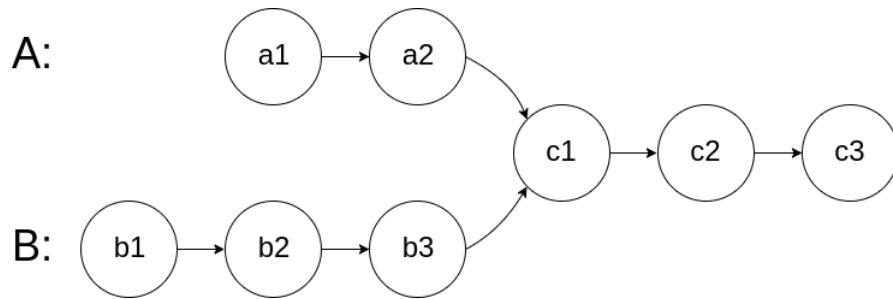
给你两个单链表的头节点 `headA` 和 `headB`, 请你找出并返回两个单链表相交的起始节点。如果两个链表没有交点, 返回 `null`。

题目数据 保证 整个链式结构中不存在环。

注意：函数返回结果后，链表必须 保持其原始结构。

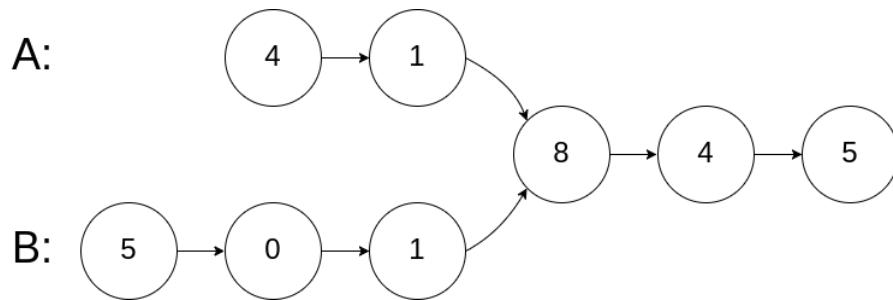
## 图示：链表相交

两个链表在节点 `c1` 开始相交：



## 示例

### 示例 1



输入：

- `intersectVal = 8`
- `listA = [4,1,8,4,5]`
- `listB = [5,0,1,8,4,5]`
- `skipA = 2`
- `skipB = 3`

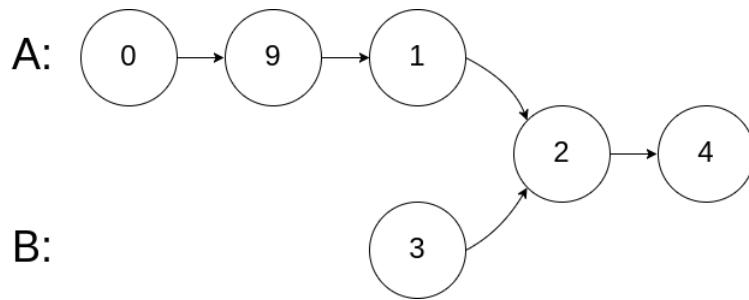
输出：

Intersected at '8'

解释：

- 相交节点的值为 `8` (注意，如果两个链表相交则不能为 `0`)。
- 从各自的表头开始算起：
  - 链表 A 为 `[4,1,8,4,5]`
  - 链表 B 为 `[5,0,1,8,4,5]`
- 在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。

### 示例 2



输入：

- `intersectVal = 2`
- `listA = [0,9,1,2,4]`

- `listB = [3,2,4]`
- `skipA = 3`
- `skipB = 1`

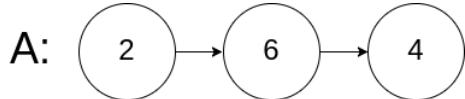
输出:

```
Intersected at '2'
```

解释:

- 相交节点的值为 `2` (注意, 如果两个链表相交则不能为 `0`)。
- 从各自的表头开始算起:
  - 链表 A 为 `[0,9,1,2,4]`
  - 链表 B 为 `[3,2,4]`
- 在 A 中, 相交节点前有 3 个节点; 在 B 中, 相交节点前有 1 个节点。

### 示例 3



输入:

- `intersectVal = 0`
- `listA = [2,6,4]`
- `listB = [1,5]`
- `skipA = 3`
- `skipB = 2`

输出:

```
null
```

解释:

- 从各自的表头开始算起:
  - 链表 A 为 `[2,6,4]`
  - 链表 B 为 `[1,5]`
- 由于这两个链表不相交, 因此 `intersectVal` 必须为 `0`, 而 `skipA` 和 `skipB` 可以是任意值。
- 返回 `null`, 因为链表不相交。

### 提示

- `listA` 中节点数目为  $m$ 。
- `listB` 中节点数目为  $n$ 。
- $0 \leq m, n \leq 3 \times 10^4$
- $1 \leq \text{Node.val} \leq 10^5$
- $0 \leq \text{skipA} \leq m$ 。  
 $0 \leq \text{skipB} \leq n$ 。
- 如果 `listA` 和 `listB` 没有交点, 则 `intersectVal = 0`。
- 如果 `listA` 和 `listB` 有交点, 则
  - $\text{intersectVal} = \text{listA}[\text{skipA} + 1]$
  - $= \text{listB}[\text{skipB} + 1]$

### 进阶

简单来说，就是求两个链表交点节点的指针

注意：交点不是数值相等，而是指针相等。

```
In [ ]: class Solution:
    def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> ListNode:
        lenA, lenB = 0, 0
        cur = headA
        while cur: # 求链表A的长度
            cur = cur.next
            lenA += 1
        cur = headB
        while cur: # 求链表B的长度
            cur = cur.next
            lenB += 1
        curA, curB = headA, headB
        if lenA > lenB: # 让curB为最长链表的头, lenB为其长度
            curA, curB = curB, curA
            lenA, lenB = lenB, lenA
        for _ in range(lenB - lenA): # 让curA和curB在同一起点上(末尾位置对齐)
            curB = curB.next
        while curA: # 遍历curA 和 curB, 遇到相同则直接返回
            if curA == curB: # 注意比较的是node对象
                return curA
            else:
                curA = curA.next
                curB = curB.next
        return None
```

## 环形链表 II

### 问题说明

给定一个链表的头节点 `head`，要求返回链表中开始入环的第一个节点。如果链表无环，则返回 `null`。

- 链表存在环的条件：如果某个节点可以通过连续跟踪 `next` 指针再次到达，则链表存在环。
- 输入的特殊说明：评测系统通过整数 `pos` 标识链表中环的情况：
  - `pos >= 0`：表示链表尾部连接到索引为 `pos` 的节点。
  - `pos == -1`：表示链表中没有环。
- 限制条件：链表不能被修改。

### 示例

#### 示例 1

链表: `[3, 2, 0, -4]`  
`pos = 1`  
输出: 返回索引为 1 的链表节点  
解释: 链表中有一个环，尾部连接到第二个节点。

#### 示例 2

链表: `[1, 2]`  
`pos = 0`  
输出: 返回索引为 0 的链表节点  
解释: 链表中有一个环，尾部连接到第一个节点。

#### 示例 3

链表: `[1]`  
`pos = -1`  
输出: `null`  
解释: 链表中没有环。

```
In [55]: # 快慢指针, 如果相遇就说明有环
# 自己写的I, 无限循环了

class Solution:
    def detectCycle(self, head: Optional[ListNode]) -> Optional[ListNode]:
        slow = ListNode(0, head)
        fast = slow.next
        pos = -1
        # fast开始遍历
        while fast.next != slow.next and fast.next is not None:
            fast = fast.next
```

```

        slow = slow.next
        pos += 1

    return pos

In [ ]: class Solution:
    def detectCycle(self, head: Optional[ListNode]) -> Optional[ListNode]:
        slow = head
        fast = head
        pos = -1
        # fast开始遍历
        while fast.next != slow.next and fast.next is not None:
            fast = fast.next.next
            slow = slow.next

        # If there is a cycle, the slow and fast pointers will eventually meet
        if slow == fast:
            # Move one of the pointers back to the start of the list
            slow = head
            while slow != fast:
                slow = slow.next
                fast = fast.next
            return slow

        return None

```

当  $n > 1$  时，即快指针在环内多绕了  $n - 1$  圈之后才和慢指针相遇，其实仍然可以通过相同的方法找到环的入口节点。这是因为链表的环是一个循环结构，快慢指针在环内的移动和绕行不会改变公式的本质关系。以下是详细解释：

## 1. 环的循环性质

环的结构使得从某个起点出发，无论走几圈，最终都可以回到这个起点。因此，无论快指针在环内绕了多少圈，入口节点的位置相对于头节点的距离  $x$  和相对于相遇点的距离  $z$  是不变的。

## 2. 公式的推导

相遇时的公式：

$$x + y = n \cdot (y + z)$$

目标是求  $x$ ：

$$x = n \cdot (y + z) - y$$

提取环的长度  $(y + z)$ ：

$$\begin{aligned} x &= (n - 1) \cdot (y + z) \\ &\quad + z \end{aligned}$$

- $n \cdot (y + z)$  表示快指针在环内走了  $n$  圈的总步数。
- $(n - 1) \cdot (y + z)$  表示多余的  $n - 1$  圈。
- 最后剩下的一部分  $z$  是从相遇点到环入口的距离。

因此，公式中的  $x = z$ ，无论  $n$  的值是多少，头指针和相遇点的指针按照每次一步的速度相遇时，总会在环入口相遇。

## 3. 为什么 $n > 1$ 的情况不需要特殊处理？

1. 快慢指针的相对关系：快指针多绕  $n - 1$  圈，并不会改变公式的数学关系，因为环的长度  $(y + z)$  已经固定。每绕一圈，快指针的额外距离正好被  $y + z$  抵消。
2. 算法行为不受影响：无论  $n$  的值是多少，相遇后重新从头节点和相遇点分别出发，每次移动一步，最终都会在环的入口节点相遇。多余的  $n - 1$  圈只是在环内循环，最后的  $z$  决定了相遇点。

## 4. 动画类比（直线与环）

- **n=1**：快指针刚好比慢指针多跑了一圈，直接相遇。
- **n=2**：快指针多跑了一圈再相遇，但入口节点位置未变。头指针和相遇点的指针仍然会在入口相遇。
- **n=3 或更多**：快指针在环内绕更多圈才相遇，结果同理。

## 5. 总结

无论  $n$  的值是 1 还是大于 1，公式  $x = z$  始终成立。这是因为环的循环性质将多余的  $n - 1$  圈折叠成了等效的一圈。因此，通过两个指针从头节点和相遇点同时出发，最终会在环的入口节点相遇。

## 242. 有效的字母异位词

给定两个字符串 s 和 t，编写一个函数来判断 t 是否是 s 的字母异位词。s 和 t 仅包含小写字母

示例 1:

输入: s = "anagram", t = "nagaram"

输出: true

示例 2:

输入: s = "rat", t = "car"

输出: false

进阶: 如果输入字符串包含 unicode 字符怎么办? 你能否调整你的解法来应对这种情况?

```
In [ ]: # 尝试1, 没考虑"aaac" "ccac" 这种情况

class Solution:
    def isAnagram(self, s: str, t: str) -> bool:

        if len(s) != len(t):
            return False

        else:
            letters_1 = set()
            letters_2 = set()
            for letter in s:
                letters_1.add(letter)

            for letter in t:
                letters_2.add(letter)

            if letters_1 == letters_2:
                return True
            else:
                return False
```

```
In [1]: # 尝试2, 将字母统计为两个dict

from collections import defaultdict

class Solution:
    def isAnagram(self, s: str, t: str) -> bool:

        def count_letters(word):
            count_dict = {}
            for letter in word:
                count_dict[letter] += 1
            return count_dict

        if len(s) != len(t):
            return False

        else:
            if count_letters(s) == count_letters(t):
                return True
            else:
                return False

# 正确
```

defaultDict用法: defaultdict(int)

另外两种方法:

1. 用ascii序号结合列表的index作为key存储。 ascii序号是 ord(a)
2. 用counter

```
In [ ]: class Solution(object):
    def isAnagram(self, s: str, t: str) -> bool:
        from collections import Counter
        a_count = Counter(s)
        b_count = Counter(t)
        return a_count == b_count

class Solution(object):
    def isAnagram(self, s: str, t: str) -> bool:
        from collections import Counter
        a_count = Counter(s)
        b_count = Counter(t)
        return a_count == b_count
```

Counter 是 dict 字典的子类，Counter 拥有类似字典的 key 键和 value 值，只不过 Counter 中的键为待计数的元素，而 value 值为对应元素出现的次数 count，为了方便介绍统一使用元素和 count 计数来表示。虽然 Counter 中的 count 表示的是计数，但是 Counter 允许 count 的值为 0 或者负值。

## 349. 两个数组的交集

### 问题描述

给定两个数组 `nums1` 和 `nums2`，返回它们的交集。  
输出结果中的每个元素一定是 唯一 的，且可以 不考虑输出结果的顺序。

### 示例

#### 示例 1

输入:

```
nums1 = [1,2,2,1]
nums2 = [2,2]
```

输出:

```
[2]
```

#### 示例 2

输入:

```
nums1 = [4,9,5]
nums2 = [9,4,9,8,4]
```

输出:

```
[9,4]
```

解释:

```
[4,9] 也是可通过的。
```

### 提示

1. 每个数组的长度不超过 1000。
2. 数组中的值范围为  $-10^9$  到  $10^9$ 。

```
In [ ]: from typing import List

def intersection(self, nums1: List[int], nums2: List[int]) -> List[int]:
    len1 = len(nums1)
    len2 = len(nums2)
    result = set()

    # 不需要交换
    # if len2>len1:
    #     temp = nums1
    #     nums1 = nums2
    #     nums2 = temp

    for num in nums1:
        if num in nums2:
            result.add(num)

    return list(result)
```

```
In [7]: # 更高效的方法
# 创建set的时候直接转换
class Solution:
    def intersection(self, nums1: list[int], nums2: list[int]) -> list[int]:
        set1 = set(nums1)
        set2 = set()
        for v in nums2:
            if v in set1:
```

```

        set2.add(v)
    return list(set2)

# 更高效的方法，把长度短的转化为set

class Solution:
    def intersection(self, nums1: List[int], nums2: List[int]) -> List[int]:
        # 把长度大的数组转换为 hash set，记为 fewerNums，另一方记为 moreNums
        if len(nums1) > len(nums2):
            moreNums = set(nums1)
            fewerNums = nums2
        else:
            moreNums = set(nums2)
            fewerNums = nums1

        # 从 moreNums 中向 ans 中添加仅在 fewerNums 中存在的数字
        ans = set()
        for v in moreNums:
            if v in fewerNums:
                ans.add(v)

        # 转换为列表再返回
        return list(ans)

```

```
In [6]: # 一行
class Solution:
    def intersection(self, nums1: list[int], nums2: list[int]) -> list[int]:
        return list(set(nums1) & set(nums2))
```

In [ ]:

## 202. 快乐数

### 问题描述

编写一个算法来判断一个数  $n$  是否是快乐数。

快乐数的定义：

- 对于一个正整数，每次将该数替换为它每一位数字的平方和。
- 重复这个过程，直到：
  - 结果变为  $1$ ，则该数为快乐数。
  - 或者陷入无限循环，无法变到  $1$ ，则该数不是快乐数。

如果  $n$  是快乐数，返回 `true`；否则返回 `false`。

---

### 示例

#### 示例 1

输入：

`n = 19`

输出：

`true`

解释：

```

 $1^2 + 9^2 = 82$ 
 $8^2 + 2^2 = 68$ 
 $6^2 + 8^2 = 100$ 
 $1^2 + 0^2 + 0^2 = 1$ 

```

---

#### 示例 2

输入：

`n = 2`

输出：

`false`

解释：无限循环，无法变为  $1$ 。

In [ ]:

```
In [10]: n = 1000
print(set(str(n)))
```

{'0', '1'}

```
In [ ]: # 第一次尝试，递归不对
class Solution:
```

```

def isHappy(self, n: int) -> bool:
    num_list = list(str(n))

    if set(num_list) & set(str(n)) != []:
        return False

    next_n = 0
    for num in num_list:
        next_n += num ** 2

    return self.isHappy(next_n)

```

In [12]: # 第二次尝试, 用set记录访问过的数字, 通过

```

class Solution:
    def isHappy(self, n: int) -> bool:
        visited = set()

        def get_next_n(n):
            return sum(int(num) ** 2 for num in str(n))

        while n not in visited:
            if n == 1:
                return True
            else:
                visited.add(n)
                n = get_next_n(n)

        return False

```

In [ ]: # 第三次尝试, 逻辑反过来, 也对

```

class Solution:
    def isHappy(self, n: int) -> bool:
        visited = set()

        def get_next_n(n):
            return sum(int(num) ** 2 for num in str(n))

        while n != 1:
            if n in visited:
                return False
            visited.add(n)
            n = get_next_n(n)

        return True

```

## 1. 两数之和

### 问题描述

给定一个整数数组 `nums` 和一个整数目标值 `target`, 请你在该数组中找出和为目标值 `target` 的那两个整数, 并返回它们的数组下标。

- 你可以假设每种输入只会对应一个答案。
- 你不能使用同一个元素两次。
- 你可以按任意顺序返回答案。

### 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]
```

### 示例 3

输入:

```
nums = [3,3]
target = 6
```

输出:

```
[0,1]
```

### 进阶

你能想出一个时间复杂度小于 ( $O(n^2)$ ) 的算法吗?

```
In [ ]: # 尝试1: 双指针遍历
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        slow = 0
        fast = 1
        while slow != fast:
            while fast != len(nums):
                sum = nums[slow] + nums[fast]
                if sum == target:
                    return [slow, fast]
                else:
                    fast += 1
            slow += 1
            fast = slow + 1
        return []
```

```
In [ ]: # 尝试2: hashmap
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        visited = dict()

        for index, value in enumerate(nums):
            margin = target - value
            if margin in visited:
                return [index, visited[margin]]
            else:
                visited[value] = index #注意这里是value
        return []
```

## 454. 四数相加 II

### 问题描述

给定四个整数数组 `nums1`、`nums2`、`nums3` 和 `nums4`，每个数组的长度都是 `n`。请计算有多少个元组  $(i, j, k, l)$  满足以下条件：

1.  $(0 \leq i, j, k, l < n)$
2.  $(\text{nums1}[i] + \text{nums2}[j] + \text{nums3}[k] + \text{nums4}[l] == 0)$

### 示例

#### 示例 1

输入：

```
nums1 = [1, 2]
nums2 = [-2, -1]
nums3 = [-1, 2]
nums4 = [0, 2]
```

输出：

2

解释：两个满足条件的元组为：

1.  $((0, 0, 0, 1)) \rightarrow (\text{nums1}[0] + \text{nums2}[0] + \text{nums3}[0] + \text{nums4}[1] = 1 + (-2) + (-1) + 2 = 0)$
2.  $((1, 1, 0, 0)) \rightarrow (\text{nums1}[1] + \text{nums2}[1] + \text{nums3}[0] + \text{nums4}[0] = 2 + (-1) + (-1) + 0 = 0)$

#### 示例 2

输入：

```
nums1 = [0]
nums2 = [0]
nums3 = [0]
nums4 = [0]
```

输出：

1

```
In [ ]: # 尝试1:

from collections import defaultdict

class Solution:
    def fourSumCount(self, nums1: List[int], nums2: List[int], nums3: List[int], nums4: List[int]) -> int:
        pair1 = defaultdict(int)
        pair2 = defaultdict(int)

        # divide and conquer

        for i, num1 in enumerate(nums1):
            for j, num2 in enumerate(nums2):
                key = num1 + num2 # this is actually the sum of num1 and num2, but as key
                pair1[key] += 1

        for num3 in nums3:
            for num4 in nums4:
                key = num3 + num4
                pair2[key] += 1

        # 在 Python 中, 直接访问不存在的键 (例如 pair2[-key]) 会抛出 KeyError, 即使使用 defaultdict。
        # 更安全的方式是用 get 方法: pair2.get(-key, 0), 如果键不存在, 返回默认值 0。

        count = 0
        for key, _ in pair1.items(): # 注意dict访问用这个方法
            # if pair2.get(-key, 0) != 0:
            if -key in pair2:
                count += pair1[key] * pair2[-key]

        return count
```

```
In [ ]: # 尝试2: 优化, 减少第三个循环

from collections import defaultdict

class Solution:
    def fourSumCount(self, nums1: List[int], nums2: List[int], nums3: List[int], nums4: List[int]) -> int:
        pair1 = defaultdict(int)

        # divide and conquer

        for num1 in nums1:
```

```

        for num2 in nums2:
            key = num1 + num2 # this is actually the sum of num1 and num2, but as key
            pair1[key] += 1

        count = 0
        for num3 in nums3:
            for num4 in nums4:
                key = num3 + num4
                if -key in pair1:
                    count += pair1[-key]

    return count

# 在 Python 中, 直接访问不存在的键 (例如 pair2[-key]) 会抛出 KeyError, 即使使用 defaultdict。
# 更安全的方式是用 get 方法: pair2.get(-key, 0), 如果键不存在, 返回默认值 0。

# 优化点:
# 之前用的是:
# if pair1.get(-key, 0) != 0
# if -key in pair1:

# in 的性能远好于 get

```

## 383. 贼金信

### 问题描述

给你两个字符串：`ransomNote` 和 `magazine`，判断 `ransomNote` 能不能由 `magazine` 里面的字符构成。

如果可以，返回 `true`；否则返回 `false`。

`magazine` 中的每个字符只能在 `ransomNote` 中使用一次。

---

### 示例

#### 示例 1

输入:

```
ransomNote = "a"
magazine = "b"
```

输出:

```
false
```

---

#### 示例 2

输入:

```
ransomNote = "aa"
magazine = "ab"
```

输出:

```
false
```

---

#### 示例 3

输入:

```
ransomNote = "aa"
magazine = "aab"
```

输出:

```
true
```

---

### 提示

1. ( $1 \leq \text{ransomNote.length}, \text{magazine.length} \leq 10^5$ )
2. `ransomNote` 和 `magazine` 由小写英文字母组成。

```
In [ ]: class Solution:
    def canConstruct(self, ransomNote: str, magazine: str) -> bool:
        magazine_set = set(magazine)
        ransomNote_set = set(ransomNote)
        if magazine_set | ransomNote_set != magazine_set:
            return False
        else:
            return True
```

```
# 理解错题意了, magazine 中的每个字符只能在 ransomNote 中使用一次
```

```
x - y # Difference 差集  
set(['a', 'c', 'e'])  
x | y # Union 并集  
set(['a', 'c', 'b', 'e', 'd', 'y', 'x', 'z'])  
x & y # Intersection 交集  
set(['b', 'd'])  
x ^ y # Symmetric difference (XOR) 补集
```

```
In [ ]: class Solution:  
    def canConstruct(self, ransomNote: str, magazine: str) -> bool:  
  
        ransomNote_dict = {}  
        magazine_dict = {}  
  
        for letter in magazine:  
            if letter in magazine_dict:  
                magazine_dict[letter] += 1  
            else:  
                magazine_dict[letter] = 1  
  
        for letter in ransomNote:  
            if letter not in magazine_dict:  
                return False  
            else:  
                magazine_dict[letter] -= 1  
                if magazine_dict[letter] < 0:  
                    return False  
  
        return True
```

```
In [ ]: # 其他版本:  
class Solution:  
    def canConstruct(self, ransomNote: str, magazine: str) -> bool:  
        counts = {}  
        for c in magazine:  
            counts[c] = counts.get(c, 0) + 1  
        for c in ransomNote:  
            if c not in counts or counts[c] == 0:  
                return False  
            counts[c] -= 1  
        return True  
  
# (版本四) 使用Counter  
  
from collections import Counter  
  
class Solution:  
    def canConstruct(self, ransomNote: str, magazine: str) -> bool:  
        return not Counter(ransomNote) - Counter(magazine)  
  
# (版本五) 使用count  
  
class Solution:  
    def canConstruct(self, ransomNote: str, magazine: str) -> bool:  
        return all(ransomNote.count(c) <= magazine.count(c) for c in set(ransomNote))  
  
# (版本六) 使用count(简单易懂)  
class Solution:  
    def canConstruct(self, ransomNote: str, magazine: str) -> bool:  
        for char in ransomNote:  
            if char in magazine and ransomNote.count(char) <= magazine.count(char):  
                continue  
            else:  
                return False  
        return True
```

## 15. 三数之和

<https://leetcode.cn/problems/3sum/solutions/284681/san-shu-zhi-he-by-leetcode-solution/>

### 问题描述

给你一个整数数组 `nums`，判断是否存在三元组 `[nums[i], nums[j], nums[k]]` 满足以下条件：

1. ( $i \neq j$ ), ( $i \neq k$ ), 且 ( $j \neq k$ )。
2. ( $nums[i] + nums[j] + nums[k] == 0$ )。

请返回所有和为 0 且不重复的三元组。

注意：答案中不可以包含重复的三元组。

---

## 示例

### 示例 1

输入：

```
nums = [-1, 0, 1, 2, -1, -4]
```

输出：

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

解释：

- ( $\text{nums}[0] + \text{nums}[1] + \text{nums}[2] = (-1) + 0 + 1 = 0$ )
- ( $\text{nums}[1] + \text{nums}[2] + \text{nums}[4] = 0 + 1 + (-1) = 0$ )
- ( $\text{nums}[0] + \text{nums}[3] + \text{nums}[4] = (-1) + 2 + (-1) = 0$ )

不同的三元组是 `[-1, 0, 1]` 和 `[-1, -1, 2]`。

---

### 示例 2

输入：

```
nums = [0, 1, 1]
```

输出：

```
[]
```

解释：唯一可能的三元组和不为 0。

---

### 示例 3

输入：

```
nums = [0, 0, 0]
```

输出：

```
[[0, 0, 0]]
```

解释：唯一可能的三元组和为 0。

---

## 提示

- ( $3 \leq \text{nums.length} \leq 3000$ )
- ( $-10^5 \leq \text{nums}[i] \leq 10^5$ )

## 第一次尝试，两个问题：

- 索引匹配问题：

在第一部分构造 record2 时，你将 `nums[slicer:]` 的枚举结果赋值给 j，但这里的 j 是相对于 `nums[slicer:]` 的索引，而不是 `nums` 的全局索引。因此，`i != j` 的判断是错误的。

- 重复三元组的问题：

即使使用 set 存储结果，当前代码依然可能出现重复三元组的情况，因为三元组  $(\text{nums}[i], \text{nums}[j], \text{nums}[k])$  是直接用数值进行存储，缺乏排序来确保唯一性。

```
In [3]: from typing import List
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        # key is (i,j,k)
        # i != j, i != k, j != k

        record2 = dict()
        result = set()

        for i, num1 in enumerate(nums):
            slicer = i + 1
            for j, num2 in enumerate(nums[slicer:]):
                if i != j:
                    if num1+num2 in record2:
                        record2[num1+num2].append([i,j])
                    else:
                        record2[num1+num2] = [[i,j]]

        for k, num3 in enumerate(nums):
            if -num3 in record2:
```

```

        for pair in record2[-num3]:
            i, j = pair
            if i != k and j != k:
                result.add((nums[i], nums[j], nums[k]))

    return list(result)

In [4]: class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        # key is (i,j,k)
        # i != j, i != k, j != k

        record = dict()
        result = set()

        for i, num1 in enumerate(nums):
            slicer = i + 1
            for j, num2 in enumerate(nums[slicer:], start = slicer): # 修正1: j从slicer开始
                if i != j:
                    if num1+num2 in record:
                        record[num1+num2].append([i,j])
                    else:
                        record[num1+num2] = [[i,j]]

        for k, num3 in enumerate(nums):
            if -num3 in record:
                for pair in record[-num3]:
                    i, j = pair
                    if i != k and j != k:
                        triplet = tuple(sorted((nums[i], nums[j], nums[k]))) # 修正2: 排序tuple, 确保唯一性
                        result.add(triplet)

    return list(result)

### 结果正确, 但超出时间限制

```

```

In [ ]: # 答案: 双指针法

class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        result = []
        nums.sort()

        for i in range(len(nums)):
            # 如果第一个元素已经大于0, 不需要进一步检查
            if nums[i] > 0:
                return result

            # 跳过相同的元素以避免重复
            if i > 0 and nums[i] == nums[i - 1]:
                continue

            left = i + 1
            right = len(nums) - 1

            while right > left:
                sum_ = nums[i] + nums[left] + nums[right]

                if sum_ < 0:
                    left += 1
                elif sum_ > 0:
                    right -= 1
                else:
                    result.append([nums[i], nums[left], nums[right]])

                    # 跳过相同的元素以避免重复
                    while right > left and nums[right] == nums[right - 1]:
                        right -= 1
                    while right > left and nums[left] == nums[left + 1]:
                        left += 1

                    right -= 1
                    left += 1

            return result

```

```

In [ ]: # 答案: 字典法
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        result = []
        nums.sort()
        # 找出a + b + c = 0
        # a = nums[i], b = nums[j], c = -(a + b)
        for i in range(len(nums)):
            # 排序之后如果第一个元素已经大于零, 那么不可能凑成三元组
            if nums[i] > 0:
                break
            if i > 0 and nums[i] == nums[i - 1]: # 三元组元素a去重
                continue
            d = {}
            # d 是用来记录b 也就是num[j] 的, 如果新的b在d中, 就跳过
            for j in range(i + 1, len(nums)):
                if j > i + 2 and nums[j] == nums[j-1] == nums[j-2]: # 三元组元素b去重

```

```

        continue
    c = 0 - (nums[i] + nums[j])
    if c in d:
        result.append([nums[i], nums[j], c])
        d.pop(c) # 三元组元素c去重
    else:
        d[nums[j]] = j
return result

```

## 18. 四数之和（难，回来复习）

### 问题描述

给你一个由  $n$  个整数组成的数组 `nums`，和一个目标值 `target`。请你找出并返回满足下述全部条件且不重复的四元组 (`[nums[a], nums[b], nums[c], nums[d]]`)（若两个四元组元素一一对应，则认为两个四元组重复）：

- $(0 \leq a, b, c, d < n)$
- $(a, b, c)$  和  $(d)$  互不相同
- $(\text{nums}[a] + \text{nums}[b] + \text{nums}[c] + \text{nums}[d] = \text{target})$

你可以按任意顺序返回答案。

### 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]]
```

### 提示

- $(4 \leq n \leq 200)$
- $(-10^9 \leq \text{nums}[i] \leq 10^9)$
- $(-10^9 \leq \text{target} \leq 10^9)$

```
In [ ]: # 用字典，遍历三个数，然后对最后一个数用dict

class Solution(object):
    def fourSum(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[List[int]]
        """
        # 创建一个字典来存储输入列表中每个数字的频率
        freq = {}
        for num in nums:
            freq[num] = freq.get(num, 0) + 1

        # 创建一个集合来存储最终答案，并遍历4个数字的所有唯一组合
        ans = set()
        for i in range(len(nums)):
            for j in range(i + 1, len(nums)):
                for k in range(j + 1, len(nums)):
                    val = target - (nums[i] + nums[j] + nums[k])
                    if val in freq:
                        # 确保没有重复
                        count = (nums[i] == val) + (nums[j] == val) + (nums[k] == val)
                        if freq[val] > count:
                            ans.add(tuple(sorted([nums[i], nums[j], nums[k], val])))

        return [list(x) for x in ans]

# 可以过测试，但是时间比较慢
```

```
In [ ]: # 双指针法

class Solution:
    def fourSum(self, nums: List[int], target: int) -> List[List[int]]:
        nums.sort()
        n = len(nums)
        result = []
        for i in range(n):
            if nums[i] > target and nums[i] > 0 and target > 0:# 剪枝 (可省)
                break
            if i > 0 and nums[i] == nums[i-1]:# 去重
                continue
            for j in range(i+1, n):
                if nums[i] + nums[j] > target and target > 0: #剪枝 (可省)
                    break
                if j > i+1 and nums[j] == nums[j-1]: # 去重
                    continue
                left, right = j+1, n-1
                while left < right:
                    s = nums[i] + nums[j] + nums[left] + nums[right]
                    if s == target:
                        result.append([nums[i], nums[j], nums[left], nums[right]])
                        while left < right and nums[left] == nums[left+1]:
                            left += 1
                        while left < right and nums[right] == nums[right-1]:
                            right -= 1
                        left += 1
                        right -= 1
                    elif s < target:
                        left += 1
                    else:
                        right -= 1
                return result
```

```
In [ ]:
```

## 344. 反转字符串

### 提示

编写一个函数，其作用是将输入的字符串反转过来。输入字符串以字符数组  $s$  的形式给出。

不要给另外的数组分配额外的空间，你必须原地修改输入数组，使用  $O(1)$  的额外空间解决这一问题。

### 示例

#### 示例 1:

输入:  $s = ["h", "e", "l", "l", "o"]$   
输出:  $["o", "l", "l", "e", "h"]$

#### 示例 2:

输入:  $s = ["H", "a", "n", "n", "a", "h"]$   
输出:  $["h", "a", "n", "n", "a", "H"]$

### 提示

- $1 \leq s.length \leq 10^5$
- $s[i]$  都是 ASCII 码表中的可打印字符

```
In [1]: from typing import List
class Solution:
    def reverseString(self, s: List[str]) -> None:
        """
        Do not return anything, modify s in-place instead.
        """
        length = len(s)
        temp = None
        for i in range(1, length//2+1):
            temp = s[i-1]
            s[i-1] = s[-i]
            s[-i] = temp
        # s[i], s[n - i - 1] = s[n - i - 1], s[i] 用这一句就能同时调换，无需用temp
```

```
In [ ]:
```

## 541. 反转字符串 II

### 题目描述

给定一个字符串  $s$  和一个整数  $k$ ，从字符串开头算起，每计数至  $2k$  个字符，就反转这  $2k$  字符中的前  $k$  个字符。

- 如果剩余字符少于  $k$  个，则将剩余字符全部反转。
- 如果剩余字符小于  $2k$  但大于或等于  $k$  个，则反转前  $k$  个字符，其余字符保持原样。

### 示例

#### 示例 1:

输入:  $s = "abcdefg"$ ,  $k = 2$   
输出: "bacdfeg"

#### 示例 2:

输入:  $s = "abcd"$ ,  $k = 2$   
输出: "bacd"

```
In [2]: # 第一次尝试，反转不成功
```

```
class Solution:
    def reverseStr(self, s: str, k: int) -> str:
        left = 0
        right = 2*k

        while right < len(s):
            remaining = s[left:]
            remaining_length = len(remaining)

            if remaining_length < k:
                s[left:].reverse()
```

```

        elif k <= remaining_length < 2*k:
            s[left:left+k].reverse()

        left += 2*k
        right += 2*k
    return s

```

字符串不可变：在 Python 中，字符串是不可变的类型，无法直接通过切片（如 `s[left:].reverse()` 或 `s[left:left+k].reverse()`）来修改字符串。`.reverse()` 是列表的方法，而不是字符串的方法。

逻辑错误：

`right` 的计算和循环条件有问题。如果字符串长度小于  $2k$  时，`right` 可能永远无法满足循环条件。即使 `remaining` 和 `remaining_length` 的逻辑正确，循环的核心功能并没有修改字符串。

无实际修改：在 `s[left:].reverse()` 或 `s[left:left+k].reverse()` 这种操作中，切片只是生成了新的字符串或子列表，实际的 `s` 并未改变。

```
In [ ]: # 第二次尝试, 转换成列表, 还是没有对列表操作
# 因为切片之后会产生新的列表

class Solution:
    def reverseStr(self, s: str, k: int) -> str:
        s = list(s)
        left = 0
        right = 2*k

        while right < len(s):
            remaining = s[left:]
            remaining_length = len(remaining)

            if remaining_length < k:
                s[left:].reverse()
            elif k <= remaining_length < 2*k:
                s[left:left+k].reverse()

            left += 2*k
            right += 2*k

        return ''.join(s) # list转换成列表的方法
```

知识点：

- str不可变，不能通过.reverse()反转
- str = ''.join(list) # 可以转化list为string
- `s[left:].reverse()` 会尝试反转从 `left` 开始的切片，但这并不会修改原列表的内容，而是创建了一个新的切片。你需要明确对原列表操作。
- `s[left:] = reversed(s[left:])` 才能反转原列表

```
In [ ]: # 根据gpt提示下的答案, 因为这里有知识点不知道

class Solution:
    def reverseStr(self, s: str, k: int) -> str:
        s = list(s)
        left = 0

        while left < len(s):
            remaining_length = len(s) - left

            if remaining_length < k:
                s[left:] = reversed(s[left:])
            elif k <= remaining_length < 2*k:
                s[left:left + k] = reversed(s[left:left + k])
            else: # 剩余字符多于 2k, 正常处理前 k 个字符
                s[left:left + k] = reversed(s[left:left + k])

            left += 2*k

        return ''.join(s) # list转换成列表的方法
```

```
In [ ]: # 答案
class Solution:
    def reverseStr(self, s: str, k: int) -> str:
        i = 0
        chars = list(s)

        while i < len(chars):
            chars[i:i + k] = chars[i:i + k][::-1] # 反转后, 更改原值为反转后值
            i += k * 2

        return ''.join(chars)
```

## 54. 替换数字

## 题目描述

给定一个字符串  $s$ , 它包含小写字母和数字字符, 请编写一个函数, 将字符串中的字母字符保持不变, 而将每个数字字符替换为 `number`。

例如, 对于输入字符串 `"a1b2c3"`, 函数应该将其转换为 `"anumberbnumbercnumber"`。

## 输入描述

输入一个字符串  $s$ ,  $s$  仅包含小写字母和数字字符。

## 输出描述

打印一个新的字符串, 其中每个数字字符都被替换为了 `number`。

## 示例

输入:

`a1b2c3`

输出:

`anumberbnumbercnumber`

```
In [5]: class Solution:
    def replaceNumber(self, s: str) -> str:
        list_s = list(s)
        for i in range(len(list_s)):
            if list_s[i] in "0123456789":
                list_s[i] = 'number'
            else:
                continue
        return ''.join(list_s)
```

```
In [ ]: # GPT优化版 (差不多)
def replaceNumber(s: str) -> str:
    result = []
    for char in s:
        if '0' <= char <= '9':
            result.append('number') # 替换数字
        else:
            result.append(char) # 保持原字符
    return ''.join(result) # 拼接结果
```

虽然用python比较简单, 但是注意list的添加复杂度 (并不会增加, 因为相当于hashmap) Python 中字符串是不可变的, 最后还是要转换为列表合并

## 时间复杂度

1. 字符串转列表:

- `list(s)` 将字符串转换为列表, 时间复杂度为  $O(n)$ , 其中  $n$  是字符串的长度。

2. 遍历列表:

- `for i in range(len(list_s))` 遍历列表中的每个元素, 执行  $n$  次循环。
- 每次判断 `if list_s[i] in "0123456789"`:
  - `"0123456789"` 是一个长度为 10 的字符串。
  - Python 的 `in` 操作符对字符串的复杂度为  $O(m)$ , 其中  $m$  是目标字符串的长度。
  - 在本例中,  $m = 10$ , 这是一个常数, 因此可以视为  $O(1)$ 。

3. 拼接字符串:

- `'''.join(list_s)` 将列表重新拼接为字符串, 时间复杂度为  $O(n)$ 。

综合上述步骤, 总的时间复杂度为:

$$\begin{aligned} O(n) + O(n \cdot 1) \\ + O(n) = O(n) \end{aligned}$$

## 空间复杂度

1. 列表存储:

- `list(s)` 创建了一个与输入字符串等长的列表, 空间复杂度为  $O(n)$ 。

2. 返回新字符串:

- `'''.join(list_s)` 会创建一个新的字符串, 长度与原字符串相同, 空间复杂度为  $O(n)$ 。

综合上述, 空间复杂度为:

$$O(n)$$

在 Python 中，替换列表中的元素不会有额外复杂度，因为列表的索引操作和赋值操作都是常数时间复杂度  $O(1)$ 。

具体来说：

## 替换列表元素的复杂度

当你执行以下代码：

```
list_s[i] = 'number'
```

1. 定位元素：

- `list_s[i]` 使用索引访问列表中的元素。
- 在 Python 的底层实现中，列表是一个动态数组，访问元素的复杂度为  $O(1)$ 。

2. 替换元素：

- 替换操作只是将新值 `'number'` 写入已经分配好的内存地址，不涉及其他操作。
- 替换操作的复杂度也是  $O(1)$ 。

因此，单次替换的复杂度是  $O(1)$ 。

```
In [6]: # 测试用例
solution = Solution()

# 举例测试
s1 = "a1b2c3"
output1 = solution.replaceNumber(s1)
print(output1) # 期望输出: "anumberbnumbercnumber"

# 边界测试: 空字符串
s2 = ""
output2 = solution.replaceNumber(s2)
print(output2) # 期望输出: ""

# 全是数字
s3 = "123456"
output3 = solution.replaceNumber(s3)
print(output3) # 期望输出: "numbernumbernumbernumbernumbernumber"

# 全是字母
s4 = "abcdef"
output4 = solution.replaceNumber(s4)
print(output4) # 期望输出: "abcdef"

# 混合测试
s5 = "abc123xyz456"
output5 = solution.replaceNumber(s5)
print(output5) # 期望输出: "abcnumbernumbernumberxyznumbernumbernumber"

anumberbnumbercnumber
numbernumbernumbernumbernumbernumber
abcdef
abcnumbernumbernumberxyznumbernumbernumber
```

```
In [ ]:
```

# 151. 反转字符串中的单词

## 题目描述

给你一个字符串  $s$ , 请你反转字符串中单词的顺序。

- 单词是由非空格字符组成的字符串。
- $s$  中使用至少一个空格将字符串中的单词分隔开。

返回单词顺序颠倒且单词之间用单个空格连接的结果字符串。

注意：

- 输入字符串  $s$  中可能会存在前导空格、尾随空格或者单词间的多个空格。
- 返回的结果字符串中, 单词间应当仅用单个空格分隔, 且不包含任何额外的空格。

## 示例

### 示例 1:

输入:  $s = \text{"the sky is blue"}$   
输出: "blue is sky the"

### 示例 2:

输入:  $s = \text{" hello world "}$   
输出: "world hello"  
解释: 反转后的字符串中不能存在前导空格和尾随空格。

### 示例 3:

输入:  $s = \text{"a good example"}$   
输出: "example good a"  
解释: 如果两个单词间有多余的空格, 反转后的字符串需要将单词间的空格减少到仅有一个。

## 提示

- $1 \leq s.length \leq 10^4$
- $s$  包含英文大小写字母、数字和空格 ' '
- $s$  中至少存在一个单词

## 进阶

- 如果字符串在你使用的编程语言中是一种可变数据类型, 请尝试使用  $O(1)$  额外空间复杂度的原地解法。

```
In [ ]: # python库, 但是会使用额外空间
class Solution:
    def reverseWords(self, s: str) -> str:
        return " ".join(s.split()[::-1])
```

```
In [ ]: # 实际是三个双指针

class Solution:
    def trim_spaces(self, s):
        # 双指针
        left, right = 0, len(s) - 1
        while left <= right and s[left] == ' ':
            left += 1
        while left <= right and s[right] == ' ':
            right -= 1 # 用改变指针的方法去掉空格?

        tmp = []
        # 处理单词中间的空格
        while left <= right:
            if s[left] != " ":
                tmp.append(s[left])
            elif not tmp or tmp[-1] != ' ':
                tmp.append(s[left])
            left += 1
        return tmp

    def reverse_string(self, list, left, right):
        while left < right:
            # 从list的头和尾反转
            list[left], list[right] = list[right], list[left]
            left += 1
            right -= 1
        return None
```

```

def reverse_each_word(self, word):
    left, right = 0, 0
    word_len = len(word)
    while left < word_len:
        while right < word_len and word[right] != ' ':
            right += 1
        self.reverse_string(word, left, right - 1)
        left = right + 1
        right = left # 重置 `right` 与 `left` 同步

def reverseWords(self, s: str) -> str:
    letter_list = self.trim_spaces(s)
    self.reverse_string(letter_list, 0, len(letter_list) - 1)
    self.reverse_each_word(letter_list)
    return ''.join(letter_list)

```

## 55. 右旋字符串

### 题目描述

字符串的右旋转操作是将字符串尾部的若干个字符转移到字符串的前面。给定一个字符串  $s$  和一个正整数  $k$ ，请编写一个函数，将字符串中的后面  $k$  个字符移到字符串的前面，实现字符串的右旋转操作。

例如，对于输入字符串 "abcdefg" 和整数  $k = 2$ ，函数应该将其转换为 "fgabcde"。

### 输入描述

输入共包含两行：

1. 第一行为一个正整数  $k$ ，代表右旋转的位数。
2. 第二行为字符串  $s$ ，代表需要旋转的字符串。

### 输出描述

输出共一行，为进行了右旋转操作后的字符串。

### 示例

输入：

```
2
abcdefg
```

```
In [8]: class Solution:
    # 第一次尝试，用string split，通过
    # python中字符串是不可变的，所以也需要额外空间
    def rotateString_split(self, k: int, s: str) -> str:
        str_len = len(s)
        string_1 = s[str_len-k:]
        string_2 = s[:str_len-k]
        new_string = string_1 + string_2
        print(new_string)
        return new_string

    # 答案中的简洁写法
    def rotateString_clean(self, k: int, s: str) -> str:
        s = s[len(s)-k:] + s[:len(s)-k]
        return s

    # 用python列表模拟答案的调换两次写法
    def rotateString(self, k: int, s: str) -> str:
        """
        使用三次反转法实现字符串右旋
        :param k: 右旋的位数
        :param s: 输入的字符串
        :return: 右旋后的字符串
        """
        # 确保旋转位数不超过字符串长度
        k = k % len(s) # 右旋 k 等价于右旋 k % length

        # 将字符串转换为列表，因为字符串是不可变的
        s_list = list(s)

        # 1. 整体反转
        s_list.reverse()
        # 示例："abcdefg" -> "gfedcba"

        # 2. 反转前 k 个字符
        s_list[:k] = reversed(s_list[:k])
        # 示例：前 k=2 个字符反转后，"gfedcba" -> "fgedcba"
```

```

# 3. 反转后 length-k 个字符
s_list[k:] = reversed(s_list[k:])
# 示例: 后 length-k=5 个字符反转后, "fgedcba" -> "fgabcde"

# 将列表转换回字符串并返回
return ''.join(s_list)

def test():
    solution = Solution()

    # 示例测试用例
    assert solution.rotateString(2, "abcdefg") == "fgabcde"
    assert solution.rotateString(3, "abcdefg") == "efgabcd"

    # 边界测试
    assert solution.rotateString(0, "abcdefg") == "abcdefg" # k=0, 字符串保持不变
    assert solution.rotateString(7, "abcdefg") == "abcdefg" # k 等于字符串长度
    assert solution.rotateString(8, "abcdefg") == "gabcdef" # k 超过字符串长度, 等价于 k=1
    assert solution.rotateString(1, "a") == "a" # 单字符测试

    # 大数据测试
    assert solution.rotateString(3, "a" * 10000) == "aaa" * 3333 + "a"

    print("所有测试用例通过! ")

# 调用测试函数
test()

```

所有测试用例通过！

## 知识点：

- python中list的反转用的是 `reversed(list)`, 需要重新赋值  
`s_list[:k] = reversed(s_list[:k])`
- 三次反转法 /\* 整体反转: \* 使用 `s_list.reverse()` 将整个字符串反转。
- 反转后的效果: 原字符串 `"abcdefg"` -> `"gfedcba"`。
- 反转前 n 个字符: \* 使用切片 `s_list[:n]` 对前 n 个字符进行反转。
- 示例: `n=2` 时, 反转 `"gfedcba"` 的前两位得到 `"fgedcba"`。
- 反转后 length-n 个字符: \* 使用切片 `s_list[n:]` 对后 length-n 个字符进行反转。
- 示例: 后 5 个字符反转后, `"fgedcba"` -> `"fgabcde"`。
- 在 C++ 中, 字符串是可变的, 可以原地修改, 因此三次反转法可以做到  $O(1)$  空间复杂度。
- 在 Python 中, 由于字符串是不可变的, 需要借助列表来模拟修改, 因此存在  $O(n)$  的空间开销。

## 28. 找出字符串中第一个匹配项的下标

### 题目描述

给你两个字符串 `haystack` 和 `needle`, 请你在 `haystack` 字符串中找出 `needle` 字符串的第一个匹配项的下标 (下标从 0 开始)。如果 `needle` 不是 `haystack` 的一部分, 则返回 `-1`。

### 示例

#### 示例 1:

输入: `haystack = "sadbutsad", needle = "sad"`  
输出: `0`  
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。  
当 `needle` 是空字符串时我们应当返回 0

In [9]: 知识点: KMP (Knuth, Morris和Pratt) 算法 KMP主要应用在字符串匹配上。

KMP的主要思想是当出现字符串不匹配时, 可以知道一部分之前已经匹配的文本内容, 可以利用这些信息避免从头再去做匹配了。

next数组就是一个前缀表 (prefix table)。

前缀表是用来回退的, 它记录了模式串与主串(文本串)不匹配的时候, 模式串应该从哪里开始重新匹配。

要在文本串: `aabaabaafa` 中查找是否出现过一个模式串: `aabaaf`。

前缀表：记录下标1之前（包括1）的字符串中，有多大长度的相同前缀后缀。

后缀：只包含尾字母不包含首字母的所有字串 f, af, aaf, baaf, abaaf,

aabaaf不是后缀因为a是首字母

前缀：包含首字母不包含尾字母的所有字符串  
a, aa, aab, aaba, aabaa, aabaaf

最长相等前后缀：

a: 0  
aa: 1 前缀a = 后缀a  
aab: 0  
aaba: 长度1, a和a  
aabaa: 长度2, a和a, aa和aa  
aabaaf: 0

前缀表就是

[0, 1, 0, 1, 2, 0]  
对应  
[a, a, b, a, a, f]

通过查找前缀表

例如遇到aabaaaf, f时, 查找aabaa, 发现前缀表对应2, 2代表最长相等前缀是aa, 我们可以从aabaa的b中开始, 也就是index对应2, 就是查表得到的2, 因为index从0开始

next()函数就在做这个功能, 查前缀表, 输出下一个

Cell In[9], line 1

知识点: KMP (Knuth, Morris和Pratt) 算法 KMP主要应用在字符串匹配上。

SyntaxError: invalid character ': ' (U+FF1A)

In [ ]:

459. 重复的子字符串 简单 相关标签 相关企业 给定一个非空的字符串 s , 检查是否可以通过由它的一个子串重复多次构成。

示例 1:

输入: s = "abab" 输出: true 解释: 可由子串 "ab" 重复两次构成。示例 2:

输入: s = "aba" 输出: false 示例 3:

输入: s = "abcabcabcabc" 输出: true 解释: 可由子串 "abc" 重复四次构成。(或子串 "abcabc" 重复两次构成。)

提示:

1 <= s.length <= 104 s 由小写英文字母组成

这两题太难, 无法集中注意力, 先跳过

## 232. 用栈实现队列

### 题目描述

请你仅使用两个栈实现先入先出队列。队列应当支持一般队列支持的所有操作 (`push`、`pop`、`peek`、`empty`)：

实现 `MyQueue` 类：

- `void push(int x)`：将元素 `x` 推到队列的末尾。
- `int pop()`：从队列的开头移除并返回元素。
- `int peek()`：返回队列开头的元素。
- `boolean empty()`：如果队列为空，返回 `true`；否则，返回 `false`。

### 说明

1. 你只能使用标准的栈操作：
  - `push to top`、`peek/pop from top`、`size` 和 `is empty`。
2. 你所使用的语言也许不支持栈。你可以使用 `list` 或者 `deque` (双端队列) 来模拟一个栈，只要是标准的栈操作即可。

### 示例

输入：

```
["MyQueue", "push", "push", "peek", "pop", "empty"]
[], [1], [2], [], [], []]
```

输出：

```
[null, null, null, 1, 1, false]
```

解释：

```
MyQueue myQueue = new MyQueue();
myQueue.push(1); // queue is: [1]
myQueue.push(2); // queue is: [1, 2] (leftmost is front of the queue)
myQueue.peek(); // return 1
myQueue.pop(); // return 1, queue is [2]
myQueue.empty(); // return false
```

### 提示

- $1 \leq x \leq 9$
- 最多调用 100 次 `push`、`pop`、`peek` 和 `empty`。
- 假设所有操作都是有效的 (例如，一个空的队列不会调用 `pop` 或者 `peek` 操作)。

### 进阶

你能否实现每个操作均摊时间复杂度为  $O(1)$  的队列？

换句话说，执行  $n$  个操作的总时间复杂度为  $O(n)$ ，即使其中一个操作可能花费较长时间。

核心思想是用两个stack模拟queue

```
In [2]: class MyQueue:

    def __init__(self):
        self.stack_in = []
        self.stack_out = []

    def push(self, x: int) -> None:
        """
        有新元素进来，就往in里面push
        """
        self.stack_in.append(x)

    def pop(self) -> int:
        """
        获取队列的头部元素（第一个元素），并移除它。
        Removes the element from in front of queue and returns that element.
        """
        if self.empty():
            return None
        if self.stack_out:
            return self.stack_out.pop()
        else:
            # 这步是最关键的，就是相当于把 stack_in 按照反转的顺序依次放进stack_out中，然后pop最后一个
            for _ in range(len(self.stack_in)):
                self.stack_out.append(self.stack_in.pop())

    def peek(self) -> int:
        """
        返回队列开头的元素。
        Returns the element at the front of the queue.
        """
        if self.empty():
            return None
        if self.stack_out:
            return self.stack_out[-1]
        else:
            return self.stack_in[0]

    def empty(self) -> bool:
        """
        判断队列是否为空。
        Returns true if the queue is empty, false otherwise.
        """
        return len(self.stack_in) == 0
```

```

        self.stack_out.append(self.stack_in.pop())
    return self.stack_out.pop()

def peek(self) -> int:
    """
    获取队列的头部元素（第一个元素），但不移除它。
    Get the front element.
    """
    # 先pop出来最前面一个元素，记下，再加到stack_out回去
    ans = self.pop()
    self.stack_out.append(ans)
    return ans

def empty(self) -> bool:
    """
    只要in或者out有元素，说明队列不为空
    """
    return not (self.stack_in or self.stack_out)

# Your MyQueue object will be instantiated and called as such:
# obj = MyQueue()
# obj.push(x)
# param_2 = obj.pop()
# param_3 = obj.peek()
# param_4 = obj.empty()

```

## 225. 用队列实现栈

### 题目描述

请你仅使用两个队列实现一个后入先出（LIFO）的栈，并支持普通栈的全部四种操作：

- `void push(int x)`：将元素 `x` 压入栈顶。
- `int pop()`：移除并返回栈顶元素。
- `int top()`：返回栈顶元素。
- `boolean empty()`：如果栈是空的，返回 `true`；否则，返回 `false`。

### 注意

1. 你只能使用队列的标准操作——也就是 `push to back`、`peek/pop from front`、`size` 和 `is empty` 这些操作。
2. 你可以使用 `list`（列表）或者 `deque`（双端队列）来模拟一个队列，只要是标准的队列操作即可。

### 示例

输入：

```
["MyStack", "push", "push", "top", "pop", "empty"]
[[], [1], [2], [], [2], []]
```

输出：

```
[null, null, null, 2, 2, false]
```

解释：

```
MyStack myStack = new MyStack();
myStack.push(1);
myStack.push(2);
myStack.top(); // 返回 2
myStack.pop(); // 返回 2
myStack.empty(); // 返回 False
```

### 进阶

你能否仅用一个队列来实现栈？

```
In [ ]: # ChatGPT 用两个list实现，但是意义不大
class MyStack:
    def __init__(self):
        self.queue1 = [] # 主队列，用于存储元素
        self.queue2 = [] # 辅助队列，用于实现 push 的调换逻辑

    def push(self, x: int) -> None:
        """
        将元素压入栈顶
        """
        # 先将新元素放入辅助队列
        self.queue2.append(x)

        # 再将 queue1 的所有元素移到 queue2
        while self.queue1:
```

```

        self.queue2.append(self.queue1.pop(0))

    # 交换 queue1 和 queue2
    self.queue1, self.queue2 = self.queue2, self.queue1

    def pop(self) -> int:
        """
        移除并返回栈顶元素
        """
        if self.empty():
            return None
        return self.queue1.pop(0)

    def top(self) -> int:
        """
        返回栈顶元素
        """
        if self.empty():
            return None
        return self.queue1[0]

    def empty(self) -> bool:
        """
        如果栈为空, 返回 True; 否则, 返回 False
        """
        return len(self.queue1) == 0

```

`deque` (双端队列) 是 Python 标准库 `collections` 中的一个类, 它提供了高效的双端队列操作, 适用于需要快速插入和删除的场景。

## 特点

- 双端队列: 支持从两端进行高效的插入和删除操作。
- 性能:
  - 从两端插入和删除操作的时间复杂度为  $O(1)$ 。
  - 中间位置的访问和操作时间复杂度为  $O(n)$  (类似列表)。
- 灵活性: 既可用作栈 (后入先出) 也可用作队列 (先进先出)。

## deque 的常用方法

### 1. 初始化

```

from collections import deque

# 创建一个空的 deque
d = deque()

# 创建一个有初始元素的 deque
d = deque([1, 2, 3])

```

### 2. 插入操作

```

# 在右侧添加元素 (类似 append)
d.append(4) # d -> deque([1, 2, 3, 4])

# 在左侧添加元素
d.appendleft(0) # d -> deque([0, 1, 2, 3, 4])

```

### 3. 删除操作

```

# 从右侧移除元素 (类似 pop)
d.pop() # 返回 4, d -> deque([0, 1, 2, 3])

# 从左侧移除元素
d.popleft() # 返回 0, d -> deque([1, 2, 3])

```

### 4. 访问元素

```

# 获取 deque 的第一个元素 (不删除)
first = d[0] # 返回 1

# 获取 deque 的最后一个元素 (不删除)
last = d[-1] # 返回 3

```

### 5. 其他操作

```

# 获取 deque 的长度
length = len(d) # 返回 3

# 检查 deque 是否为空
is_empty = len(d) == 0 # 返回 False

# 反转 deque

```

```
d.reverse() # d -> deque([3, 2, 1])  
# 清空 deque  
d.clear() # d -> deque([])
```

---

## 用法场景

### 1. 队列

`deque` 可以用作先进先出的队列：

```
from collections import deque  
  
queue = deque()  
  
# 入队  
queue.append(1)  
queue.append(2)  
queue.append(3)  
  
# 出队  
print(queue.popleft()) # 输出 1  
print(queue.popleft()) # 输出 2
```

### 2. 栈

`deque` 可以用作后入先出的栈：

```
from collections import deque  
  
stack = deque()  
  
# 入栈  
stack.append(1)  
stack.append(2)  
stack.append(3)  
  
# 出栈  
print(stack.pop()) # 输出 3  
print(stack.pop()) # 输出 2
```

### 3. 滑动窗口

`deque` 可以高效处理固定大小的滑动窗口：

```
from collections import deque  
  
window = deque(maxlen=3) # 设置最大长度为 3  
window.append(1)  
window.append(2)  
window.append(3)  
print(window) # 输出 deque([1, 2, 3])  
  
window.append(4)  
print(window) # 输出 deque([2, 3, 4]), 自动移除最左端的元素
```

---

## 与列表的对比

特性	deque	list
左端插入和删除	$O(1)$	$O(n)$
右端插入和删除	$O(1)$	$O(1)$
任意位置访问	$O(n)$	$O(1)$ (随机访问)
内存效率	更高	较低

---

## 总结

- `deque` 的优势：
  - 适合需要频繁从两端插入或删除元素的场景。
  - 可以高效实现栈、队列和双端队列的操作。
- 适用场景：
  - 队列、栈、滑动窗口或需要双端操作的情况。
- 限制：
  - 不适合需要随机访问的场景，访问任意位置的复杂度为  $O(n)$ 。

`deque` 在实现栈和队列的操作时，性能优于列表，是标准库中非常实用的数据结构。

```
In [ ]: # 答案：用两个deque实现
```

```
from collections import deque
```

```

class MyStack:

    def __init__(self):
        """
        Python普通的Queue或SimpleQueue没有类似于peek的功能
        也无法用索引访问，在实现top的时候较为困难。
        用list可以，但是在使用pop(0)的时候时间复杂度为O(n)
        因此这里使用双向队列，我们保证只执行popleft()和append()，因为deque可以用索引访问，可以实现和peek相似的功能
        in - 存所有数据
        out - 仅在pop的时候会用到
        """
        self.queue_in = deque()
        self.queue_out = deque()

    def push(self, x:int) -> None:
        # 把新元素放到队列最后
        self.queue_in.append(x)

    def pop(self) -> int:
        """
        输出队列最后一个
        1. 首先确认不空
        2. 因为队列的特殊性，FIFO，所以我们只有在pop()的时候才会使用queue_out
        3. 先把queue_in中的所有元素（除了最后一个），依次出列放进queue_out
        4. 交换in和out，此时out里只有一个元素
        5. 把out中的pop出来，即是原队列的最后一个
        tip: 这不能像栈实现队列一样，因为另一个queue也是FIFO，如果执行pop()它不能像
        stack一样从另一个pop()，所以干脆in只用来存数据，pop()的时候两个进行交换
        """
        if self.empty():
            return None

        # 这是一种反转操作
        for i in range(len(self.queue_in) - 1): # 除了最后一个都移到queue_out，剩下的是栈顶元素
            self.queue_out.append(self.queue_in.popleft()) # 只能用popleft模拟队列 FIFO

        # 然后交换in和out栈，输出栈顶元素
        self.queue_in, self.queue_out = self.queue_out, self.queue_in

        return self.queue_out.popleft()

    def top(self) -> int:
        # 获取栈顶元素，重新利用pop，最后加进top
        if self.empty():
            return None

        # 利用pop获取栈顶元素
        top_element = self.pop()

        # 将栈顶元素重新放回 queue_in
        self.queue_in.append(top_element)

        return top_element

    def empty(self) -> bool:
        return len(self.queue_in) == 0

```

## 为什么需要交换两个队列？

### 1. 队列的 FIFO 特性：

- 队列的特点是先进先出，而栈需要后入先出。
- 因此，每次 pop 时需要先将 queue\_in 的所有元素（除了最后一个）依次移动到 queue\_out，最后剩下的那个元素就是栈顶元素。

### 2. 逻辑简化：

- 每次 pop 或 top 后，queue\_in 和 queue\_out 的角色发生了转换：
  - queue\_out 中剩下的元素相当于新栈内容。
  - 为了下一次操作能够继续使用 queue\_in，需要交换两个队列。
  - 通过交换，不需要频繁拷贝数据或使用额外的标记来区分两个队列的状态。

### 3. 操作复用：

- 通过交换两个队列的角色，避免了将所有元素从 queue\_out 再移回 queue\_in。
- 每次操作后，queue\_in 始终存储栈的内容，方便后续的 push 操作。

只用一个队列实现栈，可以通过每次插入新元素时调整队列中的顺序，使得新插入的元素总是位于队列的最前面，从而模拟栈的“后入先出”（LIFO）特性。

- push(x)：将新元素加入队列末尾，然后将队列中其他所有元素重新移动到队列末尾。
- pop()：直接从队列的前端弹出元素。
- top()：直接获取队列前端的元素。
- empty()：检查队列是否为空。

```
In [ ]: from collections import deque
class MyStack:
```

```

def __init__(self):
    self.queue = deque() # 单个队列

def push(self, x: int) -> None:
    """
    将元素压入栈顶
    """
    self.queue.append(x)
    # 这步最重要！！队列中除新插入的元素外的所有元素依次移到队列末尾
    for _ in range(len(self.queue) - 1):
        self.queue.append(self.queue.popleft())

def pop(self) -> int:
    """
    移除并返回栈顶元素
    """
    if self.empty():
        return None
    return self.queue.popleft()

def top(self) -> int:
    """
    获取栈顶元素
    """
    if self.empty():
        return None
    return self.queue[0]

def empty(self) -> bool:
    """
    判断栈是否为空
    """
    return len(self.queue) == 0

```

## 20. 有效的括号

### 题目描述

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

有效字符串需满足:

1. 左括号必须用相同类型的右括号闭合。
2. 左括号必须以正确的顺序闭合。
3. 每个右括号都有一个对应的相同类型的左括号。

### 示例

示例 1:

输入:  $s = "()"$   
输出: true

示例 2:

输入:  $s = "()[]{}"$   
输出: true

示例 3:

输入:  $s = "[]"$   
输出: false

示例 4:

输入:  $s = "([])"$   
输出: true

### 提示

- $1 \leq s.length \leq 10^4$
- $s$  仅由括号 `()`、`{}` 和 `[]` 组成。

### 解题思路

1. 使用栈来模拟括号匹配过程:
  - 每遇到一个左括号, 压入栈中。
  - 每遇到一个右括号, 检查栈顶是否是对应的左括号, 若匹配则弹出栈顶, 否则字符串无效。
2. 遍历结束后:
  - 栈为空, 则字符串有效;

- 栈非空，则字符串无效。

```
In [ ]: # 用stack

from collections import deque

class Solution:
    def isValid(self, s: str) -> bool:
        # 用stack来存储
        # [ , ), ], } ]
        stack = deque()
        for sign in s:
            if sign == '(':
                stack.append(")")
            elif sign == '{':
                stack.append("}")
            elif sign == '[':
                stack.append("]")
            if sign in '}])':
                # stack.pop() # pop不能有parameter.
                # 如果栈为空或栈顶元素不匹配，返回 False
                if not stack or stack.pop() != sign: # 不能pop两次! ! 在判断中我们已经pop了一次
                    return False

        return len(stack) == 0

# count奇偶不行，因为有括号顺序的限制
```

```
In [ ]: # 方法二，使用字典
# 用字典map的方式逻辑一样，但是更简洁清晰！

class Solution:
    def isValid(self, s: str) -> bool:
        stack = []
        mapping = {
            '(': ')',
            '[': ']',
            '{': '}'
        }
        for item in s:
            if item in mapping.keys():
                stack.append(mapping[item])
            elif not stack or stack[-1] != item:
                return False
            else:
                stack.pop()
        return True if not stack else False
```

## 1047. 删除字符串中的所有相邻重复项

### 题目描述

给出由小写字母组成的字符串  $s$ ，重复项删除操作会选择两个相邻且相同的字母，并删除它们。

在  $s$  上反复执行重复项删除操作，直到无法继续删除。

在完成所有重复项删除操作后返回最终的字符串。答案保证唯一。

### 示例

输入:

```
s = "abbaca"
```

输出:

```
"ca"
```

解释:

- 在字符串 "abbaca" 中，可以删除 "bb"，由于两字母相邻且相同，这是此时唯一可以执行删除操作的重复项。
- 之后得到字符串 "aaca"，其中又只有 "aa" 可以执行重复项删除操作，所以最后的字符串为 "ca"。

```
In [ ]: # 用栈非常巧妙，因为他模拟了最终结果

from collections import deque

class Solution:
    def removeDuplicates(self, s: str) -> str:
        stack = deque()
        for l in s:
            if len(stack) > 0:
                if stack[-1] == l:
                    stack.pop()
                else:
```

```
        stack.append(l)
else:
    stack.append(l)

string = "".join(stack)
return string
```

知识点：

- stack.length没有
- len(stack)可以
- stack[-1]可以，但在用之前要确保stack不为空
- string = "".join(stack) 也可以用

# 150. 逆波兰表达式求值

## 题目描述

给你一个字符串数组 *tokens*, 表示一个根据逆波兰表示法表示的算术表达式。

请你计算该表达式, 返回一个表示表达式值的整数。

## 注意

- 有效的算符为 +、-、\* 和 /。
- 每个操作数（运算对象）都可以是一个整数或者另一个表达式。
- 两个整数之间的除法总是向零截断。
- 表达式中不含除零运算。
- 输入是一个根据逆波兰表示法表示的算术表达式。
- 答案及所有中间计算结果可以用 32 位整数表示。

## 示例

示例 1:

输入: tokens = ["2","1","+", "3", "\*"]  
输出: 9  
解释: 该算式转化为常见的中缀算术表达式为: ((2 + 1) \* 3) = 9

示例 2:

输入: tokens = ["4", "13", "5", "/", "+"]  
输出: 6  
解释: 该算式转化为常见的中缀算术表达式为: (4 + (13 / 5)) = 6

示例 3:

输入: tokens = ["10", "6", "9", "3", "+", "-11", "\*", "/", "\*", "17", "+", "5", "+"]  
输出: 22  
解释:  
- 该算式转化为常见的中缀算术表达式为:  
\$((10 \* (6 / ((9 + 3) \* -11))) + 17) + 5\$  
- 逐步计算:  
\$((10 \* (6 / (12 \* -11))) + 17) + 5\$  
\$((10 \* (6 / -132)) + 17) + 5\$  
\$((10 \* 0) + 17) + 5\$  
\$(0 + 17) + 5\$  
\$17 + 5 = 22\$

---

## 提示

- \$1 \leq \text{tokens.length} \leq 10^4\$  
- \$\text{tokens}[i]\$ 是一个算符 (+、-、\* 或 /), 或是在范围 [-200, 200] 内的一个整数。

---

## 什么是逆波兰表达式?

逆波兰表达式是一种\*\*后缀表达式\*\*, 所谓后缀就是指算符写在操作数的后面。

\*\*特点\*\*:

- 无需括号即可表示表达式优先级, 例如:
  - 中缀表达式: \$((1 + 2) \* (3 + 4))\$
  - 逆波兰表达式: \$(1 \ 2 \ + \ 3 \ 4 \ \* \ + \ \*)\$
- 适合用\*\*栈操作\*\*计算:
  - 遇到数字: 入栈;
  - 遇到算符: 从栈中弹出两个数字, 计算后将结果压回栈中。

```
In [ ]: # 第一次尝试
from collections import deque
from typing import List

class Solution:
    def evalRPN(self, tokens: List[str]) -> int:
        stack = deque()
        for token in tokens:
            if type(token) is int:
                stack.append(token)
            elif token in "+-*":
                if token == "+":
                    result = stack.pop() + stack.pop()
                elif token == "-":
                    result = stack.pop() - stack.pop()
```

```

        elif token == "*":
            result = stack.pop() * stack.pop()
        elif token == "/":
            result = stack.pop() / stack.pop()
            stack.append(result)
    else:
        raise TypeError

    return stack[-1]

```

你的代码有以下几个问题导致逻辑不正确：

1. 数字检测：

- 修正为使用 `token.lstrip('-').isdigit()` 检查是否为数字，因为输入的数字是字符串形式。

2. 除法逻辑：

- 使用 `int(a / b)` 或 `//` 进行向零截断。

3. 操作数顺序：

- 弹出两个操作数时，`b = stack.pop()` 是第二个操作数，`a = stack.pop()` 是第一个操作数。

返回栈顶元素：

使用 `stack.pop()` 而不是 `stack[-1]`，保证栈被正确清空。

除法运算：

改为 `int(num_2 / num_1)`，显式向零截断，避免符号不同导致的错误。

异常处理：

使用更具体的异常类型 (`ValueError`)，并提供详细的错误信息。

```
In [ ]: # 第二次尝试，虽然有些结果对了，但是对于复杂的输入还是错误

from collections import deque
from typing import List

class Solution:
    def evalRPN(self, tokens: List[str]) -> int:
        stack = deque()
        for token in tokens:
            # 如果是数字，直接压入栈（判断是否为数字）
            if token.lstrip('-').isdigit():
                stack.append(int(token))
            elif token in "+-*/":
                if token == "+":
                    result = stack.pop() + stack.pop()
                elif token == "-":
                    # 注意这里先弹出的是被减数，在弹出减数
                    num_1 = stack.pop()
                    num_2 = stack.pop()
                    result = num_2 - num_1
                elif token == "*":
                    result = stack.pop() * stack.pop()
                elif token == "/":
                    num_1 = stack.pop()
                    num_2 = stack.pop()
                    result = num_2 // num_1
                stack.append(result)
            else:
                raise TypeError

        return stack[-1]
```

```
In [ ]: # 第三次尝试，虽然有些结果对了，但是对于复杂的输入还是错误
# 这回对了

from collections import deque
from typing import List

class Solution:
    def evalRPN(self, tokens: List[str]) -> int:
        stack = deque()
        for token in tokens:
            # 如果是数字，直接压入栈（判断是否为数字）
            if token.lstrip('-').isdigit():
                stack.append(int(token))
            elif token in "+-*/":
                if token == "+":
                    result = stack.pop() + stack.pop()
                elif token == "-":
                    # 注意这里先弹出的是被减数，在弹出减数
                    num_1 = stack.pop()
                    num_2 = stack.pop()
                    result = num_2 - num_1
                elif token == "*":
                    result = stack.pop() * stack.pop()
                elif token == "/":
                    num_1 = stack.pop()
                    num_2 = stack.pop()
                    result = num_2 // num_1
                stack.append(result)
            else:
                raise TypeError

        return stack[-1]
```

```

        num_1 = stack.pop()
        num_2 = stack.pop()
        result = int(num_2 / num_1)
        stack.append(result)
    else:
        raise TypeError

    return stack.pop()

```

In [ ]: # 答案, 用了更高级的写法

```

from operator import add, sub, mul

def div(x, y):
    # 使用整数除法的向零取整方式
    return int(x / y) if x * y > 0 else -(abs(x) // abs(y))

class Solution(object):
    op_map = {'+': add, '-': sub, '*': mul, '/': div}

    def evalRPN(self, tokens: List[str]) -> int:
        stack = []
        for token in tokens:
            if token not in {'+', '-', '*', '/'}:
                stack.append(int(token))
            else:
                op2 = stack.pop()
                op1 = stack.pop()
                stack.append(self.op_map[token](op1, op2)) # 第一个出来的在运算符后面
        return stack.pop()

```

In [ ]: # 实际是三个双指针

```

class Solution:
    def trim_spaces(self, s):
        # 双指针
        left, right = 0, len(s) - 1
        while left <= right and s[left] == ' ':
            left += 1
        while left <= right and s[right] == ' ':
            right = right - 1 # 用改变指针的方法去掉空格?

        tmp = []
        # 处理单词中间的空格
        while left <= right:
            if s[left] != " ":
                tmp.append(s[left])
            elif not tmp or tmp[-1] != ' ': # 这里加入了一个判断, 如果 tmp 是空列表, tmp[-1] 会抛出 IndexError, 所以确保不为空
                tmp.append(s[left])
            left += 1
        return tmp

    def reverse_string(self, list, left, right):
        while left < right:
            # 从list的头和尾反转
            list[left], list[right] = list[right], list[left]
            left += 1
            right -= 1
        return None

    def reverse_each_word(self, word):
        left, right = 0, 0
        word_len = len(word)
        while left < word_len:
            while right < word_len and word[right] != ' ':
                right += 1
            self.reverse_string(word, left, right - 1)
            left = right + 1
            right = left # 重置 `right` 与 `left` 同步

    def reverseWords(self, s: str) -> str:
        letter_list = self.trim_spaces(s)
        self.reverse_string(letter_list, 0, len(letter_list) - 1)
        self.reverse_each_word(letter_list)
        return ''.join(letter_list)

```

## 239. 滑动窗口最大值

### 题目描述

给定一个整数数组  $nums$  和一个大小为  $k$  的滑动窗口，从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的  $k$  个数字。滑动窗口每次只向右移动一位。

返回滑动窗口中的最大值。

### 示例

### 示例 1:

输入:

```
nums = [1, 3, -1,  
       -3, 5, 3, 6, 7], k = 3
```

输出: [3, 3, 5, 5, 6, 7]

解释:

滑动窗口的位置	最大值
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3] 5] 3 6 7	5
1 3 -1 [-3 5] 3] 6 7	5
1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7

### 示例 2:

输入:  $nums = [1]$ ,  $k = 1$

输出: [1]

```
In [1]: # 第一次尝试, 问题如下  
from collections import deque  
from typing import List  
  
class Solution:  
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:  
        deq = deque()  
        max_list = []  
        # pre compute initial deq  
        for i in range(len(nums)-k):  
            if i < k:  
                deq.append(nums[i])  
            else:  
                max_list.append(max(deq))  
                deq.append(nums[i])  
                deq.popleft()  
  
        return max_list  
  
Cell In[1], line 8  
      for i in range(len(nums)): ^  
SyntaxError: incomplete input
```

你的代码有以下几个问题:

## 1. 滑动窗口未完全覆盖整个数组

- 问题:

```
for i in range(len(nums) - k):
```

- 这个循环的范围是 `len(nums) - k`, 导致滑动窗口未完全覆盖数组的末尾部分。
- 例如, 当  $nums = [1, 3, -1, -3, 5, 3, 6, 7]$  和  $k = 3$  时, 实际应该有 6 个滑动窗口, 但你的代码只计算了 5 个。

- 解决方法: 改为遍历整个数组:

```
for i in range(len(nums)):
```

## 2. 滑动窗口的初始化逻辑有误

- 问题:

```
if i < k:  
    deq.append(nums[i])
```

- 这里初始化了前  $k$  个元素的 `deq`, 但并没有在初始化阶段计算第一个窗口的最大值。
- 解决方法: 在遍历的过程中, 确保每次滑动窗口向右移动时计算当前窗口的最大值。

### 3. 未正确维护滑动窗口最大值

- 问题:
  - 每次直接调用 `max(deq)` 来获取当前窗口的最大值，这样的计算复杂度是  $O(k)$ ，导致整体复杂度为  $O(n \cdot k)$ ，在  $n$  和  $k$  都很大时性能较差。
- 解决方法：使用一个双端队列来存储当前窗口中元素的索引，并保持队列中索引对应的元素按值从大到小排列：
  - 保证队列头部总是当前窗口的最大值。

```
In [ ]: # 尝试2, 对了, 分三类情况, 但效率不高, 超时了
# 这个复杂度是 n*k

from collections import deque
from typing import List

class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        deq = deque()
        max_list = []
        # pre compute initial deq
        for i in range(len(nums)):
            if i < k - 1:
                deq.append(nums[i])
            elif i == k-1:
                deq.append(nums[i])
                max_list.append(max(deq))
            else:
                deq.append(nums[i])
                deq.popleft()
                max_list.append(max(deq))

        return max_list
```

### 单调队列

核心思想是 每次加入值的后，把最大值前面的所有值都弹出

这样`popleft`时弹出的就一定是最大值

如果加入的新值就是最大值，那么说明左边的元素（当前最大值）需要弹出了

```
In [ ]: from collections import deque
from typing import List

class MyQueue:
    def __init__(self):
        self.queue = deque()

    #每次弹出的时候, 比较当前要弹出的数值是否等于队列出口元素的数值, 如果相等则弹出。
    #同时pop之前判断队列当前是否为空。
    def pop(self, value):
        if self.queue and value == self.queue[0]:
            self.queue.popleft()

    #如果push的数值大于入口元素的数值, 那么就将队列后端的数值弹出, 直到push的数值小于等于队列入口元素的数值为止。
    #这样就保持了队列里的数值是单调从大到小的了。
    def push(self, value):
        while self.queue and value > self.queue[-1]:
            self.queue.pop()
        self.queue.append(value)

    def front(self):
        return self.queue[0]

class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        que = MyQueue()
        max_list = []
        # pre compute initial deq
        for i in range(len(nums)):
            if i < k - 1:
                que.push(nums[i])
            elif i == k-1:
                que.push(nums[i])
                max_list.append(que.front())
            else:
                que.pop(nums[i - k]) #滑动窗口移除最前面元素
                que.push(nums[i]) #滑动窗口前加入最后面的元素
                max_list.append(que.front())

        return max_list
```

### 347. 前 $k$ 个高频元素

## 题目描述

给你一个整数数组  $nums$  和一个整数  $k$ , 请你返回其中出现频率前  $k$  高的元素。你可以按任意顺序返回答案。

## 示例

示例 1:

输入:  $nums = [1,1,1,2,2,3]$ ,  $k = 2$   
输出:  $[1,2]$

示例 2:

输入:  $nums = [1]$ ,  $k = 1$   
输出:  $[1]$

## 提示

- $1 \leq \text{nums.length} \leq 10^5$
- $k$  的取值范围是  $[1, \text{数组中不相同的元素的个数}]$
- 题目数据保证答案唯一, 换句话说, 数组中前  $k$  个高频元素的集合是唯一的。

In [7]: # 第一次尝试, 用 deque 模拟 min heap

```
from collections import deque
from typing import List
from collections import Counter

# 先统计一遍 元素: count, 用counter
# 然后用 min heap 维护 k 个元素, 每次加进一个元素, pop出最小的 (堆顶)

class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        freq = Counter(nums)
        min_heap = MinHeap(k)

        for key, value in freq.items():
            min_heap.push(value)

        return min_heap.que

class MinHeap:
    def __init__(self, k):
        self.que = deque()
        self.k = k
    def push(self, num):
        if len(self.que) == 0:
            self.que.append(num)
        elif len(self.que) < self.k:
            if num >= self.que[0]:
                self.que.appendleft(num)
            else:
                self.que.append(num)
        # else, evaluate the input num and only keep the highest k values
        else:
            if num >= self.que[0]:
                self.que.appendleft(num)
                self.que.pop()
            elif num > self.que[-1]:
                self.que.pop()
                self.que.append(num)
```

```
AssertionError                                     Traceback (most recent call last)
Cell In[7], line 74
  71     print("所有测试用例通过! ")
  73 # 执行测试
--> 74 test_solution()

Cell In[7], line 69, in test_solution()
  67     result = solution.topKFrequent(nums, k)
  68     # 验证结果是否符合预期
--> 69     assert sorted(result) == sorted(expected), f"Test case {idx + 1} failed: {result} != {expected}"
  71 print("所有测试用例通过! ")

AssertionError: Test case 1 failed: deque([3, 2]) != [1, 2]
```

In [16]: # 第二次尝试, 用 deque 模拟 min heap, 修正最开始应该是保留最小值

```
# 这个方法在插入时需要遍历一遍queue, 效率比较低
```

```

from collections import deque
from typing import List
from collections import Counter

# 先统计一遍 元素: count, 用counter
# 然后用 min heap 维护 k 个元素, 每次加进一个元素, pop出最小的 (堆顶)

class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        freq = Counter(nums)
        min_heap = MinHeap(k)

        for key, value in freq.items():
            min_heap.push((value, key))

        return [num for _, num in min_heap.que]
    
```

```

class MinHeap:
    def __init__(self, k):
        self.que = deque() # 使用 deque 存储堆
        self.k = k

    def push(self, freq_elem):
        """freq_elem 是一个元组 (频率, 元素)"""
        freq, num = freq_elem

        # 如果堆未满, 直接插入到合适的位置
        if len(self.que) < self.k:
            self._insert(freq_elem)
        # 如果堆已满, 只在当前频率大于最小值时插入
        elif freq > self.que[0][0]:
            self.que.popleft() # 移除堆顶最小值
            self._insert(freq_elem)

    def _insert(self, freq_elem):
        """按升序插入 freq_elem (频率, 元素)"""
        freq, _ = freq_elem

        # 遍历查找插入位置
        for i in range(len(self.que)):
            if freq < self.que[i][0]: # 比较频率, 找到比当前值大的位置插入
                self.que.insert(i, freq_elem)
                return
        # 如果未找到插入位置, 直接添加到队尾
        self.que.append(freq_elem)
    
```

## 复杂度分析

- 时间复杂度:

- 统计频率:  $O(n)$ , 其中  $n$  是数组长度。
- 堆操作: 每个元素插入堆的时间复杂度为  $O(k)$ , 最多插入  $m$  个元素 ( $m$  是数组中不同的元素数)。
- 总复杂度:  $O(n + m \cdot k)$ 。

- 空间复杂度:

- 使用 `deque` 存储堆:  $O(k)$ 。
- 使用 `Counter` 存储频率:  $O(m)$ 。
- 总空间复杂度:  $O(m + k)$ 。

```

In [ ]: # 学习答案, 用heapq

#时间复杂度: O(nlogk)
#空间复杂度: O(n)
import heapq
class Solution:

    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        #要统计元素出现频率
        map_ = {} #nums[i]: 对应出现的次数
        for i in range(len(nums)):
            map_[nums[i]] = map_.get(nums[i], 0) + 1

        #对频率排序
        #定义一个小顶堆, 大小为k
        pri_que = [] #小顶堆

        #用固定大小为k的小顶堆, 扫描所有频率的数值
        for key, freq in map_.items():
            heapq.heappush(pri_que, (freq, key))
            if len(pri_que) > k: #如果堆的大小大于了K, 则队列弹出, 保证堆的大小一直为k
                heapq.heappop(pri_que)

        #找出前K个高频元素, 因为小顶堆先弹出的是最小的, 所以倒序来输出到数组
        result = [0] * k
        for i in range(k-1, -1, -1):
            result[i] = heapq.heappop(pri_que)[1]
        return result
    
```

```
In [22]: # 第三次尝试, 用heapq直接实现最小堆, 直接存储 (value, key) pair, 并按照 (value, key) 中value排序
# 23ms -> 7ms, 效率提升很多

import heapq
from collections import Counter
from typing import List
# heapq也可以兼容tuple, 但是默认按照 tuple 的 index 0 排序, heapq没有直接支持对元组第二个元素排序的选项
class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        freq = Counter(nums)
        min_heap = []
        for key, value in freq.items():
            heapq.heappush(min_heap, (value, key)) # 注意这里频率在前
            if len(min_heap) > k:
                heapq.heappop(min_heap)
        return [key for _, key in min_heap]
```

```
In [24]: def test_solution():
    solution = Solution()

    # 测试用例列表
    test_cases = [
        # 示例测试用例
        {"nums": [1, 1, 1, 2, 2, 3], "k": 2, "expected": [1, 2]},
        {"nums": [1], "k": 1, "expected": [1]},

        # 边界测试用例
        {"nums": [4, 4, 4, 4], "k": 1, "expected": [4]},
        {"nums": [1, 2, 3, 4, 5], "k": 3, "expected": [3, 4, 5]}, # 修正后

        # 一般测试用例
        {"nums": [1, 2, 2, 3, 3, 3, 4, 4, 4, 4], "k": 2, "expected": [4, 3]},
        {"nums": [5, 5, 5, 1, 1, 2, 2, 3], "k": 3, "expected": [5, 1, 2]},
        {"nums": [8, 8, 8, 8, 7, 7, 7, 6], "k": 2, "expected": [8, 7]},
    ]

    # 遍历测试用例
    for idx, case in enumerate(test_cases):
        nums, k, expected = case["nums"], case["k"], case["expected"]
        # 获取结果
        result = solution.topKFrequent(nums, k)
        # 验证结果是否符合预期
        assert sorted(result) == sorted(expected), f"Test case {idx + 1} failed: {result} != {expected}"

    print("所有测试用例通过! ")

    # 执行测试
    test_solution()
```

所有测试用例通过!

几个错误:

如果尽量保留你原来的代码逻辑, 用 `deque` 实现, 你需要修复以下问题:

### 问题 1: MinHeap 没有维护正确的堆结构

当前的 `MinHeap` 类只是简单操作 `deque` 的两端, 没有维护堆的性质。`deque` 并不是堆, 无法直接通过添加和删除来保持堆的最小堆性质。

修复方法:

- 按照最小堆的定义, 每次插入一个新元素时, 确保 `deque` 中的元素保持升序 (最小值在队首)。
- 替换元素时, 始终移除最小值 (`deque` 的左端)。

### 问题 2: MinHeap 只存储频率, 没有关联元素

在 `MinHeap.push()` 方法中, 你只存储了频率 `value`, 没有记录频率对应的元素 `key`, 导致结果返回的是频率, 而不是实际的数字。

修复方法:

- 修改 `MinHeap`, 在 `deque` 中存储 (频率, 元素) 的元组。

In [ ]:

## Python Count 用法

在 Python 中, `count` 方法用于统计字符串、列表或其他可迭代对象中某个元素出现的次数。其使用场景主要有两种: 字符串和列表。

### 1. str.count

用于统计一个子字符串在字符串中出现的次数。

### 语法

```
str.count(sub[, start[, end]])
```

- `sub` : 要统计的子字符串。
- `start` (可选) : 指定开始统计的位置 (包含)。
- `end` (可选) : 指定结束统计的位置 (不包含)。

### 示例

```
# 示例 1: 基本用法
s = "hello world"
print(s.count("o")) # 输出: 2

# 示例 2: 指定范围
print(s.count("l", 3, 8)) # 输出: 2 (统计索引 3 到 7 的部分)

# 示例 3: 子字符串不存在
print(s.count("x")) # 输出: 0
```

---

## 2. list.count

用于统计列表中某个元素出现的次数。

### 语法

```
list.count(element)
```

- `element` : 要统计的目标元素。

### 示例

```
# 示例 1: 基本用法
nums = [1, 2, 2, 3, 4, 2]
print(nums.count(2)) # 输出: 3

# 示例 2: 元素不存在
print(nums.count(5)) # 输出: 0

# 示例 3: 统计字符串元素
words = ["apple", "banana", "apple", "cherry"]
print(words.count("apple")) # 输出: 2
```

---

## 3. 其他常见用法场景

### (1) 用于统计字符或单词出现的次数

```
text = "Python is great and Python is fun"
print(text.count("Python")) # 输出: 2
```

### (2) 判断元素是否存在

虽然 `count` 可以统计元素是否出现, 但 `in` 操作通常更高效。

```
nums = [1, 2, 3, 4]
# 判断是否存在某元素
print(nums.count(5) > 0) # 输出: False
```

### (3) 结合其他方法统计复杂条件

通过 `list.count` 或 `str.count` 与 `filter` 或列表解析结合使用:

```
nums = [1, 2, 2, 3, 4, 2]
# 统计列表中大于 2 的元素数量
greater_than_two_count = sum(1 for x in nums if x > 2)
print(greater_than_two_count) # 输出: 2
```

---

## 注意事项

### 1. `count` 的性能:

- 对于较大的字符串或列表, `count` 方法的性能较好, 但如果需要更复杂的统计条件, 可以考虑其他方法。
- 2. 区分大小写:

- 对于字符串, `count` 是区分大小写的:

```
s = "Python python PYTHON"
print(s.count("Python")) # 输出: 1
```

### 3. 统计子字符串时的重叠问题:

- `str.count` 不会统计重叠的子字符串:

```
s = "aaaa"  
print(s.count("aa")) # 输出: 2
```

In [ ]:

## Python 中 counter的用法

用最高效的方式输出一个统计数列中数字出现次数的字典

在 Python 中，统计数列中数字出现次数的最高效方式是使用 `collections.Counter`，因为它在底层实现中对性能进行了优化。

### 推荐方法：使用 `collections.Counter`

代码实现

```
from collections import Counter  
  
# 示例数据  
nums = [1, 2, 2, 3, 4, 2, 3, 1, 4, 4]  
  
# 使用 Counter  
frequency = Counter(nums)  
  
print(frequency) # 输出: Counter({2: 3, 4: 3, 1: 2, 3: 2})
```

输出结果

`Counter` 是一个字典的子类，其键是列表中的元素，值是元素出现的次数：

```
Counter({2: 3, 4: 3, 1: 2, 3: 2})  
你可以像操作普通字典一样访问数据：  
  
print(frequency[2]) # 输出: 3  
print(dict(frequency)) # 转换为普通字典: {1: 2, 2: 3, 3: 2, 4: 3}
```

## heapq 最小堆，最大堆

`heapq` 是 Python 标准库中的一个模块，用于操作堆数据结构 (heap)。堆是一种特殊的完全二叉树，最常见的是最小堆，其中每个父节点都小于或等于其子节点。

Python 的 `heapq` 默认实现最小堆，可以在  $O(\log n)$  时间复杂度内完成插入和删除操作。

### 基本用法

#### 1. 导入模块

```
import heapq
```

#### 2. 最小堆基本操作

操作	方法	描述
插入元素	<code>heapq.heappush(heap, x)</code>	将元素 <code>x</code> 插入到堆中。
弹出最小值	<code>heapq.heappop(heap)</code>	移除并返回堆中的最小值。
查看最小值但不移除	<code>heap[0]</code>	直接访问堆顶元素（最小值）。
弹出并插入新元素（替换）	<code>heapq.heapreplace(heap, x)</code>	移除堆中最小值，并插入新元素 <code>x</code> 。
创建堆	<code>heapq.heapify(iterable)</code>	将列表转换为堆，原地操作，时间复杂度 $O(n)$ 。
获取前 $k$ 个最小值	<code>heapq.nsmallest(k, heap)</code>	返回堆中前 $k$ 个最小值。
获取前 $k$	<code>heapq.nlargest(k, heap)</code>	返回堆中前 $k$ 个最大值。

操作	方法	描述
个最大值		

## 示例代码

### 最小堆示例

```
import heapq

# 创建一个空堆
heap = []

# 插入元素
heapq.heappush(heap, 10)
heapq.heappush(heap, 1)
heapq.heappush(heap, 5)

print(heap) # 输出: [1, 10, 5] (最小值在堆顶, 顺序可能未排序)

# 弹出最小值
min_value = heapq.heappop(heap)
print(min_value) # 输出: 1

# 查看堆顶最小值 (不移除)
print(heap[0]) # 输出: 5
```

### 使用 `heapify` 创建堆

```
import heapq

# 从列表创建堆
nums = [4, 10, 3, 5, 1]
heapq.heapify(nums)

print(nums) # 输出: [1, 4, 3, 5, 10] (最小值在堆顶)
```

### 替换堆顶元素

```
import heapq

# 创建一个堆
nums = [1, 3, 5, 7, 9]
heapq.heapify(nums)

# 替换堆顶元素
heapq.heapreplace(nums, 2)

print(nums) # 输出: [2, 3, 5, 7, 9]
```

### 获取前 $k$ 个最小值

```
import heapq

nums = [1, 8, 3, 9, 4, 2, 7]
k = 3

# 前 3 个最小值
print(heapq.nsmallest(k, nums)) # 输出: [1, 2, 3]

# 前 3 个最大值
print(heapq.nlargest(k, nums)) # 输出: [9, 8, 7]
```

## 优点和复杂度

### 1. 时间复杂度:

- 插入:  $O(\log n)$
- 弹出最小值:  $O(\log n)$
- 创建堆:  $O(n)$

### 2. 空间复杂度:

- 堆占用的空间为  $O(n)$ 。

## 应用场景

### 1. 实时获取最小值或最大值:

- 动态维护数据流中的最小值/最大值。

### 2. 解决 Top-K 问题:

- 找到数组中前  $k$  个最小值/最大值。
3. 优先队列：
- 模拟任务调度等需要动态优先级的数据结构。

如果有更多问题，可以进一步讨论！😊

In [ ]: 在 Python 中，`heapq` 默认是最小堆。如果需要\*\*最大堆\*\*，可以通过以下两种方式实现：

---

### \*\*1. 使用取反数实现最大堆\*\*

这是 `heapq` 模拟最大堆的常见方法。通过存储元素的相反数，保持最小堆的行为，但逻辑上实现了最大堆。

#### 示例代码

```python

`import heapq`

`nums = [1, 8, 3, 9, 4, 2, 7]`

`# 创建最大堆 (取反数)`

`max_heap = [-num for num in nums]`

`heapq.heapify(max_heap)`

`# 弹出最大值`

`max_value = -heapq.heappop(max_heap)`

`print(max_value) # 输出: 9`

`# 插入新值`

`heapq.heappush(max_heap, -10)`

`print(-max_heap[0]) # 输出: 10 (堆顶最大值)`

#### 优点

- 简单高效。

- `heapq` 本身无需修改。

---

### \*\*2. 自定义比较函数 (推荐使用 `PriorityQueue`)\*\*

如果你希望避免取反数，可以使用 `queue.PriorityQueue` 或 `functools` 提供的 `key` 参数来控制排序行为。`PriorityQueue` 内部也是基于 `heapq` 实现的。

#### 示例代码

```python

`import heapq`

`class MaxHeap:`

`def __init__(self):`  
 `self.heap = []`

`def push(self, val):`

`# 插入元素时取反，保持最大堆性质`  
 `heapq.heappush(self.heap, -val)`

`def pop(self):`

`# 弹出时还原原始值`

`return -heapq.heappop(self.heap)`

`def peek(self):`

`# 查看堆顶值`

`return -self.heap[0]`

`def size(self):`

`return len(self.heap)`

```

#### 使用示例

```python

`max_heap = MaxHeap()`

`# 插入元素`

`max_heap.push(1)`

`max_heap.push(8)`

`max_heap.push(3)`

`print(max_heap.peek()) # 输出: 8 (最大值)`

`# 弹出最大值`

`print(max_heap.pop()) # 输出: 8`

`print(max_heap.pop()) # 输出: 3`

`print(max_heap.pop()) # 输出: 1`

---

### \*\*3. 使用 `PriorityQueue`\*\*

`queue.PriorityQueue` 是线程安全的堆实现，但其性能可能比直接使用 `heapq` 略低。

#### 示例代码

```python

```
from queue import PriorityQueue

pq = PriorityQueue()

# 插入元素 (取反数以实现最大堆)
pq.put((-1, "Task 1"))
pq.put((-8, "Task 8"))
pq.put((-3, "Task 3"))

# 弹出最大值 (还原值)
while not pq.empty():
    priority, task = pq.get()
    print(-priority, task)
```

#### 输出
```
8 Task 8
3 Task 3
1 Task 1
```

---

### **4. 使用 `functools` 自定义排序**

使用 `heapq` 操作带权值的元组 (`(优先级, 值)`) 来实现最大堆。

#### 示例代码
```python
import heapq
from functools import cmp_to_key

nums = [1, 8, 3, 9, 4, 2, 7]

# 自定义最大堆
max_heap = [(num, num) for num in nums] # 元组存储值
heapq.heapify(max_heap)

# 弹出最大值
print(max_heap[0])
```

#### 总结
选择：
- 小规模使用：**推荐第一种**，通过取反数简单模拟最大堆。
- 线程安全且有复杂优先级队列：**推荐第三种**
```

## 二叉树的层序遍历 (BFS)

核心思想，遍历到每一层，

每一个节点的孩子用队列记录，先进先出

然后把每一层的下一层节点数量记录下来

每push到队列一次，节点数量就 -1

如果下一层节点数量归零，就意味着这层遍历结束，

进入到下一层

```
In [ ]: # 长度法

from typing import List, Optional
from collections import deque

# 定义二叉树的节点类
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        # 如果树是空的，直接返回空列表
        if not root:
            return []

        # 初始化队列，用于存储每一层的节点
        queue = deque([root])
        # 用于存储最终的层次遍历结果
        result = []

        # 当队列不为空时，说明还有未遍历的节点
        while queue:
            # 当前层的节点值
            current_level = []

            # 遍历当前层的所有节点
            for _ in range(len(queue)):  # len(queue) 是当前层的节点数
                # 取出队首节点
                node = queue.popleft()
                # 将当前节点的值添加到当前层
                current_level.append(node.val)

                # 如果有左子节点，加入队列
                if node.left:
                    queue.append(node.left)
                # 如果有右子节点，加入队列
                if node.right:
                    queue.append(node.right)

            # 当前层结束后，将结果添加到最终结果中
            result.append(current_level)

        return result
;
```

```
In [ ]: class Solution:
    def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        if not root:
            return []

        levels = []

        def traverse(node, level):
            if not node:
                return

            if len(levels) == level:
                levels.append([])

            levels[level].append(node.val)
            traverse(node.left, level + 1)
            traverse(node.right, level + 1)

        traverse(root, 0)
        return levels
```

层序遍历 (反转方向)：最后反转一下结果数列 [::-1]

```
In [ ]: # while记录每层大小法
```

```

from collections import deque

class Solution:
    def levelOrderBottom(self, root: TreeNode) -> List[List[int]]:
        if not root:
            return []
        queue = deque([root])
        result = []

        while queue:
            level = []
            for _ in range(len(queue)):
                cur = queue.popleft()
                if cur.left:
                    level.append(cur.left)
                if cur.right:
                    level.append(cur.right)

            result.append(level)
        return result[::-1]

```

层序遍历(右视图): 每层判断一下是不是最后一个

```

In [ ]: # Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def rightSideView(self, root: Optional[TreeNode]) -> List[int]:
        if not root:
            return []

        queue = deque([root])
        result = []

        while queue:
            level_size = len(queue)
            for i in range(level_size):
                node = queue.popleft()
                if i == level_size - 1:
                    result.append(node.val)

                # 注意要加这个, 如果不加, 之后queue就空了
                # 注意这个循环在for loop内部
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)

        return result

```

## 637.二叉树的层平均值

```

In [1]: from collections import deque

class Solution:
    def averageOfLevels(self, root: TreeNode) -> List[List[int]]:

        if not root:
            return []
        queue = deque([root])
        result = []

        while queue:
            level = []
            for _ in range(len(queue)):
                node = queue.popleft()
                level.append(node.val)
                # 注意要加这个, 如果不加, 之后queue就空了
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)

            result.append(sum(level)/len(level))

        return result

```

```

NameError                                 Traceback (most recent call last)
Cell In[1], line 3
    1 from collections import deque
----> 3 class Solution:
    4     def averageOfLevels(self, root: TreeNode) -> List[List[int]]:
    5         if not root:
    6             return []

Cell In[1], line 4, in Solution()
  3 class Solution:
----> 4     def averageOfLevels(self, root: TreeNode) -> List[List[int]]:
  5         if not root:
  6             return []

NameError: name 'TreeNode' is not defined

```

## 429.N叉树的层序遍历

```

In [ ]: class Solution:
    def levelOrder(self, root: 'Node') -> List[List[int]]:
        if not root:
            return []

        result = []
        queue = collections.deque([root])

        while queue:
            level_size = len(queue)
            level = []

            for _ in range(level_size):
                node = queue.popleft()
                level.append(node.val)

                # 就是这里有不同，遍历了所有child，而不是左右了
                for child in node.children:
                    queue.append(child)

            result.append(level)

        return result

Running cells with 'Python 3.13.1' requires the ipykernel package.

Run the following command to install 'ipykernel' into the Python environment.

Command: 'c:/Python313/python.exe -m pip install ipykernel -U --user --force-reinstall'

```

```

In [ ]: # LeetCode 429. N-ary Tree Level Order Traversal
# 递归法
class Solution:
    def levelOrder(self, root: 'Node') -> List[List[int]]:
        if not root: return []
        result=[]
        def traversal(root,depth):
            if len(result)==depth:result.append([])
            result[depth].append(root.val)
            if root.children:
                for i in range(len(root.children)):
                    traversal(root.children[i],depth+1)

        traversal(root,0)
        return result

```

## 515.在每个树行中找最大值

就是每行记录一下最大值

```

In [ ]: # Definition for a binary tree node.

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

from collections import deque
from typing import Optional, List

class Solution:
    def largestValues(self, root: Optional[TreeNode]) -> List[int]:
        if not root:
            return []
        queue = deque([root])
        result = []

        while queue:
            level = []
            for _ in range(len(queue)):
                node = queue.popleft()
                level.append(node.val)
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)

            result.append(max(level))

        return result

```

```

        node = queue.popleft()
        level.append(node.val)
        # 注意要加这个, 如果不加, 之后queue就空了
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)

    result.append(max(level))

    return result

```

## 116.填充每个节点的下一个右侧节点指针

## 117.填充每个节点的下一个右侧节点指针II

In [12]: # 第一次尝试

```

"""
# Definition for a Node.
class Node:
    def __init__(self, val: int = 0, left: 'Node' = None, right: 'Node' = None, next: 'Node' = None):
        self.val = val
        self.left = left
        self.right = right
        self.next = next
"""

class Solution:
    def connect(self, root: 'Optional[Node]') -> 'Optional[Node]':
        if not root:
            return root
        queue = deque([root])

        while queue:
            level = []
            queue_len = len(queue)
            for i in range(queue_len):
                node = queue.popleft()
                level.append(node)

                if len(level) > 1:
                    level[-2].next = node

                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)

            level[-1].next = None

        return root

```

In [ ]: # 更简洁的逻辑

```

class Solution:
    def connect(self, root: 'Node') -> 'Node':
        if not root:
            return root

        queue = collections.deque([root])

        while queue:
            level_size = len(queue)
            prev = None

            for i in range(level_size):
                node = queue.popleft()

                if prev:
                    prev.next = node

                prev = node

                if node.left:
                    queue.append(node.left)

                if node.right:
                    queue.append(node.right)

        return root

```

In [ ]: from collections import deque

```

class Solution:
    def connect(self, root: 'Node') -> 'Node':
        if not root:
            return root

```

```

queue = collections.deque([root])

while queue:
    level_size = len(queue)
    prev = None

    for i in range(level_size):
        node = queue.popleft()

        # 这是很重要的一个范式
        if prev:
            prev.next = node

        prev = node

        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)

    return root

```

In [ ]:

## 104.二叉树的最大深度

```

In [ ]: class Solution:
    def connect(self, root: 'Node') -> 'Node':
        if not root:
            return 0

        queue = collections.deque([root])
        level_n = 0
        while queue:
            level_size = len(queue)
            prev = None

            for i in range(level_size):
                node = queue.popleft()

                if node.left:
                    queue.append(node.left)

                if node.right:
                    queue.append(node.right)

            level_n += 1

        return level_n

```

## 111.二叉树的最小深度

只有当左右孩子都为空的时候，才说明遍历的最低点了

```

In [ ]: # Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def minDepth(self, root: Optional[TreeNode]) -> int:
        if not root:
            return 0

        queue = collections.deque([root])
        level_n = 0
        while queue:
            level_size = len(queue)
            prev = None

            for i in range(level_size):
                node = queue.popleft()

                if node.left:
                    queue.append(node.left)
                    level_n += 1

                elif node.right:
                    queue.append(node.right)
                    level_n += 1

                else:
                    return level_n + 1

        return level_n

```

## 226. 翻转二叉树

### 问题描述

给你一棵二叉树的根节点 `root`，翻转这棵二叉树，并返回其根节点。

### 示例

#### 示例 1

输入：

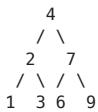
```
root = [4, 2, 7, 1, 3, 6,  
         9]
```

输出：

```
[4, 7, 2, 9, 6, 3, 1]
```

树的结构如下：

- 输入：



- 输出：



#### 示例 2

输入：

```
root = [2, 1, 3]
```

输出：

```
[2, 3, 1]
```

树的结构如下：

- 输入：



- 输出：



#### 示例 3

输入：

```
root = []
```

输出：

```
[]
```

### 提示

- 树中节点数目范围在  $[0, 100]$  内。
- 每个节点的值范围为  
 $-100 \leq \text{Node.val} \leq 100$

```

。

```

In [ ]:

注意只要把每一个节点的左右孩子翻转一下，就可以达到整体翻转的效果

这道题目使用前序遍历和后序遍历都可以，唯独中序遍历不方便，因为中序遍历会把某些节点的左右孩子翻转了两次！

In [2]: # 递归  
`# Definition for a binary tree node.  
class TreeNode:  
 def __init__(self, val=0, left=None, right=None):  
 self.val = val  
 self.left = left  
 self.right = right  
  
 class Solution:  
 def invertTree(self, node: TreeNode) -> TreeNode:  
 if not node:  
 return None  
 # 交换左右子树  
 node.left, node.right = node.right, node.left  
 # 递归地翻转子树  
 self.invertTree(node.left)  
 self.invertTree(node.right)  
 return node`

In [4]: # 递归。后序遍历  
`class Solution:  
 def invertTree(self, node: TreeNode) -> TreeNode:  
 if not node:  
 return None  
  
 self.invertTree(node.left)  
 self.invertTree(node.right)  
  
 node.left, node.right = node.right, node.left  
 return node`

In [3]: # 迭代，前序遍历  
`class Solution:  
 def invertTree(self, root: TreeNode) -> TreeNode:  
 if not root:  
 return None  
 stack = [root]  
  
 while stack:  
 node = stack.pop()  
 node.left, node.right = node.right, node.left  
  
 if node.left:  
 stack.append(node.left)  
  
 if node.right:  
 stack.append(node.right)  
 return root`

In [5]: # 迭代，层序遍历  
`from collections import deque  
  
class Solution:  
 def invertTree(self, root: TreeNode) -> TreeNode:  
 if not root:  
 return None  
  
 queue = deque([root])  
 while queue:  
 node = queue.popleft()  
 node.left, node.right = node.right, node.left  
 if node.left: queue.append(node.left)  
 if node.right: queue.append(node.right)  
 return root`

## 101. 对称二叉树

给你一个二叉树的根节点 `root`，检查它是否轴对称。

### 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

### 提示:

- 树中节点数目在范围 [1, 1000] 内
- $-100 \leq \text{Node.val} \leq 100$

```
In [ ]: # 递归

class Solution:
    def isSymmetric(self, root: Optional[TreeNode]) -> bool:
        # base case
        if not root:
            return True

        return self.compare(root.left, root.right)

    def compare(self, left, right):
        # 判断本层
        # 空节点
        if left == None and right != None: return False
        elif left != None and right == None: return False
        elif left == None and right == None: return True
        # 数值不同
        elif left.val != right.val: return False

        # 进一步判断下一层
        result = self.compare(left.left, right.right) and self.compare(left.right, right.left)
        return result
```

```
In [ ]: # 迭代: 队列

import collections

class Solution:
    def isSymmetric(self, root: TreeNode) -> bool:
        if not root:
            return True

        queue = collections.deque()
        queue.append(root.left) # 将左子树头结点加入队列
        queue.append(root.right) # 将右子树头结点加入队列

        while queue: # 接下来就要判断这两个树是否相互翻转

            leftNode = queue.popleft()
            rightNode = queue.popleft()
            if not leftNode and not rightNode: # 左节点为空、右节点为空, 此时说明是对称的
                continue

            # 左右一个节点不为空, 或者都不为空但数值不相同, 返回false
            if not leftNode or not rightNode or leftNode.val != rightNode.val:
                return False

            queue.append(leftNode.left) # 加入左节点左孩子
            queue.append(rightNode.right) # 加入右节点右孩子
            queue.append(leftNode.right) # 加入左节点右孩子
            queue.append(rightNode.left) # 加入右节点左孩子

        return True
```

```
In [ ]: # 迭代, 栈

class Solution:
    def isSymmetric(self, root: TreeNode) -> bool:
        if not root:
            return True

        stack = [] # 这里改成了栈, 本质上就是一个list
        stack.append(root.left)
        stack.append(root.right)

        while stack:

            rightNode = stack.pop()
            leftNode = stack.pop()

            if not leftNode and not rightNode:
                continue
            if not leftNode or not rightNode or leftNode.val != rightNode.val:
                return False
```

```

        stack.append(leftNode.left)
        stack.append(rightNode.right)
        stack.append(leftNode.right)
        stack.append(rightNode.left)

    return True

```

In [ ]: # 迭代，栈

```

class Solution:
    def isSymmetric(self, root: TreeNode) -> bool:
        if not root:
            return True

        queue = collections.deque([root.left, root.right])

        while queue:
            level_size = len(queue)

            if level_size % 2 != 0:
                return False

            level_vals = []
            for i in range(level_size):
                node = queue.popleft()
                if node:
                    level_vals.append(node.val)
                    queue.append(node.left)
                    queue.append(node.right)
                else:
                    level_vals.append(None)

            if level_vals != level_vals[::-1]:
                return False

    return True

```

## 222. 完全二叉树的节点个数

给你一棵完全二叉树的根节点 `root`，求出该树的节点个数。

**完全二叉树的定义：**

在完全二叉树中，除了最底层节点可能没填满外，其余每层节点数都达到最大值，并且最下面一层的节点都集中在该层最左边的若干位置。若最底层为第  $h$  层（从第 0 层开始），则该层包含 1 到  $2^h$  个节点。

---

### 示例 1

输入:

```
root = [1, 2, 3, 4, 5, 6]
```

输出:

```
6
```

---

### 示例 2

输入:

```
root = []
```

输出:

```
0
```

---

### 示例 3

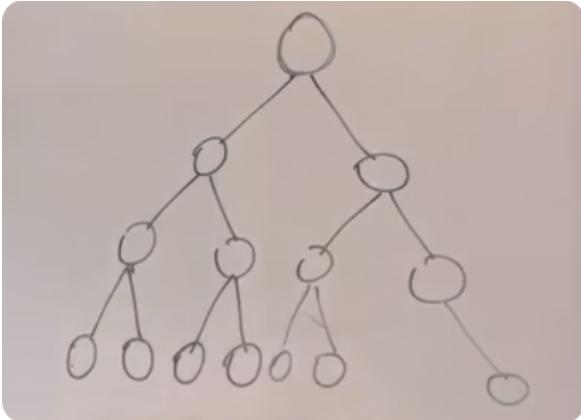
输入:

```
root = [1]
```

输出:

```
1
```

In [ ]:



如果只有一个右节点,就不是完全二叉树了

In [ ]: 满二叉树的性质:

节点数量 =  $2^{\text{层数}} - 1$

所以通过深度优先搜索,遍历最左边的节点,如果最深的子树是满二叉树,那么该层的层树就可以直接计算出节点数量

一条线一直往左递归

另一条线一直往右递归

如果两条线的节点数量相等,那么就是完全二叉树

```
In [ ]: class Solution:
    def countNodes(self, root: TreeNode) -> int:
        if not root:
            return 0
        left = root.left
        right = root.right
        leftDepth = 0 #这里初始为0是有目的的,为了下面求指数方便
        rightDepth = 0
        while left: #求左子树深度
            left = left.left
            leftDepth += 1
        while right: #求右子树深度
            right = right.right
            rightDepth += 1
        if leftDepth == rightDepth:
            return (2 << leftDepth) - 1 #注意(2<<1) 相当于2^2, 所以leftDepth初始为0
        return self.countNodes(root.left) + self.countNodes(root.right) + 1
```

你的代码并没有漏加节点数,即使遇到非完全二叉子树时,它也会递归处理左子树和右子树并正确累加节点数。

以下是代码的逻辑解释:

### 1. 完全二叉树的特殊处理:

如果左子树的深度 `leftDepth` 等于右子树的深度 `rightDepth`,说明当前子树是一个完全二叉树。可以直接使用公式 `(2 << leftDepth) - 1` 来计算节点总数,而无需进一步递归计算。

### 2. 非完全二叉树的处理:

如果左、右子树的深度不相等,则递归调用 `self.countNodes` 分别处理左子树和右子树,并累加根节点(即 `+1`),确保所有节点都被正确计数。

因此,非完全二叉树的节点数也不会漏加,因为在非完全二叉树的情况下,代码进入递归分支 `self.countNodes(root.left) + self.countNodes(root.right) + 1`,从而遍历所有节点。

## 举例说明

假设一棵非完全二叉树如下:



- 初始调用 `countNodes(root)`, `root` 为节点 1。
  - 左子树深度为 2, 右子树深度为 1(不相等)。
  - 递归调用 `countNodes(root.left)` 和 `countNodes(root.right)`。
- 处理左子树(根为节点 2):
  - 左、右子树深度相等,为 1。
  - 节点 2 的子树是完全二叉树,节点总数为  $(2 << 1) - 1 = 3$ 。
- 处理右子树(根为节点 3):
  - 左、右子树深度均为 0(没有子节点)。

- 节点总数为 1。
- 最终结果为  $3 + 1 + 1 = 5$ 。

因此，代码逻辑是正确的，完全覆盖了完全二叉树和非完全二叉树的情况，没有遗漏节点计数。

## 110. 平衡二叉树

给定一个二叉树，判断它是否是平衡二叉树。

平衡二叉树的定义：

一棵二叉树每个节点的左右子树高度差的绝对值不超过 1。

---

### 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`

---

### 示例 3

输入：  
`root = []`  
输出：  
`true`

---

### 提示

- 树中的节点数在范围 `[0, 5000]` 内
- $-10^4 \leq \text{Node.val} \leq 10^4$

### 求高度要用 后序遍历

先求左+右的高度 再在 中 进行比较

二叉树节点 的 深度：根节点到该节点的最长简单路径边的条数。

二叉树节点 的 高度：该节点到叶子节点的最长简单路径边的条数。

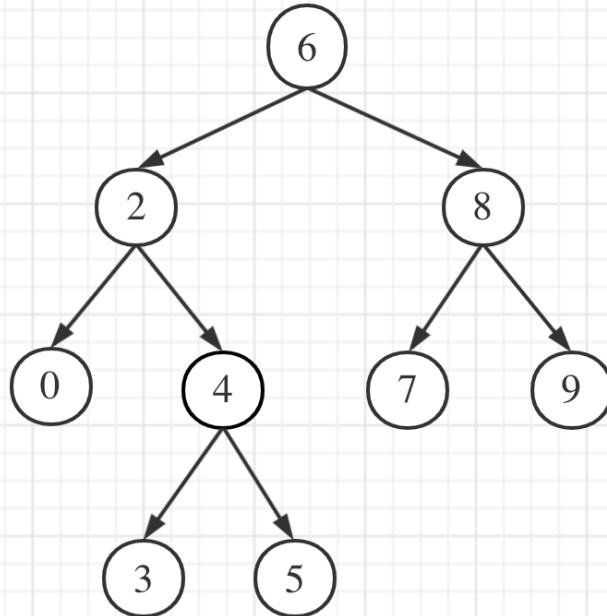
leetcode的题目中根节点深度是1

节点的高度: 4, 深度: 1

节点的高度: 3, 深度: 2

节点的高度: 2, 深度: 3

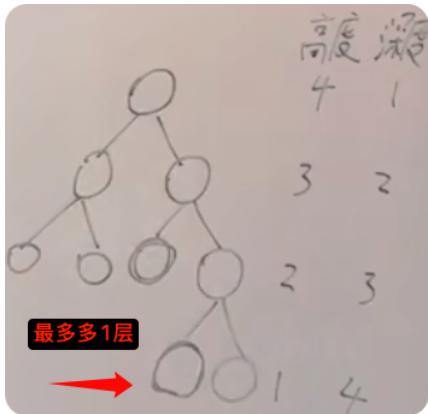
节点的高度: 1, 深度: 4



高度与深度的计算中: leetcode中都是以节点为一度, 但维基百科上是边为一度, 暂时以leetcode为准



平衡二叉树概念



递归三部曲

1. 明确递归函数的参数和返回值
  - A. 参数: 当前传入节点
  - B. 返回值: 以当前传入节点为根节点的树的高度
2. 明确终止条件: 遇到空节点终止
3. 明确单层递归逻辑: 左子树和右子树高度差值

```
In [ ]: # 递归

# Definition for a binary tree node.
from typing import Optional

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def isBalanced(self, root: Optional[TreeNode]) -> bool:
        if self.get_height(root) != -1:
            return True
```

```

    else:
        return False

def get_height(self, root: TreeNode) -> int:

    # base case (终止条件)
    if not root:
        return 0 # 空树高度为0

    left_height = self.get_height(root.left)
    right_height = self.get_height(root.right)

    # return -1 if not balanced

    # 左
    if left_height == -1:
        return -1

    # 右
    if right_height == -1:
        return -1

    # 中
    if abs(left_height - right_height) > 1:
        return -1

    # return height if balanced
    return max(left_height, right_height) + 1

```

## 上面修复了一个错误

```

# base case (终止条件)
if not root:
    return -1 # 空树高度为-1, 这不符合树的高度定义

# base case (终止条件)
if not root:
    return 0 # 空树高度为0

```

## 代码结构的「左、右、中」含义

### 1. 左 (处理左子树)

```
left_height = self.get_height(root.left)
```

这部分代码通过递归调用 `self.get_height` 处理当前节点的左子树，计算左子树的高度。如果左子树已经失衡（即返回 `-1`），后续判断会直接返回 `-1`，不再继续。

### 2. 右 (处理右子树)

```
right_height = self.get_height(root.right)
```

类似于左子树的处理，递归调用 `self.get_height` 计算右子树的高度。如果右子树已经失衡（即返回 `-1`），也会直接返回 `-1`。

### 3. 中 (处理当前节点)

```
if abs(left_height - right_height) > 1:
    return -1
```

这部分代码处理当前节点，判断左右子树的高度差是否超过 1。如果高度差大于 1，说明当前节点不平衡，返回 `-1`。  
如果当前节点平衡，则返回当前节点的高度：

```
return max(left_height, right_height) + 1
```

---

## 总结

- 左：递归计算左子树的高度，检查左子树是否平衡。
- 右：递归计算右子树的高度，检查右子树是否平衡。
- 中：检查当前节点的左右子树高度差是否超过 1，并返回节点高度或标记为不平衡。

这种「左、右、中」的遍历模式是后序遍历 (post-order traversal)，因为在处理当前节点之前，先处理了左子树和右子树。

## 为什么是后序遍历

之所以称这段代码的遍历为后序遍历 (post-order traversal)，是因为在递归过程中，对每个节点的逻辑顺序是：

1. 先处理左子树 (递归计算左子树高度)。
2. 再处理右子树 (递归计算右子树高度)。
3. 最后处理当前节点 (检查是否平衡，并计算当前节点高度)。

## 为什么是后序遍历？

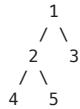
后序遍历的核心特点是：当前节点的逻辑处理依赖于左右子树的结果。

- 在这里，只有左子树和右子树都返回了各自的高度（或者确定是否平衡），才能对当前节点进行处理。

- 所以，这段代码的执行顺序是典型的后序遍历：左 → 右 → 当前节点。
- 

## 示例

假设树如下：



遍历顺序：

- 递归到节点 4，处理完左右子树后，返回高度。
- 回到节点 5，处理完左右子树后，返回高度。
- 处理节点 2（此时左右子树都已处理完）。
- 转到右子树，递归处理节点 3。
- 最后处理根节点 1。

遍历顺序为：4 → 5 → 2 → 3 → 1，符合后序遍历的特性。

## 对比前序和中序

### 前序遍历（Pre-order Traversal）

顺序：中 → 左 → 右

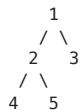
处理顺序：先处理当前节点，然后递归处理左子树，最后递归处理右子树。

#### 伪代码

```
def pre_order_traversal(node):  
    if not node:  
        return  
    # 中：处理当前节点  
    print(node.val) # 或进行其他操作  
    # 左：递归处理左子树  
    pre_order_traversal(node.left)  
    # 右：递归处理右子树  
    pre_order_traversal(node.right)
```

#### 执行示例

给定二叉树：



遍历顺序：1 → 2 → 4 → 5 → 3

- 先处理根节点 1，然后依次处理左子树（2 → 4 → 5），最后处理右子树（3）。
- 

### 中序遍历（In-order Traversal）

顺序：左 → 中 → 右

处理顺序：先递归处理左子树，然后处理当前节点，最后递归处理右子树。

#### 伪代码

```
def in_order_traversal(node):  
    if not node:  
        return  
    # 左：递归处理左子树  
    in_order_traversal(node.left)  
    # 中：处理当前节点  
    print(node.val) # 或进行其他操作  
    # 右：递归处理右子树  
    in_order_traversal(node.right)
```

#### 执行示例

给定二叉树：



遍历顺序: 4 → 2 → 5 → 1 → 3

- 先处理左子树, 得到 4 → 2 → 5; 然后处理根节点 1; 最后处理右子树 3。

## 对比总结

| 遍历类型 | 处理顺序      | 示例顺序 (树结构见上)      | 关键点               |
|------|-----------|-------------------|-------------------|
| 前序遍历 | 中 → 左 → 右 | 1 → 2 → 4 → 5 → 3 | 先处理当前节点, 再递归处理子树。 |
| 中序遍历 | 左 → 中 → 右 | 4 → 2 → 5 → 1 → 3 | 左子树完全处理后才处理当前节点。  |
| 后序遍历 | 左 → 右 → 中 | 4 → 5 → 2 → 3 → 1 | 左右子树都处理完才处理当前节点。  |

```
In [ ]: # 迭代法, 用stack, 层序遍历
# 这个不好记, 就先放弃
```

```

class Solution:
    def getDepth(self, cur):
        stack = []
        if cur is not None:
            stack.append(cur)
        depth = 0
        result = 0
        while stack:
            node = stack[-1]
            if node is not None:
                stack.pop()
                stack.append(node)          # 中
                stack.append(None)
                depth += 1
                if node.right:
                    stack.append(node.right)      # 右
                if node.left:
                    stack.append(node.left)       # 左

            else:
                node = stack.pop()
                stack.pop()
                depth -= 1
                result = max(result, depth)

        return result

    def isBalanced(self, root):
        stack = []
        if root is None:
            return True
        stack.append(root)
        while stack:
            node = stack.pop()           # 中
            if abs(self.getDepth(node.left) - self.getDepth(node.right)) > 1:
                return False
            if node.right:
                stack.append(node.right)      # 右 (空节点不入栈)
            if node.left:
                stack.append(node.left)       # 左 (空节点不入栈)
        return True

```

```
In [ ]:
```

## 257. 二叉树的所有路径

### 题目描述

给你一个二叉树的根节点 `root`, 按任意顺序, 返回所有从根节点到叶子节点的路径。

叶子节点 是指没有子节点的节点。

---

### 示例 1

输入:

```
root = [1,2,3,null,5]
```

输出:

```
["1->2->5","1->3"]
```

## 示例 2

输入:

```
root = [1]
```

输出:

```
["1"]
```

```
In [ ]: # 递归 + 回溯
# 这道题有点难，需要想象力
# 之后再复习

class Solution:
    def binaryTreePaths(self, root: Optional[TreeNode]) -> List[str]:
        result = []
        path = []
        if not root:
            return result
        self.traversal(root, path, result)
        return result

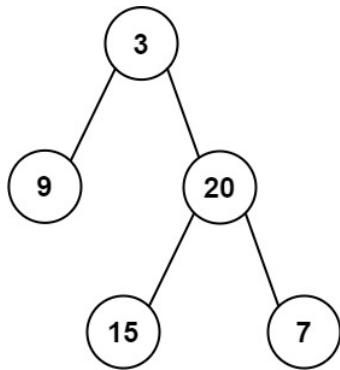
    def traversal(self, cur, path, result):
        path.append(cur.val) # 中 # 这里用一个path在每次递归的时候都记录下当前节点

        if not cur.left and not cur.right: # 到达叶子节点
            sPath = '->'.join(map(str, path))
            result.append(sPath) # 到达叶子节点，就把当前路径加入到result里面
            return
        if cur.left: # 左
            self.traversal(cur.left, path, result)
            path.pop() # 回溯 # 这里巧妙，在归的过程中，将path的最后一位踢出去，最后回到root节点的时候，path就是空的了
        if cur.right: # 右
            self.traversal(cur.right, path, result)
            path.pop() # 回溯
```

## 404. 左叶子之和

给定二叉树的根节点 root，返回所有左叶子之和。

示例 1：



输入: root = [3,9,20,null,null,15,7]

输出: 24

解释: 在这个二叉树中，有两个左叶子，分别是 9 和 15，所以返回 24

示例 2：

输入: root = [1]

输出: 0

### 知识点: 判断左叶子的逻辑

该节点的 左节点 不为空 该节点的 左节点 的 左节点 和 右节点 都为空，

```
if (node->left != NULL && node->left->left == NULL && node->left->right == NULL) { 左叶子节点处理逻辑 }
```

### 递归法

后序遍历 (左右中)

递归三部曲:

1. 确定递归函数的参数和返回值

2. 确定终止条件

```
if (root == NULL) return 0;
```

3. 确定单层递归的逻辑

```
int leftValue = sumOfLeftLeaves(root -> left);
```

In [ ]:

```
In [3]: # Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def sumOfLeftLeaves(self, root):
        if root is None:
            return 0
        if root.left is None and root.right is None:
            return 0

        # 左
        leftValue = self.sumOfLeftLeaves(root.left)

        # 左子树是左叶子的情况
        if root.left and not root.left.left and not root.left.right:
            leftValue = root.left.val

        # 右
        rightValue = self.sumOfLeftLeaves(root.right)

        return leftValue + rightValue
```

```

# 中
sum_val = leftValue + rightValue

return sum_val

```

In [4]: # 迭代法

```

class Solution:
    def sumOfLeftLeaves(self, root):
        if root is None:
            return 0
        st = [root]
        result = 0
        while st:
            node = st.pop()
            if node.left and node.left.left is None and node.left.right is None:
                result += node.left.val
            if node.right:
                st.append(node.right)
            if node.left:
                st.append(node.left)
        return result

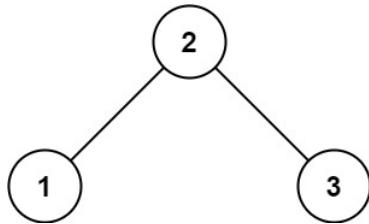
```

## 513.找树左下角的值

给定一个二叉树的根节点 root，请找出该二叉树的最底层最左边节点的值。

假设二叉树中至少有一个节点。

示例 1:

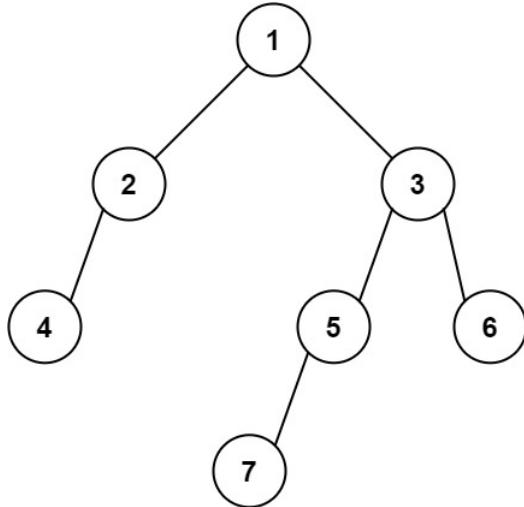


输入: root = [2,1,3]

输出: 1

示例 2:

输入: [1,2,3,4,null,5,6,null,null,7]



输出: 7

提示:

二叉树的节点个数的范围是 [1,104]

-231 <= Node.val <= 231 - 1

In [ ]:

## 递归法

## 确定递归函数的参数和返回值

参数: 根节点, 当前最长深度, 返回值: 不需要 保存变量: 最大深度, 最大值

## 确定终止条件

什么时候终止: 遇到叶子节点的时候 终止时做什么: 统计最大深度

## 确定单层递归的逻辑

用回溯

```
In [ ]: class Solution:
    def findBottomLeftValue(self, root: TreeNode) -> int:
        self.max_depth = float('-inf')
        self.result = None
        self.traversal(root, 0)
        return self.result

    def traversal(self, node, depth):
        # 前序遍历 - 中-左-右

        # 终止条件, 遇到子节点
        # 这是“中”
        if not node.left and not node.right:
            # 如果满足更新结果条件, 更新result
            if depth > self.max_depth:
                self.max_depth = depth
                self.result = node.val
            return

        if node.left:
            depth += 1
            self.traversal(node.left, depth) # 左
            depth -= 1

        if node.right:
            depth += 1
            self.traversal(node.right, depth) # 右
            depth -= 1
```

```
In [ ]: # 迭代法: 层序遍历

from collections import deque
class Solution:
    def findBottomLeftValue(self, root):
        if root is None:
            return 0

        # 用deque模拟stack, FIFO
        queue = deque()
        queue.append(root)
        result = 0
        while queue:
            size = len(queue)
            for i in range(size):
                node = queue.popleft()
                if i == 0:
                    result = node.val
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)
        return result
```

## 112. 路径总和

### 题目描述

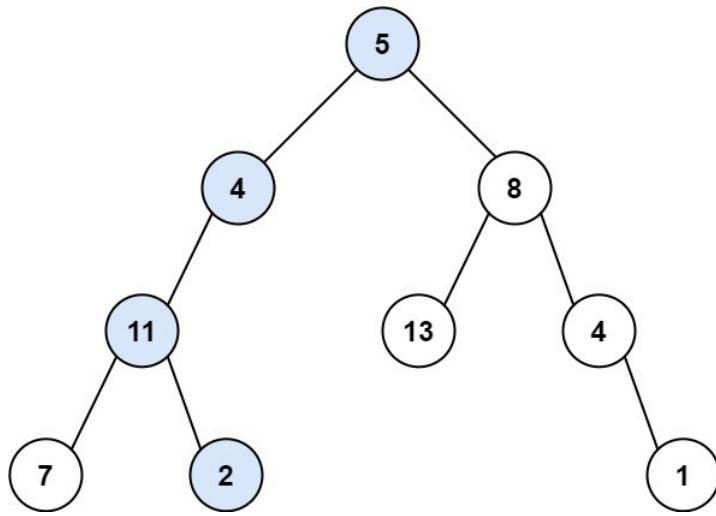
给你二叉树的根节点 `root` 和一个表示目标和的整数 `targetSum`。判断该树中是否存在 根节点到叶子节点 的路径，这条路径上所有节点值相加等于目标和 `targetSum`。如果存在，返回 `true`；否则，返回 `false`。

叶子节点 是指没有子节点的节点。

---

### 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：

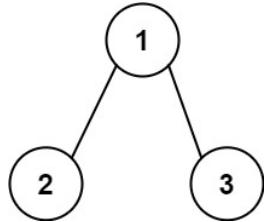
```
true
```

解释：

等于目标和的根节点到叶子节点路径如图所示。

---

## 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。
- 

## 示例 3

输入：

```
root = []
targetSum = 0
```

输出：

```
false
```

解释：

由于树是空的，所以不存在根节点到叶子节点的路径。

---

## 提示：

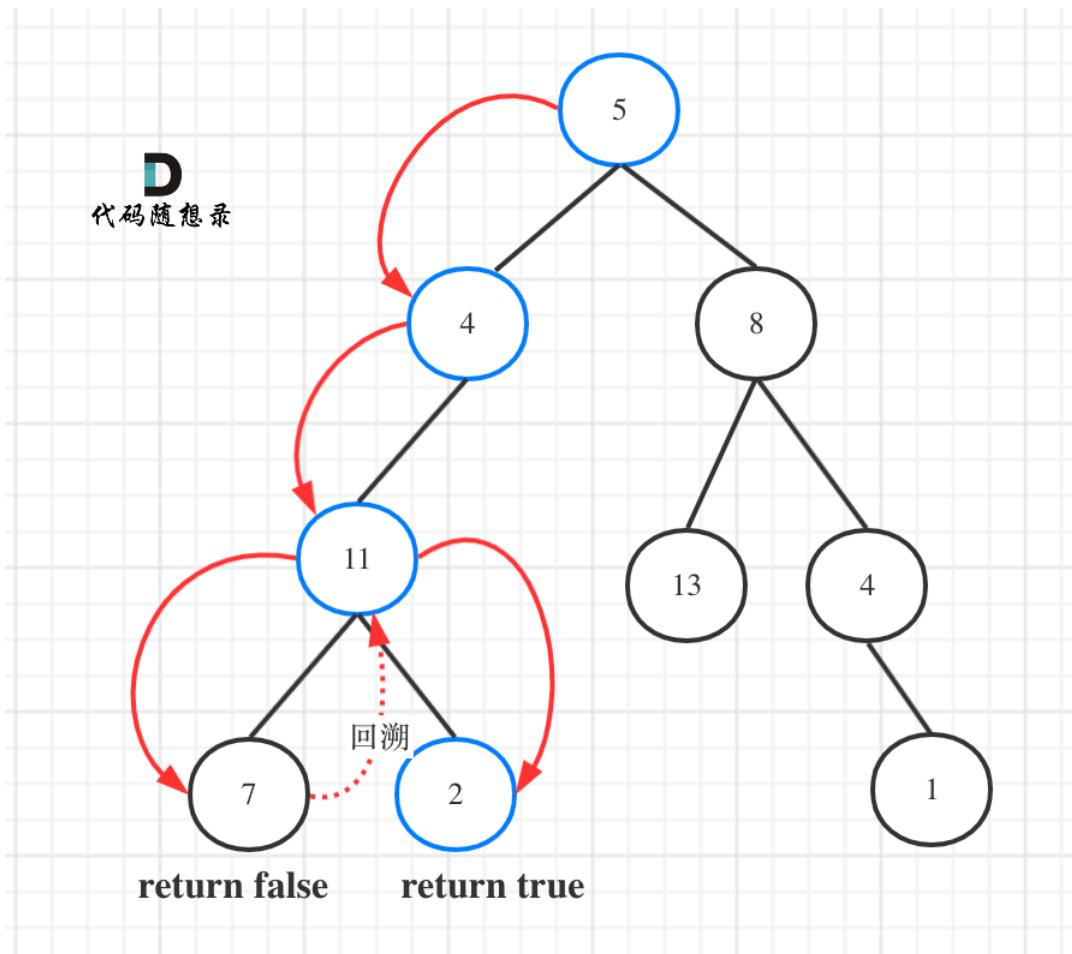
- 树中节点的数目在范围  $[0, 5000]$  内。
- $-1000 \leq \text{Node.val} \leq 1000$
- $-1000 \leq \text{targetSum} \leq 1000$

## 递归 + 回溯

### 深度优先遍历

1. 确定递归函数的参数和返回类型

2.  
3.



```
In [ ]: class Solution:
    def traversal(self, cur: TreeNode, count: int) -> bool:
        # 中
        if not cur.left and not cur.right and count == 0: # 遇到叶子节点，并且计数为0
            return True
        if not cur.left and not cur.right: # 遇到叶子节点直接返回
            return False

        if cur.left: # 左
            count -= cur.left.val
            if self.traversal(cur.left, count): # 递归，处理节点
                return True
            count += cur.left.val # 回溯，撤销处理结果

        if cur.right: # 右
            count -= cur.right.val
            if self.traversal(cur.right, count): # 递归，处理节点
                return True
            count += cur.right.val # 回溯，撤销处理结果

    return False

def hasPathSum(self, root: Optional[TreeNode], targetSum: int) -> bool:
```

```

        if root is None:
            return False
        return self.traversal(root, targetSum - root.val)

In [ ]: class Solution:
    def hasPathSum(self, root: Optional[TreeNode], targetSum: int) -> bool:
        if not root:
            return False
        # 此时栈里要放的是pair<节点指针, 路径数值>
        st = [(root, root.val)]
        while st:
            node, path_sum = st.pop()
            # 如果该节点是叶子节点了, 同时该节点的路径数值等于sum, 那么就返回true
            if not node.left and not node.right and path_sum == sum:
                return True
            # 右节点, 压进去一个节点的时候, 将该节点的路径数值也记录下来
            if node.right:
                st.append((node.right, path_sum + node.right.val))
            # 左节点, 压进去一个节点的时候, 将该节点的路径数值也记录下来
            if node.left:
                st.append((node.left, path_sum + node.left.val))
        return False

```

## 106. 从中序与后序遍历序列构造二叉树

### 题目描述

给定两个整数数组 `inorder` 和 `postorder`, 其中 `inorder` 是二叉树的中序遍历, `postorder` 是同一棵树的后序遍历, 请你构造并返回这颗二叉树。

### 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]
```

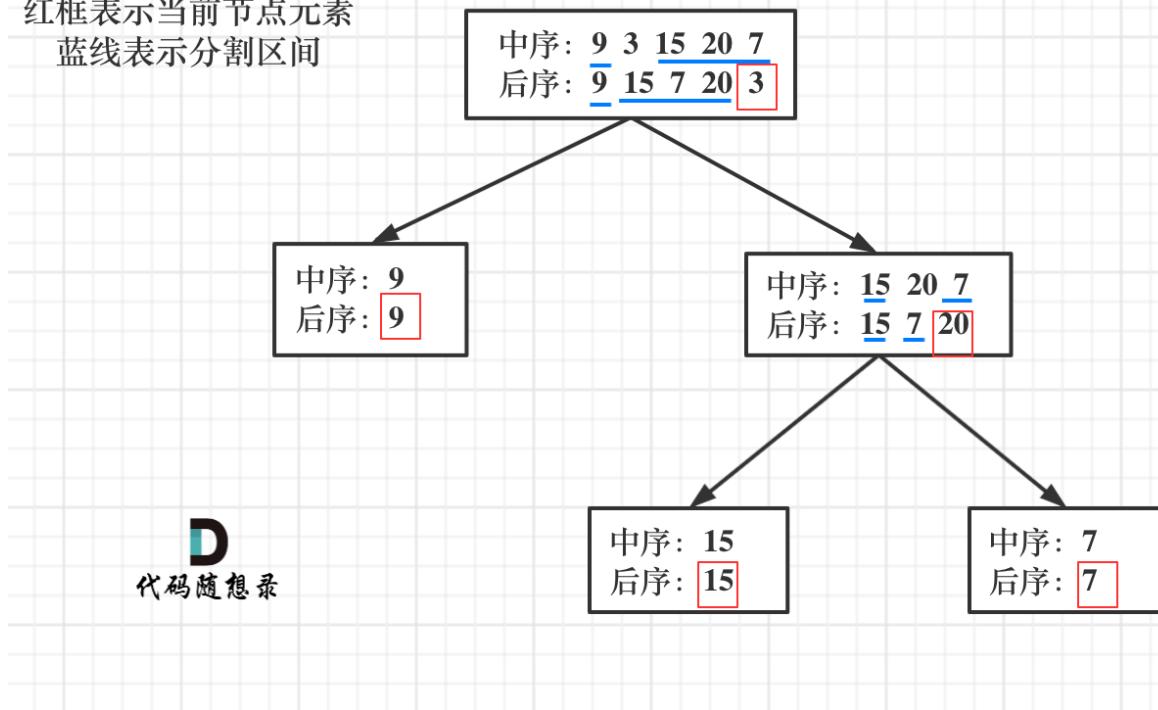
输出:

```
[-1]
```

### 提示

- $1 \leq \text{inorder.length} \leq 3000$
- $\text{postorder.length} == \text{inorder.length}$
- $-3000 \leq \text{inorder}[i], \text{postorder}[i] \leq 3000$
- `inorder` 和 `postorder` 都由不同的值组成
- `postorder` 中的每一个值都在 `inorder` 中
- `inorder` 保证是树的 中序遍历
- `postorder` 保证是树的 后序遍历

红框表示当前节点元素  
蓝线表示分割区间



以后序数组的最后一个元素为切割点，先切中序数组，根据中序数组，反过来再切后序数组。

第一步：如果数组大小为零的话，说明是空节点了。

第二步：如果不为空，那么取后序数组最后一个元素作为节点元素。

第三步：找到后序数组最后一个元素在中序数组的位置，作为切割点

第四步：切割中序数组，切成中序左数组和中序右数组（顺序别搞反了，一定是先切中序数组）

第五步：切割后序数组，切成后序左数组和后序右数组

第六步：递归处理左区间和右区间

In [13]: # 105. 从前序与中序遍历序列构造二叉树

```

from typing import List
class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> TreeNode:
        # 第一步：特殊情况讨论：树为空。或者说是递归终止条件
        if not preorder:
            return None

        # 第二步：前序遍历的第一个就是当前的中间节点。
        root_val = preorder[0]
        root = TreeNode(root_val)

        # 第三步：找切割点。
        separator_idx = inorder.index(root_val)

        # 第四步：切割inorder数组。得到inorder数组的左,右半边。
        inorder_left = inorder[:separator_idx]
        inorder_right = inorder[separator_idx + 1:]

        # 第五步：切割preorder数组。得到preorder数组的左,右半边。
        # ⚡ 重点1：中序数组大小一定跟前序数组大小是相同的。
        preorder_left = preorder[1:1 + len(inorder_left)] # @note 重点在这句
        preorder_right = preorder[1 + len(inorder_left):] # @note 重点在这句

        # 第六步：递归
        root.left = self.buildTree(preorder_left, inorder_left)
        root.right = self.buildTree(preorder_right, inorder_right)
        # 第七步：返回答案
        return root
  
```

In [14]: # 106. 从中序与后序遍历序列构造二叉树

```

class Solution:
    def buildTree(self, inorder: List[int], postorder: List[int]) -> TreeNode:
        # 第一步：特殊情况讨论：树为空。（递归终止条件）
        if not postorder:
            return None
  
```

```

# 第二步：后序遍历的最后一个就是当前的中间节点。
root_val = postorder[-1]
root = TreeNode(root_val)

# 第三步：找切割点。
separator_idx = inorder.index(root_val)

# 第四步：切割inorder数组，得到inorder数组的左，右半边。
inorder_left = inorder[:separator_idx]
inorder_right = inorder[separator_idx + 1:]

# 第五步：切割postorder数组，得到postorder数组的左，右半边。
# ⭐ 重点1：中序数组大小一定跟后序数组大小是相同的。
postorder_left = postorder[:len(inorder_left)]
postorder_right = postorder[len(inorder_left): len(postorder) - 1]

# 第六步：递归
root.left = self.buildTree(inorder_left, postorder_left)
root.right = self.buildTree(inorder_right, postorder_right)

# 第七步：返回答案
return root

```

## 654. 最大二叉树

### 题目描述

给定一个不重复的整数数组 `nums`，最大二叉树可以用下面的算法从 `nums` 递归地构建：

1. 创建一个根节点，其值为 `nums` 中的最大值。
2. 递归地在最大值左边的子数组前缀上构建左子树。
3. 递归地在最大值右边的子数组后缀上构建右子树。

返回 `nums` 构建的最大二叉树。

---

### 示例

#### 示例 1：

输入：

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

输出：

```
[6,3,5,null,2,0,null,null,1]
```

解释：

递归调用如下所示：

- `[3,2,1,6,0,5]` 中的最大值是 `6`，左边部分是 `[3,2,1]`，右边部分是 `[0,5]`。
    - `[3,2,1]` 中的最大值是 `3`，左边部分是 `[]`，右边部分是 `[2,1]`。
      - 空数组，无子节点。
      - `[2,1]` 中的最大值是 `2`，左边部分是 `[]`，右边部分是 `[1]`。
        - 空数组，无子节点。
        - 只有一个元素，所以子节点是一个值为 `1` 的节点。
    - `[0,5]` 中的最大值是 `5`，左边部分是 `[0]`，右边部分是 `[]`。
      - 只有一个元素，所以子节点是一个值为 `0` 的节点。
      - 空数组，无子节点。
- 

#### 示例 2：

输入：

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

输出：

```
[3,null,2,null,1]
```

---

### 提示

- `1 <= nums.length <= 1000`
- `0 <= nums[i] <= 1000`
- `nums` 中的所有整数互不相同

知识点: 构造树一般采用的是前序遍历, 因为先构造中间节点, 然后递归构造左子树和右子树。

递归思路:

确定递归函数的参数和返回值

参数传入的是存放元素的数组, 返回该数组构造的二叉树的头结点, 返回类型是指向节点的指针

In [ ]: