1. 两数之和

力扣题目链接(opens new window)

给定一个整数数组 nums 和一个目标值 target，请你在该数组中找出和为目标值的那 两个 整数，并返回他们的数组下标。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素不能使用两遍。

示例:

给定 nums = [2, 7, 11, 15], target = 9

因为 nums[0] + nums[1] = 2 + 7 = 9

所以返回 [0, 1]

什么时候使用哈希法?

> 当我们需要**快速判断一个元素是否出现过**，或者**是否存在于某个集合中**时，就应该第一时间想到**哈希表（Hash Table）**。

在「两数之和」这道题中：

- 我们不仅要判断某个数 `y = target - x` 是否出现过，
- 还要**知道它在数组中的下标**，以便返回 `[i, j]`。

这就要求我们使用的数据结构既能存"值"，又能存"位置" → 需要 **键值对（key-value）结构。**

为什么不能用 `list` 或 `set`?

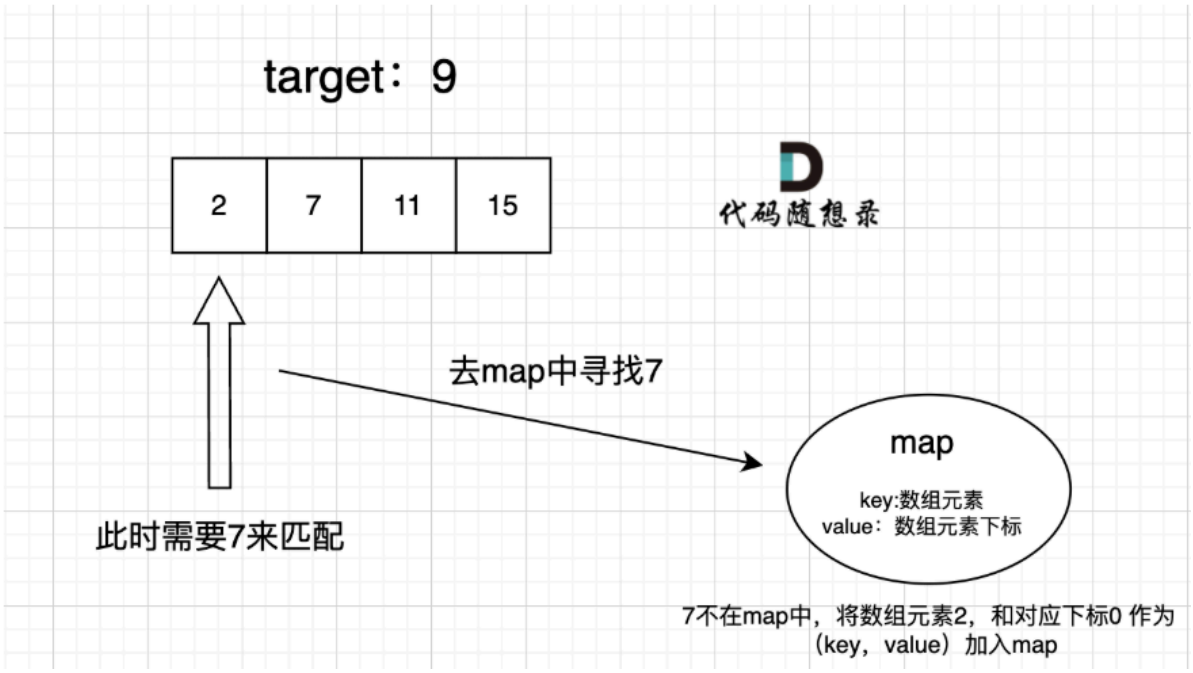
数据结构	问题
`list`（数组）	如果数值范围很大（比如 `10^9`），开数组会浪费大量内存；而且无法直接通过"值"快速查"下标"。
`set`	只能存"值"，**不能存下标。**而本题必须返回两个数的**索引**，所以 `set` 不够用。

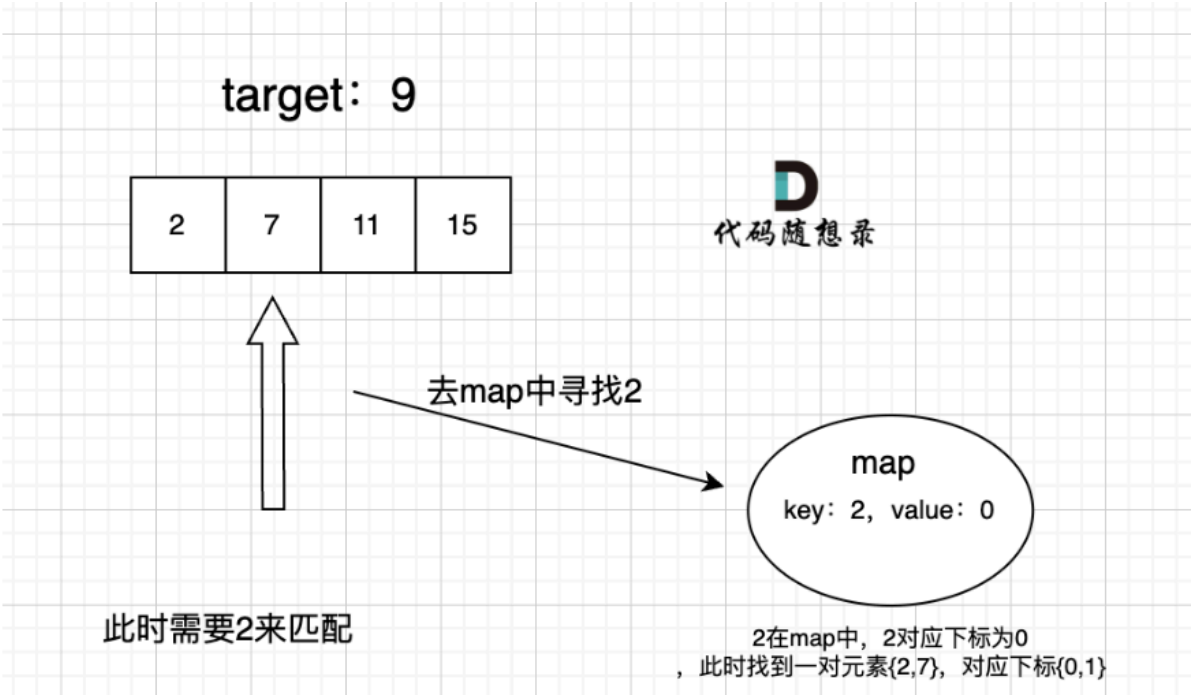
正确选择：字典 `dict`**（Python 的哈希表）**

在 Python 中：

- `dict` **就是哈希表**，底层使用哈希实现。
- 它是 **无序的（Python 3.6 以前），3.7+ 虽保持插入顺序，但逻辑上仍视为无序。**
- **平均时间复杂度：O(1) 查找、插入、删除。**

过程如下：





（版本一） 使用字典

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        records = dict()

        for index, value in enumerate(nums):
            if target - value in records:   # 遍历当前元素，并在map中寻找是否有匹配的
key
                return [records[target- value], index]
            records[value] = index    # 如果没找到匹配对，就把访问过的元素和下标加入到map
中
        return []
```

（版本二）使用集合

```python
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        #创建一个集合来存储我们目前看到的数字
        seen = set()
        for i, num in enumerate(nums):
            complement = target - num
            if complement in seen:
                return [nums.index(complement), i]
            seen.add(num)
```

（版本三）使用双指针

```python
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        num_sorted = sorted(nums)

        left = 0
        right = len(num_sorted) - 1

        while left<right:
            current_sum = num_sorted[left] +num_sorted[right]
            if current_sum == target:
                left_index=nums.index(num_sorted[left])
                right_index = nums.index(num_sorted[right])
                if left_index == right_index:
                    right_index = nums[left_index+1:].index(num_sorted[right]) +
left_index + 1
                # 从 left_index+1 开始的子数组
                # 在这个子数组中找 nums_sorted[right] 的位置
                # 转换回原始数组的全局下标
                return [left_index,right_index]

            elif current_sum < target:
                left += 1
            else:
                right -= 1
```

> **Note**
>
> ## 背景回顾
>
> 你用了 **排序 + 双指针** 找到两个**数值** `num_sorted[left]` 和 `num_sorted[right]`，它们的和等于 `target`。
>
> 但题目要求返回的是**原始数组** `nums` **中的下标**，而不是排序后的下标。
>
> 所以你需要：
>
> 1. 用 `nums.index(value)` 找这两个值在原数组中的位置。
> 2. **但如果两个值相等（比如** `[3, 3]`**，target=6），** `nums.index(3)` **会返回同一个下标（0）**，导致 `[0, 0]` —— 这是错误的！
>
> 于是你写了这个判断：

```
if left_index == right_index:
    right_index = nums[left_index+1:].index(num_sorted[right]) + left_index +
1
```

1. 为什么要这样做?

目的：**当两个目标值相等时，找到它们在原始数组中的两个不同下标**

- 比如 `nums = [3, 2, 3]`, `target = 6`
- 排序后：`num_sorted = [2, 3, 3]`
- 双指针找到 `3` 和 `3`
- `nums.index(3)` → 总是返回 **第一个 3 的下标（0）**
- 所以 `left_index = 0`, `right_index = 0` → 冲突!

你需要**在第一个 3 之后的部分**，再找一次第二个 3。

这就是 `if left_index == right_index` 分支要解决的问题。

2. 语法逐层解释

我们拆解这一行:

```
right_index = nums[left_index+1:].index(num_sorted[right]) + left_index + 1
```

步骤 1：`nums[left_index+1:]`

- 从 `left_index + 1` 开始切片，得到**原数组的后半部分**
- 例如：`nums = [3, 2, 3]`, `left_index = 0` → `nums[1:] = [2, 3]`

步骤 2：`.index(num_sorted[right])`

- 在这个子数组中找 `num_sorted[right]`（即 `3`）的**局部下标**
- `[2, 3].index(3)` → 返回 `1`

步骤 3：`+ left_index + 1`

- 把**局部下标**转换回**全局下标**
- `1 + 0 + 1 = 2` → 正确！第二个 3 在原数组的下标是 2

(版本四）暴力法

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