

## 【动画】从两数之和中，我们可以学到什么？ (Python/Java/C++/C/Go/JS/Rust)

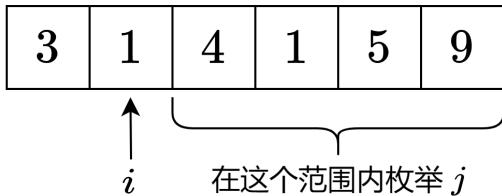
哈希表

枚举

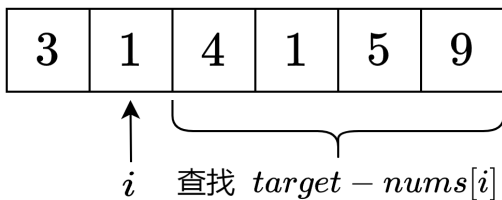
C

C++

6+

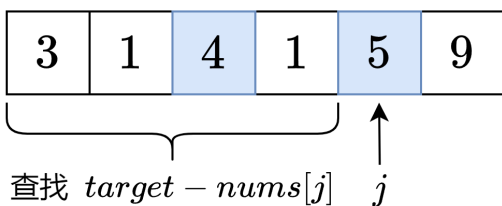
 $target = 9$ 


暴力做法：先枚举下标  $i$ ，再枚举下标  $j$ ，  
判断  $nums[i] + nums[j] = target$



变形：  $nums[j] = target - nums[i]$   
问题变成：在**一些数中找一个数**。

**哈希表**非常适合做这件事。

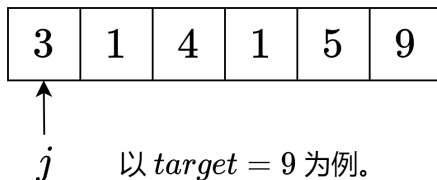


动画见下

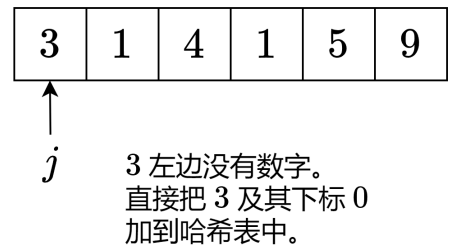
为了让代码更加好写，可以改成先枚举  $j$ 。  
一边枚举  $j$ ，一边把  $nums[j]$  和  $j$  加到哈希表中。  
具体见下面的动画。

为什么要先枚举  $j$ ，也就是右边的数呢？  
你可以试试先枚举  $i$  的写法，  
就能体会到先枚举  $j$  的好处了。

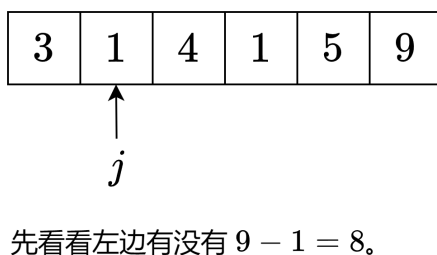
哈希表	
键	值



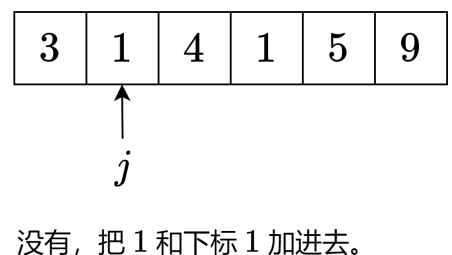
哈希表	
键	值
3	0



哈希表	
键	值
3	0



哈希表	
键	值
3	0
1	1



哈希表	
键	值
3	0
1	1

3	1	4	1	5	9
		$\uparrow$ $j$			

先看看左边有没有  $9 - 4 = 5$ 。

哈希表	
键	值
3	0
1	1
4	2

3	1	4	1	5	9
		$\uparrow$ $j$			

没有，把 4 和下标 2 加进去。

哈希表	
键	值
3	0
1	1
4	2

3	1	4	1	5	9
			$\uparrow$ $j$		

先看看左边有没有  $9 - 1 = 8$ 。

哈希表	
键	值
3	0
1	3
4	2

3	1	4	1	5	9
			$\uparrow$ $j$		

没有，把 1 和下标 3 加进去。

哈希表	
键	值
3	0
1	3
4	2

3	1	4	1	5	9
				$\uparrow$ $j$	

先看看左边有没有  $9 - 5 = 4$ 。

哈希表	
键	值
3	0
1	3
4	2

3	1	4	1	5	9
				$\uparrow$ $j = 4$	

有！那么答案为  $[2, 4]$ 。

## 答疑

问：是什么原因导致了这两种算法的快慢？

答：我用「获取了多少信息」来解释。

暴力做法每次拿两个数出来相加，和  $target$  比较，那么花费  $\mathcal{O}(1)$  的时间，只获取了  $\mathcal{O}(1)$  的信息。

而哈希表做法，每次查询都能知道  $\mathcal{O}(n)$  个数中是否有  $target - nums[j]$ ，那么花费  $\mathcal{O}(1)$  的时间，就获取了  $\mathcal{O}(n)$  的信息。

这就是为什么我们可以把暴力的  $\mathcal{O}(n^2)$  优化成  $\mathcal{O}(n)$ 。

问：力扣是如何测试题目的？为什么没有 `main` 函数？

答：简单来说，力扣评测机内部有 `main` 函数，里面会调用你写的 `twoSum` 函数（方法），传入相应的测试数据，并对比 `twoSum` 的返回值和正确答案是否一致。如果对于所有测试数据，返回值都与正确答案一致，则判定通过。

所以，我们只需编写核心逻辑，保证返回结果计算正确即可。

问：如何在本地测试代码？

答：见此。

## 暴力写法

Python3 | Java | C++ | C | Go | JavaScript | Rust

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        for (int i = 0; ; i++) { // 枚举 i
            for (int j = i + 1; j < nums.size(); j++) { // 枚举 i 右边的 j
                if (nums[i] + nums[j] == target) { // 满足要求
                    return {i, j}; // 返回两个数的下标
                }
            }
        }
        // 题目保证有解，循环中一定会 return
        // 所以这里无需 return，毕竟代码不会执行到这里
    }
};
```

## 复杂度分析

- 时间复杂度： $O(n^2)$ ，其中  $n$  为  $nums$  的长度。
- 空间复杂度： $O(1)$ 。仅用到若干额外变量。

## 哈希表写法

问：为什么下面的代码，要先查询  $idx$  是否有  $target - nums[j]$ ，再把  $nums[j]$  和  $j$  加到  $idx$  中？能不能反过来？

答：反过来写是错误的。例如  $nums = [2, 3, 1]$ ,  $target = 4$ ，如果先把  $nums[j]$  和  $j$  加到  $idx$  中，我们会认为  $2 + 2 = 4$ ，返回  $[0, 0]$ ，而正确答案应该是  $3 + 1 = 4$ ，也就是返回  $[1, 2]$ 。

原因在于，题目要求「不能使用两次相同的元素」，也就是**两个数的下标必须不同**。我们的做法是枚举右边的数的下标  $j$ ，去找左边的数的下标  $i$ 。由于找的是左边的数，如果先把右边的数加到  $idx$  中，找到的数就可能包含右边的数了，不符合题目要求。

Python3 | Java | C++ | C | Go | JavaScript | Rust

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int, int> idx; // 创建一个空哈希表
        for (int j = 0; ; j++) { // 枚举 j
            // 在左边找 nums[i]，满足 nums[i]+nums[j]=target
        }
    }
};
```

```
auto it = idx.find(target - nums[j]);  
if (it != idx.end()) { // 找到了  
    return {it->second, j}; // 返回两个数的下标  
}  
idx[nums[j]] = j; // 保存 nums[j] 和 j  
}  
}  
};
```

## 复杂度分析





- 时间复杂度： $O(n)$ ，其中  $n$  为  $nums$  的长度。
- 空间复杂度： $O(n)$ 。哈希表需要  $O(n)$  的空间。

相比暴力做法，哈希表多消耗了内存空间，但减少了运行时间，这就是「空间换时间」。

## 总结 · 练习

很多涉及到「两个变量」的题目，都可以枚举其中一个变量，把它当成常量看待，从而转化成「一个变量」的问题。

代码实现时，通常来说「枚举右，寻找左」是更加好写的。

-  [1512. 好数对的数目](#)
-  [219. 存在重复元素 II](#)
-  [1010. 总持续时间可被 60 整除的歌曲](#)
-  [2748. 美丽下标对的数目](#)
- 更多相似题目，请看 [数据结构题单](#) 第零章。

## 思考题

1. 如果  $nums$  是有序的，是否还需要哈希表？换句话说，能否做到  $O(1)$  额外空间？
2. 如果要求寻找三个数，它们的和等于  $target$  呢？