

Two Sum 问题的核心思想

Original labuladong labuladong 2019-09-22 22:08

收录于话题#手撕力扣高频面试题

https://mp.weixin.qq.com/s?__biz=MzAxODQxMDM0Mw==&mid=2247484474&idx=1&sn=dfba4be6e17d695a1907e2adcd9f0d3c&chksm=9bd7fa32aca0732406829a6d1de34b7e3144af239cc25e014f5349d73cea952d5f2b0473345a&scene=21#wechat_redirect

Two Sum 系列问题在 LeetCode 上有好几道，这篇文章就挑出有代表性的两道，介绍一下这种问题怎么解决。

TwoSum I

这个问题的最基本形式是这样：给你一个数组和一个整数target，可以保证数组中存在两个数的和为target，请你返回这两个数的索引。比如输入nums = [3,1,3,6],target = 6，算法应该返回数组[0,2]，因为 3 + 3 = 6。这个问题如何解决呢？首先最简单粗暴的办法当然是穷举了：

```
int[] twoSum(int[] nums, int target) {  
  
    for (int i = 0; i < nums.length; i++)  
        for (int j = i + 1; j < nums.length; j++)  
            if (nums[j] == target - nums[i])  
                return new int[] { i, j };  
  
    // 不存在这么两个数  
    return new int[] {-1, -1};  
}
```

这个解法非常直接，时间复杂度 $O(N^2)$ ，空间复杂度 $O(1)$ 。

更好一点的解法，可以通过一个哈希表减少时间复杂度：

```
int[] twoSum(int[] nums, int target) {  
    int n = nums.length;  
    index<Integer, Integer> index = new HashMap<>();  
    // 构造一个哈希表：元素映射到相应的索引  
    for (int i = 0; i < n; i++)  
        index.put(nums[i], i);  
  
    for (int i = 0; i < n; i++) {  
        int other = target - nums[i];  
        // 如果 other 存在且不是 nums[i] 本身  
        if (index.containsKey(other) && index.get(other) != i)  
            return new int[] {i, index.get(other)};  
    }  
  
    return new int[] {-1, -1};  
}
```

这样，由于哈希表的查询时间为 $O(1)$ ，算法的时间复杂度降低到 $O(N)$ ，但是需要 $O(N)$ 的空间复杂度来存储哈希表。不过综合来看，是要比暴力解法高效的。

我觉得 Two Sum 系列问题就是想教我们如何使用哈希表处理问题。我们接着往后看。

TwoSum II

稍微修改一下上面的问题，要求我们设计一个类，拥有两个 API：

```
class TwoSum {
    // 向数据结构中添加一个数 number
    public void add(int number);
    // 寻找当前数据结构中是否存在两个数的和为 value
    public boolean find(int value);
}
```

如何实现这两个 API 呢，我们可以仿照上一道题目，使用一个哈希表辅助 find 方法：

```
class TwoSum {
    Map<Integer, Integer> freq = new HashMap<>();

    public void add(int number) {
        // 记录 number 出现的次数
        freq.put(number, freq.getOrDefault(number, 0) + 1);
    }

    public boolean find(int value) {
        for (Integer key : freq.keySet()) {
            int other = value - key;
            // 情况一
            if (other == key && freq.get(key) > 1)
                return true;
            // 情况二
            if (other != key && freq.containsKey(other))
                return true;
        }
        return false;
    }
}
```

进行 find 的时候有两种情况，举个例子：

情况一：如果连续 add 了 [3,2,3,5]，那么 freq 是 {3:2,2:1,5:1}，执行 find(6)，由于 3 出现了两次， $3 + 3 = 6$ ，所以返回 true。

情况二：freq 是 {3:2,2:1,5:1}，执行 find(7)，那么 key 为 2，other 为 5 时算法可以返回 true。

除了上述两种情况外，find 只能返回 false 了。

对于这个解法的时间复杂度呢，add方法是 $O(1)$ ，find方法是 $O(N)$ ，空间复杂度为 $O(N)$ ，和上一道题目比较类似。

但是对于 API 的设计，是需要考虑现实情况的。比如说，我们设计的这个类，使用find方法非常频繁，那么每次都要 $O(N)$ 的时间，岂不是浪费很多时间吗？对于这种情况，我们是否可以做些优化呢？

是的，对于频繁使用find方法的场景，我们可以进行优化。我们可以参考上一道题目的暴力解法，借助哈希集合来针对性优化find方法：

```
class TwoSum {
    Set<Integer> sum = new HashSet<>();
    List<Integer> nums = new ArrayList<>();

    public void add(int number) {
        // 记录所有可能组成的和
        for (int n : nums)
            sum.add(n + number);
        nums.add(number);
    }

    public boolean find(int value) {
        return sum.contains(value);
    }
}
```

这样sum中就储存了所有加入数字可能组成的和，每次find只要花费 $O(1)$ 的时间在集合中判断一下是否存在就行了，显然非常适合频繁使用find的场景。

三、总结

对于 TwoSum 问题，一个难点就是给的数组无序。对于一个无序的数组，我们似乎什么技巧也没有，只能暴力穷举所有可能。

一般情况下，我们会首先把数组排序再考虑双指针技巧。TwoSum 启发我们，HashMap 或者 HashSet 也可以帮助我们处理无序数组相关的简单问题。

另外，设计的核心在于权衡，利用不同的数据结构，可以得到一些针对性的加强。

最后，如果 TwoSum I 中给的数组是有序的，应该如何编写算法呢？答案很简单，前文 [双指针技巧汇总](#) 写过：

```
int[] twoSum(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left < right) {
        int sum = nums[left] + nums[right];
        if (sum == target) {
            return new int[]{left, right};
        }
    }
}
```

```
    } else if (sum < target) {  
        left++; // 让 sum 大一点  
    } else if (sum > target) {  
        right--; // 让 sum 小一点  
    }  
}  
// 不存在这样两个数  
return new int[]{-1, -1};  
}
```

[动态规划之 KMP 算法详解](#)

[这个问题不简单：寻找缺失元素](#)

[二分查找的妙用：判定子序列](#)