原 java初、中、高级面试题必备——数据结构与算法—二分查找

2019年06月26日 15:59:28 在IT中穿梭旅行 阅读数 35 编辑

179.什么是二分查找?

二分查找中使用的术语:

- 目标 Target —— 你要查找的值
- 索引 Index —— 你要查找的当前位置
- 左、右指示符 Left，Right —— 我们用来维持查找空间的指标
- 中间指示符 Mid —— 我们用来应用条件来确定我们应该向左查找还是向右查找的索引

二分查找算法:

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 不存在 nums 中因此返回 -1
```


二分查找的模板示例1:

- 初始条件: `left = 0, right = length-1`
- 终止: `left > right`
- 向左查找: `right = mid-1`

- 向右查找：left = mid+1

```
class Solution {
    public int search(int[] nums, int target) {
     if(nums == null || nums.length == 0)
        return -1;
        int left = 0;
        int right = nums.length-1;
        int mid;
        while(left<=right){
             mid = (left+right)/2;
            if(target==nums[mid]){
                return mid;
            }else if(target<nums[mid]){
                right =mid-1;
            }else{
                left = mid +1;
            }
        }
        return -1;
    }
}
```

二分查找的高级模板示例2：

- 初始条件：left = 0, right = length
- 终止：left == right
- 向左查找：right = mid
- 向右查找：left = mid+1

```
int binarySearch(int[] nums, int target){
  if(nums == null || nums.length == 0)
    return -1;
```

```
4
5     int left = 0, right = nums.length;
6     while(left < right){
7       // Prevent (left + right) overflow
8       int mid = left + (right - left) / 2;
9       if(nums[mid] == target){ return mid; }
10      else if(nums[mid] < target) { left = mid + 1; }
11      else { right = mid; }
12    }
13
14    // Post-processing:
15    // End Condition: left == right
16    if(left != nums.length && nums[left] == target) return left;
17    return -1;
18  }
```

二分查找的模板示例3：

- 初始条件：left = 0, right = length-1
- 终止：left + 1 == right
- 向左查找：right = mid
- 向右查找：left = mid

```
1  int binarySearch(int[] nums, int target) {
2      if (nums == null || nums.length == 0)
3          return -1;
4
5      int left = 0, right = nums.length - 1;
6      while (left + 1 < right){
7          // Prevent (left + right) overflow
8          int mid = left + (right - left) / 2;
9          if (nums[mid] == target) {
10             return mid;
11         } else if (nums[mid] < target) {
```

```
            left = mid;
        } else {
            right = mid;
        }
    }

    // Post-processing:
    // End Condition: left + 1 == right
    if(nums[left] == target) return left;
    if(nums[right] == target) return right;
    return -1;
}
```

180.二分查找模板分析

这 3 个模板的不同之处在于：

- 左、中、右索引的分配。
- 循环或递归终止条件。
- 后处理的必要性。

模板 #1 和 #3 是最常用的，几乎所有二分查找问题都可以用其中之一轻松实现。模板 #2 更 高级一些，用于解决某些类型的问题。

这 3 个模板中的每一个都提供了一个特定的用例：

模板 #1 `(left <= right)`:

- 二分查找的最基础和最基本的形式。
- 查找条件可以在不与元素的两侧进行比较的情况下确定（或使用它周围的特定元素）。
- 不需要后处理，因为每一步中，你都在检查是否找到了元素。如果到达末尾，则知道未找到该元素。

模板 #2 `(left < right)`:

- 一种实现二分查找的高级方法。
- 查找条件需要访问元素的直接右邻居。
- 使用元素的右邻居来确定是否满足条件，并决定是向左还是向右。
- 保证查找空间在每一步中至少有 2 个元素。
- 需要进行后处理。 当你剩下 1 个元素时，循环 / 递归结束。 需要评估剩余元素是否符合条件。

模板 #3 `(left + 1 < right)`:

- 实现二分查找的另一种方法。
- 搜索条件需要访问元素的直接左右邻居。
- 使用元素的邻居来确定它是向右还是向左。
- 保证查找空间在每个步骤中至少有 3 个元素。
- 需要进行后处理。 当剩下 2 个元素时，循环 / 递归结束。 需要评估其余元素是否符合条件。

时间和空间复杂度:

时间: `O(log n)` —— 算法时间

因为二分查找是通过对查找空间中间的值应用一个条件来操作的，并因此将查找空间折半，在更糟糕的情况下，我们将不得不进行 O(log n) 次比较，其中 n 是集合中元素的数目。

> 为什么是 `log n`?
>
> - 二分查找是通过将现有数组一分为二来执行的。
> - 因此，每次调用子例程(或完成一次迭代)时，其大小都会减少到现有部分的一半。
> - 首先 **N** 变成 **N/2**，然后又变成 **N/4**，然后继续下去，直到找到元素或尺寸变为 **1**。

> - 迭代的最大次数是 `log N` (base 2) 。

空间： O(1) —— 常量空间

虽然二分查找确实需要跟踪 3 个指标，但迭代解决方案通常不需要任何其他额外空间，并且可以直接应用于集合本身，因此需要 O(1) 或常量空间。

下一章 二叉树链接：