murphy_gb

博客园 首页 新随笔 联系 订阅 管理

随笔 - 47 文章 - 1 评论 - 41

昵称: murphy_gb 园龄: 1年7个月 粉丝: 38 关注: 2 +加关注

<		>				
日	_	=	Ξ	兀	五	六
28	29	30	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	1
2	3	4	5	6	7	8



常用链接	
我的随笔	
我的评论	
我的参与	
最新评论	
我的标签	

我的标签	
算法(15)	
java(7)	
MySQL(6)	
并发(5)	
redis(3)	
数据库(3)	
设计模式(2)	
网络(2)	
工具(2)	
关于我(1)	
- 4	

详解二分查找算法

我周围的人几乎都认为二分查找很简单,但事实真的如此吗?二分查找真的很简单吗?并不简单。看看 Knuth 大佬 (发明 KMP 算法的那位)怎么说的:

Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky...

这句话可以这样理解:思路很简单,细节是魔鬼。

本文就来探究几个最常用的二分查找场景: 寻找一个数、寻找左侧边界、寻找右侧边界。

而且,我们就是要深入细节,比如while循环中的不等号是否应该带等号,mid 是否应该加一等等。 分析这些细节的差异以及出现这些差异的原因,保证你能灵活准确地写出正确的二分查找算法。

一、二分查找的框架

分析二分查找的一个技巧是: 不要出现 else, 而是把所有情况用 else if 写清楚,这样可以清楚地展现所有细节。本文都会使用 else if,旨在讲清楚,读者理解后可自行简化。

其中...标记的部分,就是可能出现细节问题的地方,当你见到一个二分查找的代码时,首先注意这几个地方。后文用实例分析这些地方能有什么样的变化。

另外声明一下,计算 mid 时需要技巧防止溢出,建议写成: mid = left + (right - left) / 2,本文暂时忽略这个问题。

二、寻找一个数(基本的二分搜索)

这个场景是最简单的,可能也是大家最熟悉的,即搜索一个数,如果存在,返回其索引,否则返回 -1。

```
int binarySearch(int[] nums, int target) {
   int left = 0;
   int right = nums.length - 1; // 注意

while(left <= right) { // 注意
   int mid = (right + left) / 2;
   if(nums[mid] == target)
        return mid;
   else if (nums[mid] < target)
        left = mid + 1; // 注意
   else if (nums[mid] > target)
        right = mid - 1; // 注意
   }
   return -1;
}
```

1. 为什么 while 循环的条件中是 <=, 而不是 <?

答: 因为初始化 right 的赋值是 nums.length - 1,即最后一个元素的索引,而不是 nums.length。

加美档案 2020年5月(1) 2020年2月(5) 2020年1月(1) 2019年12月(1) 2019年11月(1) 2019年9月(3) 2019年8月(12) 2019年7月(5) 2019年6月(11) 2019年5月(2) 2019年4月(1) 2019年3月(3) 2019年2月(1)		_
2020年2月(5) 2020年1月(1) 2019年12月(1) 2019年11月(1) 2019年9月(3) 2019年8月(12) 2019年7月(5) 2019年6月(11) 2019年5月(2) 2019年4月(1) 2019年3月(3)	随笔档案	
2020年1月(1) 2019年12月(1) 2019年11月(1) 2019年9月(3) 2019年8月(12) 2019年7月(5) 2019年6月(11) 2019年5月(2) 2019年4月(1) 2019年3月(3)	2020年5月(1)	
2019年12月(1) 2019年11月(1) 2019年9月(3) 2019年8月(12) 2019年7月(5) 2019年6月(11) 2019年5月(2) 2019年4月(1) 2019年3月(3)	2020年2月(5)	
2019年11月(1) 2019年9月(3) 2019年8月(12) 2019年7月(5) 2019年6月(11) 2019年5月(2) 2019年4月(1) 2019年3月(3)	2020年1月(1)	
2019年9月(3) 2019年8月(12) 2019年7月(5) 2019年6月(11) 2019年5月(2) 2019年4月(1) 2019年3月(3)	2019年12月(1)	
2019年8月(12) 2019年7月(5) 2019年6月(11) 2019年5月(2) 2019年4月(1) 2019年3月(3)	2019年11月(1)	
2019年7月(5) 2019年6月(11) 2019年5月(2) 2019年4月(1) 2019年3月(3)	2019年9月(3)	
2019年6月(11) 2019年5月(2) 2019年4月(1) 2019年3月(3)	2019年8月(12)	
2019年5月(2) 2019年4月(1) 2019年3月(3)	2019年7月(5)	
2019年4月(1) 2019年3月(3)	2019年6月(11)	
2019年3月(3)	2019年5月(2)	
,	2019年4月(1)	
2019年2月(1)	2019年3月(3)	
	2019年2月(1)	

最新评论

1. Re:来自一个菜鸡的秋招与春招之路

感觉楼主挺强的呀,你看的那些书面 试够用了吧,怎么这么难。

--Crisyi

2. Re:详解二分查找算法

给楼主献花! 讲的真好!

--Ingrrris

3. Re:什么是计数排序?

谢谢楼主!!!

这个的取值应该大于等于零吧,不然会 缺少一个值的。

--01000001

4. Re:数据库中的乐观锁与悲观锁

讲的还不错

--张凯歌

这二者可能出现在不同功能的二分查找中,区别是:前者相当于两端都闭区间 [left, right],后者相当于左闭右开区间 [left, right],因为索引大小为 nums.length 是越界的。

我们这个算法中使用的是 [left, right] 两端都闭的区间。**这个区间就是每次进行搜索的区间,我们不妨称为「搜索区间」(search space)**。

什么时候应该停止搜索呢? 当然,找到了目标值的时候可以终止:

```
if(nums[mid] == target)
    return mid;
```

但如果没找到,就需要 while 循环终止,然后返回 -1。那 while 循环什么时候应该终止? 搜索区间为空的时候应该终止,意味着你没得找了,就等于没找到嘛。

while(left <= right)的终止条件是 left == right + 1,写成区间的形式就是 [right + 1, right],或者带个具体的数字进去 [3, 2],可见**这时候搜索区间为空**,因为没有数字既大于等于 3 又小于等于 2 的吧。所以这时候 while 循环终止是正确的,直接返回 -1 即可。

while(left < right)的终止条件是 left == right, 写成区间的形式就是 [right, right], 或者带个具体的数字进去 [2, 2], **这时候搜索区间非空**, 还有一个数 2, 但此时 while 循环终止了。也就是说这区间 [2, 2] 被漏掉了, 索引 2 没有被搜索, 如果这时候直接返回 -1 就可能出现错误。

当然,如果你非要用 while(left < right) 也可以,我们已经知道了出错的原因,就打个补丁好了:

```
//...
while(left < right) {
    // ...
}
return nums[left] == target ? left : -1;</pre>
```

2. 为什么 left = mid + 1, right = mid - 1? 我看有的代码是 right = mid 或者 left = mid, 没有这些加加减减,到底怎么回事,怎么判断?

答: 这也是二分查找的一个难点,不过只要你能理解前面的内容,就能够很容易判断。

刚才明确了「搜索区间」这个概念,而且本算法的搜索区间是两端都闭的,即 [left, right]。那么当我们发现索引 mid 不是要找的 target 时,如何确定下一步的搜索区间呢?

当然是去搜索 [left, mid - 1] 或者 [mid + 1, right] 对不对? 因为 mid 已经搜索过,应该从搜索 $\overline{\mathsf{N}}$ $\overline{\mathsf{N}$

3. 此算法有什么缺陷?

答:至此,你应该已经掌握了该算法的所有细节,以及这样处理的原因。但是,这个算法存在局限性

比如说给你有序数组 nums = [1,2,2,2,3], target = 2, 此算法返回的索引是 2, 没错。但是如果我想得到 target 的左侧边界,即索引 1, 或者我想得到 target 的右侧边界,即索引 3, 这样的话此算法是无法处理的。

这样的需求很常见。你也许会说,找到一个 target 索引,然后向左或向右线性搜索不行吗?可以,但是不好,因为这样难以保证二分查找对数级的时间复杂度了。

我们后续的算法就来讨论这两种二分查找的算法。

三、寻找左侧边界的二分搜索

直接看代码,其中的标记是需要注意的细节:

```
int left_bound(int[] nums, int target) {
    if (nums.length == 0) return -1;
    int left = 0;
    int right = nums.length; // 注意

    while (left < right) { // 注意
        int mid = (left + right) / 2;
        if (nums[mid] == target) {
              right = mid;
        } else if (nums[mid] < target) {
              left = mid + 1;
        } else if (nums[mid] > target) {
                  right = mid; // 注意
        }
    }
    return left;
}
```

1. 为什么 while(left < right) 而不是 <= ?

5. Re:浅谈动态规划

真棒,继续写下去吧!

--洞庭湖底吃晚餐

阅读排行榜

- 1. 详解二分查找算法(29050)
- 2. Spring中用到了哪些设计模式? (1099 8)
 - 3. 什么是计数排序? (9572)
 - 4. 如何实现LRU算法? (6357)
 - 5. 数据库中的乐观锁与悲观锁(4519)

评论排行榜

- 1. 详解二分查找算法(10)
- 2. 等待唤醒 (wait / notify) 机制(5)
- 3. 如何寻找无序数组中的第K大元素? (4)
- 4. 数据库中的共享锁与排他锁(3)
- 5. TCP与UDP的主要特点(2)

推荐排行榜

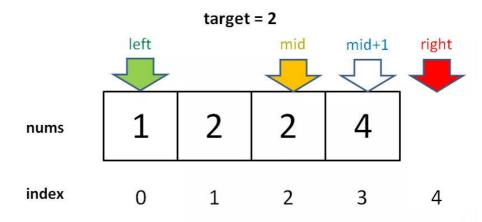
- 1. 详解二分查找算法(16)
- 2. 什么是计数排序? (9)
- 3. Spring中用到了哪些设计模式? (5)
- 4. 双指针技巧汇总(4)
- 5. 一些有趣有用的位运算(4)

答: 用相同的方法分析,因为初始化 right = nums.length 而不是 nums.length - 1。因此每次循环的「搜索区间」是 [left, right) 左闭右开。

while(left < right) 终止的条件是 left == right,此时搜索区间 [left, left)恰巧为空,所以可以正确终止。

2. 为什么没有返回 -1 的操作? 如果 nums 中不存在 target 这个值, 怎么办?

答: 因为要一步一步来, 先理解一下这个「左侧边界」有什么特殊含义:



对于这个数组, 算法会返回 1。这个 1 的含义可以这样解读: nums 中小于 2 的元素有 1 个。

比如对于有序数组 nums = [2,3,5,7], target = 1, 算法会返回 0, 含义是: nums 中小于 1 的元素有 0 个。如果 target = 8, 算法会返回 4, 含义是: nums 中小于 8 的元素有 4 个。

综上可以看出,函数的返回值(即 left 变量的值)取值区间是闭区间 [0, nums.length],所以我们简单添加两行代码就能在正确的时候 return -1:

```
while (left < right) {
    //...
}

// target 比所有数都大
if (left == nums.length) return -1;

// 类似之前算法的处理方式
return nums[left] == target ? left : -1;
```

3. 为什么 left = mid + 1, right = mid ? 和之前的算法不一样?

答:这个很好解释,因为我们的「搜索区间」是 [left, right) 左闭右开,所以当 nums[mid] 被检测之后,下一步的搜索区间应该去掉 mid 分割成两个区间,即 [left, mid) 或 [mid + 1, right)。

4. 为什么该算法能够搜索左侧边界?

答: 关键在于对于 nums[mid] == target 这种情况的处理:

```
if (nums[mid] == target)
    right = mid;
```

可见,找到 target 时不要立即返回,而是缩小「搜索区间」的上界 right, 在区间 [left, mid) 中继续搜索,即不断向左收缩,达到锁定左侧边界的目的。

5. 为什么返回 left 而不是 right?

答:返回left和right都是一样的,因为 while 终止的条件是 left == right。

四、寻找右侧边界的二分查找

寻找右侧边界和寻找左侧边界的代码差不多,只有两处不同,已标注:

```
int right_bound(int[] nums, int target) {
   if (nums.length == 0) return -1;
   int left = 0, right = nums.length;

while (left < right) {
    int mid = (left + right) / 2;
    if (nums[mid] == target) {
        left = mid + 1; // 注意
    } else if (nums[mid] < target) {
        left = mid + 1;
    } else if (nums[mid] > target) {
        right = mid;
    }
}
```

```
}
}
return left - 1; // 注意
```

1. 为什么这个算法能够找到右侧边界?

答: 类似地, 关键点还是这里:

```
if (nums[mid] == target) {
   left = mid + 1;
```

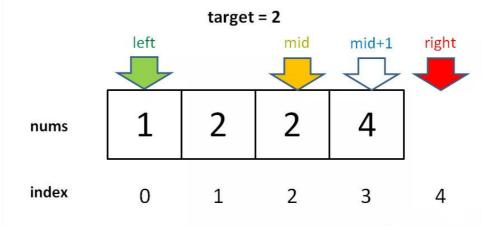
当 nums[mid] == target 时,不要立即返回,而是增大「搜索区间」的下界 left,使得区间不断向右收缩,达到锁定右侧边界的目的。

2. 为什么最后返回 left - 1 而不像左侧边界的函数,返回 left? 而且我觉得这里既然是搜索右侧边界,应该返回 right 才对。

答: 首先,while 循环的终止条件是 left == right,所以 left 和 right 是一样的,你非要体现右侧的特点,返回 right - 1 好了。

至于为什么要减一,这是搜索右侧边界的一个特殊点,关键在这个条件判断:

```
if (nums[mid] == target) {
   left = mid + 1;
   // 这样想: mid = left - 1
```



因为我们对 left 的更新必须是 left = mid + 1, 就是说 while 循环结束时, nums[left] **一定不等** \mathbf{F} target $\mathbf{7}$, \mathbf{n} nums[left - 1]**可能是**target。

至于为什么 left 的更新必须是 left = mid + 1,同左侧边界搜索,就不再赘述。

3. 为什么没有返回 -1 的操作? 如果 nums 中不存在 target 这个值,怎么办?

答: 类似之前的左侧边界搜索,因为 while 的终止条件是 left == right,就是说 left 的取值范围是 [0, nums.length],所以可以添加两行代码,正确地返回 -1:

```
while (left < right) {
    // ...
}
if (left == 0) return -1;
return nums[left-1] == target ? (left-1) : -1;</pre>
```

五、最后总结

先来梳理一下这些细节差异的因果逻辑:

第一个, 最基本的二分查找算法:

```
因为我们初始化 right = nums.length - 1
所以决定了我们的「搜索区间」是 [left, right]
所以决定了 while (left <= right)
同时也决定了 left = mid+1 和 right = mid-1

因为我们只需找到一个 target 的索引即可
所以当 nums[mid] == target 时可以立即返回
```

第二个, 寻找左侧边界的二分查找:

```
因为我们初始化 right = nums.length
所以决定了我们的「搜索区间」是 [left, right)
```

```
详解二分查找算法 - murphy gb - 博客园
所以决定了 while (left < right)
同时也决定了 left = mid+1 和 right = mid
因为我们需找到 target 的最左侧索引
所以当 nums[mid] == target 时不要立即返回
而要收紧右侧边界以锁定左侧边界
第三个, 寻找右侧边界的二分查找:
因为我们初始化 right = nums.length
所以决定了我们的「搜索区间」是 [left, right)
所以决定了 while (left < right)
同时也决定了 left = mid+1 和 right = mid
因为我们需找到 target 的最右侧索引
所以当 nums[mid] == target 时不要立即返回
而要收紧左侧边界以锁定右侧边界
又因为收紧左侧边界时必须 left = mid + 1
所以最后无论返回 left 还是 right, 必须减一
如果以上内容你都能理解,那么恭喜你,二分查找算法的细节不过如此。
通过本文, 你学会了:
1. 分析二分查找代码时,不要出现 else,全部展开成 else if 方便理解。
2. 注意「搜索区间」和 while 的终止条件, 如果存在漏掉的元素, 记得在最后检查。
3. 如需要搜索左右边界,只要在 nums[mid] == target 时做修改即可。搜索右侧时需要减一。
就算遇到其他的二分查找变形,运用这几点技巧,也能保证你写出正确的代码。LeetCode Explore
中有二分查找的专项练习,其中提供了三种不同的代码模板,现在你再去看看,很容易就知道这几个
模板的实现原理了。
标签: 算法
  好文要顶
                收藏该文
          关注我
     murphy gb
     关注 - 2
                                              16
                                                       0
     粉丝 - 38
+加关注
```

« 上一篇: j<u>ava内存模型的实现</u> » 下一篇: <u>双指针技巧汇总</u>

posted @ 2019-06-25 12:58 murphy_gb 阅读(29050) 评论(10) 编辑 收藏

评论列表

#1楼 2019-06-25 13:31 焰极天

感谢分享!!!!!

支持(1) 反对(0)

#2楼 2019-06-25 15:11 牧鱼

感谢分享!

支持(1) 反对(0)

#3楼 2020-01-14 21:30 全振宇

非常详细! 谢谢博主

支持(0) 反对(0)

#4楼 2020-02-23 19:23 kreamyu

太厉害了,非常有帮助

支持(0) 反对(0)

#5楼 2020-03-04 12:50 目标120

感谢博主,不仅将框架总结出来,还把里面的注意细节梳理了,很赞!!!

支持(0) 反对(0)

#6楼 2020-03-14 13:38 pusidun

这篇文章把二分说的很清楚了, 尤其是边界

支持(0) 反对(0)

#7楼 2020-03-22 22:10 kevindd

博主,我第一个二分条件写成right = nums.length, while条件写成left < right, 最后一句写成right = mid, 这个对不对,刚才那个写错了

支持(0) 反对(0)

#8楼 2020-03-27 17:54 andrewlee96

没啥好说的, 赞一个

支持(0) 反对(0)

#9楼 [楼主] 2020-04-01 15:04 murphy_gb

@kevindd 这样也是可以的

支持(0) 反对(0)

#10楼 2020-05-12 15:55 Ingrrris

给楼主献花! 讲的真好!

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论,请登录或注册,访问网站首页。

【推荐】了不起的开发者,势不可挡的华为,园子里的品牌专区

【推荐】有道智云周年庆,API服务大放送,注册即送100元体验金!

【推荐】超50万行VC++源码:大型组态工控、电力仿真CAD与GIS源码库

【推荐】开放下载! 《阿里巴巴大数据及AI实战》深度解析典型场景实践



相关博文:

- ·【经典算法——查找】二分查找
- ·可查找重复元素的二分查找算法
- ·二分查找算法
- ·二分查找算法 (JAVA)
- ·二分查找算法
- » 更多推荐...

最新 IT 新闻:

- · 高瓴斥资百亿认购宁德时代 本田也携37亿入股
- ·又是一年3·15, 那些曾经被点名的企业怎么样了?
- · 社交网络鼻祖转型卖二手车 1亿人的人人网消失了?
- ·深度解析: 百度十年战略抉择

·大国隐痛: 做一个操作系统有多难?

» 更多新闻...

Copyright © 2020 murphy_gb Powered by .NET Core on Kubernetes