

16 | 二分查找（下）：如何快速定位IP对应的省份地址？

2018-10-26 王争



16 | 二分查找（下）：如何快速定位IP对应的省份地址？

朗读人：修阳 11'47'' | 5.40M

通过 IP 地址来查找 IP 归属地的功能，不知道你有没有用过？没用过也没关系，你现在可以打开百度，在搜索框里随便输一个 IP 地址，就会看到它的归属地。



这个功能并不复杂，它是通过维护一个很大的 IP 地址库来实现的。地址库中包括 IP 地址范围和归属地的对应关系。

当我们想要查询 202.102.133.13 这个 IP 地址的归属地时，我们就在地址库中搜索，发现这个 IP 地址落在 [202.102.133.0, 202.102.133.255] 这个地址范围内，那我们就可以将这个 IP 地址范围对应的归属地“山东东营市”显示给用户了。

```
1 [202.102.133.0, 202.102.133.255] 山东东营市
2 [202.102.135.0, 202.102.136.255] 山东烟台
3 [202.102.156.34, 202.102.157.255] 山东青岛
4 [202.102.48.0, 202.102.48.255] 江苏宿迁
5 [202.102.49.15, 202.102.51.251] 江苏泰州
6 [202.102.56.0, 202.102.56.255] 江苏连云港
```

现在我的问题是，在庞大的地址库中逐一比对 IP 地址所在的区间，是非常耗时的。假设我们有 12 万条这样的 IP 区间与归属地的对应关系，如何快速定位出一个 IP 地址的归属地呢？

是不是觉得比较难？不要紧，等学完今天的内容，你就会发现这个问题其实很简单。

上一节我讲了二分查找的原理，并且介绍了最简单的一种二分查找的代码实现。今天我们来讲几种二分查找的变形问题。

不知道你有没有听过这样一个说法：“十个二分九个错”。二分查找虽然原理极其简单，但是想要写出没有 Bug 的二分查找并不容易。

唐纳德·克努特（Donald E.Knuth）在《计算机程序设计艺术》的第 3 卷《排序和查找》中说到：“尽管第一个二分查找算法于 1946 年出现，然而第一个完全正确的二分查找算法实现直到 1962 年才出现。”

你可能会说，我们上一节学的二分查找的代码实现并不难写啊。那是因为上一节讲的只是二分查找中最简单的一种情况，在不存在重复元素的有序数组中，查找值等于给定值的元素。最简单的二分查找写起来确实不难，但是，二分查找的变形问题就没那么好写了。

二分查找的变形问题很多，我只选择几个典型的来讲解，其他的你可以借助我今天讲的思路自己来分析。

4 种常见的二分查找变形问题

- 查找第一个值等于给定值的元素
- 查找最后一个值等于给定值的元素
- 查找第一个大于等于给定值的元素
- 查找最后一个小于等于给定值的元素

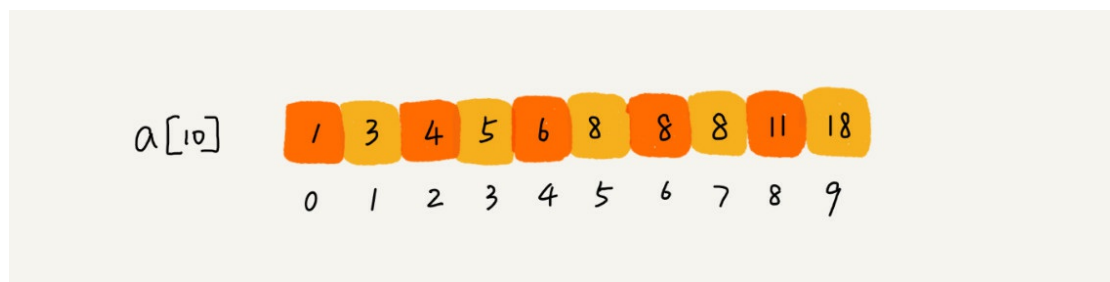
需要特别说明一点，为了简化讲解，今天的内容，我都以数据是从小到大排列为前提，如果你要处

理的数据是从大到小排列的，解决思路也是一样的。同时，我希望你最好先自己动手试着写一下这 4 个变形问题，然后再看我的讲述，这样你就会对我说的“二分查找比较难写”有更加深的体会了。

变体一：查找第一个值等于给定值的元素

上一节中的二分查找是最简单的一种，即有序数据集中不存在重复的数据，我们在其中查找值等于某个给定值的数据。如果我们将这个问题稍微修改下，有序数据集中存在重复的数据，我们希望找到第一个值等于给定值的数据，这样之前的二分查找代码还能继续工作吗？

比如下面这样一个有序数组，其中， $a[5]$ ， $a[6]$ ， $a[7]$ 的值都等于 8，是重复的数据。我们希望查找第一个等于 8 的数据，也就是下标是 5 的元素。



如果我们用上一节课讲的二分查找的代码实现，首先拿 8 与区间的中间值 $a[4]$ 比较，8 比 6 大，于是在下标 5 到 9 之间继续查找。下标 5 和 9 的中间位置是下标 7， $a[7]$ 正好等于 8，所以代码就返回了。

尽管 $a[7]$ 也等于 8，但它并不是我们想要找的第一个等于 8 的元素，因为第一个值等于 8 的元素是数组下标为 5 的元素。我们上一节讲的二分查找代码就无法处理这种情况了。所以，针对这个变形问题，我们可以稍微改造一下上一节的代码。

100 个人写二分查找就会有 100 种写法。网上有很多关于变形二分查找的实现方法，有很多写得非常简洁，比如下面这个写法。但是，尽管简洁，理解起来却非常烧脑，也很容易写错。

```
1 public int bsearch(int[] a, int n, int value) {  
2     int low = 0;  
3     int high = n - 1;  
4     while (low <= high) {  
5         int mid = low + ((high - low) >> 1);  
6         if (a[mid] >= value) {  
7             high = mid - 1;  
8         } else {  
9             low = mid + 1;  
10        }  
11    }  
12  
13    if (a[low]==value) return low;  
14    else return -1;  
15 }
```

[复制代码](#)

看完这个实现之后，你是不是觉得很不好理解？如果你只是死记硬背这个写法，我敢保证，过不了几天，你就会全都忘光，再让你写，90% 的可能会写错。所以，我换了一种实现方法，你看看是不是更容易理解呢？

```
1 public int bsearch(int[] a, int n, int value) {
2     int low = 0;
3     int high = n - 1;
4     while (low <= high) {
5         int mid = low + ((high - low) >> 1);
6         if (a[mid] > value) {
7             high = mid - 1;
8         } else if (a[mid] < value) {
9             low = mid + 1;
10        } else {
11            if ((mid == 0) || (a[mid - 1] != value)) return mid;
12            else high = mid - 1;
13        }
14    }
15    return -1;
16 }
```

我来稍微解释一下这段代码。**a[mid]** 跟要查找的 **value** 的大小关系有三种情况：大于、小于、等于。对于 **a[mid]>value** 的情况，我们需要更新 **high=mid-1**；对于 **a[mid]<value** 的情况，我们需要更新 **low=mid+1**。这两点都很好理解。那当 **a[mid]=value** 的时候应该如何处理呢？

如果我们查找的是任意一个值等于给定值的元素，当 **a[mid]** 等于要查找的值时，**a[mid]** 就是我们要找的元素。但是，如果我们求解的是第一个值等于给定值的元素，当 **a[mid]** 等于要查找的值时，我们就需要确认一下这个 **a[mid]** 是不是第一个值等于给定值的元素。

我们重点看第 11 行代码。如果 **mid** 等于 0，那这个元素已经是数组的第一个元素，那它肯定是我们要找的；如果 **mid** 不等于 0，但 **a[mid]** 的前一个元素 **a[mid-1]** 不等于 **value**，那也说明 **a[mid]** 就是我们要找的第一个值等于给定值的元素。

如果经过检查之后发现 **a[mid]** 前面的一个元素 **a[mid-1]** 也等于 **value**，那说明此时的 **a[mid]** 肯定不是我们要查找的第一个值等于给定值的元素。那我们就更新 **high=mid-1**，因为要找的元素肯定出现在 **[low, mid-1]** 之间。

对比上面的两段代码，是不是下面那种更好理解？实际上，很多人都觉得变形的二分查找很难写，主要原因是太追求第一种那样完美、简洁的写法。而对于我们做工程开发的人来说，代码易读懂、没 Bug，其实更重要，所以我觉得第二种写法更好。

变体二：查找最后一个值等于给定值的元素

前面的问题是查找第一个值等于给定值的元素，我现在把问题稍微改一下，查找最后一个值等于给定值的元素，又该如何做呢？

如果你掌握了前面的写法，那这个问题你应该很轻松就能解决。你可以先试着实现一下，然后跟我写的对比一下。

 复制代码

```
1 public int bsearch(int[] a, int n, int value) {
2     int low = 0;
3     int high = n - 1;
4     while (low <= high) {
5         int mid = low + ((high - low) >> 1);
6         if (a[mid] > value) {
7             high = mid - 1;
8         } else if (a[mid] < value) {
9             low = mid + 1;
10        } else {
11            if ((mid == n - 1) || (a[mid + 1] != value)) return mid;
12            else low = mid + 1;
13        }
14    }
15    return -1;
16 }
```

我们还是重点看第 11 行代码。如果 `a[mid]` 这个元素已经是数组中的最后一个元素了，那它肯定是要我们找的；如果 `a[mid]` 的后一个元素 `a[mid+1]` 不等于 `value`，那也说明 `a[mid]` 就是我们要找的最后一个值等于给定值的元素。

如果我们经过检查之后，发现 `a[mid]` 后面的一个元素 `a[mid+1]` 也等于 `value`，那说明当前的这个 `a[mid]` 并不是最后一个值等于给定值的元素。我们就更新 `low=mid+1`，因为要找的元素肯定出现在 `[mid+1, high]` 之间。

变体三：查找第一个大于等于给定值的元素

现在我们再来看另外一类变形问题。在有序数组中，查找第一个大于等于给定值的元素。比如，数组中存储的这样一个序列：3，4，6，7，10。如果查找第一个大于等于 5 的元素，那就是 6。

实际上，实现的思路跟前面的那两种变形问题的实现思路类似，代码写起来甚至更简洁。

 复制代码

```
1 public int bsearch(int[] a, int n, int value) {
2     int low = 0;
3     int high = n - 1;
4     while (low <= high) {
5         int mid = low + ((high - low) >> 1);
6         if (a[mid] >= value) {
7             if ((mid == 0) || (a[mid - 1] < value)) return mid;
8             else high = mid - 1;
9         } else {
10            low = mid + 1;
11        }
12    }
13    return -1;
14 }
```

如果 `a[mid]` 小于要查找的值 `value`，那要查找的值肯定在 `[mid+1, high]` 之间，所以，我们更新 `low=mid+1`。

对于 `a[mid]` 大于等于给定值 `value` 的情况，我们要先看下这个 `a[mid]` 是不是我们要找的第一个值大于等于给定值的元素。如果 `a[mid]` 前面已经没有元素，或者前面一个元素小于要查找的值 `value`，那 `a[mid]` 就是我们要找的元素。这段逻辑对应的代码是第 7 行。

如果 `a[mid-1]` 也大于等于要查找的值 `value`，那说明要查找的元素在 `[low, mid-1]` 之间，所以，我们将 `high` 更新为 `mid-1`。

变体四：查找最后一个小于等于给定值的元素

现在，我们来看最后一种二分查找的变形问题，查找最后一个小于等于给定值的元素。比如，数组中存储了这样一组数据：**3, 5, 6, 8, 9, 10**。最后一个小于等于 **7** 的元素就是 **6**。是不是有点类似上面那一种？实际上，实现思路也是一样的。

有了前面的基础，你完全可以自己写出来了，所以我就不详细分析了。我把代码贴出来，你可以写完之后对比一下。

```
1 public int bsearch7(int[] a, int n, int value) {
2     int low = 0;
3     int high = n - 1;
4     while (low <= high) {
5         int mid = low + ((high - low) >> 1);
6         if (a[mid] > value) {
7             high = mid - 1;
8         } else {
9             if ((mid == n - 1) || (a[mid + 1] > value)) return mid;
10            else low = mid + 1;
11        }
12    }
13    return -1;
14 }
```

 复制代码

解答开篇

好了，现在我们回头来看开篇的问题：如何快速定位出一个 **IP** 地址的归属地？

现在这个问题应该很简单了。如果 **IP** 区间与归属地的对应关系不经常更新，我们可以先预处理这 **12** 万条数据，让其按照起始 **IP** 从小到大排序。如何来排序呢？我们知道，**IP** 地址可以转化为 **32** 位的整型数。所以，我们可以将起始地址，按照对应的整型值的大小关系，从小到大进行排序。

然后，这个问题就可以转化为我刚讲的第四种变形问题“在有序数组中，查找最后一个小于等于某个给定值的元素”了。

当我们要查询某个 **IP** 归属地时，我们可以先通过二分查找，找到最后一个起始 **IP** 小于等于这个 **IP** 的 **IP** 区间，然后，检查这个 **IP** 是否在这个 **IP** 区间内，如果在，我们就取出对应的归属地显示；如果不在，就返回未查找到。

内容小结

上一节我说过，凡是用二分查找能解决的，绝大部分我们更倾向于用散列表或者二叉查找树。即便是二分查找在内存使用上更节省，但是毕竟内存如此紧缺的情况并不多。那二分查找真的没什么用处了吗？

实际上，上一节讲的求“值等于给定值”的二分查找确实不怎么会被用到，二分查找更适合用在“近似”查找问题，在这类问题上，二分查找的优势更加明显。比如今天讲的这几种变体问题，用其他数据结构，比如散列表、二叉树，就比较难实现了。

变体的二分查找算法写起来非常烧脑，很容易因为细节处理不好而产生 **Bug**，这些容易出错的细节

有：终止条件、区间上下界更新方法、返回值选择。所以今天的内容你最好能用自己实现一遍，对锻炼编码能力、逻辑思维、写出 **Bug free** 代码，会很有帮助。

课后思考

我们今天讲的都是非常规的二分查找问题，今天的思考题也是一个非常规的二分查找问题。如果有序数组是一个循环有序数组，比如 4, 5, 6, 1, 2, 3。针对这种情况，如何实现一个求“值等于给定值”的二分查找算法呢？

欢迎留言和我分享，我会第一时间给你反馈。



版权归极客邦科技所有，未经许可不得转载

写留言

精选留言



Smallfly

👍 13

有三种方法查找循环有序数组

一、

1. 找到分界下标，分成两个有序数组
2. 判断目标值在哪个有序数据范围内，做二分查找

二、

1. 找到最大值的下标 x ;
2. 所有元素下标 $+x$ 偏移，超过数组范围值的取模;
3. 利用偏移后的下标做二分查找;
4. 如果找到目标下标，再作 $-x$ 偏移，就是目标值实际下标。

两种情况最高时耗都在查找分界点上，所以时间复杂度是 $O(N)$ 。

复杂度有点高，能否优化呢？

三、

我们发现循环数组存在一个性质：以数组中间点为分区，会将数组分成一个有序数组和一个循环有

序数组。

如果首元素小于 **mid**，说明前半部分是有序的，后半部分是循环有序数组；
如果首元素大于 **mid**，说明后半部分是有序的，前半部分是循环有序的数组；
如果目标元素在有序数组范围中，使用二分查找；
如果目标元素在循环有序数组中，设定数组边界后，使用以上方法继续查找。

时间复杂度为 $O(\log N)$ 。

2018-10-27



Victor

👍 2

今天的IP地址归属地问题，从工程实现的角度考虑，我更偏向于直接使用关系型数据库实现。
也就是将12w条归属地与IP区间的开始、结束存入数据库中。

数据库表ip_table有如下字段：area_name | start_ip | end_ip，start_ip及end_ip均建立索引
SQL语句：

```
select area_name from ip_table where input_ip >= start_ip and input_ip <= end_ip;
```

学习算法的课程常常和自己工程开发的实际结合在一起，感觉两者是相互促进理解的过程。

2018-10-27

作者回复

数据库可以 单性能会受限

2018-10-28



charon

👍 2

用JavaScript实现的最基本的思考题：

array是传入的数组，value是要查找的值

思路是通过对比low,high的值来判断value所在的区间，不用多循环一遍找偏移量了~

```
function search(array,value){
  let low = 0;
  let high = array.length - 1;

  while(low <= high){
    let mid = low + ((high - low) >> 1);
    if(value == array[low]) return low;
    if(value == array[high]) return high;
    if(value == array[mid]) return mid;

    if(value > array[mid] && value > array[high] && array[mid] < array[low]){
      high = mid - 1;
    }else if(value < array[mid] && value < array[low] && array[mid] < array[low]){
      high = mid - 1;
    }else if(value < array[mid] && value > array[low]){
      high = mid - 1;
    }else{
      low = mid + 1;
    }
  }

  return -1
}
```

2018-10-26



komoU104



给原来的index加上偏移量。

比如原来的二分查找代码从0开始到n-1结束，现在为x到x - 1 (即n-1+x-n)。

x为开始循环处的索引，例子里为3（1所在索引）。需要扫描一遍数组找到x，复杂度O(n)。其余和普通二分查找一样，需要多判断index not out of bound。如果索引超过n了要减n。

总的复杂度还是O(n)

2018-10-26



Jeson



#查找第一个值等于给定值的元素

```
def bsearchFirst(nums, val):
    low, high = 0, len(nums) - 1
    while low <= high:
        mid = low + ((high - low) >> 1)
        if nums[mid] >= val:
            high = mid - 1
        else:
            low = mid + 1
        if nums[low] == val:
            return low
    else:
        return None
```

#查找最后一个值等于给定值的元素

```
def bsearchLast(nums, val):
    low, high = 0, len(nums) - 1
    while low <= high:
        mid = low + ((high - low) >> 1)
        if nums[mid] <= val:
            low = mid + 1
        else:
            high = mid - 1
        if nums[high] == val:
            return high
    else:
        return None
```

#查找第一个大于等于给定值的元素

```
def bsearchFirstLargerEqual(nums, val):
    low, high = 0, len(nums) - 1
    while low <= high:
        mid = low + ((high - low) >> 1)
        if nums[mid] >= val:
            high = mid - 1
        else:
            low = mid + 1
        if nums[low] >= val:
            return low
    else:
        return None
```

```

#查找最后一个小于等于给定值的元素
def bSearchLastsmallerEqual(nums, val):
    low, high = 0, len(nums) - 1
    while low <= high:
        mid = low + ((high - low) >> 1)
        if nums[mid] <= val:
            low = mid + 1
        else:
            high = mid - 1
    if nums[high] <= val:
        return high
    else:
        return None

#循环数组的二分查找
def bSearchRecycle(nums, val):
    n, offset = len(nums), 3
    low, high = 0, n - 1
    while low <= high:
        mid = low + ((high - low) >> 1)
        midIdx = (mid + offset) % n
        if nums[midIdx] == val:
            return midIdx
        elif nums[midIdx] < val:
            low = mid + 1
        else:
            high = mid - 1
    return None

if __name__ == '__main__':
    nums = [0, 0, 2, 2, 3, 3]
    val = 1
    print(bSearchLastsmallerEqual(nums, val))

    nums = [4, 5, 6, 1, 2, 3]
    val = 3
    print(bSearchRecycle(nums, val))

```

2018-10-28



勤劳的小胖子-libo

👍 1

1. 先二分遍历找到分隔点index，特征是<pre, >=next;
 2. 把数组分成二个部分，[0,index-1], [index,length-1];
 3. 分别使用二分查找，找到给定的值。
- 时间复杂度为 $2 \times \log(n)$ 。不确定有什么更好的办法。

2018-10-27



fiseasky

👍 1

二分查找的变体问题，在java sdk、net framework中有实现吗？

2018-10-27

作者回复

c++有的

2018-10-28



淤白

👍 1

1. 通过二分法算出偏移量；
2. 通过偏移量使 $[0, n-1]$ 和现在有序数组的下标关联起来；
3. 通过二分法算出结果；
4. 对结果进行偏移处理拿到最终位置。

2018-10-26



蒋礼锐

👍 1

留言有个地方写错了，不应该在 n/i 二分，应该是 $A(j-i)$ 到 $A(j)$ 。还请老师指点这样的思路是否正确

2018-10-26



tszzsk

👍 1

第二段代码：if $((mid == 0) || (a[mid - 1] != value))$ 可否改为 if $((mid == low) || (a[mid - 1] != value))$ ？以及下面变体？

2018-10-26



He110

👍 1

觉得在查找到值之后，使用 `while(arr[mid-1] == value) mid--`，这种可能好些，就是二分转遍历，如果数据量大而重复的数据量的个数不多的话，这种可能更有优势，如果是十个数据里面七八个需要查找的数据这种就肯定是二分，但是这种的话，直接遍历可能也不慢

2018-10-26

作者回复

有大量重复数据时 就慢了

2018-10-26



Monday

👍 0

二分的四种变种写法。个人觉得都是分三种情况进行讨论，再多注意判断边界值，三种情况为：

$a[mid] < value$

$a[mid] = value$

$a[mid] > value$;

思考题：自己思考了一小段时间没有好的思路，就各学友的留言，亲自实现了 **Smallfly** 的方法三，感觉比较好！谢谢！

2018-10-29



狼的诱惑

👍 0

如何查找后一个大于给定值

2018-10-29



Infinite_gao

👍 0

关于变种的3和4，在3的基础上稍加改造，第一个 \geq 给定值的前一个元素的位置就是最后一个 $<$ 给定值的位置

2018-10-29



唯她命

👍 0

讲的不错

2018-10-29



拉欧

👍 0

这道题之前做过，主要难点在于区分有序区间，其他和常规2分相同

2018-10-29



新手

👍 0

Rust实现的在循环有序无重复数组中查找值：

```
/// Binary search the element which equals to value in a sorted cycled slice
///
/// # Arguments
/// * `a` - A sorted cycled slice that hold unique values
/// * `v` - The value to be searched
///
/// # Return
/// Index of the value
///
/// # Examples
/// ```
/// extern crate binary_search;
/// use binary_search::binary_search_cycled;
///
/// let a = vec![5, 6, 7, 8, 9, 1, 2, 3, 4];
/// assert_eq!(binary_search_cycled(&a, 9), Some(4));
/// assert_eq!(binary_search_cycled(&a, 10), None);
/// ```
pub fn binary_search_cycled<T: PartialOrd>(a: &[T], v: T) -> Option<usize> {
    let mut low = 0i32;
    let mut high = a.len() as i32 - 1;

    while low <= high {
        // prevent overflow
        let mid = low + ((high - low) >> 1);
        let mid_val = &a[mid as usize];

        if *mid_val == v {
            return Some(mid as usize);
        } else if a[low as usize] <= *mid_val {
            // left side is ordered, right side is ordered or cycled
            if *mid_val > v && a[low as usize] <= v {
                high = mid - 1;
            } else {
                low = mid + 1;
            }
        } else {
            // left side is ordered or cycled, right side is ordered
            if *mid_val < v && a[high as usize] >= v {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
    }
}
```

}

None

}

2018-10-28



Jerry银银

👍 0

我提一个无关算法问题的问题：

Donald E.Kunth有一个正式的中文名，叫：高德纳。顺便提一句：他的夫人也有一个中文名，叫：高精兰(Jill)

2018-10-27

作者回复

哈哈👍

2018-10-28



Jerry银银

👍 0

对今天的题目有点疑问：

「有序数组是个循环有序数组」，给得例子是这样[4,5,6,1,2,3]，一定是前面环中的数总大于后面环的数吗？有可能是这样吗：[4,5,6,1,2,3,9,10,11,18,17,16,6,7,8]

2018-10-27

作者回复

你这个还是循环有序的吗

2018-10-28



liangjf

👍 0

一开始，对于查找第一个等于查找值的下标这个问题的第二个简例，对于十一行代码，一直理解不了。想到某个情况是，mid是目标查找值多少钱，mid-1非目标查找值，mid-2是目标查找值，那11行代码是错误的啊。

后来马上想起，这是二分查找算法，上面假设是不成立的，因为二分查找面对的数据是有序的。

2018-10-27