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。

复制代码

```
[202.102.133.0, 202.102.133.255]  山东东营市
[202.102.135.0, 202.102.136.255]  山东烟台
[202.102.156.34, 202.102.157.255] 山东青岛
[202.102.48.0, 202.102.48.255] 江苏宿迁
[202.102.49.15, 202.102.51.251] 江苏泰州
[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。

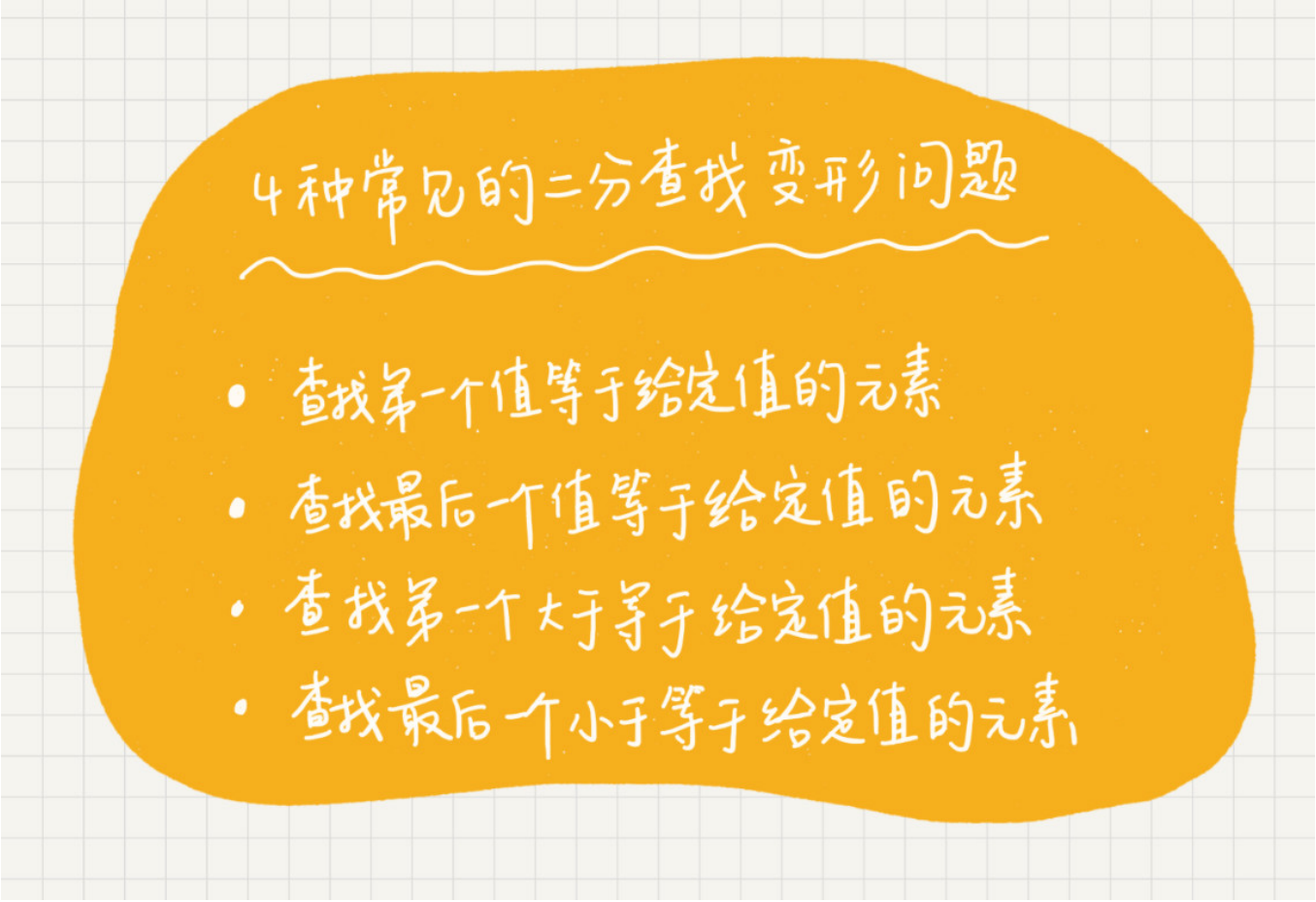
上一节我讲了二分查找的原理，并且介绍了最简单的一种二分查找的代码实现。今天我们来讲几种二分查找的变形问题。

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。

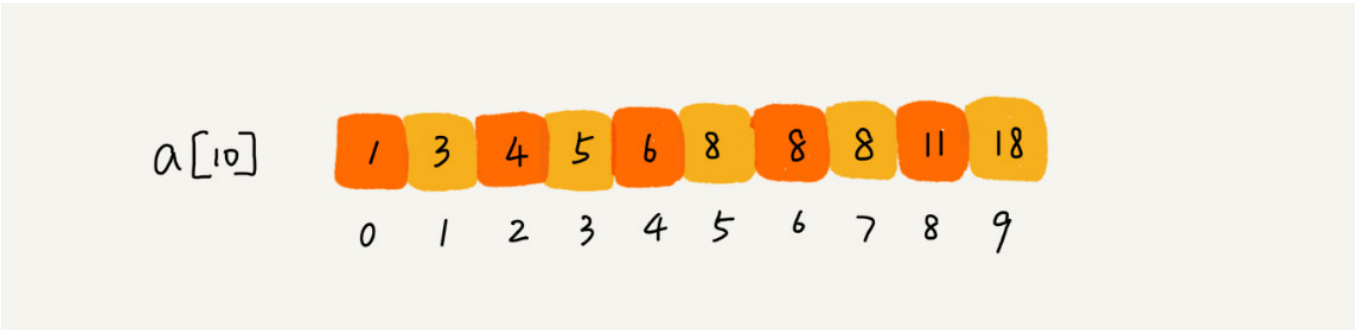


需要特别说明一点，为了简化讲解，今天的内容，我都以数据是从小到大排列为前提，如果你要处理的数据是从大到小排列的，解决思路也是一样的。同时，我希望你最好先自己动手试着写一下这 4 个变形问题，然后再看我的讲述，这样你就会对我说的“二分查找比较难写”有更加深的体会了。

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，即有序数据集合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，有序数据集合中存在重复的数据，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。

复制代码

```
public int bsearch(int[] a, int n, int value) {
  int low = 0;
  int high = n - 1;
  while (low <= high) {
    int mid = low + ((high - low) >> 1);
    if (a[mid] >= value) {
      high = mid - 1;
    } else {
      low = mid + 1;
    }
  }

  if (a[low]==value) return low;
  else return -1;
}
```

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？

复制代码

```
public int bsearch(int[] a, int n, int value) {
  int low = 0;
  int high = n - 1;
  while (low <= high) {
    int mid =  low + ((high - low) >> 1);
    if (a[mid] > value) {
      high = mid - 1;
    } else if (a[mid] < value) {
      low = mid + 1;
    } else {
      if ((mid == 0) || (a[mid - 1] != value)) return mid;
      else high = mid - 1;
    }
  }
  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，那它肯定是我们要找的；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。

复制代码

```
public int bsearch7(int[] a, int n, int value) {
  int low = 0;
  int high = n - 1;
  while (low <= high) {
    int mid =  low + ((high - low) >> 1);
    if (a[mid] > value) {
      high = mid - 1;
    } else {
      if ((mid == n - 1) || (a[mid + 1] > value)) return mid;
      else low = mid + 1;
    }
  }
  return -1;
}
```

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。

写留言

精选留言

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

komo0104 2

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

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*log(n). 不确定有什么更好的办法。

2018–10–27

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二分时，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

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了，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

YellowMax 0

思考题思路：两次二分查找

1.很容易判断循环数组是递增还是递减的（第一个值与第二个值比较或者第一个与最后一个值比较），这一步时间复杂度认为是是 O(1);

2.第一次二分（假设是递增的），那就找到第一个小于给定值（下标为0的数）的数，该操作不需要跟查找给定值一样刻意区分到底该往右还是往左（比下标为0大就往右，小就往左），记录下标，这一步时间复杂度是 O(logn);

3.第二次二分将第二部找到的下标记为 low，把该值加上数组长度减一作为 high，用唤醒缓冲区加取模的方式二分查找给定的值，这一步的时间复杂度是 O(logn);

合计 O(1)+2O(logn)，用遍历的方式找循环点个人感觉不太可行，既然是遍历，为什么不直接遍历给定值呢？虽说循环点比整个数组长度小，但是这是概率性事件，我认为它的概率跟要找的值恰好在循环体后一半的概率差不多，所以先遍历循环点再二分不如直接遍历查找给定值。

2018-10-27

Smallfly 0

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(logN)。

2018-10-27

fiseasky 0

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

2018-10-27

猫头鹰爱拿铁 0

刚动手把今天说的全写了一遍，变形真的好容易写错啊，特别是放在一快写。 我感觉掌握了普通的二分针对变形的二分要点就是找first的就是要尽可能的降低高位指针，然后关注index有没有比0小，找last就是尽可能的增大低位指针，然后关注index有没有高位越界。
题目的解法是这样先遍历一遍数组，找到边界index，然后在0和index以及index+1到array.length-1这里分别进行二分查找。如果第一个找到了就不找第二个了。
return (key=find(0,index,value))==-1?find(index+1,array.length-1,value):key;

2018-10-26

铁皮 0

我发现第一段代码(简洁版)好像有bug。
测试数组int[] nums = new int[] {1, 3, 3, 3, 3, 6, 7, 9, 12, 14, 18};
执行bsearch(nums, nums.length, 20);
出现java.lang.ArrayIndexOutOfBoundsException: 11
但是执行bsearch(nums, nums.length - 1, 20);就没有问题。

但是第二段代码就是你提倡的写法，执行bsearch(nums, nums.length, 20); 就没有问题。
但是执行bsearch(nums, nums.length - 1, 18); 返回"-1"。就不正确了。

可能还需要大量的测试

2018-10-26

charon 0

思考题：
我觉得不用找数组的偏移量，通过和low,heig,mid三个值的对比，也是可以确定要找的数值所以的区间的~

2018-10-26

Kudo 0

Python实现：

```
# 1. 查找第一个值等于给定值的元素
def bsearch(a, value):
low, high = 0, len(a)-1
while low <= high:
mid = low + (high - low) // 2
if a[mid] < value:
low = mid + 1
elif a[mid] > value:
high = mid - 1
elif a[mid] == value:
if mid == 0 or a[mid-1] != value:
return mid
else:
high = mid - 1
return -1

# 2.查找第一个值大于等于给定值的元素
def bsearch(a, value):
low, high = 0, len(a)-1
while low <= high:
mid = low + (high - low) // 2
if a[mid] < value:
low = mid + 1
if a[mid] >= value:
if mid == 0 or a[mid-1] < value:
return mid
else:
high = mid - 1
return -1
```

2018-10-26

Geek_9c1fcf 0

1.将循环数组划分成两个区间，第一个区间的数值都大于等于第二个区间，第一个区间的下标都小于第二个区间；首先比较数组中第一个位置的值a[0]和要查找的值val，若相等则直接return;若val大于a[0],则val位于第一区间，若val小于a[0],则val位于第二区间；
2.确定第一和第二区间分界线位置x。可知分界线位置x满足,a[x]>a[x+1] 条件；通过如下方法

查找位置x
int low=0,high=n-1;
while(low<=high){
int mid=low+(high-low)>>1;
if(a[mid]>a[mid+1]) {
x=mid;
return x;
}
else if(a[mid]>a[low]){
low=mid+1;
}
else{
high=mid-1;
}
}
3.利用所确定的区间和x，在此区间内应用二分查找法计算"值等于给定值"。所确定区间为[0, x] 或[x+1,n-1]

2018-10-26

Sharry 0

思考题
int cycleArraySearch(int *arr, int len, int val) {
// 计算循环周期
int cycle = len - 1;
for (int i = 0; i < len - 1; i++) {
if (arr[i] > arr[i + 1]) {
cycle = i + 1;
break;
}
}
// 确定元素位于的区间
int low = 0, high = low + cycle -1;
while (high <= len - 1) {
// 找到区间后, 进行二分查找
if (arr[low] <= val && arr[high] >= val) {
return binarySearch(arr, low, high, val);
}
low += cycle;
high += cycle;
}
return -1;
}
1. 计算循环周期

2. 确定元素位于的区间
3. 确定了区间后, 进行二分查找
时间复杂度为: O(n)

2018-10-26

🐱您的好友William🐱 0

不行不行，我之前想法有问题，既然已经遍历连接点了那么连接点之前的那段其实就已经查找过了，所以如果遍历到连接点没有找到就在后面用二分。这样做会保证算法复杂度比O（n）小，但区别不大。

2018-10-26

传说中的成大大 0

关于思考题 竟然是循环数组 可以把他看作一个环，我们取出最小的那个作为环的起点和终点 可以通过起始坐标点求得中点坐标即可求出最小的那个点，用两个指针指向一个从中点往前走，一个指针从中点往后走,然后再根据要求值的大小选择挪动指针，即可得出一个区间
伪代码:
假设通过数组起点和终点坐标 p 向左的指针为pl 向右的指针为pr 要查找的值val
如果pr > val && val < pl 则pl –= 1 pr+=1
else if pr > val || pr < val则说明不存了
所以整个算法复杂度为O(logn)
通过这种方法 把环分成两部分 满足条件部分和不满足条件部分

2018-10-26

🐱您的好友William🐱 0

思考题，是循环有序，那么我们先找到循环连接点的位置，遍历算法复杂度为O（n）。找到位置后我们就可以设置low=0，如果a[low]小于要查找的数就在连接点前面找，如果a[low]大于要查找的数就在连接点后面查找，复杂度是O（logn）。

2018-10-26

tszzsk 0

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

蒋礼锐 0

因为数组循环有序的特殊数组，所以首先要找到是多少为一循环，先假设每个循环之间是从小到大排列，每个循环内从大到小排列的数组为A,需要查找的数为value

从零遍历，step为1，如果A[i]<A[i+1]，那么循环就是每i个一循环

在n/i中使用二分查找，中间点一定是为i的倍数，终止条件为A[j*i]>value, 则value值一定在j到j+i之间，因为还是有序，再用一次二分即可找到。

时间复杂度:
第一次遍历找循环，(不知道有没有可以优化的方法)，复杂度为O(N)，假设循环为c
第二次二分，log(n/c)，n/c是因为不会查找除了整数位以外的
第三次log(c)
所以整体的时间复杂度应该是o(N)

空间复杂度为1，没有开辟新数组

2018-10-26

一周 0

思考题 首先便利一下找到最大值节点mid，然后，用第一个数值a[0]，最后一个数值a[n-1]和需要找的数值进行比较，确定是在哪个区域，然后在对应区域进行二分查找

2018-10-26

刘忽悠 0

第一段代码的想法确实妙，乍眼一看还以为这段代码有问题

2018-10-26