**[关于二分查找的思想](http://blog.csdn.net/u012333003/article/details/23951143)（加灰部分重要）**

**一、简述**

  在学习一个算法时我们要弄懂算法的原理，应用背景和实现，还有就是要学会用大O理论分析算法的时间复杂度，现在我们常碰到的时间复杂度通常为:O(1),O(logn),O(n),O(n\*n)......O(2的n次方)，时间复杂法为O(log)的目前所学算法中为二分法莫属，而且，在一系列的其他算法诸如排序等，因为二分的思想而常使得其时间复杂度降为O(nlogn)，而一般情况下是O(n\*n)。

然而，二分法虽然高效，但很具有局限性，用于二分查找的序列必须具有如下特征：

1.存储在数组中，在链表中就不能实现二分查找了。

2.序列有序

**二、二分法查找的实现**

Bentleley说，只有百分之十的人能给出二分法的正确实现，所以别小看它：

**[cpp]** [view plaincopy](http://blog.csdn.net/u012333003/article/details/23951143)[在CODE上查看代码片](https://code.csdn.net/snippets/298679)

1. #include<iostream>
2. **using** **namespace** std;
3. **int** BinarySearch(**const** **int** array[], **size\_t** low, **size\_t** high, **int** target)
4. {
5. **if** (low > high) **return** -1;
6. **int** mid = (low + high) / 2;
7. **if** (array[mid] == target)
8. **return** mid;
9. **else** **if** (array[mid] > target)
10. **return** BinarySearch(array, low, mid - 1, target);
11. **else**
12. **return** BinarySearch(array, mid + 1, high, target);
13. }
14. **int** main()
15. {
16. **int** a[5] = { 1, 2, 3, 4, 5 };
17. cout << BinarySearch(a, 0, 4, 6) << endl;
18. **return** EXIT\_SUCCESS;
19. }

基于所有递归实现都可以用栈来解递归的思想，我们可以非递归实现该函数如下：

//search for exact number

**public** **int** searchNumber(**int**[] A, **int** target) {

**int** begin = 0;

**int** end = A.length-1; //注意边界，否则找到的值如果在最大值右边，会出现边界溢出

**while** (begin <= end) {

**int** mid = (begin + end) / 2;

**if** (target > A[mid])

begin = mid + 1;

**else** **if** (target < A[mid])

end = mid - 1;

**else**

**return** mid;

}

**return** -1;

}

* 算法一： mid = (low + high) / 2

不好。在数学上这的确没有什么不同，但是在计算机里，却有很大的不同。设想你的数组很大，为方便说明，我以16位为例子，一个int型整数的最大值为65535，如果你的数组大小为40000，而你要需要查找的数据位于数组的比较后的位置，例如是下标为39800的那个数，那么在后面的查找中（begin+end）就会超出65535所能表示的范围，从而mid就会变成一个负数，这样你就永远都找不到你要找的数了，还可能发生内存错误。

* 算法二：mid = begin/2 + end/2;
* 算法三： mid = low + (high – low)/2

乍看起来，算法一简洁，算法二提取之后，跟算法一没有什么区别。但是实际上，区别是存在的。算法一的做法，在极端情况下，(low + high)存在着溢出的风险，进而得到错误的mid结果，导致程序错误。而算法二能够保证计算出来的mid，一定大于low，小于high，不存在溢出的问题

* 把加1和减1漏了

再来看看注释3和注释4，很多人在缩小搜索范围时，都会把加1和减1漏了，这会导致一个什么的后果呢，就是这个程序可能会陷入死循环，这也是一个常有的错误。而为什么可以加1和减1，因为在查找失败时，mid肯定不会等于value，所以可以放心地从mid的下一个元素（加1）或mid的前一个元素（减1）开始查找。

* 注意和找下界的区别：

首先，找下界派断条件是<，而找准确值是<=。其次，找下界是end=mid，准确值是end=mid-1。还有，找下界，end是从length开始，找准确值，end是从length-1开始。本质上，两个是等价的。但是，找准确值，如果不存在，可以返回-1。找下界，不存在，返回值是0或者**length**。

**三、用二分法寻找边界值**

有时我们并不是要寻找目标值，而是寻找到第一个大于给定值得值或者第一个小于给定值得值。

**public** **int** upper\_bound(**int**[] A, **int** target) {

**int** begin = 0;

**int** end = A.length;

**while** (begin < end) {

**int** mid = (begin + end) / 2;

**if** (target >= A[mid])

begin = mid + 1;

**else**

end = mid;

}

**return** begin-1;

}

这里和准确查找的不同之处在于准确查找包含三个分支，等于目标，大于目标，小于目标分别处理，而上界查找则只有大于和不大于两种情况，中间值大于，说明大索引还可以往中间靠拢，但不能越过该中间值，有可能该中间值就是第一个大于目标的呢。同样我们可以写出求下界的函数，如下：

**public** **int** lower\_bound(**int**[] A, **int** target) {

**int** begin = 0;

**int** end = A.length;

**while** (begin < end) {

**int** mid = (begin + end) / 2;

**if** (target > A[mid])

begin = mid + 1;

**else**

end = mid;

}

**return** begin;

}  
类似的，当寻找松散边界时也只要做简单地修改即可实现。

四、二分查找的缺陷：

二分查找的效率令人向往，但固有的缺陷亦也使人恼火，首先它要求是数组，其次还要求有序，在对元素进行删除增加操作时很耗时，时间复杂度为O(n)，而一种好的办法是使用二叉查找树，它能在O(nlogn)内构建树，也能在O(logn)内查找目标。