**二分查找法汇总**

在学习算法的过程中，我们除了要了解某个算法的基本原理、实现方式，更重要的一个环节是利用[big-O理论](http://en.wikipedia.org/wiki/Big_O_notation)来分析算法的复杂度。在时间复杂度和空间复杂度之间，我们又会更注重时间复杂度。

时间复杂度按优劣排差不多集中在：

O(1), O(log n), O(n), O(n log n), O(n2), O(nk), O(2n)

到目前位置，似乎我学到的算法中，时间复杂度是O(log n),好像就数[二分查找法](http://en.wikipedia.org/wiki/Binary_search_algorithm)，其他的诸如排序算法都是 O(n log n)或者O(n2)。但是也正是因为有二分的 O(log n), 才让很多 O(n2)缩减到只要O(n log n)。

**关于二分查找法**

[二分查找法](http://en.wikipedia.org/wiki/Binary_search_algorithm)主要是解决在“一堆数中找出指定的数”这类问题。

而想要应用二分查找法，这“一堆数”必须有一下特征：

* 存储在数组中
* 有序排列

所以如果是用[链表](http://en.wikipedia.org/wiki/Linked_list)存储的，就无法在其上应用[二分查找法](http://en.wikipedia.org/wiki/Binary_search_algorithm)了。（曽在面试被问二分查找法可以什么数据结构上使用：数组？链表？）

至于是顺序递增排列还是递减排列，数组中是否存在相同的元素都不要紧。不过一般情况，我们还是希望并假设数组是递增排列，数组中的元素互不相同。

**二分查找法的基本实现**

二分查找法在算法家族大类中属于“[分治法](http://en.wikipedia.org/wiki/Divide_and_conquer_algorithm)”，分治法基本都可以用[递归](http://en.wikipedia.org/wiki/Recursion_%28computer_science%29)来实现的，二分查找法的递归实现如下：

[复制代码](javascript:void(0);)

int bsearch(int array[], int low, int high, int target)  
{  
 if (low > high) return -1;  
   
 int mid = (low + high)/2;  
 if (array[mid]> target)  
 return binarysearch(array, low, mid -1, target);  
 if (array[mid]< target)  
 return binarysearch(array, mid+1, high, target);  
   
 //if (midValue == target)  
 return mid;  
}

[复制代码](javascript:void(0);)

不过所有的递归都可以自行定义stack来解递归，所以二分查找法也可以不用递归实现，而且它的非递归实现甚至可以不用栈，因为二分的递归其实是[尾递归](http://en.wikipedia.org/wiki/Tail_call)，它不关心递归前的所有信息。

[复制代码](javascript:void(0);)

int bsearchWithoutRecursion(int array[], int low, int high, int target)  
{  
 while(low <= high)  
 {  
 int mid = (low + high)/2;  
 if (array[mid] > target)  
 high = mid - 1;  
 else if (array[mid] < target)  
 low = mid + 1;  
 else //find the target  
 return mid;  
 }  
 //the array does not contain the target  
 return -1;  
}

[复制代码](javascript:void(0);)

**只用小于比较（<）实现二分查找法**

在前面的二分查找实现中，我们既用到了小于比较（<）也用到了大于比较（>），也可能还需要相等比较（==）。

而实际上我们只需要一个小于比较（<）就可以。因为错逻辑上讲a>b和b<a应该是有相当的逻辑值；而a==b则是等价于 !((a<b)||(b<a))，也就是说a既不小于b，也不大于b。

当然在程序的世界里， 这种[关系逻辑其实并不是完全正确](http://www.parashift.com/c++-faq-lite/newbie.html#faq-29.14)。另外，C++还允许对对象进行[运算符的重载](http://msdn.microsoft.com/en-us/library/5tk49fh2%28VS.71%29.aspx)，因此开发人员完全可以随意设计和实现这些关系运算符的逻辑值。

不过在整型数据面前，这些关系运算符之间的逻辑关系还是成立的，而且在开发过程中，我们还是会遵循这些逻辑等价关系来重载关系运算符。

干嘛要搞得那么羞涩，只用一个关系运算符呢？因为这样可以为二分查找法写一个template，又能减少对目标对象的要求。模板会是这样的：

[复制代码](javascript:void(0);)

template <typename T, typename V>  
inline int BSearch(T& array, int low, int high, V& target)  
{  
 while(!(high < low))  
 {  
 int mid = (low + high)/2;  
 if (target < array[mid])  
 high = mid - 1;  
 else if (array[mid] < target)  
 low = mid + 1;  
 else //find the target  
 return mid;  
 }  
 //the array does not contain the target  
 return -1;   
}

[复制代码](javascript:void(0);)

我们只需要求target的类型V有重载小于运算符就可以。而对于V的集合类型T，则需要有[]运算符的重载。当然其内部实现必须是O(1)的复杂度，否则也就失去了二分查找的效率。

**用二分查找法找寻边界值**

之前的都是在数组中找到一个数要与目标相等，如果不存在则返回-1。我们也可以用二分查找法找寻边界值，也就是说在**有序数组**中找到“正好大于（小于）目标数”的那个数。

用数学的表述方式就是：

     在集合中找到一个大于（小于）目标数t的数x，使得集合中的任意数要么大于（小于）等于x，要么小于（大于）等于t。

举例来说：

给予数组和目标数

int array = {2, 3, 5, 7, 11, 13, 17};  
int target = 7;

那么上界值应该是11，因为它“刚刚好”大于7；下届值则是5，因为它“刚刚好”小于7。

**用二分查找法找寻上届**

[复制代码](javascript:void(0);)

//Find the fisrt element, whose value is larger than target, in a sorted array   
int BSearchUpperBound(int array[], int low, int high, int target)  
{  
 //Array is empty or target is larger than any every element in array   
 if(low > high || target >= array[high]) return -1;  
   
 int mid = (low + high) / 2;  
 while (high > low)  
 {  
 if (array[mid] > target)  
 high = mid;  
 else  
 low = mid + 1;  
   
 mid = (low + high) / 2;  
 }  
  
 return mid;  
}

[复制代码](javascript:void(0);)

与精确查找不同之处在于，精确查找分成三类：**大于**，**小于**，**等于**（目标数）。而界限查找则分成了两类：**大于**和**不大于**。

如果当前找到的数大于目标数时，它可能就是我们要找的数，所以需要保留这个索引，也因此if (array[mid] > target)时 high=mid; 而没有减1。

**用二分查找法找寻下届**

[复制代码](javascript:void(0);)

//Find the last element, whose value is less than target, in a sorted array   
int BSearchLowerBound(int array[], int low, int high, int target)  
{  
 //Array is empty or target is less than any every element in array  
 if(high < low || target <= array[low]) return -1;  
   
 int mid = (low + high + 1) / 2; //make mid lean to large side  
 while (low < high)  
 {  
 if (array[mid] < target)  
 low = mid;  
 else  
 high = mid - 1;  
   
 mid = (low + high + 1) / 2;  
 }  
  
 return mid;  
}

[复制代码](javascript:void(0);)

下届寻找基本与上届相同，需要注意的是在取中间索引时，使用了向上取整。若同之前一样使用向下取整，那么当low == high-1，而array[low] 又小于 target时就会形成死循环。因为low无法往上爬超过high。

这两个实现都是找**严格界限**，也就是要大于或者小于。如果要找松散界限，也就是找到大于等于或者小于等于的值（即包含自身），只要对代码稍作修改就好了：

去掉判断数组边界的等号：

target >= array[high]改为 target > array[high]

在与中间值的比较中加上等号：

array[mid] > target改为array[mid] >= target

**用二分查找法找寻区域**

之前我们使用二分查找法时，都是基于**数组中的元素各不相同**。假如存在重复数据，而数组依然有序，那么我们还是可以用二分查找法判别目标数是否存在。不过，返回的index就只能是随机的重复数据中的某一个。

此时，我们会希望知道有多少个目标数存在。或者说我们希望数组的区域。

结合前面的界限查找，我们只要找到目标数的严格上届和严格下届，那么界限之间（不包括界限）的数据就是目标数的区域了。

[复制代码](javascript:void(0);)

//return type: pair<int, int>  
//the fisrt value indicate the begining of range,  
//the second value indicate the end of range.  
//If target is not find, (-1,-1) will be returned  
pair<int, int> SearchRange(int A[], int n, int target)   
{  
 pair<int, int> r(-1, -1);  
 if (n <= 0) return r;  
   
 int lower = BSearchLowerBound(A, 0, n-1, target);  
 lower = lower + 1; //move to next element  
   
 if(A[lower] == target)  
 r.first = lower;  
 else //target is not in the array  
 return r;  
   
 int upper = BSearchUpperBound(A, 0, n-1, target);  
 upper = upper < 0? (n-1):(upper - 1); //move to previous element  
   
 //since in previous search we had check whether the target is  
 //in the array or not, we do not need to check it here again  
 r.second = upper;  
   
 return r;  
}

[复制代码](javascript:void(0);)

它的时间复杂度是两次二分查找所用时间的和，也就是O(log n) + O(log n)，最后还是O(log n)。

**在轮转后的有序数组上应用二分查找法**

之前我们说过二分法是要应用在**有序**的数组上，如果是无序的，那么比较和二分就没有意义了。

不过还有一种特殊的数组上也同样可以应用，那就是“轮转后的有序数组（Rotated Sorted Array）”。它是有序数组，取期中某一个数为轴，将其之前的所有数都轮转到数组的末尾所得。比如{7, 11, 13, 17, 2, 3, 5}就是一个轮转后的有序数组。非严格意义上讲，有序数组也属于轮转后的有序数组——我们取首元素作为轴进行轮转。

下边就是二分查找法在轮转后的有序数组上的实现（假设数组中不存在相同的元素）

[复制代码](javascript:void(0);)

int SearchInRotatedSortedArray(int array[], int low, int high, int target)   
{  
 while(low <= high)  
 {  
 int mid = (low + high) / 2;  
 if (target < array[mid])  
 if (array[mid] < array[high])//the higher part is sorted  
 high = mid - 1; //the target would only be in lower part  
 else //the lower part is sorted  
 if(target < array[low])//the target is less than all elements in low part  
 low = mid + 1;  
 else  
 high = mid - 1;  
  
 else if(array[mid] < target)  
 if (array[low] < array[mid])// the lower part is sorted  
 low = mid + 1; //the target would only be in higher part  
 else //the higher part is sorted  
 if (array[high] < target)//the target is larger than all elements in higher part  
 high = mid - 1;  
 else  
 low = mid + 1;  
 else //if(array[mid] == target)  
 return mid;  
 }  
  
 return -1;  
}

[复制代码](javascript:void(0);)

对比普通的二分查找法，为了确定目标数会落在二分后的那个部分，我们需要更多的判定条件。但是我们还是实现了O(log n)的目标。

**二分查找法的缺陷**

二分查找法的O(log n)让它成为十分高效的算法。不过它的缺陷却也是那么明显的。就在它的限定之上：

**必须有序**，我们很难保证我们的数组都是有序的。当然可以在构建数组的时候进行排序，可是又落到了第二个瓶颈上：它必须是[数组](http://en.wikipedia.org/wiki/Array_data_structure)。

数组读取效率是O(1)，可是它的插入和删除某个元素的效率却是O(n)。因而导致构建有序数组变成低效的事情。

解决这些缺陷问题更好的方法应该是使用[二叉查找树](http://en.wikipedia.org/wiki/Binary_search_tree)了，最好自然是[自平衡二叉查找树](http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree)了，自能高效的（O(n log n)）构建有序元素集合，又能如同二分查找法一样快速（O(log n)）的搜寻目标数