二分搜索法（上）

“思路很简单，细节是魔鬼”这是Knuth大佬对二分搜索的评价。

在二分搜索中，最令人头疼的就是while到底是该用“<=”还是应该用“<”？mid到底该加1，还是减1？

今天，我们就来彻底弄清楚这个问题。

先给出二分搜索的基本框架：

```
//nums 待搜索数组
//target 目标数字
//length 数组长度
//return 目标所在索引位置
int BinarySearch(int nums[], int target, int length) {

    int left = 0; //左边界

    int right =...;  //根据实际情况赋值

    while (...){  //根据实际的条件进行判断
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            ...;
        } else if (nums[mid] < target) {
            left = ...;
        } else if (nums[mid] > target) {
            right = ...;
        }
    }
    return ...;
}
```

上面的代码共有三处细节需要注意：

1、分析二分搜索的一个技巧是：不要出现else，而是把所有的情况用else if写清楚，这样可以清楚的展现所有的细节。

2、对于上文中“...”省略号的部分，代表的是可能出现细节的地方，我们将在后续的例子中，用实际的代码来说明这些细节和变化。

3、计算mid的时候，需要防止溢出。

代码left + (right - left) / 2 和 (left + right)/2结果是相同的，但是如果left和right太大，直接相加会导致整形溢出，因此建议使用上面的书写方式。

基本示例

给定一个数组，搜索指定数字的索引，如果不存在，则返回-1。

这道题是最基本的二分搜索，根据题意，我们将上面的代码进行修改，得到如下结果：

```
//nums 待搜索数组
//target 目标数字
//length 数组长度
//return 目标所在索引位置
int BinarySearch(int nums[], int target, int length) {

    int left = 0; //左边界

    //细节1：我们将长度进行了减1
    int right = length - 1;

     //开始循环搜索
    while (left <= right) {  //细节2：while中使用了<=
         int mid = left + (right - left) / 2;
         if (nums[mid] == target) {
             return mid;  //细节3：返回mid
         } else if (nums[mid] < target) {
             left = mid + 1;  //细节4 左边界变化
        } else if (nums[mid] > target) {
             right = mid - 1; //细节5 有边界变化
        }
     }
     return -1;
}
```

上述代码中的细节1和2，其实是同一个问题，在这里我们统一进行解释。

首先，我们规定本题的数组存储是从0开始的，因此右边界length是我们永远也无法取到的一个索引。

最初的时刻，我们将搜索的的右边界设置为length-1，这样就保证了我们是在一个闭合的区间[left,right]内进行搜索，那么搜索应该在什么情况下结束呢？

我们很容易想到，终止的情况无外乎两种：

1、找到目标了，直接返回索引

2、找不到目标，即没有可搜索的区间了，我们需要返回-1

那么，对于我们当前的这种写法，没有可搜索的区间状况便是left>right，即left<=right的反面情况。带入数字，当左右区间分别为2的时候，即当前搜索区间为[2,2]的时候，很明显这是一个合法的区间；但是区间[3,2]即为非法。
那么，现在考虑一下，如果我把while的条件改为while(left<right)，会发生什么？

稍加测试，我们就能发现，如果这样进行编写，我们将漏掉一段区间，示例如下：

假设给定的数组为{1,3,5,7,9}，现在我们搜索目标3。

最初的时刻：left = 0，right = 5 - 1 = 4，mid = 0 + (4 - 0) / 2 = 2；

num[mid] = 5 > target，所以右边界缩小，right = mid - 1 = 1；

现在进入第二轮搜索：left = 0，right = 1，mid = 0 + (1 - 0) / 2 = 0；

num[mid] = 1 < target，所以左边界缩小，left = mid + 1 = 1；

注意：此刻的left == right，但是while执行的条件是left<right，因此搜索终止,将会返回-1!而我们的正确答案1，就这样眼睁睁的被抛弃了！

但是，现在的我们明白了问题的根本所在，因此，我们也不必慌乱，只需要在最后的时刻打个补丁即可，即判断一下nums[left]是否等于target，如果等于就返回target，反之则返回-1。

程序修改如下：

```
//nums 待搜索数组
//target 目标数字
//length 数组长度
//return 目标所在索引位置
int BinarySearch(int nums[], int target, int length) {
    int left = 0; //左边界
    //细节1：我们将长度进行了减1
    int right = length - 1;
    //开始循环搜索
    while (left < right) {  //细节2：while中使用了<
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            return mid;  //细节3：返回mid
        } else if (nums[mid] < target) {
            left = mid + 1;  //细节4 左边界变化
```

```
            } else if (nums[mid] > target) {
                right = mid; //细节5 有边界变化
            }
        }
        //打补丁，判断左边界是否满足条件
        return nums[left] == target ? left : -1;
    }
```

在这里，还有一个重要的细节需要讲解。

想必大家已经看到了，在上面的函数中**left=mid+1**，但是**right却等于mid**。以及在前一个函数中left和right却分别等于mid+1，mid-1。大家一定很奇怪，这里为什么一会是mid+1，一会又成了mid，看起来好乱啊！

实际上，这里的逻辑非常清晰，一点都不乱。考虑一下，在第一个函数中，我们搜索的区间为**[left,right]**，是个闭合区间，那么当我们对mid判断完毕后，自然而然的会将其进行排除。排除之后的结果，就是将原来的闭合区间拆分为**[left,mid-1]**以及**[mid+1,right]**。

对于第二个函数，我们原本的搜索区间为**[left,right)**是左闭右开的状态，这样，在排除了mid之后，正确的形态就应该是**[left,mid)**和**[mid+1,right)**，所以才有了上面的left=mid+1和right = mid的写法。

因此，只要我们弄懂了当前的搜索区间是什么，左右边界的取值就是水到渠成的事情。

现在，老师给大家留个问题：如果，在最初的时刻，我们将右边界设置为length，那么结果会发生什么？该如何解决？