二分查找篇

二分查找基础知识：

http://baike.baidu.com/view/610605.htm

在 LeetCode 用到此算法的主要题目有：

1. Search Insert Position
2. Search for a Range
3. Sqrt(x)
4. Search a 2D Matrix
5. Search in Rotated Sorted Array
6. Search in Rotated Sorted Array II
7. Find Minimum in Rotated Sorted Array
8. Find Minimum in Rotated Sorted Array II
9. Find Peak Element
10. Median of Two Sorted Arrays

1. Search Insert Position，l <= r，以上实现方式有一个好处，就是当循环结束时，如果没有找到目标元素，那么 l 一定停在恰好比目标大的 index 上，r 一定停在恰好比目标小的 index 上，所以个人比较推荐这种实现方式。所以直接返回 l 即可。

代码如下：

```
public int searchInsert(int[] A, int target) {
    if(A == null || A.length == 0){
        return 0;
    }
    int l = 0;
    int r = A.length - 1;
    while(l <= r){
        int m = (l + r) / 2;
        if(A[m] == target){
            return m;
        }
        if(A[m] > target){
```

```
                r = m - 1;
            } else {
                l = m + 1;
            }
        }
        return l;
    }
```

2. Search for a Range，如果直接相等的时候也向一个方向继续夹逼，如果向右夹逼，最后就会停在右边界，而向左夹逼则会停在左边界，如此用停下来的两个边界就可以知道结果了，只需要两次二分查找。实现中用到了在Search Insert Position中提到的方法，可以保证当搜索结束时，l和r所停的位置正好是目标数的后面和前面。

代码如下：

```
    public int[] searchRange(int[] A, int target) {
        int[] res = {-1, -1};
        if(A == null || A.length == 0){
            return res;
        }
        int l = 0;
        int r = A.length - 1;
        while(l <= r){
            int m = (l + r) / 2;
            if(A[m] <= target){
                l = m + 1;
            } else {
                r = m - 1;
            }
        }
        int idxR = r;
        l = 0;
        r = A.length - 1;
        while(l <= r){
            int m = (l + r) / 2;
            if(A[m] >= target){
                r = m - 1;
            } else {
                l = m + 1;
            }
```

```
        }
        int idxL = l;
        if(idxL <= idxR){
            res[0] = idxL;
            res[1] = idxR;
        }
        return res;
    }
```

做题时的感悟:

使用 l <= r 做为结束判断条件，当循环停下来时，如果不是正好找到 target，l 指向的元素恰好大于 target，r 指向的元素恰好小于 target，这里 l 和 r 可能越界，不过如果越界就说明大于（小于）target 并且是最大（最小）。我们的目标是在后面找到 target 的右边界，因为左边界已经等于 target，所以判断条件是相等则向右看，大于则向左看，根据上面说的，循环停下来时，l 指向的元素应该恰好大于 target，r 指向的元素应该等于 target，所以此时的 r 正是我们想要的。

3. Sqrt(x)，这是一道数值处理的题目，和Divide Two Integers不同，这道题一般采用数值中经常用的另一种方法：二分法。基本思路是跟二分查找类似，要求是知道结果的范围，取定左界和右界，然后每次砍掉不满足条件的一半，直到左界和右界相遇。算法的时间复杂度是O(logx)，空间复杂度是O(1)。

代码如下:

```
    // Solution 1 – 二分法
    public int sqrt(int x) {
        if(x < 0){
            return -1;
        }
        if(x == 0 || x == 1){
            return x;
        }
        int l = 1;
        int r = x / 2 + 1;
        while(l <= r){
            int m = (l + r) / 2;
            if(m <= x / m && x / (m + 1) < (m + 1)){
```

```
                return m;
            }
            if(m > x / m){
                r = m - 1;
            } else {
                l = m + 1;
            }
        }
        return -1;
    }
```

二分法在数值计算中非常常见，还是得熟练掌握。这个题目还有另一种方法，称为牛顿法。可以参考一下牛顿法-维基百科。一般牛顿法是用数值方法来解一个f(y)=0的方程。牛顿法需要记住公式：

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

代码如下：

```
    // Solution 2 -  牛顿法
    public int sqrt(int x) {
        if(x < 0){
            return -1;
        }
        if(x == 0 || x == 1){
            return x;
        }
        double lastY = 0;
        double y = 1;
        while(y != lastY){
            lastY = y;
            y = (y + x / y) / 2;
        }
        return (int)y;
    }
```

其实，这道题还有一个巧妙的做法就是用位运算来进行求解，而且最多只需要判断16位，所以时间复杂度是O(1).

代码如下：

```
// Solution 3 - Bit Manipulation
public int sqrt(int x) {
    int res = 0;
    int bit = 1 << 16;
    while(bit > 0){
        res |= bit;
        if(res == x / res){
            break;
        } else if(res > x / res){
            res ^= bit;
        }
        bit >>= 1;
    }
    return res;
}
```

做题时的感悟：

1. 注意corner case，例如x == 0时，return 0; l从1开始，r从x / 2 + 1开始。

2. 判断相等的条件不是简单的 m == x/m，而是 m<=x/m && x/(m+1)<m+1，这是因为输出是整型，sqrt(14)=3 但 3 != 14/3. 所以我们需要一个范围框住结果。

3. 判断条件里尽量用除法，避免越界。但一定要确保除数不能为0，不然只能用乘。

4. 根据二分查找算法的特性，如果不能正好m==x/m停下，那么r指向的数字将正好是结果取整的值。所以我们也可以把判断条件设为m==x/m，但return的结果是r。

5. 实际面试遇到的题目可能不是对一个整数开方，而是对一个实数。方法和整数其实是一致的，只是结束条件换成左界和右界的差的绝对值小于某一个epsilon（极小值）即可。

6. 在 java 中我们可以用==来判断两个 double 是否相等，而在 C++中我们则需要通过两个数的绝对值差小于某个极小值来判断两个 double 的相等性。实际上两个 double 因为精度问题往往是不可能每一位完全相等的，java 中只是帮我们做了这种判定。

4. Search a 2D Matrix，这道题总结下来有3种解法，如下：

（1）Linear Search - O(m + n)

（2）Double Binary Search - O(log(m) + log(n))

（3）Divide and Conquer - O(log(n)) - CC 11.6

（1）Linear Search 解法代码最容易实现，但时间复杂度最差，是线性的时间复杂度。思路即是从矩阵右上角开始向左搜索，找到第一个比目标小的元素再向下继续搜索即可。

代码如下：

```
// Linear Search - O(m + n)
public boolean searchMatrix(int[][] matrix, int target) {
    int row = 0;
    int col = matrix[0].length - 1;
    while(row < matrix.length && col >= 0){
        if(matrix[row][col] == target){
            return true;
        } else if(matrix[row][col] > target){
            col--;
        } else {
            row++;
        }
    }
    return false;
}
```

（2）Double Binary Search解法只需要先按行查找，定位出在哪一行之后再进行列查找即可，所以就是进行两次二分查找。时间复杂度是O(logm+logn)，空间上只需两个辅助变量，因而是O(1).

代码如下：

```
// Double Binary Search - O(log(m) + log(n))
public boolean searchMatrix(int[][] matrix, int target){
    if(matrix == null || matrix.length == 0 || matrix[0].length == 0){
        return false;
    }
    int l = 0;
    int r = matrix.length - 1;
    while(l <= r){
        int m = (l + r) / 2;
        if(matrix[m][0] == target){
```

```
                return true;
            } else if(matrix[m][0] > target){
                r = m - 1;
            } else {
                l = m + 1;
            }
        }
        int row = r;
        if(row < 0){
            return false;
        }
        l = 0;
        r = matrix[0].length - 1;
        while(l <= r){
            int m = (l + r) / 2;
            if(matrix[row][m] == target){
                return true;
            } else if(matrix[row][m] > target){
                r = m - 1;
            } else {
                l = m + 1;
            }
        }
        return false;
    }
```

（3）Divide and Conquer 解法时间复杂度最优，但代码过于繁琐。思路是利用左上-右下对角线的中点将矩阵划为 4 块进行分治，代码详见 CC150.

做题时的感悟:

1. Double Binary Search在进行第二次搜索前，要判断第一次搜索的结果row = r是不是合理，如果row < 0直接返回找不到。

2. 要让一个对象进行克隆，浅克隆就是两个步骤：

（1）让该类实现java.lang.Cloneable接口；

（2）重写（override）Object类的clone()方法。

具体关于浅克隆（shallow clone）Vs 深克隆（deep clone）我们会在后面进行讨论。

5. Search in Rotated Sorted Array，假设数组是A，左边缘为l，右边缘为r，中间位置是m。在每次迭代中，分三种情况：

（1）如果target==A[m]，那么m就是我们要的结果，直接返回；

（2）如果A[m]<A[r]，那么说明从m到r一定是有序的，那么我们只需要判断target是不是在m到r之间，如果是则把左边缘移到m+1，否则就target在另一半，即把右边缘移到m-1。

（3）如果A[m]>=A[r]，那么说明从l到m一定是有序的，同样只需要判断target是否在这个范围内，相应的移动边缘即可。

根据以上方法，每次我们都可以切掉一半的数据，所以算法的时间复杂度是O(logn)，空间复杂度是O(1)。

代码如下：

```
public int search(int[] A, int target) {
    if(A == null || A.length == 0){
        return -1;
    }
    int l = 0;
    int r = A.length - 1;
    while(l <= r){
        int m = (l + r) / 2;
        if(A[m] == target){
            return m;
        }
        if(A[m] < A[r]){
            if(target > A[m] && target <= A[r]){
                l = m + 1;
            } else {
                r = m - 1;
            }
        } else {
            if(target >= A[l] && target < A[m]){
                r = m - 1;
            } else {
                l = m + 1;
            }
        }
    }
    return -1;
}
```

做题时的感悟:

1. 注意边界问题，两个条件左右边界可相等，例如target>A[m] && target<=A[r] 和target>=A[l] && target<A[m]。

2. 当第一个判断使用if(A[m] < A[r])时，结果是正确的；而当使用if(A[m] > A[l])时，结果是错误的。因为mid有可能会等于left，所以有可能会跳过第一个判断，所以如果要把left放到前面判断，把判断条件变为A[m] >= A[l]即可。

3. 如果数组变成降序该如何处理这道题呢。先判断是升序还是降序，如果这些数字都是不同的，那么采样三个数就可以得出升降序。如果三个数有序，那么很容易判断，剩余的情况是中间低两边高，或者中间高两边低，以中间高的情况为例，那么就是取两边大的那一个，如果在左边，则是递增，如果是右边，则是递减。因为中间一定是最大的数字。中间低两边高的情况类似，类推一下即可。

6. Search in Rotated Sorted Array II，和Search in Rotated Sorted Array唯一的区别是这道题目中元素会有重复的情况出现。不过正是因为这个条件的出现，出现了比较复杂的case，甚至影响到了算法的时间复杂度。原来我们是依靠中间和边缘元素的大小关系，来判断哪一半是不受rotate影响，仍然有序的。而现在因为重复的出现，如果我们遇到中间和边缘相等的情况，我们就丢失了哪边有序的信息，因为哪边都有可能是有序的结果。假设原数组是{1,2,3,3,3,3,3}，那么旋转之后有可能是{3,3,3,3,3,1,2}，或者{3,1,2,3,3,3,3}，这样的我们判断左边缘和中心的时候都是3，如果我们要寻找1或者2，我们并不知道应该跳向哪一半。解决的办法只能是对边缘移动一步，直到边缘和中间不在相等或者相遇，这就导致了会有不能切去一半的可能。所以最坏情况（比如全部都是一个元素，或者只有一个元素不同于其他元素，而他就在最后一个）就会出现每次移动一步，总共是n步，算法的时间复杂度变成O(n)。

代码如下:

```
public boolean search(int[] A, int target) {
    if(A == null || A.length == 0){
        return false;
    }
    int l = 0;
    int r = A.length - 1;
    while(l <= r){
        int m = (l + r) / 2;
```

```
            if(A[m] == target){
                return true;
            }
            if(A[m] < A[r]){
                if(A[m] < target && target <= A[r]){
                    l = m + 1;
                } else {
                    r = m - 1;
                }
            } else if(A[m] > A[r]){
                if(A[l] <= target && target < A[m]){
                    r = m - 1;
                } else {
                    l = m + 1;
                }
            } else {
                r--;
            }
        }
        return false;
    }
```

做题时的感悟:

移动的指针要与判断边界的指针相同，例如，若判断条件为A[m] < A[r] 和 A[m] > A[r]，此时我们应该移动r指针，r减减。移动l指针也是一样的，不过也要相应的把判断条件改成对l的判断。

7. Find Minimum in Rotated Sorted Array，这道题是变形的Binary Search问题。解法有两种，首先介绍我自创的保存一个min值的方法，这种可以避免跳过r指针，所以可以避免一次判断。而且这个方法可以在Rotated Sorted Array题目中通用。需要找的最小值即是要找边界，所以永远要在无序的那边找。同时要保存一个最小值，同mid来比较。

代码如下:

```
    public int findMin(int[] num) {
        if(num == null || num.length == 0){
            return -1;
        }
```

```
        int l = 0;
        int r = num.length - 1;
        int min = Integer.MAX_VALUE;
        while(l <= r){
            int m = (l + r) / 2;
            if(num[m] < min){
                min = num[m];
            }
            if(num[m] < num[r]){
                r = m - 1;
            } else {
                l = m + 1;
            }
        }
        return min;
    }
```

解法二原理与解法一是相同的，注意r指针是移动到m而不是m - 1，且终止条件为l < r.

代码如下：

```
    public int findMin(int[] num) {
        if(num == null || num.length == 0){
            return -1;
        }
        int l = 0;
        int r = num.length - 1;
        while(l < r){
            if(num[l] < num[r]){
                return num[l];
            }
            int m = (l + r) / 2;
            if(num[m] >= num[l]){
                l = m + 1;
            } else {
                r = m;
            }
        }
        return num[l];
    }
```

8. Find Minimum in Rotated Sorted Array II，与Find Minimum in Rotated Sorted Array唯一的区别就是数组里可能有重复元素，同样提供两种解法，即上一道题中两种解法稍作变形即可。

解法一代码如下：

```
public int findMin(int[] num) {
    if(num == null || num.length == 0){
        return -1;
    }
    int l = 0;
    int r = num.length - 1;
    int min = Integer.MAX_VALUE;
    while(l <= r){
        int m = (l + r) / 2;
        if(num[m] < min){
            min = num[m];
        }
        if(num[m] < num[r]){
            r = m - 1;
        } else if(num[m] > num[r]){
            l = m + 1;
        } else {
            r--;
        }
    }
    return min;
}
```

解法二代码如下：

```
public int findMin(int[] num) {
    if(num == null || num.length == 0){
        return -1;
    }
    int l = 0;
    int r = num.length - 1;
    while(l < r){
        if(num[l] < num[r]){
            return num[l];
        }
```

```
            int m = (l + r) / 2;
            if(num[m] > num[l]){
                l = m + 1;
            } else if(num[m] < num[l]){
                r = m;
            } else {
                l++;
            }
        }
        return num[l];
    }
```

9. Find Peak Element，这道题同样使用 Binary Search 来解。当选定一个 mid 时，会出现以下 3 种情况：

(1) num[m] > num[m - 1] && num[m] > num[m + 1]，说明我们已经找到波峰，直接返回 m 即可。

(2) num[m] < num[m - 1] && num[m] < num[m + 1]，说明 mid 所在位置为波谷，由于题目定义 num[-1] = num[n] = -∞，所以两边都必定存在波峰，怎么移动都可以。

(3) num[m] > num[m - 1] && num[m] < num[m + 1] 或者 num[m] < num[m - 1] && num[m] > num[m + 1] 即 mid 处于上升或者下降阶段，这时我们只要向大的方向前进就一定可以找到波峰。

代码如下：

```
    public int findPeakElement(int[] num) {
        if(num == null || num.length == 0){
            return -1;
        }
        if(num.length == 1 || num[0] > num[1]){
            return 0;
        }
        if(num[num.length - 1] > num[num.length - 2]){
            return num.length - 1;
        }
        int l = 1;
        int r = num.length - 2;
        while(l <= r){
            int m = (l + r) / 2;
```

```
            if(num[m] > num[m - 1] && num[m] > num[m + 1]){
                return m;
            } else if(num[m] < num[m - 1]){
                r = m - 1;
            } else {
                l = m + 1;
            }
        }
        return -1;
    }
```

10. Median of Two Sorted Arrays，这道题比较直接的想法就是用Merge Sorted Array这个题的方法把两个有序数组合并，当合并到第(m+n)/2个元素的时候返回那个数即可，而且不用把结果数组存起来。算法时间复杂度是O(m+n)，空间复杂度是O(1)。

接下来我们考虑有没有优化的算法。优化的思想来源于order statistics. 问题等价于求两个array的第k=(m+n)/2（假设m和n分别是两个数组的元素个数）大的数是多少。基本思路是每次通过查看两个数组的第k/2大的数(假设是A[k/2],B[k/2])，如果两个A[k/2]=B[k/2]，说明当前这个数即为两个数组剩余元素的第k大的数，如果A[k/2]>B[k/2]，那么说明B的前k/2个元素都不是我们要的第k大的数，反之则排除A的前k/2个，如此每次可以排除k/2个元素，最终k=1时即为结果。总的时间复杂度为O(logk),空间复杂度也是O(logk)，即为递归栈大小。在这个题目中因为k=(m+n)/2,所以复杂度是O(log(m+n))。

实现中还是有些细节要注意的，比如有时候剩下的数不足k/2个，那么就得剩下的，而另一个数组则需要多取一些数。但是由于这种情况发生的时候，不是把一个数组全部读完，就是可以切除k/2个数，所以不会影响算法的复杂度。

这道题的优化算法主要是由order statistics派生而来，原型应该是求topK的算法，这个问题是非常经典的问题，一般有两种解法，一种是用quick select(快速排序的subroutine)，另一种是用heap。复杂度是差不多的，topK问题在海量数据处理中也是一个非常经典的问题，所以还是要重视。

代码如下：

```
public double findMedianSortedArrays(int A[], int B[]) {
    if((A.length + B.length) % 2 == 1){
        return helper(A, 0, A.length - 1, B, 0, B.length - 1, (A.length + B.length) / 2 + 1);
    } else {
        return (helper(A, 0, A.length - 1, B, 0, B.length - 1, (A.length + B.length) / 2) +
```

```
            helper(A, 0, A.length - 1, B, 0, B.length - 1, (A.length + B.length) / 2 + 1)) / 2.0;
        }
    }

    private int helper(int[] A, int i, int i2, int[] B, int j, int j2, int k){
        int m = i2 - i + 1;
        int n = j2 - j + 1;
        if(m > n){
            return helper(B, j, j2, A, i, i2, k);
        }
        if(m == 0){
            return B[j + k - 1];
        }
        if(k == 1){
            return Math.min(A[i], B[j]);
        }
        int posA = Math.min(k / 2, m);
        int posB = k - posA;
        if(A[i + posA - 1] == B[j + posB - 1]){
            return A[i + posA - 1];
        } else if(A[i + posA - 1] < B[j + posB - 1]){
            return helper(A, i + posA, i2, B, j, j + posB - 1, k - posA);
        } else {
            return helper(A, i, i + posA - 1, B, j + posB, j2, k - posB);
        }
    }
```

这道题还有一个O(log(min(m, n)))的解法，思路如下：

Given a sorted array A of length m, we can split it into two parts:

```
{ A[0], A[1], ... , A[i - 1] } | { A[i], A[i + 1], ... , A[m - 1] }
```

All elements in right part are greater than elements in left part.

The left part has "i" elements, and right part has "m - i" elements.

There are "m + 1" kinds of splits. (i = 0 ~ m)

When i = 0, the left part has "0" elements, right part has "m" elements.

When i = m, the left part has "m" elements, right part has "0" elements.

For array B, we can split it with the same way:

```
{ B[0], B[1], ... , B[j - 1] } | { B[j], B[j + 1], ... , B[n - 1] }
```

The left part has "j" elements, and right part has "n - j" elements.

Put A's left part and B's left part into one set. (Let's name this set "LeftPart")

Put A's right part and B's right part into one set. (Let's name this set "RightPart")

```
          LeftPart          |          RightPart

{ A[0], A[1], ... , A[i - 1] } | { A[i], A[i + 1], ... , A[m - 1] }

{ B[0], B[1], ... , B[j - 1] } | { B[j], B[j + 1], ... , B[n - 1] }
```

If we can ensure:

```
1) LeftPart's length == RightPart's length (or RightPart's length + 1)

2) All elements in RightPart are greater than elements in LeftPart.
```

then we split all elements in {A, B} into two parts with equal length, and one part is always greater than the other part. Then the median can be easily found.

To ensure these two conditions, we just need to ensure:

```
(1) i + j == m - i + n - j (or: m - i + n - j + 1)

       if n >= m, we just need to set:

               i = 0 ~ m, j = (m + n + 1) / 2 - i

(2) B[j - 1] <= A[i] and A[i - 1] <= B[j]

       considering edge values, we need to ensure:

               (j == 0 or i == m or B[j - 1] <= A[i]) and

               (i == 0 or j == n or A[i - 1] <= B[j])
```

So, all we need to do is:

```
Search i from 0 to m, to find an object "i" to meet condition (1) and (2) above
```

And we can do this search by binary search. How?

If B[j0 - 1] > A[i0], than the object "ix" can't be in [0, i0]. Why?

```
Because if ix < i0, than jx = (m + n + 1) / 2 - ix > j0,

than B[jx - 1] >= B[j0 - 1] > A[i0] >= A[ix].

This violates   the condition (2). So ix can't be less than i0.
```

And if A[i0 - 1] > B[j0], than the object "ix" can't be in [i0, m].

So we can do the binary search following steps described below:

```
1. set imin, imax = 0, m, than start searching in [imin, imax]

2. i = (imin + imax) / 2; j = ((m + n + 1) / 2) - i

3. if B[j - 1] > A[i]: continue searching in [i + 1, imax]

   elif A[i - 1] > B[j]: continue searching in [imax, i]

   else: bingo! this is our object "i"
```

When the object i is found, the median is:

```
max(A[i - 1], B[j - 1]) (when m + n is odd)

or (max(A[i - 1], B[j - 1]) + min(A[i], B[j])) / 2 (when m + n is even)
```

代码如下：

```
public double findMedianSortedArrays(int A[], int B[]) {
    int m = A.length;
    int n = B.length;
    if(m > n){
        return findMedianSortedArrays(B, A);
    }
    int iMin = 0;
    int iMax = m;
```

```
        while(iMin <= iMax){
            int i = (iMin + iMax) >> 1;
            int j = ((m + n + 1) >> 1) - i;
            if(j > 0 && i < m && B[j - 1] > A[i]){
                iMin = i + 1;
            } else if(i > 0 && j < n && A[i - 1] > B[j]){
                iMax = i - 1;
            } else {
                int num1 = 0;
                int num2 = 0;
                if(i == 0){
                    num1 = B[j - 1];
                } else if(j == 0){
                    num1 = A[i - 1];
                } else {
                    num1 = Math.max(A[i - 1], B[j - 1]);
                }
                if(((m + n) & 1) == 1){
                    return num1;
                }
                if(i == m){
                    num2 = B[j];
                } else if(j == n){
                    num2 = A[i];
                } else {
                    num2 = Math.min(A[i], B[j]);
                }
                return (num1 + num2) / 2.0;
            }
        }
        return -1.0;
    }
```

总体来说，二分查找算法理解起来并不算难，但在实际面试的过程中可能会出现各种变体，如何灵活的运用才是制胜的关键。我们要抓住“有序”的特点，一旦发现输入有“有序”的特点，我们就可以考虑是否可以运用二分查找算法来解决该问题。