二分查找-九章模板篇

二分查找是非常重要的基础算法，一定要牢牢掌握。面试在热身阶段或者面试官在查看你的简历的同时，很可能会让你写一道二分查找的题目。下面依然提供一个模板，由于该模板发挥稳定，所以尽量使用它，但不能死记硬背，也需要根据不同的题目做相应的变化。

在 LeetCode 用到二分查找并且可以使用模板的主要题目有：

1. Search Insert Position
2. Search for a Range
3. Search in Rotated Sorted Array
4. Search in Rotated Sorted Array II
5. Find Minimum in Rotated Sorted Array
6. Find Minimum in Rotated Sorted Array II
7. Search a 2D Matrix
8. Find Peak Element

二分查找题目[黄金]模板：

```
public int binarySearch(int[] nums, int target) {
    if(nums == null || nums.length == 0){
        return -1;
    }
    int left = 0;
    int right = nums.length - 1;
    int mid;
    while(left + 1 < right){ //  ①
        mid = left + (right - left) / 2; //  ②
        if(nums[mid] == target){ //  ③
            right = mid;
        } else if(nums[mid] < target){
            left = mid;
        } else {
            right = mid;
        }
    }
```

```
        if(nums[left] == target){ //  ④
            return left;
        }
        if(nums[right]==target){
            return right;
        }
        return -1;
    }
```

需要思考和更改的部分：

① 终止判断条件写为left + 1 < right可以保证left指针和right指针在结束时相邻或者相交。

② 这种取mid的方式可以保证当left和right都接近Integer.MAX_VALUE时也不会越界。

③ 此处需要对A[mid] == target, A[mid] > target, A[mid] < target三种情况进行处理。

④ 最后需要根据题意，对相邻或相交的两个指针做对应的判断。

Warning：由于模板稳定性好，此处所有题目一律使用模板来解，其他方法只为备用方案，参见二分查找篇。但有时其他的方法可能会更容易理解和使用，而且代码不会这样冗长，所以需要根据情况和个人喜好进行选择。

1. Search Insert Position，这道题可以有两种解法。

方法一：利用模板的稳定解法。不再赘述，套模板即可。

方法二：利用 l <= r 性质的巧妙解法。终止判断设为 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：

```
    // Solution 1 – Classical Model
    public int searchInsert(int[] A, int target) {
        if(A == null || A.length == 0){
            return 0;
        }
        int left = 0;
        int right = A.length - 1;
        int mid;
        while(left + 1 < right){
```

```
            mid = left + (right - left) / 2;
            if(A[mid] == target){
                return mid;
            }
            if(A[mid] < target){
                left = mid;
            } else {
                right = mid;
            }
        }
        if(target <= A[left]){
            return left;
        }
        if(target <= A[right]){
            return right;
        }
        return right + 1;
    }

    // Solution 2 – End with l <= r
    public int searchInsert(int[] A, int target) {
        if(A == null || A.length == 0){
            return 0;
        }
        int left = 0;
        int right = A.length - 1;
        int mid;
        while(left <= right){
            mid = left + (right - left) / 2;
            if(A[mid] == target){
                return mid;
            }
            if(A[mid] < target){
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return left;
    }
```

2. Search for a Range，这道题可以有两种解法。

方法一：利用模板的稳定解法。不再赘述，套模板即可。

方法二：利用l <= r性质的巧妙解法。如果相等的时候也向右夹逼，最后r指针就会停在右边界，而l指针就会停在右边界+1处（可能越界，但不影响结果）；而向左夹逼则l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。

```
// Solution 1 – Classical Model
public int[] searchRange(int[] A, int target) {
    int[] res = {-1, -1};
    if(A == null || A.length == 0){
        return res;
    }
    int left = 0;
    int right = A.length - 1;
    int mid;
    while(left + 1 < right){
        mid = left + (right - left) / 2;
        if(A[mid] >= target){
            right = mid;
        } else {
            left = mid;
        }
    }
    if(A[right] == target){
        res[0] = right;
    }
    if(A[left] == target){
        res[0] = left;
    }
    left = 0;
    right = A.length - 1;
    while(left + 1 < right){
        mid = left + (right - left) / 2;
        if(A[mid] <= target){
            left = mid;
        } else {
            right = mid;
        }
    }
```

```
        if(A[left] == target){
            res[1] = left;
        }
        if(A[right] == target){
            res[1] = right;
        }
        return res;
    }

    // Solution 2 – End with l <= r
    public int[] searchRange(int[] A, int target) {
        int[] res = {-1, -1};
        if(A == null || A.length == 0){
            return res;
        }
        int left = 0;
        int right = A.length - 1;
        int mid;
        while(left <= right){
            mid = left + (right - left) / 2;
            if(A[mid] >= target){
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
        int leftBound = left;
        left = 0;
        right = A.length - 1;
        while(left <= right){
            mid = left + (right - left) / 2;
            if(A[mid] <= target){
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        int rightBound = right;
        if(leftBound <= rightBound){
            res[0] = leftBound;
```

```
            res[1] = rightBound;
        }
        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. Search in Rotated Sorted Array，同样两种方法都可以解决。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)。

```
    // Solution 1 – Classical Model
    public int search(int[] A, int target) {
        if(A == null || A.length == 0){
            return -1;
        }
        int left = 0;
        int right = A.length - 1;
        int mid;
        while(left + 1 < right){
            mid = left + (right - left) / 2;
            if(A[mid] == target){
                return mid;
```

```
            }
            if(A[mid] < A[right]){
                if(A[mid] < target && target <= A[right]){
                    left = mid;
                } else {
                    right = mid;
                }
            } else {
                if(A[left] <= target && target < A[mid]){
                    right = mid;
                } else {
                    left = mid;
                }
            }
        }
        if(A[left] == target){
            return left;
        }
        if(A[right] == target){
            return right;
        }
        return -1;
}

// Solution 2 – End with l <= r
public int search(int[] A, int target) {
        if(A == null || A.length == 0){
            return -1;
        }
        int left = 0;
        int right = A.length - 1;
        int mid;
        while(left <= right){
            mid = left + (right - left) / 2;
            if(A[mid] == target){
                return mid;
            }
            if(A[mid] < A[right]){
                if(A[mid] < target && target <= A[right]){
                    left = mid + 1;
```

```
                } else {
                    right = mid - 1;
                }
            } else {
                if(A[left] <= target && target < A[mid]){
                    right = mid - 1;
                } else {
                    left = mid + 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。

4. 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)。

```
// Solution 1 – Classical Model
public boolean search(int[] A, int target) {
    if(A == null || A.length == 0){
        return false;
    }
    int left = 0;
    int right = A.length - 1;
    while(left + 1 < right){
        int mid = left + (right - left) / 2;
        if(A[mid] == target){
            return true;
        }
        if(A[mid] < A[right]){
            if(A[mid] < target && target <= A[right]){
                left = mid;
            } else {
                right = mid;
            }
        } else if(A[mid] > A[right]){
            if(A[left] <= target && target < A[mid]){
                right = mid;
            } else {
                left = mid;
            }
        } else {
            right--;
        }
    }
    if(A[left] == target){
        return true;
    }
    if(A[right] == target){
        return true;
    }
    return false;
}
```

```
// Solution 2 – End with l <= r
public boolean search(int[] A, int target) {
    if(A == null || A.length == 0){
        return false;
    }
    int left = 0;
    int right = A.length - 1;
    int mid;
    while(left <= right){
        mid = left + (right - left) / 2;
        if(A[mid] == target){
            return true;
        }
        if(A[mid] < A[right]){
            if(A[mid] < target && target <= A[right]){
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        } else if(A[mid] > A[right]){
            if(A[left] <= target && target < A[mid]){
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        } else {
            right--;
        }
    }
    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。

5. Find Minimum in Rotated Sorted Array，这道题是变形的Binary Search问题，解法依然有两种。第一种自然是模板解法。然后介绍我自创的保存一个min值的方法，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：

```
// Solution 1 – Classical Model
public int findMin(int[] num) {
    if(num == null || num.length == 0){
        return -1;
    }
    int left = 0;
    int right = num.length - 1;
    int mid;
    while(left + 1 < right){
        mid = left + (right - left) / 2;
        if(num[mid] < num[right]){
            right = mid;
        } else {
            left = mid;
        }
    }
    return num[left] < num[right] ? num[left] : num[right];
}

// Solution 2 – End with l <= r
public int findMin(int[] num) {
    if(num == null || num.length == 0){
        return -1;
    }
    int left = 0;
    int right = num.length - 1;
    int mid;
    int min = Integer.MAX_VALUE;
    while(left <= right){
        mid = left + (right - left) / 2;
        if(num[mid] < min){
            min = num[mid];
        }
```

```
            if(num[mid] < num[right]){
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
        return min;
    }
```

6. 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。

```
    // Solution 1 – Classical Model
    public int findMin(int[] num) {
        if(num == null || num.length == 0){
            return -1;
        }
        int left = 0;
        int right = num.length - 1;
        int mid;
        while(left + 1 < right){
            mid = left + (right - left) / 2;
            if(num[mid] < num[right]){
                right = mid;
            } else if(num[mid] > num[right]){
                left = mid;
            } else {
                right--;
            }
        }
        return num[left] < num[right] ? num[left] : num[right];
    }

    // Solution 2 – End with l <= r
    public int findMin(int[] num) {
        if(num == null || num.length == 0){
            return -1;
        }
        int left = 0;
        int right = num.length - 1;
```

```
        int mid;
        int min = Integer.MAX_VALUE;
        while(left <= right){
            mid = left + (right - left) / 2;
            if(num[mid] < min){
                min = num[mid];
            }
            if(num[mid] < num[right]){
                right = mid - 1;
            } else if(num[mid] > num[right]){
                left = mid + 1;
            } else {
                right--;
            }
        }
        return min;
    }
```

7. 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 || target < matrix[0][0]){
            return false;
        }
        int left = 0;
        int right = matrix.length - 1;
        int mid;
        while(left + 1 < right){
            mid = left + (right - left) / 2;
            if(matrix[mid][0] == target){
                return true;
            }
            if(matrix[mid][0] < target){
                left = mid;
            } else {
                right = mid;
            }
        }
        int row = 0;
        if(matrix[right][0] > target){
            row = right - 1;
        } else {
            row = right;
        }
```

```
        left = 0;
        right = matrix[0].length - 1;
        while(left + 1 < right){
            mid = left + (right - left) / 2;
            if(matrix[row][mid] == target){
                return true;
            }
            if(matrix[row][mid] < target){
                left = mid;
            } else {
                right = mid;
            }
        }
        if(matrix[row][left] == target || matrix[row][right] == target){
            return true;
        }
        return false;
    }

    public boolean searchMatrix(int[][] matrix, int target) {
        if(matrix == null || matrix.length == 0 || matrix[0].length == 0){
            return false;
        }
        int left = 0;
        int right = matrix.length - 1;
        while(left <= right){
            int mid = left + (right - left) / 2;
            if(matrix[mid][0] == target){
                return true;
            }
            if(matrix[mid][0] < target){
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        int row = right;
        if(row < 0){
            return false;
        }
```

```
            left = 0;
            right = matrix[0].length - 1;
            while(left <= right){
                mid = left + (right - left) / 2;
                if(matrix[row][mid] == target){
                    return true;
                }
                if(matrix[row][mid] < target){
                    left = mid + 1;
                } else {
                    right = mid - 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。

8. Find Peak Element，同样使用二分查找来解。当选定一个 mid 时，会有以下 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。

```
// Solution 1 – Classical Model
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 left = 0;
    int right = num.length - 1;
    int mid;
    while(left + 1 < right){
        mid = left + (right - left) / 2;
        if(num[mid - 1] < num[mid] && num[mid] > num[mid + 1]){
            return mid;
        }
        if(num[mid - 1] < num[mid]){
            left = mid;
        } else {
            right = mid;
        }
    }
    if(num[left - 1] < num[left] && num[left] > num[left + 1]){
        return left;
    }
    if(num[right - 1] < num[right] && num[right] > num[right + 1]){
        return right;
    }
    return -1;
}

// Solution 2 – End with l <= r
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 left = 1;
        int right = num.length - 2;
        int mid;
        while(left <= right){
            mid = left + (right - left) / 2;
            if(num[mid] > num[mid - 1] && num[mid] > num[mid + 1]){
                return mid;
            }
            if(num[mid] > num[mid - 1]){
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return -1;
    }
```

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。