

第七课

# 15分钟搞懂二分搜索与贪婪

## 二分搜索算法

- 看似简单，写对很难
- 变形很多
- 在面试中常用来考察code能力

## 贪婪算法

- 是一种比较直观的算法
- 难以证明它的正确性



## 定义

- ▶ 二分搜索也称折半搜索，是一种在有序数组中查找某一特定元素的搜索算法

## 运用前提

- ▶ 数组必须是排好序的
- ▶ 输入并不一定是数组，也可能是给定一个区间的起始和终止的位置

## 优点

- ▶ 二分搜索也称对数搜索，其时间复杂度为  $O(\lg n)$ ，是一种非常高效的搜索

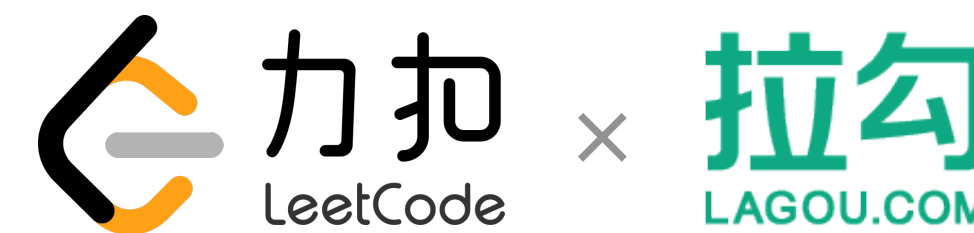
## 缺点

- ▶ 要求待查找的数组或区间是排好序的
- 若要求对数组进行动态地删除和插入操作并完成查找，平均复杂度会变为  $O(n)$
- 采取自平衡的二叉查找树
  - 可在  $O(n \log n)$  的时间内用给定的数据构建出一棵二叉查找树
  - 可在  $O(\log n)$  的时间内对数据进行搜索
  - 可在  $O(\log n)$  的时间内完成删除和插入的操作

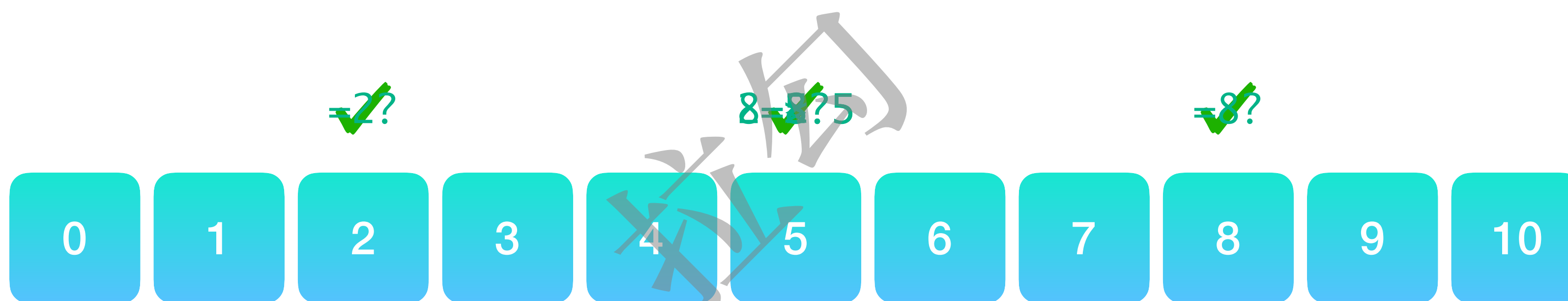
当：输入的数组或区间是有序的，且不会常变动，要求从中找出一个满足条件的元素  $\longrightarrow$  采用二分搜索 🍊

7.1

## 二分搜索 / Binary Search



### 解题思路



## 基本解题模板

- ▶ 递归
  - 优点是简洁
  - 缺点是执行消耗大
- ▶ 非递归



```
int binarySearch(int[] nums, int target, int low, int high) {  
    if (low > high) {  
        return -1;  
    }  
  
    int middle = low + (high - low) / 2;  
  
    if (nums[middle] == target) {  
        return middle;  
    }  
  
    if (target < nums[middle]) {  
        return binarySearch(nums, target, low, middle - 1);  
    } else {  
        return binarySearch(nums, target, middle + 1, high);  
    }  
}
```

## 递归的写法

- 二分搜索函数的定义中，不仅要指定数组 `nums` 和目标查找数 `target`，还要指定查找区间的起点 `low` 和终点位置 `high`。

```
int binarySearch(int[] nums, int target, int low, int high) {  
    if (low > high) {  
        return -1;  
    }  
  
    int middle = low + (high - low) / 2;  
  
    if (nums[middle] == target) {  
        return middle;  
    }  
  
    if (target < nums[middle]) {  
        return binarySearch(nums, target, low, middle - 1);  
    } else {  
        return binarySearch(nums, target, middle + 1, high);  
    }  
}
```

## 递归的写法

- 二分搜索函数的定义中，不仅要指定数组 `nums` 和目标查找数 `target`，还要指定查找区间的起点 `low` 和终点位置 `high`。
- 为避免无限循环，开始时要判断一下：如果起点位置大于终点位置，表明这是一个非法区间；或者说，已尝试了所有的搜索区间还是没找到结果。返回 `-1`。



```
int binarySearch(int[] nums, int target, int low, int high) {  
    if (low > high) {  
        return -1;  
    }  
  
    int middle = low + (high - low) / 2;  
  
    if (nums[middle] == target) {  
        return middle;  
    }  
  
    if (target < nums[middle]) {  
        return binarySearch(nums, target, low, middle - 1);  
    } else {  
        return binarySearch(nums, target, middle + 1, high);  
    }  
}
```

## 递归的写法

- 二分搜索函数的定义中，不仅要指定数组 `nums` 和目标查找数 `target`，还要指定查找区间的起点 `low` 和终点位置 `high`。
- 为避免无限循环，开始时要判断一下：如果起点位置大于终点位置，表明这是一个非法区间；或者说，已尝试了所有的搜索区间还是没找到结果。返回 `-1`。
- 接下来，取正中间那个数的下标 `middle`。

```
int binarySearch(int[] nums, int target, int low, int high) {  
    if (low > high) {  
        return -1;  
    }
```

```
    int middle = low + (high - low) / 2;
```

```
    if (nums[middle] == target) {  
        return middle;  
    }
```

```
    if (target < nums[middle]) {  
        return binarySearch(nums, target, low, middle - 1);  
    } else {  
        return binarySearch(nums, target, middle + 1, high);  
    }  
}
```

## 递归的写法

- 二分搜索函数的定义中，不仅要指定数组 `nums` 和目标查找数 `target`，还要指定查找区间的起点 `low` 和终点位置 `high`。
- 为避免无限循环，开始时要判断一下：如果起点位置大于终点位置，表明这是一个非法区间；或者说，已尝试了所有的搜索区间还是没找到结果。返回 `-1`。
- 接下来，取正中间那个数的下标 `middle`。
- 判断一下正中间的那个数是不是要找的目标数 `target`。如果是，就返回下标 `middle`。

```
int binarySearch(int[] nums, int target, int low, int high) {  
    if (low > high) {  
        return -1;  
    }  
  
    int middle = low + (high - low) / 2;  
  
    if (nums[middle] == target) {  
        return middle;  
    }  
  
    if (target < nums[middle]) {  
        return binarySearch(nums, target, low, middle - 1);  
    } else {  
        return binarySearch(nums, target, middle + 1, high);  
    }  
}
```

## 递归的写法

- 二分搜索函数的定义中，不仅要指定数组 `nums` 和目标查找数 `target`，还要指定查找区间的起点 `low` 和终点位置 `high`。
- 为避免无限循环，开始时要判断一下：如果起点位置大于终点位置，表明这是一个非法区间；或者说，已尝试了所有的搜索区间还是没找到结果。返回 `-1`。
- 接下来，取正中间那个数的下标 `middle`。
- 判断一下正中间的那个数是不是要找的目标数 `target`。如果是，就返回下标 `middle`。
- 如果发现目标数在左边，那么就递归地从左半边进行二分搜索。

```
int binarySearch(int[] nums, int target, int low, int high) {  
    if (low > high) {  
        return -1;  
    }  
  
    int middle = low + (high - low) / 2;  
  
    if (nums[middle] == target) {  
        return middle;  
    }  
  
    if (target < nums[middle]) {  
        return binarySearch(nums, target, low, middle - 1);  
    } else {  
        return binarySearch(nums, target, middle + 1, high);  
    }  
}
```

## 递归的写法

- 二分搜索函数的定义中，不仅要指定数组 `nums` 和目标查找数 `target`，还要指定查找区间的起点 `low` 和终点位置 `high`。
- 为避免无限循环，开始时要判断一下：如果起点位置大于终点位置，表明这是一个非法区间；或者说，已尝试了所有的搜索区间还是没找到结果。返回 `-1`。
- 接下来，取正中间那个数的下标 `middle`。
- 判断一下正中间的那个数是不是要找的目标数 `target`。如果是，就返回下标 `middle`。
- 如果发现目标数在左边，那么就递归地从左半边进行二分搜索。
- 否则从右半边递归地进行二分搜索。

```
int binarySearch(int[] nums, int target, int low, int high) {  
    if (low > high) {  
        return -1;  
    }  
  
    int middle = low + (high - low) / 2;  
  
    if (nums[middle] == target) {  
        return middle;  
    }  
  
    if (target < nums[middle]) {  
        return binarySearch(nums, target, low, middle - 1);  
    } else {  
        return binarySearch(nums, target, middle + 1, high);  
    }  
}
```

### 三个关键点

- 计算 `middle` 下标时，不能简单地用  $(low + high) / 2$ ，这样可能会导致溢出。
- 取左半边和右半边的区间时，左半边是  $[low, middle - 1]$ ，右半边是  $[middle + 1, high]$ ，这是两个闭区间。我们确定了 `middle` 点不是我们要找的，因此没有必要再把它加入到左、右半边了。
- 对于一个长度为奇数的数组，例如：{1, 2, 3, 4, 5}，按照  $low + (high - low) / 2$  来计算的话，`middle` 就是正中间的那个位，对于一个长度为偶数的数组，例如：{1, 2, 3, 4}，`middle` 就是正中间靠左边的一个位置。



```
int binarySearch(int[] nums, int target, int low, int high) {  
    if (low > high) {  
        return -1;  
    }  
  
    int middle = low + (high - low) / 2;  
  
    if (nums[middle] == target) {  
        return middle;  
    }  
  
    if (target < nums[middle]) {  
        return binarySearch(nums, target, low, middle - 1);  
    } else {  
        return binarySearch(nums, target, middle + 1, high);  
    }  
}
```

## ▸ 时间复杂度分析

假设要对长度为  $n$  的数组进行二分搜索，

$T(n)$  是执行时间函数，我们可以得到：

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

代入公式法得：  $a = 1, b = 2, f(n) = 1$

因此  $O(n^{\log_b a}) = O(n^0) = 1$  等于  $O(f(n))$

时间复杂度为：

$$O(n^{\log_b a} \log n) = O(\log n)$$

```
int binarySearch(int[] nums, int target, int low, int high) {  
    while (low <= high) {  
        int middle = low + (high - low) / 2;  
  
        if (nums[middle] == target) {  
            return middle;  
        }  
  
        if (target < nums[middle]) {  
            high = middle - 1;  
        } else {  
            low = middle + 1;  
        }  
    }  
  
    return -1;  
}
```

## 非递归的写法

- 在 while 循环中，判断一下搜索的区间是否有效。

```
int binarySearch(int[] nums, int target, int low, int high) {  
    while (low <= high) {  
        int middle = low + (high - low) / 2;  
  
        if (nums[middle] == target) {  
            return middle;  
        }  
  
        if (target < nums[middle]) {  
            high = middle - 1;  
        } else {  
            low = middle + 1;  
        }  
    }  
  
    return -1;  
}
```

## 非递归的写法

- 在 while 循环中，判断一下搜索的区间是否有效。
- 计算正中间数的下标。



```
int binarySearch(int[] nums, int target, int low, int high) {  
    while (low <= high) {  
        int middle = low + (high - low) / 2;
```

```
        if (nums[middle] == target) {  
            return middle;  
        }
```

```
        if (target < nums[middle]) {  
            high = middle - 1;  
        } else {  
            low = middle + 1;  
        }  
    }
```

```
    return -1;  
}
```

## 非递归的写法

- 在 while 循环中，判断一下搜索的区间是否有效。
- 计算正中间数的下标。
- 判断一下正中间的那个数是不是要找的目标数 target。如果是，就返回下标 middle。

```
int binarySearch(int[] nums, int target, int low, int high) {  
    while (low <= high) {  
        int middle = low + (high - low) / 2;  
  
        if (nums[middle] == target) {  
            return middle;  
        }  
  
        if (target < nums[middle]) {  
            high = middle - 1;  
        } else {  
            low = middle + 1;  
        }  
    }  
  
    return -1;  
}
```

## 非递归的写法

- 在 while 循环中，判断一下搜索的区间是否有效。
- 计算正中间数的下标。
- 判断一下正中间的那个数是不是要找的目标数 target。如果是，就返回下标 middle。
- 如果发现目标数在左边，调整搜索区间的终点为 middle - 1。

```
int binarySearch(int[] nums, int target, int low, int high) {  
    while (low <= high) {  
        int middle = low + (high - low) / 2;  
  
        if (nums[middle] == target) {  
            return middle;  
        }  
  
        if (target < nums[middle]) {  
            high = middle - 1;  
        } else {  
            low = middle + 1;  
        }  
    }  
  
    return -1;  
}
```

## 非递归的写法

- 在 while 循环中，判断一下搜索的区间是否有效。
- 计算正中间数的下标。
- 判断一下正中间的那个数是不是要找的目标数 target。如果是，就返回下标 middle。
- 如果发现目标数在左边，调整搜索区间的终点为 middle - 1。
- 否则，调整搜索区间的终点为 middle + 1。

```
int binarySearch(int[] nums, int target, int low, int high) {  
    while (low <= high) {  
        int middle = low + (high - low) / 2;  
  
        if (nums[middle] == target) {  
            return middle;  
        }  
  
        if (target < nums[middle]) {  
            high = middle - 1;  
        } else {  
            low = middle + 1;  
        }  
    }  
  
    return -1;  
}
```

## 非递归的写法

- 在 while 循环中，判断一下搜索的区间是否有效。
- 计算正中间数的下标。
- 判断一下正中间的那个数是不是要找的目标数 target。如果是，就返回下标 middle。
- 如果发现目标数在左边，调整搜索区间的终点为 middle - 1。
- 否则，调整搜索区间的终点为 middle + 1。
- 如果超出了搜索区间，表明无法找到目标数，返回 -1。

## 二分搜索的核心

- ▶ 确定搜索的范围和区间
- ▶ 取中间的数判断是否满足条件
- ▶ 如果不满足条件，判定应该往哪个半边继续进行搜索

## 找确定的边界

- ▶ 边界分为上边界与下边界，有时也称为左边距和右边界
- ▶ 确定的边界，指边界的数值等于要找的目标数

## 34. 最长子序列的长度

给定一个按照升序排列的整数数组 `nums`，  
和一个目标值 `target`。

找出给定目标值在数组中的开始位置和结束位置。

### 说明

你的算法时间复杂度必须是  $O(\log n)$  级别。

如果数组中不存在目标值，返回 `[-1, -1]`。



- 第一次出现的地方叫下边界 (lower bound)
- 最后一次出现的地方叫上边界 (upper bound)

## 两个成为 8 的下边界的条件

- ▶ 该数必须是 8
- ▶ 该数的左边一个数必须不是 8
  - 8 的左边有数，那么该数必须小于 8
  - 8 的左边没有数，即 8 是数组的第一个数

## 两个成为 8 的上边界的条件

- ▶ 该数必须是 8
- ▶ 该数的右边一个数必须不是 8
  - 8 的右边有数，那么该数必须大于 8
  - 8 的右边没有数，即 8 是数组的最后一个数



```
int searchLowerBound(int[] nums, int target, int low, int high)
{
    if (low > high) {
        return -1;
    }

    int middle = low + (high - low) / 2;

    if (nums[middle] == target && (middle == 0 || nums[middle - 1] < target)) {
        return middle;
    }

    if (target <= nums[middle]) {
        return searchLowerBound(nums, target, low, middle - 1);
    } else {
        return searchLowerBound(nums, target, middle + 1, high);
    }
}
```

### 递归 - 寻找下边界

- 判断是否是下边界时，先看看 middle 的数是否为 target，并判断该数是否已为数组的第一个数，或者，它左边的一个数是不是已经比它小，如果都满足，即为下边界。

```
int searchLowerBound(int[] nums, int target, int low, int high)
{
    if (low > high) {
        return -1;
    }

    int middle = low + (high - low) / 2;

    if (nums[middle] == target && (middle == 0 || nums[middle - 1] < target)) {
        return middle;
    }

    if (target <= nums[middle]) {
        return searchLowerBound(nums, target, low, middle - 1);
    } else {
        return searchLowerBound(nums, target, middle + 1, high);
    }
}
```

### 递归 - 寻找下边界

- 判断是否是下边界时，先看看 middle 的数是否为 target，并判断该数是否已为数组的第一个数，或者，它左边的一个数是不是已经比它小，如果都满足，即为下边界。
- 不满足时，如果该数等于 target，需向左继续查找

```
int searchUpperBound(int[] nums, int target, int low, int high)
{
    if (low > high) {
        return -1;
    }

    int middle = low + (high - low) / 2;

    if (nums[middle] == target && (middle == nums.length - 1 ||
    nums[middle + 1] > target)) {
        return middle;
    }

    if (target < nums[middle]) {
        return searchUpperBound(nums, target, low, middle - 1);
    } else {
        return searchUpperBound(nums, target, middle + 1, high);
    }
}
```

### 递归 - 寻找上边界

- 判断是否是上边界时，  
先看看 middle 的数是否为 target，  
并判断该数是否已为数组的最后一个数，  
或者，它右边的数是不是比它大，  
如果都满足，即为上边界。

```
int searchUpperBound(int[] nums, int target, int low, int high)
{
    if (low > high) {
        return -1;
    }
```

```
    int middle = low + (high - low) / 2;
```

```
    if (nums[middle] == target && (middle == nums.length - 1 ||
    nums[middle + 1] > target)) {
        return middle;
    }
```

```
    if (target < nums[middle]) {
        return searchUpperBound(nums, target, low, middle - 1);
    } else {
        return searchUpperBound(nums, target, middle + 1, high);
    }
}
```

### 递归 - 寻找上边界

- 判断是否是上边界时，  
先看看 middle 的数是否为 target，  
并判断该数是否已为数组的最后一个数，  
或者，它右边的数是不是比它大，  
如果都满足，即为上边界。
- 不满足时，需判断搜索方向

## 找模糊的边界

- ▶ 二分搜索可以用来查找一些模糊的边界
- ▶ 模糊的边界，即边界的值不等于目标的值，而是大于或小于目标的值

## 例题分析

从数组  $\{-2, 0, 1, 4, 7, 9, 10\}$  中找到第一个大于6的数。第一个大于6的数，在这道题里面，答案是7。



如何在有序数组中判断一个数是不是第一个大于 6 的数？

- ▶ 这个数要大于 6
- ▶ 这个数有可能是数组里的第一个数，或者它之前的一个数比 6 小

```
Integer firstGreaterThan(int[] nums, int target, int low, int high) {  
    if (low > high) {  
        return null;  
    }
```

```
    int middle = low + (high - low) / 2;
```

```
    if (nums[middle] > target && (middle == 0 || nums[middle - 1] <= target)) {  
        return middle;  
    }
```

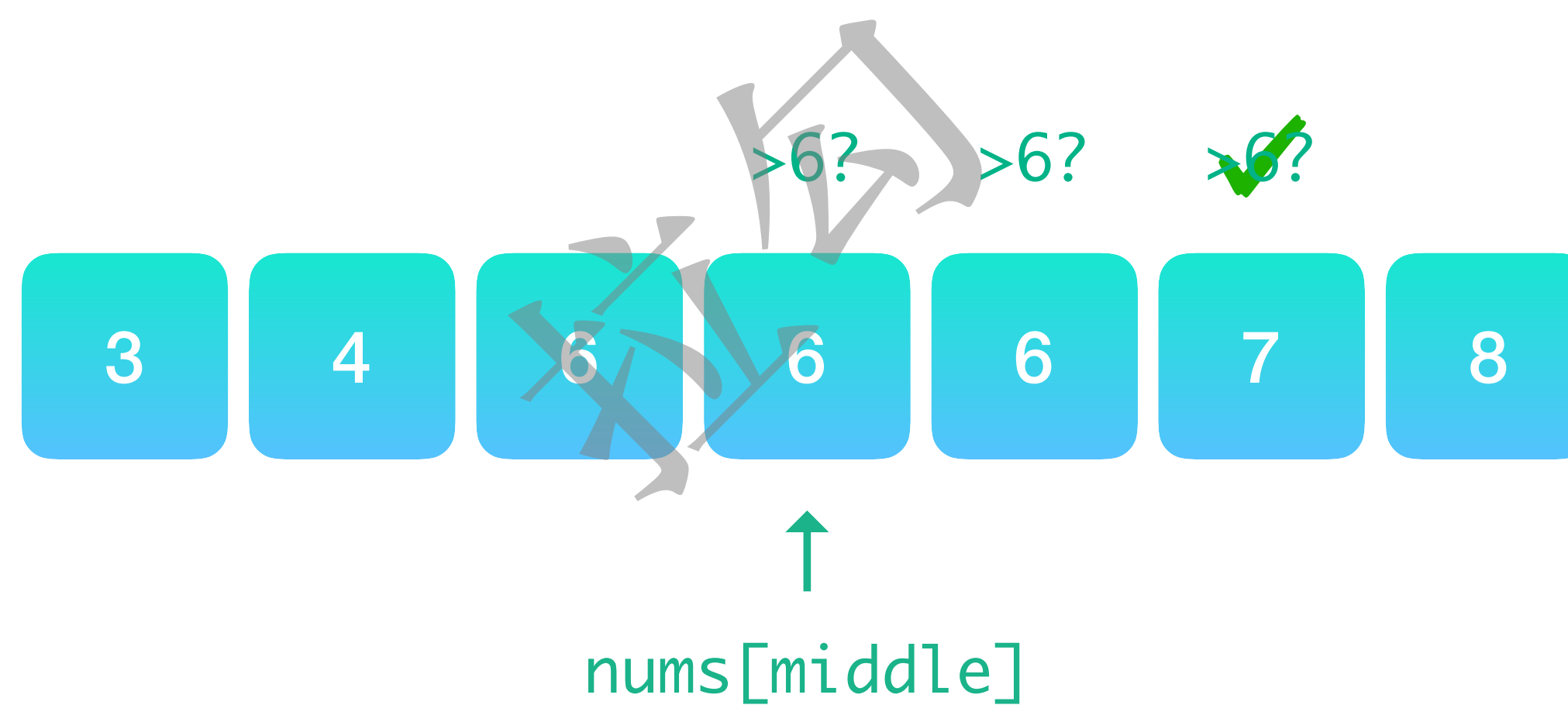
```
    if (target < nums[middle]) {  
        return firstGreaterThan(nums, target, low, middle - 1);  
    } else {  
        return firstGreaterThan(nums, target, middle + 1, high);  
    }  
}
```

判断 middle 指向的数

是否为第一个比 target 大的数时，  
须同时满足两个条件：

- middle 这个数必须大于 target
- middle 要么是第一个数，  
要么它之前的数小于或者等于 target

当不满足条件，而 middle 的数等于 target 时怎么办？





7.1

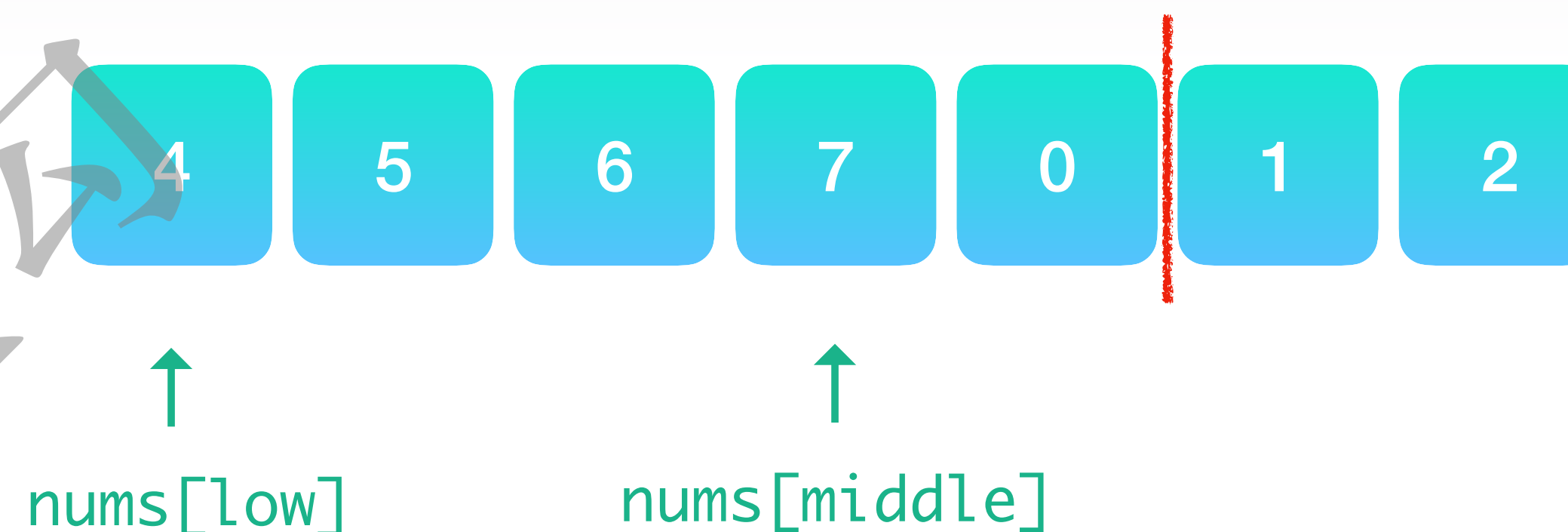
## 二分搜索 / Binary Search



## 33. 旋转过的排序数组

给定一个经过旋转了的排序数组，判断一下某个数是否在里面。

例如，给定数组为{4, 5, 6, 7, 0, 1, 2}，target等于0，答案是4，即0所在的位置下标是4。



- ▶ 如何判断左边是不是排好序的那个部分呢？
- ▶ 只要比较 $\text{nums}[\text{low}]$ 和 $\text{nums}[\text{middle}]$ 即可
  - 若 $\text{nums}[\text{low}] \leq \text{nums}[\text{middle}]$ ，则左边这部分一定是排好序的，否则右边是排好序的
- ▶ 判定出某一边是排好序的有什么用呢？
  - 若 $\text{nums}[\text{low}] \leq \text{target} \ \&\& \ \text{target} < \text{nums}[\text{middle}]$ ，则目标值在这个区间，反之在另一边

```
int binarySearch(int[] nums, int target, int low, int high) {  
    if (low > high) {  
        return -1;  
    }
```

- 判断是否已超出了搜索范围，是则返回-1

```
int middle = low + (high - low) / 2;  
if (nums[middle] == target) {  
    return middle;  
}  
  
if (nums[low] <= nums[middle]) {  
    if (nums[low] <= target && target < nums[middle]) {  
        return binarySearch(nums, target, low, middle - 1);  
    }  
    return binarySearch(nums, target, middle + 1, high);  
} else {  
    if (nums[middle] < target && target <= nums[high]) {  
        return binarySearch(nums, target, middle + 1, high);  
    }  
    return binarySearch(nums, target, low, middle - 1);  
}  
}
```

```
int binarySearch(int[] nums, int target, int low, int high) {  
    if (low > high) {  
        return -1;  
    }  
  
    int middle = low + (high - low) / 2;  
    if (nums[middle] == target) {  
        return middle;  
    }  
  
    if (nums[low] <= nums[middle]) {  
        if (nums[low] <= target && target < nums[middle]) {  
            return binarySearch(nums, target, low, middle - 1);  
        }  
        return binarySearch(nums, target, middle + 1, high);  
    } else {  
        if (nums[middle] < target && target <= nums[high]) {  
            return binarySearch(nums, target, middle + 1, high);  
        }  
        return binarySearch(nums, target, low, middle - 1);  
    }  
}
```

- 判断是否已超出了搜索范围，是则返回-1
- 取中位数

```
int binarySearch(int[] nums, int target, int low, int high) {  
    if (low > high) {  
        return -1;  
    }  
  
    int middle = low + (high - low) / 2;  
    if (nums[middle] == target) {  
        return middle;  
    }  
  
    if (nums[low] <= nums[middle]) {  
        if (nums[low] <= target && target < nums[middle]) {  
            return binarySearch(nums, target, low, middle - 1);  
        }  
        return binarySearch(nums, target, middle + 1, high);  
    } else {  
        if (nums[middle] < target && target <= nums[high]) {  
            return binarySearch(nums, target, middle + 1, high);  
        }  
        return binarySearch(nums, target, low, middle - 1);  
    }  
}
```

- 判断是否已超出了搜索范围，是则返回-1
- 取中位数
- 判断中位数是否为要找的数

```
int binarySearch(int[] nums, int target, int low, int high) {  
    if (low > high) {  
        return -1;  
    }  
  
    int middle = low + (high - low) / 2;  
    if (nums[middle] == target) {  
        return middle;  
    }  
  
    if (nums[low] <= nums[middle]) {  
        if (nums[low] <= target && target < nums[middle]) {  
            return binarySearch(nums, target, low, middle - 1);  
        }  
        return binarySearch(nums, target, middle + 1, high);  
    } else {  
        if (nums[middle] < target && target <= nums[high]) {  
            return binarySearch(nums, target, middle + 1, high);  
        }  
        return binarySearch(nums, target, low, middle - 1);  
    }  
}
```

- 判断是否已超出了搜索范围，是则返回-1
- 取中位数
- 判断中位数是否为要找的数
- 判断左半边是不是排好序的

```
int binarySearch(int[] nums, int target, int low, int high) {  
    if (low > high) {  
        return -1;  
    }  
  
    int middle = low + (high - low) / 2;  
    if (nums[middle] == target) {  
        return middle;  
    }  
  
    if (nums[low] <= nums[middle]) {  
        if (nums[low] <= target && target < nums[middle]) {  
            return binarySearch(nums, target, low, middle - 1);  
        }  
        return binarySearch(nums, target, middle + 1, high);  
    } else {  
        if (nums[middle] < target && target <= nums[high]) {  
            return binarySearch(nums, target, middle + 1, high);  
        }  
        return binarySearch(nums, target, low, middle - 1);  
    }  
}
```

- 判断是否已超出了搜索范围，是则返回-1
- 取中位数
- 判断中位数是否为要找的数
- 判断左半边是不是排好序的
- 是则，判断目标值是否在左半边



```
int binarySearch(int[] nums, int target, int low, int high) {  
    if (low > high) {  
        return -1;  
    }  
  
    int middle = low + (high - low) / 2;  
    if (nums[middle] == target) {  
        return middle;  
    }  
  
    if (nums[low] <= nums[middle]) {  
        if (nums[low] <= target && target < nums[middle]) {  
            return binarySearch(nums, target, low, middle - 1);  
        }  
        return binarySearch(nums, target, middle + 1, high);  
    } else {  
        if (nums[middle] < target && target <= nums[high]) {  
            return binarySearch(nums, target, middle + 1, high);  
        }  
        return binarySearch(nums, target, low, middle - 1);  
    }  
}
```

- 判断是否已超出了搜索范围，是则返回-1
- 取中位数
- 判断中位数是否为要找的数
- 判断左半边是不是排好序的
- 是则，判断目标值是否在左半边
- 是则，在左半边继续进行二分搜索



```
int binarySearch(int[] nums, int target, int low, int high) {  
    if (low > high) {  
        return -1;  
    }  
  
    int middle = low + (high - low) / 2;  
    if (nums[middle] == target) {  
        return middle;  
    }  
  
    if (nums[low] <= nums[middle]) {  
        if (nums[low] <= target && target < nums[middle]) {  
            return binarySearch(nums, target, low, middle - 1);  
        }  
        return binarySearch(nums, target, middle + 1, high);  
    } else {  
        if (nums[middle] < target && target <= nums[high]) {  
            return binarySearch(nums, target, middle + 1, high);  
        }  
        return binarySearch(nums, target, low, middle - 1);  
    }  
}
```

- 判断是否已超出了搜索范围，是则返回-1
- 取中位数
- 判断中位数是否为要找的数
- 判断左半边是不是排好序的
- 是则，判断目标值是否在左半边
- 是则，在左半边继续进行二分搜索
- 否则，在右半边进行二分搜索

```
int binarySearch(int[] nums, int target, int low, int high) {  
    if (low > high) {  
        return -1;  
    }  
  
    int middle = low + (high - low) / 2;  
    if (nums[middle] == target) {  
        return middle;  
    }  
  
    if (nums[low] <= nums[middle]) {  
        if (nums[low] <= target && target < nums[middle]) {  
            return binarySearch(nums, target, low, middle - 1);  
        }  
        return binarySearch(nums, target, middle + 1, high);  
    } else {  
        if (nums[middle] < target && target <= nums[high]) {  
            return binarySearch(nums, target, middle + 1, high);  
        }  
        return binarySearch(nums, target, low, middle - 1);  
    }  
}
```

- 判断是否已超出了搜索范围，是则返回-1
- 取中位数
- 判断中位数是否为要找的数
- 判断左半边是不是排好序的
- 是则，判断目标值是否在左半边
- 是则，在左半边继续进行二分搜索
- 否则，在右半边进行二分搜索
- 若右半边是排好序的那一半，判断目标值是否在右边

```
int binarySearch(int[] nums, int target, int low, int high) {  
    if (low > high) {  
        return -1;  
    }  
  
    int middle = low + (high - low) / 2;  
    if (nums[middle] == target) {  
        return middle;  
    }  
  
    if (nums[low] <= nums[middle]) {  
        if (nums[low] <= target && target < nums[middle]) {  
            return binarySearch(nums, target, low, middle - 1);  
        }  
        return binarySearch(nums, target, middle + 1, high);  
    } else {  
        if (nums[middle] < target && target <= nums[high]) {  
            return binarySearch(nums, target, middle + 1, high);  
        }  
        return binarySearch(nums, target, low, middle - 1);  
    }  
}
```

- 判断是否已超出了搜索范围，是则返回-1
- 取中位数
- 判断中位数是否为要找的数
- 判断左半边是不是排好序的
- 是则，判断目标值是否在左半边
- 是则，在左半边继续进行二分搜索
- 否则，在右半边进行二分搜索
- 若右半边是排好序的那一半，判断目标值是否在右边
- 是则，在右半边继续进行二分搜索

```
int binarySearch(int[] nums, int target, int low, int high) {  
    if (low > high) {  
        return -1;  
    }  
  
    int middle = low + (high - low) / 2;  
    if (nums[middle] == target) {  
        return middle;  
    }  
  
    if (nums[low] <= nums[middle]) {  
        if (nums[low] <= target && target < nums[middle]) {  
            return binarySearch(nums, target, low, middle - 1);  
        }  
        return binarySearch(nums, target, middle + 1, high);  
    } else {  
        if (nums[middle] < target && target <= nums[high]) {  
            return binarySearch(nums, target, middle + 1, high);  
        }  
        return binarySearch(nums, target, low, middle - 1);  
    }  
}
```

- 判断是否已超出了搜索范围，是则返回-1
- 取中位数
- 判断中位数是否为要找的数
- 判断左半边是不是排好序的
- 是则，判断目标值是否在左半边
- 是则，在左半边继续进行二分搜索
- 否则，在右半边进行二分搜索
- 若右半边是排好序的那一半，判断目标值是否在右边
- 是则，在右半边继续进行二分搜索
- 否则，在左半边进行二分搜索

## 不定长的边界

有一段不知道具体长度的日志文件，里面记录了每次登陆的时间戳，已知日志是按顺序从头到尾记录的，没有记录日志的地方为空。那么，当前日志的长度是多少？

{ 2019-01-14, 2019-01-17, ... , 2019-08-04, .... , null, null, null ... }

## 直观做法

- ▶ 顺序遍历这个数组，一直遍历下去
- ▶ 当发现第一个null的时候，我们就知道了日志的总数量了

## 二分搜索做法

- ▶ 一开始设置low = 0, high = 1
- ▶ 只要logs[high]不为null, high \*= 2
- ▶ 当logs[high]为null的时候，可以在区间[0, high]进行普通的二分搜索了



```
int getUpperBound(String[] logs, int high) {  
    if (logs[high] == null) {  
        return high;  
    }  
    return getUpperBound(logs, high * 2);  
}
```

```
int binarySearch(String[] logs, int low, int high) {  
    if (low > high) {  
        return -1;  
    }  
  
    int middle = low + (high - low) / 2;  
    if (logs[middle] == null && logs[middle - 1] != null) {  
        return middle;  
    }  
    if (logs[middle] == null) {  
        return binarySearch(logs, low, middle - 1);  
    } else {  
        return binarySearch(logs, middle + 1, high);  
    }  
}
```

- 判先通过getUpperBound函数不断地去试探在什么位置会出现空的日志

```
int getUpperBound(String[] logs, int high) {  
    if (logs[high] == null) {  
        return high;  
    }  
    return getUpperBound(logs, high * 2);  
}
```

```
int binarySearch(String[] logs, int low, int high) {  
    if (low > high) {  
        return -1;  
    }  
  
    int middle = low + (high - low) / 2;  
    if (logs[middle] == null && logs[middle - 1] != null) {  
        return middle;  
    }  
    if (logs[middle] == null) {  
        return binarySearch(logs, low, middle - 1);  
    } else {  
        return binarySearch(logs, middle + 1, high);  
    }  
}
```

- ▶ 判先通过getUpperBound函数不断地去试探在什么位置会出现空的日志
- ▶ 运用二分搜索的方法去寻找日志的长度



## 定义

- ▶ 贪婪是一种在每一步选中都采取在当前状态下最好或最优的选择，从而希望导致结果是最好或最优的算法。

## 优点

- ▶ 对于一些问题，贪婪算法非常的直观有效

## 缺点

- ▶ 往往，它得到的结果并不是正确的
- ▶ 贪婪算法容易过早地做出决定，从而没有办法达到最优解

## 贪婪算法的反例

有一些物品，每个物品都有一定的价值和重量，现在有一个背包，背包能够承受的总重量一定，要求在不超过背包总承受总量的前提下，尽可能让装入背包中的物品总价值最大，问怎么装？

### 有三种不同的贪心策略

- ▶ 选取价值最大的物品
- ▶ 选择重量最轻的物品
- ▶ 选取价值/重量比最大的物品

## 选取价值最大的物品策略

- ▶ 贪婪物品有：A B C
  - ▶ 重量分别是：25, 10, 10
  - ▶ 价值分别是：100, 80, 80
- 拉勾
- ▶ 根据策略，首先选取物品A，接下来就不能再去选其他物品了
  - ▶ 但是，如果选取B和C，结果会更好

## 选择重量最轻的物品策略

- ▶ 贪婪物品有：A B C
- ▶ 重量分别是：25, 10, 10
- ▶ 价值分别是：100, 5, 5

- ▶ 根据策略，首先选取物品B和C，接下来就不能选A了
- ▶ 但是，如果选A，价值更大

## 选取价值/重量比最大的物品策略

- ▶ 贪婪物品有：A B C
- ▶ 重量分别是：25, 10, 10
- ▶ 价值分别是：25, 10, 10
- ▶ 根据策略，三种物品的价值/重量比都是一样
- ▶ 如果选A，答案就不对了。应该选B和C

## 贪婪的弊端

- 总是做出在当前看来是最好的选择
- 不从整体的角度去考虑，仅对局部的最优解感兴趣

## 什么问题适用贪婪算法

- 只有当那些局部最优策略能产生全局最优策略的时候

## 253.会议室II

有给定一系列会议的起始时间和结束时间，求最少需要多少个会议室就可以让这些会议顺利召开。

### 最暴力的做法

- 把所有的会议组合找出来
- 从最长的组合开始检查，看看各个会议之间有没有冲突，直到发现一组会议没有冲突
- 很明显，这样的解法是非常没有效率的

### 贪婪做法

- 会议都是按照它们的起始时间顺序进行的
- 要给新的就要开始的会议找会议室时，先看当前有无空会议室
- 有则在空会议室开会，无则开设一间新会议室

```
int minMeetingRooms(Interval[] intervals) {  
    if (intervals == null || intervals.length == 0)  
        return 0;
```

```
    Arrays.sort(intervals, new Comparator<Interval>() {  
        public int compare(Interval a, Interval b) { return a.start - b.start; }  
    });
```

```
    PriorityQueue<Interval> heap = new  
    PriorityQueue<Interval>(intervals.length, new Comparator<Interval>() {  
        public int compare(Interval a, Interval b) { return a.end - b.end; }  
    });
```

```
    heap.offer(intervals[0]);  
    for (int i = 1; i < intervals.length; i++) {  
        Interval interval = heap.poll();  
        if (intervals[i].start >= interval.end) {  
            interval.end = intervals[i].end;  
        } else {  
            heap.offer(intervals[i]);  
        }  
        heap.offer(interval);  
    }  
    return heap.size();  
}
```

- 将输入的一系列会议按照会议的起始时间排序



```
int minMeetingRooms(Interval[] intervals) {  
    if (intervals == null || intervals.length == 0)  
        return 0;  
  
    Arrays.sort(intervals, new Comparator<Interval>() {  
        public int compare(Interval a, Interval b) { return a.start - b.start; }  
    });
```

```
    PriorityQueue<Interval> heap = new  
    PriorityQueue<Interval>(intervals.length, new Comparator<Interval>() {  
        public int compare(Interval a, Interval b) { return a.end - b.end; }  
    });
```

```
    heap.offer(intervals[0]);  
    for (int i = 1; i < intervals.length; i++) {  
        Interval interval = heap.poll();  
        if (intervals[i].start >= interval.end) {  
            interval.end = intervals[i].end;  
        } else {  
            heap.offer(intervals[i]);  
        }  
        heap.offer(interval);  
    }  
    return heap.size();  
}
```

- 将输入的一系列会议按照会议的起始时间排序
- 用一个最小堆来维护目前开辟的所有会议室，最小堆里的会议室按照会议的结束时间排序

```
int minMeetingRooms(Interval[] intervals) {  
    if (intervals == null || intervals.length == 0)  
        return 0;  
  
    Arrays.sort(intervals, new Comparator<Interval>() {  
        public int compare(Interval a, Interval b) { return a.start - b.start; }  
    });  
  
    PriorityQueue<Interval> heap = new  
    PriorityQueue<Interval>(intervals.length, new Comparator<Interval>() {  
        public int compare(Interval a, Interval b) { return a.end - b.end; }  
    });  
  
    heap.offer(intervals[0]);  
    for (int i = 1; i < intervals.length; i++) {  
        Interval interval = heap.poll();  
        if (intervals[i].start >= interval.end) {  
            interval.end = intervals[i].end;  
        } else {  
            heap.offer(intervals[i]);  
        }  
        heap.offer(interval);  
    }  
    return heap.size();  
}
```

- 将输入的一系列会议按照会议的起始时间排序
- 用一个最小堆来维护目前开辟的所有会议室，最小堆里的会议室按照会议的结束时间排序
- 让第一个会议在第一个会议室里举行

```
int minMeetingRooms(Interval[] intervals) {  
    if (intervals == null || intervals.length == 0)  
        return 0;  
  
    Arrays.sort(intervals, new Comparator<Interval>() {  
        public int compare(Interval a, Interval b) { return a.start - b.start; }  
    });  
  
    PriorityQueue<Interval> heap = new  
    PriorityQueue<Interval>(intervals.length, new Comparator<Interval>() {  
        public int compare(Interval a, Interval b) { return a.end - b.end; }  
    });  
  
    heap.offer(intervals[0]);  
    for (int i = 1; i < intervals.length; i++) {  
        Interval interval = heap.poll();  
        if (intervals[i].start >= interval.end) {  
            interval.end = intervals[i].end;  
        } else {  
            heap.offer(intervals[i]);  
        }  
        heap.offer(interval);  
    }  
    return heap.size();  
}
```

- 将输入的一系列会议按照会议的起始时间排序
- 用一个最小堆来维护目前开辟的所有会议室，最小堆里的会议室按照会议的结束时间排序
- 让第一个会议在第一个会议室里举行
- 从第二个会议开始，每个会议都从最小堆里取出一个会议室

```
int minMeetingRooms(Interval[] intervals) {  
    if (intervals == null || intervals.length == 0)  
        return 0;  
  
    Arrays.sort(intervals, new Comparator<Interval>() {  
        public int compare(Interval a, Interval b) { return a.start - b.start; }  
    });  
  
    PriorityQueue<Interval> heap = new  
    PriorityQueue<Interval>(intervals.length, new Comparator<Interval>() {  
        public int compare(Interval a, Interval b) { return a.end - b.end; }  
    });  
  
    heap.offer(intervals[0]);  
    for (int i = 1; i < intervals.length; i++) {  
        Interval interval = heap.poll();  
        if (intervals[i].start >= interval.end) {  
            interval.end = intervals[i].end;  
        } else {  
            heap.offer(intervals[i]);  
        }  
        heap.offer(interval);  
    }  
    return heap.size();  
}
```

- 将输入的一系列会议按照会议的起始时间排序
- 用一个最小堆来维护目前开辟的所有会议室，最小堆里的会议室按照会议的结束时间排序
- 让第一个会议在第一个会议室里举行
- 从第二个会议开始，每个会议都从最小堆里取出一个会议室
- 若当前要开的会议可以等会议室被腾出才开始，那么就可以重复利用这个会议室

```
int minMeetingRooms(Interval[] intervals) {  
    if (intervals == null || intervals.length == 0)  
        return 0;  
  
    Arrays.sort(intervals, new Comparator<Interval>() {  
        public int compare(Interval a, Interval b) { return a.start - b.start; }  
    });  
  
    PriorityQueue<Interval> heap = new  
    PriorityQueue<Interval>(intervals.length, new Comparator<Interval>() {  
        public int compare(Interval a, Interval b) { return a.end - b.end; }  
    });  
  
    heap.offer(intervals[0]);  
    for (int i = 1; i < intervals.length; i++) {  
        Interval interval = heap.poll();  
        if (intervals[i].start >= interval.end) {  
            interval.end = intervals[i].end;  
        } else {  
            heap.offer(intervals[i]);  
        }  
        heap.offer(interval);  
    }  
    return heap.size();  
}
```

- 将输入的一系列会议按照会议的起始时间排序
- 用一个最小堆来维护目前开辟的所有会议室，最小堆里的会议室按照会议的结束时间排序
- 让第一个会议在第一个会议室里举行
- 从第二个会议开始，每个会议都从最小堆里取出一个会议室
- 若当前要开的会议可以等会议室被腾出才开始，那么就可以重复利用这个会议室
- 否则，开辟一个新的会议室



```
int minMeetingRooms(Interval[] intervals) {  
    if (intervals == null || intervals.length == 0)  
        return 0;  
  
    Arrays.sort(intervals, new Comparator<Interval>() {  
        public int compare(Interval a, Interval b) { return a.start - b.start; }  
    });  
  
    PriorityQueue<Interval> heap = new  
    PriorityQueue<Interval>(intervals.length, new Comparator<Interval>() {  
        public int compare(Interval a, Interval b) { return a.end - b.end; }  
    });  
  
    heap.offer(intervals[0]);  
    for (int i = 1; i < intervals.length; i++) {  
        Interval interval = heap.poll();  
        if (intervals[i].start >= interval.end) {  
            interval.end = intervals[i].end;  
        } else {  
            heap.offer(intervals[i]);  
        }  
        heap.offer(interval);  
    }  
    return heap.size();  
}
```

- 将输入的一系列会议按照会议的起始时间排序
- 用一个最小堆来维护目前开辟的所有会议室，最小堆里的会议室按照会议的结束时间排序
- 让第一个会议在第一个会议室里举行
- 从第二个会议开始，每个会议都从最小堆里取出一个会议室
- 若当前要开的会议可以等会议室被腾出才开始，那么就可以重复利用这个会议室
- 否则，开辟一个新的会议室
- 接下来，把旧的会议室也放入到最小堆里

```
int minMeetingRooms(Interval[] intervals) {  
    if (intervals == null || intervals.length == 0)  
        return 0;  
  
    Arrays.sort(intervals, new Comparator<Interval>() {  
        public int compare(Interval a, Interval b) { return a.start - b.start; }  
    });  
  
    PriorityQueue<Interval> heap = new  
    PriorityQueue<Interval>(intervals.length, new Comparator<Interval>() {  
        public int compare(Interval a, Interval b) { return a.end - b.end; }  
    });  
  
    heap.offer(intervals[0]);  
    for (int i = 1; i < intervals.length; i++) {  
        Interval interval = heap.poll();  
        if (intervals[i].start >= interval.end) {  
            interval.end = intervals[i].end;  
        } else {  
            heap.offer(intervals[i]);  
        }  
        heap.offer(interval);  
    }  
    return heap.size();  
}
```

- 将输入的一系列会议按照会议的起始时间排序
- 用一个最小堆来维护目前开辟的所有会议室，最小堆里的会议室按照会议的结束时间排序
- 让第一个会议在第一个会议室里举行
- 从第二个会议开始，每个会议都从最小堆里取出一个会议室
- 若当前要开的会议可以等会议室被腾出才开始，那么就可以重复利用这个会议室
- 否则，开辟一个新的会议室
- 接下来，把旧的会议室也放入到最小堆里
- 最小堆里的会议室个数就是最少的会议个数

## 为什么贪婪算法在此处成立

- ▶ 每当遇到一个新的会议时，总是贪婪地从所有会议室里找出最先结束会议的那个

## 为什么这样可以产生最优结果

- ▶ 若选择的会议室中会议未结束，则意味着需要开辟一个新会议室，这已经不是当前的最优解了