

# 基础算法

---

## 基础算法

### 排序算法

快速排序

归并排序

堆排序

215. 数组中的第K个最大元素

### 二分查找

算法

定义

示例分析

执行过程

代码

易错点

题目

300. 最长递增子序列

35. 搜索插入位置

34. 在排序数组中查找元素的第一个和最后一个位置

题目变种：找出第一个大于目标元素的元素索引

题目变种：找出最后一个小于目标元素的元素索引

33. 搜索旋转排序数组

81. 搜索旋转排序数组 II

153. 寻找旋转排序数组中的最小值

74. 搜索二维矩阵

50. Pow(x, n)

162. 寻找峰值

69. x 的平方根

回溯

46. 全排列

栈

20. 有效的括号

其他

146. LRU 缓存机制

# 排序算法

---

排序方法比较

快速排序和堆排序

1.对于快速排序来说，数据是顺序访问的。而对于堆排序来说，数据是跳着访问的。这样对 CPU 缓存是不友好的

2.相同的数据，排序过程中，堆排序的数据交换次数要多于快速排序。

所以上面两条也就说明了在实际开发中，堆排序的性能不如快速排序性能好。

# 快速排序

## 912. 排序数组

给你一个整数数组 nums，请你将该数组升序排列。

示例 1：

输入： nums = [5,2,3,1]

输出： [1,2,3,5]

示例 2：

输入： nums = [5,1,1,2,0,0]

输出： [0,0,1,1,2,5]

- 思路

- 快速排序的主要思想是将待排序序列划分，得到两部分，其中前一部分数据都比某元素小，后一部分数据都比某元素大；然后递归调用函数对前后两部分数据分别进行快速排序，得到有序序列
- 定义函数 quickSortHelper(nums, left, right) 函数对 [left:right] 区间划分并排序，得到前后两个有序区间。然后递归调用 quickSort(nums, 0, pos - 1) 和 quickSort(nums, pos + 1, right) 即可
- 随机获取 pivot，否则测试用例无法通过噢



- 代码

```
// https://leetcode-cn.com/problems/sort-an-array/solution/pai-xu-shu-zu-by-leetcode-solution/
// 快速排序
class Solution {
public:
    vector<int> sortArray(vector<int>& nums) {
        srand((unsigned)time(NULL));
        quickSort(nums, 0, nums.size() - 1);
        return nums;
    }

    void quickSort(vector<int> &nums, int left, int right) {
        if (left < right) {
            int pos = quickSortHelper(nums, left, right);
            quickSort(nums, left, pos - 1);
            quickSort(nums, pos + 1, right);
        }
    }
}
```

```

int quickSortHelper(vector<int> &nums, int left, int right) {
    int pos = rand() % (right - left + 1) + left;
    swap(nums[pos], nums[right]);

    int pivot = nums[right];
    int i = left - 1;
    for (int j = left; j < right; ++j) {
        if (nums[j] <= pivot) {
            ++i;
            swap(nums[i], nums[j]);
        }
    }
    swap(nums[+i], nums[right]);
    return i;
}
};

```

- 复杂度

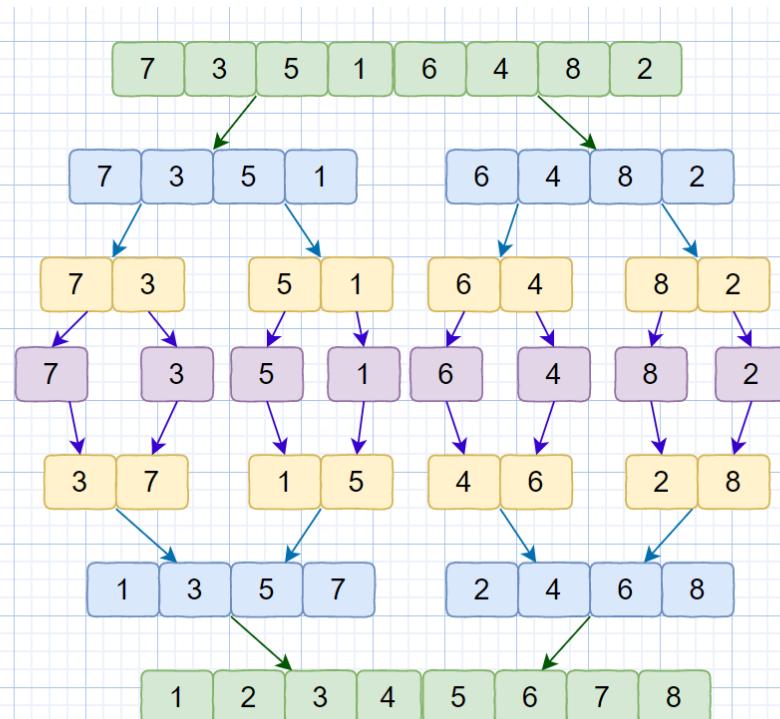
- 时间复杂度：基于随机选取主元的快排时间复杂度为 $O(n \log n)$ ，其中 $n$ 为数组长度，证明见 [算法导论第七章](#)
- 空间复杂度： $O(h)$ ，其中 $h$ 为快排递归调用的栈空间。最坏为 $O(n)$ ，最优为 $O(\log n)$

## 归并排序

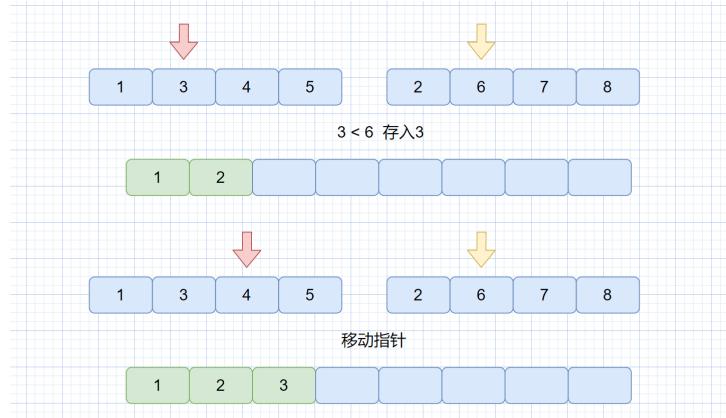
题目见上

- 思路

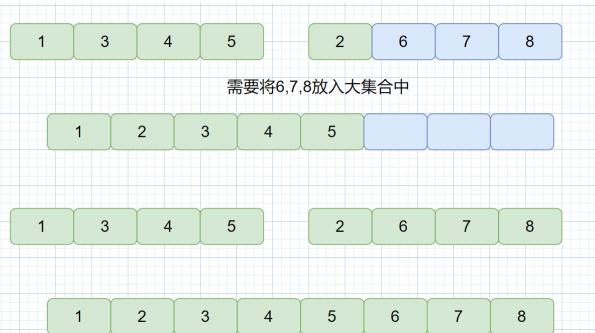
- 归并，表示合并、并入，在数据结构中，就是 **将两个或两个以上的有序表合并成一个新的有序表**，归并排序就是使用 **归并** 的思想实现的排序方法
- 归并排序使用的是 **分治思想**，顾名思义就是分而治之。讲一个大问题分解成若干个小的子问题来解决，小的子问题解决了，那么大问题也就解决了。
- 归并排序如下图



- 第一步：创建一个临时数组，用于存储归并结果，长度为两个待排序数组的区间长度和
- 第二步：自左向右比较两个指针指向的值，将较小的那个存入临时数组中，移动该指针，继续比较，直到某个数组全部元素都存入到临时数组中



- 当某个数组元素全部放入临时数组中，则将另一个数组剩余元素全部存入临时数组中



- 将临时数组元素拷贝回原数组

- 代码

```
// https://leetcode-cn.com/problems/sort-an-array/solution/ni-jue-dui-neng-gou-kan-dong-de-gui-bing-dqko/
// 归并排序
class Solution {
public:
    vector<int> sortArray(vector<int>& nums) {
        vector<int> tmp(nums.size(), 0);
        mergeSort(nums, 0, nums.size() - 1, tmp);
        return nums;
    }
    void mergeSort(vector<int> &nums, int left, int right, vector<int> &tmp)
    {
        if (left < right) {
            int mid = left + ((right - left) >> 1);
            mergeSort(nums, left, mid, tmp);
            mergeSort(nums, mid + 1, right, tmp);
            merge(nums, left, mid, right);
        }
    }
    void merge(vector<int> &nums, int left, int mid, int right, vector<int> &tmp)
    {
        // l区间[left, mid], r区间[mid + 1, right]
        int l = left, r = mid + 1, idx = 0;
        // 比较两区间序列大小，将小的值存入tmp中
        while (l <= mid && r <= right) {
            if (nums[l] <= nums[r]) {
                tmp[idx] = nums[l];
                l++;
            } else {
                tmp[idx] = nums[r];
                r++;
            }
            idx++;
        }
        if (l > mid) {
            for (int i = r; i <= right; i++) {
                tmp[idx] = nums[i];
                idx++;
            }
        } else {
            for (int i = l; i <= mid; i++) {
                tmp[idx] = nums[i];
                idx++;
            }
        }
    }
}
```

```

        tmp[idx++] = nums[l++];
    } else {
        tmp[idx++] = nums[r++];
    }
}
// 将剩余元素存入tmp中
while (l <= mid) {
    tmp[idx++] = nums[l++];
}
while (r <= right) {
    tmp[idx++] = nums[r++];
}
for (int i = left, k = 0; i <= right; ++i, ++k) {
    nums[i] = tmp[k];
}
}
};


```

- 复杂度

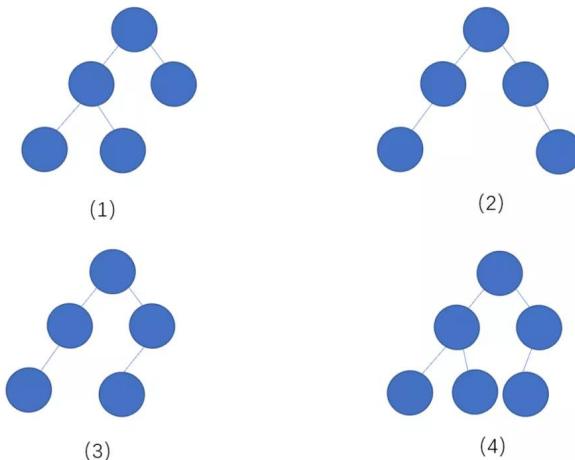
- 时间复杂度： $O(n \log n)$ ，归并一次需要扫描待排序数组，为 $O(n)$ ；归并排序把待排序数组一层层折半分组，由完全二叉树深度可知，共需要 $\log n$ 次，故总时间复杂度为 $O(n \log n)$ 。并且，归并排序效率与原数组有序程度无关，故最好、最坏、平均情况下时间复杂度都一样。
- 空间复杂度： $O(n)$ ，临时数组空间为 $O(n)$ ，因为空间复杂度问题，导致不如快排应用广泛

## 堆排序

题目见上

- 思路

- 堆排序利用了堆结构，而堆又需要了解完全二叉树
- 完全二叉树：叶子结点只能出现在最下层和次下层，且最下层的叶子结点集中在数的左部，即除了最后一层，其他层节点个数都是满的，并且最后一层的叶子结点必须靠左

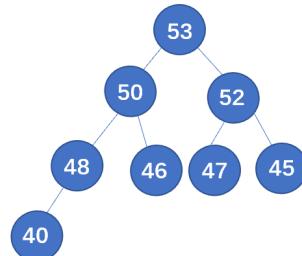


上面的几个例子中，(1) (4) 为完全二叉树，(2) (3) 不是完全二叉树，因为他们不满足最后一层叶子节点必须靠左。通过上面的几个例子，我们了解了什么是完全二叉树，

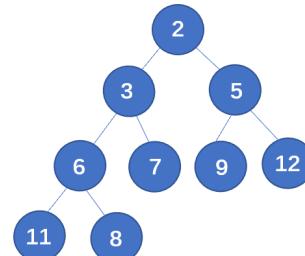
- 二叉堆：

- 必须是完全二叉树
- 二叉堆中的每一个节点，都必须大于等于（或小于等于）其子树中每个节点的值

若是每个节点大于等于子树节点，称为 **大顶堆**；若小于等于子树中每个节点，称为 **小顶堆**



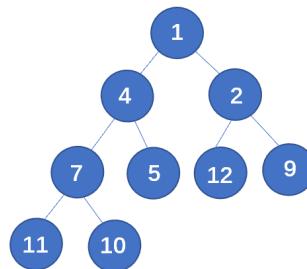
大顶堆



小顶堆

- 那二叉堆如何存储呢？

- 因为堆是完全二叉树，故可使用数组存储，具体思想见下图，我们仅按照顺序将节点存入数组即可，如下通过小顶堆演示，并且从下标1开始存储，省略计算。后续将二叉堆简称为堆



index    0    1    2    3    4    5    6    7    8    9

- 为什么可使用数组存储堆呢？首先看值为1的根节点，其下标为1；它的左子节点，值为4，下标为2；右子节点值为2，下标为3；发现其中的关系了吗？数组中，某节点（非叶子节点）的下标为  $i$ ，则其左子节点下标为  $2 * i$ （这里可直接相乘得到左孩子，若下标从0开始，则需要  $2 * i + 1$ ），右子节点下标为  $2 * i + 1$ ，其父节点下标为  $i / 2$ 。所以完全可以根据索引找到某节点的左子节点、右子节点和父节点，所以使用数组存储完全没问题

- 那如何利用堆进行排序呢？堆排序其实主要有两个步骤：建堆和排序

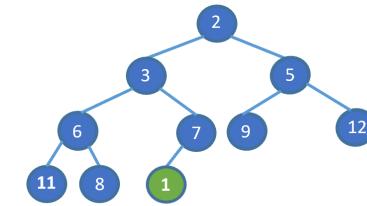
- 建堆

- 用数组来存储大顶（小顶）堆，此时的元素已经满足某节点大于等于（或小于等于）子树节点，但是随机给我们一个数组，此时并不一定满足上述要求，所以我们需要调整数组，使其满足大顶堆或小顶堆的要求。这个就是堆化，也可以称其为建堆。

- 建堆有两种方法

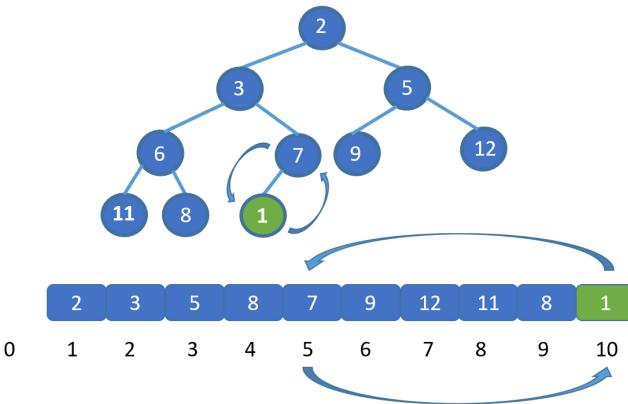
- 上浮操作：就是不断插入元素进行建堆

- 说之前我们先来了解下，如何往已经建好的堆里，插入新的元素，我们直接看例子吧，一下就懂啦

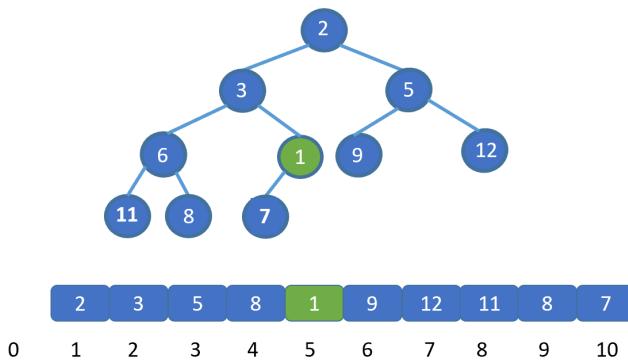


2	3	5	8	7	9	12	11	8	10	1
0	1	2	3	4	5	6	7	8	9	10

- 假设让我们插入新的元素 1（绿色节点），我们发现此时 1 小于其父节点的值 7，并不遵守小顶堆的规则，那我们则需要移动元素 1。让 1 与 7 交换。（如果新插入元素大于父节点的值，则说明插入新节点后仍满足小顶堆规则，无需交换）。
- 我们可以用数组保存堆，并且可以通过  $i/2$  得到其父节点的值，那么此时我们就明白怎么做啦。



- 将插入节点与其父节点，交换。



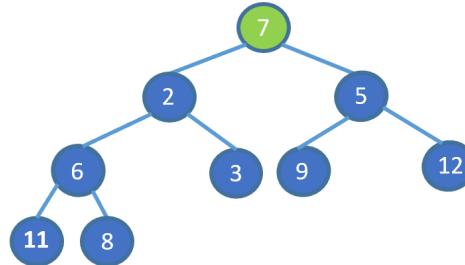
- 交换之后，我们继续将新插入元素，也就是 1 与其父节点比较，如果大于其父节点，则无需交换，循环结束。若小于则需要继续交换，直到 1 到达适合他的地方。大家是不是已经直道该怎么做啦！
- 其实很简单，我们只需将新加入元素与其父节点比较，判断是否小于堆顶元素（小顶堆），如果小于则进行交换，（让更小的节点为父节点）直到符合堆的规则位置，大顶堆则相反。

我们发现，我们新插入的元素是不是一层层的上浮，直到找到属于自己的位置，我们将这个操作称之为上浮操作。

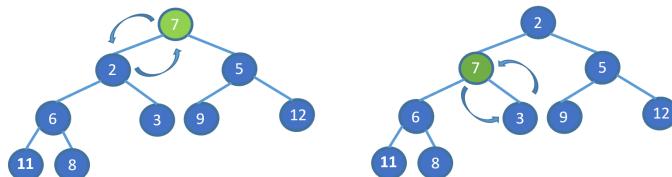
- 那我们知道了上浮，岂不是就可以实现建堆了？是的，我们则可以依次遍历数组，就好比不断往堆中插入新元素，直至遍历结束，这样我们就完成了建堆，这种方法当然是可以的。

- 下沉操作：遍历父节点，不断将其下沉，进行建堆

- 给我们一个无序数组（不满足堆的要求），见下图



- 我们发现，7 位于堆顶，但是此时并不满足小顶堆的要求，我们需要把 7 放到属于它的位置，我们应该怎么做呢？我们需要与孩子节点中最小的那个交换，因为我们需要交换后，父节点小于两个孩子节点，如果我们第一步，7 与 5 进行交换的话，则同样不能满足小顶堆。



- 那我们怎么判断节点找到属于它的位置了呢？主要有两个情况

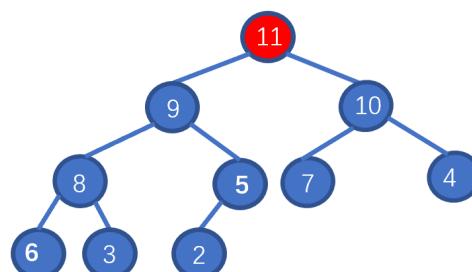
- 待下沉元素小于（大于）两个子节点，此时符合堆的规则，无序下沉，例如上图中的 6
  - 下沉为叶子节点，此时没有子节点，例如 7 下沉到最后变成了叶子节点。

我们将上面的操作称之为下沉操作。

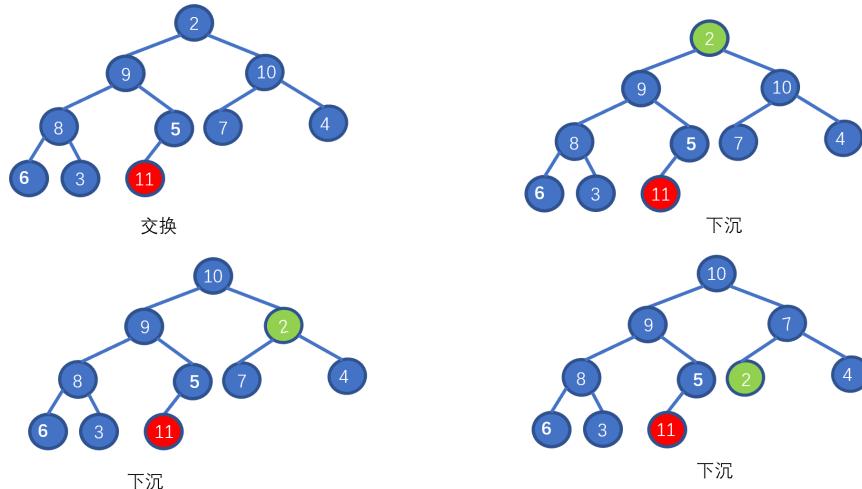
- 那么建堆就是从最后一个非叶子节点开始，依次执行下沉操作。执行完毕后我们就能够完成堆化

- 排序

- 了解排序之前我们先来，看一下如何删除堆顶元素，我们需要保证的是，删除堆顶元素后，其他元素仍能满足堆的要求，我们思考一下如何实现呢？见下图



- 假设我们想要去除堆顶的 11，那我们则需要将其与堆的最后一个节点交换也就是 2，2 然后再执行下沉操作，执行完毕后仍能满足堆的要求，见下图



- 排序就是，逐渐将堆顶元素与最后一个元素交换，然后执行下沉操作，完成后，得到的数组就是排序好的数组。

#### ■ 堆排序流程

- 建堆，通过下沉操作建堆效率更高，具体过程是，找到最后一个非叶子节点，然后从后往前遍历执行下沉操作。
- 排序，将堆顶元素（代表最大元素）与最后一个元素交换，然后新的堆顶元素进行下沉操作，遍历执行上述操作，则可以完成排序。

- 代码

```
// https://leetcode-cn.com/problems/sort-an-array/solution/dong-hua-mo-ni-yi-ge-po-dui-pai-wo-gao-l-i6mt/
// 堆排序
class Solution {
public:
    vector<int> sortArray(vector<int>& nums) {
        // 拷贝到临时数组中
        std::vector<int> tmp(nums.size() + 1, 0);
        for (int i = 0; i < nums.size(); ++i) {
            tmp[i + 1] = nums[i];
        }

        // 建堆
        for (int i = nums.size() / 2; i >= 1; --i) {
            sink(tmp, i, nums.size());
        }

        // 排序
        int k = nums.size();
        while (k > 1) {
            swap(tmp[1], tmp[k--]);
            sink(tmp, 1, k);
        }

        // 拷贝回原数组
        for (int i = 1; i < tmp.size(); ++i) {
            nums[i - 1] = tmp[i];
        }
    }
}
```

```

        return nums;
    }

    void sink(vector<int> &tmp, int idx, int size) {
        while (2 * idx <= size) {
            int child_idx = 2 * idx;
            // 找出子节点最大或最小的那个
            if (child_idx + 1 <= size && tmp[child_idx] < tmp[child_idx + 1]) {
                ++child_idx;
            }
            if (tmp[child_idx] > tmp[idx]) {
                swap(tmp[child_idx], tmp[idx]);
            } else {
                break;
            }
            idx = child_idx;
        }
    }
};

```

- 复杂度

- 时间复杂度：\$O(n \log n)\$ 建堆的时间复杂度为 \$O(n)\$，排序过程的时间复杂度为 \$O(n \log n)\$，所以总的空间复杂度为 \$O(n \log n)\$
- 空间复杂度：这里需要注意，我们上面的描述过程中，为了更直观的描述，空出数组的第一位，这样我们就可以通过 \$i \* 2\$ 和 \$i \* 2 + 1\$ 来求得左孩子节点和右孩子节点。我们也可以根据 \$i \* 2 + 1\$ 和 \$i \* 2 + 2\$ 来获取孩子节点，这样则不需要临时数组来处理原数组，将所有元素后移一位，所以堆排序的空间复杂度为 \$O(1)\$，是原地排序算法。

## 215. 数组中的第K个最大元素

在未排序的数组中找到第 k 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

示例 1：

输入: [3,2,1,5,6,4] 和 k = 2

输出: 5

示例 2：

输入: [3,2,3,1,2,4,5,5,6] 和 k = 4

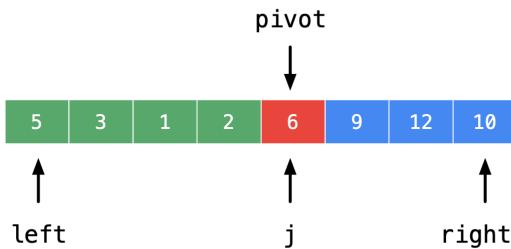
输出: 4

- 快速选择

- 思路：借助快排中的 `partition` 操作，定位到最终排序中索引为 `len - k` 的目标元素，记住要随机切分元素
  - 快速排序虽然快，但是如果实现得不好，在遇到特殊测试用例的时候，时间复杂度会变得很高。如果你使用 `partition` 的方法完成这道题，时间排名不太理想，可以考虑一下是什么问题，这个问题很常见。
  - `partition` 操作
    - 对于某个索引 `j`，`nums[j]` 已经排定，即 `nums[j]` 经过 `partition` 操作后会放置在最终应该在的地方
    - `nums[left] -> nums[j - 1]` 中的所有元素都不大于 `nums[j]`
    - `nums[j + 1] -> nums[right]` 中的所有元素都不小于 `nums[j]`

经过一次 partition , “6” 呆在了它最终应该呆的位置，因为：

- 1、区间  $[left, j - 1]$  里的元素小于等于 6；
- 2、区间  $[j + 1, right]$  里的元素大于等于 6。



注意：选择 pivot 的时候，应该先在数组中随机选择一个元素，然后再执行 partition。以避免递归树加深、复杂度最坏的情况出现。

- partition 操作总能排定一个元素，还能够知道这个元素它最终所在的位置，这样每经过一次 partition (切分) 操作就能缩小搜索的范围，这样的思想叫做“减而治之”(是“分而治之”思想的特例)。

- 代码

```
class Solution {
public:
    // 快速选择
    int findKthLargest(vector<int>& nums, int k) {
        return quickSelect(nums, k);
    }

    int quickSelect(vector<int> &nums, int k) {
        vector<int> biggers, equals, smaller;
        int pos = rand() % nums.size();
        int pivot = nums[pos];
        for (auto num : nums) {
            if (num > pivot) {
                biggers.push_back(num);
            } else if (num == pivot) {
                equals.push_back(num);
            } else {
                smaller.push_back(num);
            }
        }

        if (k <= biggers.size()) {
            return quickSelect(biggers, k);
        }
        if (biggers.size() + equals.size() < k) {
            return quickSelect(smaller, k - biggers.size() -
equals.size());
        }
        return pivot;
    }

    // 桶排序
    /*
    int findKthLargest(vector<int>& nums, int k) {
        if (nums.size() < k) { return 0; }
        vector<int> tmp(20001, 0);
        for (int i = 0; i < nums.size(); ++i) {
```

```

        ++tmp[nums[i] + 10000];
    }
    for (int i = 20000; i >= 0; --i) {
        k -= tmp[i];
        if (k <= 0) {
            return i - 10000;
        }
    }
    return 0;
}
*/
};

```

- 复杂度

- 时间复杂度:  $O(n)$ ,  $n$ 是数组长度
- 空间复杂度:  $O(1)$ , 原地排序

## 二分查找

<https://leetcode-cn.com/problems/find-first-and-last-position-of-element-in-sorted-array/solution/yi-wen-dai-ni-gao-ding-er-fen-cha-zhao-j-ymwl/>

### 算法

#### 定义

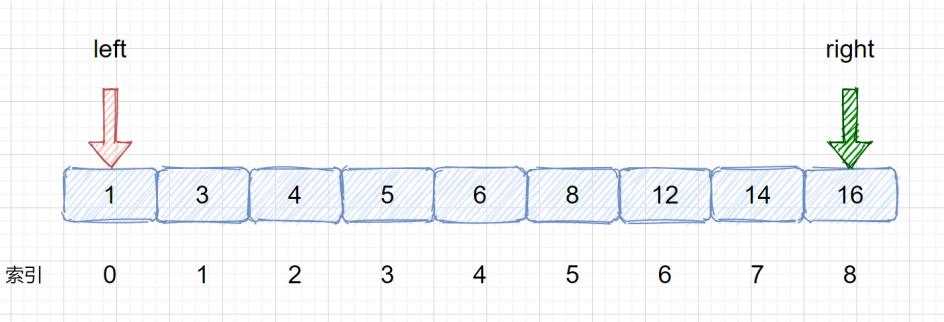
二分查找也称为折半查找 (Binary Search)，是一种在有序数组中查找某一特定元素的搜索算法。我们可以从定义可知，运用二分搜索的前提是数组必须是有序的，这里需要注意的是，输入不一定是数组，也可以是数组中某一区间的起始位置和终止位置

#### 示例分析

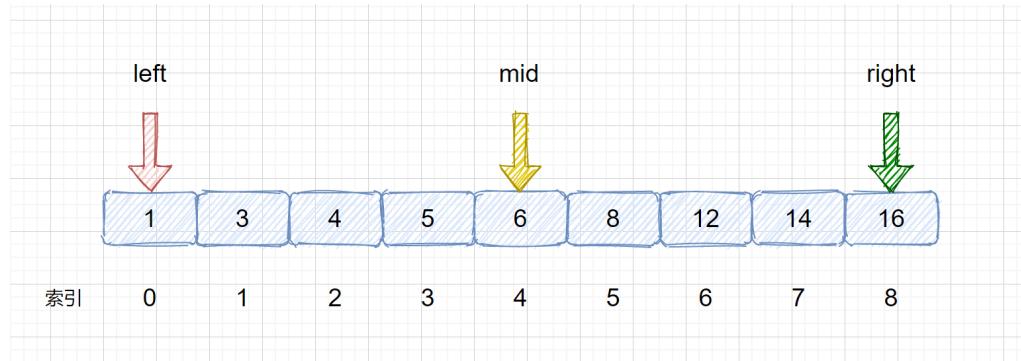
通过上述二分查找定义，知道了二分查找算法的作用及要求，那么该算法的具体执行过程是怎样的呢？下面举例帮助理解。比如存在 `nums` 数组，查询元素 8 的索引

```
int[ ]  nums = {1,3,4,5,6,8,12,14,16}; target = 8
```

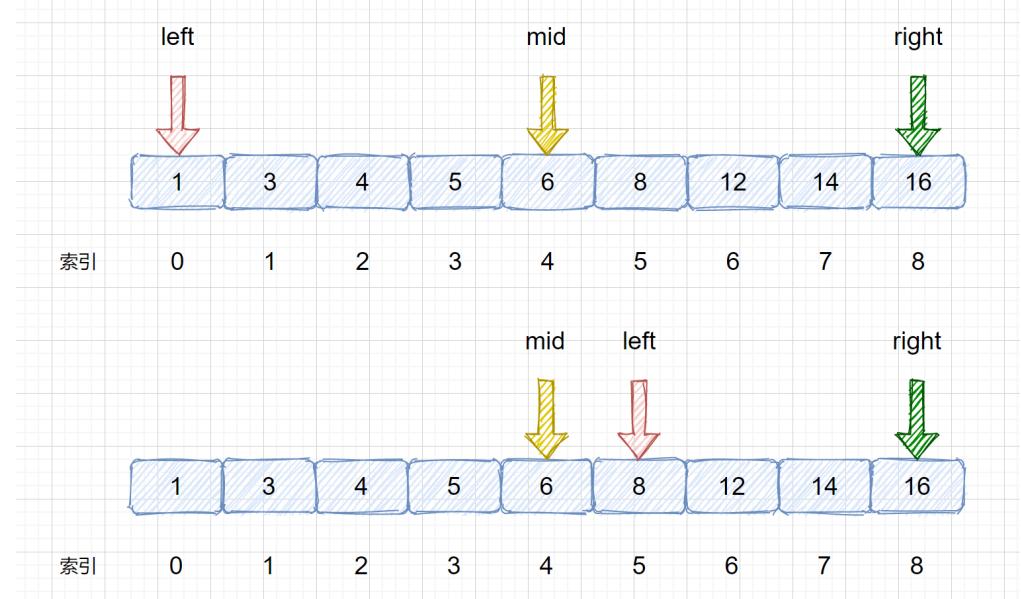
1. 需要定义两个指针分别指向数组头部及尾部，这是在整个数组中查询的情况。若查询某区间，则可以输入数组、起始位置、终止位置进行查询



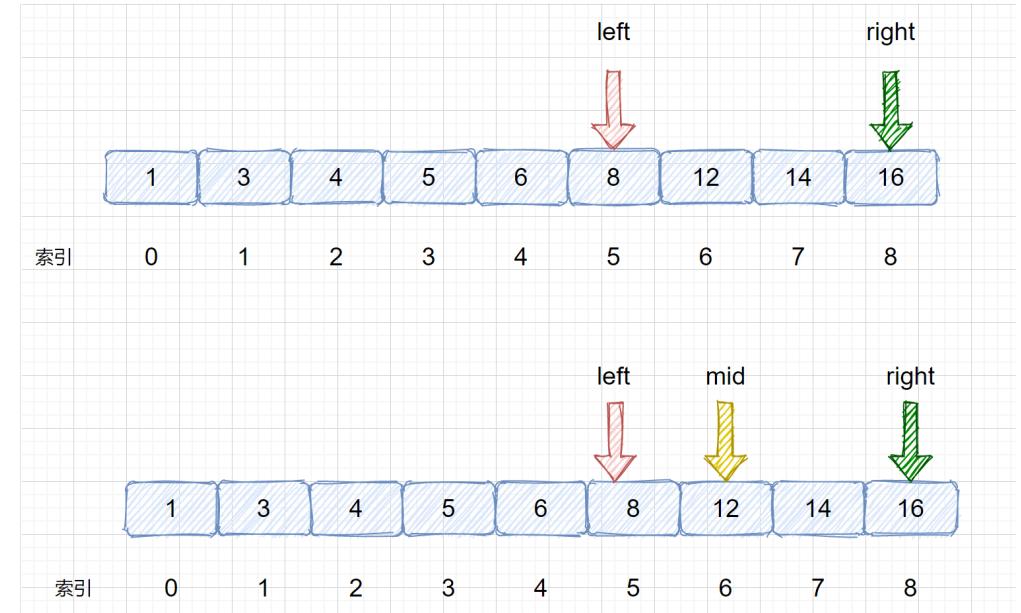
2. 找出mid，该索引为 `mid = (left + right) / 2`，这样写可能溢出，改进写法 `mid = left + (right - left) / 2` 或者 `left + ((right - left) >> 1)`，找到两指针的中点索引，使用位运算写法更快点。那么此时 `mid = 0 + (8 - 0) / 2 = 4`



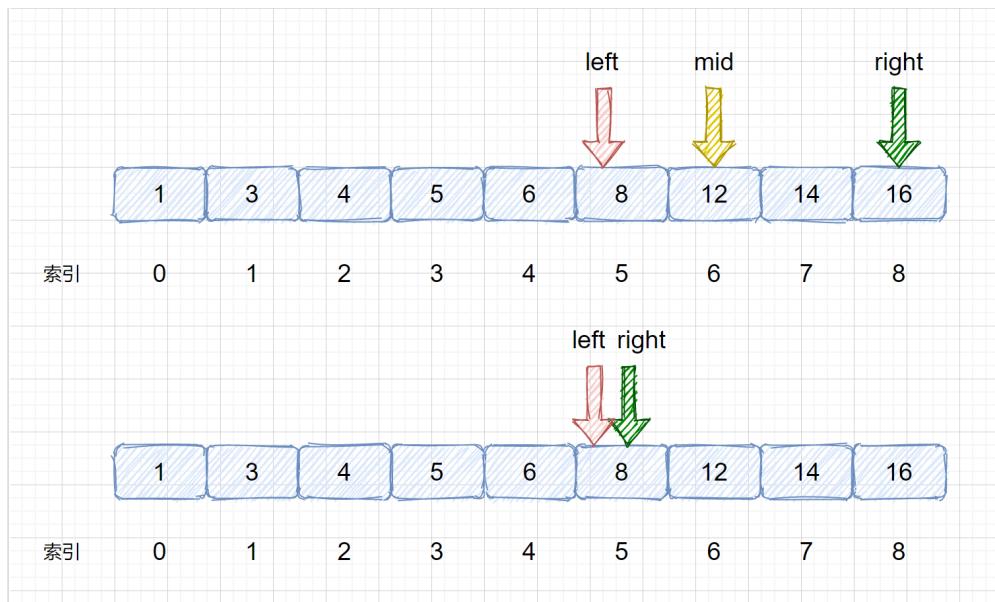
3. 此时我们的 `mid = 4`, `nums[mid] = 6 < target`, 那么此时需要移动 `left` 指针, 让 `left = mid + 1`, 下次则可在新的 `[left, right]` 区间内搜索目标值, 下图为移动前和移动后



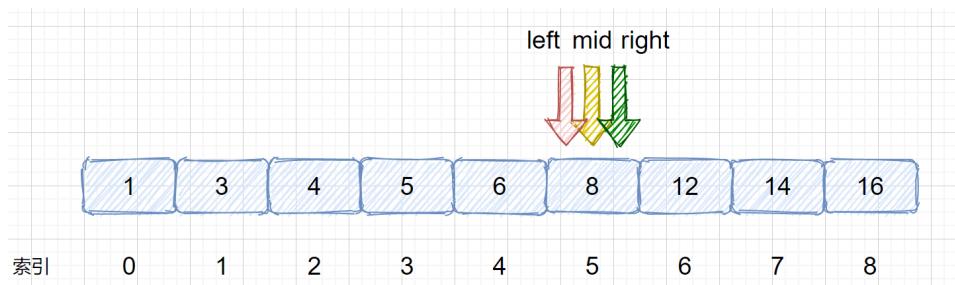
4. 此时需要重新计算 `[left, right]` 区间的 `mid` 值, `mid = 5 + (8 - 5) / 2 = 6`, 然后将 `nums[left]` 和 `target` 继续比较, 进而决定下次移动 `left` 指针还是 `right` 指针, 见下图



5. 我们发现 `target < nums[mid]`, 则需要移动 `right` 指针, 则 `right = mid - 1`, 移动过后 `left, right` 会重合, 这里需要注意一个重点, 后面做详细描述



6. 我们需要在  $[left, right]$  区间继续计算 mid 值，则  $mid = 5 + (5 - 5) / 2 = 5$ ，见下图，此时比较  $\text{nums}[mid]$  和  $\text{target}$ ，发现两值相等，返回  $mid$  即可，若不相等，则跳出循环，返回 -1



## 执行过程

- 从排序数组或区间中，取出中间位置元素，与目标值比较，判断是否相等，若相等，则返回
- 若不等，则比较大小，根据比较结果决定是在  $mid$  的左半部分还是右半部分继续搜索。若  $\text{nums}[mid] < \text{target}$ ，则在右半区间继续搜索，即移动左指针  $left = mid + 1$ ；若  $\text{target} < \text{nums}[mid]$ ，则在左半区间继续搜索，即移动右指针  $right = mid - 1$



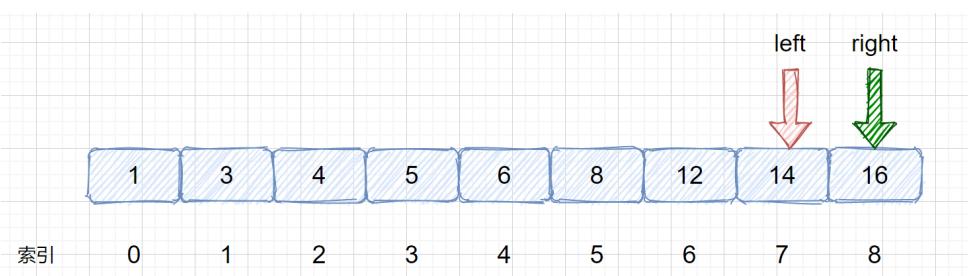
## 代码

```
// 迭代
int binarySearch(vector<int> nums, int target) {
    // 注意循环条件
    while (left <= right) {
        int mid = left + ((right - left) >> 1);
        if (nums[mid] == target) {
            return mid;
        } else if (nums[mid] < target) { // 右半部分
            left = mid + 1;
        } else if (target < nums[mid]) { // 左半部分
            right = mid - 1;
        }
    }
    // 没有找到目标元素，返回-1
    return -1;
}

// 递归
int binarySearch(vector<int> nums, int target, int left, int right) {
    if (left <= right) {
        int mid = left + ((right - left) >> 1);
        if (nums[mid] == target) {
            return mid;
        } else if (nums[mid] < target) { // 右半区间
            return binarySearch(nums, target, mid + 1, right);
        } else if (target < nums[mid]) { // 左半区间
            return binarySearch(nums, target, left, mid - 1);
        }
    }
    // 没有找到目标元素，返回-1
    return -1;
}
```

## 易错点

1. 计算 mid 时，不能使用 `(left + right) / 2`，可能会溢出
2. `while (left <= right) {}` 注意括号内为 `left <= right`，而不是 `left < right`。比如刚才例子，若设置 `left < right`，则当执行到最后一步时，即 `left, right` 重叠时，则会跳出循环，返回 `-1`，表示不存在目标元素，但真实不是这样的，`left` 和 `right` 此时指向的就是目标元素，但此时 `left == right` 导致跳出循环
3. `left = mid + 1; right = mid - 1` 而不是 `left = mid; right = mid`。思考下图，当 `target = 16` 时，此时 `left = 7, right = 8`，计算 `mid = 7 + (8 - 7) / 2 = 7`，若设置 `left = mid` 的话，则会死循环，`mid` 会一直等于 `7`



# 题目

## 300. 最长递增子序列

给你一个整数数组 `nums`，找到其中最长严格递增子序列的长度。

**子序列** 是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，`[3, 6, 2, 7]` 是数组 `[0, 3, 1, 6, 2, 2, 7]` 的子序列

**示例 1：**

```
输入: nums = [10, 9, 2, 5, 3, 7, 101, 18]
输出: 4
解释: 最长递增子序列是 [2, 3, 7, 101]，因此长度为 4 。
```

**示例 2：**

```
输入: nums = [0, 1, 0, 3, 2, 3]
输出: 4
```

**示例 3：**

```
输入: nums = [7, 7, 7, 7, 7, 7, 7]
输出: 1
```

- 思路：二分查找
  - 无序列表最关键的一句在于：`数组 d[i]` 表示长度为 `i` 的最长上升子序列的末尾元素的最小值，即在数组 `1, 2, 3, 4, 5, 6` 中长度为3的上升子序列可以为 `1, 2, 3` 也可以为 `2, 3, 4` 等等但是 `d[3]=3`，即子序列末尾元素最小为3。
  - 无序列表解释清了数组d的含义之后，我们接着需要证明数组d具有单调性，即证明 `i < j` 时，`d[i] < d[j]`，使用反证法，假设存在 `k < j` 时，`d[k] > d[j]`，但在长度为 `j`，末尾元素为 `d[j]` 的子序列A中，将后 `j-i` 个元素减掉，可以得到一个长度为 `i` 的子序列B，其末尾元素 `t1` 必然小于 `d[j]`（因为在子序列A中，`t1` 的位置上在 `d[j]` 的后面），而我们假设数组d必须符合 表示长度为 `i` 的最长上升子序列的末尾元素的最小值，此时长度为 `i` 的子序列的末尾元素 `t1 < d[j] < d[k]`，即 `t1 < d[k]`，所以 `d[k]` 不是最小的，与题设相矛盾，因此可以证明其单调性
  - 无序列表证明单调性有两个好处：1. 可以使用二分法；2. 数组d的长度即为最长子序列的长度；
- 代码

```
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        vector<int> path;
        for (auto num : nums) {
            if (path.empty() || num > path.back()) {
                path.push_back(num);
            } else {
                int first_larger_idx = binarySearch(path, num);
                path[first_larger_idx] = num;
            }
        }
        return path.size();
    }
}
```

```

int binarySearch(vector<int> &path, int target) {
    int res = 0;
    int left = 0, right = path.size() - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (path[mid] >= target) {
            res = mid;
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }
    return res;
};


```

## 35. 搜索插入位置

给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。如果目标值不存在于数组中，返回它将会被按顺序插入的位置。

你可以假设数组中无重复元素。

示例 1:

输入: [1,3,5,6], 5

输出: 2

示例 2:

输入: [1,3,5,6], 2

输出: 1

示例 3:

输入: [1,3,5,6], 7

输出: 4

示例 4:

输入: [1,3,5,6], 0

输出: 0

- 二分查找

- 思路：题目就是二分查找，只不过需要一点改写。返回值改成 `left`。并且在循环结束时，  
`left = mid + 1 = 1`，正好是插入位置



- 代码

```
// https://leetcode-cn.com/problems/find-first-and-last-position-of-element-in-sorted-array/solution/yi-wen-dai-ni-gao-ding-er-fen-cha-zhao-j-yml/
// 二分查找
class Solution {
public:
    int searchInsert(vector<int>& nums, int target) {
        int left = 0, right = nums.size() - 1;
        while (left <= right) {
            int mid = left + ((right - left) >> 1);
            if (nums[mid] == target) {
                return mid;
            } else if (target < nums[mid]) {
                right = mid - 1;
            } else if (nums[mid] < target) {
                left = mid + 1;
            }
        }
        return left;
    }
};
```

- 复杂度

- 时间复杂度:  $O(\log n)$ , 其中  $n$  为数组长度
- 空间复杂度:  $O(1)$ , 只需要常数空间存放若干变量

## 34. 在排序数组中查找元素的第一个和最后一个位置

给定一个按照升序排列的整数数组  $\text{nums}$ , 和一个目标值  $\text{target}$ 。找出给定目标值在数组中的开始位置和结束位置。

如果数组中不存在目标值  $\text{target}$ , 返回  $[-1, -1]$ 。

进阶:

你可以设计并实现时间复杂度为  $O(\log n)$  的算法解决此问题吗?

示例 1:

输入: nums = [5,7,7,8,8,10], target = 8

输出: [3,4]

示例 2:

输入: nums = [5,7,7,8,8,10], target = 6

输出: [-1,-1]

示例 3:

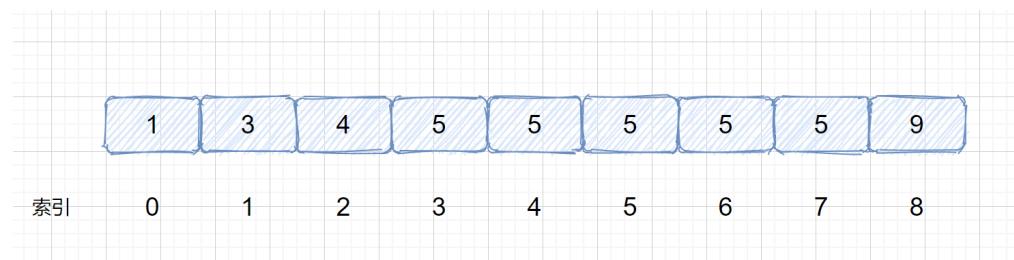
输入: nums = [], target = 0

输出: [-1,-1]

- 二分查找

- 思路

- 上面的题目讲解了如何使用二分查找在数组或区间内查找特定值的索引位置。但如果数组中存在重复元素呢？比如下图



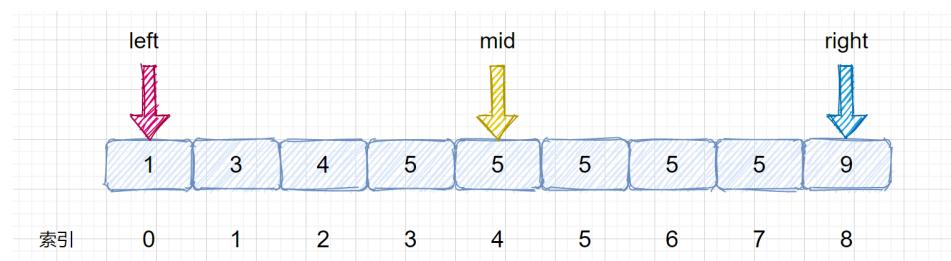
此时数组中含有多个5，使用二分查找是否含有5很容易，但若想获取第一个5和最后一个5出现的位置该怎么做呢？当然遍历可以解决，若使用二分可以吗？

- 逐个分析，先找出目标元素的下边界，那么如何寻找呢？

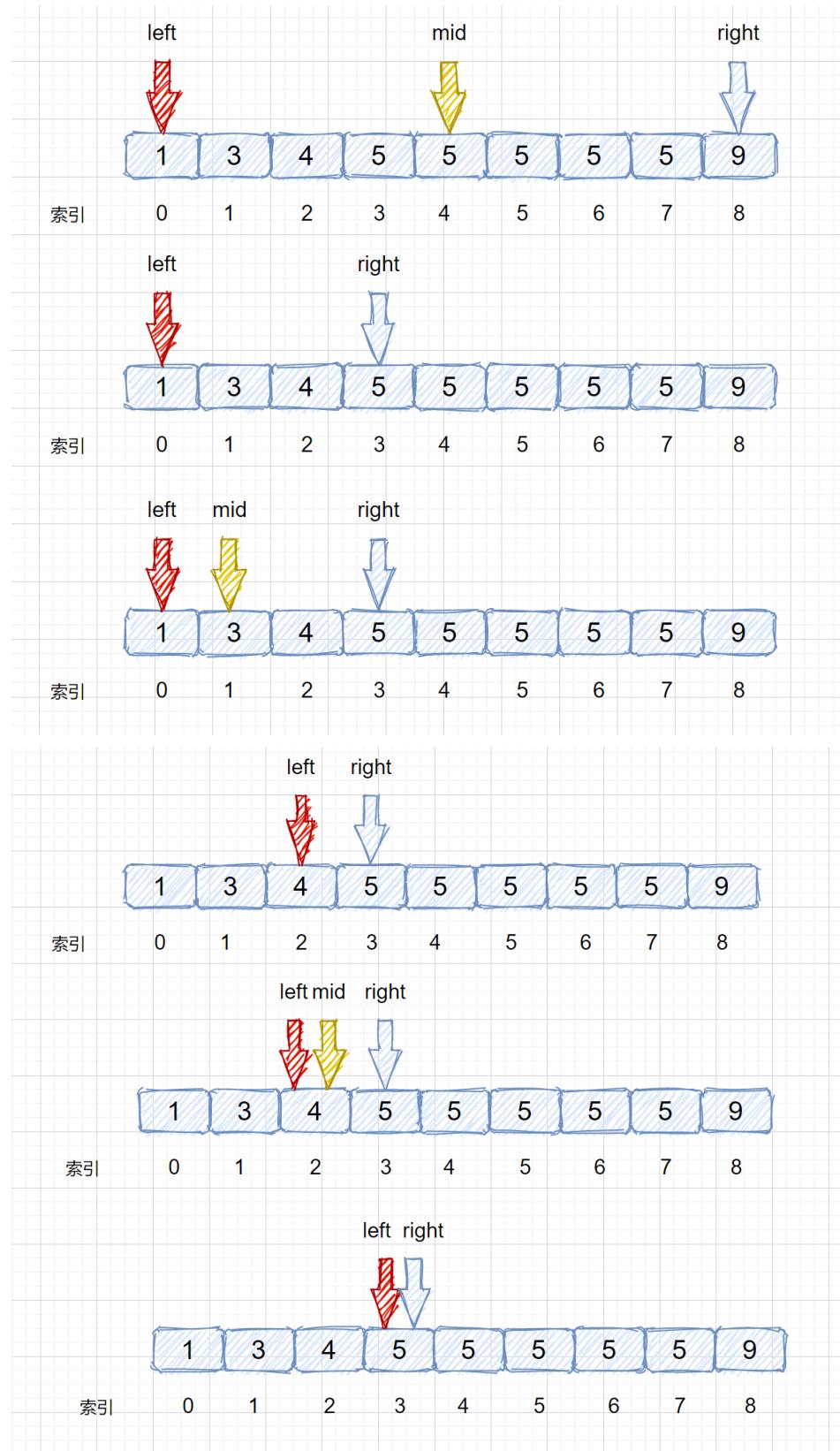
- 重点分析刚才二分查找的核心代码

```
if (nums[mid] == target) {  
    return mid;  
} else if (target < nums[mid]) { // 移动右指针，左半区间查找  
    right = mid - 1;  
} else if (nums[mid] < target) { // 移动左指针，右半区间查找  
    left = mid + 1;  
}
```

- 只需要修改上述代码即可。思考下，如果 `nums[mid] == target` 时，并不能确定当前 `mid` 点是否是目标元素的左边界，所以此时不能返回下标，比如下图



此时 `mid = 4, nums[mid] = 5`，但此时 `mid` 指向的并不是第一个5，所以需要继续查找，因为要找下边界，所以需要在左半区间内继续查找5，那么就应该在 `target <= nums[mid]` 时，让 `right = mid - 1` 即可，这样就能继续在 `mid` 左半区间内继续查找。



原理很简单，就是将**小于和等于一起处理**，当 `target <= nums[mid]` 时，移动右指针 `right = mid - 1`。需要注意的是，下边界最后的返回值是 `left`，当上图结束循环时 `left = 3, right = 2`，返回 `left` 刚好是下边界，具体执行过程如下



	1	3	4	5	5	5	5	9	
索引	0	1	2	3	4	5	6	7	8

- 计算上边界的条件正好和下边界相反即可

- 条件逻辑

- 下边界: `target <= nums[mid], right = mid - 1; target > nums[mid], left = mid + 1; return left;`
- 上边界: `target < nums[mid], right = mid - 1; target >= nums[mid], left = mid + 1; return right;`

- 代码

```
// https://leetcode-cn.com/problems/find-first-and-last-position-of-element-in-sorted-array/solution/yi-wen-dai-ni-gao-ding-er-fen-cha-zhao-j-yml/
// 二分查找
class Solution {
public:
    vector<int> searchRange(vector<int>& nums, int target) {
        int low = lowBound(nums, target);
        int upper = upperBound(nums, target);
        // 不存在的情况
        if (low > upper) {
            return vector<int> {-1, -1};
        }
        return vector<int> {low, upper};
    }

    // 计算下边界
    int lowBound(vector<int> &nums, int target) {
        int left = 0, right = nums.size() - 1;
        while (left <= right) {
            int mid = left + ((right - left) >> 1);
            if (target <= nums[mid]) {
                // 当目标值小于等于nums[mid]时，继续在左区间检索，找到第一个数
                right = mid - 1;
            } else if (nums[mid] < target) {
                // 目标值大于nums[mid]时，则在右区间继续检索，找到第一个等于目标值
                left = mid + 1;
            }
        }
        return left;
    }
}
```

```

//计算上边界
int upperBound(vector<int> &nums, int target) {
    int left = 0, right = nums.size() - 1;
    while (left <= right) {
        int mid = left + ((right - left) >> 1);
        if (target < nums[mid]) {
            right = mid - 1;
        } else if (nums[mid] <= target) {
            left = mid + 1;
        }
    }
    return right;
}

```

- 复杂度

- 时间复杂度： $O(\log n)$ ，其中  $n$  为数组长度，二分查找时间复杂度为  $O(\log n)$ ，一共执行两次
- 空间复杂度： $O(1)$ ，只需要常数空间存放若干变量

## 题目变种：找出第一个大于目标元素的元素索引

我们在上面的变种中，描述了如何找出目标元素在数组中的上下边界，然后我们下面来看一个新的变种，

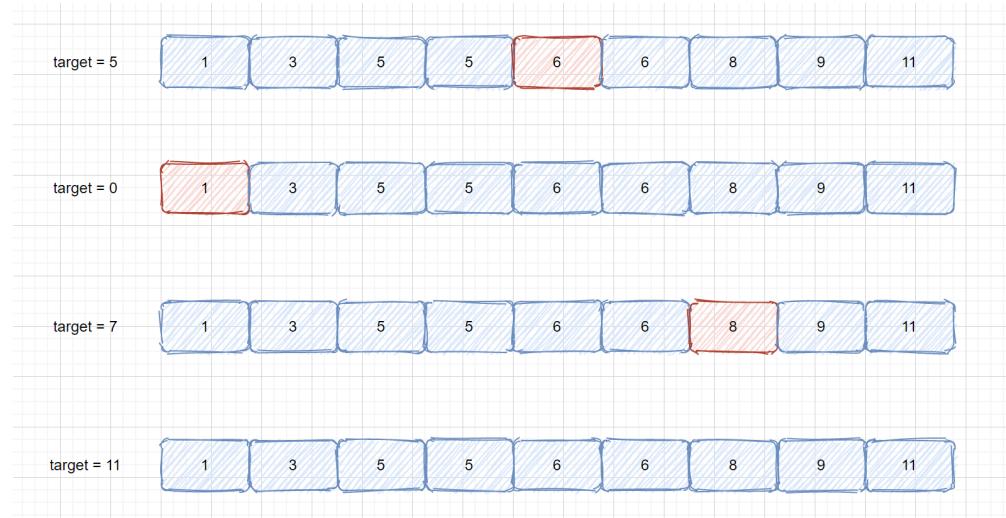
如何从数组或区间中找出第一个大于或最后一个小于目标元素的数的索引，例  $\text{nums} = \{1, 3, 5, 5, 6, 6, 8, 9, 11\}$  我们希望找出第一个大于 5 的元素的索引，那我们需要返回 4，因为 5 的后面为 6，第一个 6 的索引为 4，如果希望找出最后一个小于 6 的元素，那我们则会返回 3，因为 6 的前面为 5 最后一个 5 的索引为 3。

好啦题目我们已经了解，下面我们先来看一下如何在数组或区间中找出第一个大于目标元素的数吧。

- 二分查找

- 思路

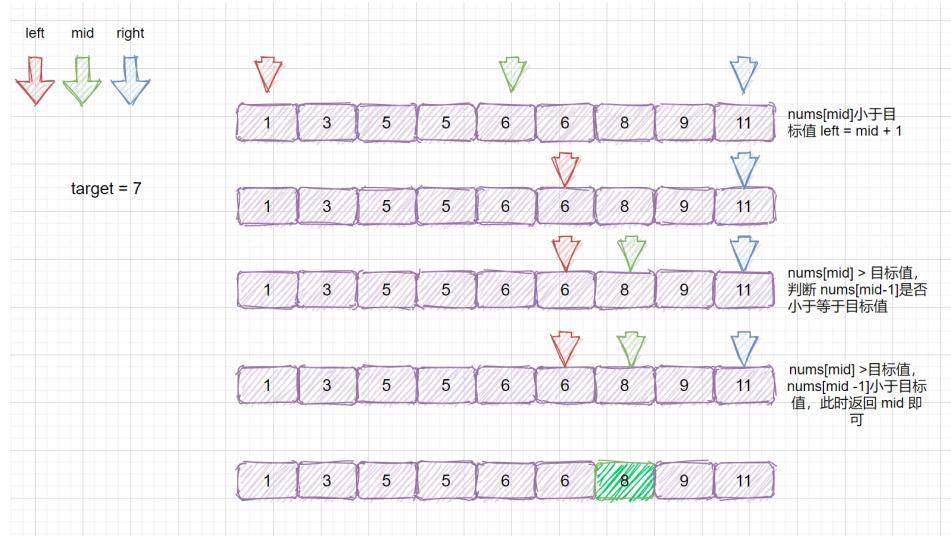
- 找出第一个大于目标元素的元素，有以下几种情况



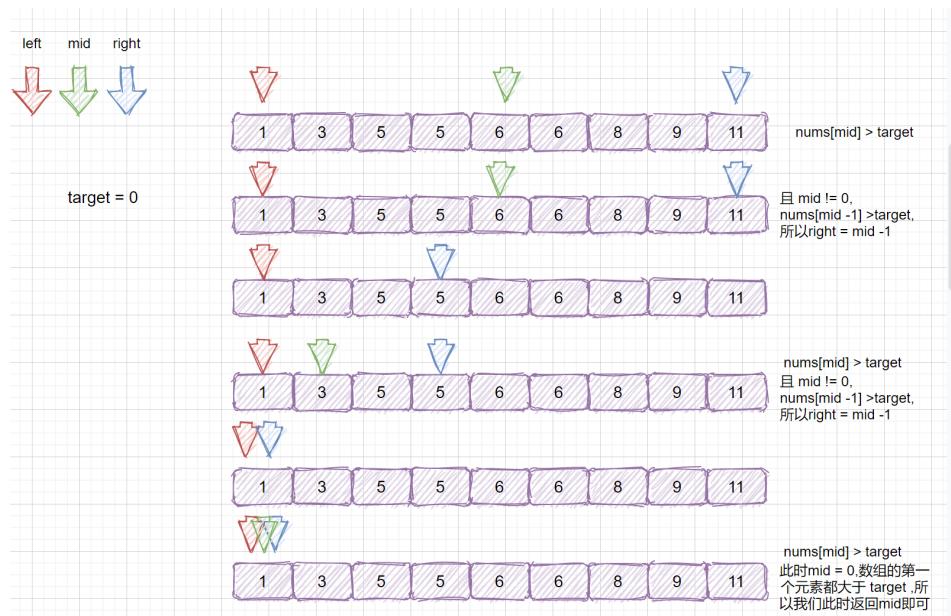
- 数组包含目标元素，找出在目标元素后的第一个元素，比如图中  $\text{target} = 5$
- 数组不包含目标元素，且数组中的所有元素都大于目标元素，此时返回数组第一个元素即可，比如图中  $\text{target} = 0$

- 数组不包含目标元素，且数组中的部分元素大于目标元素，此时需要返回第一个大于它的元素，比如图中 `target = 7`
- 数组不包含目标元素，且数组中的所有元素都小于它，此时返回 `-1` 即可，表示没有查询到，比如图中 `target = 11`
- 既然分析完所有情况，下面描述下执行过程，比如 `nums = {1, 3, 5, 5, 6, 6, 8, 9, 11}`  
`target = 7`

■ `target = 7` 的情况



■ `target = 0` 的情况



○ 代码

```
int findFirstGreaterThanOrEqualTarget(vector<int> &nums, int target) {
    int left = 0, right = nums.size() - 1;
    while (left <= right) {
        int mid = left + ((right - left) >> 1);
        // 大于目标值的情况
        if (nums[mid] <= target) {
            left = mid + 1;
        } else if (target < nums[mid]) {
            if (mid == 0 || nums[mid - 1] <= target) {
                return mid;
            } else {
                right = mid - 1;
            }
        }
    }
    return -1;
}
```

```

        }
    }
}

//所有元素都小于目标元素
return -1;
}

```

- 复杂度

- 时间复杂度:  $O(\log n)$ , 其中  $n$  为数组长度
- 空间复杂度:  $O(1)$ , 只需要常数空间存放若干变量

## 题目变种：找出最后一个小于目标元素的元素索引

通过上面的例子我们应该可以完全理解了那个变种，下面我们继续来看以下这种情况，那就是如何找到最后一个小于目标数的元素。还是上面那个例子

```
nums = {1,3,5,5,6,6,8,9,11} target = 7
```

查找最后一个小于目标数的元素，比如我们的目标数为 7，此时他前面的数为 6，最后一个 6 的索引为 5，此时我们返回 5 即可，如果目标数元素为 12，那么我们最后一个元素为 11，仍小于目标数，那么我们此时返回 8，即可。这个变种其实算是上面变种的相反情况，上面的会了，这个也完全可以搞定了

下面我们看一下代码吧。

```

int findFirstGreaterThanOrEqual(vector<int> &nums, int target) {
    int left = 0, right = nums.size() - 1;
    while (left <= right) {
        int mid = left + ((right - left) >> 1);
        //小于目标值
        if (nums[mid] < target) {
            //看看是不是当前区间的最后一位，如果当前小于，后一位大于，返回当前值即可
            if (mid == right || nums[mid + 1] >= target) {
                return mid;
            } else {
                left = mid + 1;
            }
        } else if (target < nums[mid]) {
            right = mid - 1;
        }
    }
    //没有查询到的情况
    return -1;
}

```

## 33. 搜索旋转排序数组

整数数组  $\text{nums}$  按升序排列，数组中的值互不相同。

在传递给函数之前， $\text{nums}$  在预先未知的某个下标  $k$  ( $0 \leq k < \text{nums.length}$ ) 上进行了旋转，使数组变为  $[\text{nums}[k], \text{nums}[k+1], \dots, \text{nums}[n-1], \text{nums}[0], \text{nums}[1], \dots, \text{nums}[k-1]]$  (下标从 0 开始计数)。例如， $[0,1,2,4,5,6,7]$  在下标 3 处经旋转后可能变为  $[4,5,6,7,0,1,2]$ 。

给你旋转后的数组  $\text{nums}$  和一个整数  $\text{target}$ ，如果  $\text{nums}$  中存在这个目标值  $\text{target}$ ，则返回它的下标，否则返回  $-1$ 。

示例 1：

输入: nums = [4,5,6,7,0,1,2], target = 0

输出: 4

示例 2:

输入: nums = [4,5,6,7,0,1,2], target = 3

输出: -1

示例 3:

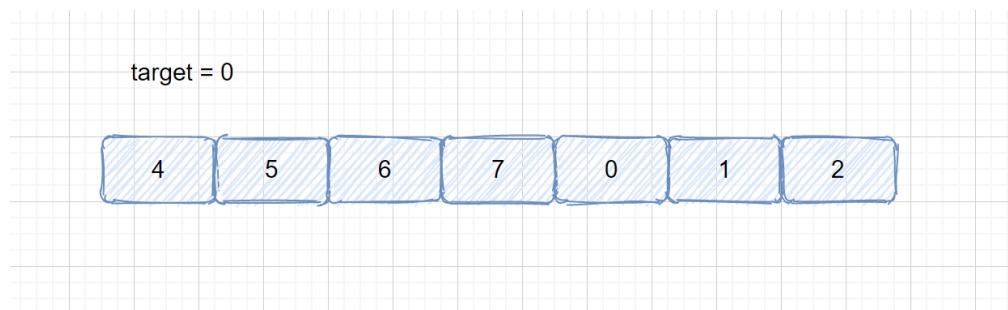
输入: nums = [1], target = 0

输出: -1

- 二分查找

- 思路

- 二分查找可在不完全有序的情况下使用。如下图，数组虽然不完全有序，但可看做是一个完全有序的数组翻折而来，或者可理解为两个有序数组，且第二个数组的最大值小第一个数组的最小值，然后拼接而来得到的一个不完全有序的数组。在这个数组中，寻找 target，找到后返回其索引，如果找不到则返回-1.



- 普通方法：直接遍历数组即可，但是是否可以使用二分查找呢？

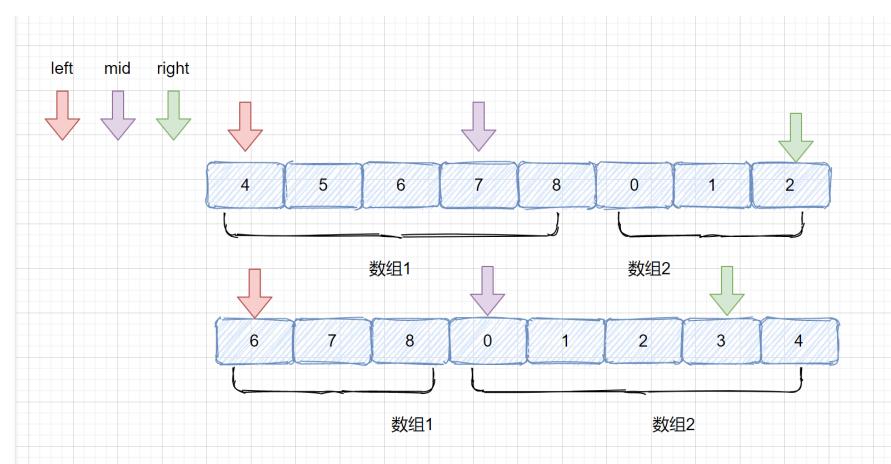
- 思路（重点）

- 定义 数组1 和 数组2；数组中第一段有序区间称为 数组1，同理第二段有序区间称为 数组2

- 首先分析 mid 点会落在哪里？存在两种情况

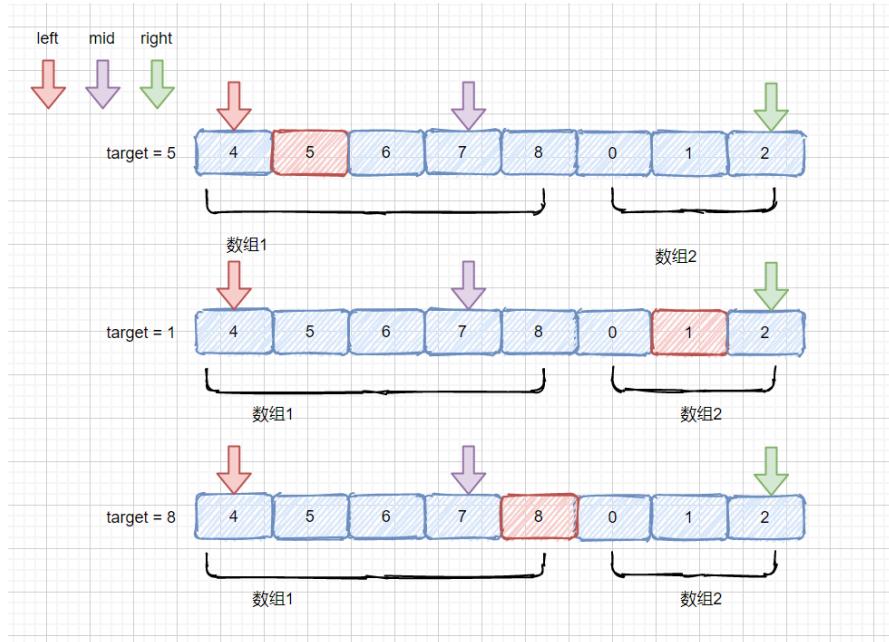
- left 点和 mid 点落在同一段，比如同时落在 数组1 或 数组2 :那么怎么判断呢？可以使用 `nums[mid]` 和 `nums[left]` 来判断，因为 mid 一定会落在 `[left, right]` 之间。则条件是 `nums[left] <= nums[mid]`

- left 点和 mid 点落在两段， left 在 数组1，而 mid 点落在 数组2，则条件是 `nums[mid] < nums[left]`

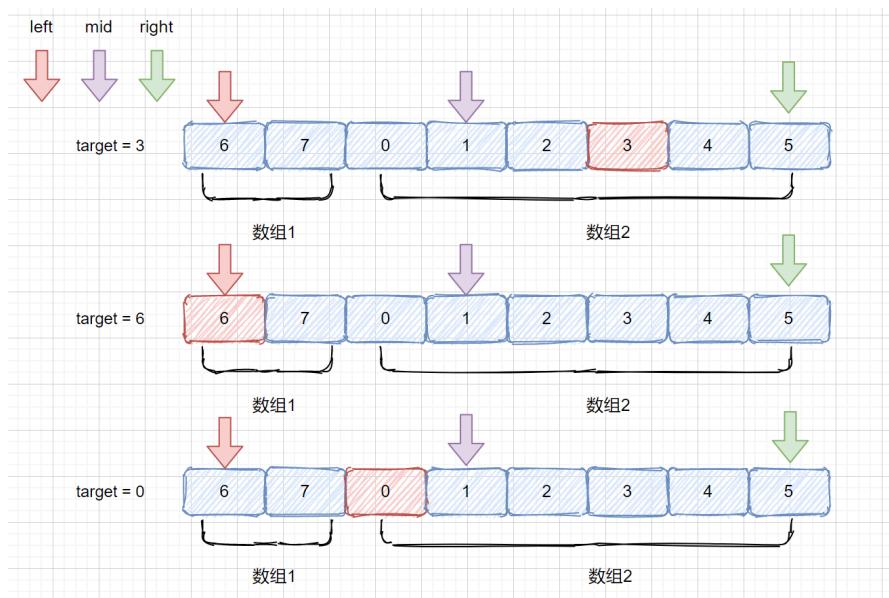


- 然后在确定了 left 点和 mid 点关系后，再寻找 mid 点和 target 间的关系，

- 首先以 left 点和 mid 点在同一数组的情况下举例分析；当然 mid 点和 target 也无非两种关系



- target 在 mid 点左边: 如上图 target=5 的情况, 此时  $4 \leq target < 7$ , 即  $nums[left] \leq target \&& target < nums[mid]$ , 此时移动 right 指针  $right = mid - 1$ , 让 left, right 都落在 数组1 中, 那么之后的查找就都在 数组1 中了, 完全有序的数组
- target 在 mid 点右边: 如上图 target = 1、target=8 的情况, 即  $target > nums[mid] \&& target < nums[left]$ , 此时移动 left 指针,  $left = mid + 1$ , 这样逐渐 left 和 right 会走到一个有序数组中
- 然后以 left 和 mid 在不同数组的情况分析, 同上, mid 点和 target 存在两种关系



- target 在 mid 点右边, 如上图 target = 3 的情况, 此时  $1 < target \leq 5$ , 即  $nums[mid] < target \&& target \leq nums[right]$ , 此时移动 left 指针  $left = mid + 1$ , 将 left, right 指针都落在 数组2 中, 那么之后的查找就在 数组2 中了, 完全有序的数组
- target 在 mid 点左边, 如上图 target = 6、target = 0 的情况, 即  $target > nums[right] \&& target < nums[mid]$ , 此时移动 right 指针,  $right = mid - 1$ , 这样 left, right 指针逐渐走到同一个有序数组中。

- 代码

```
// https://leetcode-cn.com/problems/find-first-and-last-position-of-
element-in-sorted-array/solution/yi-wen-dai-ni-gao-ding-er-fen-cha-zhao-
j-yml/
// 二分查找
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int n = nums.size();
        int left = 0, right = n - 1;
        while (left <= right) {
            int mid = left + ((right - left) >> 1);
            if (nums[mid] == target) {
                return mid;
            }
            // mid 和 left 落在同一数组: 数组1或数组2
            if (nums[mid] >= nums[left]) {
                // target 落在[left, mid)之间, 移动right, 后面就是在完全有序的
                // 区间内查找
                if (nums[left] <= target && target < nums[mid]) {
                    right = mid - 1;
                } else if (nums[mid] < target || target < nums[left]) {
                    // target落在(mid, right], 可能在数组1, 也可能是数组2; 调整left, 逐渐调整到完全
                    // 有序的区间
                    left = mid + 1;
                }
            } else if (nums[mid] < nums[left]) { // mid和left落在不同数
                // 组, left在数组1, mid在数组2
                // 有序区间, target落在(mid, right]之间, 移动left, 后面就是在
                // 完全有序的区间内查找
                if (nums[mid] < target && target <= nums[right]) {
                    left = mid + 1;
                } else if (target < nums[mid] || target > nums[right]) {
                    // target落在[left, mid), 可能在数组2, 也可能是数组1, 调整right, 逐渐调整到完全
                    // 有序的区间
                    right = mid - 1;
                }
            }
        }
        // 没有查找到
        return -1;
    }
};
```

- 复杂度

- 时间复杂度:  $O(\log n)$ , 其中  $n$  为数组长度, 二分查找时间复杂度为  $O(\log n)$ , 一  
共执行两次
- 空间复杂度:  $O(1)$ , 只需要常数空间存放若干变量

## 81. 搜索旋转排序数组 II

已知存在一个按非降序排列的整数数组  $\text{nums}$ , 数组中的值不必互不相同。

在传递给函数之前，`nums` 在预先未知的某个下标 `k` ( $0 \leq k < \text{nums.length}$ ) 上进行了旋转，使数组变为 `[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]` (下标从 0 开始计数)。例如，`[0,1,2,4,4,5,6,6,7]` 在下标 5 处经旋转后可能变为 `[4,5,6,6,7,0,1,2,4,4]`。

给你 旋转后的 数组 `nums` 和一个整数 `target`，请你编写一个函数来判断给定的目标值是否存在于数组中。如果 `nums` 中存在这个目标值 `target`，则返回 `true`，否则返回 `false`。

示例 1：

输入：`nums = [2,5,6,0,0,1,2]`, `target = 0`

输出：`true`

示例 2：

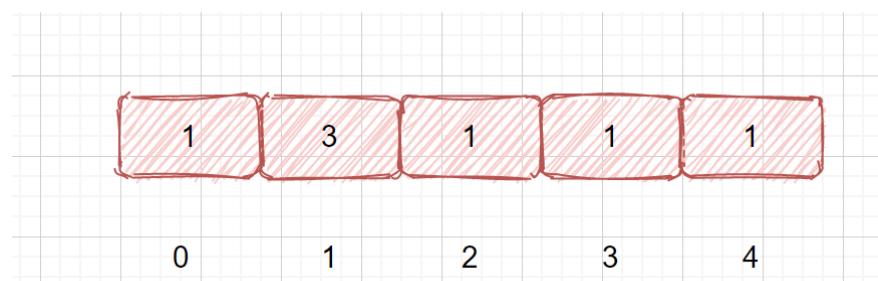
输入：`nums = [2,5,6,0,0,1,2]`, `target = 3`

输出：`false`

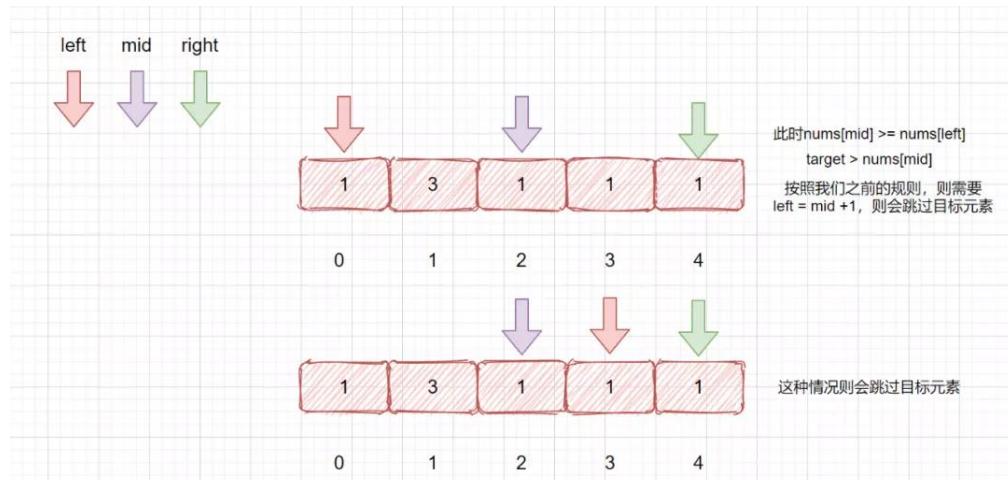
- 二分查找

- 思路

- 数组不完全有序时，不包含重复元素的问题解法已在上面分析过。那么包含重复元素的情况下该怎么做呢？见下图。



- 如上图，若继续使用上题代码，则会报错，为什么呢？现在分析下，如下图，`target=3`，若还是之前的判断条件，则会跳过 `target` 元素，导致出错。



- 所以，需要改进判断条件，将重复元素去掉。当 `nums[left] == nums[mid]` 时，让 `left++`，比如 `[1, 3, 1, 1, 1]`，此时 `nums[left] == nums[mid]`，则 `left++`。这样会错过目标值吗？并不会，只是去掉了某些重复元素，如果此时目标为 `3`，则 `left++`，然后就变成了上题中无重复元素的情况。

- 代码

```
// https://leetcode-cn.com/problems/find-first-and-last-position-of-element-in-sorted-array/solution/yi-wen-dai-ni-gao-ding-er-fen-cha-zhao-j-ymw1/
// 二分查找
```

```

class Solution {
public:
    bool search(vector<int>& nums, int target) {
        int n = nums.size();
        int left = 0, right = n - 1;
        while (left <= right) {
            int mid = left + ((right - left) >> 1);
            if (target == nums[mid]) {
                return true;
            }
            if (nums[left] == nums[mid]) {
                ++left;
                continue;
            }
            if (nums[left] < nums[mid]) {
                if (nums[left] <= target && target < nums[mid]) {
                    right = mid - 1;
                } else if (target > nums[mid] || target < nums[left]) {
                    left = mid + 1;
                }
            } else if (nums[left] > nums[mid]) {
                if (nums[mid] < target && target <= nums[right]) {
                    left = mid + 1;
                } else if (target < nums[mid] || target > nums[right]) {
                    right = mid - 1;
                }
            }
        }
        return false;
    }
};

```

- 复杂度

- 时间复杂度:  $O(n)$ , 其中  $n$  为数组长度, 最坏情况下数组元素均相等且不为  $target$ , 需要访问所有位置才能出结果
- 空间复杂度:  $O(1)$ , 只需要常数空间存放若干变量

## 153. 寻找旋转排序数组中的最小值

已知一个长度为  $n$  的数组, 预先按照升序排列, 经由 1 到  $n$  次 旋转 后, 得到输入数组。例如, 原数组  $nums = [0,1,2,4,5,6,7]$  在变化后可能得到:

若旋转 4 次, 则可以得到  $[4,5,6,7,0,1,2]$

若旋转 7 次, 则可以得到  $[0,1,2,4,5,6,7]$

注意, 数组  $[a[0], a[1], a[2], \dots, a[n-1]]$  旋转一次 的结果为数组  $[a[n-1], a[0], a[1], a[2], \dots, a[n-2]]$

。

给你一个元素值 互不相同 的数组  $nums$ , 它原来是一个升序排列的数组, 并按上述情形进行了多次旋转。请你找出并返回数组中的 最小元素 。

示例 1:

输入:  $nums = [3,4,5,1,2]$

输出: 1

解释: 原数组为  $[1,2,3,4,5]$ , 旋转 3 次得到输入数组。

示例 2:

输入: `nums = [4,5,6,7,0,1,2]`

输出: 0

解释: 原数组为 `[0,1,2,4,5,6,7]` , 旋转 4 次得到输入数组。

示例 3:

输入: `nums = [11,13,15,17]`

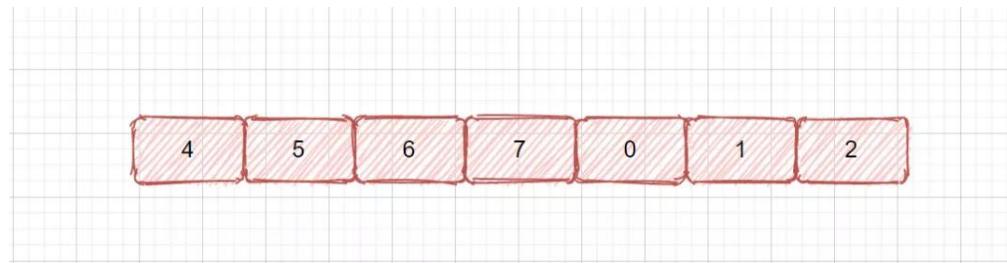
输出: 11

解释: 原数组为 `[11,13,15,17]` , 旋转 4 次得到输入数组。

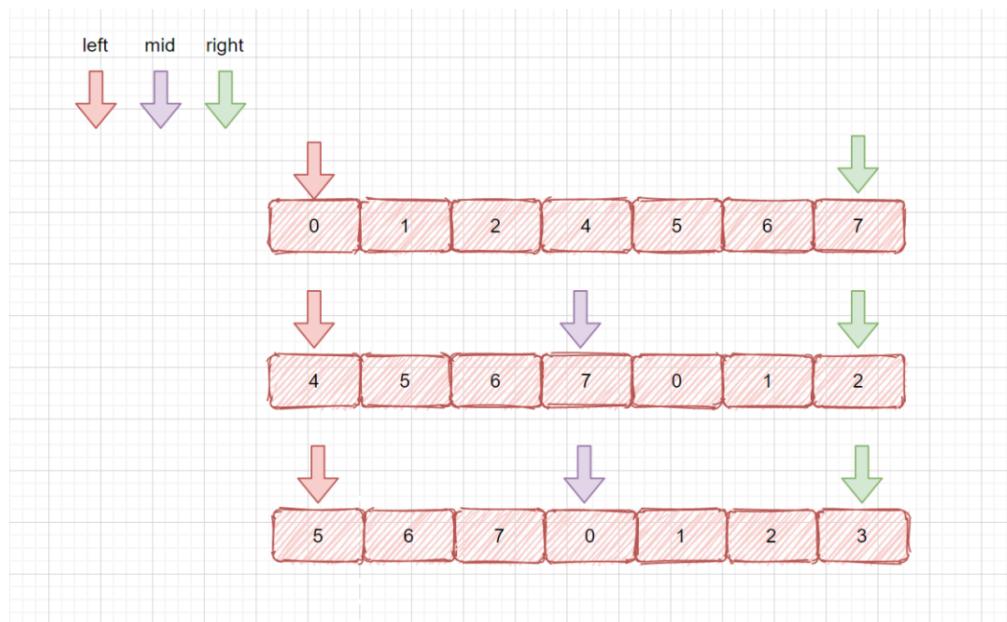
- 二分查找

- 思路

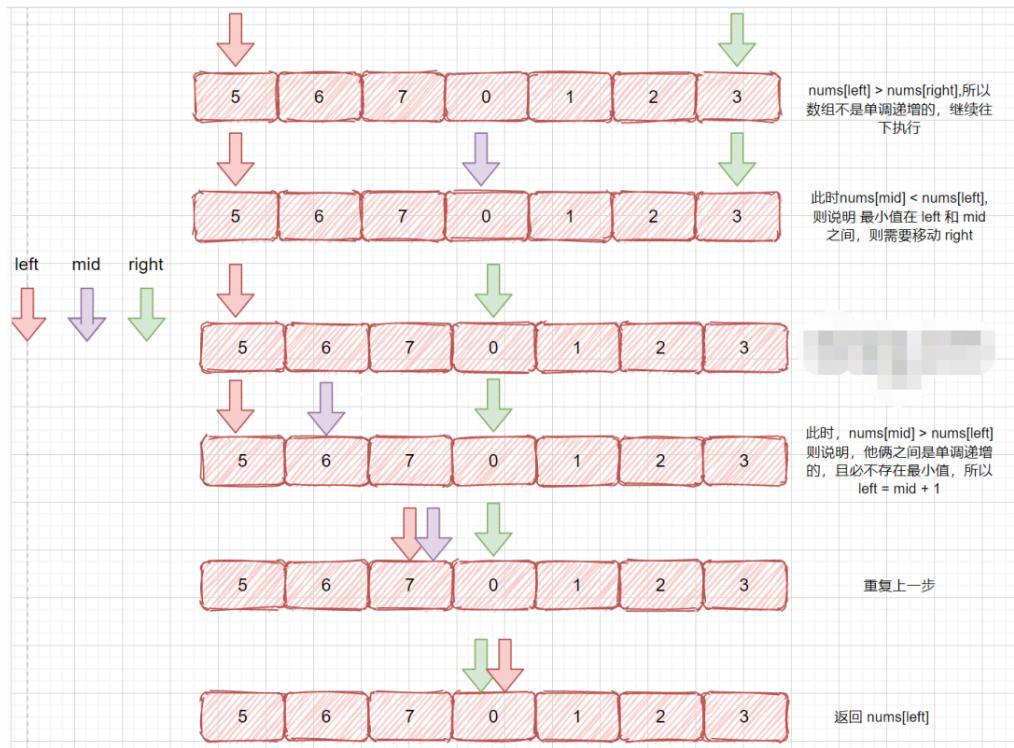
- 寻找最小值和[33. 搜索旋转排序数组](#)很相似, 只不过一个是搜索目标元素, 一个搜索最小值。搜索目标元素比较容易处理, 那么搜索最小值呢? 见下图



- 我们需要在一个旋转数组中, 查找其中的最小值, 如果数组完全有序, 那么只需要返回第一个元素即可。但此刻的数组并不完全有序, 则需要考虑以下情况



- 数组完全有序, `nums[left] < nums[right]` , 此时返回 `nums[left]` 即可
    - `left` 和 `mid` 都在前半部分, 单调递增区间内, 此时需要移动 `left = mid + 1` , 在右半部分继续查找
    - `left` 在前半部分, `mid` 在后半部分, 则最小值在 `[left, mid]` 之间, 此时需要移动 `right = mid`。注意不是 `right = mid - 1` , 这样会漏掉最小值, 因为此时的 `mid` 可能就是指向最小值, 所以 `right = mid`
    - 上面分析了搜索最小值可能出现的情况, 那么现在看下具体示例。`nums = [5,6,7,0,1,2,3]` , 具体执行过程如下



## ○ 代码

```
// https://leetcode-cn.com/problems/find-minimum-in-rotated-sorted-
array/solution/yi-wen-dai-ni-gao-ding-er-fen-cha-zhao-j-00kj/
// 二分查找
class Solution {
public:
    int findMin(vector<int>& nums) {
        int left = 0, right = nums.size() - 1;
        while (left <= right) {
            // 单调递增时直接返回
            if (nums[left] <= nums[right]) {
                return nums[left];
            }
            int mid = left + ((right - left) >> 1);
            // [left, mid]间单调递增, 说明最小值一定不在[left, mid]区间
            if (nums[left] <= nums[mid]) {
                left = mid + 1;
            } else if (nums[left] > nums[mid]) {
                // mid小于left, 说明最小值一定在[left, mid]区间, 移动right
                right = mid;
            }
        }
        return -1;
    }
};
```

## ○ 复杂度

- 时间复杂度:  $O(\log n)$ , 其中  $n$  为数组长度
- 空间复杂度:  $O(1)$ , 只需要常数空间存放若干变量

## 74. 搜索二维矩阵

编写一个高效的算法来判断  $m \times n$  矩阵中，是否存在一个目标值。该矩阵具有如下特性：

每行中的整数从左到右按升序排列。

每行的第一个整数大于前一行的最后一个整数。

示例 1：

1	3	5	7
10	11	16	20
23	30	34	60

输入： matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3

输出：true

示例 2：

1	3	5	7
10	11	16	20
23	30	34	60

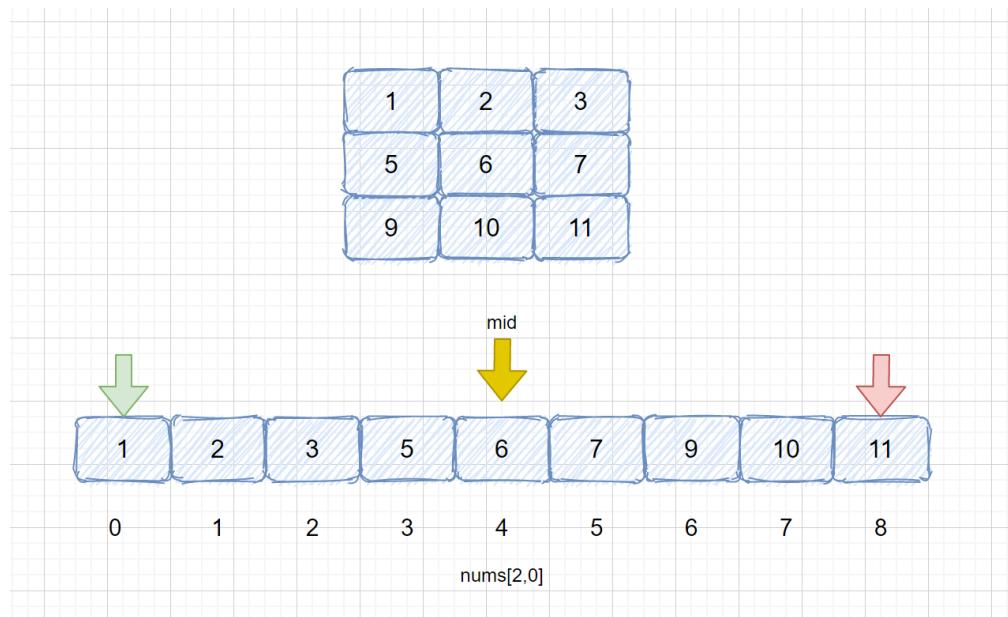
输入： matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 13

输出：false

- 二分查找

- 思路

- 我们需要从一个二维矩阵中，搜索是否存在目标元素 7，如何使用二分查找呢？其实完全可以将二维矩阵想象成一个有序的一维数组，然后用二分查找。比如如图的二维矩阵，共有 9 个元素，定义 `left = 0, right = 9 - 1 - 8`，定义和一维数组相同，然后求 `mid = left + ((right - left) >> 1) = 4`，如何将 `mid = 4` 转化到二维坐标呢？可以直接使用 `row = mid / 3 = 1, col = mid % 3 = 1`，得到了 `nums[mid] = matrix[row][col] = matrix[1][1]`



- 总之思路就是将二维矩阵假想为一个一维数组，这个数组是二维矩阵一层一层拼接得到，是完全有序的。然后定义两个指针，指向数组头部和尾部，计算 mid 再转化成二维坐标，在和 target 相比较，大于则移动 left，小于则移动 right

- 代码

```
// https://leetcode-cn.com/problems/search-a-2d-matrix/solution/yi-wen-dai-ni-gao-ding-duo-ge-er-fen-cha-2h19/
// 二分查找
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        if (matrix.size() == 0) {
            return false;
        }
        int rows = matrix.size(), cols = matrix[0].size();
        int left = 0, right = rows * cols - 1; // 行数乘列数 - 1, 右指针
        while (left <= right) {
            int mid = left + ((right - left) >> 1);
            // 将一维坐标变为二维坐标
            int row = mid / cols, col = mid % cols;
            if (matrix[row][col] == target) {
                return true;
            } else if (matrix[row][col] < target) {
                left = mid + 1;
            } else if (target < matrix[row][col]) {
                right = mid - 1;
            }
        }
        return false;
    }
};
```

- 复杂度

- 时间复杂度：\$O(\log mn)\$，其中 n 为数组长度
- 空间复杂度：\$O(1)\$，只需要常数空间存放若干变量

## 50. Pow(x, n)

实现 pow(x, n)，即计算 x 的 n 次幂函数（即， $x^n$ ）。

示例 1：

输入：x = 2.00000, n = 10

输出：1024.00000

示例 2：

输入：x = 2.10000, n = 3

输出：9.26100

示例 3：

输入：x = 2.00000, n = -2

输出：0.25000

解释： $2^{-2} = 1/2^2 = 1/4 = 0.25$

- 分治：快速幂+递归

- 思路

- 暴力：遍历  $n$  次，每次乘  $x$ ，得到  $x^n$
- 快速幂算法的本质就是分治算法。
  - 比如，要计算  $x^{64}$ ，可以按照  $x \rightarrow x^2 \rightarrow x^4 \rightarrow x^8 \rightarrow x^{16} \rightarrow x^{32} \rightarrow x^{64}$  的顺序，从  $x$  开始，每次直接把上一次的进行平方，计算 6 次就可得到  $x^{64}$  的值。而暴力需要计算 63 次  $x$
  - 再比如计算  $x^{77}$ ，按照  $x \rightarrow x^2 \rightarrow x^4 \rightarrow x^9 \rightarrow x^{19} \rightarrow x^{38} \rightarrow x^{77}$  的顺序，在  $x \rightarrow x^2, x^4 \rightarrow x^9, x^{19} \rightarrow x^{38}$  这些步骤中，直接把上一次的结果进行平方，而在  $x^4 \rightarrow x^9, x^9 \rightarrow x^{19}, x^{19} \rightarrow x^{38}$  这些步骤中，在把上一次的结果进行平方后，还需额外乘一个  $x$
  - 直接从左到右推导比较困难，因为在每一步中，不知道在将上一次的结果平方后，是否需要额外乘  $x$ ，但如果从右往左看，分治思想就十分明显了
    - 当计算  $x^n$  时，先递归计算出  $y = x^{\lfloor \frac{n}{2} \rfloor}$ ，其中  $\lfloor \cdot \rfloor$  表示对  $a$  进行下取整
    - 根据递归计算的结果，如果  $n$  为偶数，那么  $x^n = y^2$ ；若  $n$  为奇数，则  $x^n = y^2 \times x$
    - 递归的边界为  $n = 0$ ，任意数的 0 次方均为 1
    - 由于递归会使得指数减少一半，因此递归层数为  $O(\log n)$
    - 注意：当指数  $n$  为负数时，可以计算  $x^{-n}$  再取倒数得到结果，因此只需要考虑  $n$  为自然数的情况

- 代码

```
// https://leetcode-cn.com/problems/powx-n/solution/powx-n-by-leetcode-solution/
// 分治：快速幂+递归
class Solution {
public:
    double myPow(double x, int n) {
        long long N = n;
        return N >= 0 ? quickMul(x, N) : 1.0 / quickMul(x, -N);
    }

    double quickMul(double x, long long N) {
        if (N == 0) {
            return 1.0;
        }
        double y = quickMul(x, N / 2);
        return N % 2 == 0 ? y * y : y * y * x;
    }
};
```

- 复杂度

- 时间复杂度： $O(\log n)$ ，递归层数
- 空间复杂度： $O(\log n)$ ，递归层数

- 分治：快速幂+迭代

- 思路

$$\begin{aligned}
 n &= 9 \\
 &= 1001_b \\
 &= 1 \times 1 + 0 \times 2 + 0 \times 4 + 1 \times 8 \\
 &\quad (b_1 \times 2^0 + b_2 \times 2^1 + b_3 \times 2^2 + b_4 \times 2^3)
 \end{aligned}$$

$$x^n = x^9 = x^{1 \times 1} x^{0 \times 2} x^{0 \times 4} x^{1 \times 8}$$

1 蓝色数字代表  $b_i$

1 橙色数字代表  $2^{i-1}$

- 当  $x = 0.0$  时：直接返回  $0.0$ ，以避免后续操作报错。分析：数字的正数次幂恒为  $0$ ； $0$  的  $0$  次幂和负数次幂没有意义，因此直接返回  $0.0$  即可。
- 初始化 `res = 1`
- 当  $n < 0$  时：把问题转化至  $n \geq 0$  的范围内，即执行  $x = 1/x$ ,  $n = -n$ 。
- 循环计算：当  $n = 0$  时跳出。
  - 当  $n \& 1 = 1$  时：将当前  $x$  乘入  $res$ （即  $res *= x$ ）。
  - 执行  $x = x^2$ （即  $x *= x$ ）。
  - 执行  $n$  右移一位（即  $n >>= 1$ ）。
- 返回  $res$ 。

- 代码

```

// 原理https://leetcode-cn.com/problems/powx-n/solution/50-powx-n-kuai-su-mi-qing-xi-tu-jie-by-jyd/
// 代码 https://leetcode-cn.com/problems/powx-n/solution/powx-n-by-leetcode-solution/
// 分治：快速幂+迭代
class Solution {
public:
    double myPow(double x, int n) {
        long long N = n;
        return N >= 0? quickMul(x, N) : 1.0 / quickMul(x, -N);
    }

    double quickMul(double x, long long N) {
        double res = 1.0;
        while (N) {
            if (N % 2 == 1) {
                res *= x;
            }
            x *= x;
            N /= 2;
        }
        return res;
    }
};

```

- 复杂度

- 时间复杂度： $O(\log n)$
- 空间复杂度： $O(1)$

## 162. 寻找峰值

峰值元素是指其值大于左右相邻值的元素。

给你一个输入数组 `nums`, 找到峰值元素并返回其索引。数组可能包含多个峰值, 在这种情况下, 返回任何一个峰值所在位置即可。

你可以假设 `nums[-1] = nums[n] = -∞`。

示例 1:

输入: `nums = [1,2,3,1]`

输出: 2

解释: 3 是峰值元素, 你的函数应该返回其索引 2。

示例 2:

输入: `nums = [1,2,1,3,5,6,4]`

输出: 1 或 5

解释: 你的函数可以返回索引 1, 其峰值元素为 2;

或者返回索引 5, 其峰值元素为 6。

- 二分查找

- 思路

- 元素大的一侧一定会有峰值

- 我们可将 `nums` 数组看做交替的升序和降序序列, 因为`$nums[-1] = nums[n] = -\infty$`, 所以一定数组一定存在峰值
    - 在简单的二分查找中, 处理的是有序序列, 并通过减少搜索空间来查找目标元素。本题中, 对二分查找简单修改。首先从数组 `nums` 中找到中间元素 `mid`,
      - 若 `mid` 恰好位于降序序列或局部下降坡度中 `nums[mid] > nums[mid + 1]`, 说明峰值在本元素左边, 则修改 `right = mid - 1`, 继续查找左半区间
      - 若 `mid` 恰好位于升序序列或局部上升坡度中 `num[mid] < nums[mid + 1]`, 说明峰值在本元素右边, 则修改 `left = mid + 1`, 继续查找右半区间    - 注意, 因为`$nums[-1] = nums[n] = -\infty$`, 其实数组是完全有序, 那也存在峰值的哦
    - 这里代码不能使用 `while (left <= right)` 写法, 因为数组若为 `[1], [1, 2]` 这种的, 会越界

- 代码

```
// https://leetcode-cn.com/problems/find-peak-element/solution/xun-zhao-feng-zhi-by-leetcode/
// 二分查找
class Solution {
public:
    int findPeakElement(vector<int>& nums) {
        int left = 0, right = nums.size() - 1;
        while (left < right) {
            int mid = left + ((right - left) >> 1);
            if (nums[mid] > nums[mid + 1]) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left;
    }
}
```

```
};
```

- 复杂度

- 时间复杂度： $O(\log_2(n))$  每一步都将搜索空间减半。因此，总的搜索空间只需要  $\log_2(n)$  步。其中  $n$  为  $\text{nums}$  数组的长度。
- 空间复杂度： $O(1)$ 。只使用了常数空间。

## 69. x 的平方根

实现 `int sqrt(int x)` 函数。

计算并返回  $x$  的平方根，其中  $x$  是非负整数。

由于返回类型是整数，结果只保留整数的部分，小数部分将被舍去。

示例 1：

输入: 4

输出: 2

示例 2：

输入: 8

输出: 2

说明: 8 的平方根是 2.82842...,

由于返回类型是整数，小数部分将被舍去。

- 二分查找

- 思路

- 由于  $x$  平方根的整数部分  $\text{ans}$  是满足  $k^2 \leq x$  的最大  $k$  值，因此我们可以对  $k$  进行二分查找，从而得到答案。
  - 二分查找的下界为  $0$ ，上界可以粗略地设定为  $x$ 。在二分查找的每一步中，我们只需要比较中间元素  $\text{mid}$  的平方与  $x$  的大小关系，并通过比较的结果调整上下界的范围。由于我们所有的运算都是整数运算，不会存在误差，因此在得到最终的答案  $\text{ans}$  后，也就不再需要再去尝试  $\text{ans} + 1$  了。

- 代码

```
// https://leetcode-cn.com/problems/sqrtx/solution/x-de-ping-fang-gen-
by-leetcode-solution/
// 二分查找
class Solution {
public:
    int mySqrt(int x) {
        int l = 0, r = x, res = -1;
        while (l <= r) {
            int mid = l + ((r - l) >> 1);
            if ((long long) mid * mid <= x) {
                res = mid;
                l = mid + 1;
            } else {
                r = mid - 1;
            }
        }
        return res;
    }
};
```

- 复杂度
  - 时间复杂度:  $O(n!)$
  - 空间复杂度:  $O(n)$ 。只使用了常数空间。

- 牛顿迭代法: 再看吧

## 回溯

### 46. 全排列

给定一个不含重复数字的数组  $\text{nums}$ , 返回其所有可能的全排列。你可以按任意顺序返回答案。

示例 1:

输入:  $\text{nums} = [1, 2, 3]$

输出:  $[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]$

示例 2:

输入:  $\text{nums} = [0, 1]$

输出:  $[[0, 1], [1, 0]]$

示例 3:

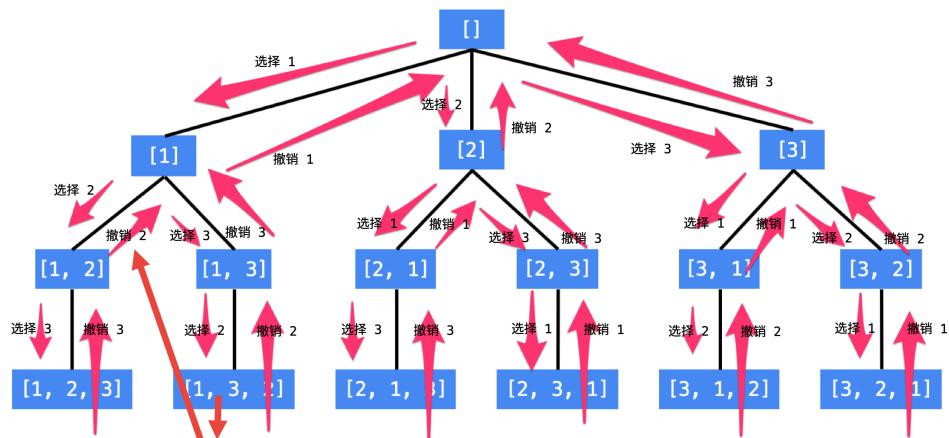
输入:  $\text{nums} = [1]$

输出:  $[[1]]$

- 回溯

- 思路

- 尝试在纸上写3个数字、4个数字、5个数字的全排列，相信可找到以下方法，以数组  $[1, 2, 3]$  的全排列为例
  - 先写以 1 开头的全排列,  $[1, 2, 3], [1, 3, 2]$ , 即  $1 + [2, 3]$  的全排列（注意，**递归结构体现在这里**）
  - 再写以 2 开头的全排列,  $[2, 1, 3], [2, 3, 1]$ , 即  $2 + [1, 3]$  的全排列
  - 最后写以 3 开头的全排列,  $[3, 1, 2], [3, 2, 1]$ , 即  $3 + [1, 2]$  的全排列
- 总结搜索方法：按顺序枚举每一位可能出现的情况，已经选择的数字不选择。按照这种策略可做到**不重不漏**。这样的思路，可使用树形结构表示



- 每个节点表示了求解全排列问题的不同阶段**，这些阶段通过遍历的**不同的值**体现，称之为**状态**
- 使用深搜有**回头**的过程，在回头以后，**状态遍历需要设置成为和先前一样**，因此回到上层节点的过程中，需要撤销上次选择，称之为**状态重置**

- 深搜，借助系统栈，保存状态遍历，在编码中只需要注意遍历到相应节点时，状态遍历的值是正确的。具体做法是：往下走一层的时候，`path` 遍历在尾部追加，而往回走的时候，需要撤销上一次的选择，也是在尾部操作，因此 `path` 变量是一个栈
- 深搜通过 **回溯**操作，实现了全局使用一份状态遍历的效果
- **设计状态变量**：以下都成为状态遍历
  - 递归终止条件 `idx == len`，表示当前程序递归到了第几层，也就是全排列中下标 `index`
  - 布尔数组 `visited`，表示某些数字是否被选择过

- 代码

```

class Solution {
public:
    vector<vector<int>> permute(vector<int>& nums) {
        vector<vector<int>> res;
        vector<int> path;
        unordered_set<int> used;

        backtrack(nums, path, res, used);

        return res;
    }

    void backtrack(vector<int> &nums, vector<int> path,
    vector<vector<int>> &res, unordered_set<int> &used) {
        // 2. 结束条件
        if (path.size() == nums.size()) {
            res.push_back(path);
            return;
        }

        // 3. 回溯遍历
        for (int i = 0; i < nums.size(); ++i) {
            if (used.find(nums[i]) != used.end()) { continue; }
            path.push_back(nums[i]);
            used.insert(nums[i]);

            backtrack(nums, path, res, used);

            path.pop_back();
            used.erase(nums[i]);
        }
    }
};

```

- 复杂度

- 时间复杂度： $O(N \times N!)$
- 空间复杂度： $O(N \times N!)$
- 回溯（优化版，不使用 `visited` 和 `path`）
  - 思路
    - 定义递归函数 `backtrack(idx, output)` 表示从左往右填到第`idx`个位置，当前排列为 `output`。那么整个递归函数分为两个情况：

- 如果  $\text{textit}\{idx\} == n$ , 说明我们已经填完了  $n$  个位置 (注意下标从  $0$  开始) , 找到了一个可行的解, 我们将  $\text{textit}\{output\}$  放入答案数组中, 递归结束。
- 如果  $\text{textit}\{idx\} < n$ , 我们要考虑这第  $\text{textit}\{idx\}$  个位置我们要填哪个数。根据题目要求我们肯定不能填已经填过的数, 因此很容易想到的一个处理手段是我们定义一个标记数组  $\text{textit}\{visited\}[]$  来标记已经填过的数, 那么在填第  $\text{textit}\{idx\}$  个数的时候我们遍历题目给定的  $n$  个数, 如果这个数没有被标记过, 我们就尝试填入, 并将其标记, 继续尝试填下一个位置, 即调用函数  $\text{backtrack}(idx + 1, output)$ 。回溯的时候要撤销这一个位置填的数以及标记, 并继续尝试其他没被标记过的数。
- 使用标记数组来处理填过的数是一个很直观的思路, 但是可不可以去掉这个标记数组呢? 毕竟标记数组也增加了我们算法的空间复杂度。
  - 答案是可以的, 我们可以将题目给定的  $n$  个数的数组  $\text{textit}\{nums\}$  划分成左右两个部分, 左边的表示已经填过的数, 右边表示待填的数, 我们在回溯的时候只要动态维护这个数组即可。
  - 具体来说, 假设我们已经填到第  $\text{textit}\{idx\}$  个位置, 那么  $\text{textit}\{nums\}$  数组中  $[0, \text{textit}\{idx\}-1]$  是已填过的数的集合,  $[\text{textit}\{idx\}, n-1]$  是待填的数的集合。我们肯定是尝试用  $i$  里的数  $[\text{textit}\{idx\}, n-1]$  去填第  $\text{textit}\{idx\}$  个数, 假设待填的数的下标为  $i$ , 那么填完以后我们将第  $i$  个数和第  $\text{textit}\{idx\}$  个数交换, 即能使得在填第  $\text{textit}\{idx\}+1$  个数的时候  $\text{textit}\{nums\}$  数组的  $[0, first]$  部分为已填过的数,  $[\text{textit}\{idx\}+1, n-1]$  为待填的数, 回溯的时候交换回来即能完成撤销操作。
  - 举个简单的例子, 假设我们有  $[2, 5, 8, 9, 10]$  这 5 个数要填入, 已经填到第 3 个位置, 已经填了  $[8, 9]$  两个数, 那么这个数组目前为  $[8, 9 | 2, 5, 10]$  这样的状态, 分隔符区分了左右两个部分。假设这个位置我们要填  $10$  这个数, 为了维护数组, 我们将  $2$  和  $10$  交换, 即能使得数组继续保持分隔符左边的数已经填过, 右边的待填  $[8, 9, 10 | 2, 5]$
  - 当然善于思考的读者肯定已经发现这样生成的全排列并不是按字典序存储在答案数组中的, 如果题目要求按字典序输出, 那么请还是用标记数组或者其他方法。

- 代码

```
// https://leetcode-cn.com/problems/permute/solution/quan-pai-lie-by-leetcode-solution-2/
// 回溯优化版, 不使用visited和path
class Solution {
public:
    vector<vector<int>> permute(vector<int>& nums) {
        int len = nums.size();
        vector<vector<int>> res;
        backtrack(res, nums, 0, len);
        return res;
    }
    void backtrack(vector<vector<int>> &res, vector<int> &output, int idx, int len) {
        if (idx == len) {
            res.push_back(output);
            return;
        }
        for (int i = idx; i < len; ++i) {
            swap(output[i], output[idx]);
            backtrack(res, output, idx + 1, len);
            swap(output[i], output[idx]);
        }
    }
}
```

```
    }  
};
```

- 复杂度
  - 时间复杂度:  $O(N \times N!)$
  - 空间复杂度:  $O(N \times N!)$

## 栈

### 20. 有效的括号

给定一个只包括 `'('`, `')'`, `'{'`, `'}'`, `'['`, `']'` 的字符串 `s`，判断字符串是否有效。

有效字符串需满足：

1. 左括号必须用相同类型的右括号闭合。
2. 左括号必须以正确的顺序闭合。
3. 每个右括号都有一个对应的相同类型的左括号。

#### 示例 1：

```
输入: s = "()"  
输出: true
```

#### 示例 2：

```
输入: s = "[]{}"  
输出: true
```

#### 示例 3：

```
输入: s = "()"  
输出: false
```

- 思路：栈
- 代码

```
class Solution {  
public:  
    bool isValid(string s) {  
        stack<char> stk;  
  
        for (int i = 0; i < s.size(); ++i) {  
            if (s[i] == '(' || s[i] == '{' || s[i] == '[') {  
                stk.push(s[i]);  
            } else {  
                if (stk.empty() ||  
                    (s[i] == ')' && stk.top() != '(') ||  
                    (s[i] == '}' && stk.top() != '{') ||  
                    (s[i] == ']' && stk.top() != '[')) {  
                    return false;  
                }  
                stk.pop();  
            }  
        }  
        return stk.empty();  
    }  
};
```

```
    }
}
return stk.empty();
};

};
```

## 其他

### 146. LRU 缓存机制

运用你所掌握的数据结构，设计和实现一个 LRU (最近最少使用) 缓存机制。

实现 LRUCache 类：

LRUCache(int capacity) 以正整数作为容量 capacity 初始化 LRU 缓存

int get(int key) 如果关键字 key 存在于缓存中，则返回关键字的值，否则返回 -1。

void put(int key, int value) 如果关键字已经存在，则变更其数据值；如果关键字不存在，则插入该组「关键字-值」。当缓存容量达到上限时，它应该在写入新数据之前删除最久未使用的数据值，从而为新的数据值留出空间。

进阶：你是否可以在 O(1) 时间复杂度内完成这两种操作？

示例：

输入

```
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "put", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
```

输出

```
[null, null, null, 1, null, -1, null, -1, 3, 4]
```

解释

```
LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // 缓存是 {1=1}
lRUCache.put(2, 2); // 缓存是 {1=1, 2=2}
lRUCache.get(1); // 返回 1
lRUCache.put(3, 3); // 该操作会使得关键字 2 作废，缓存是 {1=1, 3=3}
lRUCache.get(2); // 返回 -1 (未找到)
lRUCache.put(4, 4); // 该操作会使得关键字 1 作废，缓存是 {4=4, 3=3}
lRUCache.get(1); // 返回 -1 (未找到)
lRUCache.get(3); // 返回 3
lRUCache.get(4); // 返回 4
```

- 哈希表+双向链表

- 思路

- LRU 缓存机制可以通过哈希表辅以双向链表实现，我们用一个哈希表和一个双向链表维护所有在缓存中的键值对。
      - 双向链表按照被使用的顺序存储了这些键值对，靠近头部的键值对是最近使用的，而靠近尾部的键值对是最久未使用的。
      - 哈希表即为普通的哈希映射（HashMap），通过缓存数据的键映射到其在双向链表中的位置。
      - 这样以来，我们首先使用哈希表进行定位，找出缓存项在双向链表中的位置，随后将其移动到双向链表的头部，即可在 \$O(1)\$ 的时间内完成 `get` 或者 `put` 操作。具体的方法如下

- 对于 `get` 操作，首先判断 `key` 是否存在
  - 如果 `key` 不存在，则返回 `-1`
  - 如果 `key` 存在，则 `key` 对应的节点时最近被使用的节点。通过哈希表定位到该节点在双向链表中的位置，并将其移动到双向链表的头部，最后返回该节点的值。
- 对于 `put` 操作，首先判断 `key` 是否存在：
  - 如果 `key` 不存在，使用 `key` 和 `value` 创建一个新的节点，在双向链表的头部添加该节点，并将 `key` 和该节点添加进哈希表中。然后判断双向链表的节点数是否超出容量，如果超出容量，则删除双向链表的尾部节点，并删除哈希表中对应的项；
  - 如果 `key` 存在，则与 `get` 操作类似，先通过哈希表定位，再将对应的节点的值更新为 `value`，并将该节点移到双向链表的头部。
- 上述各项操作中，访问哈希表的时间复杂度为  $O(1)$ ，在双向链表的头部添加节点、在双向链表的尾部删除节点的复杂度也为  $O(1)$ 。而将一个节点移到双向链表的头部，可以分成「删除该节点」和「在双向链表的头部添加节点」两步操作，都可以在  $O(1)$  时间内完成
- 在双向链表的实现中，使用一个 `伪头部 (dummy head)` 和 `伪尾部 (dummy tail)` 标记界限，这样在添加节点和删除节点的时候就不需要检查相邻的节点是否存在。

- 代码

```

struct DLinkedNode {
    int key, value;
    DLinkedNode* prev;
    DLinkedNode* next;
    DLinkedNode(): key(0), value(0), prev(nullptr), next(nullptr) {}
    DLinkedNode(int _key, int _value): key(_key), value(_value),
    prev(nullptr), next(nullptr) {}
};

class LRUCache {
private:
    unordered_map<int, DLinkedNode*> cache;
    DLinkedNode* head;
    DLinkedNode* tail;
    int size;
    int capacity;

public:
    LRUCache(int _capacity): capacity(_capacity), size(0) {
        // 使用伪头部和伪尾部节点
        head = new DLinkedNode();
        tail = new DLinkedNode();
        head->next = tail;
        tail->prev = head;
    }

    int get(int key) {
        if (!cache.count(key)) {
            return -1;
        }
        // 如果 key 存在，先通过哈希表定位，再移到头部
        DLinkedNode* node = cache[key];
        moveToHead(node);
    }

    void put(int key, int value) {
        if (!cache.count(key)) {
            if (size < capacity) {
                DLinkedNode* node = new DLinkedNode(key, value);
                cache[key] = node;
                moveT

```

```

        return node->value;
    }

    void put(int key, int value) {
        if (!cache.count(key)) {
            // 如果 key 不存在, 创建一个新的节点
            DLinkedNode* node = new DLinkedNode(key, value);
            // 添加进哈希表
            cache[key] = node;
            // 添加至双向链表的头部
            addToHead(node);
            ++size;
            if (size > capacity) {
                // 如果超出容量, 删除双向链表的尾部节点
                DLinkedNode* removed = removeTail();
                // 删除哈希表中对应的项
                cache.erase(removed->key);
                // 防止内存泄漏
                delete removed;
                --size;
            }
        }
        else {
            // 如果 key 存在, 先通过哈希定位, 再修改 value, 并移到头部
            DLinkedNode* node = cache[key];
            node->value = value;
            moveToHead(node);
        }
    }

    void addToHead(DLinkedNode* node) {
        node->prev = head;
        node->next = head->next;
        head->next->prev = node;
        head->next = node;
    }

    void removeNode(DLinkedNode* node) {
        node->prev->next = node->next;
        node->next->prev = node->prev;
    }

    void moveToHead(DLinkedNode* node) {
        removeNode(node);
        addToHead(node);
    }

    DLinkedNode* removeTail() {
        DLinkedNode* node = tail->prev;
        removeNode(node);
        return node;
    }
};

```

- 复杂度

- 时间复杂度：对于 `put` 和 `get` 都是  $O(1)$ 。

- 空间复杂度： $O(\text{capacity})$ ，因为哈希表和双向链表最多存储  $\text{capacity} + 1$  个元素。