2.1. 二分法题目汇总

- (0704) binary-search
  - 升序数组中查找目标值
  - 输入：nums = [−1,0,3,5,9,12], target = 9 输出: 4
  - 数组 target 定位,[left = 0,right = n-1, left <= right],二分查找
  - 时间复杂度：O(log(n))， 空间复杂度：O(1)

```rust
impl Solution {
    pub fn search(nums: Vec<i32>, target: i32) -> i32 {
        let (mut left, mut right) = (0_i32, nums.len() as i32);
        while left < right {
            let mid = left + (right - left) / 2;
            match nums[mid as usize].cmp(&target) {
                std::cmp::Ordering::Less => left = mid + 1,
                std::cmp::Ordering::Greater => right = mid,
                std::cmp::Ordering::Equal => return mid,
            }
        }
        -1
    }
}
```

- (0035) search-insert-position
  - 升序数组中查找目标值，若不存在则返回插入位置
  - 输入: nums = [1,3,5,6], target = 2 输出: 1
  - 二分查找的变种,返回 right 位置
  - 时间复杂度：O(log(n))， 空间复杂度：O(n)

```rust
impl Solution {
    pub fn search_insert(nums: Vec<i32>, target: i32) -> i32 {
        let (mut left, mut right) = (0_i32, nums.len() as i32);
        while left < right {
            let mid = left + (right - left) / 2;
            match nums[mid as usize].cmp(&target) {
                std::cmp::Ordering::Less => left = mid + 1,
                std::cmp::Ordering::Greater => right = mid,
                std::cmp::Ordering::Equal => return mid,
            }
        }
        right
    }
}
```

- (0744) find-smallest-letter-greater-than-target
  - 查找非降序数组中比目标字母大的最小字母
  - 输入: letters = [“c”,"f","j"], target = “c” 输出: “f”
  - 求解字母的最小距离，二分查找
  - 时间复杂度：O(log(n))， 空间复杂度：O(1)

```rust
impl Solution {
    pub fn next_greatest_letter(letters: Vec<char>, target: char) -> char {
        // pub fn binary_search_by<F>(&self, f: F) -> Result<usize, usize>
        // where F: FnMut(&T) -> Ordering,
        // f: 一个闭包，它接受一个引用到切片元素类型 &T，并返回 std::cmp::Ordering
        match letters.binary_search_by(|&c| {
            if c <= target {std::cmp::Ordering::Less}
            else {std::cmp::Ordering::Greater}
        }) {
            Ok(i) | Err(i) if i < letters.len() => letters[i],
            _ => letters[0],
        }
    }
}
```

- (1351) count-negative-numbers-in-a-sorted-matrix
  - 统计非严格递减顺序排列矩阵（二维数组）中的负数的个数
  - 输入：grid = [[4,3,2,-1],[3,2,1,-1],[1,1,-1,-2],[−1,-1,-2,-3]] 输出：8
  - 线性(遍历)+二分查找返回 0 的插入位置
  - 时间复杂度：O(mlog(n))，空间复杂度：O(1)

```
use std::cmp::Ordering;
impl Solution {
    pub fn count_negatives(grid: Vec<Vec<i32>>) -> i32 {
        let (mut total_cnt, mut m, mut n) = (0_i32, grid.len(), grid[0].len());
        for i in 0..m as usize {
            if (grid[i][0] < 0) {
                total_cnt += n as i32;
                continue;
            }
            let (mut left, mut right) = (0_usize, n as usize);
            while (left < right) {
                let mid = left + (right - left) / 2;
                match grid[i][mid].cmp(&0) {
                    Ordering::Less => right = mid,
                    Ordering::Greater => left = mid + 1,
                    Ordering::Equal => left = mid + 1,
                }
            }
            total_cnt += n as i32 - right as i32;
        }
        total_cnt
    }
}
```

- (0034) find-first-and-last-position-of-element-in-sorted-array
  - 确定非降序数组的目标值的第一个和最后一个位置
  - 输入：nums = [5,7,7,8,8,10], target = 8 输出：[3,4]
  - 利用两次二分查找，分别找到左右边界，针对相等情况的 mid 赋值处理
  - 时间复杂度：O(log(n))，空间复杂度：O(1)

```
use std::cmp::Ordering;
impl Solution {
    pub fn search_range(nums: Vec<i32>, target: i32) -> Vec<i32> {
        if nums.is_empty() {
            return vec![-1, -1];
        }
        let left_border = Self::binary_search(&nums, target, true);
        if left_border == nums.len() as i32 || nums[left_border as usize] != target {
            return vec![-1, -1];
        }
        let right_border = Self::binary_search(&nums, target, false) - 1;
        vec![left_border, right_border]
    }
    fn binary_search(nums: &[i32], target: i32, left_biased: bool) -> i32 {
        let (mut low, mut high) = (0_i32, nums.len() as i32);
        // equal 分支是互斥
        while low < high {
            let mid = low + (high - low) / 2;
            match nums[mid as usize].cmp(&target) {
                Ordering::Less => low = mid + 1,
                Ordering::Greater => high = mid,
                Ordering::Equal if left_biased => high = mid,
                Ordering::Equal => low = mid + 1,
            }
        }
        low
    }
}
```

- (0436) find-right-interval
  - 寻找右区间，即找到每个区间的右边界，例如[1,2]的右边界是[2,3]
  - 输入：intervals = [[3,4],[2,3],[1,2]] 输出：[−1,0,1]
  - 输入：intervals = [[1,2]] 输出：[−1]
  - 暴力遍历/先进行排序+二分查找
  - 时间复杂度：O(nlog(n))，空间复杂度：O(n)

```rust
use std::cmp::Ordering;
impl Solution {
    pub fn find_right_interval(intervals: Vec<Vec<i32>>) -> Vec<i32> {
        let n = intervals.len();
        let mut sorted_intervals: Vec<(i32, i32, usize)> = intervals
            .into_iter() // 创建一个迭代器
            .enumerate() // 将迭代器转换为产生 (index, value) 对的迭代器
            .map(|(i, v)| (v[0], v[1], i)) // 对迭代器中的每个元素应用一个函数
            .collect(); // 将迭代器转换为一个集合
        // 根据每个子数组的第一个元素进行排序，采用 unstable，加快速度
        sorted_intervals.sort_unstable_by_key(|&(start, _, _)| start);
        // 建立结果数组
        let mut result = vec![-1; n];
        // 引用使用
        // 1. 避免所有权转移：通过使用 &sorted_intervals 而不是直接使用 sorted_intervals，
        //    我们避免了将 sorted_intervals 的所有权转移到 for 循环中。这样，我们可以在循环后继续使用
        //    sorted_intervals，如果需要的话。
        //    使用引用允许我们在不转移所有权的情况下使用数据，这在需要多次访问同一数据的场景中特别有用
        // 2. 提高效率：使用引用可以避免不必要的数据复制。如果我们不使用引用，Rust 会尝试复制 sorted_intervals 中
        //    的每个元素，这对于大型数据结构来说可能会非常耗时和内存密集。
        // 3. 模式匹配：在 for &(_, end, original_index) in &sorted_intervals 中，我们使用 & 来解构引用。
        //    这允许我们直接访问元组中的各个字段，而不需要通过引用操作符来访问。
        // 4. 不可变借用：使用 & 创建了对 sorted_intervals 的不可变借用。这保证了在遍历过程中，sorted_intervals
        //    不会被修改，这对于维护数据的一致性很重要。
        // 5. 与二分查找函数的一致性：注意 binary_search 函数接受的是 &[(i32, i32, usize)] 类型的参数。
        //    通过在循环中使用 &sorted_intervals，我们确保了类型的一致性，可以直接将其传递给 binary_search 函数。
        for &(_, end, original_index) in &sorted_intervals {
            // 针对第二个元素进行二分搜索
            let right_index = Solution::binary_search(&sorted_intervals, end);
            if right_index < n as usize {
                result[original_index] = sorted_intervals[right_index].2 as i32;
            }
        }
        result
    }

    fn binary_search(intervals: &[(i32, i32, usize)], target: i32) -> usize {
        let (mut low, mut high) = (0_usize, intervals.len() as usize);
        while low < high {
            let mid = low + (high - low) / 2;
            match intervals[mid].0.cmp(&target) {
                Ordering::Less => low = mid + 1,
                Ordering::Greater | Ordering::Equal => high = mid,
            }
        }
        low
    }
}
```

- (0981) time-based-key-value-store
  - 基于时间的键值存储，基于键返回最近时间戳的值
  - 输入：["TimeMap", "set", "get", "get", "set", "get", "get"]
    [[], ["foo", "bar", 1], ["foo", 1], ["foo", 3], ["foo", "bar2", 4], ["foo", 4], ["foo", 5]]
  - 输出：[null, null, "bar", "bar", null, "bar2", "bar2"]
  - 二分查找(时间戳二分查找)+哈希表(键+(时间戳,值))
  - 时间复杂度：O(log(n))，空间复杂度：O(n)

```rust
use std::collections::HashMap;

struct TimeMap {
    map: HashMap<String, Vec<(i32, String)>>,
} // 哈希表键为字符串，值为元组的数组（时间戳，字符串）

impl TimeMap {
    fn new() -> Self {
        TimeMap {
            map: HashMap::new(),
        }
    }

    fn set(&mut self, key: String, value: String, timestamp: i32) {
        self.map
            .entry(key) // entry(): 返回一个 Entry 枚举，允许我们在一个操作中插入或修改值
            .or_insert_with(Vec::new) // or_insert_with(): 如果键不存在，则使用提供的闭包创建新值
            .push((timestamp, value)); // push(): 将新的 (timestamp, value) 元组添加到 Vec 的末尾
    }

    fn get(&self, key: String, timestamp: i32) -> String {
        // get(): 尝试获取与键关联的值的引用。返回 Option<&V>
        if let Some(pairs) = self.map.get(&key) {
            // binary_search_by_key(): 使用闭包从元素中提取用于比较的键来执行二分查找
            // &timestamp: 要查找的目标值
            // |&(t, _)| t: 闭包，从每个元素 &(t, _) 中提取时间戳 t 作为比较键
            match pairs.binary_search_by_key(&timestamp, |&(t, _)| t) {
                // 如果找到精确匹配，Ok(i) 返回匹配元素的索引
                Ok(i) => pairs[i].1.clone(),
                // 如果没有找到精确匹配，Err(i) 返回可以插入元素以保持排序的索引
                // i > 0 确保我们不会在空列表上尝试访问 [i - 1]
                Err(i) if i > 0 => pairs[i - 1].1.clone(),
                // 处理列表为空或时间戳小于所有现有时间戳的情况
                _ => String::new(),
            }
        } else {
            String::new()
        }
    }
}
```

- (1146) snapshot-array
  - 快照数组包含建立，设置，快照，检索操作
  - 输入：["SnapshotArray","set","snap","set","get"] [[3],[0,5],[],[0,6],[0,0]]
  - 输出：[null,null,0,null,5]
  - 基于数组存储 BTreeMap 快照/快照数组的二分查找
  - 时间复杂度：O(log(n))，空间复杂度：O(mn)

```rust
struct SnapshotArray {
    snap_cnt: i32,  // 当前快照的计数器
    data: Vec<Vec<(i32, i32)>>,  // 存储数组数据的二维向量，每个元素是（快照 ID，值）的元组
}

impl SnapshotArray {
    fn new(length: i32) -> Self {
        SnapshotArray {
            snap_cnt: 0,
            // 创建一个长度为 length 的向量，每个元素都是一个空的 Vec，但预分配了容量 1
            data: vec![Vec::with_capacity(1); length as usize],
        }
    }

    fn set(&mut self, index: i32, val: i32) {
        let index = index as usize;
        // Some 是 Rust 中 Option 枚举的一个变体，用于表示可能存在的值
        // last() 返回 Option<&T>，如果向量非空，它返回 Some(&T)，否则返回 None
        if let Some(&(snap, _)) = self.data[index].last() {
            // 如果最后一个元素的快照 ID 等于当前快照 ID，我们需要更新它而不是添加新元素
            if snap == self.snap_cnt {
                self.data[index].pop();  // 移除最后一个元素
            }
        }
        // 添加新的（快照 ID，值）元组到指定索引的向量中
        self.data[index].push((self.snap_cnt, val));
    }

    fn snap(&mut self) -> i32 {
        let current_snap = self.snap_cnt;
        self.snap_cnt += 1;  // 增加快照计数器
        current_snap  // 返回当前快照 ID
    }

    fn get(&self, index: i32, snap_id: i32) -> i32 {
        let index = index as usize;
        // 使用二分查找在指定索引的向量中查找快照 ID
        match self.data[index].binary_search_by_key(&snap_id, |&(snap, _)| snap) {
            Ok(i) => self.data[index][i].1,  // 如果找到精确匹配，返回对应的值
            Err(i) if i > 0 => self.data[index][i - 1].1,  // 如果没有精确匹配，返回前一个快照的值
            _ => 0,  // 如果索引为 0 或向量为空，返回默认值 0
        }
    }
}
```

- (0033) search-in-rotated-sorted-array
  - 搜索旋转后数组的 target
  - 输入：nums = [4,5,6,7,0,1,2], target = 0 输出：4
  - 二分查找+判断左右有序区间
  - 时间复杂度：O(log(n))，空间复杂度：O(1)

```rust
impl Solution {
    pub fn search(nums: Vec<i32>, target: i32) -> i32 {
        let (mut left, mut right) = (0, nums.len() as i32 - 1);

        while left <= right {
            let mid = left + (right - left) / 2;
            if nums[mid as usize] == target {
                return mid;
            }

            if nums[left as usize] <= nums[mid as usize] {
                if nums[left as usize] <= target && target < nums[mid as usize] {
                    right = mid - 1;
                } else {
                    left = mid + 1;
                }
            } else {
                if nums[mid as usize] < target && target <= nums[right as usize] {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }
        }

        -1
    }
}
```

- (0153) find-minimum-in-rotated-sorted-array
  - 搜索旋转后数组的最小值
  - 输入：nums = [4,5,6,7,0,1,2] 输出：0
  - 二分查找+判断左右有序区间
  - 时间复杂度：O(log(n))，空间复杂度：O(1)

```rust
impl Solution {
    pub fn find_min(nums: Vec<i32>) -> i32 {
        let mut left = 0;
        let mut right = nums.len() - 1;

        while left < right {
            let mid = left + (right - left) / 2;

            if nums[mid] > nums[right] {
                // Minimum is in the right half
                left = mid + 1;
            } else {
                // Minimum is in the left half or mid itself
                right = mid;
            }
        }

        nums[left]
    }
}
```

- (0081) search-in-rotated-sorted-array-ii
  - 搜索旋转后有重复元素数组的 target
  - 输入：nums = [2,5,6,0,0,1,2], target = 0 输出：true
  - 二分查找+判断左右有序区间+相等情况边界缩小处理
  - 时间复杂度：O(log(n))，空间复杂度：O(1)

```rust
impl Solution {
    pub fn search(nums: Vec<i32>, target: i32) -> bool {
        let (mut left,mut right) = (0,nums.len() as i32 - 1);
        while left <= right {
            let mid = left + (right - left) / 2;
            let mid_usize = mid as usize;
            if nums[mid_usize] == target { return true;}

            if nums[left as usize] == nums[mid_usize] && nums[mid_usize] == nums[right as usize] {
                left += 1;
                right -= 1;
                continue;
            }

            let left_sorted = nums[left as usize] <= nums[mid_usize];
            let target_in_left = if left_sorted {
                (nums[left as usize]..nums[mid_usize]).contains(&target)
            } else {
                target <= nums[right as usize] || target > nums[mid_usize]
            };

            if target_in_left { right = mid - 1;}
            else { left = mid + 1; }
        }

        false
    }
}
```

- (0374) guess-number-higher-or-lower
  - 猜数字大小，猜测数字是否等于目标
  - 输入：n = 10, pick = 6 输出：6
  - 二分查找+判断大小
  - 时间复杂度：O(log(n))，空间复杂度：O(1)

```rust
/**
 * Forward declaration of guess API.
 * @param  num   your guess
 * @return       -1 if num is higher than the picked number
 *            1 if num is lower than the picked number
 *               otherwise return 0
 */
use std::cmp::Ordering;
impl Solution {
    unsafe fn guessNumber(n: i32) -> i32 {
        let (mut low, mut high) = (1, n);
        while low <= high {
            let mid = low + (high - low) / 2;
            let res = guess(mid as i32);
            match guess(mid) {
                -1 => high = mid - 1,
                1 => low = mid + 1,
                _ => return mid,
            }
        }
        low
    }
}
```

- (0278) first-bad-version
  - 第一个错误版本，版本回溯
  - 输入：n = 5, bad = 4 输出：4
  - 二分查找
  - 时间复杂度：O(log(n))，空间复杂度：O(1)

```
impl Solution {
    pub fn first_bad_version(&self, n: i32) -> i32 {
        let (mut left, mut right) = (1_i32, n as i32);
        while left < right {
            let mid = left + (right - left) / 2;
            match self.isBadVersion(mid) {
                false => left = mid + 1,
                true => right = mid,
            }
        }
        left
    }
}
```

- (0074) search-a-2d-matrix
  - 搜索二维矩阵，矩阵行列单调递增有序
  - 输入：matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3 输出：true
  - 输入：matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 13 输出：false
  - 行+列二分查找
  - 时间复杂度：O(log(mn))，空间复杂度：O(1)

```
use std::cmp::Ordering;
impl Solution {
    pub fn search_matrix(matrix: Vec<Vec<i32>>, target: i32) -> bool {
        let (outter_length, inner_length) = (matrix.len() as usize, matrix[0].len() as usize);
        if target < matrix[0][0] || target > matrix[outter_length - 1][inner_length - 1] {
            return false;
        }
        let (mut outter_left, mut outter_right) = (0_i32, outter_length as i32);
        let (mut inner_left, mut inner_right) = (0_i32, inner_length as i32);
        while (outter_left < outter_right) {
            let outter_mid = outter_left + (outter_right - outter_left) / 2;
            match matrix[outter_mid as usize][0].cmp(&target) {
                Ordering::Less => outter_left = outter_mid + 1,
                Ordering::Greater => outter_right = outter_mid,
                Ordering::Equal => return true,
            }
        }
        while (inner_left < inner_right) {
            let inner_mid = inner_left + (inner_right - inner_left) / 2;
            match matrix[outter_right as usize - 1][inner_mid as usize].cmp(&target) {
                Ordering::Less => inner_left = inner_mid + 1,
                Ordering::Greater => inner_right = inner_mid,
                Ordering::Equal => return true,
            }
        }
        false
    }
}
```

- (0367) valid-perfect-square
  - 验证完全平方数
  - 输入：num = 16 输出：true
  - 二分查找(1-num/2)
  - 时间复杂度：O(log(n))，空间复杂度：O(1)

```
impl Solution {
    pub fn is_perfect_square(num: i32) -> bool {
        if num == 1 { return true;} // special case num == 1
        let (mut left, mut right) = (1_i64, (num as i64) / 2);
        while left <= right {
            let mid = left + (right - left) / 2;
            match (mid * mid).cmp(&(num as i64)) {
                std::cmp::Ordering::Less => left = mid + 1,
                std::cmp::Ordering::Greater => right = mid - 1,
                std::cmp::Ordering::Equal => return true,
            }
        }
        false
    }
}
```

- (0069) sqrtx
  - x 的平方根
  - 输入:x = 8 输出:2
  - (8 的算术平方根是 2.8..., 由于返回类型是整数，小数部分将被舍去。)
  - 二分查找(1-x/2)
  - 时间复杂度：O(log(n))，空间复杂度：O(1)

```
impl Solution {
    pub fn my_sqrt(x: i32) -> i32 {
        if x == 0 || x == 1 {return x;}
        let (mut left, mut right) = (1_i32, x);
        while left < right {
            let mid = left + (right - left + 1) / 2;
            match (mid as i64 * mid as i64).cmp(&(x as i64)) {
                std::cmp::Ordering::Less | std::cmp::Ordering::Equal => left = mid,
                std::cmp::Ordering::Greater => right = mid - 1,
            }
        }
        left
    }
}
```

- (0441) arranging-coins
  - 排列硬币，n 枚硬币排列成金字塔形状
  - 输入：n = 5 输出：2;输入：n = 8 输出：3
  - 二分查找
  - 时间复杂度：O(log(n))，空间复杂度：O(1)

```
impl Solution {
    pub fn arrange_coins(n: i32) -> i32 {
        let (mut left, mut right) = (1_i32, n as i32);
        while left < right {
            let mid = left + (right - left + 1) / 2;
            match (mid as i64 * (mid as i64 + 1)).cmp(&(2 * n as i64)) {
                std::cmp::Ordering::Less | std::cmp::Ordering::Equal => left = mid,
                std::cmp::Ordering::Greater => right = mid - 1,
            }
        }
        left
    }
}
```

- (1539) kth-missing-positive-number
  - 寻找第 k 个缺失的正整数
  - 输入：arr = [2,3,4,7,11], k = 5 输出：9
  - 缺失的正整数包括 [1,5,6,8,9,10,12,13,…] 。第 5 个缺失的正整数为 9 。
  - 找规律发现$a_i - i - 1 = p_i$, 二分查找
  - 时间复杂度：O(log(n)) ，空间复杂度：O(1)

```
use std::cmp::Ordering;
impl Solution {
    pub fn find_kth_positive(arr: Vec<i32>, k: i32) -> i32 {
        let (mut left_index, mut right_index) = (0_i32, arr.len() as i32 - 1);
        let (mut left, mut right) = (arr[0] - 1, arr[arr.len() - 1] - arr.len() as i32);
        if(k <= left){return k;}
        if(k > right){
            return arr[right_index as usize] + (k - right);
        }
        while(left_index <= right_index){
            let mid = left_index + (right_index - left_index) / 2;
            let cal_res = arr[mid as usize] - mid - 1;
            match cal_res.cmp(&k){
                Ordering::Less => left_index = mid + 1,
                Ordering::Greater | Ordering::Equal => right_index = mid - 1,
            }
        }
        arr[right_index as usize] + k - (arr[right_index as usize] - right_index - 1)

    }
}
```

- (0275) h-index-ii
  - 找到高引用指数：至少 n 篇论文引用次数大于 n
  - 输入：citations = [0,1,3,5,6] 输出：3
  - 升序数组二分查找
  - 时间复杂度：O(log(n)) ，空间复杂度：O(1)

```
use std::cmp::Ordering;
impl Solution {
    pub fn h_index(citations: Vec<i32>) -> i32 {
        let (mut left, mut right, length) = (0_i32, citations.len() as i32 - 1, citations.len() as i32);
        while left <= right{
            let mid = left + (right - left) / 2;
            let len = length - mid;
            match &citations[mid as usize].cmp(&len){
                Ordering::Less => left = mid + 1,
                Ordering::Greater => right = mid - 1,
                Ordering::Equal => return len,
            }
        }
        length - left
    }
}
```

- (0540) single-element-in-a-sorted-array
  - 找到有序数组中的单一元素
  - 输入: nums = [1,1,2,3,3,4,4,8,8] 输出: 2
  - 选取偶数坐标进行二分搜索
  - 时间复杂度：O(log(n))，空间复杂度：O(1)

```
impl Solution {
    pub fn single_non_duplicate(nums: Vec<i32>) -> i32 {
        let (mut left, mut right) = (0, nums.len() - 1);
        while left < right {
            let mid = left + (right - left) / 2;
            // 确保 mid 是偶数索引
            let mid = if mid % 2 == 1 { mid - 1 } else { mid };
            match nums[mid].cmp(&nums[mid + 1]) {
                std::cmp::Ordering::Equal => left = mid + 2,
                _ => right = mid,
            }
        }
        nums[left]
    }
}
```

- (0658) find-k-closest-elements
  - 给定一个 排序好 的数组 arr，两个整数 k 和 x，从数组中找到最靠近 x（两数之差最小）的 k 个数。返回的结果必须要是按升序排好的。
  - 输入：arr = [1,2,3,4,5], k = 4, x = 3 输出：[1,2,3,4]
  - 确定 x 的位置,再进行移动左右边界
  - 时间复杂度：O(klog(n))，空间复杂度：O(1)

```
impl Solution {
    pub fn find_closest_elements(arr: Vec<i32>, k: i32, x: i32) -> Vec<i32> {
        let n = arr.len();
        let (mut left, mut right) = (
            (arr.partition_point(|&num| num < x) as i32) - 1,
            arr.partition_point(|&num| num < x) as i32,
        );
        (0..k).for_each(|_| match (left >= 0, right < n as i32) {
            (false, true) => right += 1,
            (true, false) => left -= 1,
            (true, true) => {
                if x - arr[left as usize] <= arr[right as usize] - x {left -= 1}
                else {right += 1}
            }
            _ => unreachable!(),
        });

        arr[(left + 1) as usize..right as usize].to_vec()
    }
}
```

- (0852) peak-index-in-a-mountain-array
  - 给定一个长度为 n 的整数 山脉 数组 arr，其中的值递增到一个峰值元素然后递减。
  - 输入：arr = [0,2,1,0] 输出：1
  - 判断右邻接元素是否小于自己,二分搜索
  - 时间复杂度：O(log(n))，空间复杂度：O(1)

```rust
impl Solution {
    pub fn peak_index_in_mountain_array(arr: Vec<i32>) -> i32 {
        let (mut left, mut right) = (0, arr.len() - 1);
        while left < right {
            let mid = left + (right - left) / 2;
            if arr[mid] < arr[mid + 1] {
                left = mid + 1;
            } else {
                right = mid;
            }
        }
        left as i32
    }
}
```

- (0004) median-of-two-sorted-arrays
  - 给定两个大小分别为 m 和 n 的正序（从小到大）数组 nums1 和 nums2。请你找出并返回这两个正序数组的 中位数 。算法的时间复杂度应该为 O(log (m+n)) 。
  - 输入：nums1 = [1,2], nums2 = [3,4] 输出：2.50000
  - 输入：nums1 = [1,3], nums2 = [2] 输出：2.00000
  - 对两个数组进行排序后,用二分搜索确定
  - 时间复杂度：O(log (m+n)) ， 空间复杂度：O(m+n)

```rust
// 对于两个排序数组合并后的中位数,实际上我们是要找到一个点,将所有元素分成两半。
// 左半部分的所有元素都小于或等于右半部分的所有元素。
impl Solution {
    pub fn find_median_sorted_arrays(nums1: Vec<i32>, nums2: Vec<i32>) -> f64 {
        let (m, n) = (nums1.len(), nums2.len());

        // 确保 nums1 是较短的数组
        if m > n {
            return Self::find_median_sorted_arrays(nums2, nums1);
        }

        let (mut low, mut high) = (0, m);
        let mut result = 0.0;

        while low <= high {
            // 左半部分的总元素个数应该是 (m + n + 1) / 2
            // nums1 中的分割点
            let partition_x = (low + high) / 2;
            // nums2 中的分割点
            let partition_y = (m + n + 1) / 2 - partition_x;

            let max_left_x = if partition_x == 0 {i32::MIN} else {nums1[partition_x - 1]};
            let min_right_x = if partition_x == m {i32::MAX} else {nums1[partition_x]};
            let max_left_y = if partition_y == 0 {i32::MIN} else {nums2[partition_y - 1]};
            let min_right_y = if partition_y == n {i32::MAX} else {nums2[partition_y]};

            if max_left_x <= min_right_y && max_left_y <= min_right_x {
                // 找到正确的分割点
                if (m + n) % 2 == 0 {
                    result = (max_left_x.max(max_left_y) as f64
                        + min_right_x.min(min_right_y) as f64)
                        / 2.0;
                } else {
                    result = max_left_x.max(max_left_y) as f64;
                }
                break;
            } else if max_left_x > min_right_y {
                // 需要向左移动
                high = partition_x - 1;
            } else {
                // 需要向右移动
                low = partition_x + 1;
            }
        }

        result
    }
}
```