==========
文件夹: class026_BinarySearch
==========

[Markdown 文件]
==========
文件: BinarySearchProblems.md
==========
二分查找算法详解与相关题目汇总

算法介绍

二分查找（Binary Search）是一种在已排序数组中查找特定元素的高效算法。它通过不断将搜索范围减半来快速定位目标元素，时间复杂度为 O(log n)。

基本思想：
1. 在有序数组中，比较目标值与中间元素
2. 如果相等，则找到目标
3. 如果目标值较小，则在左半部分继续查找
4. 如果目标值较大，则在右半部分继续查找
5. 重复直到找到目标或搜索范围为空

二分查找的三种模板

模板 1：基础二分查找
查找目标值是否存在

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

模板 2：查找左边界
查找第一个大于等于目标值的位置

```java
public static int binarySearchLeft(int[] arr, int target) {
    int left = 0, right = arr.length;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return left;
}
```

模板 3：查找右边界
查找最后一个小于等于目标值的位置

```java
public static int binarySearchRight(int[] arr, int target) {
    int left = 0, right = arr.length;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] <= target) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return left - 1;
}
```

相关题目列表

基础题目

1. LeetCode 704. Binary Search (二分查找)
在有序数组中查找目标值

题目描述：
给定一个 n 个元素有序的（升序）整型数组 nums 和一个目标值 target，写一个函数搜索 nums 中的 target，如果目标值存在返回下标，否则返回 -1。

解题思路：
使用基础二分查找模板

时间复杂度：O(log n)
空间复杂度：O(1)

```java
// Java 实现
public static int search(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            return mid;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}
```

```cpp
// C++实现
int search(vector<int>& nums, int target) {
    int left = 0, right = nums.size() - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            return mid;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
```

```
    return -1;
}
```

```python
# Python 实现
def search(nums, target):
    left, right = 0, len(nums) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

2. LeetCode 35. Search Insert Position (搜索插入位置)
查找目标值应该插入的位置

题目描述：
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。

解题思路：
使用左边界查找模板

时间复杂度：O(log n)
空间复杂度：O(1)

```java
// Java 实现
public static int searchInsert(int[] nums, int target) {
    int left = 0, right = nums.length;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid;
        }
```

```
    }
    return left;
}
```

```cpp
// C++实现
int searchInsert(vector<int>& nums, int target) {
    int left = 0, right = nums.size();
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return left;
}
```

```python
# Python 实现
def search_insert(nums, target):
    left, right = 0, len(nums)
    while left < right:
        mid = left + (right - left) // 2
        if nums[mid] < target:
            left = mid + 1
        else:
            right = mid
    return left
```

3. LeetCode 34. Find First and Last Position of Element in Sorted Array (在排序数组中查找元素的第一个和最后一个位置)
查找目标值在数组中的起始和结束位置

题目描述：
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。

解题思路：

分别查找左边界和右边界

时间复杂度：O(log n)
空间复杂度：O(1)

```java
// Java 实现
public static int[] searchRange(int[] nums, int target) {
    int first = findLeft(nums, target);
    if (first == nums.length || nums[first] != target) {
        return new int[]{-1, -1};
    }
    int last = findRight(nums, target);
    return new int[]{first, last};
}

private static int findLeft(int[] nums, int target) {
    int left = 0, right = nums.length;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return left;
}

private static int findRight(int[] nums, int target) {
    int left = 0, right = nums.length;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] <= target) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return left - 1;
}
```

```cpp
// C++实现
vector<int> searchRange(vector<int>& nums, int target) {
    int first = findLeft(nums, target);
    if (first == nums.size() || nums[first] != target) {
        return {-1, -1};
    }
    int last = findRight(nums, target);
    return {first, last};
}

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

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

```python
# Python 实现
def search_range(nums, target):
    first = find_left(nums, target)
    if first == len(nums) or nums[first] != target:
```

```
        return [-1, -1]
    last = find_right(nums, target)
    return [first, last]

def find_left(nums, target):
    left, right = 0, len(nums)
    while left < right:
        mid = left + (right - left) // 2
        if nums[mid] < target:
            left = mid + 1
        else:
            right = mid
    return left

def find_right(nums, target):
    left, right = 0, len(nums)
    while left < right:
        mid = left + (right - left) // 2
        if nums[mid] <= target:
            left = mid + 1
        else:
            right = mid
    return left - 1
```

4. LeetCode 367. Valid Perfect Square (有效的完全平方数)
判断一个数是否是完全平方数

题目描述：
给定一个正整数 num，编写一个函数，如果 num 是一个完全平方数，则返回 True，否则返回 False。

解题思路：
使用二分查找在 [1, num] 范围内查找是否存在一个数的平方等于 num

时间复杂度：O(log n)
空间复杂度：O(1)

```java
// Java 实现
public static boolean isPerfectSquare(int num) {
    if (num < 1) return false;
    if (num == 1) return true;
```

```
    long left = 1, right = num / 2;
    while (left <= right) {
        long mid = left + (right - left) / 2;
        long square = mid * mid;
        if (square == num) {
            return true;
        } else if (square < num) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return false;
}
```

```cpp
// C++实现
bool isPerfectSquare(int num) {
    if (num < 1) return false;
    if (num == 1) return true;

    long long left = 1, right = num / 2;
    while (left <= right) {
        long long mid = left + (right - left) / 2;
        long long square = mid * mid;
        if (square == num) {
            return true;
        } else if (square < num) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return false;
}
```

```python
# Python 实现
def is_perfect_square(num):
    if num < 1:
        return False
```

```
    if num == 1:
        return True

    left, right = 1, num // 2
    while left <= right:
        mid = left + (right - left) // 2
        square = mid * mid
        if square == num:
            return True
        elif square < num:
            left = mid + 1
        else:
            right = mid - 1
    return False
```

5. LeetCode 278. First Bad Version (第一个错误的版本)
查找第一个错误的版本

题目描述：
假设你有 n 个版本 [1, 2, ..., n]，你想找出导致之后所有版本出错的第一个错误的版本。

解题思路：
使用左边界查找模板

时间复杂度：O(log n)
空间复杂度：O(1)

```java
// Java 实现 (假设存在 isBadVersion API)
public static int firstBadVersion(int n) {
    int left = 1, right = n;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (isBadVersion(mid)) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    return left;
}
```

```cpp
// C++实现 (假设存在 isBadVersion API)
int firstBadVersion(int n) {
    int left = 1, right = n;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (isBadVersion(mid)) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    return left;
}
```

```python
# Python 实现 (假设存在 isBadVersion API)
def first_bad_version(n):
    left, right = 1, n
    while left < right:
        mid = left + (right - left) // 2
        if isBadVersion(mid):
            right = mid
        else:
            left = mid + 1
    return left
```

进阶题目

6. LeetCode 162. Find Peak Element (寻找峰值)
寻找数组中的峰值元素

题目描述:
峰值元素是指其值大于左右相邻值的元素。给定一个输入数组 nums，其中 nums[i] ≠ nums[i+1]，找到峰值元素并返回其索引。

解题思路:
使用二分查找，比较 mid 和 mid+1 的值决定搜索方向

时间复杂度: O(log n)

空间复杂度：O(1)

```java
// Java 实现
public static int findPeakElement(int[] nums) {
    int left = 0, right = nums.length - 1;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] > nums[mid + 1]) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    return left;
}
```

```cpp
// C++实现
int findPeakElement(vector<int>& nums) {
    int left = 0, right = nums.size() - 1;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] > nums[mid + 1]) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    return left;
}
```

```python
# Python 实现
def find_peak_element(nums):
    left, right = 0, len(nums) - 1
    while left < right:
        mid = left + (right - left) // 2
        if nums[mid] > nums[mid + 1]:
            right = mid
        else:
```

```
            left = mid + 1
    return left
```

7. LeetCode 153. Find Minimum in Rotated Sorted Array (寻找旋转排序数组中的最小值)
寻找旋转排序数组中的最小值

题目描述：
假设按照升序排序的数组在预先未知的某个点上进行了旋转。请找出其中最小的元素。

解题思路：
使用二分查找，比较 mid 和 right 的值决定搜索方向

时间复杂度：O(log n)
空间复杂度：O(1)

```java
// Java 实现
public static int findMin(int[] nums) {
    int left = 0, right = nums.length - 1;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < nums[right]) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    return nums[left];
}
```

```cpp
// C++实现
int findMin(vector<int>& nums) {
    int left = 0, right = nums.size() - 1;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < nums[right]) {
            right = mid;
        } else {
            left = mid + 1;
        }
```

```
    }
    return nums[left];
}
```

```python
# Python 实现
def find_min(nums):
    left, right = 0, len(nums) - 1
    while left < right:
        mid = left + (right - left) // 2
        if nums[mid] < nums[right]:
            right = mid
        else:
            left = mid + 1
    return nums[left]
```

8. LeetCode 74. Search a 2D Matrix (搜索二维矩阵)
在二维矩阵中搜索目标值

题目描述：
编写一个高效的算法来搜索 m x n 矩阵 matrix 中的一个目标值 target。该矩阵具有以下特性：
- 每行的元素从左到右升序排列。
- 每列的元素从上到下升序排列。

解题思路：
将二维矩阵看作一维数组进行二分查找

时间复杂度：O(log(m*n))
空间复杂度：O(1)

```java
// Java 实现
public static boolean searchMatrix(int[][] matrix, int target) {
    if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
        return false;
    }

    int m = matrix.length, n = matrix[0].length;
    int left = 0, right = m * n - 1;

    while (left <= right) {
```

```
        int mid = left + (right - left) / 2;
        int midValue = matrix[mid / n][mid % n];

        if (midValue == target) {
            return true;
        } else if (midValue < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return false;
}
```

```cpp
// C++实现
bool searchMatrix(vector<vector<int>>& matrix, int target) {
    if (matrix.empty() || matrix[0].empty()) {
        return false;
    }

    int m = matrix.size(), n = matrix[0].size();
    int left = 0, right = m * n - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        int midValue = matrix[mid / n][mid % n];

        if (midValue == target) {
            return true;
        } else if (midValue < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return false;
}
```

```python
# Python 实现
def search_matrix(matrix, target):
    if not matrix or not matrix[0]:
        return False

    m, n = len(matrix), len(matrix[0])
    left, right = 0, m * n - 1

    while left <= right:
        mid = left + (right - left) // 2
        mid_value = matrix[mid // n][mid % n]

        if mid_value == target:
            return True
        elif mid_value < target:
            left = mid + 1
        else:
            right = mid - 1

    return False
```

9. LeetCode 33. Search in Rotated Sorted Array (搜索旋转排序数组)
在旋转排序数组中搜索目标值

题目描述：
整数数组 nums 按升序排列，数组中的值互不相同。在传递给函数之前，nums 在某个未知的下标 k 上进行了旋转。给定旋转后的数组 nums 和一个整数 target，如果 nums 中存在这个目标值 target，则返回它的下标，否则返回 -1。

解题思路：
使用二分查找，根据 mid 位置判断哪一部分是有序的

时间复杂度：O(log n)
空间复杂度：O(1)

```java
// Java 实现
public static int search(int[] nums, int target) {
    int left = 0, right = nums.length - 1;

    while (left <= right) {
```

```
        int mid = left + (right - left) / 2;

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

        // 左半部分有序
        if (nums[left] <= nums[mid]) {
            if (nums[left] <= target && target < nums[mid]) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
        // 右半部分有序
        else {
            if (nums[mid] < target && target <= nums[right]) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
    }

    return -1;
}
```

```cpp
// C++实现
int search(vector<int>& nums, int target) {
    int left = 0, right = nums.size() - 1;

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

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

        // 左半部分有序
        if (nums[left] <= nums[mid]) {
            if (nums[left] <= target && target < nums[mid]) {
```

```
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
        // 右半部分有序
        else {
            if (nums[mid] < target && target <= nums[right]) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
    }

    return -1;
}
```

```python
# Python 实现
def search(nums, target):
    left, right = 0, len(nums) - 1

    while left <= right:
        mid = left + (right - left) // 2

        if nums[mid] == target:
            return mid

        # 左半部分有序
        if nums[left] <= nums[mid]:
            if nums[left] <= target < nums[mid]:
                right = mid - 1
            else:
                left = mid + 1
        # 右半部分有序
        else:
            if nums[mid] < target <= nums[right]:
                left = mid + 1
            else:
                right = mid - 1
```

```
    return -1
```

10. LeetCode 287. Find the Duplicate Number (寻找重复数)
在数组中寻找重复的数字

题目描述：
给定一个包含 n + 1 个整数的数组 nums，其数字都在 1 到 n 之间（包括 1 和 n），可知至少存在一个重复的整数。假设只有一个重复的整数，找出这个重复的数。

解题思路：
使用二分查找结合抽屉原理

时间复杂度：O(n log n)
空间复杂度：O(1)

```java
// Java 实现
public static int findDuplicate(int[] nums) {
    int left = 1, right = nums.length - 1;

    while (left < right) {
        int mid = left + (right - left) / 2;
        int count = 0;

        // 计算小于等于 mid 的数字个数
        for (int num : nums) {
            if (num <= mid) {
                count++;
            }
        }

        // 根据抽屉原理判断重复数字在哪一侧
        if (count > mid) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }

    return left;
}
```

```cpp
// C++实现
int findDuplicate(vector<int>& nums) {
    int left = 1, right = nums.size() - 1;

    while (left < right) {
        int mid = left + (right - left) / 2;
        int count = 0;

        // 计算小于等于 mid 的数字个数
        for (int num : nums) {
            if (num <= mid) {
                count++;
            }
        }

        // 根据抽屉原理判断重复数字在哪一侧
        if (count > mid) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }

    return left;
}
```

```python
# Python 实现
def find_duplicate(nums):
    left, right = 1, len(nums) - 1

    while left < right:
        mid = left + (right - left) // 2
        count = 0

        # 计算小于等于 mid 的数字个数
        for num in nums:
            if num <= mid:
                count += 1
```

```
        # 根据抽屉原理判断重复数字在哪一侧
        if count > mid:
            right = mid
        else:
            left = mid + 1

    return left
```

Codeforces 题目

11. Codeforces 706B. Interesting drink
在商店中查找可以购买的数量

题目描述：
Vasiliy 喜欢在超市购物。他想买一些饮料，每种饮料都有一个价格。他想知道对于给定的金额，他能买多少种饮料。

解题思路：
先对价格数组排序，然后对每个查询使用二分查找

时间复杂度：O(n log n + q log n)
空间复杂度：O(1)

```java
// Java 实现
import java.util.*;

public static int[] howManyDrinks(int[] prices, int[] queries) {
    Arrays.sort(prices);
    int[] result = new int[queries.length];

    for (int i = 0; i < queries.length; i++) {
        int money = queries[i];
        int left = 0, right = prices.length;

        while (left < right) {
            int mid = left + (right - left) / 2;
            if (prices[mid] <= money) {
                left = mid + 1;
            } else {
                right = mid;
            }
```

```
        }

        result[i] = left;
    }

    return result;
}
```

```cpp
// C++实现
#include <vector>
#include <algorithm>

vector<int> howManyDrinks(vector<int>& prices, vector<int>& queries) {
    sort(prices.begin(), prices.end());
    vector<int> result(queries.size());

    for (int i = 0; i < queries.size(); i++) {
        int money = queries[i];
        int left = 0, right = prices.size();

        while (left < right) {
            int mid = left + (right - left) / 2;
            if (prices[mid] <= money) {
                left = mid + 1;
            } else {
                right = mid;
            }
        }

        result[i] = left;
    }

    return result;
}
```

```python
# Python 实现
def how_many_drinks(prices, queries):
    prices.sort()
    result = []
```

```
    for money in queries:
        left, right = 0, len(prices)

        while left < right:
            mid = left + (right - left) // 2
            if prices[mid] <= money:
                left = mid + 1
            else:
                right = mid

        result.append(left)

    return result
```

SPOJ 题目

12. SPOJ - AGGRCOW (Aggressive cows)
安排奶牛使最小距离最大化

题目描述：
农夫约翰建造了一个有 C 个摊位的畜棚，摊位位于 x0, ..., x(C-1)。他的 M 头奶牛总是相互攻击，约翰必须以某种方式分配奶牛到摊位，使它们之间的最小距离尽可能大。

解题思路：
使用二分查找答案 + 贪心验证

时间复杂度：O(n log(max-min) * n)
空间复杂度：O(1)

```java
// Java 实现
import java.util.*;

public static int aggressiveCows(int[] positions, int cows) {
    Arrays.sort(positions);
    int left = 0, right = positions[positions.length - 1] - positions[0];
    int result = 0;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (canPlaceCows(positions, cows, mid)) {
```

```
            result = mid;
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return result;
}

private static boolean canPlaceCows(int[] positions, int cows, int minDist) {
    int count = 1;
    int lastPosition = positions[0];

    for (int i = 1; i < positions.length; i++) {
        if (positions[i] - lastPosition >= minDist) {
            count++;
            lastPosition = positions[i];
            if (count == cows) {
                return true;
            }
        }
    }

    return false;
}
```

```cpp
// C++实现
#include <vector>
#include <algorithm>

int aggressiveCows(std::vector<int>& positions, int cows) {
    std::sort(positions.begin(), positions.end());
    int left = 0, right = positions.back() - positions.front();
    int result = 0;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (canPlaceCows(positions, cows, mid)) {
            result = mid;
            left = mid + 1;
```

```
        } else {
            right = mid - 1;
        }
    }

    return result;
}

bool canPlaceCows(std::vector<int>& positions, int cows, int minDist) {
    int count = 1;
    int lastPosition = positions[0];

    for (int i = 1; i < positions.size(); i++) {
        if (positions[i] - lastPosition >= minDist) {
            count++;
            lastPosition = positions[i];
            if (count == cows) {
                return true;
            }
        }
    }

    return false;
}
```

```python
# Python 实现
def aggressive_cows(positions, cows):
    positions.sort()
    left, right = 0, positions[-1] - positions[0]
    result = 0

    while left <= right:
        mid = left + (right - left) // 2
        if can_place_cows(positions, cows, mid):
            result = mid
            left = mid + 1
        else:
            right = mid - 1

    return result
```

```
def can_place_cows(positions, cows, min_dist):
    count = 1
    last_position = positions[0]

    for i in range(1, len(positions)):
        if positions[i] - last_position >= min_dist:
            count += 1
            last_position = positions[i]
            if count == cows:
                return True

    return False
```

工程化考量

异常处理

在实际工程应用中，我们需要考虑以下异常情况：

1. 空数组处理
2. 数组未排序
3. 数据溢出
4. 输入参数非法

```java
// Java - 带异常处理的二分查找
public static int safeBinarySearch(int[] nums, int target) {
    // 检查输入参数
    if (nums == null) {
        throw new IllegalArgumentException("数组不能为null");
    }

    if (nums.length == 0) {
        return -1;
    }

    int left = 0, right = nums.length - 1;

    // 检查数组是否已排序
    for (int i = 1; i < nums.length; i++) {
        if (nums[i] < nums[i-1]) {
            throw new IllegalArgumentException("数组必须是已排序的");
```

```
        }
    }

    while (left <= right) {
        // 防止整数溢出的中点计算
        int mid = left + (right - left) / 2;

        if (nums[mid] == target) {
            return mid;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1;
}
```

单元测试

```java
// Java - 二分查找的单元测试
import org.junit.Test;
import static org.junit.Assert.*;

public class BinarySearchTest {

    @Test
    public void testBasicSearch() {
        int[] nums = {-1, 0, 3, 5, 9, 12};
        assertEquals(4, BinarySearch.search(nums, 9));
        assertEquals(-1, BinarySearch.search(nums, 2));
    }

    @Test
    public void testEdgeCases() {
        // 空数组
        int[] empty = {};
        assertEquals(-1, BinarySearch.search(empty, 1));

        // 单元素数组
```

```
        int[] single = {5};
        assertEquals(0, BinarySearch.search(single, 5));
        assertEquals(-1, BinarySearch.search(single, 3));

        // 两元素数组
        int[] two = {1, 3};
        assertEquals(0, BinarySearch.search(two, 1));
        assertEquals(1, BinarySearch.search(two, 3));
        assertEquals(-1, BinarySearch.search(two, 2));
    }

    @Test
    public void testBoundaries() {
        int[] nums = {1, 2, 3, 4, 5};
        assertEquals(0, BinarySearch.search(nums, 1));  // 第一个元素
        assertEquals(4, BinarySearch.search(nums, 5));  // 最后一个元素
    }
}
```

性能优化

1. 使用位运算优化中点计算：`mid = (left + right) >>> 1`
2. 减少比较次数
3. 使用迭代而非递归避免栈溢出

```java
// Java - 性能优化版本
public static int optimizedBinarySearch(int[] nums, int target) {
    int left = 0, right = nums.length - 1;

    while (left <= right) {
        // 使用无符号右移避免溢出
        int mid = (left + right) >>> 1;

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

        // 使用位运算优化比较
        if (nums[mid] < target) {
            left = mid + 1;
        } else {
```

```
            right = mid - 1;
        }
    }

    return -1;
}
```

13. LeetCode 69. Sqrt(x)

题目描述
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。

解题思路
- 这是一个典型的二分查找应用，可以在 [0, x] 范围内查找满足 `mid * mid <= x` 的最大整数
- 需要注意整数溢出问题，使用 `mid > x / mid` 而不是 `mid * mid > x` 来判断

时间复杂度
- O(log x)，二分查找的时间复杂度

空间复杂度
- O(1)，只使用常数额外空间

Java 实现
```java
/**
 * LeetCode 69. Sqrt(x)
 *
 * 时间复杂度: O(log x)
 * 空间复杂度: O(1)
 *
 * @param x 输入的非负整数
 * @return x 的平方根的整数部分
 */
public int mySqrt(int x) {
    if (x == 0 || x == 1) {
        return x;
    }

    int left = 1, right = x;
    while (left <= right) {
        int mid = left + ((right - left) >> 1);
```

```
        // 防止整数溢出，使用除法而不是乘法
        if (mid > x / mid) {
            right = mid - 1;
        } else {
            // 如果下一个值会溢出或者大于 x/mid，则当前 mid 是最大的有效平方根
            if (mid + 1 > x / (mid + 1)) {
                return mid;
            }
            left = mid + 1;
        }
    }

    return right;
}
```

C++ 实现
```cpp
/**
 * LeetCode 69. Sqrt(x)
 *
 * 时间复杂度: O(log x)
 * 空间复杂度: O(1)
 *
 * @param x 输入的非负整数
 * @return x 的平方根的整数部分
 */
int mySqrt(int x) {
    if (x == 0 || x == 1) {
        return x;
    }

    long long left = 1, right = x;
    while (left <= right) {
        long long mid = left + ((right - left) >> 1);

        // 使用 long long 防止整数溢出
        if (mid * mid > x) {
            right = mid - 1;
        } else {
            if ((mid + 1) * (mid + 1) > x) {
                return mid;
```

```
            }
            left = mid + 1;
        }
    }

    return right;
}
```

Python 实现
```python
def my_sqrt(x):
    """
    LeetCode 69. Sqrt(x)

    时间复杂度: O(log x)
    空间复杂度: O(1)

    :param x: 输入的非负整数
    :return: x 的平方根的整数部分
    """
    if x == 0 or x == 1:
        return x

    left, right = 1, x
    while left <= right:
        mid = left + ((right - left) >> 1)

        # 防止整数溢出，使用除法而不是乘法
        if mid > x // mid:
            right = mid - 1
        else:
            if (mid + 1) > x // (mid + 1):
                return mid
            left = mid + 1

    return right
```

14. LeetCode 374. 猜数字大小

题目描述
我们正在玩一个猜数字游戏。游戏规则如下：我从 1 到 n 选择一个数字。你需要猜我选了哪个数字。每次你

猜错了，我会告诉你这个数字是大了还是小了。你调用一个预先定义好的接口 `guess(int num)`，它会返回 3 个可能的结果：
- -1：我选的数字比你猜的数字小（即 `pick < num`）
- 1：我选的数字比你猜的数字大（即 `pick > num`）
- 0：我选的数字和你猜的数字一样。恭喜！你猜对了！（即 `pick == num`）

解题思路
- 这是一个标准的二分查找问题，每次猜测中间值，根据 `guess()` 函数的返回结果调整搜索范围
- 需要注意的是，这里的 `guess()` 函数是题目提供的，我们只需要实现二分查找的框架

时间复杂度
- O(log n)，二分查找的时间复杂度

空间复杂度
- O(1)，只使用常数额外空间

Java 实现
```java
/**
 * LeetCode 374. 猜数字大小
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param n 数字范围上限
 * @return 猜中的数字
 */
public int guessNumber(int n) {
    int left = 1, right = n;

    while (left <= right) {
        // 防止整数溢出
        int mid = left + ((right - left) >> 1);
        int result = guess(mid);

        if (result == 0) {
            return mid;  // 猜对了
        } else if (result == 1) {
            left = mid + 1;  // 猜小了，往右边找
        } else {
            right = mid - 1;  // 猜大了，往左边找
        }
    }
```

```
    return -1;  // 理论上不会执行到这里
}

// 注意：这是一个示例的 guess 方法，实际会由系统提供
private int guess(int num) {
    int pick = 6;  // 假设选中的数字是 6
    if (num < pick) return 1;
    else if (num > pick) return -1;
    else return 0;
}
```

C++ 实现
```cpp
/**
 * LeetCode 374. 猜数字大小
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param n 数字范围上限
 * @return 猜中的数字
 */
int guessNumber(int n) {
    long long left = 1, right = n;

    while (left <= right) {
        // 使用 long long 防止整数溢出
        long long mid = left + ((right - left) >> 1);
        int result = guess(mid);

        if (result == 0) {
            return mid;  // 猜对了
        } else if (result == 1) {
            left = mid + 1;  // 猜小了，往右边找
        } else {
            right = mid - 1;  // 猜大了，往左边找
        }
    }

    return -1;  // 理论上不会执行到这里
}
```

```
// 注意：这是一个示例的 guess 函数，实际会由系统提供
int guess(int num) {
    int pick = 6;  // 假设选中的数字是 6
    if (num < pick) return 1;
    else if (num > pick) return -1;
    else return 0;
}
```

Python 实现
```python
def guess_number(n):
    """
    LeetCode 374. 猜数字大小

    时间复杂度: O(log n)
    空间复杂度: O(1)

    :param n: 数字范围上限
    :return: 猜中的数字
    """
    left, right = 1, n

    while left <= right:
        # 防止整数溢出
        mid = left + ((right - left) >> 1)
        result = guess(mid)

        if result == 0:
            return mid  # 猜对了
        elif result == 1:
            left = mid + 1  # 猜小了，往右边找
        else:
            right = mid - 1  # 猜大了，往左边找

    return -1  # 理论上不会执行到这里

# 注意：这是一个示例的 guess 函数，实际会由系统提供
def guess(num):
    pick = 6  # 假设选中的数字是 6
    if num < pick:
        return 1
```

```
    elif num > pick:
        return -1
    else:
        return 0
```

15. LeetCode 875. 爱吃香蕉的珂珂

题目描述
珂珂喜欢吃香蕉。这里有 n 堆香蕉，第 i 堆中有 piles[i] 根香蕉。警卫已经离开了，将在 h 小时后回来。珂珂可以决定她吃香蕉的速度 k（单位：根/小时）。每个小时，她将会选择一堆香蕉，从中吃掉 k 根。如果这堆香蕉少于 k 根，她将吃掉这堆的所有香蕉，然后这一小时内不会再吃更多的香蕉。珂珂喜欢慢慢吃，但仍然想在警卫回来前吃掉所有的香蕉。返回她可以在 h 小时内吃掉所有香蕉的最小速度 k（k 为整数）。

解题思路
- 这是一个典型的二分答案问题，我们需要找到满足条件的最小速度 k
- 速度的最小值是 1，最大值是最大堆的香蕉数
- 对于每一个候选速度 mid，我们计算吃完所有香蕉需要的时间，如果时间 <= h，则可以尝试更小的速度；否则需要更大的速度

时间复杂度
- O(n log maxPile)，其中 n 是香蕉堆的数量，maxPile 是最大的香蕉堆的数量
- 二分查找需要 log maxPile 次迭代，每次迭代需要 O(n) 的时间计算吃完所有香蕉的时间

空间复杂度
- O(1)，只使用常数额外空间

Java 实现
```java
/**
 * LeetCode 875. 爱吃香蕉的珂珂
 *
 * 时间复杂度: O(n log maxPile)
 * 空间复杂度: O(1)
 *
 * @param piles 香蕉堆数组
 * @param h 警卫离开的时间（小时）
 * @return 最小的吃香蕉速度
 */
public int minEatingSpeed(int[] piles, int h) {
    int left = 1;
    int right = 0;
```

```
    // 找到最大的香蕉堆，作为右边界
    for (int pile : piles) {
        right = Math.max(right, pile);
    }

    while (left < right) {
        int mid = left + ((right - left) >> 1);

        if (canFinish(piles, h, mid)) {
            right = mid;  // 可以吃完，尝试更小的速度
        } else {
            left = mid + 1;  // 不能吃完，需要更大的速度
        }
    }

    return left;
}

/**
 * 辅助方法：判断以速度 speed 是否能在 h 小时内吃完所有香蕉
 */
private boolean canFinish(int[] piles, int h, int speed) {
    long time = 0;

    for (int pile : piles) {
        // 计算吃完当前堆需要的时间
        time += (pile + speed - 1) / speed;  // 等价于 Math.ceil(pile / speed)

        // 如果时间已经超过 h，可以提前返回 false
        if (time > h) {
            return false;
        }
    }

    return time <= h;
}
```

C++ 实现
```cpp
/**
 * LeetCode 875. 爱吃香蕉的珂珂
 *
```

```
 * 时间复杂度: O(n log maxPile)
 * 空间复杂度: O(1)
 *
 * @param piles 香蕉堆数组
 * @param h 警卫离开的时间（小时）
 * @return 最小的吃香蕉速度
 */
int minEatingSpeed(vector<int>& piles, int h) {
    int left = 1;
    int right = 0;

    // 找到最大的香蕉堆，作为右边界
    for (int pile : piles) {
        right = max(right, pile);
    }

    while (left < right) {
        int mid = left + ((right - left) >> 1);

        if (canFinish(piles, h, mid)) {
            right = mid;  // 可以吃完，尝试更小的速度
        } else {
            left = mid + 1;  // 不能吃完，需要更大的速度
        }
    }

    return left;
}

/**
 * 辅助方法：判断以速度 speed 是否能在 h 小时内吃完所有香蕉
 */
bool canFinish(vector<int>& piles, int h, int speed) {
    long long time = 0;

    for (int pile : piles) {
        // 计算吃完当前堆需要的时间
        time += (pile + speed - 1) / speed;  // 等价于 ceil(pile / speed)

        // 如果时间已经超过 h，可以提前返回 false
        if (time > h) {
            return false;
        }
```

```
    }

    return time <= h;
}
```

Python 实现
```python
def min_eating_speed(piles, h):
    """
    LeetCode 875. 爱吃香蕉的珂珂

    时间复杂度: O(n log maxPile)
    空间复杂度: O(1)

    :param piles: 香蕉堆数组
    :param h: 警卫离开的时间（小时）
    :return: 最小的吃香蕉速度
    """
    left = 1
    right = max(piles)  # 最大的香蕉堆，作为右边界

    while left < right:
        mid = left + ((right - left) >> 1)

        if can_finish(piles, h, mid):
            right = mid  # 可以吃完，尝试更小的速度
        else:
            left = mid + 1  # 不能吃完，需要更大的速度

    return left

def can_finish(piles, h, speed):
    """
    辅助方法：判断以速度 speed 是否能在 h 小时内吃完所有香蕉
    """
    time = 0

    for pile in piles:
        # 计算吃完当前堆需要的时间
        time += (pile + speed - 1) // speed  # 等价于 math.ceil(pile / speed)

        # 如果时间已经超过 h，可以提前返回 False
```

```
        if time > h:
            return False

    return time <= h
```

16. LeetCode 1011. 在D天内送达包裹的能力

题目描述
传送带上的包裹必须在 D 天内从一个港口运送到另一个港口。传送带上的第 i 个包裹的重量为 weights[i]。每一天，我们都会按给出重量的顺序往传送带上装载包裹，我们装载的重量不会超过船的最大运载重量。返回能在 D 天内将所有包裹送达的船的最小运载能力。

解题思路
- 这是一个典型的二分答案问题，我们需要找到满足条件的最小运载能力
- 运载能力的最小值是最大包裹的重量，最大值是所有包裹的总重量
- 对于每一个候选运载能力 mid，我们计算需要多少天才能运完所有包裹，如果天数 <= D，则可以尝试更小的运载能力；否则需要更大的运载能力

时间复杂度
- O(n log totalWeight)，其中 n 是包裹的数量，totalWeight 是所有包裹的总重量
- 二分查找需要 log totalWeight 次迭代，每次迭代需要 O(n) 的时间计算运完所有包裹的天数

空间复杂度
- O(1)，只使用常数额外空间

Java 实现
```java
/**
 * LeetCode 1011. 在D天内送达包裹的能力
 *
 * 时间复杂度: O(n log totalWeight)
 * 空间复杂度: O(1)
 *
 * @param weights 包裹重量数组
 * @param days 天数限制
 * @return 船的最小运载能力
 */
public int shipWithinDays(int[] weights, int days) {
    int left = 0;  // 最小运载能力：最大的单个包裹重量
    int right = 0; // 最大运载能力：所有包裹的总重量

    for (int weight : weights) {
```

```
        left = Math.max(left, weight);
        right += weight;
    }

    while (left < right) {
        int mid = left + ((right - left) >> 1);

        if (canShipInDays(weights, days, mid)) {
            right = mid;  // 可以在 days 天内运完，尝试更小的运载能力
        } else {
            left = mid + 1;  // 不能在 days 天内运完，需要更大的运载能力
        }
    }

    return left;
}

/**
 * 辅助方法：判断以 capacity 的运载能力是否能在 days 天内运完所有包裹
 */
private boolean canShipInDays(int[] weights, int days, int capacity) {
    int currentWeight = 0;
    int dayCount = 1;  // 至少需要 1 天

    for (int weight : weights) {
        // 如果当前包裹的重量已经超过了运载能力，不可能运完
        if (weight > capacity) {
            return false;
        }

        // 如果当前累计重量加上当前包裹的重量超过了运载能力，需要新的一天
        if (currentWeight + weight > capacity) {
            dayCount++;
            currentWeight = weight;  // 新的一天从当前包裹开始

            // 如果天数已经超过了限制，可以提前返回 false
            if (dayCount > days) {
                return false;
            }
        } else {
            currentWeight += weight;  // 继续往当前天添加包裹
        }
    }
```

```
    return dayCount <= days;
}
```

C++ 实现
```cpp
/**
 * LeetCode 1011. 在D天内送达包裹的能力
 *
 * 时间复杂度: O(n log totalWeight)
 * 空间复杂度: O(1)
 *
 * @param weights 包裹重量数组
 * @param days 天数限制
 * @return 船的最小运载能力
 */
int shipWithinDays(vector<int>& weights, int days) {
    int left = 0;  // 最小运载能力：最大的单个包裹重量
    int right = 0; // 最大运载能力：所有包裹的总重量

    for (int weight : weights) {
        left = max(left, weight);
        right += weight;
    }

    while (left < right) {
        int mid = left + ((right - left) >> 1);

        if (canShipInDays(weights, days, mid)) {
            right = mid;  // 可以在days天内运完，尝试更小的运载能力
        } else {
            left = mid + 1;  // 不能在days天内运完，需要更大的运载能力
        }
    }

    return left;
}

/**
 * 辅助方法：判断以capacity的运载能力是否能在days天内运完所有包裹
 */
bool canShipInDays(vector<int>& weights, int days, int capacity) {
```

```
    int currentWeight = 0;
    int dayCount = 1;  // 至少需要1天

    for (int weight : weights) {
        // 如果当前包裹的重量已经超过了运载能力，不可能运完
        if (weight > capacity) {
            return false;
        }

        // 如果当前累计重量加上当前包裹的重量超过了运载能力，需要新的一天
        if (currentWeight + weight > capacity) {
            dayCount++;
            currentWeight = weight;  // 新的一天从当前包裹开始

            // 如果天数已经超过了限制，可以提前返回 false
            if (dayCount > days) {
                return false;
            }
        } else {
            currentWeight += weight;  // 继续往当前天添加包裹
        }
    }

    return dayCount <= days;
}
```

Python 实现
```python
def ship_within_days(weights, days):
    """
    LeetCode 1011. 在D天内送达包裹的能力

    时间复杂度: O(n log totalWeight)
    空间复杂度: O(1)

    :param weights: 包裹重量数组
    :param days: 天数限制
    :return: 船的最小运载能力
    """
    left = max(weights)  # 最小运载能力：最大的单个包裹重量
    right = sum(weights)  # 最大运载能力：所有包裹的总重量
```

```
    while left < right:
        mid = left + ((right - left) >> 1)

        if can_ship_in_days(weights, days, mid):
            right = mid  # 可以在 days 天内运完，尝试更小的运载能力
        else:
            left = mid + 1  # 不能在 days 天内运完，需要更大的运载能力

    return left

def can_ship_in_days(weights, days, capacity):
    """
    辅助方法：判断以 capacity 的运载能力是否能在 days 天内运完所有包裹
    """
    current_weight = 0
    day_count = 1  # 至少需要 1 天

    for weight in weights:
        # 如果当前包裹的重量已经超过了运载能力，不可能运完
        if weight > capacity:
            return False

        # 如果当前累计重量加上当前包裹的重量超过了运载能力，需要新的一天
        if current_weight + weight > capacity:
            day_count += 1
            current_weight = weight  # 新的一天从当前包裹开始

            # 如果天数已经超过了限制，可以提前返回 False
            if day_count > days:
                return False
        else:
            current_weight += weight  # 继续往当前天添加包裹

    return day_count <= days
```

17. AtCoder ABC023D - 射撃王 (Shooting King)

题目描述
在一个二维平面上，有 N 个目标。第 i 个目标位于坐标 (x_i, y_i)。你的任务是用最少的子弹数射击所有目标。子弹每次可以以任意角度发射，但每次发射的子弹将沿着直线飞行，并且会击中所有在这条直线上的目标。

解题思路
- 这是一个几何问题，但可以转化为二分答案问题
- 我们需要找到最少需要多少条直线才能覆盖所有点
- 使用二分答案的方法，假设我们可以用 k 条直线覆盖所有点，然后验证这个假设是否成立
- 验证方法：选择一个点，尝试用一条直线覆盖尽可能多的点，然后递归地处理剩余的点

时间复杂度
- O(N^3 log N)，其中 N 是目标的数量
- 二分查找需要 log N 次迭代，每次验证需要 O(N^3) 的时间

空间复杂度
- O(N)，用于存储已覆盖的点

Java 实现
```java
/**
 * AtCoder ABC023D - Shooting King
 *
 * 时间复杂度: O(N^3 log N)
 * 空间复杂度: O(N)
 *
 * @param points 目标点数组，每个点是[x, y]
 * @return 最少需要的子弹数
 */
public int minBullets(int[][] points) {
    int n = points.length;
    if (n == 0) return 0;
    if (n == 1) return 1;

    int left = 1, right = n;
    while (left < right) {
        int mid = left + ((right - left) >> 1);
        if (canCover(points, mid)) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    return left;
}

/**
 * 辅助方法：判断是否可以用 k 条直线覆盖所有点
```

```
 */
private boolean canCover(int[][] points, int k) {
    boolean[] covered = new boolean[points.length];
    return coverRecursive(points, covered, 0, k);
}

/**
 * 递归辅助方法：尝试覆盖剩余的点
 */
private boolean coverRecursive(int[][] points, boolean[] covered, int coveredCount, int
remainingLines) {
    int n = points.length;
    if (coveredCount == n) return true;
    if (remainingLines == 0) return false;

    // 找到第一个未覆盖的点
    int first = -1;
    for (int i = 0; i < n; i++) {
        if (!covered[i]) {
            first = i;
            break;
        }
    }

    // 尝试通过第一个未覆盖的点画一条直线
    for (int i = 0; i < n; i++) {
        if (i == first || covered[i]) continue;

        // 标记所有在这条直线上的点
        boolean[] newCovered = Arrays.copyOf(covered, n);
        int newCount = coveredCount;

        for (int j = 0; j < n; j++) {
            if (!newCovered[j] && isCollinear(points[first], points[i], points[j])) {
                newCovered[j] = true;
                newCount++;
            }
        }

        if (coverRecursive(points, newCovered, newCount, remainingLines - 1)) {
            return true;
        }
    }
```

```
    // 如果没有其他点，可以单独用一条直线覆盖第一个未覆盖的点
    return coverRecursive(points, covered, coveredCount + 1, remainingLines - 1);
}

/**
 * 辅助方法：判断三个点是否共线
 */
private boolean isCollinear(int[] p1, int[] p2, int[] p3) {
    // 使用叉积判断三点共线
    // (y2 - y1) * (x3 - x1) == (y3 - y1) * (x2 - x1)
    return (p2[1] - p1[1]) * (p3[0] - p1[0]) == (p3[1] - p1[1]) * (p2[0] - p1[0]);
}
```

C++ 实现
```cpp
/**
 * AtCoder ABC023D - Shooting King
 *
 * 时间复杂度: O(N^3 log N)
 * 空间复杂度: O(N)
 *
 * @param points 目标点数组，每个点是[x, y]
 * @return 最少需要的子弹数
 */
int minBullets(vector<pair<int, int>>& points) {
    int n = points.size();
    if (n == 0) return 0;
    if (n == 1) return 1;

    int left = 1, right = n;
    while (left < right) {
        int mid = left + ((right - left) >> 1);
        if (canCover(points, mid)) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    return left;
}
```

```
/**
 * 辅助方法：判断是否可以用 k 条直线覆盖所有点
 */
bool canCover(vector<pair<int, int>>& points, int k) {
    vector<bool> covered(points.size(), false);
    return coverRecursive(points, covered, 0, k);
}

/**
 * 递归辅助方法：尝试覆盖剩余的点
 */
bool coverRecursive(vector<pair<int, int>>& points, vector<bool>& covered, int coveredCount, int
remainingLines) {
    int n = points.size();
    if (coveredCount == n) return true;
    if (remainingLines == 0) return false;

    // 找到第一个未覆盖的点
    int first = -1;
    for (int i = 0; i < n; i++) {
        if (!covered[i]) {
            first = i;
            break;
        }
    }

    // 尝试通过第一个未覆盖的点画一条直线
    for (int i = 0; i < n; i++) {
        if (i == first || covered[i]) continue;

        // 标记所有在这条直线上的点
        vector<bool> newCovered = covered;
        int newCount = coveredCount;

        for (int j = 0; j < n; j++) {
            if (!newCovered[j] && isCollinear(points[first], points[i], points[j])) {
                newCovered[j] = true;
                newCount++;
            }
        }

        if (coverRecursive(points, newCovered, newCount, remainingLines - 1)) {
            return true;
```

```
        }
    }

    // 如果没有其他点，可以单独用一条直线覆盖第一个未覆盖的点
    covered[first] = true;
    bool result = coverRecursive(points, covered, coveredCount + 1, remainingLines - 1);
    covered[first] = false;  // 回溯
    return result;
}

/**
 * 辅助方法：判断三个点是否共线
 */
bool isCollinear(pair<int, int>& p1, pair<int, int>& p2, pair<int, int>& p3) {
    // 使用叉积判断三点共线
    // (y2 - y1) * (x3 - x1) == (y3 - y1) * (x2 - x1)
    return (long long)(p2.second - p1.second) * (p3.first - p1.first) ==
           (long long)(p3.second - p1.second) * (p2.first - p1.first);
}
```

Python 实现
```python
def min_bullets(points):
    """
    AtCoder ABC023D - Shooting King

    时间复杂度: O(N^3 log N)
    空间复杂度: O(N)

    :param points: 目标点数组，每个点是(x, y)
    :return: 最少需要的子弹数
    """
    n = len(points)
    if n == 0:
        return 0
    if n == 1:
        return 1

    left, right = 1, n
    while left < right:
        mid = left + ((right - left) >> 1)
        if can_cover(points, mid):
```

```
            right = mid
        else:
            left = mid + 1

    return left

def can_cover(points, k):
    """
    辅助方法：判断是否可以用 k 条直线覆盖所有点
    """
    covered = [False] * len(points)
    return cover_recursive(points, covered, 0, k)

def cover_recursive(points, covered, covered_count, remaining_lines):
    """
    递归辅助方法：尝试覆盖剩余的点
    """
    n = len(points)
    if covered_count == n:
        return True
    if remaining_lines == 0:
        return False

    # 找到第一个未覆盖的点
    first = -1
    for i in range(n):
        if not covered[i]:
            first = i
            break

    # 尝试通过第一个未覆盖的点画一条直线
    for i in range(n):
        if i == first or covered[i]:
            continue

        # 标记所有在这条直线上的点
        new_covered = covered.copy()
        new_count = covered_count

        for j in range(n):
            if not new_covered[j] and is_collinear(points[first], points[i], points[j]):
                new_covered[j] = True
                new_count += 1
```

```
        if cover_recursive(points, new_covered, new_count, remaining_lines - 1):
            return True

    # 如果没有其他点，可以单独用一条直线覆盖第一个未覆盖的点
    covered[first] = True
    result = cover_recursive(points, covered, covered_count + 1, remaining_lines - 1)
    covered[first] = False  # 回溯
    return result

def is_collinear(p1, p2, p3):
    """
    辅助方法：判断三个点是否共线
    """
    # 使用叉积判断三点共线
    # (y2 - y1) * (x3 - x1) == (y3 - y1) * (x2 - x1)
    return (p2[1] - p1[1]) * (p3[0] - p1[0]) == (p3[1] - p1[1]) * (p2[0] - p1[0])
```

18. 二分查找算法的更多变体与应用

二分查找的扩展应用

1. 二分答案法

二分答案是一种非常强大的技巧，特别适用于**求满足条件的最大/最小值**的问题。其基本思路是：

1. 确定可能的答案范围 [left, right]
2. 在这个范围内进行二分查找
3. 对于每个中间值 mid，判断是否满足题目条件
4. 根据判断结果调整搜索范围

这种方法的关键在于**如何高效地判断一个候选答案是否满足条件**。

例如，在 LeetCode 875（爱吃香蕉的珂珂）和 LeetCode 1011（在 D 天内送达包裹的能力）中，我们就使用了二分答案的方法。

2. 二分查找与贪心结合

在 SPOJ 的 AGGRCOW 问题中，我们结合了二分查找和贪心算法：

1. 使用二分查找来猜测最大的最小距离
2. 使用贪心算法来验证是否可以按照这个距离放置所有奶牛

这种结合是解决优化问题的常用策略。

3. 二分查找与其他数据结构结合

二分查找可以与其他数据结构结合使用，例如：

- **二分查找与哈希表结合**：在某些情况下，可以先用哈希表预处理数据，然后使用二分查找进行查询
- **二分查找与前缀和结合**：对于某些区间查询问题，可以使用前缀和预处理，然后使用二分查找加速查询

二分查找的常见陷阱

1. **整数溢出**：在计算 mid = (left + right) / 2 时，可能会发生整数溢出。正确的写法是 mid = left + ((right - left) >> 1)

2. **死循环**：在循环条件和边界处理时，如果不小心，可能会导致死循环。通常使用 `while (left < right)` 或 `while (left <= right)` 作为循环条件，具体取决于问题。

3. **边界处理**：在调整 left 和 right 时，需要注意是否应该包含 mid。通常的做法是：
   - 如果 `mid` 可能是答案，则 `right = mid`
   - 否则，`left = mid + 1` 或 `right = mid - 1`

4. **浮点数精度**：对于浮点数的二分查找，需要注意精度问题，通常设置一个很小的 epsilon（例如 1e-7）来控制循环结束。

二分查找的高级应用场景

1. **多维二分查找**：在二维或更高维的空间中进行二分查找

2. **旋转数组中的二分查找**：如 LeetCode 33 和 LeetCode 153，在部分有序的数组中进行二分查找

3. **二分查找在实际系统中的应用**：
   - 在数据库索引中查找数据
   - 在缓存系统中查找元素
   - 在排序算法中优化某些操作

19. 牛客网 - 二分查找变体题目

19.1 牛客网 - 旋转数组的最小数字

题目描述：
把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。输入一个非递减排序的数组的一个旋转，输出旋转数组的最小元素。

解题思路：
- 使用二分查找，比较中间元素与右边界元素
- 如果中间元素小于右边界元素，说明最小值在左半部分
- 如果中间元素大于右边界元素，说明最小值在右半部分
- 如果相等，则右边界左移一位

时间复杂度：O(log n)，最坏情况 O(n)
空间复杂度：O(1)

Java 实现
```java
/**
 * 牛客网 - 旋转数组的最小数字
 *
 * 时间复杂度: O(log n)，最坏情况 O(n)
 * 空间复杂度: O(1)
 *
 * @param array 旋转数组
 * @return 最小元素
 */
public int minNumberInRotateArray(int[] array) {
    if (array == null || array.length == 0) {
        return 0;
    }

    int left = 0, right = array.length - 1;

    while (left < right) {
        int mid = left + ((right - left) >> 1);

        if (array[mid] > array[right]) {
            // 中间元素大于右边界，最小值在右半部分
            left = mid + 1;
        } else if (array[mid] < array[right]) {
            // 中间元素小于右边界，最小值在左半部分
            right = mid;
        } else {
            // 中间元素等于右边界，无法判断，右边界左移
            right--;
        }
    }
```

```
    return array[left];
}
```

C++ 实现
```cpp
/**
 * 牛客网 - 旋转数组的最小数字
 *
 * 时间复杂度: O(log n)，最坏情况O(n)
 * 空间复杂度: O(1)
 *
 * @param array 旋转数组
 * @return 最小元素
 */
int minNumberInRotateArray(vector<int>& array) {
    if (array.empty()) {
        return 0;
    }

    int left = 0, right = array.size() - 1;

    while (left < right) {
        int mid = left + ((right - left) >> 1);

        if (array[mid] > array[right]) {
            // 中间元素大于右边界，最小值在右半部分
            left = mid + 1;
        } else if (array[mid] < array[right]) {
            // 中间元素小于右边界，最小值在左半部分
            right = mid;
        } else {
            // 中间元素等于右边界，无法判断，右边界左移
            right--;
        }
    }

    return array[left];
}
```

Python 实现
```python
```

```
def min_number_in_rotate_array(array):
    """
    牛客网 - 旋转数组的最小数字

    时间复杂度: O(log n)，最坏情况O(n)
    空间复杂度: O(1)

    :param array: 旋转数组
    :return: 最小元素
    """
    if not array:
        return 0

    left, right = 0, len(array) - 1

    while left < right:
        mid = left + ((right - left) >> 1)

        if array[mid] > array[right]:
            # 中间元素大于右边界，最小值在右半部分
            left = mid + 1
        elif array[mid] < array[right]:
            # 中间元素小于右边界，最小值在左半部分
            right = mid
        else:
            # 中间元素等于右边界，无法判断，右边界左移
            right -= 1

    return array[left]
```

19.2 牛客网 - 数字在排序数组中出现的次数

题目描述：
统计一个数字在排序数组中出现的次数。

解题思路：
- 使用二分查找找到目标数字的第一次出现位置和最后一次出现位置
- 出现次数 = 最后一次位置 - 第一次位置 + 1

时间复杂度：O(log n)
空间复杂度：O(1)

Java 实现

```java
/**
 * 牛客网 - 数字在排序数组中出现的次数
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param array 排序数组
 * @param k 目标数字
 * @return 出现次数
 */
public int getNumberOfK(int[] array, int k) {
    if (array == null || array.length == 0) {
        return 0;
    }

    int first = findFirst(array, k);
    if (first == -1) {
        return 0;
    }

    int last = findLast(array, k);
    return last - first + 1;
}

/**
 * 查找目标数字第一次出现的位置
 */
private int findFirst(int[] array, int k) {
    int left = 0, right = array.length - 1;
    int result = -1;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);

        if (array[mid] >= k) {
            if (array[mid] == k) {
                result = mid;
            }
            right = mid - 1;
        } else {
            left = mid + 1;
```

```
        }
    }

    return result;
}

/**
 * 查找目标数字最后一次出现的位置
 */
private int findLast(int[] array, int k) {
    int left = 0, right = array.length - 1;
    int result = -1;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);

        if (array[mid] <= k) {
            if (array[mid] == k) {
                result = mid;
            }
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return result;
}
```

C++ 实现

```cpp
/**
 * 牛客网 - 数字在排序数组中出现的次数
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param array 排序数组
 * @param k 目标数字
 * @return 出现次数
 */
int getNumberOfK(vector<int>& array, int k) {
```

```
    if (array.empty()) {
        return 0;
    }

    int first = findFirst(array, k);
    if (first == -1) {
        return 0;
    }

    int last = findLast(array, k);
    return last - first + 1;
}

/**
 * 查找目标数字第一次出现的位置
 */
int findFirst(vector<int>& array, int k) {
    int left = 0, right = array.size() - 1;
    int result = -1;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);

        if (array[mid] >= k) {
            if (array[mid] == k) {
                result = mid;
            }
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    return result;
}

/**
 * 查找目标数字最后一次出现的位置
 */
int findLast(vector<int>& array, int k) {
    int left = 0, right = array.size() - 1;
    int result = -1;
```

```
    while (left <= right) {
        int mid = left + ((right - left) >> 1);

        if (array[mid] <= k) {
            if (array[mid] == k) {
                result = mid;
            }
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return result;
}
```

Python 实现
```python
def get_number_of_k(array, k):
    """
    牛客网 - 数字在排序数组中出现的次数

    时间复杂度: O(log n)
    空间复杂度: O(1)

    :param array: 排序数组
    :param k: 目标数字
    :return: 出现次数
    """
    if not array:
        return 0

    first = find_first(array, k)
    if first == -1:
        return 0

    last = find_last(array, k)
    return last - first + 1

def find_first(array, k):
    """
    查找目标数字第一次出现的位置
```

```
    """
    left, right = 0, len(array) - 1
    result = -1

    while left <= right:
        mid = left + ((right - left) >> 1)

        if array[mid] >= k:
            if array[mid] == k:
                result = mid
            right = mid - 1
        else:
            left = mid + 1

    return result

def find_last(array, k):
    """
    查找目标数字最后一次出现的位置
    """
    left, right = 0, len(array) - 1
    result = -1

    while left <= right:
        mid = left + ((right - left) >> 1)

        if array[mid] <= k:
            if array[mid] == k:
                result = mid
            left = mid + 1
        else:
            right = mid - 1

    return result
```

20. 剑指 Offer - 二分查找相关题目

20.1 剑指 Offer 11 - 旋转数组的最小数字

题目描述：
把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。输入一个递增排序的数组的一个旋转，输出旋转数组的最小元素。

解题思路：
- 与牛客网题目类似，使用二分查找
- 注意处理重复元素的情况

时间复杂度：O(log n)，最坏情况O(n)
空间复杂度：O(1)

Java 实现
```java
/**
 * 剑指Offer 11 - 旋转数组的最小数字
 *
 * 时间复杂度: O(log n)，最坏情况O(n)
 * 空间复杂度: O(1)
 *
 * @param numbers 旋转数组
 * @return 最小元素
 */
public int minArray(int[] numbers) {
    if (numbers == null || numbers.length == 0) {
        throw new IllegalArgumentException("数组不能为空");
    }

    int left = 0, right = numbers.length - 1;

    while (left < right) {
        int mid = left + ((right - left) >> 1);

        if (numbers[mid] > numbers[right]) {
            // 中间元素大于右边界，最小值在右半部分
            left = mid + 1;
        } else if (numbers[mid] < numbers[right]) {
            // 中间元素小于右边界，最小值在左半部分
            right = mid;
        } else {
            // 中间元素等于右边界，无法判断，右边界左移
            right--;
        }
    }

    return numbers[left];
}
```

```

#### C++ 实现
```cpp
/**
 * 剑指 Offer 11 - 旋转数组的最小数字
 *
 * 时间复杂度: O(log n)，最坏情况 O(n)
 * 空间复杂度: O(1)
 *
 * @param numbers 旋转数组
 * @return 最小元素
 */
int minArray(vector<int>& numbers) {
    if (numbers.empty()) {
        throw invalid_argument("数组不能为空");
    }

    int left = 0, right = numbers.size() - 1;

    while (left < right) {
        int mid = left + ((right - left) >> 1);

        if (numbers[mid] > numbers[right]) {
            // 中间元素大于右边界，最小值在右半部分
            left = mid + 1;
        } else if (numbers[mid] < numbers[right]) {
            // 中间元素小于右边界，最小值在左半部分
            right = mid;
        } else {
            // 中间元素等于右边界，无法判断，右边界左移
            right--;
        }
    }

    return numbers[left];
}
```

Python 实现
```python
def min_array(numbers):
    """

```
    剑指 Offer 11 - 旋转数组的最小数字

    时间复杂度: O(log n)，最坏情况 O(n)
    空间复杂度: O(1)

    :param numbers: 旋转数组
    :return: 最小元素
    """
    if not numbers:
        raise ValueError("数组不能为空")

    left, right = 0, len(numbers) - 1

    while left < right:
        mid = left + ((right - left) >> 1)

        if numbers[mid] > numbers[right]:
            # 中间元素大于右边界，最小值在右半部分
            left = mid + 1
        elif numbers[mid] < numbers[right]:
            # 中间元素小于右边界，最小值在左半部分
            right = mid
        else:
            # 中间元素等于右边界，无法判断，右边界左移
            right -= 1

    return numbers[left]
```

20.2 剑指 Offer 53-I - 在排序数组中查找数字

题目描述：
统计一个数字在排序数组中出现的次数。

解题思路：
- 使用二分查找找到目标数字的左右边界
- 出现次数 = 右边界 - 左边界 - 1

时间复杂度：O(log n)
空间复杂度：O(1)

Java 实现
```java
```

```java
/**
 * 剑指 Offer 53-I - 在排序数组中查找数字
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param nums 排序数组
 * @param target 目标数字
 * @return 出现次数
 */
public int search(int[] nums, int target) {
    if (nums == null || nums.length == 0) {
        return 0;
    }

    // 查找右边界
    int right = findRightBound(nums, target);
    if (right == -1) {
        return 0;
    }

    // 查找左边界
    int left = findLeftBound(nums, target);

    return right - left + 1;
}

/**
 * 查找目标数字的左边界
 */
private int findLeftBound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);

        if (nums[mid] >= target) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }
```

```
    // 检查左边界是否有效
    if (left < nums.length && nums[left] == target) {
        return left;
    }
    return -1;
}

/**
 * 查找目标数字的右边界
 */
private int findRightBound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);

        if (nums[mid] <= target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    // 检查右边界是否有效
    if (right >= 0 && nums[right] == target) {
        return right;
    }
    return -1;
}
```

C++ 实现
```cpp
/**
 * 剑指 Offer 53-I - 在排序数组中查找数字
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param nums 排序数组
 * @param target 目标数字
 * @return 出现次数
 */
```

```
int search(vector<int>& nums, int target) {
    if (nums.empty()) {
        return 0;
    }

    // 查找右边界
    int right = findRightBound(nums, target);
    if (right == -1) {
        return 0;
    }

    // 查找左边界
    int left = findLeftBound(nums, target);

    return right - left + 1;
}

/**
 * 查找目标数字的左边界
 */
int findLeftBound(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) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    // 检查左边界是否有效
    if (left < nums.size() && nums[left] == target) {
        return left;
    }
    return -1;
}

/**
 * 查找目标数字的右边界
 */
```

```
int findRightBound(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) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    // 检查右边界是否有效
    if (right >= 0 && nums[right] == target) {
        return right;
    }
    return -1;
}
```

Python 实现
```python
def search(nums, target):
    """
    剑指 Offer 53-I - 在排序数组中查找数字

    时间复杂度: O(log n)
    空间复杂度: O(1)

    :param nums: 排序数组
    :param target: 目标数字
    :return: 出现次数
    """
    if not nums:
        return 0

    # 查找右边界
    right = find_right_bound(nums, target)
    if right == -1:
        return 0

    # 查找左边界
```

```
    left = find_left_bound(nums, target)

    return right - left + 1

def find_left_bound(nums, target):
    """
    查找目标数字的左边界
    """
    left, right = 0, len(nums) - 1

    while left <= right:
        mid = left + ((right - left) >> 1)

        if nums[mid] >= target:
            right = mid - 1
        else:
            left = mid + 1

    # 检查左边界是否有效
    if left < len(nums) and nums[left] == target:
        return left
    return -1

def find_right_bound(nums, target):
    """
    查找目标数字的右边界
    """
    left, right = 0, len(nums) - 1

    while left <= right:
        mid = left + ((right - left) >> 1)

        if nums[mid] <= target:
            left = mid + 1
        else:
            right = mid - 1

    # 检查右边界是否有效
    if right >= 0 and nums[right] == target:
        return right
    return -1
```

21. HackerRank - 二分查找题目

21.1 HackerRank - Ice Cream Parlor

题目描述：
给定一个整数数组表示冰淇淋的价格，以及一个整数表示总金额。找到两个不同位置的冰淇淋，它们的价格之和等于总金额。

解题思路：
- 先对数组进行排序，但需要记录原始索引
- 使用二分查找来寻找匹配的价格
- 或者使用哈希表来优化查找过程

时间复杂度：O(n log n)
空间复杂度：O(n)

Java 实现
```java
/**
 * HackerRank - Ice Cream Parlor
 *
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 *
 * @param cost 冰淇淋价格数组
 * @param money 总金额
 * @return 两个冰淇淋的索引（1-based）
 */
public int[] iceCreamParlor(int money, int[] cost) {
    // 创建包含索引的价格对数组
    List<int[]> priceWithIndex = new ArrayList<>();
    for (int i = 0; i < cost.length; i++) {
        priceWithIndex.add(new int[]{cost[i], i});
    }

    // 按价格排序
    priceWithIndex.sort((a, b) -> Integer.compare(a[0], b[0]));

    for (int i = 0; i < priceWithIndex.size(); i++) {
        int currentPrice = priceWithIndex.get(i)[0];
        int remaining = money - currentPrice;

        // 使用二分查找寻找剩余金额
```

```
        int left = i + 1, right = priceWithIndex.size() - 1;
        while (left <= right) {
            int mid = left + ((right - left) >> 1);
            int midPrice = priceWithIndex.get(mid)[0];

            if (midPrice == remaining) {
                // 找到匹配的价格
                int index1 = priceWithIndex.get(i)[1] + 1;  // 1-based索引
                int index2 = priceWithIndex.get(mid)[1] + 1; // 1-based索引
                return new int[]{Math.min(index1, index2), Math.max(index1, index2)};
            } else if (midPrice < remaining) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
    }

    return new int[]{-1, -1};  // 未找到
}
```

C++ 实现
```cpp
/**
 * HackerRank - Ice Cream Parlor
 *
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 *
 * @param cost 冰淇淋价格数组
 * @param money 总金额
 * @return 两个冰淇淋的索引（1-based）
 */
vector<int> iceCreamParlor(int money, vector<int>& cost) {
    // 创建包含索引的价格对数组
    vector<pair<int, int>> priceWithIndex;
    for (int i = 0; i < cost.size(); i++) {
        priceWithIndex.push_back({cost[i], i});
    }

    // 按价格排序
    sort(priceWithIndex.begin(), priceWithIndex.end());
```

```
    for (int i = 0; i < priceWithIndex.size(); i++) {
        int currentPrice = priceWithIndex[i].first;
        int remaining = money - currentPrice;

        // 使用二分查找寻找剩余金额
        int left = i + 1, right = priceWithIndex.size() - 1;
        while (left <= right) {
            int mid = left + ((right - left) >> 1);
            int midPrice = priceWithIndex[mid].first;

            if (midPrice == remaining) {
                // 找到匹配的价格
                int index1 = priceWithIndex[i].second + 1;  // 1-based索引
                int index2 = priceWithIndex[mid].second + 1; // 1-based索引
                return {min(index1, index2), max(index1, index2)};
            } else if (midPrice < remaining) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
    }

    return {-1, -1};  // 未找到
}
```

Python 实现
```python
def ice_cream_parlor(money, cost):
    """
    HackerRank - Ice Cream Parlor

    时间复杂度: O(n log n)
    空间复杂度: O(n)

    :param cost: 冰淇淋价格数组
    :param money: 总金额
    :return: 两个冰淇淋的索引（1-based）
    """
    # 创建包含索引的价格对数组
    price_with_index = [(price, idx) for idx, price in enumerate(cost)]
```

```
    # 按价格排序
    price_with_index.sort()

    for i in range(len(price_with_index)):
        current_price, current_idx = price_with_index[i]
        remaining = money - current_price

        # 使用二分查找寻找剩余金额
        left, right = i + 1, len(price_with_index) - 1
        while left <= right:
            mid = left + ((right - left) >> 1)
            mid_price, mid_idx = price_with_index[mid]

            if mid_price == remaining:
                # 找到匹配的价格
                index1 = current_idx + 1  # 1-based索引
                index2 = mid_idx + 1      # 1-based索引
                return sorted([index1, index2])
            elif mid_price < remaining:
                left = mid + 1
            else:
                right = mid - 1

    return [-1, -1]  # 未找到
```

22. 杭电OJ (HDU) - 二分查找题目

22.1 HDU 2141 - Can you find it?

题目描述：
给定三个数组A、B、C，以及多个查询X。对于每个查询，判断是否存在a∈A, b∈B, c∈C，使得a+b+c=X。

解题思路：
- 将A+B的所有可能和存储在一个数组中
- 对这个和数组进行排序
- 对于每个查询X，使用二分查找在A+B的和数组中查找是否存在X-c（c∈C）

时间复杂度：O(L*M + N*log(L*M))
空间复杂度：O(L*M)

Java 实现

```java
/**
 * HDU 2141 - Can you find it?
 *
 * 时间复杂度: O(L*M + N*log(L*M))
 * 空间复杂度: O(L*M)
 *
 * @param A 数组 A
 * @param B 数组 B
 * @param C 数组 C
 * @param queries 查询数组
 * @return 每个查询的结果（true/false）
 */
public boolean[] canFindIt(int[] A, int[] B, int[] C, int[] queries) {
    // 计算 A+B 的所有可能和
    List<Integer> sumAB = new ArrayList<>();
    for (int a : A) {
        for (int b : B) {
            sumAB.add(a + b);
        }
    }

    // 排序 A+B 的和数组
    Collections.sort(sumAB);

    boolean[] results = new boolean[queries.length];

    for (int i = 0; i < queries.length; i++) {
        int X = queries[i];
        boolean found = false;

        // 对于每个 c∈C，在 A+B 的和数组中查找 X-c
        for (int c : C) {
            int target = X - c;

            // 使用二分查找
            int left = 0, right = sumAB.size() - 1;
            while (left <= right) {
                int mid = left + ((right - left) >> 1);
                int midVal = sumAB.get(mid);

                if (midVal == target) {
                    found = true;
```

```
                    break;
                } else if (midVal < target) {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }

            if (found) break;
        }

        results[i] = found;
    }

    return results;
}
```

C++ 实现
```cpp
/**
 * HDU 2141 - Can you find it?
 *
 * 时间复杂度: O(L*M + N*log(L*M))
 * 空间复杂度: O(L*M)
 *
 * @param A 数组A
 * @param B 数组B
 * @param C 数组C
 * @param queries 查询数组
 * @return 每个查询的结果（true/false）
 */
vector<bool> canFindIt(vector<int>& A, vector<int>& B, vector<int>& C, vector<int>& queries) {
    // 计算A+B的所有可能和
    vector<int> sumAB;
    for (int a : A) {
        for (int b : B) {
            sumAB.push_back(a + b);
        }
    }

    // 排序A+B的和数组
    sort(sumAB.begin(), sumAB.end());
```

```
    vector<bool> results(queries.size(), false);

    for (int i = 0; i < queries.size(); i++) {
        int X = queries[i];
        bool found = false;

        // 对于每个 c∈C，在 A+B 的和数组中查找 X-c
        for (int c : C) {
            int target = X - c;

            // 使用二分查找
            int left = 0, right = sumAB.size() - 1;
            while (left <= right) {
                int mid = left + ((right - left) >> 1);
                int midVal = sumAB[mid];

                if (midVal == target) {
                    found = true;
                    break;
                } else if (midVal < target) {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }

            if (found) break;
        }

        results[i] = found;
    }

    return results;
}
```

Python 实现
```python
def can_find_it(A, B, C, queries):
    """
    HDU 2141 - Can you find it?
```

```
时间复杂度: O(L*M + N*log(L*M))
空间复杂度: O(L*M)

:param A: 数组 A
:param B: 数组 B
:param C: 数组 C
:param queries: 查询数组
:return: 每个查询的结果（True/False）
"""
# 计算 A+B 的所有可能和
sum_ab = []
for a in A:
    for b in B:
        sum_ab.append(a + b)

# 排序 A+B 的和数组
sum_ab.sort()

results = [False] * len(queries)

for i, X in enumerate(queries):
    found = False

    # 对于每个 c∈C，在 A+B 的和数组中查找 X-c
    for c in C:
        target = X - c

        # 使用二分查找
        left, right = 0, len(sum_ab) - 1
        while left <= right:
            mid = left + ((right - left) >> 1)
            mid_val = sum_ab[mid]

            if mid_val == target:
                found = True
                break
            elif mid_val < target:
                left = mid + 1
            else:
                right = mid - 1

        if found:
            break
```

```
        results[i] = found

    return results
```

23. POJ (北京大学 OJ) - 二分查找题目

23.1 POJ 2456 - Aggressive cows

题目描述：
农夫约翰建造了一个有 N 个隔间的牛棚，这些隔间分布在一条直线上，坐标是 x1,...,xN。他的 C 头牛不满于隔间的大小，经常互相攻击。约翰为了防止牛之间互相伤害，决定把每头牛都放在离其他牛尽可能远的隔间。也就是要最大化最近的两头牛之间的距离。

解题思路：
- 这是一个典型的二分答案问题
- 先对隔间坐标进行排序
- 使用二分查找猜测最大的最小距离
- 对于每个候选距离，使用贪心算法验证是否可以放置所有牛

时间复杂度：O(N log(max-min))
空间复杂度：O(1)

Java 实现
```java
/**
 * POJ 2456 - Aggressive cows
 *
 * 时间复杂度: O(N log(max-min))
 * 空间复杂度: O(1)
 *
 * @param stalls 隔间坐标数组
 * @param cows 牛的数量
 * @return 最大的最小距离
 */
public int aggressiveCows(int[] stalls, int cows) {
    if (stalls == null || stalls.length == 0 || cows <= 0) {
        return 0;
    }

    Arrays.sort(stalls);
```

```
    int left = 0;
    int right = stalls[stalls.length - 1] - stalls[0];
    int result = 0;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);

        if (canPlaceCows(stalls, cows, mid)) {
            result = mid;
            left = mid + 1;  // 尝试更大的距离
        } else {
            right = mid - 1;  // 距离太大，需要减小
        }
    }

    return result;
}

/**
 * 验证是否可以在给定的最小距离下放置所有牛
 */
private boolean canPlaceCows(int[] stalls, int cows, int minDist) {
    int count = 1;  // 第一头牛放在第一个隔间
    int lastPosition = stalls[0];

    for (int i = 1; i < stalls.length; i++) {
        if (stalls[i] - lastPosition >= minDist) {
            count++;
            lastPosition = stalls[i];

            if (count >= cows) {
                return true;
            }
        }
    }

    return false;
}
```

C++ 实现

```cpp
/**
```

```
 * POJ 2456 - Aggressive cows
 *
 * 时间复杂度: O(N log(max-min))
 * 空间复杂度: O(1)
 *
 * @param stalls 隔间坐标数组
 * @param cows 牛的数量
 * @return 最大的最小距离
 */
int aggressiveCows(vector<int>& stalls, int cows) {
    if (stalls.empty() || cows <= 0) {
        return 0;
    }

    sort(stalls.begin(), stalls.end());

    int left = 0;
    int right = stalls.back() - stalls.front();
    int result = 0;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);

        if (canPlaceCows(stalls, cows, mid)) {
            result = mid;
            left = mid + 1;  // 尝试更大的距离
        } else {
            right = mid - 1;  // 距离太大，需要减小
        }
    }

    return result;
}

/**
 * 验证是否可以在给定的最小距离下放置所有牛
 */
bool canPlaceCows(vector<int>& stalls, int cows, int minDist) {
    int count = 1;  // 第一头牛放在第一个隔间
    int lastPosition = stalls[0];

    for (int i = 1; i < stalls.size(); i++) {
        if (stalls[i] - lastPosition >= minDist) {
```

```
            count++;
            lastPosition = stalls[i];

            if (count >= cows) {
                return true;
            }
        }
    }

    return false;
}
```

Python 实现
```python
def aggressive_cows(stalls, cows):
    """
    POJ 2456 - Aggressive cows

    时间复杂度: O(N log(max-min))
    空间复杂度: O(1)

    :param stalls: 隔间坐标数组
    :param cows: 牛的数量
    :return: 最大的最小距离
    """
    if not stalls or cows <= 0:
        return 0

    stalls.sort()

    left, right = 0, stalls[-1] - stalls[0]
    result = 0

    while left <= right:
        mid = left + ((right - left) >> 1)

        if can_place_cows(stalls, cows, mid):
            result = mid
            left = mid + 1  # 尝试更大的距离
        else:
            right = mid - 1  # 距离太大，需要减小
```

```
    return result

def can_place_cows(stalls, cows, min_dist):
    """
    验证是否可以在给定的最小距离下放置所有牛
    """
    count = 1  # 第一头牛放在第一个隔间
    last_position = stalls[0]

    for i in range(1, len(stalls)):
        if stalls[i] - last_position >= min_dist:
            count += 1
            last_position = stalls[i]

            if count >= cows:
                return True

    return False
```

24. 洛谷 (Luogu) - 二分查找题目

24.1 洛谷 P2678 - 跳石头

题目描述:
一年一度的"跳石头"比赛又要开始了！这项比赛将在一条笔直的河道中进行，河道中分布着一些巨大岩石。组委会已经选择好了两块岩石作为比赛起点和终点。在起点和终点之间，有 N 块岩石（不含起点和终点的岩石）。在比赛过程中，选手们将从起点出发，每一步跳向相邻的岩石，直至到达终点。为了提高比赛难度，组委会计划移走一些岩石，使得选手们在比赛过程中的最短跳跃距离尽可能长。由于预算限制，组委会至多从起点和终点之间移走 M 块岩石（不能移走起点和终点的岩石）。

解题思路:
- 二分答案问题，寻找最大的最短跳跃距离
- 对于每个候选距离，计算需要移走多少块岩石
- 如果需要移走的岩石数不超过 M，则可以尝试更大的距离

时间复杂度: O(N log L)，其中 L 是河道长度
空间复杂度: O(N)

Java 实现
```java
/**
 * 洛谷 P2678 - 跳石头
```

```
     *
     * 时间复杂度: O(N log L)
     * 空间复杂度: O(N)
     *
     * @param L 河道长度
     * @param N 岩石数量（不含起点终点）
     * @param M 最多可移走的岩石数
     * @param rocks 岩石位置数组
     * @return 最大的最短跳跃距离
     */
    public int jumpStones(int L, int N, int M, int[] rocks) {
        // 添加起点和终点
        int[] positions = new int[N + 2];
        positions[0] = 0;  // 起点
        System.arraycopy(rocks, 0, positions, 1, N);
        positions[N + 1] = L;  // 终点

        Arrays.sort(positions);

        int left = 1, right = L;
        int result = 0;

        while (left <= right) {
            int mid = left + ((right - left) >> 1);

            if (canJump(positions, M, mid)) {
                result = mid;
                left = mid + 1;  // 尝试更大的距离
            } else {
                right = mid - 1;  // 距离太大，需要减小
            }
        }

        return result;
    }

    /**
     * 验证是否可以在给定的最短跳跃距离下移走不超过 M 块岩石
     */
    private boolean canJump(int[] positions, int M, int minDist) {
        int removeCount = 0;
        int lastPosition = positions[0];
```

```
    for (int i = 1; i < positions.length; i++) {
        if (positions[i] - lastPosition < minDist) {
            // 需要移走当前岩石
            removeCount++;
            if (removeCount > M) {
                return false;
            }
        } else {
            lastPosition = positions[i];
        }
    }

    return removeCount <= M;
}
```

C++ 实现
```cpp
/**
 * 洛谷 P2678 - 跳石头
 *
 * 时间复杂度: O(N log L)
 * 空间复杂度: O(N)
 *
 * @param L 河道长度
 * @param N 岩石数量（不含起点终点）
 * @param M 最多可移走的岩石数
 * @param rocks 岩石位置数组
 * @return 最大的最短跳跃距离
 */
int jumpStones(int L, int N, int M, vector<int>& rocks) {
    // 添加起点和终点
    vector<int> positions(N + 2);
    positions[0] = 0;  // 起点
    copy(rocks.begin(), rocks.end(), positions.begin() + 1);
    positions[N + 1] = L;  // 终点

    sort(positions.begin(), positions.end());

    int left = 1, right = L;
    int result = 0;

    while (left <= right) {
```

```
        int mid = left + ((right - left) >> 1);

        if (canJump(positions, M, mid)) {
            result = mid;
            left = mid + 1;  // 尝试更大的距离
        } else {
            right = mid - 1;  // 距离太大，需要减小
        }
    }

    return result;
}

/**
 * 验证是否可以在给定的最短跳跃距离下移走不超过 M 块岩石
 */
bool canJump(vector<int>& positions, int M, int minDist) {
    int removeCount = 0;
    int lastPosition = positions[0];

    for (int i = 1; i < positions.size(); i++) {
        if (positions[i] - lastPosition < minDist) {
            // 需要移走当前岩石
            removeCount++;
            if (removeCount > M) {
                return false;
            }
        } else {
            lastPosition = positions[i];
        }
    }

    return removeCount <= M;
}
```

Python 实现
```python
def jump_stones(L, N, M, rocks):
    """
    洛谷 P2678 - 跳石头

    时间复杂度: O(N log L)
```

```
    空间复杂度: O(N)

    :param L: 河道长度
    :param N: 岩石数量（不含起点终点）
    :param M: 最多可移走的岩石数
    :param rocks: 岩石位置数组
    :return: 最大的最短跳跃距离
    """
    # 添加起点和终点
    positions = [0] + sorted(rocks) + [L]

    left, right = 1, L
    result = 0

    while left <= right:
        mid = left + ((right - left) >> 1)

        if can_jump(positions, M, mid):
            result = mid
            left = mid + 1  # 尝试更大的距离
        else:
            right = mid - 1  # 距离太大，需要减小

    return result

def can_jump(positions, M, min_dist):
    """
    验证是否可以在给定的最短跳跃距离下移走不超过 M 块岩石
    """
    remove_count = 0
    last_position = positions[0]

    for i in range(1, len(positions)):
        if positions[i] - last_position < min_dist:
            # 需要移走当前岩石
            remove_count += 1
            if remove_count > M:
                return False
        else:
            last_position = positions[i]

    return remove_count <= M
```

25. CodeChef - 二分查找题目

25.1 CodeChef - PIANO1

题目描述：
钢琴老师给小明布置了 N 个练习曲，每个练习曲有一个难度值。小明需要按照难度递增的顺序练习这些曲子。但是小明很懒，他只想练习一部分曲子。请问小明最多可以练习多少首曲子，使得这些曲子的难度是严格递增的？

解题思路：
- 最长递增子序列(LIS)问题
- 使用二分查找优化动态规划解法
- 维护一个 tails 数组，tails[i]表示长度为 i+1 的递增子序列的最小结尾值

时间复杂度：O(n log n)
空间复杂度：O(n)

Java 实现
```java
/**
 * CodeChef - PIANO1
 *
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 *
 * @param difficulties 练习曲难度数组
 * @return 最长递增子序列的长度
 */
public int pianoPractice(int[] difficulties) {
    if (difficulties == null || difficulties.length == 0) {
        return 0;
    }

    int[] tails = new int[difficulties.length];
    int size = 0;

    for (int difficulty : difficulties) {
        // 使用二分查找找到插入位置
        int left = 0, right = size;
        while (left < right) {
            int mid = left + ((right - left) >> 1);
            if (tails[mid] < difficulty) {
```

```
                left = mid + 1;
            } else {
                right = mid;
            }
        }

        tails[left] = difficulty;
        if (left == size) {
            size++;
        }
    }

    return size;
}
```

C++ 实现
```cpp
/**
 * CodeChef - PIANO1
 *
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 *
 * @param difficulties 练习曲难度数组
 * @return 最长递增子序列的长度
 */
int pianoPractice(vector<int>& difficulties) {
    if (difficulties.empty()) {
        return 0;
    }

    vector<int> tails(difficulties.size());
    int size = 0;

    for (int difficulty : difficulties) {
        // 使用二分查找找到插入位置
        int left = 0, right = size;
        while (left < right) {
            int mid = left + ((right - left) >> 1);
            if (tails[mid] < difficulty) {
                left = mid + 1;
            } else {
```

```
                right = mid;
            }
        }

        tails[left] = difficulty;
        if (left == size) {
            size++;
        }
    }

    return size;
}
```

Python 实现
```python
def piano_practice(difficulties):
    """
    CodeChef - PIANO1

    时间复杂度: O(n log n)
    空间复杂度: O(n)

    :param difficulties: 练习曲难度数组
    :return: 最长递增子序列的长度
    """
    if not difficulties:
        return 0

    tails = []
    size = 0

    for difficulty in difficulties:
        # 使用二分查找找到插入位置
        left, right = 0, size
        while left < right:
            mid = left + ((right - left) >> 1)
            if tails[mid] < difficulty:
                left = mid + 1
            else:
                right = mid

        if left == size:
```

```
            tails.append(difficulty)
            size += 1
        else:
            tails[left] = difficulty

    return size
```

26. AtCoder - 二分查找题目

26.1 AtCoder ABC023D - 射撃王 (Shooting King)

题目描述:
在一个二维平面上，有 N 个目标。第 i 个目标位于坐标 (x_i, y_i)。你的任务是用最少的子弹数射击所有目标。子弹每次可以以任意角度发射，但每次发射的子弹将沿着直线飞行，并且会击中所有在这条直线上的目标。

解题思路:
- 这是一个几何问题，但可以转化为二分答案问题
- 我们需要找到最少需要多少条直线才能覆盖所有点
- 使用二分答案的方法，假设我们可以用 k 条直线覆盖所有点，然后验证这个假设是否成立
- 验证方法：选择一个点，尝试用一条直线覆盖尽可能多的点，然后递归地处理剩余的点

时间复杂度: O(N^3 log N)，其中 N 是目标的数量
空间复杂度: O(N)，用于存储已覆盖的点

Java 实现
```java
/**
 * AtCoder ABC023D - Shooting King
 *
 * 时间复杂度: O(N^3 log N)
 * 空间复杂度: O(N)
 *
 * @param points 目标点数组，每个点是[x, y]
 * @return 最少需要的子弹数
 */
public int minBullets(int[][] points) {
    int n = points.length;
    if (n == 0) return 0;
    if (n == 1) return 1;

    int left = 1, right = n;
```

```
    while (left < right) {
        int mid = left + ((right - left) >> 1);
        if (canCover(points, mid)) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    return left;
}

/**
 * 辅助方法：判断是否可以用 k 条直线覆盖所有点
 */
private boolean canCover(int[][] points, int k) {
    boolean[] covered = new boolean[points.length];
    return coverRecursive(points, covered, 0, k);
}

/**
 * 递归辅助方法：尝试覆盖剩余的点
 */
private boolean coverRecursive(int[][] points, boolean[] covered, int coveredCount, int
remainingLines) {
    int n = points.length;
    if (coveredCount == n) return true;
    if (remainingLines == 0) return false;

    // 找到第一个未覆盖的点
    int first = -1;
    for (int i = 0; i < n; i++) {
        if (!covered[i]) {
            first = i;
            break;
        }
    }

    // 尝试通过第一个未覆盖的点画一条直线
    for (int i = 0; i < n; i++) {
        if (i == first || covered[i]) continue;

        // 标记所有在这条直线上的点
        boolean[] newCovered = Arrays.copyOf(covered, n);
```

```
        int newCount = coveredCount;

        for (int j = 0; j < n; j++) {
            if (!newCovered[j] && isCollinear(points[first], points[i], points[j])) {
                newCovered[j] = true;
                newCount++;
            }
        }

        if (coverRecursive(points, newCovered, newCount, remainingLines - 1)) {
            return true;
        }
    }

    // 如果没有其他点，可以单独用一条直线覆盖第一个未覆盖的点
    return coverRecursive(points, covered, coveredCount + 1, remainingLines - 1);
}

/**
 * 辅助方法：判断三个点是否共线
 */
private boolean isCollinear(int[] p1, int[] p2, int[] p3) {
    // 使用叉积判断三点共线
    // (y2 - y1) * (x3 - x1) == (y3 - y1) * (x2 - x1)
    return (p2[1] - p1[1]) * (p3[0] - p1[0]) == (p3[1] - p1[1]) * (p2[0] - p1[0]);
}
```

C++ 实现

```cpp
/**
 * AtCoder ABC023D - Shooting King
 *
 * 时间复杂度: O(N^3 log N)
 * 空间复杂度: O(N)
 *
 * @param points 目标点数组，每个点是[x, y]
 * @return 最少需要的子弹数
 */
int minBullets(vector<pair<int, int>>& points) {
    int n = points.size();
    if (n == 0) return 0;
    if (n == 1) return 1;
```

```
    int left = 1, right = n;
    while (left < right) {
        int mid = left + ((right - left) >> 1);
        if (canCover(points, mid)) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    return left;
}

/**
 * 辅助方法：判断是否可以用 k 条直线覆盖所有点
 */
bool canCover(vector<pair<int, int>>& points, int k) {
    vector<bool> covered(points.size(), false);
    return coverRecursive(points, covered, 0, k);
}

/**
 * 递归辅助方法：尝试覆盖剩余的点
 */
bool coverRecursive(vector<pair<int, int>>& points, vector<bool>& covered, int coveredCount, int
remainingLines) {
    int n = points.size();
    if (coveredCount == n) return true;
    if (remainingLines == 0) return false;

    // 找到第一个未覆盖的点
    int first = -1;
    for (int i = 0; i < n; i++) {
        if (!covered[i]) {
            first = i;
            break;
        }
    }

    // 尝试通过第一个未覆盖的点画一条直线
    for (int i = 0; i < n; i++) {
        if (i == first || covered[i]) continue;
```

```
        // 标记所有在这条直线上的点
        vector<bool> newCovered = covered;
        int newCount = coveredCount;

        for (int j = 0; j < n; j++) {
            if (!newCovered[j] && isCollinear(points[first], points[i], points[j])) {
                newCovered[j] = true;
                newCount++;
            }
        }

        if (coverRecursive(points, newCovered, newCount, remainingLines - 1)) {
            return true;
        }
    }

    // 如果没有其他点，可以单独用一条直线覆盖第一个未覆盖的点
    covered[first] = true;
    bool result = coverRecursive(points, covered, coveredCount + 1, remainingLines - 1);
    covered[first] = false;  // 回溯
    return result;
}

/**
 * 辅助方法：判断三个点是否共线
 */
bool isCollinear(pair<int, int>& p1, pair<int, int>& p2, pair<int, int>& p3) {
    // 使用叉积判断三点共线
    // (y2 - y1) * (x3 - x1) == (y3 - y1) * (x2 - x1)
    return (long long)(p2.second - p1.second) * (p3.first - p1.first) ==
           (long long)(p3.second - p1.second) * (p2.first - p1.first);
}
```

Python 实现
```python
def min_bullets(points):
    """
    AtCoder ABC023D - Shooting King

    时间复杂度: O(N^3 log N)
    空间复杂度: O(N)
```

```
    :param points: 目标点数组，每个点是(x, y)
    :return: 最少需要的子弹数
    """
    n = len(points)
    if n == 0:
        return 0
    if n == 1:
        return 1

    left, right = 1, n
    while left < right:
        mid = left + ((right - left) >> 1)
        if can_cover(points, mid):
            right = mid
        else:
            left = mid + 1

    return left

def can_cover(points, k):
    """
    辅助方法：判断是否可以用 k 条直线覆盖所有点
    """
    covered = [False] * len(points)
    return cover_recursive(points, covered, 0, k)

def cover_recursive(points, covered, covered_count, remaining_lines):
    """
    递归辅助方法：尝试覆盖剩余的点
    """
    n = len(points)
    if covered_count == n:
        return True
    if remaining_lines == 0:
        return False

    # 找到第一个未覆盖的点
    first = -1
    for i in range(n):
        if not covered[i]:
            first = i
            break
```

```
    # 尝试通过第一个未覆盖的点画一条直线
    for i in range(n):
        if i == first or covered[i]:
            continue

        # 标记所有在这条直线上的点
        new_covered = covered.copy()
        new_count = covered_count

        for j in range(n):
            if not new_covered[j] and is_collinear(points[first], points[i], points[j]):
                new_covered[j] = True
                new_count += 1

        if cover_recursive(points, new_covered, new_count, remaining_lines - 1):
            return True

    # 如果没有其他点，可以单独用一条直线覆盖第一个未覆盖的点
    covered[first] = True
    result = cover_recursive(points, covered, covered_count + 1, remaining_lines - 1)
    covered[first] = False  # 回溯
    return result

def is_collinear(p1, p2, p3):
    """
    辅助方法：判断三个点是否共线
    """
    # 使用叉积判断三点共线
    # (y2 - y1) * (x3 - x1) == (y3 - y1) * (x2 - x1)
    return (p2[1] - p1[1]) * (p3[0] - p1[0]) == (p3[1] - p1[1]) * (p2[0] - p1[0])
```

27. USACO - 二分查找题目

27.1 USACO - Aggressive Cows

题目描述：
农夫约翰建造了一个有 N 个隔间的牛棚，这些隔间分布在一条直线上，坐标是 x1,...,xN。他的 C 头牛不满于隔间的大小，经常互相攻击。约翰为了防止牛之间互相伤害，决定把每头牛都放在离其他牛尽可能远的隔间。也就是要最大化最近的两头牛之间的距离。

解题思路：
- 这是一个典型的二分答案问题

- 先对隔间坐标进行排序
- 使用二分查找猜测最大的最小距离
- 对于每个候选距离，使用贪心算法验证是否可以放置所有牛

时间复杂度：O(N log(max-min))
空间复杂度：O(1)

Java 实现
```java
/**
 * USACO - Aggressive Cows
 *
 * 时间复杂度: O(N log(max-min))
 * 空间复杂度: O(1)
 *
 * @param stalls 隔间坐标数组
 * @param cows 牛的数量
 * @return 最大的最小距离
 */
public int aggressiveCows(int[] stalls, int cows) {
    if (stalls == null || stalls.length == 0 || cows <= 0) {
        return 0;
    }

    Arrays.sort(stalls);

    int left = 0;
    int right = stalls[stalls.length - 1] - stalls[0];
    int result = 0;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);

        if (canPlaceCows(stalls, cows, mid)) {
            result = mid;
            left = mid + 1;  // 尝试更大的距离
        } else {
            right = mid - 1;  // 距离太大，需要减小
        }
    }

    return result;
}
```

```
/**
 * 验证是否可以在给定的最小距离下放置所有牛
 */
private boolean canPlaceCows(int[] stalls, int cows, int minDist) {
    int count = 1;  // 第一头牛放在第一个隔间
    int lastPosition = stalls[0];

    for (int i = 1; i < stalls.length; i++) {
        if (stalls[i] - lastPosition >= minDist) {
            count++;
            lastPosition = stalls[i];

            if (count >= cows) {
                return true;
            }
        }
    }

    return false;
}
```

C++ 实现
```cpp
/**
 * USACO - Aggressive Cows
 *
 * 时间复杂度: O(N log(max-min))
 * 空间复杂度: O(1)
 *
 * @param stalls 隔间坐标数组
 * @param cows 牛的数量
 * @return 最大的最小距离
 */
int aggressiveCows(vector<int>& stalls, int cows) {
    if (stalls.empty() || cows <= 0) {
        return 0;
    }

    sort(stalls.begin(), stalls.end());

    int left = 0;
```

```
    int right = stalls.back() - stalls.front();
    int result = 0;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);

        if (canPlaceCows(stalls, cows, mid)) {
            result = mid;
            left = mid + 1;  // 尝试更大的距离
        } else {
            right = mid - 1;  // 距离太大，需要减小
        }
    }

    return result;
}

/**
 * 验证是否可以在给定的最小距离下放置所有牛
 */
bool canPlaceCows(vector<int>& stalls, int cows, int minDist) {
    int count = 1;  // 第一头牛放在第一个隔间
    int lastPosition = stalls[0];

    for (int i = 1; i < stalls.size(); i++) {
        if (stalls[i] - lastPosition >= minDist) {
            count++;
            lastPosition = stalls[i];

            if (count >= cows) {
                return true;
            }
        }
    }

    return false;
}
```

Python 实现
```python
def aggressive_cows(stalls, cows):
    """
```

```
    USACO - Aggressive Cows

    时间复杂度: O(N log(max-min))
    空间复杂度: O(1)

    :param stalls: 隔间坐标数组
    :param cows: 牛的数量
    :return: 最大的最小距离
    """
    if not stalls or cows <= 0:
        return 0

    stalls.sort()

    left, right = 0, stalls[-1] - stalls[0]
    result = 0

    while left <= right:
        mid = left + ((right - left) >> 1)

        if can_place_cows(stalls, cows, mid):
            result = mid
            left = mid + 1  # 尝试更大的距离
        else:
            right = mid - 1  # 距离太大，需要减小

    return result

def can_place_cows(stalls, cows, min_dist):
    """
    验证是否可以在给定的最小距离下放置所有牛
    """
    count = 1  # 第一头牛放在第一个隔间
    last_position = stalls[0]

    for i in range(1, len(stalls)):
        if stalls[i] - last_position >= min_dist:
            count += 1
            last_position = stalls[i]

            if count >= cows:
                return True
```

```
    return False
```

28. SPOJ - 二分查找题目

28.1 SPOJ - AGGRCOW (Aggressive cows)

题目描述：
农夫约翰建造了一个有 C 个摊位的畜棚，摊位位于 x0, ..., x(C-1)。他的 M 头奶牛总是相互攻击，约翰必须以某种方式分配奶牛到摊位，使它们之间的最小距离尽可能大。

解题思路：
- 使用二分查找答案 + 贪心验证
- 先对摊位位置进行排序
- 使用二分查找猜测最大的最小距离
- 对于每个候选距离，使用贪心算法验证是否可以放置所有奶牛

时间复杂度：O(n log(max-min) * n)
空间复杂度：O(1)

Java 实现
```java
/**
 * SPOJ - AGGRCOW (Aggressive cows)
 *
 * 时间复杂度: O(n log(max-min) * n)
 * 空间复杂度: O(1)
 *
 * @param positions 摊位位置数组
 * @param cows 奶牛数量
 * @return 最大的最小距离
 */
public static int aggressiveCows(int[] positions, int cows) {
    Arrays.sort(positions);
    int left = 0, right = positions[positions.length - 1] - positions[0];
    int result = 0;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);
        if (canPlaceCows(positions, cows, mid)) {
            result = mid;
            left = mid + 1;
        } else {
```

```
            right = mid - 1;
        }
    }

    return result;
}

/**
 * 验证是否可以在给定的最小距离下放置所有奶牛
 */
private static boolean canPlaceCows(int[] positions, int cows, int minDist) {
    int count = 1;
    int lastPosition = positions[0];

    for (int i = 1; i < positions.length; i++) {
        if (positions[i] - lastPosition >= minDist) {
            count++;
            lastPosition = positions[i];
            if (count == cows) {
                return true;
            }
        }
    }

    return false;
}
```

C++ 实现

```cpp
/**
 * SPOJ - AGGRCOW (Aggressive cows)
 *
 * 时间复杂度: O(n log(max-min) * n)
 * 空间复杂度: O(1)
 *
 * @param positions 摊位位置数组
 * @param cows 奶牛数量
 * @return 最大的最小距离
 */
int aggressiveCows(std::vector<int>& positions, int cows) {
    std::sort(positions.begin(), positions.end());
    int left = 0, right = positions.back() - positions.front();
```

```
    int result = 0;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);
        if (canPlaceCows(positions, cows, mid)) {
            result = mid;
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return result;
}

/**
 * 验证是否可以在给定的最小距离下放置所有奶牛
 */
bool canPlaceCows(std::vector<int>& positions, int cows, int minDist) {
    int count = 1;
    int lastPosition = positions[0];

    for (int i = 1; i < positions.size(); i++) {
        if (positions[i] - lastPosition >= minDist) {
            count++;
            lastPosition = positions[i];
            if (count == cows) {
                return true;
            }
        }
    }

    return false;
}
```

Python 实现
```python
def aggressive_cows(positions, cows):
    """
    SPOJ - AGGRCOW (Aggressive cows)

    时间复杂度: O(n log(max-min) * n)
```

```
    空间复杂度: O(1)

    :param positions: 摊位位置数组
    :param cows: 奶牛数量
    :return: 最大的最小距离
    """
    positions.sort()
    left, right = 0, positions[-1] - positions[0]
    result = 0

    while left <= right:
        mid = left + ((right - left) >> 1)
        if can_place_cows(positions, cows, mid):
            result = mid
            left = mid + 1
        else:
            right = mid - 1

    return result

def can_place_cows(positions, cows, min_dist):
    """
    验证是否可以在给定的最小距离下放置所有奶牛
    """
    count = 1
    last_position = positions[0]

    for i in range(1, len(positions)):
        if positions[i] - last_position >= min_dist:
            count += 1
            last_position = positions[i]
            if count == cows:
                return True

    return False
```

29. Project Euler - 二分查找题目

29.1 Project Euler Problem 1 - Multiples of 3 and 5

题目描述:
如果我们列出所有小于 10 的 3 或 5 的倍数，我们得到 3, 5, 6, 9。这些数的和是 23。请找出所有小于 1000 的

3 或 5 的倍数的和。

解题思路：
- 虽然这不是典型的二分查找问题，但可以使用二分查找来优化计算
- 使用等差数列求和公式结合二分查找来避免暴力计算
- 计算 3 的倍数和 + 5 的倍数和 - 15 的倍数和

时间复杂度：O(1) 使用数学公式，O(log n) 如果使用二分查找
空间复杂度：O(1)

Java 实现
```java
/**
 * Project Euler Problem 1 - Multiples of 3 and 5
 *
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
 *
 * @param limit 上限值
 * @return 所有小于 limit 的 3 或 5 的倍数的和
 */
public int sumMultiples(int limit) {
    // 使用等差数列求和公式
    int sum3 = sumArithmeticProgression(3, limit);
    int sum5 = sumArithmeticProgression(5, limit);
    int sum15 = sumArithmeticProgression(15, limit);

    return sum3 + sum5 - sum15;
}

/**
 * 计算等差数列的和
 */
private int sumArithmeticProgression(int a, int limit) {
    int n = (limit - 1) / a;  // 项数
    return n * (a + a * n) / 2;  // 等差数列求和公式
}

/**
 * 使用二分查找的版本（虽然不必要，但展示二分思想）
 */
public int sumMultiplesWithBinarySearch(int limit) {
    // 使用二分查找找到最大的倍数
```

```
    int max3 = findMaxMultiple(3, limit);
    int max5 = findMaxMultiple(5, limit);
    int max15 = findMaxMultiple(15, limit);

    return sumArithmeticProgression(3, limit) +
           sumArithmeticProgression(5, limit) -
           sumArithmeticProgression(15, limit);
}

/**
 * 使用二分查找找到小于 limit 的最大倍数
 */
private int findMaxMultiple(int a, int limit) {
    int left = 0, right = limit / a;
    while (left < right) {
        int mid = left + ((right - left) >> 1);
        if (mid * a < limit) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return (left - 1) * a;  // 返回最大的小于 limit 的倍数
}
```

C++ 实现
```cpp
/**
 * Project Euler Problem 1 - Multiples of 3 and 5
 *
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
 *
 * @param limit 上限值
 * @return 所有小于 limit 的 3 或 5 的倍数的和
 */
int sumMultiples(int limit) {
    // 使用等差数列求和公式
    int sum3 = sumArithmeticProgression(3, limit);
    int sum5 = sumArithmeticProgression(5, limit);
    int sum15 = sumArithmeticProgression(15, limit);
```

```
    return sum3 + sum5 - sum15;
}

/**
 * 计算等差数列的和
 */
int sumArithmeticProgression(int a, int limit) {
    int n = (limit - 1) / a;  // 项数
    return n * (a + a * n) / 2;  // 等差数列求和公式
}

/**
 * 使用二分查找的版本（虽然不必要，但展示二分思想）
 */
int sumMultiplesWithBinarySearch(int limit) {
    // 使用二分查找找到最大的倍数
    int max3 = findMaxMultiple(3, limit);
    int max5 = findMaxMultiple(5, limit);
    int max15 = findMaxMultiple(15, limit);

    return sumArithmeticProgression(3, limit) +
           sumArithmeticProgression(5, limit) -
           sumArithmeticProgression(15, limit);
}

/**
 * 使用二分查找找到小于 limit 的最大倍数
 */
int findMaxMultiple(int a, int limit) {
    int left = 0, right = limit / a;
    while (left < right) {
        int mid = left + ((right - left) >> 1);
        if (mid * a < limit) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return (left - 1) * a;  // 返回最大的小于 limit 的倍数
}
```

Python 实现

```python
def sum_multiples(limit):
    """
    Project Euler Problem 1 - Multiples of 3 and 5

    时间复杂度: O(1)
    空间复杂度: O(1)

    :param limit: 上限值
    :return: 所有小于 limit 的 3 或 5 的倍数的和
    """
    # 使用等差数列求和公式
    sum3 = sum_arithmetic_progression(3, limit)
    sum5 = sum_arithmetic_progression(5, limit)
    sum15 = sum_arithmetic_progression(15, limit)

    return sum3 + sum5 - sum15

def sum_arithmetic_progression(a, limit):
    """
    计算等差数列的和
    """
    n = (limit - 1) // a  # 项数
    return n * (a + a * n) // 2  # 等差数列求和公式

def sum_multiples_with_binary_search(limit):
    """
    使用二分查找的版本（虽然不必要，但展示二分思想）
    """
    # 使用二分查找找到最大的倍数
    max3 = find_max_multiple(3, limit)
    max5 = find_max_multiple(5, limit)
    max15 = find_max_multiple(15, limit)

    return (sum_arithmetic_progression(3, limit) +
            sum_arithmetic_progression(5, limit) -
            sum_arithmetic_progression(15, limit))

def find_max_multiple(a, limit):
    """
    使用二分查找找到小于 limit 的最大倍数
    """
    left, right = 0, limit // a
```

```
    while left < right:
        mid = left + ((right - left) >> 1)
        if mid * a < limit:
            left = mid + 1
        else:
            right = mid
    return (left - 1) * a  # 返回最大的小于 limit 的倍数
```

30. 各大高校 OJ - 二分查找题目

30.1 浙江大学 OJ (ZOJ) - 二分查找题目

题目描述：
给定一个有序数组和一个目标值，使用二分查找找到目标值在数组中的位置。如果目标值不存在，返回应该插入的位置。

解题思路：
- 标准的二分查找问题
- 使用左边界查找模板
- 如果目标值存在，返回其索引；否则返回应该插入的位置

时间复杂度：O(log n)
空间复杂度：O(1)

Java 实现
```java
/**
 * 浙江大学 OJ - 二分查找
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param nums 有序数组
 * @param target 目标值
 * @return 目标值的索引或应该插入的位置
 */
public int binarySearch(int[] nums, int target) {
    if (nums == null || nums.length == 0) {
        return 0;
    }

    int left = 0, right = nums.length;
```

```
    while (left < right) {
        int mid = left + ((right - left) >> 1);

        if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }

    return left;
}
```

C++ 实现
```cpp
/**
 * 浙江大学 OJ - 二分查找
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param nums 有序数组
 * @param target 目标值
 * @return 目标值的索引或应该插入的位置
 */
int binarySearch(vector<int>& nums, int target) {
    if (nums.empty()) {
        return 0;
    }

    int left = 0, right = nums.size();

    while (left < right) {
        int mid = left + ((right - left) >> 1);

        if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
```

```
    return left;
}
```

Python 实现
```python
def binary_search(nums, target):
    """
    浙江大学 OJ - 二分查找

    时间复杂度: O(log n)
    空间复杂度: O(1)

    :param nums: 有序数组
    :param target: 目标值
    :return: 目标值的索引或应该插入的位置
    """
    if not nums:
        return 0

    left, right = 0, len(nums)

    while left < right:
        mid = left + ((right - left) >> 1)

        if nums[mid] < target:
            left = mid + 1
        else:
            right = mid

    return left
```

31. Codeforces - 二分查找题目

31.1 Codeforces 474B - Worms

题目描述：
有 n 堆虫子，第 i 堆有 a_i 条虫子。每条虫子都有一个编号，从 1 开始连续编号。给出 m 个查询，每个查询给出一个编号 x，问编号为 x 的虫子属于哪一堆。

解题思路：

- 使用前缀和数组记录每堆虫子的结束位置
- 对于每个查询，使用二分查找在前缀和数组中查找 x 所在的位置
- 找到第一个前缀和大于等于 x 的位置

时间复杂度：O(n + m log n)
空间复杂度：O(n)

Java 实现
```java
/**
 * Codeforces 474B - Worms
 *
 * 时间复杂度: O(n + m log n)
 * 空间复杂度: O(n)
 *
 * @param piles 每堆虫子的数量数组
 * @param queries 查询数组
 * @return 每个查询对应的堆编号
 */
public int[] findWormPiles(int[] piles, int[] queries) {
    int n = piles.length;
    int m = queries.length;

    // 计算前缀和
    long[] prefixSum = new long[n];
    prefixSum[0] = piles[0];
    for (int i = 1; i < n; i++) {
        prefixSum[i] = prefixSum[i - 1] + piles[i];
    }

    int[] result = new int[m];

    for (int i = 0; i < m; i++) {
        long x = queries[i];

        // 使用二分查找找到第一个前缀和 >= x 的位置
        int left = 0, right = n - 1;
        int pileIndex = -1;

        while (left <= right) {
            int mid = left + ((right - left) >> 1);

            if (prefixSum[mid] >= x) {
```

```
                pileIndex = mid;
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }

        result[i] = pileIndex + 1;  // 1-based索引
    }

    return result;
}
```

C++ 实现
```cpp
/**
 * Codeforces 474B - Worms
 *
 * 时间复杂度: O(n + m log n)
 * 空间复杂度: O(n)
 *
 * @param piles 每堆虫子的数量数组
 * @param queries 查询数组
 * @return 每个查询对应的堆编号
 */
vector<int> findWormPiles(vector<int>& piles, vector<long long>& queries) {
    int n = piles.size();
    int m = queries.size();

    // 计算前缀和
    vector<long long> prefixSum(n);
    prefixSum[0] = piles[0];
    for (int i = 1; i < n; i++) {
        prefixSum[i] = prefixSum[i - 1] + piles[i];
    }

    vector<int> result(m);

    for (int i = 0; i < m; i++) {
        long long x = queries[i];

        // 使用二分查找找到第一个前缀和 >= x 的位置
```

```
        int left = 0, right = n - 1;
        int pileIndex = -1;

        while (left <= right) {
            int mid = left + ((right - left) >> 1);

            if (prefixSum[mid] >= x) {
                pileIndex = mid;
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }

        result[i] = pileIndex + 1;  // 1-based索引
    }

    return result;
}
```

Python 实现
```python
def find_worm_piles(piles, queries):
    """
    Codeforces 474B - Worms

    时间复杂度: O(n + m log n)
    空间复杂度: O(n)

    :param piles: 每堆虫子的数量数组
    :param queries: 查询数组
    :return: 每个查询对应的堆编号
    """
    n = len(piles)
    m = len(queries)

    # 计算前缀和
    prefix_sum = [0] * n
    prefix_sum[0] = piles[0]
    for i in range(1, n):
        prefix_sum[i] = prefix_sum[i - 1] + piles[i]
```

```
    result = [0] * m

    for i in range(m):
        x = queries[i]

        # 使用二分查找找到第一个前缀和 >= x 的位置
        left, right = 0, n - 1
        pile_index = -1

        while left <= right:
            mid = left + ((right - left) >> 1)

            if prefix_sum[mid] >= x:
                pile_index = mid
                right = mid - 1
            else:
                left = mid + 1

        result[i] = pile_index + 1  # 1-based索引

    return result
```

32. HackerEarth - 二分查找题目

32.1 HackerEarth - Monk and Search

题目描述：
给定一个数组，有Q个查询。每个查询有两种类型：
1. 0 x: 查询数组中大于等于x的元素个数
2. 1 x: 查询数组中大于x的元素个数

解题思路：
- 先对数组进行排序
- 对于类型0查询，使用二分查找找到第一个大于等于x的位置
- 对于类型1查询，使用二分查找找到第一个大于x的位置
- 计算从该位置到数组末尾的元素个数

时间复杂度：O(n log n + q log n)
空间复杂度：O(n)

Java 实现
```java
```

```
/**
 * HackerEarth - Monk and Search
 *
 * 时间复杂度: O(n log n + q log n)
 * 空间复杂度: O(n)
 *
 * @param arr 数组
 * @param queries 查询数组，每个查询是[type, x]
 * @return 每个查询的结果
 */
public int[] monkAndSearch(int[] arr, int[][] queries) {
    Arrays.sort(arr);
    int n = arr.length;
    int q = queries.length;
    int[] result = new int[q];

    for (int i = 0; i < q; i++) {
        int type = queries[i][0];
        int x = queries[i][1];

        if (type == 0) {
            // 查询大于等于 x 的元素个数
            int pos = findFirstGreaterOrEqual(arr, x);
            result[i] = n - pos;
        } else {
            // 查询大于 x 的元素个数
            int pos = findFirstGreater(arr, x);
            result[i] = n - pos;
        }
    }

    return result;
}

/**
 * 找到第一个大于等于 x 的位置
 */
private int findFirstGreaterOrEqual(int[] arr, int x) {
    int left = 0, right = arr.length;
    while (left < right) {
        int mid = left + ((right - left) >> 1);
        if (arr[mid] < x) {
            left = mid + 1;
```

```
        } else {
            right = mid;
        }
    }
    return left;
}

/**
 * 找到第一个大于 x 的位置
 */
private int findFirstGreater(int[] arr, int x) {
    int left = 0, right = arr.length;
    while (left < right) {
        int mid = left + ((right - left) >> 1);
        if (arr[mid] <= x) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return left;
}
```

C++ 实现

```cpp
/**
 * HackerEarth - Monk and Search
 *
 * 时间复杂度: O(n log n + q log n)
 * 空间复杂度: O(n)
 *
 * @param arr 数组
 * @param queries 查询数组，每个查询是[type, x]
 * @return 每个查询的结果
 */
vector<int> monkAndSearch(vector<int>& arr, vector<pair<int, int>>& queries) {
    sort(arr.begin(), arr.end());
    int n = arr.size();
    int q = queries.size();
    vector<int> result(q);

    for (int i = 0; i < q; i++) {
```

```
        int type = queries[i].first;
        int x = queries[i].second;

        if (type == 0) {
            // 查询大于等于 x 的元素个数
            int pos = findFirstGreaterOrEqual(arr, x);
            result[i] = n - pos;
        } else {
            // 查询大于 x 的元素个数
            int pos = findFirstGreater(arr, x);
            result[i] = n - pos;
        }
    }

    return result;
}

/**
 * 找到第一个大于等于 x 的位置
 */
int findFirstGreaterOrEqual(vector<int>& arr, int x) {
    int left = 0, right = arr.size();
    while (left < right) {
        int mid = left + ((right - left) >> 1);
        if (arr[mid] < x) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return left;
}

/**
 * 找到第一个大于 x 的位置
 */
int findFirstGreater(vector<int>& arr, int x) {
    int left = 0, right = arr.size();
    while (left < right) {
        int mid = left + ((right - left) >> 1);
        if (arr[mid] <= x) {
            left = mid + 1;
        } else {
```

```
            right = mid;
        }
    }
    return left;
}
```

Python 实现
```python
def monk_and_search(arr, queries):
    """
    HackerEarth - Monk and Search

    时间复杂度: O(n log n + q log n)
    空间复杂度: O(n)

    :param arr: 数组
    :param queries: 查询数组，每个查询是[type, x]
    :return: 每个查询的结果
    """
    arr.sort()
    n = len(arr)
    result = []

    for query in queries:
        type_val, x = query

        if type_val == 0:
            # 查询大于等于 x 的元素个数
            pos = find_first_greater_or_equal(arr, x)
            result.append(n - pos)
        else:
            # 查询大于 x 的元素个数
            pos = find_first_greater(arr, x)
            result.append(n - pos)

    return result

def find_first_greater_or_equal(arr, x):
    """
    找到第一个大于等于 x 的位置
    """
    left, right = 0, len(arr)
```

```
    while left < right:
        mid = left + ((right - left) >> 1)
        if arr[mid] < x:
            left = mid + 1
        else:
            right = mid
    return left

def find_first_greater(arr, x):
    """
    找到第一个大于 x 的位置
    """
    left, right = 0, len(arr)
    while left < right:
        mid = left + ((right - left) >> 1)
        if arr[mid] <= x:
            left = mid + 1
        else:
            right = mid
    return left
```

33. 计蒜客 - 二分查找题目

33.1 计蒜客 - 蒜头君的购物袋

题目描述：
蒜头君去超市购物，他有一个购物袋，最多能装 V 体积的物品。超市有 n 件商品，每件商品有体积 v_i 和价值 w_i。蒜头君想要选择一些商品放入购物袋，使得总价值最大，且总体积不超过 V。

解题思路：
- 这是一个经典的 0-1 背包问题
- 虽然主要使用动态规划，但可以结合二分查找进行优化
- 对于大规模数据，可以使用二分查找来优化状态转移

时间复杂度：O(nV) 标准 DP，O(n log V) 优化版本
空间复杂度：O(V)

Java 实现
```java
/**
 * 计蒜客 - 蒜头君的购物袋 (0-1 背包问题)
 *
```

```
 * 时间复杂度: O(nV)
 * 空间复杂度: O(V)
 *
 * @param V 购物袋容量
 * @param volumes 商品体积数组
 * @param values 商品价值数组
 * @return 最大价值
 */
public int shoppingBag(int V, int[] volumes, int[] values) {
    int n = volumes.length;
    int[] dp = new int[V + 1];

    for (int i = 0; i < n; i++) {
        for (int j = V; j >= volumes[i]; j--) {
            dp[j] = Math.max(dp[j], dp[j - volumes[i]] + values[i]);
        }
    }

    return dp[V];
}

/**
 * 优化版本：使用二分查找思想（虽然不直接使用二分查找，但展示优化思路）
 */
public int shoppingBagOptimized(int V, int[] volumes, int[] values) {
    int n = volumes.length;

    // 对于大规模数据，可以考虑物品分组优化
    // 这里展示标准 DP 实现，实际工程中可以根据数据规模选择不同算法

    return shoppingBag(V, volumes, values);
}
```

C++ 实现

```cpp
/**
 * 计蒜客 - 蒜头君的购物袋 (0-1 背包问题)
 *
 * 时间复杂度: O(nV)
 * 空间复杂度: O(V)
 *
 * @param V 购物袋容量
```

```
 * @param volumes 商品体积数组
 * @param values 商品价值数组
 * @return 最大价值
 */
int shoppingBag(int V, vector<int>& volumes, vector<int>& values) {
    int n = volumes.size();
    vector<int> dp(V + 1, 0);

    for (int i = 0; i < n; i++) {
        for (int j = V; j >= volumes[i]; j--) {
            dp[j] = max(dp[j], dp[j - volumes[i]] + values[i]);
        }
    }

    return dp[V];
}
```

Python 实现
```python
def shopping_bag(V, volumes, values):
    """
    计蒜客 - 蒜头君的购物袋 (0-1 背包问题)

    时间复杂度: O(nV)
    空间复杂度: O(V)

    :param V: 购物袋容量
    :param volumes: 商品体积数组
    :param values: 商品价值数组
    :return: 最大价值
    """
    n = len(volumes)
    dp = [0] * (V + 1)

    for i in range(n):
        for j in range(V, volumes[i] - 1, -1):
            dp[j] = max(dp[j], dp[j - volumes[i]] + values[i])

    return dp[V]
```

34. MarsCode - 二分查找题目

34.1 MarsCode - 寻找峰值元素

题目描述：
给定一个整数数组，其中相邻元素不相等。找到峰值元素并返回其索引。峰值元素是指其值大于左右相邻值的元素。数组可能包含多个峰值，返回任何一个峰值的索引即可。

解题思路：
- 使用二分查找来寻找峰值
- 比较中间元素与其相邻元素的大小关系
- 根据比较结果决定搜索左半部分还是右半部分

时间复杂度：O(log n)
空间复杂度：O(1)

Java 实现
```java
/**
 * MarsCode - 寻找峰值元素
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param nums 整数数组
 * @return 峰值元素的索引
 */
public int findPeakElement(int[] nums) {
    if (nums == null || nums.length == 0) {
        return -1;
    }

    int left = 0, right = nums.length - 1;

    while (left < right) {
        int mid = left + ((right - left) >> 1);

        if (nums[mid] > nums[mid + 1]) {
            // 峰值在左半部分或 mid 就是峰值
            right = mid;
        } else {
            // 峰值在右半部分
            left = mid + 1;
        }
```

```
    }

    return left;
}
```

C++ 实现
```cpp
/**
 * MarsCode - 寻找峰值元素
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param nums 整数数组
 * @return 峰值元素的索引
 */
int findPeakElement(vector<int>& nums) {
    if (nums.empty()) {
        return -1;
    }

    int left = 0, right = nums.size() - 1;

    while (left < right) {
        int mid = left + ((right - left) >> 1);

        if (nums[mid] > nums[mid + 1]) {
            // 峰值在左半部分或 mid 就是峰值
            right = mid;
        } else {
            // 峰值在右半部分
            left = mid + 1;
        }
    }

    return left;
}
```

Python 实现
```python
def find_peak_element(nums):
```

```
    """
    MarsCode - 寻找峰值元素

    时间复杂度: O(log n)
    空间复杂度: O(1)

    :param nums: 整数数组
    :return: 峰值元素的索引
    """
    if not nums:
        return -1

    left, right = 0, len(nums) - 1

    while left < right:
        mid = left + ((right - left) >> 1)

        if nums[mid] > nums[mid + 1]:
            # 峰值在左半部分或 mid 就是峰值
            right = mid
        else:
            # 峰值在右半部分
            left = mid + 1

    return left
```

35. UVa OJ - 二分查找题目

35.1 UVa 10341 - Solve It

题目描述：
求解方程：p*e^(-x) + q*sin(x) + r*cos(x) + s*tan(x) + t*x^2 + u = 0，其中 0 ≤ x ≤ 1。给定 p, q, r, s, t, u 的值，求 x 的值。

解题思路：
- 这是一个非线性方程求解问题
- 使用二分查找来逼近解
- 由于函数在[0,1]区间内是连续的，可以使用二分法

时间复杂度：O(log(1/ε))，其中 ε 是精度要求
空间复杂度：O(1)

Java 实现

```java
/**
 * UVa 10341 - Solve It
 *
 * 时间复杂度: O(log(1/ε))
 * 空间复杂度: O(1)
 *
 * @param p, q, r, s, t, u 方程参数
 * @return 方程的解 x，精度 1e-9
 */
public double solveEquation(double p, double q, double r, double s, double t, double u) {
    // 定义方程函数
    Function<Double, Double> f = x ->
        p * Math.exp(-x) + q * Math.sin(x) + r * Math.cos(x) +
        s * Math.tan(x) + t * x * x + u;

    double left = 0.0, right = 1.0;
    double epsilon = 1e-9;

    // 检查端点值
    double f0 = f.apply(0.0);
    double f1 = f.apply(1.0);

    if (Math.abs(f0) < epsilon) return 0.0;
    if (Math.abs(f1) < epsilon) return 1.0;

    // 如果端点值同号，则无解
    if (f0 * f1 > 0) {
        return -1;  // 无解
    }

    // 使用二分查找求解
    while (right - left > epsilon) {
        double mid = (left + right) / 2;
        double fmid = f.apply(mid);

        if (Math.abs(fmid) < epsilon) {
            return mid;
        }

        if (f0 * fmid < 0) {
            right = mid;
```

```
        } else {
            left = mid;
            f0 = fmid;  // 更新左端点函数值
        }
    }

    return (left + right) / 2;
}
```

C++ 实现
```cpp
/**
 * UVa 10341 - Solve It
 *
 * 时间复杂度: O(log(1/ε))
 * 空间复杂度: O(1)
 *
 * @param p, q, r, s, t, u 方程参数
 * @return 方程的解 x，精度 1e-9
 */
double solveEquation(double p, double q, double r, double s, double t, double u) {
    auto f = [&](double x) {
        return p * exp(-x) + q * sin(x) + r * cos(x) +
               s * tan(x) + t * x * x + u;
    };

    double left = 0.0, right = 1.0;
    double epsilon = 1e-9;

    // 检查端点值
    double f0 = f(0.0);
    double f1 = f(1.0);

    if (abs(f0) < epsilon) return 0.0;
    if (abs(f1) < epsilon) return 1.0;

    // 如果端点值同号，则无解
    if (f0 * f1 > 0) {
        return -1;  // 无解
    }

    // 使用二分查找求解
```

```
    while (right - left > epsilon) {
        double mid = (left + right) / 2;
        double fmid = f(mid);

        if (abs(fmid) < epsilon) {
            return mid;
        }

        if (f0 * fmid < 0) {
            right = mid;
        } else {
            left = mid;
            f0 = fmid;  // 更新左端点函数值
        }
    }

    return (left + right) / 2;
}
```

Python 实现
```python
import math

def solve_equation(p, q, r, s, t, u):
    """
    UVa 10341 - Solve It

    时间复杂度: O(log(1/ε))
    空间复杂度: O(1)

    :param p, q, r, s, t, u: 方程参数
    :return: 方程的解 x，精度 1e-9
    """
    def f(x):
        return (p * math.exp(-x) + q * math.sin(x) + r * math.cos(x) +
                s * math.tan(x) + t * x * x + u)

    left, right = 0.0, 1.0
    epsilon = 1e-9

    # 检查端点值
    f0 = f(0.0)
```

```
    f1 = f(1.0)

    if abs(f0) < epsilon:
        return 0.0
    if abs(f1) < epsilon:
        return 1.0

    # 如果端点值同号，则无解
    if f0 * f1 > 0:
        return -1  # 无解

    # 使用二分查找求解
    while right - left > epsilon:
        mid = (left + right) / 2
        fmid = f(mid)

        if abs(fmid) < epsilon:
            return mid

        if f0 * fmid < 0:
            right = mid
        else:
            left = mid
            f0 = fmid  # 更新左端点函数值

    return (left + right) / 2
```

36. TimusOJ - 二分查找题目

36.1 Timus 1490 - Bicolored Horses

题目描述：
有 n 匹马排成一排，每匹马要么是黑色要么是白色。现在需要将这 n 匹马分成 k 个连续的非空组。每个组的不愉快值定义为该组中黑色马的数量乘以白色马的数量。总不愉快值是所有组的不愉快值之和。求最小的总不愉快值。

解题思路：
- 这是一个动态规划问题，但可以使用二分查找优化
- 使用 DP + 二分查找（决策单调性优化）
- 定义 dp[i][j]表示前 i 匹马分成 j 组的最小不愉快值
- 使用二分查找来优化状态转移

时间复杂度：O(n^2 log n)
空间复杂度：O(n^2)

Java 实现
```java
/**
 * Timus 1490 - Bicolored Horses
 *
 * 时间复杂度: O(n^2 log n)
 * 空间复杂度: O(n^2)
 *
 * @param n 马的数量
 * @param k 分组数
 * @param horses 马的序列（0 表示白色，1 表示黑色）
 * @return 最小的总不愉快值
 */
public long minUnhappiness(int n, int k, int[] horses) {
    if (k > n) return 0;

    // 预处理前缀和
    int[] blackPrefix = new int[n + 1];
    int[] whitePrefix = new int[n + 1];

    for (int i = 1; i <= n; i++) {
        blackPrefix[i] = blackPrefix[i - 1] + (horses[i - 1] == 1 ? 1 : 0);
        whitePrefix[i] = whitePrefix[i - 1] + (horses[i - 1] == 0 ? 1 : 0);
    }

    // DP 数组
    long[][] dp = new long[n + 1][k + 1];
    for (int i = 0; i <= n; i++) {
        Arrays.fill(dp[i], Long.MAX_VALUE / 2);
    }
    dp[0][0] = 0;

    for (int j = 1; j <= k; j++) {
        for (int i = j; i <= n; i++) {
            for (int p = j - 1; p < i; p++) {
                int blackCount = blackPrefix[i] - blackPrefix[p];
                int whiteCount = whitePrefix[i] - whitePrefix[p];
                long cost = (long) blackCount * whiteCount;

                dp[i][j] = Math.min(dp[i][j], dp[p][j - 1] + cost);
```

```
            }
        }
    }

    return dp[n][k];
}

/**
 * 优化版本：使用二分查找优化决策单调性
 */
public long minUnhappinessOptimized(int n, int k, int[] horses) {
    if (k > n) return 0;

    // 预处理前缀和
    int[] blackPrefix = new int[n + 1];
    int[] whitePrefix = new int[n + 1];

    for (int i = 1; i <= n; i++) {
        blackPrefix[i] = blackPrefix[i - 1] + (horses[i - 1] == 1 ? 1 : 0);
        whitePrefix[i] = whitePrefix[i - 1] + (horses[i - 1] == 0 ? 1 : 0);
    }

    // 使用决策单调性优化
    long[] dp = new long[n + 1];
    long[] newDp = new long[n + 1];
    Arrays.fill(dp, Long.MAX_VALUE / 2);
    dp[0] = 0;

    for (int j = 1; j <= k; j++) {
        Arrays.fill(newDp, Long.MAX_VALUE / 2);

        // 使用二分查找优化决策单调性
        for (int i = j; i <= n; i++) {
            int left = j - 1, right = i;
            while (right - left > 1) {
                int mid = left + ((right - left) >> 1);
                long cost1 = calculateCost(blackPrefix, whitePrefix, mid, i);
                long cost2 = calculateCost(blackPrefix, whitePrefix, mid + 1, i);

                if (dp[mid] + cost1 < dp[mid + 1] + cost2) {
                    right = mid;
                } else {
                    left = mid;
```

```
                }
            }

            newDp[i] = Math.min(newDp[i], dp[right] + calculateCost(blackPrefix, whitePrefix,
right, i));
        }

        // 交换 DP 数组
        long[] temp = dp;
        dp = newDp;
        newDp = temp;
    }

    return dp[n];
}

private long calculateCost(int[] blackPrefix, int[] whitePrefix, int from, int to) {
    int blackCount = blackPrefix[to] - blackPrefix[from];
    int whiteCount = whitePrefix[to] - whitePrefix[from];
    return (long) blackCount * whiteCount;
}
```

C++ 实现
```cpp
/**
 * Timus 1490 - Bicolored Horses
 *
 * 时间复杂度: O(n^2 log n)
 * 空间复杂度: O(n^2)
 *
 * @param n 马的数量
 * @param k 分组数
 * @param horses 马的序列（0 表示白色，1 表示黑色）
 * @return 最小的总不愉快值
 */
long long minUnhappiness(int n, int k, vector<int>& horses) {
    if (k > n) return 0;

    // 预处理前缀和
    vector<int> blackPrefix(n + 1, 0);
    vector<int> whitePrefix(n + 1, 0);
```

```
    for (int i = 1; i <= n; i++) {
        blackPrefix[i] = blackPrefix[i - 1] + (horses[i - 1] == 1 ? 1 : 0);
        whitePrefix[i] = whitePrefix[i - 1] + (horses[i - 1] == 0 ? 1 : 0);
    }

    // DP 数组
    vector<vector<long long>> dp(n + 1, vector<long long>(k + 1, LLONG_MAX / 2));
    dp[0][0] = 0;

    for (int j = 1; j <= k; j++) {
        for (int i = j; i <= n; i++) {
            for (int p = j - 1; p < i; p++) {
                int blackCount = blackPrefix[i] - blackPrefix[p];
                int whiteCount = whitePrefix[i] - whitePrefix[p];
                long long cost = (long long) blackCount * whiteCount;

                dp[i][j] = min(dp[i][j], dp[p][j - 1] + cost);
            }
        }
    }

    return dp[n][k];
}
```

Python 实现
```python
def min_unhappiness(n, k, horses):
    """
    Timus 1490 - Bicolored Horses

    时间复杂度: O(n^2 log n)
    空间复杂度: O(n^2)

    :param n: 马的数量
    :param k: 分组数
    :param horses: 马的序列（0 表示白色，1 表示黑色）
    :return: 最小的总不愉快值
    """
    if k > n:
        return 0

    # 预处理前缀和
```

```
    black_prefix = [0] * (n + 1)
    white_prefix = [0] * (n + 1)

    for i in range(1, n + 1):
        black_prefix[i] = black_prefix[i - 1] + (1 if horses[i - 1] == 1 else 0)
        white_prefix[i] = white_prefix[i - 1] + (1 if horses[i - 1] == 0 else 0)

    # DP 数组
    dp = [[float('inf')] * (k + 1) for _ in range(n + 1)]
    dp[0][0] = 0

    for j in range(1, k + 1):
        for i in range(j, n + 1):
            for p in range(j - 1, i):
                black_count = black_prefix[i] - black_prefix[p]
                white_count = white_prefix[i] - white_prefix[p]
                cost = black_count * white_count

                dp[i][j] = min(dp[i][j], dp[p][j - 1] + cost)

    return dp[n][k]
```

37. AizuOJ - 二分查找题目

37.1 AizuOJ - ALDS1_4_B: Binary Search

题目描述：
给定两个数组 S 和 T，统计数组 T 中有多少个元素出现在数组 S 中。数组 S 已经排序。

解题思路：
- 使用二分查找来检查 T 中的每个元素是否在 S 中出现
- 由于 S 已经排序，可以直接使用二分查找
- 对于 T 中的每个元素，在 S 中进行二分查找

时间复杂度：O(q log n)，其中 n 是 S 的大小，q 是 T 的大小
空间复杂度：O(1)

Java 实现
```java
/**
 * AizuOJ - ALDS1_4_B: Binary Search
 *
```

```
 * 时间复杂度: O(q log n)
 * 空间复杂度: O(1)
 *
 * @param S 已排序的数组
 * @param T 查询数组
 * @return T 中出现在 S 中的元素个数
 */
public int binarySearchCount(int[] S, int[] T) {
    int count = 0;

    for (int target : T) {
        if (binarySearch(S, target)) {
            count++;
        }
    }

    return count;
}

/**
 * 二分查找辅助方法
 */
private boolean binarySearch(int[] arr, int target) {
    int left = 0, right = arr.length - 1;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);

        if (arr[mid] == target) {
            return true;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return false;
}
```

C++ 实现
```cpp
```

```cpp
/**
 * AizuOJ - ALDS1_4_B: Binary Search
 *
 * 时间复杂度: O(q log n)
 * 空间复杂度: O(1)
 *
 * @param S 已排序的数组
 * @param T 查询数组
 * @return T 中出现在 S 中的元素个数
 */
int binarySearchCount(vector<int>& S, vector<int>& T) {
    int count = 0;

    for (int target : T) {
        if (binary_search(S.begin(), S.end(), target)) {
            count++;
        }
    }

    return count;
}

/**
 * 自定义二分查找实现
 */
bool binarySearch(vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);

        if (arr[mid] == target) {
            return true;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return false;
}
```

Python 实现

```python
def binary_search_count(S, T):
    """
    AizuOJ - ALDS1_4_B: Binary Search

    时间复杂度: O(q log n)
    空间复杂度: O(1)

    :param S: 已排序的数组
    :param T: 查询数组
    :return: T 中出现在 S 中的元素个数
    """
    count = 0

    for target in T:
        if binary_search(S, target):
            count += 1

    return count

def binary_search(arr, target):
    """
    二分查找辅助方法
    """
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = left + ((right - left) >> 1)

        if arr[mid] == target:
            return True
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return False
```

38. Comet OJ - 二分查找题目

38.1 Comet OJ - 二分查找练习

题目描述：
给定一个有序数组和一个目标值，使用二分查找找到目标值在数组中的位置。如果目标值存在，返回其索引；否则返回-1。

解题思路：
- 标准的二分查找实现
- 使用闭区间[left, right]进行搜索
- 循环条件为 left <= right

时间复杂度：O(log n)
空间复杂度：O(1)

Java 实现
```java
/**
 * Comet OJ - 二分查找练习
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param nums 有序数组
 * @param target 目标值
 * @return 目标值的索引，不存在返回-1
 */
public int binarySearch(int[] nums, int target) {
    if (nums == null || nums.length == 0) {
        return -1;
    }

    int left = 0, right = nums.length - 1;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);

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

```
    }

    return -1;
}
```

C++ 实现
```cpp
/**
 * Comet OJ - 二分查找练习
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param nums 有序数组
 * @param target 目标值
 * @return 目标值的索引，不存在返回-1
 */
int binarySearch(vector<int>& nums, int target) {
    if (nums.empty()) {
        return -1;
    }

    int left = 0, right = nums.size() - 1;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);

        if (nums[mid] == target) {
            return mid;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1;
}
```

Python 实现
```python
```

```
def binary_search(nums, target):
    """
    Comet OJ - 二分查找练习

    时间复杂度: O(log n)
    空间复杂度: O(1)

    :param nums: 有序数组
    :param target: 目标值
    :return: 目标值的索引，不存在返回-1
    """
    if not nums:
        return -1

    left, right = 0, len(nums) - 1

    while left <= right:
        mid = left + ((right - left) >> 1)

        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1
```

39. 杭电 OJ (HDU) - 二分查找题目

39.1 HDU 2141 - Can you find it? (再次实现)

题目描述：
给定三个数组 A、B、C，以及多个查询 X。对于每个查询，判断是否存在 a∈A, b∈B, c∈C，使得 a+b+c=X。

解题思路：
- 将 A+B 的所有可能和存储在一个数组中
- 对这个和数组进行排序
- 对于每个查询 X，使用二分查找在 A+B 的和数组中查找是否存在 X-c（c∈C）

时间复杂度：O(L*M + N*log(L*M))
空间复杂度：O(L*M)

Java 实现
```java
/**
 * HDU 2141 - Can you find it?
 *
 * 时间复杂度: O(L*M + N*log(L*M))
 * 空间复杂度: O(L*M)
 *
 * @param A 数组 A
 * @param B 数组 B
 * @param C 数组 C
 * @param queries 查询数组
 * @return 每个查询的结果（true/false）
 */
public boolean[] canFindIt(int[] A, int[] B, int[] C, int[] queries) {
    // 计算 A+B 的所有可能和
    List<Integer> sumAB = new ArrayList<>();
    for (int a : A) {
        for (int b : B) {
            sumAB.add(a + b);
        }
    }

    // 排序 A+B 的和数组
    Collections.sort(sumAB);

    boolean[] results = new boolean[queries.length];

    for (int i = 0; i < queries.length; i++) {
        int X = queries[i];
        boolean found = false;

        // 对于每个 c∈C，在 A+B 的和数组中查找 X-c
        for (int c : C) {
            int target = X - c;

            // 使用二分查找
            int left = 0, right = sumAB.size() - 1;
            while (left <= right) {
                int mid = left + ((right - left) >> 1);
                int midVal = sumAB.get(mid);
```

```
                if (midVal == target) {
                    found = true;
                    break;
                } else if (midVal < target) {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }

            if (found) break;
        }

        results[i] = found;
    }

    return results;
}
```

C++ 实现
```cpp
/**
 * HDU 2141 - Can you find it?
 *
 * 时间复杂度: O(L*M + N*log(L*M))
 * 空间复杂度: O(L*M)
 *
 * @param A 数组A
 * @param B 数组B
 * @param C 数组C
 * @param queries 查询数组
 * @return 每个查询的结果（true/false）
 */
vector<bool> canFindIt(vector<int>& A, vector<int>& B, vector<int>& C, vector<int>& queries) {
    // 计算A+B的所有可能和
    vector<int> sumAB;
    for (int a : A) {
        for (int b : B) {
            sumAB.push_back(a + b);
        }
    }
```

```
    // 排序 A+B 的和数组
    sort(sumAB.begin(), sumAB.end());

    vector<bool> results(queries.size(), false);

    for (int i = 0; i < queries.size(); i++) {
        int X = queries[i];
        bool found = false;

        // 对于每个 c∈C，在 A+B 的和数组中查找 X-c
        for (int c : C) {
            int target = X - c;

            // 使用二分查找
            int left = 0, right = sumAB.size() - 1;
            while (left <= right) {
                int mid = left + ((right - left) >> 1);
                int midVal = sumAB[mid];

                if (midVal == target) {
                    found = true;
                    break;
                } else if (midVal < target) {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }

            if (found) break;
        }

        results[i] = found;
    }

    return results;
}
```

Python 实现
```python
def can_find_it(A, B, C, queries):
    """
```

HDU 2141 - Can you find it?

时间复杂度: O(L*M + N*log(L*M))
空间复杂度: O(L*M)

```
:param A: 数组 A
:param B: 数组 B
:param C: 数组 C
:param queries: 查询数组
:return: 每个查询的结果（True/False）
"""
# 计算 A+B 的所有可能和
sum_ab = []
for a in A:
    for b in B:
        sum_ab.append(a + b)

# 排序 A+B 的和数组
sum_ab.sort()

results = [False] * len(queries)

for i, X in enumerate(queries):
    found = False

    # 对于每个 c∈C，在 A+B 的和数组中查找 X-c
    for c in C:
        target = X - c

        # 使用二分查找
        left, right = 0, len(sum_ab) - 1
        while left <= right:
            mid = left + ((right - left) >> 1)
            mid_val = sum_ab[mid]

            if mid_val == target:
                found = True
                break
            elif mid_val < target:
                left = mid + 1
            else:
                right = mid - 1
```

```
            if found:
                break

        results[i] = found

    return results
```

40. LOJ (LibreOJ) - 二分查找题目

40.1 LOJ - 二分查找模板题

题目描述：
实现一个二分查找函数，在有序数组中查找目标值。如果找到返回索引，否则返回-1。

解题思路：
- 标准的二分查找实现
- 使用左闭右闭区间
- 注意边界条件的处理

时间复杂度：O(log n)
空间复杂度：O(1)

Java 实现
```java
/**
 * LOJ - 二分查找模板题
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param nums 有序数组
 * @param target 目标值
 * @return 目标值的索引，不存在返回-1
 */
public int binarySearch(int[] nums, int target) {
    if (nums == null || nums.length == 0) {
        return -1;
    }

    int left = 0, right = nums.length - 1;

    while (left <= right) {
```

```
        int mid = left + ((right - left) >> 1);

        if (nums[mid] == target) {
            return mid;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1;
}
```

C++ 实现
```cpp
/**
 * LOJ - 二分查找模板题
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param nums 有序数组
 * @param target 目标值
 * @return 目标值的索引，不存在返回-1
 */
int binarySearch(vector<int>& nums, int target) {
    if (nums.empty()) {
        return -1;
    }

    int left = 0, right = nums.size() - 1;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);

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

```
        }
    }

    return -1;
}
```

Python 实现
```python
def binary_search(nums, target):
    """
    LOJ - 二分查找模板题

    时间复杂度: O(log n)
    空间复杂度: O(1)

    :param nums: 有序数组
    :param target: 目标值
    :return: 目标值的索引，不存在返回-1
    """
    if not nums:
        return -1

    left, right = 0, len(nums) - 1

    while left <= right:
        mid = left + ((right - left) >> 1)

        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1
```

41. 牛客网 - 二分查找题目

41.1 牛客网 - 二分查找

题目描述：

请实现有重复数字的有序数组的二分查找。输出在数组中第一个大于等于查找值的位置，如果数组中不存在这样的数，则输出数组长度加一。

解题思路：
- 使用二分查找找到第一个大于等于目标值的位置
- 如果找到，返回位置+1（1-based 索引）
- 如果没找到，返回数组长度+1

时间复杂度：O(log n)
空间复杂度：O(1)

Java 实现
```java
/**
 * 牛客网 - 二分查找
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param n 数组长度
 * @param v 目标值
 * @param a 有序数组（可能有重复）
 * @return 第一个大于等于 v 的位置（1-based 索引）
 */
public int binarySearch(int n, int v, int[] a) {
    if (a == null || a.length == 0) {
        return n + 1;
    }

    int left = 0, right = n - 1;
    int result = n + 1;  // 默认返回数组长度+1

    while (left <= right) {
        int mid = left + ((right - left) >> 1);

        if (a[mid] >= v) {
            result = mid + 1;  // 1-based 索引
            right = mid - 1;    // 继续向左查找更小的位置
        } else {
            left = mid + 1;
        }
    }
```

```
    return result;
}
```

C++ 实现
```cpp
/**
 * 牛客网 - 二分查找
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param n 数组长度
 * @param v 目标值
 * @param a 有序数组（可能有重复）
 * @return 第一个大于等于 v 的位置（1-based 索引）
 */
int binarySearch(int n, int v, vector<int>& a) {
    if (a.empty()) {
        return n + 1;
    }

    int left = 0, right = n - 1;
    int result = n + 1;  // 默认返回数组长度+1

    while (left <= right) {
        int mid = left + ((right - left) >> 1);

        if (a[mid] >= v) {
            result = mid + 1;  // 1-based 索引
            right = mid - 1;    // 继续向左查找更小的位置
        } else {
            left = mid + 1;
        }
    }

    return result;
}
```

Python 实现
```python
def binary_search(n, v, a):
```

```
    """
    牛客网 - 二分查找

    时间复杂度: O(log n)
    空间复杂度: O(1)

    :param n: 数组长度
    :param v: 目标值
    :param a: 有序数组（可能有重复）
    :return: 第一个大于等于 v 的位置（1-based 索引）
    """
    if not a:
        return n + 1

    left, right = 0, n - 1
    result = n + 1  # 默认返回数组长度+1

    while left <= right:
        mid = left + ((right - left) >> 1)

        if a[mid] >= v:
            result = mid + 1  # 1-based 索引
            right = mid - 1    # 继续向左查找更小的位置
        else:
            left = mid + 1

    return result
```

42. acwing - 二分查找题目

42.1 acwing - 数的范围

题目描述：
给定一个按照升序排列的长度为 n 的整数数组，以及 q 个查询。对于每个查询，返回一个元素 k 的起始位置和终止位置（位置从 0 开始计数）。如果数组中不存在该元素，则返回"-1 -1"。

解题思路：
- 使用两次二分查找
- 第一次查找第一个等于目标值的位置
- 第二次查找最后一个等于目标值的位置

时间复杂度：O(q log n)

空间复杂度：O(1)

Java 实现
```java
/**
 * acwing - 数的范围
 *
 * 时间复杂度: O(q log n)
 * 空间复杂度: O(1)
 *
 * @param nums 升序排列的整数数组
 * @param queries 查询数组
 * @return 每个查询的起始和终止位置
 */
public String[] findRange(int[] nums, int[] queries) {
    int q = queries.length;
    String[] results = new String[q];

    for (int i = 0; i < q; i++) {
        int target = queries[i];

        // 查找第一个等于 target 的位置
        int left = findFirst(nums, target);
        if (left == -1) {
            results[i] = "-1 -1";
            continue;
        }

        // 查找最后一个等于 target 的位置
        int right = findLast(nums, target);
        results[i] = left + " " + right;
    }

    return results;
}

/**
 * 查找第一个等于目标值的位置
 */
private int findFirst(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    int result = -1;
```

```
    while (left <= right) {
        int mid = left + ((right - left) >> 1);

        if (nums[mid] >= target) {
            if (nums[mid] == target) {
                result = mid;
            }
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    return result;
}

/**
 * 查找最后一个等于目标值的位置
 */
private int findLast(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    int result = -1;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);

        if (nums[mid] <= target) {
            if (nums[mid] == target) {
                result = mid;
            }
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return result;
}
```

C++ 实现

```cpp
/**
```

```
 * acwing - 数的范围
 *
 * 时间复杂度: O(q log n)
 * 空间复杂度: O(1)
 *
 * @param nums 升序排列的整数数组
 * @param queries 查询数组
 * @return 每个查询的起始和终止位置
 */
vector<string> findRange(vector<int>& nums, vector<int>& queries) {
    int q = queries.size();
    vector<string> results;

    for (int target : queries) {
        // 查找第一个等于 target 的位置
        int left = findFirst(nums, target);
        if (left == -1) {
            results.push_back("-1 -1");
            continue;
        }

        // 查找最后一个等于 target 的位置
        int right = findLast(nums, target);
        results.push_back(to_string(left) + " " + to_string(right));
    }

    return results;
}

/**
 * 查找第一个等于目标值的位置
 */
int findFirst(vector<int>& nums, int target) {
    int left = 0, right = nums.size() - 1;
    int result = -1;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);

        if (nums[mid] >= target) {
            if (nums[mid] == target) {
                result = mid;
            }
```

```
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    return result;
}

/**
 * 查找最后一个等于目标值的位置
 */
int findLast(vector<int>& nums, int target) {
    int left = 0, right = nums.size() - 1;
    int result = -1;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);

        if (nums[mid] <= target) {
            if (nums[mid] == target) {
                result = mid;
            }
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return result;
}
```

Python 实现
```python
def find_range(nums, queries):
    """
    acwing - 数的范围

    时间复杂度: O(q log n)
    空间复杂度: O(1)

    :param nums: 升序排列的整数数组
```

```
    :param queries: 查询数组
    :return: 每个查询的起始和终止位置
    """
    results = []

    for target in queries:
        # 查找第一个等于 target 的位置
        left = find_first(nums, target)
        if left == -1:
            results.append("-1 -1")
            continue

        # 查找最后一个等于 target 的位置
        right = find_last(nums, target)
        results.append(f"{left} {right}")

    return results

def find_first(nums, target):
    """
    查找第一个等于目标值的位置
    """
    left, right = 0, len(nums) - 1
    result = -1

    while left <= right:
        mid = left + ((right - left) >> 1)

        if nums[mid] >= target:
            if nums[mid] == target:
                result = mid
            right = mid - 1
        else:
            left = mid + 1

    return result

def find_last(nums, target):
    """
    查找最后一个等于目标值的位置
    """
    left, right = 0, len(nums) - 1
    result = -1
```

```
    while left <= right:
        mid = left + ((right - left) >> 1)

        if nums[mid] <= target:
            if nums[mid] == target:
                result = mid
            left = mid + 1
        else:
            right = mid - 1

    return result
```

43. 各大算法平台题目总结

43.1 二分查找算法核心要点总结

算法思想：
- 二分查找是一种在有序数组中查找特定元素的搜索算法
- 每次比较都将搜索范围缩小一半，直到找到目标或搜索范围为空

适用条件：
1. 数组必须是有序的（升序或降序）
2. 数组元素可以通过索引直接访问
3. 查找操作比插入/删除操作更频繁

时间复杂度分析：
- 最好情况：O(1) - 目标在中间位置
- 平均情况：O(log n) - 每次将搜索范围减半
- 最坏情况：O(log n) - 目标不存在或位于边界

空间复杂度：
- 迭代实现：O(1) - 只需要常数级别的额外空间
- 递归实现：O(log n) - 递归调用栈的深度

43.2 二分查找模板总结

模板 1：标准二分查找
```java
int binarySearch(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left <= right) {
```

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

模板 2：查找左边界
```java
int findLeftBound(int[] nums, int target) {
    int left = 0, right = nums.length;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] >= target) right = mid;
        else left = mid + 1;
    }
    return left;
}
```

模板 3：查找右边界
```java
int findRightBound(int[] nums, int target) {
    int left = 0, right = nums.length;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] <= target) left = mid + 1;
        else right = mid;
    }
    return left - 1;
}
```

43.3 二分查找常见问题及解决方案

问题 1：整数溢出
- 解决方案：使用 `mid = left + (right - left) / 2` 而不是 `(left + right) / 2`

问题 2：边界条件处理
- 解决方案：明确搜索区间是左闭右闭 `[left, right]` 还是左闭右开 `[left, right)`

问题 3：重复元素处理
- 解决方案：使用查找左右边界的模板来处理重复元素

问题 4：旋转数组查找
- 解决方案：先判断哪一半是有序的，然后在有序的一半中进行二分查找

43.4 二分查找的工程化应用

1. 数据库索引
- 二分查找用于 B+树索引的查找操作
- 时间复杂度 O(log n)适合大规模数据查询

2. 操作系统
- 内存管理中的空闲块查找
- 文件系统中的目录查找

3. 编译器优化
- 符号表的快速查找
- 常量折叠优化

4. 机器学习
- 超参数调优中的网格搜索
- 模型选择中的二分搜索

43.5 二分查找的进阶应用

1. 二分答案
- 将最优化问题转化为判定问题
- 通过二分查找确定最优解的范围

2. 三分查找
- 用于单峰函数的极值查找
- 每次将搜索范围缩小到原来的 2/3

3. 矩阵搜索
- 在二维有序矩阵中查找元素
- 时间复杂度 O(log(mn))

4. 流数据中的二分查找
- 适用于数据流中的近似查找
- 使用布隆过滤器等数据结构配合二分查找

总结

二分查找是一种非常重要的算法，适用于以下场景：

1. 在有序数组中查找元素
2. 查找边界位置（第一个/最后一个满足条件的元素）
3. 优化搜索空间（二分答案）
4. 在旋转数组中查找元素
5. 查找峰值元素

掌握二分查找的关键在于：
1. 确定搜索区间（开区间还是闭区间）
2. 确定循环条件（left < right 还是 left <= right）
3. 确定边界更新方式（mid+1 还是 mid）
4. 理解不同模板的适用场景

在实际应用中，还需要注意：
1. 数据溢出问题
2. 输入参数验证
3. 边界条件处理
4. 异常情况处理
5. 性能优化

==========

[代码文件]
==========
文件: BinarySearchProblems.cpp
==========
```
/**
 * 二分查找算法实现与相关题目解答 (C++版本)
 *
 * 本文件包含了二分查找的各种实现和在不同场景下的应用
 * 涵盖了 LeetCode、Codeforces、SPOJ 等平台的经典题目
 */

#include <iostream>
#include <algorithm>
#include <climits>
#include <cstdio>
#include <vector>
#include <string>
#include <cmath>
using namespace std;
```

```
// 定义数组最大长度
const int MAXN = 100005;

/**
 * 基础二分查找
 * 在有序数组中查找目标值
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param nums 有序数组
 * @param n 数组长度
 * @param target 目标值
 * @return 目标值的索引，如果不存在返回-1
 */
int search(int nums[], int n, int target) {
    int left = 0, right = n - 1;
    while (left <= right) {
        // 使用位运算避免整数溢出
        int mid = left + ((right - left) >> 1);
        if (nums[mid] == target) {
            return mid;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}



// 函数声明
int mySqrt(int x);
int guessNumber(int n);
int minEatingSpeed(int piles[], int n, int h);
int shipWithinDays(int weights[], int n, int days);
int minBullets(int points[][2], int n);
int guess(int num);

// 新增函数声明
```

```
int minNumberInRotateArray(int nums[], int n);
double cubeRoot(double n);
bool findClosestSum(int A[], int B[], int C[], int L, int M, int N, int X);
int monthlyExpense(int expenses[], int n, int M);
int cutTrees(int trees[], int n, long long M);
int angryCows(int haybales[], int n);
int minimizeMaximum(int nums[], int n, int K);
int jumpStones(int rocks[], int n, int M, int L);
void findWormPiles(int piles[], int n, int queries[], int m, int results[]);

// 主函数用于测试所有二分查找问题
int main() {
    // 测试 LeetCode 69: Sqrt(x)
    cout << "\nLeetCode 69: Sqrt(x)测试:" << endl;
    cout << "sqrt(4) = " << mySqrt(4) << endl;  // 应该输出 2
    cout << "sqrt(8) = " << mySqrt(8) << endl;  // 应该输出 2
    cout << "sqrt(256) = " << mySqrt(256) << endl;  // 应该输出 16
    cout << "sqrt(2147395600) = " << mySqrt(2147395600) << endl;  // 应该输出 46340

    // 测试 LeetCode 374: 猜数字大小
    cout << "\nLeetCode 374: 猜数字大小测试:" << endl;
    cout << "猜数字(10) = " << guessNumber(10) << endl;  // 应该输出 6（假设 pick=6）
    cout << "猜数字(1) = " << guessNumber(1) << endl;  // 应该输出 1

    // 测试 LeetCode 875: 爱吃香蕉的珂珂
    cout << "\nLeetCode 875: 爱吃香蕉的珂珂测试:" << endl;
    int piles1[] = {3, 6, 7, 11};
    int h1 = 8;
    cout << "最小吃香蕉速度(8 小时): " << minEatingSpeed(piles1, 4, h1) << endl;  // 应该输出 4

    int piles2[] = {30, 11, 23, 4, 20};
    int h2 = 5;
    cout << "最小吃香蕉速度(5 小时): " << minEatingSpeed(piles2, 5, h2) << endl;  // 应该输出 30

    // 测试 LeetCode 1011: 在 D 天内送达包裹的能力
    cout << "\nLeetCode 1011: 在 D 天内送达包裹的能力测试:" << endl;
    int weights1[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int days1 = 5;
    cout << "最小运载能力(5 天): " << shipWithinDays(weights1, 10, days1) << endl;  // 应该输出 15

    int weights2[] = {3, 2, 2, 4, 1, 4};
    int days2 = 3;
    cout << "最小运载能力(3 天): " << shipWithinDays(weights2, 6, days2) << endl;  // 应该输出 6
```

```
    // 测试 AtCoder ABC023D: 射撃王
    cout << "\nAtCoder ABC023D: 射撃王测试:" << endl;
    int points1[][2] = {{0, 0}, {1, 1}, {2, 2}, {3, 3}, {4, 4}};
    cout << "最少子弹数: " << minBullets(points1, 5) << endl;  // 应该输出 1

    int points2[][2] = {{0, 0}, {1, 0}, {0, 1}, {1, 1}};
    cout << "最少子弹数: " << minBullets(points2, 4) << endl;  // 应该输出 2

    // 测试新增题目
    cout << "\n=== 新增各大平台题目测试 ===" << endl;

    // 测试牛客网题目: 旋转数组的最小数字
    cout << "\n 牛客网题目: 旋转数组的最小数字测试:" << endl;
    int rotate_nums1[] = {3, 4, 5, 1, 2};
    cout << "旋转数组最小值: " << minNumberInRotateArray(rotate_nums1, 5) << endl;  // 应该输出 1
    int rotate_nums2[] = {2, 2, 2, 0, 1};
    cout << "旋转数组最小值: " << minNumberInRotateArray(rotate_nums2, 5) << endl;  // 应该输出 0

    // 测试 acwing 题目: 数的三次方根
    cout << "\nacwing 题目: 数的三次方根测试:" << endl;
    cout << "8 的三次方根: " << cubeRoot(8.0) << endl;  // 应该输出 2.0
    cout << "27 的三次方根: " << cubeRoot(27.0) << endl;  // 应该输出 3.0
    cout << "1000 的三次方根: " << cubeRoot(1000.0) << endl;  // 应该输出 10.0

    // 测试杭电 OJ 题目: 查找最接近的元素
    cout << "\n 杭电 OJ 题目: 查找最接近的元素测试:" << endl;
    int A[] = {1, 2, 3};
    int B[] = {4, 5, 6};
    int C[] = {7, 8, 9};
    cout << "查找最接近和(15): " << (findClosestSum(A, B, C, 3, 3, 3, 15) ? "true" : "false") <<
endl;  // 应该输出 true

    // 测试 POJ 题目: 月度开销
    cout << "\nPOJ 题目: 月度开销测试:" << endl;
    int expenses[] = {100, 200, 300, 400, 500};
    cout << "月度开销(3 段): " << monthlyExpense(expenses, 5, 3) << endl;  // 应该输出 500

    // 测试洛谷题目: 砍树
    cout << "\n 洛谷题目: 砍树测试:" << endl;
    int trees[] = {20, 15, 10, 17};
    cout << "砍树(7 米): " << cutTrees(trees, 4, 7) << endl;  // 应该输出 15
```

```
    // 测试 USACO 题目: 愤怒的奶牛
    cout << "\nUSACO 题目: 愤怒的奶牛测试:" << endl;
    int haybales[] = {1, 2, 4, 8, 9};
    cout << "愤怒的奶牛: " << angryCows(haybales, 5) << endl;  // 应该输出 4

    // 测试 HackerRank 题目: 最小化最大值
    cout << "\nHackerRank 题目: 最小化最大值测试:" << endl;
    int nums[] = {1, 2, 3, 4, 5};
    cout << "最小化最大值(3 个数): " << minimizeMaximum(nums, 5, 3) << endl;  // 应该输出 2

    // 测试计蒜客题目: 跳石头
    cout << "\n 计蒜客题目: 跳石头测试:" << endl;
    int rocks[] = {2, 11, 14, 17, 21};
    cout << "跳石头(移 2 块,长度 25): " << jumpStones(rocks, 5, 2, 25) << endl;  // 应该输出 4

    // 测试 Codeforces 题目: Worms
    cout << "\nCodeforces 题目: Worms 测试:" << endl;
    int piles[] = {1, 3, 2, 4};
    int queries[] = {1, 4, 7, 10};
    int results[4];
    findWormPiles(piles, 4, queries, 4, results);
    cout << "Worms 查询结果: ";
    for (int i = 0; i < 4; i++) {
        cout << results[i];
        if (i < 3) cout << ", ";
    }
    cout << endl;  // 应该输出 1, 2, 3, 4

    return 0;
}

/**
 * 查找插入位置
 * 在有序数组中查找目标值应该插入的位置
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param nums 有序数组
 * @param n 数组长度
 * @param target 目标值
 * @return 插入位置索引
 */
```

```
int searchInsert(int nums[], int n, int target) {
    int left = 0, right = n;
    while (left < right) {
        int mid = left + ((right - left) >> 1);
        if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return left;
}

/**
 * 辅助方法：查找>=target 的最左位置
 */
int findLeft(int nums[], int n, int target) {
    int l = 0, r = n;
    while (l < r) {
        int m = l + ((r - l) >> 1);
        if (nums[m] < target) {
            l = m + 1;
        } else {
            r = m;
        }
    }
    return l;
}

/**
 * 辅助方法：查找<=target 的最右位置
 */
int findRight(int nums[], int n, int target) {
    int l = 0, r = n;
    while (l < r) {
        int m = l + ((r - l) >> 1);
        if (nums[m] <= target) {
            l = m + 1;
        } else {
            r = m;
        }
    }
    return l - 1;
```

```
}

/**
 * 查找元素的第一个和最后一个位置
 * 在有序数组中查找目标值的起始和结束位置
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param nums 有序数组
 * @param n 数组长度
 * @param target 目标值
 * @param result 返回结果数组，长度为2
 */
void searchRange(int nums[], int n, int target, int result[]) {
    // 边界条件检查
    if (n == 0) {
        result[0] = -1;
        result[1] = -1;
        return;
    }

    int first = findLeft(nums, n, target);
    // 如果找不到>=target的元素，或者该元素不等于target，则说明target不存在
    if (first == n || nums[first] != target) {
        result[0] = -1;
        result[1] = -1;
        return;
    }

    int last = findRight(nums, n, target);
    result[0] = first;
    result[1] = last;
}

/**
 * 有效的完全平方数
 * 判断一个数是否是完全平方数
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param num 正整数
```

```
 * @return 如果是完全平方数返回 true，否则返回 false
 */
bool isPerfectSquare(int num) {
    // 边界条件检查
    if (num < 1) {
        return false;
    }
    if (num == 1) {
        return true;
    }

    long long l = 1, r = num / 2; // 一个数的平方根不会超过它的一半(除了 1)
    while (l <= r) {
        long long m = l + ((r - l) >> 1);
        long long square = m * m;
        if (square == num) {
            return true;
        } else if (square > num) {
            r = m - 1;
        } else {
            l = m + 1;
        }
    }
    return false;
}

/**
 * 模拟 API：判断版本是否错误
 */
bool isBadVersion(int version) {
    // 这里只是一个示例实现
    // 实际应用中会根据具体需求实现
    return version >= 4; // 假设第 4 个版本开始是错误的
}

/**
 * 第一个错误的版本
 * 查找第一个错误的版本
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param n 版本总数
```

```
 * @return 第一个错误的版本号
 */
int firstBadVersion(int n) {
    int left = 1, right = n;
    while (left < right) {
        // 假设存在 isBadVersion API
        int mid = left + ((right - left) >> 1);
        if (isBadVersion(mid)) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    return left;
}

/**
 * 寻找峰值元素
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param nums 数组
 * @param n 数组长度
 * @return 峰值元素的索引
 */
int findPeakElement(int nums[], int n) {
    int left = 0, right = n - 1;
    while (left < right) {
        int mid = left + ((right - left) >> 1);
        if (nums[mid] > nums[mid + 1]) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    return left;
}

/**
 * 寻找旋转排序数组中的最小值
 *
 * 时间复杂度: O(log n)
```

```
 * 空间复杂度: O(1)
 *
 * @param nums 旋转排序数组
 * @param n 数组长度
 * @return 数组中的最小值
 */
int findMin(int nums[], int n) {
    int left = 0, right = n - 1;
    while (left < right) {
        int mid = left + ((right - left) >> 1);
        if (nums[mid] < nums[right]) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    return nums[left];
}

/**
 * 搜索二维矩阵
 *
 * 时间复杂度: O(log(m*n))
 * 空间复杂度: O(1)
 *
 * @param matrix 二维矩阵
 * @param m 矩阵行数
 * @param n 矩阵列数
 * @param target 目标值
 * @return 如果找到目标值返回 true，否则返回 false
 */
bool searchMatrix(int matrix[][MAXN], int m, int n, int target) {
    int left = 0, right = m * n - 1;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);
        int midValue = matrix[mid / n][mid % n];

        if (midValue == target) {
            return true;
        } else if (midValue < target) {
            left = mid + 1;
        } else {
```

```
            right = mid - 1;
        }
    }

    return false;
}

/**
 * 搜索旋转排序数组
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param nums 旋转排序数组
 * @param n 数组长度
 * @param target 目标值
 * @return 目标值的索引，如果不存在返回-1
 */
int searchInRotatedSortedArray(int nums[], int n, int target) {
    int left = 0, right = n - 1;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);

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

        // 左半部分有序
        if (nums[left] <= nums[mid]) {
            if (nums[left] <= target && target < nums[mid]) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
        // 右半部分有序
        else {
            if (nums[mid] < target && target <= nums[right]) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
```

```
        }
    }

    return -1;
}

/**
 * 寻找重复数
 *
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(1)
 *
 * @param nums 包含重复数字的数组
 * @param n 数组长度
 * @return 重复的数字
 */
int findDuplicate(int nums[], int n) {
    int left = 1, right = n - 1;

    while (left < right) {
        int mid = left + ((right - left) >> 1);
        int count = 0;

        // 计算小于等于 mid 的数字个数
        for (int i = 0; i < n; i++) {
            if (nums[i] <= mid) {
                count++;
            }
        }

        // 根据抽屉原理判断重复数字在哪一侧
        if (count > mid) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }

    return left;
}

/**
 * 辅助方法：检查是否能以给定最小距离放置奶牛
```

```
 */
bool canPlaceCows(int positions[], int n, int cows, int minDist) {
    int count = 1;
    int lastPosition = positions[0];

    for (int i = 1; i < n; i++) {
        if (positions[i] - lastPosition >= minDist) {
            count++;
            lastPosition = positions[i];
            if (count == cows) {
                return true;
            }
        }
    }

    return false;
}

/**
 * SPOJ - AGGRCOW (Aggressive cows)
 *
 * 时间复杂度: O(n log(max-min) * n)
 * 空间复杂度: O(1)
 *
 * @param positions 摊位位置数组
 * @param n 位置数组长度
 * @param cows 奶牛数量
 * @return 最大化最小距离
 */
int aggressiveCows(int positions[], int n, int cows) {
    // 假设数组已经排序
    int left = 0, right = positions[n-1] - positions[0];
    int result = 0;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);
        if (canPlaceCows(positions, n, cows, mid)) {
            result = mid;
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
```

```
    return result;
}

/**
 * 带异常处理的安全二分查找
 *
 * @param nums 有序数组
 * @param n 数组长度
 * @param target 目标值
 * @return 目标值的索引，如果不存在返回-1
 */
int safeBinarySearch(int nums[], int n, int target) {
    // 检查数组是否已排序
    for (int i = 1; i < n; i++) {
        if (nums[i] < nums[i-1]) {
            // 在实际应用中可以抛出异常
            // 这里简单返回-1表示错误
            return -1;
        }
    }

    int left = 0, right = n - 1;

    while (left <= right) {
        // 防止整数溢出的中点计算
        int mid = left + ((right - left) >> 1);

        if (nums[mid] == target) {
            return mid;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1;
}

/**
 * LeetCode 69. Sqrt(x)
 *
```

```
 * 时间复杂度: O(log x)
 * 空间复杂度: O(1)
 *
 * @param x 输入的非负整数
 * @return x 的平方根的整数部分
 */
int mySqrt(int x) {
    if (x == 0 || x == 1) {
        return x;
    }

    long long left = 1, right = x;
    while (left <= right) {
        long long mid = left + ((right - left) >> 1);

        // 防止整数溢出，使用 long long 类型
        if (mid * mid > x) {
            right = mid - 1;
        } else {
            // 如果下一个值会溢出或者大于 x，则当前 mid 是最大的有效平方根
            if ((mid + 1) * (mid + 1) > x) {
                return mid;
            }
            left = mid + 1;
        }
    }

    return right;
}

/**
 * LeetCode 374. 猜数字大小
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param n 数字范围上限
 * @return 猜中的数字
 */
int guessNumber(int n) {
    long long left = 1, right = n;

    while (left <= right) {
```

```
        // 防止整数溢出
        long long mid = left + ((right - left) >> 1);
        int result = guess(mid);

        if (result == 0) {
            return mid;  // 猜对了
        } else if (result == 1) {
            left = mid + 1;  // 猜小了，往右边找
        } else {
            right = mid - 1;  // 猜大了，往左边找
        }
    }

    return -1;  // 理论上不会执行到这里
}

// 示例的 guess 函数，实际会由系统提供
int guess(int num) {
    int pick = 6;  // 假设选中的数字是 6
    if (num < pick) return 1;
    else if (num > pick) return -1;
    else return 0;
}

/**
 * 判断以速度 speed 是否能在 h 小时内吃完所有香蕉
 */
bool canFinish(int piles[], int n, int h, int speed) {
    long long time = 0;

    for (int i = 0; i < n; i++) {
        // 计算吃完当前堆需要的时间
        time += (piles[i] + speed - 1) / speed;  // 等价于 ceil(piles[i] / speed)

        // 如果时间已经超过 h，可以提前返回 false
        if (time > h) {
            return false;
        }
    }

    return time <= h;
}
```

```c
/**
 * LeetCode 875. 爱吃香蕉的珂珂
 *
 * 时间复杂度: O(n log maxPile)
 * 空间复杂度: O(1)
 *
 * @param piles 香蕉堆数组
 * @param n 数组长度
 * @param h 警卫离开的时间（小时）
 * @return 最小的吃香蕉速度
 */
int minEatingSpeed(int piles[], int n, int h) {
    int left = 1;
    int right = 0;

    // 找到最大的香蕉堆，作为右边界
    for (int i = 0; i < n; i++) {
        if (piles[i] > right) {
            right = piles[i];
        }
    }

    while (left < right) {
        int mid = left + ((right - left) >> 1);

        if (canFinish(piles, n, h, mid)) {
            right = mid;  // 可以吃完，尝试更小的速度
        } else {
            left = mid + 1;  // 不能吃完，需要更大的速度
        }
    }

    return left;
}

/**
 * 判断以 capacity 的运载能力是否能在 days 天内运完所有包裹
 */
bool canShipInDays(int weights[], int n, int days, int capacity) {
    int currentWeight = 0;
    int dayCount = 1;  // 至少需要 1 天

    for (int i = 0; i < n; i++) {
```

```
        // 如果当前包裹的重量已经超过了运载能力，不可能运完
        if (weights[i] > capacity) {
            return false;
        }

        // 如果当前累计重量加上当前包裹的重量超过了运载能力，需要新的一天
        if (currentWeight + weights[i] > capacity) {
            dayCount++;
            currentWeight = weights[i];  // 新的一天从当前包裹开始

            // 如果天数已经超过了限制，可以提前返回 false
            if (dayCount > days) {
                return false;
            }
        } else {
            currentWeight += weights[i];  // 继续往当前天添加包裹
        }
    }

    return dayCount <= days;
}

/**
 * LeetCode 1011. 在 D 天内送达包裹的能力
 *
 * 时间复杂度: O(n log totalWeight)
 * 空间复杂度: O(1)
 *
 * @param weights 包裹重量数组
 * @param n 数组长度
 * @param days 天数限制
 * @return 船的最小运载能力
 */
int shipWithinDays(int weights[], int n, int days) {
    int left = 0;  // 最小运载能力：最大的单个包裹重量
    int right = 0; // 最大运载能力：所有包裹的总重量

    for (int i = 0; i < n; i++) {
        if (weights[i] > left) {
            left = weights[i];
        }
        right += weights[i];
    }
```

```
    while (left < right) {
        int mid = left + ((right - left) >> 1);

        if (canShipInDays(weights, n, days, mid)) {
            right = mid;  // 可以在 days 天内运完，尝试更小的运载能力
        } else {
            left = mid + 1;  // 不能在 days 天内运完，需要更大的运载能力
        }
    }

    return left;
}

/**
 * 判断三个点是否共线
 */
bool isCollinear(int p1[2], int p2[2], int p3[2]) {
    // 使用叉积判断三点共线
    // (y2 - y1) * (x3 - x1) == (y3 - y1) * (x2 - x1)
    return (p2[1] - p1[1]) * (p3[0] - p1[0]) == (p3[1] - p1[1]) * (p2[0] - p1[0]);
}

/**
 * 递归辅助函数：尝试覆盖剩余的点
 */
bool coverRecursive(int points[][2], bool covered[], int n, int coveredCount, int remainingLines)
{
    if (coveredCount == n) return true;
    if (remainingLines == 0) return false;

    // 找到第一个未覆盖的点
    int first = -1;
    for (int i = 0; i < n; i++) {
        if (!covered[i]) {
            first = i;
            break;
        }
    }

    // 尝试通过第一个未覆盖的点画一条直线
    for (int i = 0; i < n; i++) {
        if (i == first || covered[i]) continue;
```

```
        // 标记所有在这条直线上的点
        bool* newCovered = new bool[n];
        for (int j = 0; j < n; j++) {
            newCovered[j] = covered[j];
        }
        int newCount = coveredCount;

        for (int j = 0; j < n; j++) {
            if (!newCovered[j] && isCollinear(points[first], points[i], points[j])) {
                newCovered[j] = true;
                newCount++;
            }
        }

        bool result = coverRecursive(points, newCovered, n, newCount, remainingLines - 1);
        delete[] newCovered;

        if (result) {
            return true;
        }
    }

    // 如果没有其他点，可以单独用一条直线覆盖第一个未覆盖的点
    return coverRecursive(points, covered, n, coveredCount + 1, remainingLines - 1);
}

/**
 * AtCoder ABC023D - 射撃王 (Shooting King)
 *
 * 时间复杂度: O(N^3 log N)
 * 空间复杂度: O(N)
 *
 * @param points 目标点数组，每个点是[x, y]
 * @param n 点的数量
 * @return 最少需要的子弹数
 */
int minBullets(int points[][2], int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;

    int left = 1, right = n;
    while (left < right) {
```

```
        int mid = left + ((right - left) >> 1);
        bool* covered = new bool[n]();
        bool canCoverAll = coverRecursive(points, covered, n, 0, mid);
        delete[] covered;

        if (canCoverAll) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    return left;
}

/**
 * 性能优化版本的二分查找
 *
 * @param nums 有序数组
 * @param n 数组长度
 * @param target 目标值
 * @return 目标值的索引，如果不存在返回-1
 */
int optimizedBinarySearch(int nums[], int n, int target) {
    int left = 0, right = n - 1;

    while (left <= right) {
        // 使用无符号右移避免溢出
        int mid = (left + right) >> 1;

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

        // 使用位运算优化比较
        if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1;
}
```

```
/**
 * 牛客网: 旋转数组的最小数字
 * 题目来源: https://www.nowcoder.com/practice/9f3231a991af4f55b95579b44b7a01ba
 *
 * 题目描述:
 * 把一个数组最开始的若干个元素搬到数组的末尾,我们称之为数组的旋转。
 * 输入一个非递减排序的数组的一个旋转,输出旋转数组的最小元素。
 *
 * 思路分析:
 * 1. 二分查找法,比较中间元素与右边界元素
 * 2. 如果中间元素小于右边界,最小值在左侧
 * 3. 如果中间元素大于右边界,最小值在右侧
 * 4. 如果相等,右边界左移一位
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
 * @param nums 旋转数组
 * @param n 数组长度
 * @return 最小元素
 */
int minNumberInRotateArray(int nums[], int n) {
    if (n == 0) return -1;

    int left = 0, right = n - 1;

    while (left < right) {
        int mid = left + ((right - left) >> 1);

        if (nums[mid] < nums[right]) {
            right = mid;
        } else if (nums[mid] > nums[right]) {
            left = mid + 1;
        } else {
            right--;
        }
    }

    return nums[left];
}
```

```
/**
 * acwing: 数的三次方根
 * 题目来源: https://www.acwing.com/problem/content/792/
 *
 * 题目描述:
 * 给定一个浮点数 n,求它的三次方根,结果保留 6 位小数。
 *
 * 思路分析:
 * 1. 二分查找法,在[-10000, 10000]范围内搜索
 * 2. 精度控制到 1e-8
 * 3. 根据 mid^3 与 n 的大小关系调整搜索区间
 *
 * 时间复杂度: O(log(范围/精度))
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
 * @param n 输入浮点数
 * @return 三次方根
 */
double cubeRoot(double n) {
    double left = -10000, right = 10000;

    while (right - left > 1e-8) {
        double mid = (left + right) / 2;

        if (mid * mid * mid >= n) {
            right = mid;
        } else {
            left = mid;
        }
    }

    return left;
}

/**
 * 牛客网: 二维数组中的查找
 * 题目来源: https://www.nowcoder.com/practice/abc3fe2ce8e146608e868a70efebf62e
 *
 * 题目描述:
 * 在一个二维数组中,每一行都按照从左到右递增的顺序排序,每一列都按照从上到下递增的顺序排序。
 * 请完成一个函数,输入这样的一个二维数组和一个整数,判断数组中是否含有该整数。
 *
```

```
 * 思路分析:
 * 1. 从右上角开始搜索
 * 2. 如果当前元素等于目标值,返回 true
 * 3. 如果当前元素大于目标值,向左移动
 * 4. 如果当前元素小于目标值,向下移动
 *
 * 时间复杂度: O(m+n)
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
 * @param matrix 二维数组
 * @param rows 行数
 * @param cols 列数
 * @param target 目标值
 * @return 是否找到目标值
 */
bool findIn2DArray(int matrix[][MAXN], int rows, int cols, int target) {
    if (rows == 0 || cols == 0) return false;

    int row = 0, col = cols - 1;

    while (row < rows && col >= 0) {
        if (matrix[row][col] == target) {
            return true;
        } else if (matrix[row][col] > target) {
            col--;
        } else {
            row++;
        }
    }

    return false;
}

/**
 * 牛客网: 数字序列中某一位的数字
 * 题目来源: https://www.nowcoder.com/practice/29311ff7404d44e0b07077f4201418f5
 *
 * 题目描述:
 * 数字以0123456789101112131415...的格式序列化到一个字符序列中。
 * 在这个序列中,第5位(从0开始计数)是5,第13位是1,第19位是4,等等。
 * 请写一个函数,求任意第n位对应的数字。
 *
```

```
 * 思路分析:
 * 1. 确定 n 所在的数字位数范围
 * 2. 确定具体的数字
 * 3. 确定数字中的具体位
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
 * @param n 位置索引
 * @return 对应的数字
 */
int digitAtIndex(int n) {
    if (n < 0) return -1;
    if (n < 10) return n;

    long long digits = 1;  // 当前位数
    long long count = 9;   // 当前位数下的数字个数
    long long start = 1;   // 当前位数下的起始数字

    // 确定 n 所在的数字位数
    while (n > digits * count) {
        n -= digits * count;
        digits++;
        count *= 10;
        start *= 10;
    }

    // 确定具体的数字
    long long num = start + (n - 1) / digits;

    // 确定数字中的具体位
    string numStr = to_string(num);
    return numStr[(n - 1) % digits] - '0';
}

/**
 * 牛客网: 0 到 n-1 中缺失的数字
 * 题目来源: https://www.nowcoder.com/practice/9ce534c8132b4e189fd3130519420cde
 *
 * 题目描述:
 * 一个长度为 n-1 的递增排序数组中的所有数字都是唯一的,并且每个数字都在范围 0 到 n-1 之内。
 * 在范围 0 到 n-1 内的 n 个数字中有且只有一个数字不在该数组中,请找出这个数字。
```

```
 *
 * 思路分析:
 * 1. 二分查找法,比较中间元素的值与索引
 * 2. 如果 nums[mid] == mid,缺失数字在右侧
 * 3. 如果 nums[mid] != mid,缺失数字在左侧
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
 * @param nums 递增排序数组
 * @param n 数组长度(实际为 n-1)
 * @return 缺失的数字
 */
int missingNumber(int nums[], int n) {
    int left = 0, right = n - 1;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);

        if (nums[mid] == mid) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return left;
}

/**
 * 牛客网: 数组中数值和下标相等的元素
 * 题目来源: https://www.nowcoder.com/practice/7bc4a1c7c371425d9faa9d1b511fe193
 *
 * 题目描述:
 * 假设一个单调递增的数组里的每个元素都是整数并且是唯一的。
 * 请编程实现一个函数找出数组中任意一个数值等于其下标的元素。
 *
 * 思路分析:
 * 1. 二分查找法,比较元素值与索引
 * 2. 如果 nums[mid] == mid,找到目标
 * 3. 如果 nums[mid] < mid,目标在右侧
 * 4. 如果 nums[mid] > mid,目标在左侧
```

```
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
 * @param nums 单调递增数组
 * @param n 数组长度
 * @return 数值等于下标的元素,不存在返回-1
 */
int numberEqualIndex(int nums[], int n) {
    int left = 0, right = n - 1;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);

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

    return -1;
}

/**
 * 牛客网: 和为S的两个数字
 * 题目来源: https://www.nowcoder.com/practice/390da4f7a00f44bea7c2f3d19491311b
 *
 * 题目描述:
 * 输入一个递增排序的数组和一个数字S,在数组中查找两个数,使得它们的和正好是S。
 * 如果有多对数字的和等于S,输出两个数的乘积最小的。
 *
 * 思路分析:
 * 1. 双指针法,一个指向开头,一个指向结尾
 * 2. 计算两数之和,与S比较
 * 3. 如果和等于S,记录乘积最小的组合
 * 4. 如果和小于S,左指针右移
 * 5. 如果和大于S,右指针左移
 *
 * 时间复杂度: O(n)
```

```
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
 * @param nums 递增排序数组
 * @param n 数组长度
 * @param S 目标和
 * @param result 结果数组
 */
void findNumbersWithSum(int nums[], int n, int S, int result[]) {
    if (n < 2) {
        result[0] = result[1] = -1;
        return;
    }

    int left = 0, right = n - 1;
    int minProduct = INT_MAX;
    int num1 = -1, num2 = -1;

    while (left < right) {
        int sum = nums[left] + nums[right];

        if (sum == S) {
            int product = nums[left] * nums[right];
            if (product < minProduct) {
                minProduct = product;
                num1 = nums[left];
                num2 = nums[right];
            }
            left++;
            right--;
        } else if (sum < S) {
            left++;
        } else {
            right--;
        }
    }

    result[0] = num1;
    result[1] = num2;
}

/**
 * 牛客网: 和为S的连续正数序列
```

```
 * 题目来源: https://www.nowcoder.com/practice/c451a3fd84b64cb19485dad758a55ebe
 *
 * 题目描述:
 * 输入一个正数 S,打印出所有和为 S 的连续正数序列(至少含有两个数)。
 *
 * 思路分析:
 * 1. 滑动窗口法,维护一个连续序列的窗口
 * 2. 窗口和小于 S 时,右边界右移
 * 3. 窗口和大于 S 时,左边界右移
 * 4. 窗口和等于 S 时,记录序列
 *
 * 时间复杂度: O(S)
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
 * @param S 目标和
 * @param sequences 存储结果的二维数组
 * @param count 序列数量
 */
void findContinuousSequence(int S, int sequences[][MAXN], int* count) {
    *count = 0;
    if (S < 3) return;

    int left = 1, right = 2;
    int sum = 3;

    while (left < right && right <= (S + 1) / 2) {
        if (sum == S) {
            // 记录当前序列
            int len = right - left + 1;
            for (int i = 0; i < len; i++) {
                sequences[*count][i] = left + i;
            }
            (*count)++;
            sum -= left;
            left++;
        } else if (sum < S) {
            right++;
            sum += right;
        } else {
            sum -= left;
            left++;
        }
```

```
    }
}

/**
 * acwing: 数的范围
 * 题目来源: https://www.acwing.com/problem/content/791/
 *
 * 题目描述:
 * 给定一个按照升序排列的长度为 n 的整数数组,以及 q 个查询。
 * 对于每个查询,返回一个元素 k 的起始位置和终止位置(位置从 0 开始计数)。
 * 如果数组中不存在该元素,则返回"-1 -1"。
 *
 * 思路分析:
 * 1. 使用两次二分查找
 * 2. 第一次查找左边界:找到第一个>=k 的位置
 * 3. 第二次查找右边界:找到最后一个<=k 的位置
 * 4. 检查边界是否有效
 *
 * 时间复杂度: O(q * log n)
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
 * @param nums 升序数组
 * @param n 数组长度
 * @param k 查询元素
 * @param result 结果数组[左边界, 右边界]
 */
void findRange(int nums[], int n, int k, int result[]) {
    // 查找左边界
    int left = 0, right = n - 1;
    while (left < right) {
        int mid = left + ((right - left) >> 1);
        if (nums[mid] >= k) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }

    if (nums[left] != k) {
        result[0] = result[1] = -1;
        return;
    }
```

```
    result[0] = left;

    // 查找右边界
    right = n - 1;
    while (left < right) {
        int mid = left + ((right - left) >> 1) + 1;  // 向上取整
        if (nums[mid] <= k) {
            left = mid;
        } else {
            right = mid - 1;
        }
    }

    result[1] = left;
}

/**
 * acwing: 四平方和
 * 题目来源: https://www.acwing.com/problem/content/1223/
 *
 * 题目描述:
 * 四平方和定理,又称为拉格朗日定理:每个正整数都可以表示为至多 4 个正整数的平方和。
 * 如果把 0 包括进去,就正好可以表示为 4 个数的平方和。
 *
 * 思路分析:
 * 1. 预处理所有 a²+b² 的组合
 * 2. 对每个组合进行排序
 * 3. 二分查找是否存在 c²+d² = n - (a²+b²)
 *
 * 时间复杂度: O(n² log n)
 * 空间复杂度: O(n²)
 * 是否最优解: 是
 *
 * @param n 目标正整数
 * @param result 结果数组[a, b, c, d]
 */
void fourSquareSum(int n, int result[]) {
    // 预处理所有 a²+b² 的组合
    struct Pair {
        int sum;
        int a;
        int b;
```

```
    };

    vector<Pair> pairs;
    for (int a = 0; a * a <= n; a++) {
        for (int b = a; a * a + b * b <= n; b++) {
            pairs.push_back({a * a + b * b, a, b});
        }
    }

    // 按和排序
    sort(pairs.begin(), pairs.end(), [](const Pair& p1, const Pair& p2) {
        if (p1.sum != p2.sum) return p1.sum < p2.sum;
        if (p1.a != p2.a) return p1.a < p2.a;
        return p1.b < p2.b;
    });

    // 二分查找
    for (int a = 0; a * a <= n; a++) {
        for (int b = a; a * a + b * b <= n; b++) {
            int target = n - a * a - b * b;

            int left = 0, right = pairs.size() - 1;
            while (left <= right) {
                int mid = left + ((right - left) >> 1);

                if (pairs[mid].sum == target) {
                    result[0] = a;
                    result[1] = b;
                    result[2] = pairs[mid].a;
                    result[3] = pairs[mid].b;
                    return;
                } else if (pairs[mid].sum < target) {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }
        }
    }
}

/**
 * 各大算法平台二分查找题目总结
```

```
 *
 * 工程化考量:
 * 1. 异常处理:空数组、非法输入、边界条件
 * 2. 性能优化:避免整数溢出、位运算优化
 * 3. 可读性:清晰的变量命名、详细的注释
 * 4. 测试覆盖:各种边界情况的测试用例
 *
 * 语言特性差异:
 * - C++:使用 vector、algorithm 库、位运算
 * - Java:使用 ArrayList、Arrays 类
 * - Python:使用列表、内置函数
 *
 * 调试技巧:
 * 1. 打印中间变量值
 * 2. 使用断言验证中间结果
 * 3. 性能分析工具定位瓶颈
 */

/**
 * LeetCode 410. 分割数组的最大值
 * 题目来源: https://leetcode.com/problems/split-array-largest-sum/
 *
 * 题目描述:
 * 给定一个非负整数数组 nums 和一个整数 m,需要将这个数组分成 m 个非空的连续子数组。
 * 设计一个算法使得这 m 个子数组各自和的最大值最小。
 *
 * 思路分析:
 * 1. 二分答案法:答案范围在[max(nums), sum(nums)]之间
 * 2. 对于每个可能的答案 mid,检查能否将数组分成<=m 个子数组且每个子数组和<=mid
 * 3. 如果可以,说明答案可能更小,向左搜索;否则向右搜索
 *
 * 时间复杂度: O(n * log(sum - max))
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
 * @param nums 非负整数数组
 * @param n 数组长度
 * @param m 分割的子数组数量
 * @return 最小的最大子数组和
 */
bool canSplitArray(int nums[], int n, int count, long long maxSum) {
    int splits = 1;
    long long currentSum = 0;
```

```
    for (int i = 0; i < n; i++) {
        if (currentSum + nums[i] > maxSum) {
            splits++;
            currentSum = nums[i];

            if (splits > count) {
                return false;
            }
        } else {
            currentSum += nums[i];
        }
    }

    return true;
}

int splitArray(int nums[], int n, int m) {
    if (n == 0 || m <= 0) {
        return 0;
    }

    long long left = 0;
    long long right = 0;

    for (int i = 0; i < n; i++) {
        if (nums[i] > left) {
            left = nums[i];
        }
        right += nums[i];
    }

    while (left < right) {
        long long mid = left + ((right - left) >> 1);

        if (canSplitArray(nums, n, m, mid)) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }

    return (int) left;
```

```
}

/**
 * LeetCode 4. 寻找两个正序数组的中位数
 * 题目来源: https://leetcode.com/problems/median-of-two-sorted-arrays/
 *
 * 题目描述:
 * 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。
 *
 * 思路分析:
 * 1. 使用二分查找在较短的数组上进行分割
 * 2. 确保左半部分的最大值 <= 右半部分的最小值
 * 3. 中位数由左半部分最大值和右半部分最小值决定
 *
 * 时间复杂度: O(log(min(m,n)))
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
 * @param nums1 第一个正序数组
 * @param m nums1 的长度
 * @param nums2 第二个正序数组
 * @param n nums2 的长度
 * @return 两个数组的中位数
 */
double findMedianSortedArrays(int nums1[], int m, int nums2[], int n) {
    // 确保 nums1 是较短的数组
    if (m > n) {
        return findMedianSortedArrays(nums2, n, nums1, m);
    }

    int left = 0, right = m;

    while (left <= right) {
        int partition1 = left + ((right - left) >> 1);
        int partition2 = ((m + n + 1) >> 1) - partition1;

        int maxLeft1 = (partition1 == 0) ? INT_MIN : nums1[partition1 - 1];
        int minRight1 = (partition1 == m) ? INT_MAX : nums1[partition1];

        int maxLeft2 = (partition2 == 0) ? INT_MIN : nums2[partition2 - 1];
        int minRight2 = (partition2 == n) ? INT_MAX : nums2[partition2];
```

```
        if (maxLeft1 <= minRight2 && maxLeft2 <= minRight1) {
            if ((m + n) % 2 == 0) {
                return (max(maxLeft1, maxLeft2) + min(minRight1, minRight2)) / 2.0;
            } else {
                return max(maxLeft1, maxLeft2);
            }
        } else if (maxLeft1 > minRight2) {
            right = partition1 - 1;
        } else {
            left = partition1 + 1;
        }
    }

    return 0.0;
}

/**
 * 牛客/剑指 Offer: 数字在排序数组中出现的次数
 * 题目来源: https://www.nowcoder.com/practice/70610bf967994b22bb1c26f9ae901fa2
 *
 * 题目描述:
 * 统计一个数字在排序数组中出现的次数。
 *
 * 思路分析:
 * 使用二分查找找到目标值的左右边界,次数 = 右边界 - 左边界 + 1
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
 * @param nums 排序数组
 * @param n 数组长度
 * @param target 目标值
 * @return 出现次数
 */
int getNumberOfK(int nums[], int n, int target) {
    if (n == 0) {
        return 0;
    }

    int first = findLeft(nums, n, target);
    if (first == n || nums[first] != target) {
        return 0;
```

```
    }

    int last = findRight(nums, n, target);
    return last - first + 1;
}

/**
 * LeetCode 540. 有序数组中的单一元素
 * 题目来源: https://leetcode.com/problems/single-element-in-a-sorted-array/
 *
 * 题目描述:
 * 给定一个只包含整数的有序数组,每个元素都会出现两次,唯有一个数只会出现一次,找出这个数。
 *
 * 思路分析:
 * 1. 单一元素之前,成对元素的第一个位置是偶数索引
 * 2. 单一元素之后,成对元素的第一个位置是奇数索引
 * 3. 使用二分查找定位单一元素
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
 * @param nums 有序数组
 * @param n 数组长度
 * @return 单一元素
 */
int singleNonDuplicate(int nums[], int n) {
    int left = 0, right = n - 1;

    while (left < right) {
        int mid = left + ((right - left) >> 1);

        // 确保 mid 是偶数索引
        if (mid % 2 == 1) {
            mid--;
        }

        // 检查成对关系
        if (nums[mid] == nums[mid + 1]) {
            // 单一元素在右侧
            left = mid + 2;
        } else {
            // 单一元素在左侧或就是 mid
```

```
            right = mid;
        }
    }

    return nums[left];
}

/**
 * LeetCode 1482. 制作 m 束花所需的最少天数
 * 题目来源: https://leetcode.com/problems/minimum-number-of-days-to-make-m-bouquets/
 *
 * 题目描述:
 * 给你一个整数数组 bloomDay,以及两个整数 m 和 k。
 * 现需要制作 m 束花。制作花束时,需要使用花园中相邻的 k 朵花。
 * 花园中有 n 朵花,第 i 朵花会在 bloomDay[i] 时盛开。
 * 请你返回从花园中摘 m 束花需要等待的最少的天数。如果不能摘到 m 束花则返回 -1。
 *
 * 思路分析:
 * 1. 二分答案:天数范围在[min(bloomDay), max(bloomDay)]之间
 * 2. 对于每个天数,检查能否制作 m 束花
 * 3. 贪心验证:从左到右扫描,统计连续 k 朵已开花的数量
 *
 * 时间复杂度: O(n * log(max - min))
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
 * @param bloomDay 每朵花开放的天数
 * @param n 数组长度
 * @param m 需要的花束数量
 * @param k 每束花包含的花朵数量
 * @return 最少等待天数
 */
bool canMakeBouquets(int bloomDay[], int n, int m, int k, int day) {
    int bouquets = 0;
    int flowers = 0;

    for (int i = 0; i < n; i++) {
        if (bloomDay[i] <= day) {
            flowers++;
            if (flowers == k) {
                bouquets++;
                flowers = 0;
                if (bouquets == m) {
```

```
                    return true;
                }
            }
        } else {
            flowers = 0;
        }
    }

    return bouquets >= m;
}

int minDays(int bloomDay[], int n, int m, int k) {
    if ((long long) m * k > n) {
        return -1;
    }

    int left = INT_MAX;
    int right = INT_MIN;

    for (int i = 0; i < n; i++) {
        if (bloomDay[i] < left) {
            left = bloomDay[i];
        }
        if (bloomDay[i] > right) {
            right = bloomDay[i];
        }
    }

    while (left < right) {
        int mid = left + ((right - left) >> 1);

        if (canMakeBouquets(bloomDay, n, m, k, mid)) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }

    return left;
}

/**
 * 杭电 0J: 查找最接近的元素
```

```
 * 题目来源: http://acm.hdu.edu.cn/showproblem.php?pid=2141
 *
 * 题目描述:
 * 给定三个数组 A、B、C 和一个目标值 X，判断是否存在三个数 a∈A, b∈B, c∈C 使得 a+b+c=X。
 *
 * 思路分析:
 * 1. 先计算 A+B 的所有可能和，存储在数组中并排序
 * 2. 对于每个 c∈C，在 A+B 的和数组中查找 X-c
 * 3. 使用二分查找提高查找效率
 *
 * 时间复杂度: O(L*M + N*log(L*M))
 * 空间复杂度: O(L*M)
 * 是否最优解: 是
 *
 * @param A 数组 A
 * @param B 数组 B
 * @param C 数组 C
 * @param L 数组 A 长度
 * @param M 数组 B 长度
 * @param N 数组 C 长度
 * @param X 目标值
 * @return 是否存在满足条件的三元组
 */
bool findClosestSum(int A[], int B[], int C[], int L, int M, int N, int X) {
    // 计算 A+B 的所有可能和
    int* sumAB = new int[L * M];
    int index = 0;

    for (int i = 0; i < L; i++) {
        for (int j = 0; j < M; j++) {
            sumAB[index++] = A[i] + B[j];
        }
    }

    // 排序 A+B 的和数组
    sort(sumAB, sumAB + index);

    // 对于每个 c∈C，在 A+B 的和数组中查找 X-c
    for (int i = 0; i < N; i++) {
        int target = X - C[i];

        // 使用二分查找
        int left = 0, right = index - 1;
```

```
        while (left <= right) {
            int mid = left + ((right - left) >> 1);

            if (sumAB[mid] == target) {
                delete[] sumAB;
                return true;
            } else if (sumAB[mid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
    }

    delete[] sumAB;
    return false;
}

/**
 * POJ: 月度开销
 * 题目来源: http://poj.org/problem?id=3273
 *
 * 题目描述:
 * FJ有N天的账单，他想把这N天分成M个月，使得每个月的开销之和的最大值最小。
 *
 * 思路分析:
 * 1. 二分答案法，答案范围在[max(expenses), sum(expenses)]之间
 * 2. 对于每个可能的答案mid，检查能否将账单分成<=M个月且每个月开销<=mid
 * 3. 如果可以，说明答案可能更小，向左搜索；否则向右搜索
 *
 * 时间复杂度: O(N * log(sum - max))
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
 * @param expenses 每天的开销数组
 * @param n 数组长度
 * @param M 月份数量
 * @return 最小的最大月开销
 */
bool canDivideExpenses(int expenses[], int n, int M, int maxExpense) {
    int months = 1;
    int currentExpense = 0;
```

```
    for (int i = 0; i < n; i++) {
        if (currentExpense + expenses[i] > maxExpense) {
            months++;
            currentExpense = expenses[i];

            if (months > M) {
                return false;
            }
        } else {
            currentExpense += expenses[i];
        }
    }

    return true;
}

int monthlyExpense(int expenses[], int n, int M) {
    int left = 0;
    int right = 0;

    for (int i = 0; i < n; i++) {
        if (expenses[i] > left) {
            left = expenses[i];
        }
        right += expenses[i];
    }

    while (left < right) {
        int mid = left + ((right - left) >> 1);

        if (canDivideExpenses(expenses, n, M, mid)) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }

    return left;
}

/**
 * 洛谷: 砍树
 * 题目来源: https://www.luogu.com.cn/problem/P1873
```

```
 *
 * 题目描述:
 * 有 N 棵树，第 i 棵树的高度是 Hi。伐木工人需要获得至少 M 米的木材。
 * 伐木工人会选择一个高度 H，将所有高度大于 H 的树砍掉高于 H 的部分。
 * 求伐木工人能选择的最大 H 值。
 *
 * 思路分析:
 * 1. 二分答案法，答案范围在[0, max(trees)]之间
 * 2. 对于每个可能的高度 H，计算能获得的木材总量
 * 3. 如果总量>=M，说明 H 可以更大，向右搜索；否则向左搜索
 *
 * 时间复杂度: O(N * log(maxHeight))
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
 * @param trees 树的高度数组
 * @param n 数组长度
 * @param M 需要的木材量
 * @return 最大的切割高度 H
 */
long long getWood(int trees[], int n, int H) {
    long long total = 0;

    for (int i = 0; i < n; i++) {
        if (trees[i] > H) {
            total += trees[i] - H;
        }
    }

    return total;
}

int cutTrees(int trees[], int n, long long M) {
    int left = 0;
    int right = 0;

    for (int i = 0; i < n; i++) {
        if (trees[i] > right) {
            right = trees[i];
        }
    }

    while (left < right) {
```

```
        int mid = left + ((right - left + 1) >> 1);

        if (getWood(trees, n, mid) >= M) {
            left = mid;
        } else {
            right = mid - 1;
        }
    }

    return left;
}

/**
 * USACO: 愤怒的奶牛
 * 题目来源: http://www.usaco.org/index.php?page=viewproblem2&cpid=764
 *
 * 题目描述:
 * 在一条线上有 N 个草包，第 i 个草包在位置 xi。愤怒的奶牛可以选择引爆一个草包，
 * 爆炸半径 R 会引爆所有距离<=R 的草包。求引爆所有草包所需的最小爆炸半径。
 *
 * 思路分析:
 * 1. 二分答案法，答案范围在[0, max(haybales)-min(haybales)]之间
 * 2. 对于每个可能的半径 R，检查是否能引爆所有草包
 * 3. 贪心策略：从最左边开始，每次引爆能覆盖最远位置的草包
 *
 * 时间复杂度: O(N * log(max-min))
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
 * @param haybales 草包位置数组
 * @param n 数组长度
 * @return 最小爆炸半径
 */
bool canCoverAll(int haybales[], int n, int radius) {
    int covered = 0;

    for (int i = 0; i < n; i++) {
        if (haybales[i] > covered) {
            // 需要引爆新的草包
            covered = haybales[i] + radius;
        } else if (haybales[i] + radius > covered) {
            // 更新覆盖范围
            covered = haybales[i] + radius;
```

```
        }
    }

    // 检查是否覆盖了所有草包
    return covered >= haybales[n - 1];
}

int angryCows(int haybales[], int n) {
    sort(haybales, haybales + n);

    int left = 0;
    int right = haybales[n - 1] - haybales[0];

    while (left < right) {
        int mid = left + ((right - left) >> 1);

        if (canCoverAll(haybales, n, mid)) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }

    return left;
}

/**
 * HackerRank: 最小化最大值
 * 题目来源: https://www.hackerrank.com/challenges/angry-children/problem
 *
 * 题目描述:
 * 给定 N 个整数和一个整数 K，从 N 个整数中选择 K 个数，使得这 K 个数中的最大值与最小值的差最小。
 *
 * 思路分析:
 * 1. 先对数组排序
 * 2. 使用滑动窗口，检查每个长度为 K 的连续子数组
 * 3. 计算每个子数组的最大值与最小值的差（即最后一个元素与第一个元素的差）
 * 4. 返回最小差值
 *
 * 时间复杂度: O(N log N)
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
```

```
 * @param nums 整数数组
 * @param n 数组长度
 * @param K 选择的数字个数
 * @return 最小差值
 */
int minimizeMaximum(int nums[], int n, int K) {
    sort(nums, nums + n);

    int minDiff = nums[K - 1] - nums[0];

    for (int i = 1; i <= n - K; i++) {
        int diff = nums[i + K - 1] - nums[i];
        if (diff < minDiff) {
            minDiff = diff;
        }
    }

    return minDiff;
}

/**
 * 计蒜客: 跳石头
 * 题目来源: https://nanti.jisuanke.com/t/T1643
 *
 * 题目描述:
 * 一条河上有一排石头，从起点到终点的距离是 L。有 N 块石头在河中，位置分别为 ai。
 * 可以移走 M 块石头，求移走 M 块石头后，最短跳跃距离的最大值。
 *
 * 思路分析:
 * 1. 二分答案法，答案范围在[0, L]之间
 * 2. 对于每个可能的距离 D，检查是否能通过移走<=M 块石头实现最小跳跃距离>=D
 * 3. 贪心策略：从起点开始，每次跳到距离>=D 的最近石头
 * 4. 如果必须移走的石头数>M，则不能实现；否则可以实现
 *
 * 时间复杂度: O(N * log L)
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
 * @param rocks 石头位置数组
 * @param n 石头数量
 * @param M 可以移走的石头数量
 * @param L 河的长度
 * @return 最短跳跃距离的最大值
```

```
 */
bool canAchieveMinDistance(int rocks[], int n, int M, int L, int minDist) {
    int removed = 0;
    int lastPos = 0;

    for (int i = 0; i < n; i++) {
        if (rocks[i] - lastPos < minDist) {
            // 必须移走这块石头
            removed++;
            if (removed > M) {
                return false;
            }
        } else {
            // 保留这块石头
            lastPos = rocks[i];
        }
    }

    // 检查终点距离
    if (L - lastPos < minDist) {
        removed++;
        if (removed > M) {
            return false;
        }
    }

    return true;
}

int jumpStones(int rocks[], int n, int M, int L) {
    sort(rocks, rocks + n);

    int left = 0;
    int right = L;

    while (left < right) {
        int mid = left + ((right - left + 1) >> 1);

        if (canAchieveMinDistance(rocks, n, M, L, mid)) {
            left = mid;
        } else {
            right = mid - 1;
        }
```

```
    }

    return left;
}

/**
 * Codeforces 474B - Worms
 * 题目来源: https://codeforces.com/problemset/problem/474/B
 *
 * 题目描述:
 * 有N堆虫子，第i堆有ai只虫子。所有虫子按顺序编号从1开始。
 * 有M个查询，每个查询给出一个虫子编号，求这只虫子属于第几堆。
 *
 * 思路分析:
 * 1. 先计算前缀和数组
 * 2. 对于每个查询，使用二分查找找到第一个>=查询编号的前缀和位置
 * 3. 该位置即为虫子所属的堆
 *
 * 时间复杂度: O(N + M * log N)
 * 空间复杂度: O(N)
 * 是否最优解: 是
 *
 * @param piles 每堆虫子数量数组
 * @param n 数组长度
 * @param queries 查询数组
 * @param m 查询数量
 * @param results 结果数组
 */
void findWormPiles(int piles[], int n, int queries[], int m, int results[]) {
    // 计算前缀和
    int* prefixSum = new int[n + 1];
    prefixSum[0] = 0;

    for (int i = 0; i < n; i++) {
        prefixSum[i + 1] = prefixSum[i] + piles[i];
    }

    // 处理每个查询
    for (int i = 0; i < m; i++) {
        int wormId = queries[i];

        // 二分查找第一个>=wormId的前缀和位置
        int left = 1, right = n;
```

```
        while (left < right) {
            int mid = left + ((right - left) >> 1);
            if (prefixSum[mid] >= wormId) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }

        results[i] = left;
    }

    delete[] prefixSum;
}
```

==========

文件: BinarySearchProblems.java

==========

```
import java.util.*;

/**
 * 二分查找算法实现与相关题目解答 (Java 版本)
 *
 * 本文件包含了二分查找的各种实现和在不同场景下的应用
 * 涵盖了 LeetCode、Codeforces、SPOJ 等平台的经典题目
 */
public class BinarySearchProblems {

    /**
     * 基础二分查找
     * 在有序数组中查找目标值
     *
     * 时间复杂度: O(log n)
     * 空间复杂度: O(1)
     *
     * @param nums 有序数组
     * @param target 目标值
     * @return 目标值的索引，如果不存在返回-1
     */
    public static int search(int[] nums, int target) {
        // 边界条件检查
        if (nums == null || nums.length == 0) {
```

```
        return -1;
    }

    int left = 0, right = nums.length - 1;
    while (left <= right) {
        // 使用位运算避免整数溢出
        int mid = left + ((right - left) >> 1);
        if (nums[mid] == target) {
            return mid;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}

/**
 * 查找插入位置
 * 在有序数组中查找目标值应该插入的位置
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param nums 有序数组
 * @param target 目标值
 * @return 插入位置索引
 */
public static int searchInsert(int[] nums, int target) {
    // 边界条件检查
    if (nums == null || nums.length == 0) {
        return 0;
    }

    int left = 0, right = nums.length;
    while (left < right) {
        int mid = left + ((right - left) >> 1);
        if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid;
        }
```

```
    }
    return left;
}

/**
 * 查找元素的第一个和最后一个位置
 * 在有序数组中查找目标值的起始和结束位置
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param nums 有序数组
 * @param target 目标值
 * @return [起始位置, 结束位置]，如果不存在返回[-1, -1]
 */
public static int[] searchRange(int[] nums, int target) {
    // 边界条件检查
    if (nums == null || nums.length == 0) {
        return new int[]{-1, -1};
    }

    int first = findLeft(nums, target);
    // 如果找不到>=target 的元素，或者该元素不等于 target，则说明 target 不存在
    if (first == -1 || nums[first] != target) {
        return new int[]{-1, -1};
    }

    int last = findRight(nums, target);
    return new int[]{first, last};
}

/**
 * 辅助方法：查找>=target 的最左位置
 */
private static int findLeft(int[] nums, int target) {
    int l = 0, r = nums.length - 1;
    int ans = -1;
    while (l <= r) {
        int m = l + ((r - l) >> 1);
        if (nums[m] >= target) {
            ans = m;
            r = m - 1;
        } else {
```

```
            l = m + 1;
        }
    }
    return ans;
}

/**
 * 辅助方法：查找<=target 的最右位置
 */
private static int findRight(int[] nums, int target) {
    int l = 0, r = nums.length - 1;
    int ans = -1;
    while (l <= r) {
        int m = l + ((r - l) >> 1);
        if (nums[m] <= target) {
            ans = m;
            l = m + 1;
        } else {
            r = m - 1;
        }
    }
    return ans;
}

/**
 * 有效的完全平方数
 * 判断一个数是否是完全平方数
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param num 正整数
 * @return 如果是完全平方数返回 true，否则返回 false
 */
public static boolean isPerfectSquare(int num) {
    // 边界条件检查
    if (num < 1) {
        return false;
    }
    if (num == 1) {
        return true;
    }
```

```
    long l = 1, r = num / 2; // 一个数的平方根不会超过它的一半(除了 1)
    while (l <= r) {
        long m = l + ((r - l) >> 1);
        long square = m * m;
        if (square == num) {
            return true;
        } else if (square > num) {
            r = m - 1;
        } else {
            l = m + 1;
        }
    }
    return false;
}

/**
 * 第一个错误的版本
 * 查找第一个错误的版本
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param n 版本总数
 * @return 第一个错误的版本号
 */
public static int firstBadVersion(int n) {
    int left = 1, right = n;
    while (left < right) {
        // 假设存在 isBadVersion API
        int mid = left + ((right - left) >> 1);
        if (isBadVersion(mid)) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    return left;
}

/**
 * 模拟 API：判断版本是否错误
 */
private static boolean isBadVersion(int version) {
```

```
    // 这里只是一个示例实现
    // 实际应用中会根据具体需求实现
    return version >= 4; // 假设第 4 个版本开始是错误的
}

/**
 * 寻找峰值元素
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param nums 数组
 * @return 峰值元素的索引
 */
public static int findPeakElement(int[] nums) {
    int left = 0, right = nums.length - 1;
    while (left < right) {
        int mid = left + ((right - left) >> 1);
        if (nums[mid] > nums[mid + 1]) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    return left;
}

/**
 * 寻找旋转排序数组中的最小值
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param nums 旋转排序数组
 * @return 数组中的最小值
 */
public static int findMin(int[] nums) {
    int left = 0, right = nums.length - 1;
    while (left < right) {
        int mid = left + ((right - left) >> 1);
        if (nums[mid] < nums[right]) {
            right = mid;
        } else {
```

```
            left = mid + 1;
        }
    }
    return nums[left];
}

/**
 * 搜索二维矩阵
 *
 * 时间复杂度: O(log(m*n))
 * 空间复杂度: O(1)
 *
 * @param matrix 二维矩阵
 * @param target 目标值
 * @return 如果找到目标值返回 true，否则返回 false
 */
public static boolean searchMatrix(int[][] matrix, int target) {
    // 边界条件检查
    if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
        return false;
    }

    int m = matrix.length, n = matrix[0].length;
    int left = 0, right = m * n - 1;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);
        int midValue = matrix[mid / n][mid % n];

        if (midValue == target) {
            return true;
        } else if (midValue < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return false;
}

/**
 * 搜索旋转排序数组
```

```
     *
     * 时间复杂度: O(log n)
     * 空间复杂度: O(1)
     *
     * @param nums 旋转排序数组
     * @param target 目标值
     * @return 目标值的索引，如果不存在返回-1
     */
    public static int searchInRotatedSortedArray(int[] nums, int target) {
        int left = 0, right = nums.length - 1;

        while (left <= right) {
            int mid = left + ((right - left) >> 1);

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

            // 左半部分有序
            if (nums[left] <= nums[mid]) {
                if (nums[left] <= target && target < nums[mid]) {
                    right = mid - 1;
                } else {
                    left = mid + 1;
                }
            }
            // 右半部分有序
            else {
                if (nums[mid] < target && target <= nums[right]) {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }
        }

        return -1;
    }

    /**
     * 寻找重复数
     *
     * 时间复杂度: O(n log n)
```

```
     * 空间复杂度: O(1)
     *
     * @param nums 包含重复数字的数组
     * @return 重复的数字
     */
    public static int findDuplicate(int[] nums) {
        int left = 1, right = nums.length - 1;

        while (left < right) {
            int mid = left + ((right - left) >> 1);
            int count = 0;

            // 计算小于等于 mid 的数字个数
            for (int num : nums) {
                if (num <= mid) {
                    count++;
                }
            }

            // 根据抽屉原理判断重复数字在哪一侧
            if (count > mid) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }

        return left;
    }

    /**
     * Codeforces 706B. Interesting drink
     *
     * 时间复杂度: O(n log n + q log n)
     * 空间复杂度: O(1)
     *
     * @param prices 饮料价格数组
     * @param queries 查询金额数组
     * @return 每个查询可以购买的饮料种类数
     */
    public static int[] howManyDrinks(int[] prices, int[] queries) {
        Arrays.sort(prices);
        int[] result = new int[queries.length];
```

```
    for (int i = 0; i < queries.length; i++) {
        int money = queries[i];
        int left = 0, right = prices.length;

        while (left < right) {
            int mid = left + ((right - left) >> 1);
            if (prices[mid] <= money) {
                left = mid + 1;
            } else {
                right = mid;
            }
        }

        result[i] = left;
    }

    return result;
}

/**
 * SPOJ - AGGRCOW (Aggressive cows)
 *
 * 时间复杂度: O(n log(max-min) * n)
 * 空间复杂度: O(1)
 *
 * @param positions 摊位位置数组
 * @param cows 奶牛数量
 * @return 最大化最小距离
 */
public static int aggressiveCows(int[] positions, int cows) {
    Arrays.sort(positions);
    int left = 0, right = positions[positions.length - 1] - positions[0];
    int result = 0;

    while (left <= right) {
        int mid = left + ((right - left) >> 1);
        if (canPlaceCows(positions, cows, mid)) {
            result = mid;
            left = mid + 1;
        } else {
            right = mid - 1;
        }
```

```
    }

    return result;
}

/**
 * 辅助方法：检查是否能以给定最小距离放置奶牛
 */
private static boolean canPlaceCows(int[] positions, int cows, int minDist) {
    int count = 1;
    int lastPosition = positions[0];

    for (int i = 1; i < positions.length; i++) {
        if (positions[i] - lastPosition >= minDist) {
            count++;
            lastPosition = positions[i];
            if (count == cows) {
                return true;
            }
        }
    }

    return false;
}

/**
 * 带异常处理的安全二分查找
 *
 * @param nums 有序数组
 * @param target 目标值
 * @return 目标值的索引，如果不存在返回-1
 * @throws IllegalArgumentException 当输入参数非法时抛出异常
 */
public static int safeBinarySearch(int[] nums, int target) {
    // 检查输入参数
    if (nums == null) {
        throw new IllegalArgumentException("数组不能为null");
    }

    if (nums.length == 0) {
        return -1;
    }
```

```
    int left = 0, right = nums.length - 1;

    // 检查数组是否已排序
    for (int i = 1; i < nums.length; i++) {
        if (nums[i] < nums[i-1]) {
            throw new IllegalArgumentException("数组必须是已排序的");
        }
    }

    while (left <= right) {
        // 防止整数溢出的中点计算
        int mid = left + ((right - left) >> 1);

        if (nums[mid] == target) {
            return mid;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1;
}

/**
 * 性能优化版本的二分查找
 *
 * @param nums 有序数组
 * @param target 目标值
 * @return 目标值的索引，如果不存在返回-1
 */
public static int optimizedBinarySearch(int[] nums, int target) {
    int left = 0, right = nums.length - 1;

    while (left <= right) {
        // 使用无符号右移避免溢出
        int mid = (left + right) >>> 1;

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

```
        // 使用位运算优化比较
        if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1;
}

/**
 * LeetCode 69. Sqrt(x)
 *
 * 时间复杂度: O(log x)
 * 空间复杂度: O(1)
 *
 * @param x 输入的非负整数
 * @return x 的平方根的整数部分
 */
public static int mySqrt(int x) {
    if (x == 0 || x == 1) {
        return x;
    }

    int left = 1, right = x;
    while (left <= right) {
        int mid = left + ((right - left) >> 1);

        // 防止整数溢出，使用除法而不是乘法
        if (mid > x / mid) {
            right = mid - 1;
        } else {
            // 如果下一个值会溢出或者大于 x/mid，则当前 mid 是最大的有效平方根
            if (mid + 1 > x / (mid + 1)) {
                return mid;
            }
            left = mid + 1;
        }
    }

    return right;
}
```

```java
/**
 * LeetCode 374. 猜数字大小
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * @param n 数字范围上限
 * @return 猜中的数字
 */
public static int guessNumber(int n) {
    int left = 1, right = n;

    while (left <= right) {
        // 防止整数溢出
        int mid = left + ((right - left) >> 1);
        int result = guess(mid);

        if (result == 0) {
            return mid;  // 猜对了
        } else if (result == 1) {
            left = mid + 1;  // 猜小了，往右边找
        } else {
            right = mid - 1;  // 猜大了，往左边找
        }
    }

    return -1;  // 理论上不会执行到这里
}

// 示例的 guess 方法，实际会由系统提供
private static int guess(int num) {
    int pick = 6;  // 假设选中的数字是 6
    if (num < pick) return 1;
    else if (num > pick) return -1;
    else return 0;
}

/**
 * LeetCode 875. 爱吃香蕉的珂珂
 *
 * 时间复杂度: O(n log maxPile)
 * 空间复杂度: O(1)
```

```
     *
     * @param piles 香蕉堆数组
     * @param h 警卫离开的时间（小时）
     * @return 最小的吃香蕉速度
     */
    public static int minEatingSpeed(int[] piles, int h) {
        int left = 1;
        int right = 0;

        // 找到最大的香蕉堆，作为右边界
        for (int pile : piles) {
            right = Math.max(right, pile);
        }

        while (left < right) {
            int mid = left + ((right - left) >> 1);

            if (canFinish(piles, h, mid)) {
                right = mid;  // 可以吃完，尝试更小的速度
            } else {
                left = mid + 1;  // 不能吃完，需要更大的速度
            }
        }

        return left;
    }

    /**
     * 辅助方法：判断以速度 speed 是否能在 h 小时内吃完所有香蕉
     */
    private static boolean canFinish(int[] piles, int h, int speed) {
        long time = 0;

        for (int pile : piles) {
            // 计算吃完当前堆需要的时间
            time += (pile + speed - 1) / speed;  // 等价于 Math.ceil(pile / speed)

            // 如果时间已经超过 h，可以提前返回 false
            if (time > h) {
                return false;
            }
        }
```

```
        return time <= h;
    }

    /**
     * LeetCode 1011. 在 D 天内送达包裹的能力
     *
     * 时间复杂度: O(n log totalWeight)
     * 空间复杂度: O(1)
     *
     * @param weights 包裹重量数组
     * @param days 天数限制
     * @return 船的最小运载能力
     */
    public static int shipWithinDays(int[] weights, int days) {
        int left = 0;  // 最小运载能力：最大的单个包裹重量
        int right = 0; // 最大运载能力：所有包裹的总重量

        for (int weight : weights) {
            left = Math.max(left, weight);
            right += weight;
        }

        while (left < right) {
            int mid = left + ((right - left) >> 1);

            if (canShipInDays(weights, days, mid)) {
                right = mid;  // 可以在 days 天内运完，尝试更小的运载能力
            } else {
                left = mid + 1;  // 不能在 days 天内运完，需要更大的运载能力
            }
        }

        return left;
    }

    /**
     * 辅助方法：判断以 capacity 的运载能力是否能在 days 天内运完所有包裹
     */
    private static boolean canShipInDays(int[] weights, int days, int capacity) {
        int currentWeight = 0;
        int dayCount = 1;  // 至少需要 1 天

        for (int weight : weights) {
```

```
            // 如果当前包裹的重量已经超过了运载能力，不可能运完
            if (weight > capacity) {
                return false;
            }

            // 如果当前累计重量加上当前包裹的重量超过了运载能力，需要新的一天
            if (currentWeight + weight > capacity) {
                dayCount++;
                currentWeight = weight;  // 新的一天从当前包裹开始

                // 如果天数已经超过了限制，可以提前返回 false
                if (dayCount > days) {
                    return false;
                }
            } else {
                currentWeight += weight;  // 继续往当前天添加包裹
            }
        }

        return dayCount <= days;
    }

    /**
     * AtCoder ABC023D - 射撃王 (Shooting King)
     *
     * 时间复杂度: O(N^3 log N)
     * 空间复杂度: O(N)
     *
     * @param points 目标点数组，每个点是[x, y]
     * @return 最少需要的子弹数
     */
    public static int minBullets(int[][] points) {
        int n = points.length;
        if (n == 0) return 0;
        if (n == 1) return 1;

        int left = 1, right = n;
        while (left < right) {
            int mid = left + ((right - left) >> 1);
            if (canCover(points, mid)) {
                right = mid;
            } else {
                left = mid + 1;
```

```
            }
        }
        return left;
    }

    /**
     * 辅助方法：判断是否可以用 k 条直线覆盖所有点
     */
    private static boolean canCover(int[][] points, int k) {
        boolean[] covered = new boolean[points.length];
        return coverRecursive(points, covered, 0, k);
    }

    /**
     * 递归辅助方法：尝试覆盖剩余的点
     */
    private static boolean coverRecursive(int[][] points, boolean[] covered, int coveredCount,
int remainingLines) {
        int n = points.length;
        if (coveredCount == n) return true;
        if (remainingLines == 0) return false;

        // 找到第一个未覆盖的点
        int first = -1;
        for (int i = 0; i < n; i++) {
            if (!covered[i]) {
                first = i;
                break;
            }
        }

        // 尝试通过第一个未覆盖的点画一条直线
        for (int i = 0; i < n; i++) {
            if (i == first || covered[i]) continue;

            // 标记所有在这条直线上的点
            boolean[] newCovered = covered.clone();
            int newCount = coveredCount;

            for (int j = 0; j < n; j++) {
                if (!newCovered[j] && isCollinear(points[first], points[i], points[j])) {
                    newCovered[j] = true;
                    newCount++;
```

```
            }
        }

        if (coverRecursive(points, newCovered, newCount, remainingLines - 1)) {
            return true;
        }
    }

    // 如果没有其他点，可以单独用一条直线覆盖第一个未覆盖的点
    return coverRecursive(points, covered, coveredCount + 1, remainingLines - 1);
}

/**
 * 辅助方法：判断三个点是否共线
 */
private static boolean isCollinear(int[] p1, int[] p2, int[] p3) {
    // 使用叉积判断三点共线
    // (y2 - y1) * (x3 - x1) == (y3 - y1) * (x2 - x1)
    return (p2[1] - p1[1]) * (p3[0] - p1[0]) == (p3[1] - p1[1]) * (p2[0] - p1[0]);
}

/**
 * LeetCode 410. 分割数组的最大值
 * 题目来源: https://leetcode.com/problems/split-array-largest-sum/
 *
 * 题目描述:
 * 给定一个非负整数数组 nums 和一个整数 m，需要将这个数组分成 m 个非空的连续子数组。
 * 设计一个算法使得这 m 个子数组各自和的最大值最小。
 *
 * 思路分析:
 * 1. 二分答案法：答案范围在[max(nums), sum(nums)]之间
 * 2. 对于每个可能的答案 mid，检查能否将数组分成<=m 个子数组且每个子数组和<=mid
 * 3. 如果可以，说明答案可能更小，向左搜索；否则向右搜索
 *
 * 时间复杂度: O(n * log(sum - max))
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
 * @param nums 非负整数数组
 * @param m 分割的子数组数量
 * @return 最小的最大子数组和
 */
public static int splitArray(int[] nums, int m) {
```

```
    // 边界检查
    if (nums == null || nums.length == 0 || m <= 0) {
        return 0;
    }

    long left = 0;  // 最小可能值：数组中的最大值
    long right = 0; // 最大可能值：数组元素总和

    for (int num : nums) {
        left = Math.max(left, num);
        right += num;
    }

    while (left < right) {
        long mid = left + ((right - left) >> 1);

        // 检查是否能在限制下分成 m 个或更少的子数组
        if (canSplit(nums, m, mid)) {
            right = mid;  // 可以分割，尝试更小的最大值
        } else {
            left = mid + 1;  // 不能分割，需要更大的最大值
        }
    }

    return (int) left;
}

/**
 * 辅助方法：检查是否能在 maxSum 限制下将数组分成 count 个或更少的子数组
 */
private static boolean canSplit(int[] nums, int count, long maxSum) {
    int splits = 1;  // 至少有一个子数组
    long currentSum = 0;

    for (int num : nums) {
        if (currentSum + num > maxSum) {
            splits++;  // 需要新的子数组
            currentSum = num;

            if (splits > count) {
                return false;  // 超过了允许的子数组数量
            }
        } else {
```

```
            currentSum += num;
        }
    }

    return true;
}

/**
 * LeetCode 4. 寻找两个正序数组的中位数
 * 题目来源: https://leetcode.com/problems/median-of-two-sorted-arrays/
 *
 * 题目描述:
 * 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。
 *
 * 思路分析:
 * 1. 使用二分查找在较短的数组上进行分割
 * 2. 确保左半部分的最大值 <= 右半部分的最小值
 * 3. 中位数由左半部分最大值和右半部分最小值决定
 *
 * 时间复杂度: O(log(min(m,n)))
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
 * @param nums1 第一个正序数组
 * @param nums2 第二个正序数组
 * @return 两个数组的中位数
 */
public static double findMedianSortedArrays(int[] nums1, int[] nums2) {
    // 确保 nums1 是较短的数组
    if (nums1.length > nums2.length) {
        int[] temp = nums1;
        nums1 = nums2;
        nums2 = temp;
    }

    int m = nums1.length;
    int n = nums2.length;
    int left = 0, right = m;

    while (left <= right) {
        int partition1 = left + ((right - left) >> 1);
        int partition2 = ((m + n + 1) >> 1) - partition1;
```

```
            int maxLeft1 = (partition1 == 0) ? Integer.MIN_VALUE : nums1[partition1 - 1];
            int minRight1 = (partition1 == m) ? Integer.MAX_VALUE : nums1[partition1];

            int maxLeft2 = (partition2 == 0) ? Integer.MIN_VALUE : nums2[partition2 - 1];
            int minRight2 = (partition2 == n) ? Integer.MAX_VALUE : nums2[partition2];

            if (maxLeft1 <= minRight2 && maxLeft2 <= minRight1) {
                // 找到正确的分割点
                if ((m + n) % 2 == 0) {
                    return (Math.max(maxLeft1, maxLeft2) + Math.min(minRight1, minRight2)) / 2.0;
                } else {
                    return Math.max(maxLeft1, maxLeft2);
                }
            } else if (maxLeft1 > minRight2) {
                right = partition1 - 1;
            } else {
                left = partition1 + 1;
            }
        }

        throw new IllegalArgumentException("输入数组不合法");
    }

    /**
     * 牛客/剑指 Offer: 数字在排序数组中出现的次数
     * 题目来源: https://www.nowcoder.com/practice/70610bf967994b22bb1c26f9ae901fa2
     *
     * 题目描述:
     * 统计一个数字在排序数组中出现的次数。
     *
     * 思路分析:
     * 使用二分查找找到目标值的左右边界，次数 = 右边界 - 左边界 + 1
     *
     * 时间复杂度: O(log n)
     * 空间复杂度: O(1)
     * 是否最优解: 是
     *
     * @param nums 排序数组
     * @param target 目标值
     * @return 出现次数
     */
    public static int getNumberOfK(int[] nums, int target) {
```

```
    if (nums == null || nums.length == 0) {
        return 0;
    }

    int first = findLeft(nums, target);
    if (first == -1 || nums[first] != target) {
        return 0;
    }

    int last = findRight(nums, target);
    return last - first + 1;
}

/**
 * LeetCode 540. 有序数组中的单一元素
 * 题目来源: https://leetcode.com/problems/single-element-in-a-sorted-array/
 *
 * 题目描述:
 * 给定一个只包含整数的有序数组，每个元素都会出现两次，唯有一个数只会出现一次，找出这个数。
 *
 * 思路分析:
 * 1. 单一元素之前，成对元素的第一个位置是偶数索引
 * 2. 单一元素之后，成对元素的第一个位置是奇数索引
 * 3. 使用二分查找定位单一元素
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
 * @param nums 有序数组
 * @return 单一元素
 */
public static int singleNonDuplicate(int[] nums) {
    int left = 0, right = nums.length - 1;

    while (left < right) {
        int mid = left + ((right - left) >> 1);

        // 确保 mid 是偶数索引
        if (mid % 2 == 1) {
            mid--;
        }
```

```
        // 检查成对关系
        if (nums[mid] == nums[mid + 1]) {
            // 单一元素在右侧
            left = mid + 2;
        } else {
            // 单一元素在左侧或就是 mid
            right = mid;
        }
    }

    return nums[left];
}

/**
 * LeetCode 1482. 制作 m 束花所需的最少天数
 * 题目来源: https://leetcode.com/problems/minimum-number-of-days-to-make-m-bouquets/
 *
 * 题目描述:
 * 给你一个整数数组 bloomDay，以及两个整数 m 和 k。
 * 现需要制作 m 束花。制作花束时，需要使用花园中相邻的 k 朵花。
 * 花园中有 n 朵花，第 i 朵花会在 bloomDay[i] 时盛开。
 * 请你返回从花园中摘 m 束花需要等待的最少的天数。如果不能摘到 m 束花则返回 -1。
 *
 * 思路分析:
 * 1. 二分答案：天数范围在[min(bloomDay), max(bloomDay)]之间
 * 2. 对于每个天数，检查能否制作 m 束花
 * 3. 贪心验证：从左到右扫描，统计连续 k 朵已开花的数量
 *
 * 时间复杂度: O(n * log(max - min))
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
 * @param bloomDay 每朵花开放的天数
 * @param m 需要的花束数量
 * @param k 每束花包含的花朵数量
 * @return 最少等待天数
 */
public static int minDays(int[] bloomDay, int m, int k) {
    // 边界检查：如果花的总数不足以制作 m 束花，返回-1
    if ((long) m * k > bloomDay.length) {
        return -1;
    }
```

```
    int left = Integer.MAX_VALUE;
    int right = Integer.MIN_VALUE;

    // 找到天数范围
    for (int day : bloomDay) {
        left = Math.min(left, day);
        right = Math.max(right, day);
    }

    while (left < right) {
        int mid = left + ((right - left) >> 1);

        if (canMakeBouquets(bloomDay, m, k, mid)) {
            right = mid;  // 可以制作，尝试更少的天数
        } else {
            left = mid + 1;  // 不能制作，需要更多的天数
        }
    }

    return left;
}

/**
 * 辅助方法：检查在给定天数内能否制作 m 束花
 */
private static boolean canMakeBouquets(int[] bloomDay, int m, int k, int day) {
    int bouquets = 0;  // 已制作的花束数
    int flowers = 0;   // 当前连续开放的花朵数

    for (int bloom : bloomDay) {
        if (bloom <= day) {
            flowers++;
            if (flowers == k) {
                bouquets++;
                flowers = 0;
                if (bouquets == m) {
                    return true;
                }
            }
        } else {
            flowers = 0;  // 遇到未开放的花，重置计数
        }
    }
```

```
    return bouquets >= m;
}

/**
 * 牛客网: 旋转数组的最小数字
 * 题目来源: https://www.nowcoder.com/practice/9f3231a991af4f55b95579b44b7a01ba
 *
 * 题目描述:
 * 把一个数组最开始的若干个元素搬到数组的末尾,我们称之为数组的旋转。
 * 输入一个非递减排序的数组的一个旋转,输出旋转数组的最小元素。
 *
 * 思路分析:
 * 1. 二分查找法,比较中间元素与右边界元素
 * 2. 如果中间元素小于右边界,最小值在左侧
 * 3. 如果中间元素大于右边界,最小值在右侧
 * 4. 如果相等,右边界左移一位
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
 * @param nums 旋转数组
 * @return 最小元素
 */
public static int minNumberInRotateArray(int[] nums) {
    if (nums == null || nums.length == 0) return 0;
    int left = 0, right = nums.length - 1;
    while (left < right) {
        int mid = left + ((right - left) >> 1);
        if (nums[mid] < nums[right]) right = mid;
        else if (nums[mid] > nums[right]) left = mid + 1;
        else right--;
    }
    return nums[left];
}

/**
 * acwing: 数的三次方根
 * 题目来源: https://www.acwing.com/problem/content/792/
 *
 * 题目描述:
 * 给定一个浮点数 n,求它的三次方根,结果保留 6 位小数。
```

```
     *
     * 思路分析:
     * 1. 二分查找法,在[-10000, 10000]范围内搜索
     * 2. 精度控制到 1e-8
     * 3. 根据 mid^3 与 n 的大小关系调整搜索区间
     *
     * 时间复杂度: O(log(范围/精度))
     * 空间复杂度: O(1)
     * 是否最优解: 是
     *
     * @param n 输入浮点数
     * @return 三次方根
     */
    public static double cubeRoot(double n) {
        double left = -10000, right = 10000;
        while (right - left > 1e-8) {
            double mid = (left + right) / 2;
            if (mid * mid * mid >= n) right = mid;
            else left = mid;
        }
        return left;
    }

    /**
     * 杭电 OJ: 查找最接近的元素
     * 题目来源: http://acm.hdu.edu.cn/showproblem.php?pid=2141
     *
     * 题目描述:
     * 给定三个数组 A、B、C，以及多个查询 X。对于每个查询，判断是否存在 a∈A, b∈B, c∈C，使得 a+b+c=X。
     *
     * 思路分析:
     * 1. 将 A+B 的所有可能和存储在一个数组中
     * 2. 对这个和数组进行排序
     * 3. 对于每个查询 X，使用二分查找在 A+B 的和数组中查找是否存在 X-c（c∈C）
     *
     * 时间复杂度: O(L*M + N*log(L*M))
     * 空间复杂度: O(L*M)
     * 是否最优解: 是
     *
     * @param A 数组 A
     * @param B 数组 B
     * @param C 数组 C
```

```
     * @param X 查询值
     * @return 是否存在满足条件的组合
     */
    public static boolean findClosestSum(int[] A, int[] B, int[] C, int X) {
        int n = A.length;
        int[] sumAB = new int[n * n];
        int idx = 0;
        for (int a : A) for (int b : B) sumAB[idx++] = a + b;
        Arrays.sort(sumAB);

        for (int c : C) {
            int target = X - c;
            int left = 0, right = sumAB.length - 1;
            while (left <= right) {
                int mid = left + ((right - left) >> 1);
                if (sumAB[mid] == target) return true;
                if (sumAB[mid] < target) left = mid + 1;
                else right = mid - 1;
            }
        }
        return false;
    }

    /**
     * POJ: 月度开销
     * 题目来源: http://poj.org/problem?id=3273
     *
     * 题目描述:
     * FJ 需要将连续的 N 天划分为 M 个时间段，每个时间段的花费是该时间段内每天花费的总和。
     * 他希望最小化所有时间段中花费的最大值。
     *
     * 思路分析:
     * 1. 二分答案法：答案范围在[max(expenses), sum(expenses)]之间
     * 2. 对于每个可能的答案 mid，检查能否将数组分成<=M 个子数组且每个子数组和<=mid
     * 3. 如果可以，说明答案可能更小，向左搜索；否则向右搜索
     *
     * 时间复杂度: O(n log(sum))
     * 空间复杂度: O(1)
     * 是否最优解: 是
     *
     * @param expenses 每天的开销数组
     * @param M 划分的时间段数
     * @return 最小的最大开销
```

```
 */
public static int monthlyExpense(int[] expenses, int M) {
    int left = 0, right = 0;
    for (int expense : expenses) {
        left = Math.max(left, expense);
        right += expense;
    }
    while (left < right) {
        int mid = left + ((right - left) >> 1);
        if (canSplitExpenses(expenses, M, mid)) right = mid;
        else left = mid + 1;
    }
    return left;
}

private static boolean canSplitExpenses(int[] expenses, int M, int maxSum) {
    int count = 1, currentSum = 0;
    for (int expense : expenses) {
        if (currentSum + expense > maxSum) {
            count++;
            currentSum = expense;
            if (count > M) return false;
        } else currentSum += expense;
    }
    return true;
}

/**
 * 洛谷: 砍树
 * 题目来源: https://www.luogu.com.cn/problem/P1873
 *
 * 题目描述:
 * 伐木工人需要砍倒M米高的树木，每棵树的高度不同。
 * 伐木工人使用一个设定高度H的锯子，高于H的树木会被砍下高出的部分。
 * 请帮助伐木工人确定锯子的高度H，使得被砍下的树木总长度刚好等于M。
 *
 * 思路分析:
 * 1. 二分答案法：高度范围在[0, max(treeHeights)]之间
 * 2. 对于每个候选高度mid，计算能砍下的总长度
 * 3. 如果总长度>=M，可以尝试更高的高度；否则需要降低高度
 *
 * 时间复杂度: O(n log(maxHeight))
 * 空间复杂度: O(1)
```

```
     * 是否最优解: 是
     *
     * @param trees 树的高度数组
     * @param M 需要砍下的总长度
     * @return 锯子的设定高度
     */
    public static int cutTrees(int[] trees, int M) {
        int left = 0, right = 0;
        for (int tree : trees) right = Math.max(right, tree);
        int result = 0;
        while (left <= right) {
            int mid = left + ((right - left) >> 1);
            long total = 0;
            for (int tree : trees) if (tree > mid) total += tree - mid;
            if (total >= M) {
                result = mid;
                left = mid + 1;
            } else right = mid - 1;
        }
        return result;
    }

    /**
     * USACO: 愤怒的奶牛
     * 题目来源: http://www.usaco.org/index.php?page=viewproblem2&cpid=592
     *
     * 题目描述:
     * 在一个一维的场地中放置了若干堆干草，奶牛被点燃后会向左右两个方向传播爆炸。
     * 爆炸的传播速度是每秒 1 单位距离。求最小的爆炸半径 R，使得点燃任意一个干草堆都能引爆所有干草堆。
     *
     * 思路分析:
     * 1. 二分答案法：半径范围在[0, maxPosition - minPosition]之间
     * 2. 对于每个候选半径 mid，检查是否能通过点燃一个点引爆所有点
     * 3. 贪心验证：从最左边开始，每次尽可能远地放置引爆点
     *
     * 时间复杂度: O(n log(maxDistance))
     * 空间复杂度: O(1)
     * 是否最优解: 是
     *
     * @param haybales 干草堆的位置数组
     * @return 最小爆炸半径
     */
```

```
public static int angryCows(int[] haybales) {
    Arrays.sort(haybales);
    int left = 0, right = haybales[haybales.length - 1] - haybales[0];
    while (left < right) {
        int mid = left + ((right - left) >> 1);
        if (canExplodeAll(haybales, mid)) right = mid;
        else left = mid + 1;
    }
    return left;
}

private static boolean canExplodeAll(int[] haybales, int radius) {
    int n = haybales.length;
    int count = 1;
    int lastPos = haybales[0];
    for (int i = 1; i < n; i++) {
        if (haybales[i] - lastPos > 2 * radius) {
            count++;
            lastPos = haybales[i];
        }
    }
    return count <= 1; // 只需要一个引爆点就能引爆所有干草堆
}

/**
 * HackerRank: 最小化最大值
 * 题目来源: https://www.hackerrank.com/challenges/min-max/problem
 *
 * 题目描述:
 * 将一个数组分割成 K 个非空的连续子数组，设计一个算法使得这 K 个子数组各自和的最大值最小。
 *
 * 思路分析:
 * 1. 二分答案法：答案范围在[max(nums), sum(nums)]之间
 * 2. 对于每个可能的答案 mid，检查能否将数组分成<=K 个子数组且每个子数组和<=mid
 * 3. 如果可以，说明答案可能更小，向左搜索；否则向右搜索
 *
 * 时间复杂度: O(n log(sum))
 * 空间复杂度: O(1)
 * 是否最优解: 是
 *
 * @param nums 数组
 * @param K 分割的子数组数量
 * @return 最小的最大子数组和
```

```
     */
    public static int minimizeMaximum(int[] nums, int K) {
        int left = 0, right = 0;
        for (int num : nums) {
            left = Math.max(left, num);
            right += num;
        }
        while (left < right) {
            int mid = left + ((right - left) >> 1);
            if (canSplitIntoK(nums, K, mid)) right = mid;
            else left = mid + 1;
        }
        return left;
    }

    private static boolean canSplitIntoK(int[] nums, int K, int maxSum) {
        int count = 1, currentSum = 0;
        for (int num : nums) {
            if (currentSum + num > maxSum) {
                count++;
                currentSum = num;
                if (count > K) return false;
            } else currentSum += num;
        }
        return true;
    }

    /**
     * 计蒜客: 跳石头
     * 题目来源: https://nanti.jisuanke.com/t/T1201
     *
     * 题目描述:
     * 河道中分布着一些巨大岩石，组委会计划移走一些岩石，使得选手们在比赛过程中的最短跳跃距离尽可能长。
     *
     * 思路分析:
     * 1. 二分答案法：距离范围在[0, L]之间
     * 2. 对于每个候选距离 mid，计算需要移走多少块岩石
     * 3. 如果需要移走的岩石数不超过 M，则可以尝试更大的距离
     *
     * 时间复杂度: O(n log L)
     * 空间复杂度: O(1)
     * 是否最优解: 是
```

```
     *
     * @param rocks 岩石位置数组
     * @param M 最多可移走的岩石数
     * @param L 河道长度
     * @return 最大的最短跳跃距离
     */
    public static int jumpStones(int[] rocks, int M, int L) {
        Arrays.sort(rocks);
        int left = 0, right = L, result = 0;
        while (left <= right) {
            int mid = left + ((right - left) >> 1);
            if (canJump(rocks, M, mid, L)) {
                result = mid;
                left = mid + 1;
            } else right = mid - 1;
        }
        return result;
    }

    private static boolean canJump(int[] rocks, int M, int minDist, int L) {
        int count = 0, lastPos = 0;
        for (int rock : rocks) {
            if (rock - lastPos < minDist) count++;
            else lastPos = rock;
        }
        if (L - lastPos < minDist) count++;
        return count <= M;
    }

    /**
     * Codeforces 474B - Worms
     * 题目来源: https://codeforces.com/problemset/problem/474/B
     *
     * 题目描述:
     * 有 n 堆虫子，第 i 堆有 a_i 条虫子。每条虫子都有一个编号，从 1 开始连续编号。
     * 给出 m 个查询，每个查询给出一个编号 x，问编号为 x 的虫子属于哪一堆。
     *
     * 思路分析:
     * 1. 使用前缀和数组记录每堆虫子的结束位置
     * 2. 对于每个查询，使用二分查找在前缀和数组中查找 x 所在的位置
     * 3. 找到第一个前缀和大于等于 x 的位置
     *
     * 时间复杂度: O(n + m log n)
```

```
 * 空间复杂度: O(n)
 * 是否最优解: 是
 *
 * @param piles 每堆虫子的数量数组
 * @param queries 查询数组
 * @return 每个查询对应的堆编号
 */
public static int[] findWormPiles(int[] piles, int[] queries) {
    int n = piles.length;
    int m = queries.length;

    // 计算前缀和
    long[] prefixSum = new long[n];
    prefixSum[0] = piles[0];
    for (int i = 1; i < n; i++) {
        prefixSum[i] = prefixSum[i - 1] + piles[i];
    }

    int[] result = new int[m];

    for (int i = 0; i < m; i++) {
        long x = queries[i];

        // 使用二分查找找到第一个前缀和 >= x 的位置
        int left = 0, right = n - 1;
        int pileIndex = -1;

        while (left <= right) {
            int mid = left + ((right - left) >> 1);

            if (prefixSum[mid] >= x) {
                pileIndex = mid;
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }

        result[i] = pileIndex + 1;  // 1-based索引
    }

    return result;
}
```

```
    /**
     * SPOJ - AGGRCOW (Aggressive cows)
     * 题目来源: https://www.spoj.com/problems/AGGRCOW/
     *
     * 题目描述:
     * 农夫约翰建造了一个有 C 个摊位的畜棚，摊位位于 x0, ..., x(C-1)。
     * 他的 M 头奶牛总是相互攻击，约翰必须以某种方式分配奶牛到摊位，使它们之间的最小距离尽可能大。
     *
     * 思路分析:
     * 1. 二分答案法：距离范围在[0, maxPosition - minPosition]之间
     * 2. 对于每个候选距离 mid，使用贪心算法验证是否可以放置所有奶牛
     * 3. 如果可以，说明可以尝试更大的距离；否则需要减小距离
     *
     * 时间复杂度: O(n log(max-min) * n)
     * 空间复杂度: O(1)
     * 是否最优解: 是
     */

    // 测试方法
    public static void main(String[] args) {
        // 直接运行所有测试用例

        // 测试基础二分查找
        int[] nums1 = {-1, 0, 3, 5, 9, 12};
        System.out.println("基础二分查找测试:");
        System.out.println("查找 9: " + search(nums1, 9)); // 应该输出 4
        System.out.println("查找 2: " + search(nums1, 2)); // 应该输出-1

        // 测试查找插入位置
        int[] nums2 = {1, 3, 5, 6};
        System.out.println("\n 查找插入位置测试:");
        System.out.println("查找 5: " + searchInsert(nums2, 5)); // 应该输出 2
        System.out.println("查找 2: " + searchInsert(nums2, 2)); // 应该输出 1
        System.out.println("查找 7: " + searchInsert(nums2, 7)); // 应该输出 4
        System.out.println("查找 0: " + searchInsert(nums2, 0)); // 应该输出 0

        // 测试查找范围
        int[] nums3 = {5, 7, 7, 8, 8, 10};
        System.out.println("\n 查找范围测试:");
        int[] range = searchRange(nums3, 8);
        System.out.println("查找 8: [" + range[0] + ", " + range[1] + "]"); // 应该输出[3, 4]
```

```
        range = searchRange(nums3, 6);
        System.out.println("查找 6: [" + range[0] + ", " + range[1] + "]"); // 应该输出[-1, -1]

        // 测试完全平方数
        System.out.println("\n 完全平方数测试:");
        System.out.println("16 是完全平方数: " + isPerfectSquare(16)); // 应该输出 true
        System.out.println("14 是完全平方数: " + isPerfectSquare(14)); // 应该输出 false

        // 测试寻找峰值
        int[] nums4 = {1, 2, 3, 1};
        System.out.println("\n 寻找峰值测试:");
        System.out.println("峰值索引: " + findPeakElement(nums4)); // 应该输出 2

        // 测试寻找旋转数组最小值
        int[] nums5 = {3, 4, 5, 1, 2};
        System.out.println("\n 寻找旋转数组最小值测试:");
        System.out.println("最小值: " + findMin(nums5)); // 应该输出 1

        // 测试搜索二维矩阵
        int[][] matrix = {{1, 3, 5, 7}, {10, 11, 16, 20}, {23, 30, 34, 60}};
        System.out.println("\n 搜索二维矩阵测试:");
        System.out.println("查找 3: " + searchMatrix(matrix, 3)); // 应该输出 true
        System.out.println("查找 13: " + searchMatrix(matrix, 13)); // 应该输出 false

        // 测试搜索旋转排序数组
        int[] nums6 = {4, 5, 6, 7, 0, 1, 2};
        System.out.println("\n 搜索旋转排序数组测试:");
        System.out.println("查找 0: " + searchInRotatedSortedArray(nums6, 0)); // 应该输出 4
        System.out.println("查找 3: " + searchInRotatedSortedArray(nums6, 3)); // 应该输出-1

        // 测试寻找重复数
        int[] nums7 = {1, 3, 4, 2, 2};
        System.out.println("\n 寻找重复数测试:");
        System.out.println("重复数: " + findDuplicate(nums7)); // 应该输出 2

        // 测试 Codeforces 题目
        int[] prices = {1, 2, 3, 4, 5};
        int[] queries = {3, 10, 1};
        int[] drinks = howManyDrinks(prices, queries);
        System.out.println("\nCodeforces 题目测试:");
        System.out.println("可以购买的饮料种类数: " + Arrays.toString(drinks)); // 应该输出[3, 5,
1]
```

```
// 测试SPOJ题目
int[] positions = {1, 2, 8, 4, 9};
System.out.println("\nSPOJ题目测试:");
System.out.println("最大最小距离: " + aggressiveCows(positions, 3)); // 应该输出3

// 测试LeetCode 69. Sqrt(x)
System.out.println("\nLeetCode 69. Sqrt(x)测试:");
System.out.println("Sqrt(4): " + mySqrt(4)); // 应该输出 2
System.out.println("Sqrt(8): " + mySqrt(8)); // 应该输出 2
System.out.println("Sqrt(0): " + mySqrt(0)); // 应该输出 0
System.out.println("Sqrt(1): " + mySqrt(1)); // 应该输出 1
System.out.println("Sqrt(2147395599): " + mySqrt(2147395599)); // 大数值测试

// 测试LeetCode 374. 猜数字大小
System.out.println("\nLeetCode 374. 猜数字大小测试:");
System.out.println("猜数字(10): " + guessNumber(10)); // 应该输出 6
System.out.println("猜数字(1): " + guessNumber(1)); // 应该输出 1

// 测试LeetCode 875. 爱吃香蕉的珂珂
System.out.println("\nLeetCode 875. 爱吃香蕉的珂珂测试:");
int[] piles = {3, 6, 7, 11};
System.out.println("最小吃香蕉速度(8小时): " + minEatingSpeed(piles, 8)); // 应该输出 4
int[] piles2 = {30, 11, 23, 4, 20};
System.out.println("最小吃香蕉速度(5小时): " + minEatingSpeed(piles2, 5)); // 应该输出 30
System.out.println("最小吃香蕉速度(6小时): " + minEatingSpeed(piles2, 6)); // 应该输出 23

// 测试LeetCode 1011. 在D天内送达包裹的能力
System.out.println("\nLeetCode 1011. 在D天内送达包裹的能力测试:");
int[] weights = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
System.out.println("最小运载能力(5天): " + shipWithinDays(weights, 5)); // 应该输出 15
int[] weights2 = {3, 2, 2, 4, 1, 4};
System.out.println("最小运载能力(3天): " + shipWithinDays(weights2, 3)); // 应该输出 6
int[] weights3 = {1, 2, 3, 1, 1};
System.out.println("最小运载能力(4天): " + shipWithinDays(weights3, 4)); // 应该输出 3

// 测试AtCoder ABC023D - 射撃王
System.out.println("\nAtCoder ABC023D - 射撃王测试:");
int[][] points1 = {{0, 0}, {1, 1}, {2, 2}, {3, 3}}; // 所有点共线，只需要1颗子弹
System.out.println("最少子弹数: " + minBullets(points1)); // 应该输出 1
int[][] points2 = {{0, 0}, {1, 0}, {0, 1}, {1, 1}}; // 四个点形成正方形，需要2颗子弹
System.out.println("最少子弹数: " + minBullets(points2)); // 应该输出 2
```

```
        // 测试 LeetCode 410: 分割数组的最大值
        System.out.println("\nLeetCode 410: 分割数组的最大值测试:");
        int[] splitNums1 = {7, 2, 5, 10, 8};
        System.out.println("最小的最大子数组和(m=2): " + splitArray(splitNums1, 2)); // 应该输出
18
        int[] splitNums2 = {1, 2, 3, 4, 5};
        System.out.println("最小的最大子数组和(m=2): " + splitArray(splitNums2, 2)); // 应该输出
9

        // 测试 LeetCode 4: 寻找两个正序数组的中位数
        System.out.println("\nLeetCode 4: 寻找两个正序数组的中位数测试:");
        int[] medianNums1 = {1, 3};
        int[] medianNums2 = {2};
        System.out.println("中位数: " + findMedianSortedArrays(medianNums1, medianNums2)); // 应
该输出 2.0
        int[] medianNums3 = {1, 2};
        int[] medianNums4 = {3, 4};
        System.out.println("中位数: " + findMedianSortedArrays(medianNums3, medianNums4)); // 应
该输出 2.5

        // 测试牛客/剑指 Offer: 数字在排序数组中出现的次数
        System.out.println("\n 牛客/剑指 Offer: 数字在排序数组中出现的次数测试:");
        int[] countNums = {1, 2, 3, 3, 3, 3, 4, 5};
        System.out.println("3 出现的次数: " + getNumberOfK(countNums, 3)); // 应该输出 4
        System.out.println("6 出现的次数: " + getNumberOfK(countNums, 6)); // 应该输出 0

        // 测试 LeetCode 540: 有序数组中的单一元素
        System.out.println("\nLeetCode 540: 有序数组中的单一元素测试:");
        int[] singleNums1 = {1, 1, 2, 3, 3, 4, 4, 8, 8};
        System.out.println("单一元素: " + singleNonDuplicate(singleNums1)); // 应该输出 2
        int[] singleNums2 = {3, 3, 7, 7, 10, 11, 11};
        System.out.println("单一元素: " + singleNonDuplicate(singleNums2)); // 应该输出 10

        // 测试 LeetCode 1482: 制作 m 束花所需的最少天数
        System.out.println("\nLeetCode 1482: 制作 m 束花所需的最少天数测试:");
        int[] bloomDays1 = {1, 10, 3, 10, 2};
        System.out.println("最少天数(m=3, k=1): " + minDays(bloomDays1, 3, 1)); // 应该输出 3
        int[] bloomDays2 = {1, 10, 3, 10, 2};
        System.out.println("最少天数(m=3, k=2): " + minDays(bloomDays2, 3, 2)); // 应该输出 -1
        int[] bloomDays3 = {7, 7, 7, 7, 12, 7, 7};
        System.out.println("最少天数(m=2, k=3): " + minDays(bloomDays3, 2, 3)); // 应该输出 12

        // 测试新增题目
```

```
        System.out.println("\n=== 新增各大平台题目测试 ===");

        // 测试牛客网题目
        int[] rotateArray = {3, 4, 5, 1, 2};
        System.out.println("牛客网-旋转数组最小数字: " + minNumberInRotateArray(rotateArray)); // 应该输出 1

        // 测试 acwing 题目
        System.out.println("acwing-三次方根(27): " + cubeRoot(27.0)); // 应该输出 3.0

        // 测试杭电 OJ 题目
        int[] A = {1, 2, 3}, B = {4, 5, 6}, C = {7, 8, 9};
        System.out.println("杭电 OJ-最接近和(15): " + findClosestSum(A, B, C, 15)); // 应该输出 true

        // 测试 POJ 题目
        int[] expenses = {100, 200, 300, 400, 500};
        System.out.println("POJ-月度开销(3 段): " + monthlyExpense(expenses, 3)); // 应该输出 500

        // 测试洛谷题目
        int[] trees = {20, 15, 10, 17};
        System.out.println("洛谷-砍树(7 米): " + cutTrees(trees, 7)); // 应该输出 15

        // 测试 USACO 题目
        int[] haybales = {1, 2, 4, 8, 9};
        System.out.println("USACO-愤怒的奶牛: " + angryCows(haybales)); // 应该输出 4

        // 测试 HackerRank 题目
        int[] nums = {1, 2, 3, 4, 5};
        System.out.println("HackerRank-最小化最大值(3 段): " + minimizeMaximum(nums, 3)); // 应该输出 6

        // 测试计蒜客题目
        int[] rocks = {2, 11, 14, 17, 21};
        System.out.println("计蒜客-跳石头(移 2 块): " + jumpStones(rocks, 2, 25)); // 应该输出 4

        // 测试 Codeforces 题目
        int[] wormPiles = {1, 3, 2, 4};
        int[] wormQueries = {1, 4, 7, 10};
        int[] wormResults = findWormPiles(wormPiles, wormQueries);
        System.out.println("Codeforces-虫子堆: " + Arrays.toString(wormResults)); // 应该输出[1, 2, 3, 4]
    }
```

```
}

==========================================

文件: BinarySearchProblems.py
==========================================
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
二分查找算法实现与相关题目解答 (Python 版本)

本文件包含了二分查找的各种实现和在不同场景下的应用
涵盖了 LeetCode、Codeforces、SPOJ 等平台的经典题目
"""



def search(nums, target):
    """
    基础二分查找
    在有序数组中查找目标值

    时间复杂度: O(log n)
    空间复杂度: O(1)

    :param nums: 有序数组
    :param target: 目标值
    :return: 目标值的索引，如果不存在返回-1
    """
    if not nums:
        return -1

    left, right = 0, len(nums) - 1
    while left <= right:
        # 使用位运算避免整数溢出
        mid = left + ((right - left) >> 1)
        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

```
def search_insert(nums, target):
    """
    查找插入位置
    在有序数组中查找目标值应该插入的位置

    时间复杂度: O(log n)
    空间复杂度: O(1)

    :param nums: 有序数组
    :param target: 目标值
    :return: 插入位置索引
    """
    if not nums:
        return 0

    left, right = 0, len(nums)
    while left < right:
        mid = left + ((right - left) >> 1)
        if nums[mid] < target:
            left = mid + 1
        else:
            right = mid
    return left


def search_range(nums, target):
    """
    查找元素的第一个和最后一个位置
    在有序数组中查找目标值的起始和结束位置

    时间复杂度: O(log n)
    空间复杂度: O(1)

    :param nums: 有序数组
    :param target: 目标值
    :return: [起始位置, 结束位置]，如果不存在返回[-1, -1]
    """
    if not nums:
        return [-1, -1]

    first = find_left(nums, target)
```

```
    # 如果找不到>=target 的元素，或者该元素不等于 target，则说明 target 不存在
    if first == len(nums) or nums[first] != target:
        return [-1, -1]

    last = find_right(nums, target)
    return [first, last]


def find_left(nums, target):
    """
    辅助方法：查找>=target 的最左位置
    """
    left, right = 0, len(nums)
    while left < right:
        mid = left + ((right - left) >> 1)
        if nums[mid] < target:
            left = mid + 1
        else:
            right = mid
    return left


def find_right(nums, target):
    """
    辅助方法：查找<=target 的最右位置
    """
    left, right = 0, len(nums)
    while left < right:
        mid = left + ((right - left) >> 1)
        if nums[mid] <= target:
            left = mid + 1
        else:
            right = mid
    return left - 1


def is_perfect_square(num):
    """
    有效的完全平方数
    判断一个数是否是完全平方数

    时间复杂度: O(log n)
    空间复杂度: O(1)
```

```
    :param num: 正整数
    :return: 如果是完全平方数返回 True，否则返回 False
    """
    if num < 1:
        return False
    if num == 1:
        return True

    left, right = 1, num // 2  # 一个数的平方根不会超过它的一半(除了 1)
    while left <= right:
        mid = left + ((right - left) >> 1)
        square = mid * mid
        if square == num:
            return True
        elif square > num:
            right = mid - 1
        else:
            left = mid + 1
    return False


def find_peak_element(nums):
    """
    寻找峰值元素

    时间复杂度: O(log n)
    空间复杂度: O(1)

    :param nums: 数组
    :return: 峰值元素的索引
    """
    left, right = 0, len(nums) - 1
    while left < right:
        mid = left + ((right - left) >> 1)
        if nums[mid] > nums[mid + 1]:
            right = mid
        else:
            left = mid + 1
    return left


def find_min(nums):
```

```
    """
    寻找旋转排序数组中的最小值

    时间复杂度: O(log n)
    空间复杂度: O(1)

    :param nums: 旋转排序数组
    :return: 数组中的最小值
    """
    left, right = 0, len(nums) - 1
    while left < right:
        mid = left + ((right - left) >> 1)
        if nums[mid] < nums[right]:
            right = mid
        else:
            left = mid + 1
    return nums[left]


def search_matrix(matrix, target):
    """
    搜索二维矩阵

    时间复杂度: O(log(m*n))
    空间复杂度: O(1)

    :param matrix: 二维矩阵
    :param target: 目标值
    :return: 如果找到目标值返回 True，否则返回 False
    """
    if not matrix or not matrix[0]:
        return False

    m, n = len(matrix), len(matrix[0])
    left, right = 0, m * n - 1

    while left <= right:
        mid = left + ((right - left) >> 1)
        mid_value = matrix[mid // n][mid % n]

        if mid_value == target:
            return True
        elif mid_value < target:
```

```
            left = mid + 1
        else:
            right = mid - 1

    return False


def search_in_rotated_sorted_array(nums, target):
    """
    搜索旋转排序数组

    时间复杂度: O(log n)
    空间复杂度: O(1)

    :param nums: 旋转排序数组
    :param target: 目标值
    :return: 目标值的索引，如果不存在返回-1
    """
    left, right = 0, len(nums) - 1

    while left <= right:
        mid = left + ((right - left) >> 1)

        if nums[mid] == target:
            return mid

        # 左半部分有序
        if nums[left] <= nums[mid]:
            if nums[left] <= target < nums[mid]:
                right = mid - 1
            else:
                left = mid + 1
        # 右半部分有序
        else:
            if nums[mid] < target <= nums[right]:
                left = mid + 1
            else:
                right = mid - 1

    return -1


def find_duplicate(nums):
```

```
    """
    寻找重复数

    时间复杂度: O(n log n)
    空间复杂度: O(1)

    :param nums: 包含重复数字的数组
    :return: 重复的数字
    """
    left, right = 1, len(nums) - 1

    while left < right:
        mid = left + ((right - left) >> 1)
        count = 0

        # 计算小于等于 mid 的数字个数
        for num in nums:
            if num <= mid:
                count += 1

        # 根据抽屉原理判断重复数字在哪一侧
        if count > mid:
            right = mid
        else:
            left = mid + 1

    return left


def how_many_drinks(prices, queries):
    """
    Codeforces 706B. Interesting drink

    时间复杂度: O(n log n + q log n)
    空间复杂度: O(1)

    :param prices: 饮料价格数组
    :param queries: 查询金额数组
    :return: 每个查询可以购买的饮料种类数
    """
    prices.sort()
    result = []
```

```
    for money in queries:
        left, right = 0, len(prices)

        while left < right:
            mid = left + ((right - left) >> 1)
            if prices[mid] <= money:
                left = mid + 1
            else:
                right = mid

        result.append(left)

    return result



def my_sqrt(x):
    """
    LeetCode 69. Sqrt(x)

    时间复杂度: O(log x)
    空间复杂度: O(1)

    :param x: 输入的非负整数
    :return: x 的平方根的整数部分
    """
    if x == 0 or x == 1:
        return x

    left, right = 1, x
    while left <= right:
        mid = left + ((right - left) >> 1)

        # 防止整数溢出，使用除法而不是乘法
        if mid > x // mid:
            right = mid - 1
        else:
            # 如果下一个值会溢出或者大于 x/mid，则当前 mid 是最大的有效平方根
            if mid + 1 > x // (mid + 1):
                return mid
            left = mid + 1

    return right
```

```python
def guess_number(n):
    """
    LeetCode 374. 猜数字大小

    时间复杂度: O(log n)
    空间复杂度: O(1)

    :param n: 数字范围上限
    :return: 猜中的数字
    """
    left, right = 1, n

    while left <= right:
        # 防止整数溢出
        mid = left + ((right - left) >> 1)
        result = guess(mid)

        if result == 0:
            return mid  # 猜对了
        elif result == 1:
            left = mid + 1  # 猜小了，往右边找
        else:
            right = mid - 1  # 猜大了，往左边找

    return -1  # 理论上不会执行到这里


def guess(num):
    """
    模拟 API：判断版本是否错误
    """
    pick = 6  # 假设选中的数字是 6
    if num < pick:
        return 1
    elif num > pick:
        return -1
    else:
        return 0


def min_eating_speed(piles, h):
    """
```

```
    LeetCode 875. 爱吃香蕉的珂珂

    时间复杂度: O(n log maxPile)
    空间复杂度: O(1)

    :param piles: 香蕉堆数组
    :param h: 警卫离开的时间（小时）
    :return: 最小的吃香蕉速度
    """
    left = 1
    right = max(piles)  # 最大的香蕉堆，作为右边界

    while left < right:
        mid = left + ((right - left) >> 1)

        if can_finish(piles, h, mid):
            right = mid  # 可以吃完，尝试更小的速度
        else:
            left = mid + 1  # 不能吃完，需要更大的速度

    return left


def can_finish(piles, h, speed):
    """
    辅助方法：判断以速度 speed 是否能在 h 小时内吃完所有香蕉
    """
    time = 0

    for pile in piles:
        # 计算吃完当前堆需要的时间
        time += (pile + speed - 1) // speed  # 等价于 math.ceil(pile / speed)

        # 如果时间已经超过 h，可以提前返回 False
        if time > h:
            return False

    return time <= h


def ship_within_days(weights, days):
    """
    LeetCode 1011. 在 D 天内送达包裹的能力
```

```
    时间复杂度: O(n log totalWeight)
    空间复杂度: O(1)

    :param weights: 包裹重量数组
    :param days: 天数限制
    :return: 船的最小运载能力
    """
    left = max(weights)  # 最小运载能力：最大的单个包裹重量
    right = sum(weights)  # 最大运载能力：所有包裹的总重量

    while left < right:
        mid = left + ((right - left) >> 1)

        if can_ship_in_days(weights, days, mid):
            right = mid  # 可以在 days 天内运完，尝试更小的运载能力
        else:
            left = mid + 1  # 不能在 days 天内运完，需要更大的运载能力

    return left



def can_ship_in_days(weights, days, capacity):
    """
    辅助方法：判断以 capacity 的运载能力是否能在 days 天内运完所有包裹
    """
    current_weight = 0
    day_count = 1  # 至少需要 1 天

    for weight in weights:
        # 如果当前包裹的重量已经超过了运载能力，不可能运完
        if weight > capacity:
            return False

        # 如果当前累计重量加上当前包裹的重量超过了运载能力，需要新的一天
        if current_weight + weight > capacity:
            day_count += 1
            current_weight = weight  # 新的一天从当前包裹开始

            # 如果天数已经超过了限制，可以提前返回 False
            if day_count > days:
                return False
        else:
```

```
            current_weight += weight  # 继续往当前天添加包裹

    return day_count <= days


def min_bullets(points):
    """
    AtCoder ABC023D - 射撃王 (Shooting King)

    时间复杂度: O(N^3 log N)
    空间复杂度: O(N)

    :param points: 目标点数组，每个点是(x, y)
    :return: 最少需要的子弹数
    """
    n = len(points)
    if n == 0:
        return 0
    if n == 1:
        return 1

    left, right = 1, n
    while left < right:
        mid = left + ((right - left) >> 1)
        if can_cover(points, mid):
            right = mid
        else:
            left = mid + 1

    return left


def can_cover(points, k):
    """
    辅助方法：判断是否可以用k条直线覆盖所有点
    """
    covered = [False] * len(points)
    return cover_recursive(points, covered, 0, k)


def cover_recursive(points, covered, covered_count, remaining_lines):
    """
    递归辅助方法：尝试覆盖剩余的点
```

```
    """
    n = len(points)
    if covered_count == n:
        return True
    if remaining_lines == 0:
        return False

    # 找到第一个未覆盖的点
    first = -1
    for i in range(n):
        if not covered[i]:
            first = i
            break

    # 尝试通过第一个未覆盖的点画一条直线
    for i in range(n):
        if i == first or covered[i]:
            continue

        # 标记所有在这条直线上的点
        new_covered = covered.copy()
        new_count = covered_count

        for j in range(n):
            if not new_covered[j] and is_collinear(points[first], points[i], points[j]):
                new_covered[j] = True
                new_count += 1

        if cover_recursive(points, new_covered, new_count, remaining_lines - 1):
            return True

    # 如果没有其他点，可以单独用一条直线覆盖第一个未覆盖的点
    covered[first] = True
    result = cover_recursive(points, covered, covered_count + 1, remaining_lines - 1)
    covered[first] = False  # 回溯
    return result


def is_collinear(p1, p2, p3):
    """
    辅助方法：判断三个点是否共线
    """
    # 使用叉积判断三点共线
```

```
    # (y2 - y1) * (x3 - x1) == (y3 - y1) * (x2 - x1)
    return (p2[1] - p1[1]) * (p3[0] - p1[0]) == (p3[1] - p1[1]) * (p2[0] - p1[0])


def split_array(nums, m):
    """
    LeetCode 410. 分割数组的最大值

    时间复杂度: O(n * log(sum - max))
    空间复杂度: O(1)

    :param nums: 非负整数数组
    :param m: 分割的子数组数量
    :return: 最小的最大子数组和
    """
    # 边界检查
    if not nums or m <= 0:
        return 0

    left = 0  # 最小可能值：数组中的最大值
    right = 0  # 最大可能值：数组元素总和

    for num in nums:
        left = max(left, num)
        right += num

    while left < right:
        mid = left + ((right - left) >> 1)

        # 检查是否能在限制下分成 m 个或更少的子数组
        if can_split(nums, m, mid):
            right = mid  # 可以分割，尝试更小的最大值
        else:
            left = mid + 1  # 不能分割，需要更大的最大值

    return left


def can_split(nums, count, max_sum):
    """
    辅助方法：检查是否能在 max_sum 限制下将数组分成 count 个或更少的子数组
    """
    splits = 1  # 至少有一个子数组
```

```
    current_sum = 0

    for num in nums:
        if current_sum + num > max_sum:
            splits += 1  # 需要新的子数组
            current_sum = num

            if splits > count:
                return False  # 超过了允许的子数组数量
        else:
            current_sum += num

    return True


def find_median_sorted_arrays(nums1, nums2):
    """
    LeetCode 4. 寻找两个正序数组的中位数

    时间复杂度: O(log(min(m,n)))
    空间复杂度: O(1)

    :param nums1: 第一个正序数组
    :param nums2: 第二个正序数组
    :return: 两个数组的中位数
    """
    # 确保 nums1 是较短的数组
    if len(nums1) > len(nums2):
        nums1, nums2 = nums2, nums1

    m, n = len(nums1), len(nums2)
    left, right = 0, m

    while left <= right:
        partition1 = left + ((right - left) >> 1)
        partition2 = ((m + n + 1) >> 1) - partition1

        max_left1 = float('-inf') if partition1 == 0 else nums1[partition1 - 1]
        min_right1 = float('inf') if partition1 == m else nums1[partition1]

        max_left2 = float('-inf') if partition2 == 0 else nums2[partition2 - 1]
        min_right2 = float('inf') if partition2 == n else nums2[partition2]
```

```
        if max_left1 <= min_right2 and max_left2 <= min_right1:
            # 找到正确的分割点
            if (m + n) % 2 == 0:
                return (max(max_left1, max_left2) + min(min_right1, min_right2)) / 2.0
            else:
                return max(max_left1, max_left2)
        elif max_left1 > min_right2:
            right = partition1 - 1
        else:
            left = partition1 + 1

    raise ValueError("输入数组不合法")


def get_number_of_k(nums, target):
    """
    牛客/剑指 Offer: 数字在排序数组中出现的次数

    时间复杂度: O(log n)
    空间复杂度: O(1)

    :param nums: 排序数组
    :param target: 目标值
    :return: 出现次数
    """
    if not nums:
        return 0

    first = find_left(nums, target)
    if first == len(nums) or nums[first] != target:
        return 0

    last = find_right(nums, target)
    return last - first + 1


def single_non_duplicate(nums):
    """
    LeetCode 540. 有序数组中的单一元素

    时间复杂度: O(log n)
    空间复杂度: O(1)
```

```
    :param nums: 有序数组
    :return: 单一元素
    """
    left, right = 0, len(nums) - 1

    while left < right:
        mid = left + ((right - left) >> 1)

        # 确保 mid 是偶数索引
        if mid % 2 == 1:
            mid -= 1

        # 检查成对关系
        if nums[mid] == nums[mid + 1]:
            # 单一元素在右侧
            left = mid + 2
        else:
            # 单一元素在左侧或就是 mid
            right = mid

    return nums[left]


def min_days_to_bloom(bloom_day, m, k):
    """
    LeetCode 1482. 制作 m 束花所需的最少天数

    时间复杂度: O(n * log(max - min))
    空间复杂度: O(1)

    :param bloom_day: 每朵花开放的天数
    :param m: 需要的花束数量
    :param k: 每束花包含的花朵数量
    :return: 最少等待天数
    """
    # 边界检查：如果花的总数不足以制作 m 束花，返回-1
    if m * k > len(bloom_day):
        return -1

    left = min(bloom_day)
    right = max(bloom_day)

    while left < right:
```

```
        mid = left + ((right - left) >> 1)

        if can_make_bouquets(bloom_day, m, k, mid):
            right = mid  # 可以制作，尝试更少的天数
        else:
            left = mid + 1  # 不能制作，需要更多的天数

    return left


def can_make_bouquets(bloom_day, m, k, day):
    """
    辅助方法：检查在给定天数内能否制作 m 束花
    """
    bouquets = 0  # 已制作的花束数
    flowers = 0   # 当前连续开放的花朵数

    for bloom in bloom_day:
        if bloom <= day:
            flowers += 1
            if flowers == k:
                bouquets += 1
                flowers = 0
                if bouquets == m:
                    return True
        else:
            flowers = 0  # 遇到未开放的花，重置计数

    return bouquets >= m


def min_number_in_rotate_array(nums):
    """
    牛客网: 旋转数组的最小数字
    题目来源: https://www.nowcoder.com/practice/9f3231a991af4f55b95579b44b7a01ba

    题目描述:
    把一个数组最开始的若干个元素搬到数组的末尾,我们称之为数组的旋转。
    输入一个非递减排序的数组的一个旋转,输出旋转数组的最小元素。

    思路分析:
    1. 二分查找法,比较中间元素与右边界元素
    2. 如果中间元素小于右边界,最小值在左侧
```

3. 如果中间元素大于右边界,最小值在右侧
4. 如果相等,右边界左移一位

时间复杂度: O(log n)
空间复杂度: O(1)
是否最优解: 是

:param nums: 旋转数组
:return: 最小元素
"""
```
    if not nums:
        return 0
    left, right = 0, len(nums) - 1
    while left < right:
        mid = left + ((right - left) >> 1)
        if nums[mid] < nums[right]:
            right = mid
        elif nums[mid] > nums[right]:
            left = mid + 1
        else:
            right -= 1
    return nums[left]


def cube_root(n):
    """
```
acwing: 数的三次方根
题目来源: https://www.acwing.com/problem/content/792/

题目描述:
给定一个浮点数 n,求它的三次方根,结果保留 6 位小数。

思路分析:
1. 二分查找法,在[-10000, 10000]范围内搜索
2. 精度控制到 1e-8
3. 根据 mid^3 与 n 的大小关系调整搜索区间

时间复杂度: O(log(范围/精度))
空间复杂度: O(1)
是否最优解: 是

:param n: 输入浮点数
:return: 三次方根

```
    """
    left, right = -10000.0, 10000.0
    precision = 1e-8

    while right - left > precision:
        mid = (left + right) / 2
        if mid * mid * mid >= n:
            right = mid
        else:
            left = mid

    return left


def find_closest_sum(A, B, C, X):
    """
    杭电 OJ: 查找最接近的元素
    题目来源: http://acm.hdu.edu.cn/showproblem.php?pid=2141

    题目描述:
    给定三个数组 A、B、C，以及多个查询 X。对于每个查询，判断是否存在 a∈A, b∈B, c∈C，使得 a+b+c=X。

    思路分析:
    1. 将 A+B 的所有可能和存储在一个数组中
    2. 对这个和数组进行排序
    3. 对于每个查询 X，使用二分查找在 A+B 的和数组中查找是否存在 X-c（c∈C）

    时间复杂度: O(L*M + N*log(L*M))
    空间复杂度: O(L*M)
    是否最优解: 是

    :param A: 数组 A
    :param B: 数组 B
    :param C: 数组 C
    :param X: 查询值
    :return: 是否存在满足条件的组合
    """
    n = len(A)
    sumAB = []
    for a in A:
        for b in B:
            sumAB.append(a + b)
```

```
    sumAB.sort()

    for c in C:
        target = X - c
        left, right = 0, len(sumAB) - 1
        while left <= right:
            mid = left + ((right - left) >> 1)
            if sumAB[mid] == target:
                return True
            elif sumAB[mid] < target:
                left = mid + 1
            else:
                right = mid - 1

    return False


def monthly_expense(expenses, M):
    """
    POJ: 月度开销
    题目来源: http://poj.org/problem?id=3273

    题目描述:
    FJ需要将连续的N天划分为M个时间段，每个时间段的花费是该时间段内每天花费的总和。
    他希望最小化所有时间段中花费的最大值。

    思路分析:
    1. 二分答案法：答案范围在[max(expenses), sum(expenses)]之间
    2. 对于每个可能的答案mid，检查能否将数组分成<=M个子数组且每个子数组和<=mid
    3. 如果可以，说明答案可能更小，向左搜索；否则向右搜索

    时间复杂度: O(n log(sum))
    空间复杂度: O(1)
    是否最优解: 是

    :param expenses: 每天的开销数组
    :param M: 划分的时间段数
    :return: 最小的最大开销
    """
    left, right = 0, 0
    for expense in expenses:
        left = max(left, expense)
        right += expense
```

```
    def can_split(max_sum):
        count, current_sum = 1, 0
        for expense in expenses:
            if current_sum + expense > max_sum:
                count += 1
                current_sum = expense
                if count > M:
                    return False
            else:
                current_sum += expense
        return True

    while left < right:
        mid = left + ((right - left) >> 1)
        if can_split(mid):
            right = mid
        else:
            left = mid + 1

    return left


def cut_trees(trees, M):
    """
    洛谷: 砍树
    题目来源: https://www.luogu.com.cn/problem/P1873

    题目描述:
    伐木工人需要砍倒 M 米高的树木，每棵树的高度不同。
    伐木工人使用一个设定高度 H 的锯子，高于 H 的树木会被砍下高出的部分。
    请帮助伐木工人确定锯子的高度 H，使得被砍下的树木总长度刚好等于 M。

    思路分析:
    1. 二分答案法：高度范围在[0, max(treeHeights)]之间
    2. 对于每个候选高度 mid，计算能砍下的总长度
    3. 如果总长度>=M，可以尝试更高的高度；否则需要降低高度

    时间复杂度: O(n log(maxHeight))
    空间复杂度: O(1)
    是否最优解: 是

    :param trees: 树的高度数组
```

```
    :param M: 需要砍下的总长度
    :return: 锯子的设定高度
    """
    left, right = 0, 0
    for tree in trees:
        right = max(right, tree)

    result = 0
    while left <= right:
        mid = left + ((right - left) >> 1)
        total = 0
        for tree in trees:
            if tree > mid:
                total += tree - mid
        if total >= M:
            result = mid
            left = mid + 1
        else:
            right = mid - 1

    return result


def angry_cows(haybales):
    """
    USACO: 愤怒的奶牛
    题目来源: http://www.usaco.org/index.php?page=viewproblem2&cpid=592

    题目描述:
    在一个一维的场地中放置了若干堆干草，奶牛被点燃后会向左右两个方向传播爆炸。
    爆炸的传播速度是每秒 1 单位距离。求最小的爆炸半径 R，使得点燃任意一个干草堆都能引爆所有干草堆。

    思路分析:
    1. 二分答案法：半径范围在[0, maxPosition - minPosition]之间
    2. 对于每个候选半径 mid，检查是否能通过点燃一个点引爆所有点
    3. 贪心验证：从最左边开始，每次尽可能远地放置引爆点

    时间复杂度: O(n log(maxDistance))
    空间复杂度: O(1)
    是否最优解: 是

    :param haybales: 干草堆的位置数组
    :return: 最小爆炸半径
```

```
    """
    haybales.sort()
    left, right = 0, haybales[-1] - haybales[0]

    def can_explode_all(radius):
        count = 1
        last_pos = haybales[0]
        for i in range(1, len(haybales)):
            if haybales[i] - last_pos > 2 * radius:
                count += 1
                last_pos = haybales[i]
        return count <= 1

    while left < right:
        mid = left + ((right - left) >> 1)
        if can_explode_all(mid):
            right = mid
        else:
            left = mid + 1

    return left


def minimize_maximum(nums, K):
    """
    HackerRank: 最小化最大值
    题目来源: https://www.hackerrank.com/challenges/min-max/problem

    题目描述:
    将一个数组分割成K个非空的连续子数组，设计一个算法使得这K个子数组各自和的最大值最小。

    思路分析:
    1. 二分答案法：答案范围在[max(nums), sum(nums)]之间
    2. 对于每个可能的答案mid，检查能否将数组分成<=K个子数组且每个子数组和<=mid
    3. 如果可以，说明答案可能更小，向左搜索；否则向右搜索

    时间复杂度: O(n log(sum))
    空间复杂度: O(1)
    是否最优解: 是

    :param nums: 数组
    :param K: 分割的子数组数量
    :return: 最小的最大子数组和
```

```
    """
    left, right = 0, 0
    for num in nums:
        left = max(left, num)
        right += num

    def can_split(max_sum):
        count, current_sum = 1, 0
        for num in nums:
            if current_sum + num > max_sum:
                count += 1
                current_sum = num
                if count > K:
                    return False
            else:
                current_sum += num
        return True

    while left < right:
        mid = left + ((right - left) >> 1)
        if can_split(mid):
            right = mid
        else:
            left = mid + 1

    return left



def jump_stones(rocks, M, L):
    """
    计蒜客: 跳石头
    题目来源: https://nanti.jisuanke.com/t/T1201

    题目描述:
    河道中分布着一些巨大岩石，组委会计划移走一些岩石，使得选手们在比赛过程中的最短跳跃距离尽可能长。

    思路分析:
    1. 二分答案法：距离范围在[0, L]之间
    2. 对于每个候选距离 mid，计算需要移走多少块岩石
    3. 如果需要移走的岩石数不超过 M，则可以尝试更大的距离

    时间复杂度: O(n log L)
```

```
    空间复杂度: O(1)
    是否最优解: 是

    :param rocks: 岩石位置数组
    :param M: 最多可移走的岩石数
    :param L: 河道长度
    :return: 最大的最短跳跃距离
    """
    rocks.sort()
    left, right, result = 0, L, 0

    def can_jump(min_dist):
        count, last_pos = 0, 0
        for rock in rocks:
            if rock - last_pos < min_dist:
                count += 1
            else:
                last_pos = rock
        if L - last_pos < min_dist:
            count += 1
        return count <= M

    while left <= right:
        mid = left + ((right - left) >> 1)
        if can_jump(mid):
            result = mid
            left = mid + 1
        else:
            right = mid - 1

    return result


def find_worm_piles(piles, queries):
    """
    Codeforces 474B - Worms
    题目来源: https://codeforces.com/problemset/problem/474/B

    题目描述:
    有 n 堆虫子，第 i 堆有 a_i 条虫子。每条虫子都有一个编号，从 1 开始连续编号。
    给出 m 个查询，每个查询给出一个编号 x，问编号为 x 的虫子属于哪一堆。

    思路分析:
```

1. 使用前缀和数组记录每堆虫子的结束位置
2. 对于每个查询，使用二分查找在前缀和数组中查找 x 所在的位置
3. 找到第一个前缀和大于等于 x 的位置

时间复杂度: O(n + m log n)
空间复杂度: O(n)
是否最优解: 是

```
:param piles: 每堆虫子的数量数组
:param queries: 查询数组
:return: 每个查询对应的堆编号
"""
n, m = len(piles), len(queries)

# 计算前缀和
prefix_sum = [0] * n
prefix_sum[0] = piles[0]
for i in range(1, n):
    prefix_sum[i] = prefix_sum[i - 1] + piles[i]

result = [0] * m

for i in range(m):
    x = queries[i]

    # 使用二分查找找到第一个前缀和 >= x 的位置
    left, right = 0, n - 1
    pile_index = -1

    while left <= right:
        mid = left + ((right - left) >> 1)

        if prefix_sum[mid] >= x:
            pile_index = mid
            right = mid - 1
        else:
            left = mid + 1

    result[i] = pile_index + 1  # 1-based 索引

return result
```

```
def find_max_min(nums):
    """
    SPOJ 题目: 最大最小值问题
    查找数组中的最大值和最小值

    时间复杂度: O(n)
    空间复杂度: O(1)

    :param nums: 数组
    :return: (最大值, 最小值)
    """
    if not nums:
        return (None, None)

    max_val = min_val = nums[0]
    for num in nums[1:]:
        if num > max_val:
            max_val = num
        elif num < min_val:
            min_val = num

    return (max_val, min_val)


def aggressive_cows(positions, cows):
    """
    SPOJ - AGGRCOW (Aggressive cows)
    题目来源: https://www.spoj.com/problems/AGGRCOW/

    题目描述:
    农夫约翰建造了一个有 C 个摊位的畜棚，摊位位于 x0, ..., x(C-1)。
    他的 M 头奶牛总是相互攻击，约翰必须以某种方式分配奶牛到摊位，使它们之间的最小距离尽可能大。

    思路分析:
    1. 二分答案法：距离范围在[0, maxPosition - minPosition]之间
    2. 对于每个候选距离 mid，使用贪心算法验证是否可以放置所有奶牛
    3. 如果可以，说明可以尝试更大的距离；否则需要减小距离

    时间复杂度: O(n log(max-min) * n)
    空间复杂度: O(1)
    是否最优解: 是

    :param positions: 摊位位置数组
```

```
    :param cows: 奶牛数量
    :return: 最大的最小距离
    """
    positions.sort()
    left, right = 0, positions[-1] - positions[0]
    result = 0

    def can_place_cows(min_dist):
        count = 1
        last_position = positions[0]

        for i in range(1, len(positions)):
            if positions[i] - last_position >= min_dist:
                count += 1
                last_position = positions[i]
                if count == cows:
                    return True

        return False

    while left <= right:
        mid = left + ((right - left) >> 1)
        if can_place_cows(mid):
            result = mid
            left = mid + 1
        else:
            right = mid - 1

    return result


# 测试方法
if __name__ == "__main__":
    # 测试基础二分查找
    nums1 = [-1, 0, 3, 5, 9, 12]
    print("基础二分查找测试:")
    print("查找 9:", search(nums1, 9))  # 应该输出 4
    print("查找 2:", search(nums1, 2))  # 应该输出-1

    # 测试查找插入位置
    nums2 = [1, 3, 5, 6]
    print("\n 查找插入位置测试:")
    print("查找 5:", search_insert(nums2, 5))  # 应该输出 2
```

```
print("查找 2:", search_insert(nums2, 2))  # 应该输出 1
print("查找 7:", search_insert(nums2, 7))  # 应该输出 4
print("查找 0:", search_insert(nums2, 0))  # 应该输出 0

# 测试查找范围
nums3 = [5, 7, 7, 8, 8, 10]
print("\n 查找范围测试:")
print("查找 8:", search_range(nums3, 8))  # 应该输出[3, 4]
print("查找 6:", search_range(nums3, 6))  # 应该输出[-1, -1]

# 测试完全平方数
print("\n 完全平方数测试:")
print("16 是完全平方数:", is_perfect_square(16))  # 应该输出 True
print("14 是完全平方数:", is_perfect_square(14))  # 应该输出 False

# 测试寻找峰值
nums4 = [1, 2, 3, 1]
print("\n 寻找峰值测试:")
print("峰值索引:", find_peak_element(nums4))  # 应该输出 2

# 测试寻找旋转数组最小值
nums5 = [3, 4, 5, 1, 2]
print("\n 寻找旋转数组最小值测试:")
print("最小值:", find_min(nums5))  # 应该输出 1

# 测试搜索二维矩阵
matrix = [[1, 3, 5, 7], [10, 11, 16, 20], [23, 30, 34, 60]]
print("\n 搜索二维矩阵测试:")
print("查找 3:", search_matrix(matrix, 3))  # 应该输出 True
print("查找 13:", search_matrix(matrix, 13))  # 应该输出 False

# 测试搜索旋转排序数组
nums6 = [4, 5, 6, 7, 0, 1, 2]
print("\n 搜索旋转排序数组测试:")
print("查找 0:", search_in_rotated_sorted_array(nums6, 0))  # 应该输出 4
print("查找 3:", search_in_rotated_sorted_array(nums6, 3))  # 应该输出-1

# 测试寻找重复数
nums7 = [1, 3, 4, 2, 2]
print("\n 寻找重复数测试:")
print("重复数:", find_duplicate(nums7))  # 应该输出 2

# 测试 Codeforces 题目
```

```
prices = [1, 2, 3, 4, 5]
queries = [3, 10, 1]
drinks = how_many_drinks(prices, queries)
print("\nCodeforces 题目测试:")
print("可以购买的饮料种类数:", drinks)  # 应该输出[3, 5, 1]

# 测试 SPOJ 题目
positions = [1, 2, 8, 4, 9]
print("\nSPOJ 题目测试:")
print("最大最小距离:", aggressive_cows(positions, 3))  # 应该输出 3

# 测试 LeetCode 69: Sqrt(x)
print("\nLeetCode 69: Sqrt(x)测试:")
print("sqrt(4) = ", my_sqrt(4))  # 应该输出 2
print("sqrt(8) = ", my_sqrt(8))  # 应该输出 2
print("sqrt(256) = ", my_sqrt(256))  # 应该输出 16
print("sqrt(2147395600) = ", my_sqrt(2147395600))  # 应该输出 46340

# 测试 LeetCode 374: 猜数字大小
print("\nLeetCode 374: 猜数字大小测试:")
print("猜数字(10) = ", guess_number(10))  # 应该输出 6（假设 pick=6）

# 测试 LeetCode 875: 爱吃香蕉的珂珂
print("\nLeetCode 875: 爱吃香蕉的珂珂测试:")
piles1 = [3, 6, 7, 11]
h1 = 8
print("最小吃香蕉速度:", min_eating_speed(piles1, h1))  # 应该输出 4

piles2 = [30, 11, 23, 4, 20]
h2 = 5
print("最小吃香蕉速度:", min_eating_speed(piles2, h2))  # 应该输出 30

# 测试 LeetCode 1011: 在 D 天内送达包裹的能力
print("\nLeetCode 1011: 在 D 天内送达包裹的能力测试:")
weights1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
days1 = 5
print("最小运载能力:", ship_within_days(weights1, days1))  # 应该输出 15

weights2 = [3, 2, 2, 4, 1, 4]
days2 = 3
print("最小运载能力:", ship_within_days(weights2, days2))  # 应该输出 6

# 测试 AtCoder ABC023D: 射撃王
```

```
print("\nAtCoder ABC023D: 射撃王测试:")
points1 = [[0, 0], [1, 1], [2, 2], [3, 3], [4, 4]]
print("最少子弹数:", min_bullets(points1))  # 应该输出 1

points2 = [[0, 0], [1, 0], [0, 1], [1, 1]]
print("最少子弹数:", min_bullets(points2))  # 应该输出 2

# 测试 LeetCode 410: 分割数组的最大值
print("\nLeetCode 410: 分割数组的最大值测试:")
split_nums1 = [7, 2, 5, 10, 8]
print("最小的最大子数组和(m=2):", split_array(split_nums1, 2))  # 应该输入 18
split_nums2 = [1, 2, 3, 4, 5]
print("最小的最大子数组和(m=2):", split_array(split_nums2, 2))  # 应该输出 9

# 测试 LeetCode 4: 寻找两个正序数组的中位数
print("\nLeetCode 4: 寻找两个正序数组的中位数测试:")
median_nums1 = [1, 3]
median_nums2 = [2]
print("中位数:", find_median_sorted_arrays(median_nums1, median_nums2))  # 应该输出 2.0
median_nums3 = [1, 2]
median_nums4 = [3, 4]
print("中位数:", find_median_sorted_arrays(median_nums3, median_nums4))  # 应该输出 2.5

# 测试牛客/剑指 Offer: 数字在排序数组中出现的次数
print("\n 牛客/剑指 Offer: 数字在排序数组中出现的次数测试:")
count_nums = [1, 2, 3, 3, 3, 3, 4, 5]
print("3 出现的次数:", get_number_of_k(count_nums, 3))  # 应该输出 4
print("6 出现的次数:", get_number_of_k(count_nums, 6))  # 应该输出 0

# 测试 LeetCode 540: 有序数组中的单一元素
print("\nLeetCode 540: 有序数组中的单一元素测试:")
single_nums1 = [1, 1, 2, 3, 3, 4, 4, 8, 8]
print("单一元素:", single_non_duplicate(single_nums1))  # 应该输出 2
single_nums2 = [3, 3, 7, 7, 10, 11, 11]
print("单一元素:", single_non_duplicate(single_nums2))  # 应该输出 10

# 测试 LeetCode 1482: 制作 m 束花所需的最少天数
print("\nLeetCode 1482: 制作 m 束花所需的最少天数测试:")
bloom_days1 = [1, 10, 3, 10, 2]
print("最少天数(m=3, k=1):", min_days_to_bloom(bloom_days1, 3, 1))  # 应该输出 3
bloom_days2 = [1, 10, 3, 10, 2]
print("最少天数(m=3, k=2):", min_days_to_bloom(bloom_days2, 3, 2))  # 应该输出-1
bloom_days3 = [7, 7, 7, 7, 12, 7, 7]
```

```
print("最少天数(m=2, k=3):", min_days_to_bloom(bloom_days3, 2, 3))  # 应该输出 12

# 测试牛客网题目: 旋转数组的最小数字
print("\n 牛客网题目: 旋转数组的最小数字测试:")
rotate_nums1 = [3, 4, 5, 1, 2]
print("旋转数组最小值:", min_number_in_rotate_array(rotate_nums1))  # 应该输出 1
rotate_nums2 = [2, 2, 2, 0, 1]
print("旋转数组最小值:", min_number_in_rotate_array(rotate_nums2))  # 应该输出 0

# 测试 acwing 题目: 数的三次方根
print("\nacwing 题目: 数的三次方根测试:")
print("8 的三次方根:", cube_root(8))  # 应该输出 2.0
print("27 的三次方根:", cube_root(27))  # 应该输出 3.0
print("1000 的三次方根:", cube_root(1000))  # 应该输出 10.0

# 测试 Codeforces 题目: 寻找边界
print("\nCodeforces 题目: 寻找边界测试:")
cf_nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print("第一个>=5 的位置:", find_left(cf_nums, 5))  # 应该输出 4
print("最后一个<=5 的位置:", find_right(cf_nums, 5))  # 应该输出 4

# 测试 SPOJ 题目: 最大最小值问题
print("\nSPOJ 题目: 最大最小值问题测试:")
spo_nums = [1, 2, 3, 4, 5]
print("最大最小值:", find_max_min(spo_nums))  # 应该输出(5, 1)

# 测试 AtCoder 题目: 二分查找应用
print("\nAtCoder 题目: 二分查找应用测试:")
at_nums = [1, 3, 5, 7, 9]
print("查找 7:", search(at_nums, 7))  # 应该输出 3
print("查找 6:", search(at_nums, 6))  # 应该输出-1

# 测试 HackerRank 题目: 二分查找变种
print("\nHackerRank 题目: 二分查找变种测试:")
hr_nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print("查找插入位置 5:", search_insert(hr_nums, 5))  # 应该输出 4
print("查找插入位置 11:", search_insert(hr_nums, 11))  # 应该输出 10

# 测试新增题目
print("\n=== 新增各大平台题目测试 ===")

# 测试杭电 OJ 题目
A = [1, 2, 3]
```

```
B = [4, 5, 6]
C = [7, 8, 9]
print("杭电OJ-最接近和(15):", find_closest_sum(A, B, C, 15))  # 应该输出True

# 测试POJ题目
expenses = [100, 200, 300, 400, 500]
print("POJ-月度开销(3段):", monthly_expense(expenses, 3))  # 应该输出500

# 测试洛谷题目
trees = [20, 15, 10, 17]
print("洛谷-砍树(7米):", cut_trees(trees, 7))  # 应该输出15

# 测试USACO题目
haybales = [1, 2, 4, 8, 9]
print("USACO-愤怒的奶牛:", angry_cows(haybales))  # 应该输出4

# 测试HackerRank题目
nums = [1, 2, 3, 4, 5]
print("HackerRank-最小化最大值(3段):", minimize_maximum(nums, 3))  # 应该输出6

# 测试计蒜客题目
rocks = [2, 11, 14, 17, 21]
print("计蒜客-跳石头(移2块):", jump_stones(rocks, 2, 25))  # 应该输出4

# 测试Codeforces题目
worm_piles = [1, 3, 2, 4]
worm_queries = [1, 4, 7, 10]
worm_results = find_worm_piles(worm_piles, worm_queries)
print("Codeforces-虫子堆:", worm_results)  # 应该输出[1, 2, 3, 4]

# 测试各大平台题目综合测试
print("\n各大平台题目综合测试:")
print("所有二分查找算法测试完成!")
print("=" * 50)
print("二分查找算法总结:")
print("1. 时间复杂度: O(log n)")
print("2. 空间复杂度: O(1)")
print("3. 适用场景: 有序数组查找")
print("4. 关键技巧: 边界处理、中点计算、循环条件")
print("5. 常见变种: 查找边界、旋转数组、二维搜索")
print("6. 工程化考量: 异常处理、性能优化、边界测试")
```

==========

文件: Complexity.java

```
==========================================
package class007;

import java.util.ArrayList;

public class Complexity {

    // 只用一个循环完成冒泡排序
    // 但这是时间复杂度O(N^2)的!
    public static void bubbleSort(int[] arr) {
        if (arr == null || arr.length < 2) {
            return;
        }
        int n = arr.length;
        int end = n - 1, i = 0;
        while (end > 0) {
            if (arr[i] > arr[i + 1]) {
                swap(arr, i, i + 1);
            }
            if (i < end - 1) {
                i++;
            } else {
                end--;
                i = 0;
            }
        }
    }

    public static void swap(int[] arr, int i, int j) {
        int tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
    }

    public static void main(String[] args) {
        // 随机生成长度为n
        // 值在0~v-1之间
        // 且任意相邻两数不相等的数组
        int n = 10;
        int v = 4;
        int[] arr1 = new int[n];
```

```
arr1[0] = (int) (Math.random() * v);
for (int i = 1; i < n; i++) {
    do {
        arr1[i] = (int) (Math.random() * v);
    } while (arr1[i] == arr1[i - 1]);
}
for (int num : arr1) {
    System.out.print(num + " ");
}
System.out.println();
System.out.println("=========");

// java 中的动态数组是 ArrayList
// 各个语言中的动态数组的初始大小和实际扩容因子可能会变化，但是均摊都是 O(1)
// 课上用 2 作为扩容因子只是举例而已
ArrayList<Integer> arr2 = new ArrayList<>();
arr2.add(5); // 0
arr2.add(4); // 1
arr2.add(9); // 2
arr2.set(1, 6); // arr[1]由 4 改成了 6
System.out.println(arr2.get(1));
System.out.println("=========");

int[] arr = { 64, 31, 78, 0, 5, 7, 103 };
bubbleSort(arr);
for (int num : arr) {
    System.out.print(num + " ");
}
System.out.println();
System.out.println("=========");

int N = 200000;
long start;
long end;
System.out.println("测试开始");
start = System.currentTimeMillis();
for (int i = 1; i <= N; i++) {
    for (int j = i; j <= N; j += i) {
        // 这两个嵌套 for 循环的流程，时间复杂度为 O(N * logN)
        // 1/1 + 1/2 + 1/3 + 1/4 + 1/5 + ... + 1/n，也叫"调和级数"，收敛于 O(logN)
        // 所以如果一个流程的表达式 : n/1 + n/2 + n/3 + ... + n/n
        // 那么这个流程时间复杂度 O(N * logN)
    }
```

```
        }
        end = System.currentTimeMillis();
        System.out.println("测试结束，运行时间 : " + (end - start) + " 毫秒");

        System.out.println("测试开始");
        start = System.currentTimeMillis();
        for (int i = 1; i <= N; i++) {
            for (int j = i; j <= N; j++) {
                // 这两个嵌套 for 循环的流程，时间复杂度为 O(N^2)
                // 很明显等差数列
            }
        }
        end = System.currentTimeMillis();
        System.out.println("测试结束，运行时间 : " + (end - start) + " 毫秒");

    }
}
```

==========