二分查找（Binary Search）算法

王思成

Oct 01, 2025

在计算机科学中，二分查找是一种高效且基础的搜索算法。它之所以重要，是因为它能在海量数据中快速定位目标。想象一下，如果你需要从一本百万词汇的字典中找到「Algorithm」这个词，逐页翻阅的线性查找方式可能需要数小时，而二分查找则通过不断对半分割搜索区间，在短短几秒内完成。这种效率源于其 O(log n) 的时间复杂度，意味着即使数据规模巨大，查找次数也仅以对数增长。二分查找不仅是面试中的高频考点，更是许多实际系统如数据库索引和路由表的核心组件。本文将从零开始，深入解析二分查找的思想、实现细节和常见陷阱，帮助读者彻底掌握这一算法。

二分查找的重要性体现在其惊人的效率和应用广泛性上。以一个引人入胜的场景为例，当你在有序数组中搜索一个元素时，线性查找需要逐个比较，而二分查找通过每次将搜索区间减半，迅速缩小范围。例如，在百万级数据中，线性查找可能需要百万次比较，而二分查找仅需约二十次。这种效率提升在现实世界中至关重要，例如在大型数据库或游戏逻辑中快速检索数据。本文的目标不仅是教会读者写对二分查找代码，更要深入理解其每个细节，包括循环条件、中间值计算和边界更新等经典难题，从而在实际应用中游刃有余。

1　算法思想与前提条件

二分查找的核心思想是分而治之，它将有序的搜索区间不断对半分割，通过比较中间元素与目标值，逐步缩小范围直至找到目标或确认不存在。然而，二分查找并非万能，它依赖于两个必要前提：数据集必须有序，无论是升序还是降序排列；同时，数据结构必须支持随机访问，例如数组可以在常数时间内通过索引访问任意元素，而链表则不适合标准二分查找。基本步骤包括确定初始搜索区间、计算中间位置、比较中间元素与目标值，并根据比较结果调整边界。具体来说，对于升序数组，首先设置左右指针为数组的起始和结束索引，然后循环计算中间索引，如果中间元素等于目标值则返回索引；如果中间元素小于目标值，则调整左指针到中间索引加一的位置；如果中间元素大于目标值，则调整右指针到中间索引减一的位置。重复这一过程直到找到目标或区间无效。

2　图解算法流程

由于本文避免使用图片，我们将通过文字详细描述二分查找的流程。假设有一个升序数组 [1, 3, 5, 7, 9, 11]，目标是查找数字 7。初始时，左指针指向索引 0，右指针指向索引 5，搜索区间为 [0, 5]。第一步计算中间索引，使用公式 mid = left + (right - left) // 2，得到 mid = 2，对应元素 5。比较 5 和 7，由于 5 小于 7，目标在右侧，因此调整左指针为 mid + 1 = 3。新区间为 [3, 5]，中间索引 mid = 4，对应元素 9。比较 9 和 7，由于 9 大于 7，目标在左侧，调整右指针为 mid - 1 = 3。新区间为 [3, 3]，中间索引 mid = 3，对应元素 7，与目标相等，返回索引 3。如果目标不存在，例如查找数字 4，过程类似，但最终左指针超过右指针，返回 -1 表示未找到。

3 关键实现细节与“坑点”剖析

二分查找的实现看似简单，但细节决定成败。首先，循环条件的选择至关重要。常见的有两种：while (left <= right) 和 while (left < right)。前者表示搜索区间为闭区间 [left, right]，当 left 等于 right 时，区间仍有一个元素需要检查，这是最推荐的方式，因为它不易出错。后者表示左闭右开区间 [left, right)，当 left 等于 right 时区间为空，需要更细致的边界处理，容易导致遗漏或错误。其次，中间值计算时，直接使用 mid = (left + right) / 2 可能存在整数溢出风险，当 left 和 right 都很大时，它们的和可能超出整型最大值。解决方案是使用 mid = left + (right - left) // 2 或位运算 mid = (left + right) >> 1（在 Java 等语言中），这些方法避免了溢出问题。最后，边界更新必须排除已检查的元素，即 left = mid + 1 和 right = mid - 1，这是因为 arr[mid] 已经被比较过，如果不排除，可能导致死循环，尤其是在区间缩小到单个元素时。

4 标准实现代码（迭代版本）

以下是二分查找的迭代版本 Python 代码，我们将逐段解读其关键部分。

```
def binary_search(arr, target):
    """
    在升序数组 arr 中查找 target。
    找到则返回其索引，否则返回-1。
    """
    left, right = 0, len(arr) - 1 # 初始化搜索区间为闭区间 [0, n-1]

    while left <= right: # 当区间不为空时循环
        mid = left + (right - left) // 2 # 防止溢出
        # mid = (left + right) // 2 # 在 Python 中通常不会溢出，但好习惯是使用上面的写法

        if arr[mid] == target:
            return mid # 找到目标，返回索引
        elif arr[mid] < target:
            left = mid + 1 # 目标在右侧，调整左边界
        else: # arr[mid] > target
            right = mid - 1 # 目标在左侧，调整右边界

    return -1 # 未找到目标
```

在这段代码中，首先初始化左指针 left 为 0，右指针 right 为数组长度减一，这定义了初始搜索区间为闭区间 [0, n-1]。循环条件使用 left <= right，确保在区间内所有元素都被检查。中间索引 mid 的计算采用 left + (right - left) // 2，这是一种防止整数溢出的安全方法，尽管在 Python 中整数不会溢出，但养成这种习惯有助于跨语言应用。在循环体内，通过比较 arr[mid] 与 target 决定下一步操作：如果相等，直接返回 mid；如果 arr[mid] 小于 target，说明目标在右侧，因此将 left 更新为 mid + 1，排除已检查的中间元素；如果 arr[mid]

大于 target，则将 right 更新为 mid - 1。如果循环结束仍未找到目标，返回 -1。

5 其他实现方式

除了迭代版本，二分查找还可以通过递归实现。递归版本将搜索过程分解为子问题，每次递归调用处理一半的区间。例如，在递归函数中，基础情况是区间为空或找到目标，否则根据中间值比较结果递归调用左半部分或右半部分。递归实现的空间复杂度为 O(log n)，因为递归调用栈的深度与数据规模的对数成正比，而迭代版本的空间复杂度为 O(1)，只使用固定变量。另一种常见实现是使用左闭右开区间 [left, right)，在这种方式下，初始右指针为数组长度，循环条件为 left < right，边界更新时 right 直接设为 mid，而不是 mid - 1。这种方式需要更仔细的初始化，但有时在特定场景下更简洁。
递归版本代码示例如下：

```
def binary_search_recursive(arr, target, left, right):
    if left > right:
        return -1
    mid = left + (right - left) // 2
    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        return binary_search_recursive(arr, target, mid + 1, right)
    else:
        return binary_search_recursive(arr, target, left, mid - 1)
```

这里，递归函数接受数组、目标值和当前区间左右指针作为参数。如果左指针超过右指针，返回 -1 表示未找到；否则计算中间索引，比较后递归调用左或右半部分。这种实现虽然直观，但需要注意递归深度可能导致的栈溢出问题。

6 复杂度分析

二分查找的时间复杂度为 O(log n)，其中 n 是数据规模。这是因为每次循环或递归调用都将问题规模减半，最坏情况下需要执行 log ☒ (n) 次操作。例如，当 n=100 万时，log ☒ (n) 约等于 20，而线性查找可能需要 100 万次比较，凸显了二分查找的高效性。空间复杂度方面，迭代版本为 O(1)，因为它只使用常数级别的额外空间；递归版本为 O(log n)，由于递归调用栈的深度与 log n 成正比。用 LaTeX 公式表示，时间复杂度为 (O(\log n))，空间复杂度迭代版本为 (O(1))，递归版本为 (O(\log n))。

7 常见变体与应用场景

二分查找有多种变体，适用于不同场景。例如，在重复元素数组中寻找第一个等于目标的元素，需要调整边界更新策略，在找到目标后继续向左搜索；寻找最后一个等于目标的元素则向右搜索。另一种变体是寻找第一个大于等于目标的元素，常用于插入位置确定，例如在排序列表中插入新值。这些变体在数据库索引和范围查询中广泛应用，通过微调比较和边界逻辑，可以解决更复杂的问题。每个变体的核心在于理解搜索区间的定义和边界排除

原则，确保算法正确性。

二分查找的核心思想是通过分治策略在有序数据中高效搜索，其前提是数据有序且支持随机访问。实现时需注意循环条件推荐使用闭区间、中间值计算防溢出、边界更新排除已检查元素。关键口诀包括区间定义清晰、中间计算安全、边界更新彻底。通过实践，读者可以在力扣等平台练习相关题目，如二分查找和搜索插入位置，以巩固理解。掌握这些细节后，二分查找将不再令人畏惧，而是成为解决实际问题的有力工具。

8 思考题

为了进一步加深理解，请思考以下问题：如果数组是降序排列，代码需要如何修改？二分查找是否可以用在链表上，如果能，时间复杂度是多少？除了搜索，二分思想还能解决哪些问题，例如求平方根或在旋转数组中搜索？这些问题鼓励读者探索算法的灵活性和应用边界。