C++算法讲义： 二分查找

1. 引言

- **问题引入：猜数字游戏**
  从 1 到 1000 猜一个数字，你会怎么猜?
  - **线性查找 (老实人)**：从 1 开始逐个猜。最坏情况猜 1000 次，$O(N)$。
  - **二分查找 (聪明人)**：每次猜中间，根据反馈排除一半区间。最坏情况只需 10 次 ($2^{10} > 1000$)，$O(\log N)$。
    当数据量达到 42 亿时，线性查找需 42 亿次，而二分查找仅需 32 次，效率差距天壤之别!
- **核心前提**
  二分查找之所以高效，是因为它有一个**前提**：**数据必须具有单调性（即有序）**。如果数据是乱序的，折半策略将失效。就像在乱七八糟的纸堆里找东西，你不可能"折半"查找。
- **学习价值**
  掌握二分查找是 C++ 算法学习者的必备技能：
  - **高效搜索**：在大量有序数据中快速定位，提升程序性能。
  - **解决变体问题**：很多复杂问题通过转化，也能应用二分查找思想。
  - **面试高频考点**：它不仅考察代码实现，更考察对边界和变体问题的处理能力。
- **学习目标**
  学完此部分，你应该能：
  1. 理解二分查找的**分治原理**和**对数时间复杂度**。
  2. 掌握数据**有序性**这一核心前提。
  3. 熟练编写**经典二分查找模板**，并理解其闭区间 `l, r`、`l <= r` 循环条件和 `mid` 计算方式。
  4. 掌握修改模板，解决**查找第一个/最后一个出现的目标值**等变体问题。
  5. 识别**死循环陷阱**，并正确处理 `mid` 计算与边界更新关系。

---

2. 核心思想

2.1 二分查找的定义

二分查找（Binary Search）是一种基于**分治策略**的高效搜索算法。它利用数据的**有序性**，每轮将搜索范围减半，直到找到目标元素或搜索区间为空。

类比词典查单词：打开中间页，如果单词靠前就往前翻，靠后就往后翻，每次都排除一半无关的页码。

2.2 基本原理：三指针与区间缩减

在有序数组 `arr` 中查找 `target`，我们使用三个"指针"：

- `l` (左边界)：搜索区间的头部索引。
- `r` (右边界)：搜索区间的尾部索引。
- `mid` (中间点)：`mid = l + (r - l) / 2`。

核心步骤（循环执行）：

1. **计算 `mid`**：`mid = l + (r - l) / 2` (向下取整)。这种写法能**有效防止 `l + r` 溢出**。
2. **比较 `arr[mid]` 与 `target`**：

- `arr[mid] < target`：目标在 `mid` 的右边，更新 `l = mid + 1`。
- `arr[mid] > target`：目标在 `mid` 的左边，更新 `r = mid - 1`。
- `arr[mid] == target`：找到目标，返回 `mid`。

2.3 经典二分查找模板 (闭区间 `[l, r]`)

我们推荐使用**闭区间** `[l, r]` 的写法，因为它清晰直观，不易出错。

```
// 在有序数组 nums 中查找 target，找到返回下标，没找到返回 -1
int bs(const vector<int>& nums, int target) {
    int l = 0;
    int r = nums.size() - 1; // r 是最后一个元素的下标，[l, r] 闭区间

    // 循环条件：当 l <= r 时，表示搜索区间 [l, r] 依然有效
    // 即使 l == r，区间中仍有一个元素，需要检查
    while (l <= r) {
        int mid = l + (r - l) / 2; // 防溢出写法，向下取整

        if (nums[mid] == target) {
            return mid;          // 找到了，直接返回
        } else if (nums[mid] < target) {
            l = mid + 1;       // 目标在右半边，新的左边界变为 mid + 1
        } else { // nums[mid] > target
            r = mid - 1;      // 目标在左半边，新的右边界变为 mid - 1
        }
    }
    // 循环结束，说明搜索区间为空（l > r），target 不存在于数组中
    return -1;
}
```

代码解读要点：

1. `r = nums.size() - 1;`：`r` 初始化为最后一个元素的下标，定义 `[l, r]` 为闭区间。
2. `while (l <= r)`：关键条件。当 `l == r` 时，区间中仍有 1 个元素，需要检查。
3. `l = mid + 1;` 和 `r = mid - 1;`：当 `nums[mid]` 不等于 `target` 时，`mid` 肯定不是目标，所以可以直接排除 `mid`。

---

3. 实例

3.1 查找第一个出现的元素 (OJ:B0189)

题目： 有序数组 `nums` 可能包含重复元素。查找 `target` 第一次出现的索引，未找到返回 -1。

解决思路：找到了别停，继续向左收缩！
当 `nums[mid] == target` 时，先记录 `mid` 可能是答案，然后向左 (`r = mid - 1`) 继续搜索，看是否存在更靠前的 `target`。

```
// 查找第一次出现 target 的索引
int bsFirst(const vector<int>& nums, int target) {
    int l = 0;            // 左边界索引
    int r = nums.size() - 1; // 右边界索引
    int ans = -1;        // 记录可能的结果
```

```
    while (l <= r) {
        int mid = l + (r - l) / 2;

        if (nums[mid] == target) {
            ans = mid;       // 记录当前索引
            r = mid - 1;     // 继续向左收缩，找更靠前的
        } else if (nums[mid] < target) {
            l = mid + 1;      // 目标在右侧
        } else { // nums[mid] > target
            r = mid - 1;     // 目标在左侧
        }
    }
    return ans; // 返回记录的第一个 target 的索引
}
```

流程 (查找 `2` 在 `[1, 2, 2, 2, 3, 5, 7, 9]`):

1. `l=0, r=7, mid=3` (指向第二个 `2`)。`nums[3]==2`。`ans=3`，`r=2`。
2. `l=0, r=2, mid=1` (指向第一个 `2`)。`nums[1]==2`。`ans=1`，`r=0`。
3. `l=0, r=0, mid=0` (指向 `1`)。`nums[0]==1 < 2`。`l=1`。
4. `l=1, r=0`。`l > r`，循环结束。返回 `ans = 1`。

3.2 查找最后一个出现的元素

题目： 寻找这个数字在序列中**最后一次出现**的下标。

思路：与查找第一个出现的元素类似，但策略相反——**找到了别停，继续向右收缩！**
当 `nums[mid] == target` 时，先记录 `mid` 可能是答案，然后向右(`l = mid + 1`)继续搜索，看是否存在更靠后的 `target`。

```
// 查找最后一个出现 target 的索引
int bsLast(const vector<int>& nums, int target) {
    int l = 0;
    int r = nums.size() - 1;
    int ans = -1;

    while (l <= r) {
        int mid = l + (r - l) / 2;

        if (nums[mid] == target) {
            ans = mid;       // 记录当前索引
            l = mid + 1;     // 继续向右收缩，找更靠后的
        } else if (nums[mid] < target) {
            l = mid + 1;      // 目标在右侧
        } else { // nums[mid] > target
            r = mid - 1;     // 目标在左侧
        }
    }
    return ans;
}
```

3.3 区间收缩排除法 (左闭右开 `[l, r)`)

这是一种不同风格的二分查找模板，它不是在循环中“找到就返回”，而是不断缩小区间，直到区间只剩下一个元素，最后再检查这个元素是不是我们要找的。

```
// 查找第一个等于或大于 target 的元素
int bsExclude(const vector<int>& nums, int target) {
    int l = 0;
    int r = nums.size(); // [l, r) 左闭右开，r 初始化为 size

    // 循环条件：当 l < r 时，表示搜索区间 [l, r) 依然有效
    // 最终 l == r，指向第一个满足条件的元素或 size
    while (l < r) {
        int mid = l + (r - l) / 2; // 向下取整
        if (nums[mid] >= target) {
            r = mid; // mid 可能是答案，或者答案在 mid 左边，所以把 r 收缩到 mid (不排除
mid)
        } else { // nums[mid] < target
            l = mid + 1; // mid 太小，排除 mid，把 l 移到 mid + 1
        }
    }
    // 循环结束时，l == r。这个 l 就是第一个 >= target 的位置。
    // 需要后处理：检查 l 是否越界且 nums[l] 是否等于 target。
    if (l < nums.size() && nums[l] == target) {
        return l;
    } else {
        return -1;
    }
}
```

代码解读要点：

1. **区间定义：** `[l, r)` **(左闭右开)。** `r` 初始化为 `nums.size()`。
2. `while (l < r)`：当 `l == r` 时循环终止。
3. `if (nums[mid] >= target) r = mid;`：`mid` 可能含有答案，因此 `mid` 不被排除。
4. `else l = mid + 1;`：`mid` 肯定不是答案，因此 `mid` 被排除。
5. **后处理**：循环结束时 `l == r`，这个 `l` 是第一个 `>= target` 的位置。需要额外检查。

3.4 陷阱：二分查找的死循环

死循环常发生在**区间只剩 2 个元素**时 (`l`, `l+1`)。
原因：`mid` 的取整方向与边界更新方向不一致。

- 如果策略是 `l = mid` (保留 `mid` 作为左边界)，且 `mid` 是**向下取整** (偏向 `l`)，那么当 `r = l+1` 时 `mid` 永远等于 `l`，导致 `l = mid` 陷入 `l=l` 的死循环。
- **解决方案**：
  - **当 `l = mid` 更新时，`mid` 必须**向上取整：`mid = l + (r - l + 1) / 2`。
  - **当 `r = mid` 更新时，`mid` 必须**向下取整：`mid = l + (r - l) / 2`。

逻辑本质： 二分查找的死循环，永远发生在区间还剩 2 个元素的时候。`mid` 必须跳过那个被保留的边界，去测试另一个元素，这样区间才能缩小。

示例：寻找最后一个 `target`，使用 `[l, r)` 和 `l = mid` 策略时，正确的 `mid` 计算

```cpp
// 查找最后一个等于 target 的元素
int bsLastOpt(const vector<int>& nums, int target) {
    int l = 0;
    int r = nums.size(); // [l, r) 左闭右开

    int lastIdx = -1; // 记录可能的最后一个 target 索引

    while (l < r) {
        // mid 向上取整，以避免 l = mid 时死循环
        int mid = l + (r - l + 1) / 2; // 注意这里 +1，让 mid 偏向右边

        if (nums[mid] <= target) { // mid 可能是答案，或答案在 mid 右边，包含 mid
            lastIdx = mid; // 记录当前可能的答案
            l = mid;       // 保留 mid，向右搜索 [mid, r)
        } else { // nums[mid] > target // mid 太大了，答案在 mid 左边，不包含 mid
            r = mid - 1; // 排除 mid，向左搜索 [l, mid-1)
        }
    }
    // 循环结束，检查 lastIdx 是否有效且等于 target
    if (lastIdx != -1 && nums[lastIdx] == target) {
        return lastIdx;
    } else {
        return -1;
    }
}
```

---

4. 总结

- **要点回顾**
    1. **核心**：利用有序性，每次排除一半搜索空间，$O(\log N)$。
    2. **前提**：数据必须有序。
    3. **经典模板 (`[l, r]` 闭区间)**：`r = size - 1; while (l <= r); l = mid + 1; r = mid - 1;` 找到即可返回。
    4. **变体**：
        - **查找第一个**：`nums[mid] == target` 时，`ans = mid; r = mid - 1;` (向左收缩)。
        - **查找最后一个**：`nums[mid] == target` 时，`ans = mid; l = mid + 1;` (向右收缩)。
    5. **区间收缩排除法 (`[l, r)` 左闭右开)**：`r = size; while (l < r);` 最终 `l == r` 需后处理。
    6. **死循环**：`mid` 取整方向需与边界更新方向一致。
        - `l = mid` 时，`mid` 向上取整 (`mid = l + (r - l + 1) / 2`)。
        - `r = mid` 时，`mid` 向下取整 (`mid = l + (r - l) / 2`)。