BStemplate

二分答案模板。

```
l = mid, r = mid
初始化区间时， l 和 r 都设置成开区间 ()
二分到的 mid 都是值域范围内的
最后一定会跑到分界点，
小的找最大就是 l
大的找最小就是 r
找到那一头依然是开区间没动就是无解
```

这个模板确实非常优秀，是竞赛中非常流行的一种写法，被称为 “**左闭右开**” 区间二分法。它的核心思想是维护一个左闭右开的区间 `[lo, hi)`，其中 `lo` 是当前已知的不满足条件的最大值，`hi` 是当前已知的满足条件的最小值。通过不断将区间缩小，最终 `hi` 会收敛到最小的满足条件的解。

1. ”大找小 r“模板解析

我们来逐行解析一下这个模板：

```
// 1. 初始化区间 [lo, hi)
// lo 通常设置为一个不可能的小值（小于所有可能的答案）
// hi 通常设置为一个不可能的大值（大于所有可能的答案）
i64 lo = -1, hi = 1E9 + 1;

// 2. 循环条件：只要区间长度大于 1，就继续二分
// 因为 [lo, hi) 是左闭右开区间，当 lo + 1 == hi 时，区间内已经没有元素了
while (lo + 1 < hi) {
    // 3. 计算中间值 mid
    // 这种写法可以避免 lo + hi 溢出，和 lo + (hi - lo) / 2 等价
    i64 mid = lo + (hi - lo) / 2;

    // 4. 核心判断：检查 mid 是否满足题目条件
    if (check(mid)) {
        // 4.1 如果满足条件：
        // 说明 mid 是一个可行解，但可能不是最小的可行解
        // 我们需要在 [lo, mid) 区间内继续寻找更小的可行解
        hi = mid;
    } else {
```

```
            // 4.2 如果不满足条件：
            // 说明 mid 不是一个可行解，且所有小于 mid 的值也不可能是可行解
            // 我们需要在 [mid, hi) 区间内继续寻找
            lo = mid;
        }
    }

    // 5. 循环结束后，hi 就是最小的满足条件的解
    cout << hi << "\n";
```

核心思想

这个模板的核心思想是 **“排除法”**。它通过 `check` 函数的结果，不断地将不可能包含答案的区间部分排除掉。

- **当 `check(mid)` 为 `true` 时**：`mid` 是一个可行解。但因为我们要找的是**最小的**可行解，所以所有大于 `mid` 的值都可以被排除。因此，我们将 `hi` 移动到 `mid`，使得新的搜索区间变为 `[lo, mid)`。
- **当 `check(mid)` 为 `false` 时**：`mid` 不是一个可行解。并且，由于答案具有单调性（这是二分答案的前提），所有小于 `mid` 的值也都不可能是可行解。因此，我们将 `lo` 移动到 `mid`，使得新的搜索区间变为 `[mid, hi)`。

优点

这个模板相比你之前提供的 “闭区间” 模板，有以下几个显著的优点：

1. **代码更简洁**：
    - 不需要额外的变量（如 `ans`）来记录答案。
    - 循环条件 `lo + 1 < hi` 非常直观。
    - 区间调整时只需要 `hi = mid` 或 `lo = mid`，不需要 `+1` 或 `-1` 的操作，**有效避免了因边界处理不当而导致的死循环或漏解**。
2. **逻辑更清晰**：
    - 它清晰地将区间划分为 “不满足条件的区域” 和 “满足条件的区域”。
    - `lo` 始终代表着 “最大的不满足条件的值”，`hi` 始终代表着 “最小的满足条件的值”。
    - 循环的过程就是不断地缩小这两个区域之间的距离，直到它们相邻，此时 `hi` 就是我们想要的答案。

适用场景

这个模板特别适合解决 **“寻找最小的满足条件的解”** 这类问题。例如：

- 寻找第一个大于等于目标值的元素。
- 寻找最小的可行方案（如最小的时间、最小的代价等）。

如果题目要求的是 **“寻找最大的满足条件的解”**，这个模板同样可以使用，只是需要对 `check` 函数和区间调整逻辑进行一些微调。通常的做法是：

- 修改 `check` 函数，使其判断 `mid` 是否**不**满足条件。
- 或者，在循环结束后，返回 `lo` 而不是 `hi`。

示例

让我们用一个简单的例子来演示一下这个模板的工作原理。

问题：在一个有序数组 `[1, 3, 5, 7, 9]` 中，寻找第一个大于等于 `6` 的元素。

- **初始区间**：`lo = -1`，`hi = 5` (因为数组长度为 5，索引从 0 到 4)。区间 `[-1, 5)` 包含了所有可能的索引。
- **第一次循环**：`lo + 1 = 0 < hi = 5`，循环继续。
  - `mid = -1 + (5 - (-1)) / 2 = -1 + 3 = 2`。
  - `check(2)`：判断 `nums[2]` (值为 5) 是否大于等于 6？`false`。
  - 因为不满足，所以 `lo = mid = 2`。新的区间是 `[2, 5)`。
- **第二次循环**：`lo + 1 = 3 < hi = 5`，循环继续。
  - `mid = 2 + (5 - 2) / 2 = 2 + 1 = 3`。
  - `check(3)`：判断 `nums[3]` (值为 7) 是否大于等于 6？`true`。
  - 因为满足，所以 `hi = mid = 3`。新的区间是 `[2, 3)`。
- **第三次循环**：`lo + 1 = 3` 不小于 `hi = 3`，循环结束。
- **最终结果**：`hi = 3`，即数组中第一个大于等于 6 的元素的索引是 3，对应的值是 7。

这个示例完美地展示了模板如何一步步收敛到正确的答案。

总结

你喜欢的这个模板是一个非常强大且高效的二分答案实现方式。它的**简洁性**和**鲁棒性**使其成为算法竞赛中的首选。

记住它的核心：

- 维护一个左闭右开区间 `[lo, hi)`。
- `lo` 是最大的不满足条件的值。
- `hi` 是最小的满足条件的值。
- 循环直到 `lo` 和 `hi` 相邻，`hi` 即为答案。

继续使用并熟悉这个模板，它会成为你解决二分问题的得力助手！

这个模板是上一个模板的 “镜像” 版本，也是非常常用的二分答案写法。它同样使用了**左闭右开**区间 `[lo, hi)`，但它求解的是**最大的满足条件的解**。

2. 模板解析

```
// 1. 初始化区间 [lo, hi)
// lo 通常设置为一个不可能的小值（小于所有可能的答案）
// hi 通常设置为一个不可能的大值（大于所有可能的答案）
i64 lo = -1, hi = 1E9 + 1;

// 2. 循环条件：只要区间长度大于 1，就继续二分
while (lo + 1 < hi) {
    // 3. 计算中间值 mid
    i64 mid = lo + (hi - lo) / 2;

    // 4. 核心判断：检查 mid 是否满足题目条件
    if (check(mid)) {
        // 4.1 如果满足条件：
        // 说明 mid 是一个可行解，但可能不是最大的可行解
        // 我们需要在 [mid, hi) 区间内继续寻找更大的可行解
        lo = mid;
    } else {
        // 4.2 如果不满足条件：
        // 说明 mid 不是一个可行解，且所有大于 mid 的值也不可能是可行解
        // 我们需要在 [lo, mid) 区间内继续寻找
        hi = mid;
    }
}

// 5. 循环结束后，lo 就是最大的满足条件的解
cout << lo << "\n";
```

核心思想

这个模板的核心思想同样是 **“排除法”**，但它寻找的是**最大的 满足条件的解。

- **当 `check(mid)` 为 `true` 时**：`mid` 是一个可行解。但因为我们要找的是**最大的**可行解，所以所有小于 `mid` 的值都可以被排除。因此，我们将 `lo` 移动到 `mid`，使得新的搜索区间变为 `[mid, hi)`。
- **当 `check(mid)` 为 `false` 时**：`mid` 不是一个可行解。并且，由于答案具有单调性，所有大于 `mid` 的值也都不可能是可行解。因此，我们将 `hi` 移动到 `mid`，使得新的搜索区间变为 `[lo, mid)`。

优点

和上一个模板一样，这个模板也具有以下优点：

1. **代码简洁**：不需要额外的 `ans` 变量，区间调整也不需要 `+1` 或 `-1`。
2. **逻辑清晰**：它清晰地将区间划分为 “满足条件的区域” 和 “不满足条件的区域”。`lo` 始终代表着 “最大的满足条件的值”，`hi` 始终代表着 “最小的不满足条件的值”。
3. **鲁棒性强**：边界处理简单，不容易出错。

适用场景

这个模板特别适合解决 **“寻找最大的满足条件的解”** 这类问题。例如：

- 寻找最后一个小于等于目标值的元素。
- 寻找最大的可行方案（如最大的距离、最大的承载量等）。

示例

让我们用一个例子来演示一下这个模板的工作原理。

问题：在一个有序数组 `[1, 3, 5, 7, 9]` 中，寻找最后一个小于等于 `6` 的元素。

- **初始区间**：`lo = -1`，`hi = 5`。区间 `[-1, 5)` 包含了所有可能的索引。
- **第一次循环**：`lo + 1 = 0 < hi = 5`，循环继续。
  - `mid = -1 + (5 - (-1)) / 2 = 2`。
  - `check(2)`：判断 `nums[2]` (值为 5) 是否小于等于 6？`true`。
  - 因为满足，所以 `lo = mid = 2`。新的区间是 `[2, 5)`。
- **第二次循环**：`lo + 1 = 3 < hi = 5`，循环继续。
  - `mid = 2 + (5 - 2) / 2 = 3`。
  - `check(3)`：判断 `nums[3]` (值为 7) 是否小于等于 6？`false`。
  - 因为不满足，所以 `hi = mid = 3`。新的区间是 `[2, 3)`。
- **第三次循环**：`lo + 1 = 3` 不小于 `hi = 3`，循环结束。
- **最终结果**：`lo = 2`，即数组中最后一个小于等于 6 的元素的索引是 2，对应的值是 5。

两个模板的对比

现在我们来总结一下这两个非常相似但用途不同的模板：

特性	模板一 (求最小解)	模板二 (求最大解)
目标	寻找**最小的**满足条件的解	寻找**最大的**满足条件的解
`check` 为 `true` 时	`hi = mid` (尝试寻找更小的解)	`lo = mid` (尝试寻找更大的解)
循环结束后输出	`hi`	`lo`
示例	第一个 >= 6 的元素	最后一个 <= 6 的元素

总结

你提供的这两个模板是二分答案算法中最核心、最常用的实现方式。它们都基于左闭右开区间 `[lo, hi)`，通过简洁的逻辑实现了高效的搜索。

- **模板一**：`if (check(mid)) hi = mid;` → 输出 `hi` → 求**最小解**。
- **模板二**：`if (check(mid)) lo = mid;` → 输出 `lo` → 求**最大解**。

掌握了这两个模板，并能根据题目要求灵活选择，你就可以解决绝大多数二分答案问题了。