二分查找类问题的通用写法

魔鬼在于细节，简单常用的二分查找的确有很多种写法，而且稍不留神就容易出错……

赞同高赞的一句话：有多少种写法不重要，重要的是要会写一种对的。事实上，这些不同的写法本质上都是一样的效果，只不过在不同的地方含着各自的 ± 1 或 $=$ 而已，通过变量代换可以相互转换。因此最重要的一点是，**确定自己一定正确的那一种写法。**

高赞

二分查找有几种写法？它们的区别是什么？ - Jason Li的回答 - 知乎 https://www.zhihu.com/question/36132386/answer/530313852

的回答很精彩。然而他选择了左闭右开 [first, last) 的搜索策略，尽管给出了一部分理由，但私以为并不足以让左闭右开策略"一统江湖"。因为

1. Dijkstra的题外话和二分搜索的语境并不相同；
2. 从最终效果上看，不论是计算mid还是最终确定的更新策略，其中只要做变量代换 left = first, right = last - 1，整理一下就自然得到也很优美的[left, right]策略；
3. 从思考逻辑上看， [first, last) 的搜索策略由于区间性质的不对称，在迭代步骤和问题对称（从求最小变为求最大）上都有不自然的地方。

   比如，更新的时候first = mid + 1，而last = mid。

   在求最大问题时扭转思考的过程就更多了……甚至作者直接推荐了找上界用互补。

另外，高赞答主只说了普通的二分搜索下的各种情况，然而还有一些问题虽然是使用二分搜索的方法，但并不是找某个value这种直白的表述。本文要做的是总结这类问题的一个通用写法框架。

说着说着写跑偏了。总之大概意思就是左闭右闭 [left, right] 也很好的！而且我还有推广！各位客官来看看我的写法吧！

伪代码：

先抛出最终结论的伪代码，稍候再慢慢解释 min/max 和 P 指什么：

```
//在[0, n - 1]上二分查找类问题的通用写法：

//公共部分初始化：
left = 0, right = n - 1;//注要点：此时[left, right]必须恰好是求解的范围。如果少了可能漏解，如果大了可能报错/出错。

//如果是 min mid 满足 P 的情况：
while (left <= right) {
    mid = left + (right - left)/2;
    //或者 mid = left + (right + 1 - left)/2;
    if (mid 满足 P)
        right = mid - 1;
    else
        left = mid + 1;
```

```
}
return left;

//如果是max mid 满足 P 的情况:
while (left <= right) {
    mid = left + (right - left)/2;
    //或者 mid = left + (right + 1 - left)/2;
    if (mid 满足 P)
        left = mid + 1;
    else
        right = mid - 1;
}
return right;

//公共部分结尾:
处理无解等边界情况 //无解的情况就是返回值不在区间[0, n - 1]中。对于返回 left 的情况就是 left = n; 对于返回 right 的情况就是 right = -1.
```

基本问题

首先先明确一些基本问题，熟悉闭区间写法的同学可以跳过，当然再读一遍温习一下也不错:

1. 循环条件

本文采用左闭右闭式的搜索策略，也就是对数组 [0, n - 1]，初始化 `left = 0, right = n - 1`。

因此在`while`语句中判断时，条件为 `while(left <= right)`。因为循环的判定条件从逻辑上讲，就是集合非空。那么在左闭右闭的写法下，这一逻辑就是 left <= right 。

2. 更新策略

左闭右闭式的写法，更新策略是 `left = mid + 1`与 `right = mid - 1`。这是因为如果采用`left = mid`这种保守的更新策略，那么循环有可能会死在 left = mid, right = left + 1这种情况下。因此双侧都“激进更新”是确保循环可以终止的稳健策略。

3.中点的计算

建议`mid = lower_mid = left + (right - left) /2`

或者`mid = upper_mid = left + (right + 1 - left) /2`

不建议`mid = lower_mid = (left + right) /2`

和`mid = upper_mid = (left + right + 1) /2`

原因是因为后面的两种写法有可能溢出。而至于mid选用向左还是向右取整，在我们选了如上所述的更新策略后，并不会影响算法的正确性，二者均可。

进阶问题

1. 抽象问题

适用二分搜索求解的问题，最简单的无非类似于“寻找唯一的val”，“寻找小于val的最大数”，“不小于val的最小数”，“第一个出现的val”，“最后一个出现的val”，“比val大的第一个数”......

不论怎么说，这些问题总可以抽象为

$$\begin{aligned} \min \text{或} \max \quad & mid \\ s.t. \quad & mid \text{ 满足 条件} P \end{aligned}$$

的形式。其中 条件P 用某个不等式表示。

> 举几个例子，假设序列是单调递增（不减）的。那么“寻找小于val的最大数” 就是：
>
> $$\begin{aligned} \max \quad & mid \\ s.t. \quad & mid < val \end{aligned}$$
>
> “第一个出现的val”就是“不小于val的最小数”，即：
>
> $$\begin{aligned} \min \quad & mid \\ s.t. \quad & mid \geq val \end{aligned}$$
>
> “比val大的第一个数”就是：
>
> $$\begin{aligned} \min \quad & mid \\ s.t. \quad & mid > val \end{aligned}$$
>
> 诸如此类……

在这一步我们可以明确问题，甚至可以解决除了单纯找数以外更复杂些的问题（参见练习题）。**更主要的是这一步可以让写代码的时候不需要在“==”的情况上纠结。**把条件P在逻辑上确定清楚之后，代码就是`if (mid 满足 P)`怎么怎么样，`else`怎么怎么样。

前面的都是小怪。剩下两个最重要的问题，就是

1. 当满足条件P的时候更新 left 还是 right ?
2. 最终输出的答案是什么?

2. 分析循环不变式

要解答这两个问题，我们必须回到循环不变式里考虑。在循环中到底什么是始终保持不变的?

注意到，不论`if (mid 满足 P)`之后究竟写谁，反正要么我们会更新 left ，要么我们会更新 right 。也就是说 left 和 right 始终是对“是否满足 P”这一性质的划分。

由于每次判断完是否满足P之后，left 和 right 都会跨越 mid，把 mid “甩在身后”，也就是说 **left 的左侧(不含 left)**与 **right 的右侧(不含 right)**会始终保持它的固定性质——其中**一个是必定满足P**，**另一个是必定不满足P**。而我们循环的区间[left, right]则是待定区间，即不确定是否满足P，需要继续循环。

而当循环结束时，必定 left = right + 1。也就是说**原来的序列[0, n-1]被划分为了[0, right]和[left, n-1]两部分。其中一部分满足P，另一部分不满足P。**

（以上解释如果放图就更好了，但我懒得画。）

这时候注意到了，既然我们是“二分法求解问题”，那么也就是说我们此前已经判断过了这个问题适用二分法，最后的答案必定应该出现在结果的“中间”。（就像前面的几个例子一样）因此，我们就简单得到了判断的方式——如果这个问题是

$$\begin{aligned} \min \quad & mid \\ s.t. \quad & mid \text{ 满足 条件} P \end{aligned}$$

那么[left, n-1]就是满足P的集合，最终答案就是 left；

同理如果这个问题是

$$\begin{aligned} \max \quad & mid \\ s.t. \quad & mid \text{ 满 足 条 件} P \end{aligned}$$

那么[0, right]就是满足P的部分，最终答案就是 right。

这就回答了输出结果的问题。

对于 “min类问题” 的情况，如果 mid 满足P，我们就会希望 mid 再小一点，也就是把 right 减小继续搜索，所以此时应该更新 right；同理对于 “max类问题” 的情况，如果 mid 满足P，我们就会希望 mid 再大一点，也就是把 left 增大继续搜索，所以此时应该更新 left。

这就回答了更新策略的问题。

因此，只要把问题看清（确定是在min还是max 和 确定最终要求的答案到底要满足什么条件）那么二分法的各种变体在循环条件上都不会再有问题了。

一道练习题：

题目：

LeetCode 275. H指数II

分析：

本题作为这个策略的应用例子，相当于在 最大化文献数量，在满足最低影响力不太低的条件下。而文章数量可以用 n - mid 表示， 那么 max n - mid 就相当于 min mid。其中 mid 要满足的条件P是 n - mid <= citations[mid]。如果想不清楚为什么是小于等于，有取巧的办法：首先相等的结果是可以的，所以有“=”；其次既然想min mid，也就说 mid 很大的时候一定成立，所以就是“<”。

C++代码：

```
class Solution {
public:
    int hIndex(vector<int>& citations) {
        int n = citations.size();
        int left = 0;
        int right = n - 1;
        int mid = n;//mid的初值并没用，我只是习惯性的选择了初始化为最弱情况 h = 0
        while (left <= right){//搜索闭区间[left, right]
            mid = left + (right-left)/2;//防止边界条件溢出的好习惯
            if ( n - mid <= citations[mid]){//满足性质P: 引用次数不少于论文数
                right = mid - 1;//min问题成立的情况更新right
            }
            else{//不满足性质P
                left = mid + 1;
            }
        }
        return n - left;//min问题返回答案left, 只不过本题要输出的结果是 n - min 也就是 n
- left
    }
};
```