二分查找高效判定子序列

相关推荐：

- twoSum问题的核心思想
- 经典动态规划：子集背包问题

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

392.判断子序列

二分查找本身不难理解，难在巧妙地运用二分查找技巧。对于一个问题，你可能都很难想到它跟二分查找有关，比如前文 最长递增子序列 就借助一个纸牌游戏衍生出二分查找解法。

今天再讲一道巧用二分查找的算法问题：如何判定字符串 `s` 是否是字符串 `t` 的子序列（可以假定 `s` 长度比较小，且 `t` 的长度非常大）。举两个例子：

s = "abc", t = "**a**h**b**gd**c**", return true.

s = "axc", t = "ahbgdc", return false.

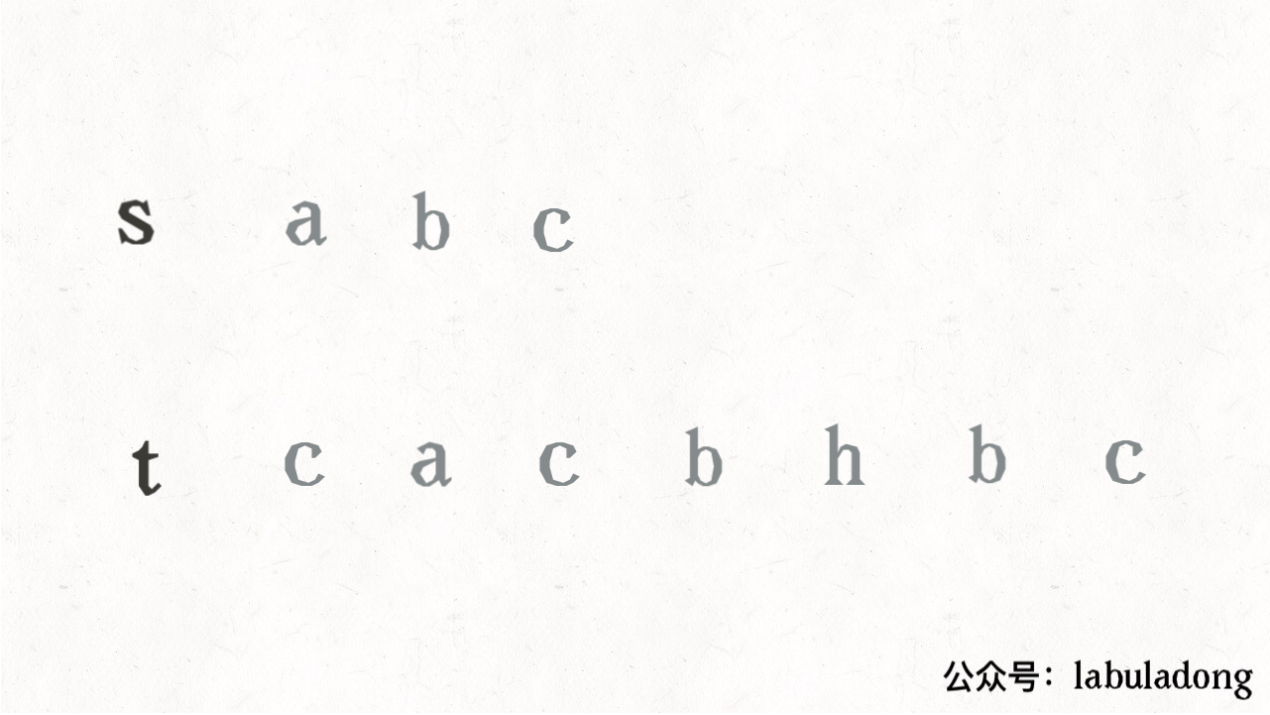
题目很容易理解，而且看起来很简单，但很难想到这个问题跟二分查找有关吧？

一、问题分析

首先，一个很简单的解法是这样的：

```
bool isSubsequence(string s, string t) {
    int i = 0, j = 0;
    while (i < s.size() && j < t.size()) {
        if (s[i] == t[j]) i++;
        j++;
    }
    return i == s.size();
}
```

其思路也非常简单，利用双指针 `i, j` 分别指向 `s, t`，一边前进一边匹配子序列：



读者也许会问，这不就是最优解法了吗，时间复杂度只需 O(N)，N 为 `t` 的长度。

是的，如果仅仅是这个问题，这个解法就够好了，**不过这个问题还有 follow up**：

如果给你一系列字符串 `s1,s2,...` 和字符串 `t`，你需要判定每个串 `s` 是否是 `t` 的子序列（可以假定 `s` 较短，`t` 很长）。

```
boolean[] isSubsequence(String[] sn, String t);
```

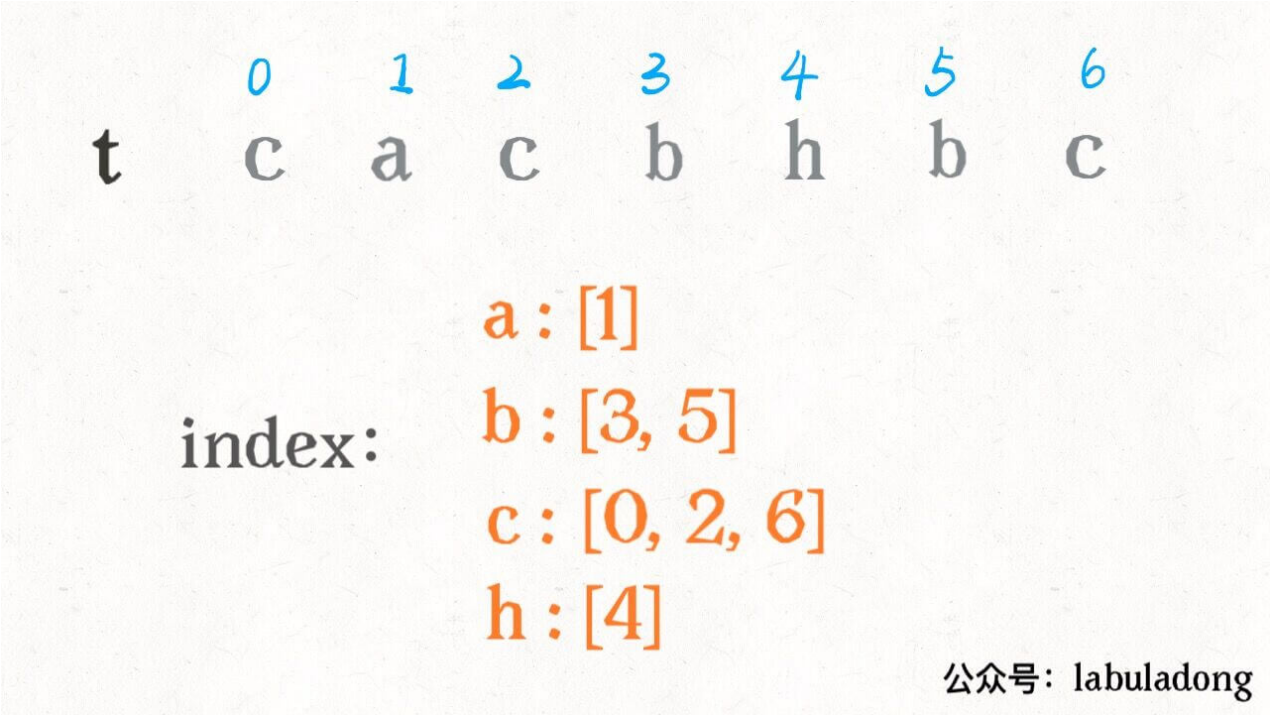
你也许会问，这不是很简单吗，还是刚才的逻辑，加个 for 循环不就行了？

可以，但是此解法处理每个 `s` 时间复杂度仍然是 O(N)，而如果巧妙运用二分查找，可以将时间复杂度降低，大约是 O(MlogN)。由于 N 相对 M 大很多，所以后者效率会更高。

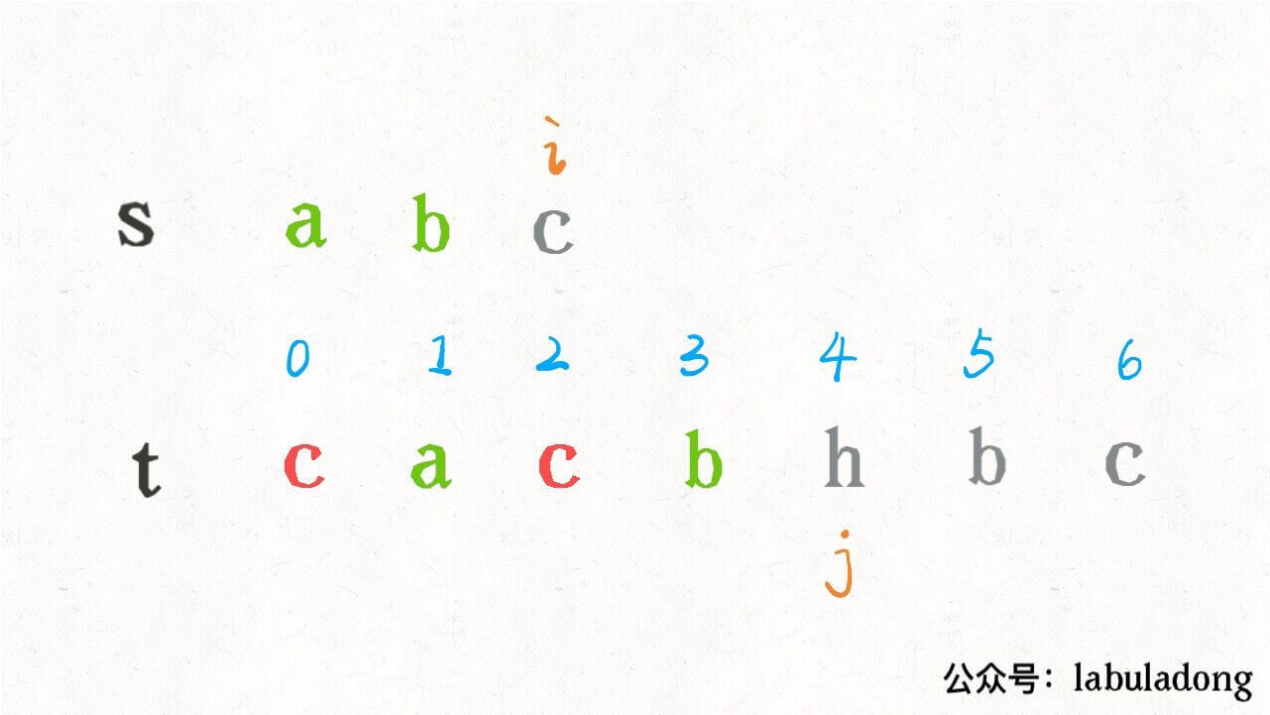
二、二分思路

二分思路主要是对 `t` 进行预处理，用一个字典 `index` 将每个字符出现的索引位置按顺序存储下来：

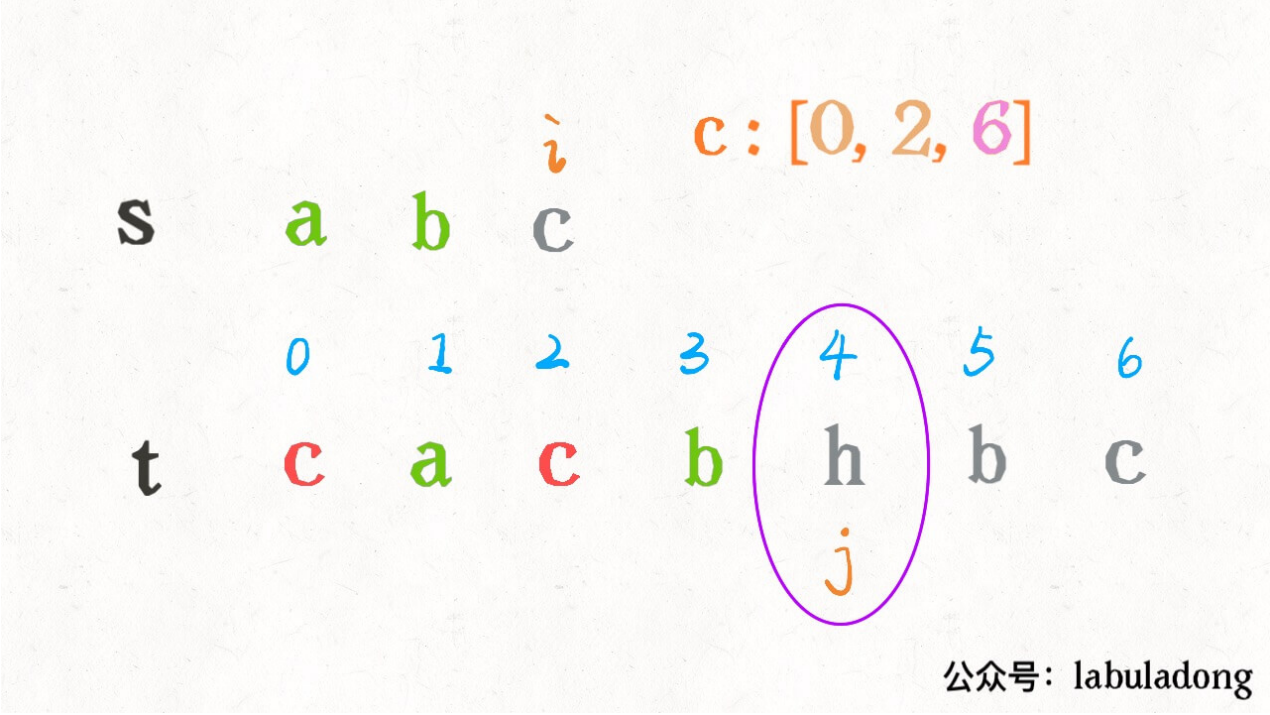
```
int m = s.length(), n = t.length();
ArrayList<Integer>[] index = new ArrayList[256];
// 先记下 t 中每个字符出现的位置
for (int i = 0; i < n; i++) {
    char c = t.charAt(i);
    if (index[c] == null)
        index[c] = new ArrayList<>();
    index[c].add(i);
}
```



比如对于这个情况，匹配了 "ab"，应该匹配 "c" 了：



按照之前的解法，我们需要 `j` 线性前进扫描字符 "c"，但借助 `index` 中记录的信息，**可以二分搜索 `index[c]` 中比 j 大的那个索引**，在上图的例子中，就是在 `[0,2,6]` 中搜索比 4 大的那个索引：



这样就可以直接得到下一个 "c" 的索引。现在的问题就是，如何用二分查找计算那个恰好比 4 大的索引呢？答案是，寻找左侧边界的二分搜索就可以做到。

三、再谈二分查找

在前文 二分查找详解 中，详解了如何正确写出三种二分查找算法的细节。二分查找返回目标值 `val` 的索引，对于搜索**左侧边界**的二分查找，有一个特殊性质：

当 `val` 不存在时，得到的索引恰好是比 `val` 大的最小元素索引。

什么意思呢，就是说如果在数组 `[0,1,3,4]` 中搜索元素 2，算法会返回索引 2，也就是元素 3 的位置，元素 3 是数组中大于 2 的最小元素。所以我们可以利用二分搜索避免线性扫描。

```
// 查找左侧边界的二分查找
int left_bound(ArrayList<Integer> arr, int tar) {
    int lo = 0, hi = arr.size();
    while (lo < hi) {
        int mid = lo + (hi - lo) / 2;
        if (tar > arr.get(mid)) {
            lo = mid + 1;
        } else {
            hi = mid;
        }
    }
    return lo;
}
```

以上就是搜索左侧边界的二分查找，等会儿会用到，其中的细节可以参见前文《二分查找详解》，这里不再赘述。

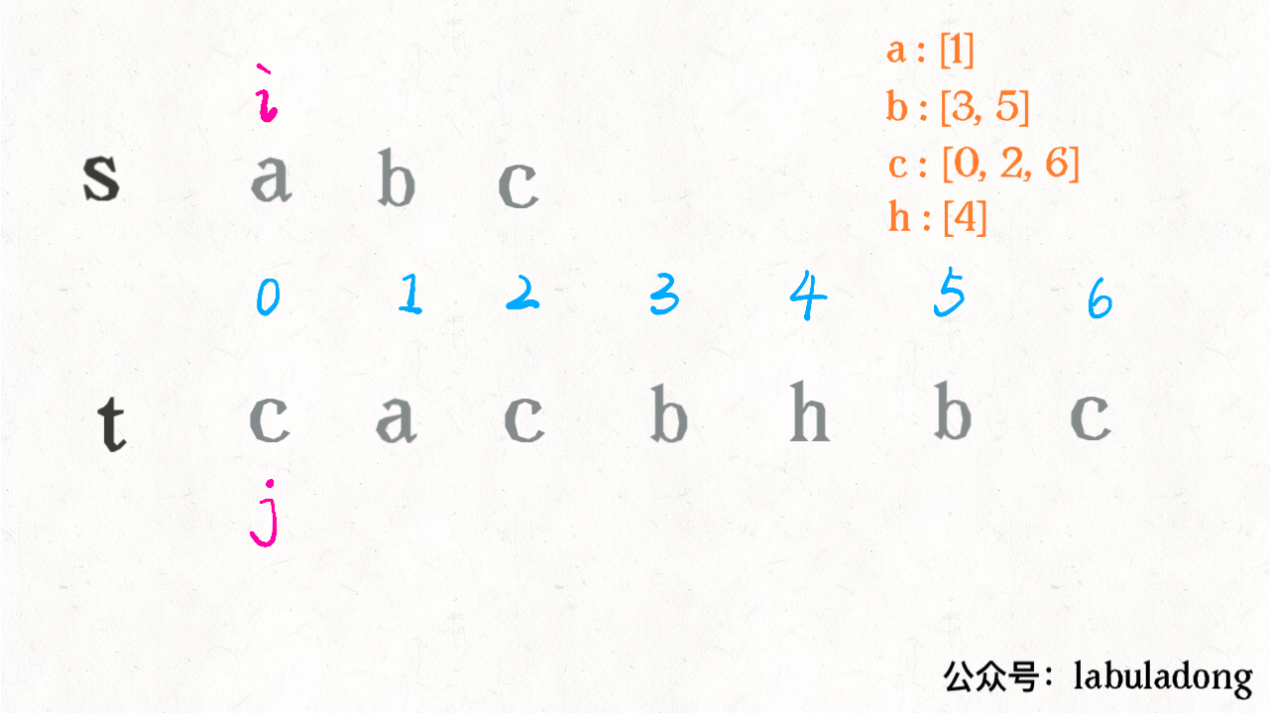
四、代码实现

这里以单个字符串 `s` 为例，对于多个字符串 `s`，可以把预处理部分抽出来。

```
boolean isSubsequence(String s, String t) {
    int m = s.length(), n = t.length();
    // 对 t 进行预处理
    ArrayList<Integer>[] index = new ArrayList[256];
    for (int i = 0; i < n; i++) {
        char c = t.charAt(i);
        if (index[c] == null)
            index[c] = new ArrayList<>();
        index[c].add(i);
    }

    // 串 t 上的指针
    int j = 0;
    // 借助 index 查找 s[i]
    for (int i = 0; i < m; i++) {
        char c = s.charAt(i);
        // 整个 t 压根儿没有字符 c
        if (index[c] == null) return false;
        int pos = left_bound(index[c], j);
        // 二分搜索区间中没有找到字符 c
        if (pos == index[c].size()) return false;
        // 向前移动指针 j
        j = index[c].get(pos) + 1;
    }
    return true;
}
```

算法执行的过程是这样的：



可见借助二分查找，算法的效率是可以大幅提升的。

刷算法，学套路，认准 labuladong，公众号和 在线电子书 持续更新最新文章。

本小抄即将出版，微信扫码关注公众号，后台回复「小抄」限时免费获取，回复「进群」可进刷题群一起刷题，带你搞定 LeetCode。

==其他语言代码==