实验四——文本索引

姓名：王越洋 学号：22009200894

1.实验背景与目的

编写一个构建大块文本索引的程序，然后进行快速搜索，来查找某个字符串在该文本中的出现位置。

2.实验内容

从程序 BoyerMoore.java 开始。 这段代码基本上提供了一个完整的解决方案，但为了使其正常工作，你必须进行一些小的更改，因为它们在许多细节上与此处指定的问题不同，并且因为缺少各种小的代码。 你可能需要编辑此代码，或从头开始编写自己的代码。 再次，必须仔细考虑“比较”功能。

3.代码实现

1. 算法概述

Boyer-Moore 算法是一种高效的字符串匹配算法，它通过利用模式串与文本之间的不同部分进行跳跃来减少比较次数。其核心思想是通过预处理模式串来实现跳跃，优化搜索过程。该实现使用了坏字符规则，但不包括强健后缀规则。

2. 算法实现

在这个实现中：

R 是字符集的基数，这里是 256，表示使用的是 ASCII 字符集。

right 数组是坏字符跳过数组，它存储了模式串中每个字符最后一次出现的位置。初始化时，如果某个字符不在模式串中，其值为 -1。

search 方法用来在给定文本中搜索模式串的所有完整匹配，返回匹配的次数。

main 方法处理输入，读取文本和查询文件，统计模式串在文本中的匹配次数并输出。

3. 算法流程

（1）预处理阶段：

创建 right 数组，记录模式串中每个字符最后出现的位置。

（2）搜索阶段：

从文本的左端开始匹配模式串，逐字符从右向左比较。如果遇到不匹配的字符，使用 right 数组来确定跳跃距离。

如果整个模式串匹配成功，则计数器加一，继续搜索下一个位置。

如果没有匹配，则返回 0。

4. 代码分析

```
1. // 构造方法，预处理模式串
2. public BoyerMoore(String pat) {
3.     this.R = 256;  // ASCII 字符集大小
4.     this.pat = pat;
5.     right = new int[R];
```

```
    for (int c = 0; c < R; c++) right[c] = -1;
    for (int j = 0; j < pat.length(); j++) right[pat.charAt(j)] = j;
}

// 在文本中搜索模式串的所有完整匹配
public int search(String txt) {
    int m = pat.length();
    int n = txt.length();
    int skip;
    int count = 0;

    for (int i = 0; i <= n - m; i += skip) {
        skip = 0;
        for (int j = m - 1; j >= 0; j--) {
            if (pat.charAt(j) != txt.charAt(i + j)) {
                skip = Math.max(1, j - right[txt.charAt(i + j)]);
                break;
            }
        }
        if (skip == 0) {
            count++;
            i++;  // 查找下一个位置
        }
    }

    return count;
}
```

比较功能：

（1）字符逐一比较

在 Boyer-Moore 算法中，比较是通过从模式串的右端开始，与文本中当前位置的字符进行逐一比较的方式来实现的。具体步骤如下：

假设模式串 pat 的长度是 m，文本 txt 的长度是 n，要在 txt 中查找 pat 的出现位置。

算法从文本 txt 中的每个可能位置（即从 i = 0 到 i = n - m）开始，逐个字符与模式串 pat 进行比较。

比较从模式串的最右侧字符（即 pat[m-1]）开始，逐步向左移动，直到匹配或不匹配为止。

```
for (int j = m - 1; j >= 0; j--) {
    if (pat.charAt(j) != txt.charAt(i + j)) {
        skip = Math.max(1, j - right[txt.charAt(i + j)]);
        break;
    }
```

```
6. }
```

- pat.charAt(j) 是模式串的当前字符。
- txt.charAt(i + j) 是文本中的字符。
- 当遇到不匹配时，skip 计算跳过的字符数，通过坏字符规则确定下一个可能的匹配位置。

（2）坏字符规则

坏字符规则是 Boyer-Moore 算法中最核心的“比较”功能，基于模式串中每个字符的最右出现位置来跳过一些不必要的字符。这个规则大大提高了算法的效率，避免了对每个字符的逐个比较。

右移规则： 当某个字符在模式串中没有匹配时，算法将模式串向右移动，跳过所有不可能匹配的位置。具体来说，若模式串中的字符在文本中没有匹配到，则通过坏字符规则（即当前字符在模式串中的最右位置）计算出**跳跃的距离**。

```
skip = Math.max(1, j - right[txt.charAt(i + j)]);
```

在这个过程中：

right[txt.charAt(i + j)] 是文本中当前字符在模式串中的最右位置。

如果当前字符不存在于模式串中，则 right[txt.charAt(i + j)] 会为 -1。

skip 是计算出的跳跃步数，保证模式串能够跳过不必要的匹配位置

（3）模式串匹配

在每一次字符比较时，从模式串的右端开始向左端比较字符，直到找出模式串的完整匹配或者发现某个字符不匹配。

如果所有字符匹配，则模式串完全匹配文本的一部分，算法返回匹配的起始位置。

如果发现不匹配，则根据坏字符规则计算跳跃步数，跳过已经匹配过的字符。

（4）核心逻辑

比较功能的关键在于通过跳跃减少不必要的比较，使得在每次不匹配时，能够尽量跳过尽可能多的字符。

字符逐一比较：从模式串的右侧字符开始，逐个比较文本中的字符。

坏字符规则：当发现不匹配时，计算跳跃步数，跳过不必要的字符。

高效跳跃：通过最右位置的坏字符规则来跳过文本中的多个字符，从而避免逐个字符的比较，显著提高效率。

4. 结果分析

输出结果如图所示：

```
2 (Boyer-Moore) falling Time: 14 μs
31 (Boyer-Moore) March Hare Time: 12 μs
1 (Boyer-Moore) miserable Mock Turtle Time: 7
31 (Boyer-Moore) moment Time: 12 μs
134 (Boyer-Moore) own Time: 10 μs
87 (Boyer-Moore) went Time: 8 μs
-- (Boyer-Moore) ??? Time: 9 μs
Boyer-Moore average time: 1060 μs
Brute Force average time: 1504 μs
```

5.总结

在本次实验中，我实现了基于 Boyer-Moore 算法的字符串匹配程序，并通过与其他算法进行对比，分析了该算法在实际应用中的性能优势与不足。以下是本次实验的几个关键收获与总结：

1. Boyer-Moore 算法的原理与应用

Boyer-Moore 算法是一种非常高效的字符串匹配算法，特别适用于模式串较短而文本较长的情况。其核心思想基于两个规则：

坏字符规则：当模式串与文本的字符不匹配时，根据不匹配字符在模式串中最后出现的位置来决定跳跃的距离，从而避免了模式串和文本中已经匹配的部分重新比较。

好后缀规则（未实现）：如果出现了部分匹配的后缀，可以利用模式串中相同后缀的位置信息来跳过不必要的比较，进一步提高匹配效率。

在实验中，我只实现了坏字符规则，通过预处理模式串来生成一个数组记录每个字符在模式串中最后一次出现的位置。然后，在实际匹配过程中，利用这个信息决定跳过多少个字符，减少无效的比较。

2. 性能比较与优化

在实验过程中，我将 Boyer-Moore 算法与暴力匹配算法进行了对比。暴力匹配算法的时间复杂度为 O(mn)，其中 m 是模式串的长度，n 是文本的长度。暴力算法每次都从文本的每个字符开始逐一比较，效率较低。而 Boyer-Moore 算法通过利用坏字符规则的跳跃，能够在某些情况下将匹配时间大幅减少，尤其是在模式串与文本不匹配的部分时，能够跳过大量字符比较。

我通过分析实验结果，发现 Boyer-Moore 算法的性能在处理大规模文本时展现出了明显的优势，尤其是当模式串较短且文本较长时。相比暴力匹配算法，Boyer-Moore 能够大幅度减少字符比较次数，从而提高整体匹配效率。

3. 处理大规模文本的挑战与解决方案

在实验中，我使用了一个相对较大的文本文件进行测试，并处理了多个查询模式串。面对大规模文本时，如何保证匹配的效率是一个挑战。通过 Boyer-Moore 算法的优化策略，能够有效地减少不必要的字符比较，显著提高了查询速度。

尽管 Boyer-Moore 算法在很多情况下表现出色，但仍然存在一些局限性。对于某些特殊模式串（如模式串中有大量重复字符的情况），该算法可能无法充分发挥其跳跃的优势，仍然会进行较多的比较。因此，对于不同类型的文本和模式串，选择合适的算法非常重要。

4. 实验中的问题与思考

输入格式处理：如何从文件中正确读取文本和查询模式串，并确保格式一致性，避免读取错误。

5. 未来的改进

引入好后缀规则：进一步完善 Boyer-Moore 算法，结合好后缀规则，使得在大规模文本中进行模式串匹配时，效率能够得到进一步提升。

并行处理：对于非常大的文本文件，可以考虑使用并行算法来加速字符串匹配过程，例如使用多线程将文本分段处理，进而加速查询过程。

与其他算法对比：除了 Boyer-Moore，还可以尝试实现并比较其他字符串匹配算法，如 KMP 算法、Rabin-Karp 算法等，进一步优化匹配效率。