Solution to Problem 5.3.11

Problem Construct a worst-case example for the Boyer-Moore algorithm, demonstrating that it is not linear-time in the worst case.

—

Background on the Boyer-Moore Algorithm

The Boyer-Moore string matching algorithm is typically efficient, leveraging two main heuristics:

1. **Bad Character Rule**: On a mismatch, the algorithm shifts the pattern based on the position of the mismatched character.
2. **Good Suffix Rule**: On a mismatch, the pattern is shifted based on suffix information to maximize the shift.

Although the algorithm often achieves linear-time performance, specific input patterns can degrade its performance to $O(m \cdot n)$, where m is the pattern length and n is the text length.

—

Worst-Case Input Construction

A worst-case input can be constructed as follows:

- **Text**: A long string of repeating characters.
- **Pattern**: A pattern that closely matches the text but has a mismatched character at the end.

—

Example

- **Text**: `AAAAAAAAAAAAAAAAAA`
- **Pattern**: `AAAAAB`

—

Explanation of the Worst Case

1. **Bad Character Rule**:
   - The mismatch occurs at the last character of the pattern (`B` vs `A`).
   - The bad character rule shifts the pattern by only 1 because the mismatched character (`B`) does not appear in the text.
2. **Good Suffix Rule**:

- The good suffix rule provides no additional benefit because the mismatch occurs at the end of the pattern.
- This causes the pattern to shift by only 1 position repeatedly.

3. **Performance**:
   - Every character of the text is compared with every character of the pattern, leading to $O(m \cdot n)$ complexity.

Code Demonstration

Below is a Java implementation demonstrating the behavior of the Boyer-Moore algorithm for the worst-case input.

```java
// Java Implementation of Boyer-Moore Algorithm
public class BoyerMooreWorstCase {
    public static void main(String[] args) {
        String text = "AAAAAAAAAAAAAAAAAA";
        String pattern = "AAAAAB";

        BoyerMoore bm = new BoyerMoore(pattern);
        int position = bm.search(text);

        System.out.println("Pattern found at position: " +
            position);
    }
}

class BoyerMoore {
    private final int R = 256; // Number of characters in
        the alphabet
    private int[] right;
    private String pat;

    public BoyerMoore(String pat) {
        this.pat = pat;
        right = new int[R];
        for (int c = 0; c < R; c++) {
            right[c] = -1; // Initialize all characters to
                -1
        }
        for (int j = 0; j < pat.length(); j++) {
            right[pat.charAt(j)] = j; // Rightmost position
                of each character
        }
    }

    public int search(String txt) {
```

```
        int n = txt.length();
        int m = pat.length();
        int skip;
        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) return i; // Pattern found
        }
        return n; // Pattern not found
    }
}
```

—

Analysis of the Example

1. The mismatch at `B` causes the pattern to shift by only 1 position.
2. Both the bad character rule and the good suffix rule fail to optimize the shifts.
3. The algorithm performs $O(m \cdot n)$ comparisons in this scenario.

—

Summary

The Boyer-Moore algorithm degrades to $O(m \cdot n)$ in cases where:

- The text contains repeating characters.
- The pattern almost matches the text but mismatches at the last character.

For the example:

- **Text**: `AAAAAAAAAAAAAAAAA`
- **Pattern**: `AAAAAB`

This demonstrates that the Boyer-Moore algorithm is not always linear-time.