Homework 9: Pattern Matching

● Graded

Student

Vidit Dharmendra Pokharna

Total Points

150 / 100 pts

Autograder Score

150.0 / 100.0

Question 2

Feedback & Manual Grading

0 / 0 pts

✓ **+ 0 pts** Correct

Flawless job, keep it up for the final stretch of the semester!
- Emily W □□ ^ □ ^ □□✿

Autograder Results

Autograder Output

If you're seeing this message, everything compiled and ran properly!
-CS1332 TAs

Submitted Files

PatternMatching.java

Download

```java
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * Your implementations of various string searching algorithms.
 *
 * @author Vidit Pokharna
 * @version 1.0
 * @userid vpokharna3
 * @GTID 903772087
 *
 * Collaborators: LIST ALL COLLABORATORS YOU WORKED WITH HERE
 *
 * Resources: LIST ALL NON-COURSE RESOURCES YOU CONSULTED HERE
 */
public class PatternMatching {

    /**
     * Knuth-Morris-Pratt (KMP) algorithm relies on the failure table (also
     * called failure function). Works better with small alphabets.
     *
     * Make sure to implement the buildFailureTable() method before implementing
     * this method.
     *
     * @param pattern    the pattern you are searching for in a body of text
     * @param text       the body of text where you search for pattern
     * @param comparator you MUST use this to check if characters are equal
     * @return list containing the starting index for each match found
     * @throws java.lang.IllegalArgumentException if the pattern is null or has
     *                                            length 0
     * @throws java.lang.IllegalArgumentException if text or comparator is null
     */
    public static List<Integer> kmp(CharSequence pattern, CharSequence text,
                                    CharacterComparator comparator) {
        if (pattern == null || pattern.length() == 0) {
            throw new IllegalArgumentException("The pattern given is invalid");
        } else if (text == null) {
            throw new IllegalArgumentException("The text given is invalid");
        } else if (comparator == null) {
            throw new IllegalArgumentException("The comparator given is invalid");
        }

        int m = pattern.length();
        int n = text.length();

        List<Integer> result = new ArrayList<>();
        if (m > n) {
            return result;
        }

        int[] failureTable = buildFailureTable(pattern, comparator);

        int i = 0;
        int j = 0;

        while (i <= n - m) {

            while (j < pattern.length() && comparator.compare(text.charAt(i + j), pattern.charAt(j)) == 0) {
                j++;
            }

            if (j == 0) {
                i++;
            } else {
                if (j == pattern.length()) {
                    result.add(i);
                }

                i = i + j - failureTable[j - 1];
                j = failureTable[j - 1];
            }

        }

        return result;
    }

    /**
     * Builds failure table that will be used to run the Knuth-Morris-Pratt
     * (KMP) algorithm.
     *
     * The table built should be the length of the input pattern.
     *
     * Note that a given index i will contain the length of the largest prefix
     * of the pattern indices [0..i] that is also a suffix of the pattern
     * indices [1..i]. This means that index 0 of the returned table will always
     * be equal to 0
     *
     * Ex.
     * pattern:       a  b  a  b  a  c
     * failureTable: [0, 0, 1, 2, 3, 0]
     *
     * If the pattern is empty, return an empty array.
     *
     * @param pattern    a pattern you're building a failure table for
     * @param comparator you MUST use this to check if characters are equal
     * @return integer array holding your failure table
     * @throws java.lang.IllegalArgumentException if the pattern or comparator
     *                                            is null
     */
    public static int[] buildFailureTable(CharSequence pattern,
                                          CharacterComparator comparator) {
        if (pattern == null) {
            throw new IllegalArgumentException("The pattern given is invalid");
        } else if (comparator == null) {
            throw new IllegalArgumentException("The comparator given is invalid");
        }

        int m = pattern.length();
        int[] failureTable = new int[m];

        if (m == 0) {
            return failureTable;
        }

        int i = 0;
        int j = 1;
        failureTable[0] = 0;

        while (j < m) {
            if (comparator.compare(pattern.charAt(i), pattern.charAt(j)) == 0) {
                failureTable[j] = i + 1;
                i++;
                j++;
            } else {
                if (i == 0) {
                    failureTable[j] = 0;
                    j++;
                } else {
                    i = failureTable[i - 1];
                }
            }
        }

        return failureTable;
    }

    /**
     * Boyer Moore algorithm that relies on last occurrence table. Works better
     * with large alphabets.
     *
     * Make sure to implement the buildLastTable() method before implementing
```

```java
     * this method. Do NOT implement the Galil Rule in this method.
     *
     * Note: You may find the getOrDefault() method from Java's Map class
     * useful.
     *
     * @param pattern    the pattern you are searching for in a body of text
     * @param text       the body of text where you search for the pattern
     * @param comparator you MUST use this to check if characters are equal
     * @return list containing the starting index for each match found
     * @throws java.lang.IllegalArgumentException if the pattern is null or has
     *                                            length 0
     * @throws java.lang.IllegalArgumentException if text or comparator is null
     */
    public static List<Integer> boyerMoore(CharSequence pattern,
                                           CharSequence text,
                                           CharacterComparator comparator) {
        if (pattern == null || pattern.length() == 0) {
            throw new IllegalArgumentException("The pattern given is invalid");
        } else if (text == null) {
            throw new IllegalArgumentException("The text given is invalid");
        } else if (comparator == null) {
            throw new IllegalArgumentException("The comparator given is invalid");
        }

        int m = pattern.length();
        int n = text.length();

        List<Integer> result = new ArrayList<>();
        if (m > n) {
            return result;
        }

        Map<Character, Integer> lastTable = buildLastTable(pattern);

        int i = 0;

        while (i <= n - m) {
            int j = m - 1;
            while ((j >= 0) && (comparator.compare(pattern.charAt(j), text.charAt(i + j)) == 0)) {
                j--;
            }

            if (j == -1) {
                result.add(i);
                i++;
            } else {
                int shift = 0;
                if (lastTable.get(text.charAt(i + j)) != null) {
                    shift = lastTable.get(text.charAt(i + j));
                } else {
                    shift = -1;
                }

                if (shift < j) {
                    i = i + j - shift;
                } else {
                    i++;
                }
            }
        }

        return result;
    }

    /**
     * Builds last occurrence table that will be used to run the Boyer Moore
     * algorithm.
     *
     * Note that each char x will have an entry at table.get(x).
     * Each entry should be the last index of x where x is a particular
     * character in your pattern.
     * If x is not in the pattern, then the table will not contain the key x,
     * and you will have to check for that in your Boyer Moore implementation.
     *
     * Ex. pattern = octocat
     *
     * table.get(o) = 3
     * table.get(c) = 4
     * table.get(t) = 6
     * table.get(a) = 5
     * table.get(everything else) = null, which you will interpret in
     * Boyer-Moore as -1
     *
     * If the pattern is empty, return an empty map.
     *
     * @param pattern a pattern you are building last table for
     * @return a Map with keys of all of the characters in the pattern mapping
     * to their last occurrence in the pattern
     * @throws java.lang.IllegalArgumentException if the pattern is null
     */
    public static Map<Character, Integer> buildLastTable(CharSequence pattern) {
        if (pattern == null) {
            throw new IllegalArgumentException("The pattern given is invalid");
        }

        Map<Character, Integer> lastTable = new HashMap<>();

        int m = pattern.length();
        for (int a = 0; a < m; a++) {
            lastTable.put(pattern.charAt(a), a);
        }

        return lastTable;
    }

    /**
     * Prime base used for Rabin-Karp hashing.
     * DO NOT EDIT!
     */
    private static final int BASE = 113;

    /**
     * Runs the Rabin-Karp algorithm. This algorithms generates hashes for the
     * pattern and compares this hash to substrings of the text before doing
     * character by character comparisons.
     *
     * When the hashes are equal and you do character comparisons, compare
     * starting from the beginning of the pattern to the end, not from the end
     * to the beginning.
     *
     * You must use the Rabin-Karp Rolling Hash for this implementation. The
     * formula for it is:
     *
     * sum of: c * BASE ^ (pattern.length - 1 - i)
     *   c is the integer value of the current character, and
     *   i is the index of the character
     *
     * We recommend building the hash for the pattern and the first m characters
     * of the text by starting at index (m - 1) to efficiently exponentiate the
     * BASE. This allows you to avoid using Math.pow().
     *
     * Note that if you were dealing with very large numbers here, your hash
     * will likely overflow; you will not need to handle this case.
     * You may assume that all powers and calculations CAN be done without
     * overflow. However, be careful with how you carry out your calculations.
     * For example, if BASE^(m - 1) is a number that fits into an int, it's
     * possible for BASE^m will overflow. So, you would not want to do
     * BASE^m / BASE to calculate BASE^(m - 1).
     *
     * Ex. Hashing "bunn" as a substring of "bunny" with base 113
     * = (b * 113 ^ 3) + (u * 113 ^ 2) + (n * 113 ^ 1) + (n * 113 ^ 0)
     * = (98 * 113 ^ 3) + (117 * 113 ^ 2) + (110 * 113 ^ 1) + (110 * 113 ^ 0)
     * = 142910419
     *
     * Another key point of this algorithm is that updating the hash from
     * one substring to the next substring must be O(1). To update the hash,
     * subtract the oldChar times BASE raised to the length - 1, multiply by
     * BASE, and add the newChar as shown by this formula:
     * (oldHash - oldChar * BASE ^ (pattern.length - 1)) * BASE + newChar
     *
     * Ex. Shifting from "bunn" to "unny" in "bunny" with base 113
     * hash("unny") = (hash("bunn") - b * 113 ^ 3) * 113 + y
     *              = (142910419 - 98 * 113 ^ 3) * 113 + 121
     *              = 170236090
     *
     * Keep in mind that calculating exponents is not O(1) in general, so you'll
     * need to keep track of what BASE^(m - 1) is for updating the hash.
     *
     * Do NOT use Math.pow() in this method.
     * Do NOT implement your own version of Math.pow().
     *
     * @param pattern    a string you're searching for in a body of text
     * @param text       the body of text where you search for pattern
     * @param comparator you MUST use this to check if characters are equal
     * @return list containing the starting index for each match found
     * @throws java.lang.IllegalArgumentException if the pattern is null or has
     *                                            length 0
     * @throws java.lang.IllegalArgumentException if text or comparator is null
     */
    public static List<Integer> rabinKarp(CharSequence pattern,
                                          CharSequence text,
                                          CharacterComparator comparator) {
        if (pattern == null || pattern.length() == 0) {
            throw new IllegalArgumentException("The pattern given is invalid");
        } else if (text == null) {
            throw new IllegalArgumentException("The text given is invalid");
        } else if (comparator == null) {
            throw new IllegalArgumentException("The comparator given is invalid");
        }

        int m = pattern.length();
        int n = text.length();

        List<Integer> result = new ArrayList<>();
        if (m > n) {
            return result;
        }

        int patternHash = 0;
        int textHash = 0;
        int multiplier = 1;
        for (int i = pattern.length() - 1; i >= 0; i--) {
            patternHash += pattern.charAt(i) * multiplier;
            textHash += text.charAt(i) * multiplier;
            if (i > 0) {
                multiplier *= BASE;
```

```java
            }
        }

        int i = 0;
        while (i <= n - m) {
            if (patternHash == textHash) {
                int j = 0;
                while (j < pattern.length() && comparator.compare(pattern.charAt(j), text.charAt(i + j)) == 0) {
                    j++;
                }

                if (j == m) {
                    result.add(i);
                }
            }

            i++;
            if (i <= n - m) {
                textHash =
                        (textHash - text.charAt(i - 1) * multiplier) * BASE + text.charAt(i + pattern.length() - 1);

            }
        }

        return result;
    }

    /**
     * This method is OPTIONAL and for extra credit only.
     *
     * The Galil Rule is an addition to Boyer Moore that optimizes how we shift the pattern
     * after a full match. Please read Extra Credit: Galil Rule section in the homework pdf for details.
     *
     * Make sure to implement the buildLastTable() method and buildFailureTable() method
     * before implementing this method.
     *
     * @param pattern    the pattern you are searching for in a body of text
     * @param text       the body of text where you search for the pattern
     * @param comparator you MUST use this to check if characters are equal
     * @return list containing the starting index for each match found
     * @throws java.lang.IllegalArgumentException if the pattern is null or has
     *                                            length 0
     * @throws java.lang.IllegalArgumentException if text or comparator is null
     */
    public static List<Integer> boyerMooreGalilRule(CharSequence pattern,
                                                    CharSequence text,
                                                    CharacterComparator comparator) {
        if (pattern == null || pattern.length() == 0) {
            throw new IllegalArgumentException("The pattern given is invalid");
        } else if (text == null) {
            throw new IllegalArgumentException("The text given is invalid");
        } else if (comparator == null) {
            throw new IllegalArgumentException("The comparator given is invalid");
        }

        int m = pattern.length();
        int n = text.length();

        List<Integer> result = new ArrayList<>();
        if (m > n) {
            return result;
        }

        int periodK = m - buildFailureTable(pattern, comparator)[m - 1];
        Map<Character, Integer> lastTable = buildLastTable(pattern);

        int i = 0;
        int w = 0;

        while (i <= n - m) {
            int j = m - 1;

            while (j >= w && comparator.compare(pattern.charAt(j), text.charAt(i + j)) == 0) {
                j--;
            }

            if (j < w) {
                result.add(i);
                w = m - periodK;
                i += periodK;
            } else {
                w = 0;

                int shift = 0;
                if (lastTable.get(text.charAt(i + j)) != null) {
                    shift = lastTable.get(text.charAt(i + j));
                } else {
                    shift = -1;
                }

                if (shift < j) {
                    i = i + j - shift;
                } else {
                    i++;
                }
            }
        }

        return result;
    }
}

```