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();
```

```
47
48          List<Integer> result = new ArrayList<>();
49          if (m > n) {
50              return result;
51          }
52
53          int[] failureTable = buildFailureTable(pattern, comparator);
54
55          int i = 0;
56          int j = 0;
57
58          while (i <= n - m) {
59
60              while (j < pattern.length() && comparator.compare(text.charAt(i + j), pattern.charAt(j)) == 0) {
61                  j++;
62              }
63
64              if (j == 0) {
65                  i++;
66              } else {
67                  if (j == pattern.length()) {
68                      result.add(i);
69                  }
70
71                  i = i + j - failureTable[j - 1];
72                  j = failureTable[j - 1];
73              }
74
75          }
76
77          return result;
78      }
79
80      /**
81       * Builds failure table that will be used to run the Knuth-Morris-Pratt
82       * (KMP) algorithm.
83       *
84       * The table built should be the length of the input pattern.
85       *
86       * Note that a given index i will contain the length of the largest prefix
87       * of the pattern indices [0..i] that is also a suffix of the pattern
88       * indices [1..i]. This means that index 0 of the returned table will always
89       * be equal to 0
90       *
91       * Ex.
92       * pattern:      a  b  a  b  a  c
93       * failureTable: [0, 0, 1, 2, 3, 0]
94       *
95       * If the pattern is empty, return an empty array.
```

```
96      *
97      * @param pattern    a pattern you're building a failure table for
98      * @param comparator you MUST use this to check if characters are equal
99      * @return integer array holding your failure table
100     * @throws java.lang.IllegalArgumentException if the pattern or comparator
101     *                                  is null
102     */
103    public static int[] buildFailureTable(CharSequence pattern,
104                                          CharacterComparator comparator) {
105        if (pattern == null) {
106            throw new IllegalArgumentException("The pattern given is invalid");
107        } else if (comparator == null) {
108            throw new IllegalArgumentException("The comparator given is invalid");
109        }
110
111        int m = pattern.length();
112        int[] failureTable = new int[m];
113
114        if (m == 0) {
115            return failureTable;
116        }
117
118        int i = 0;
119        int j = 1;
120        failureTable[0] = 0;
121
122        while (j < m) {
123            if (comparator.compare(pattern.charAt(i), pattern.charAt(j)) == 0) {
124                failureTable[j] = i + 1;
125                i++;
126                j++;
127            } else {
128                if (i == 0) {
129                    failureTable[j] = 0;
130                    j++;
131                } else {
132                    i = failureTable[i - 1];
133                }
134            }
135        }
136
137        return failureTable;
138    }
139
140    /**
141     * Boyer Moore algorithm that relies on last occurrence table. Works better
142     * with large alphabets.
143     *
144     * Make sure to implement the buildLastTable() method before implementing
```

```
145     * this method. Do NOT implement the Galil Rule in this method.
146     *
147     * Note: You may find the getOrDefault() method from Java's Map class
148     * useful.
149     *
150     * @param pattern    the pattern you are searching for in a body of text
151     * @param text       the body of text where you search for the pattern
152     * @param comparator you MUST use this to check if characters are equal
153     * @return list containing the starting index for each match found
154     * @throws java.lang.IllegalArgumentException if the pattern is null or has
155     *                                            length 0
156     * @throws java.lang.IllegalArgumentException if text or comparator is null
157     */
158    public static List<Integer> boyerMoore(CharSequence pattern,
159                                           CharSequence text,
160                                           CharacterComparator comparator) {
161        if (pattern == null || pattern.length() == 0) {
162            throw new IllegalArgumentException("The pattern given is invalid");
163        } else if (text == null) {
164            throw new IllegalArgumentException("The text given is invalid");
165        } else if (comparator == null) {
166            throw new IllegalArgumentException("The comparator given is invalid");
167        }
168
169        int m = pattern.length();
170        int n = text.length();
171
172        List<Integer> result = new ArrayList<>();
173        if (m > n) {
174            return result;
175        }
176
177        Map<Character, Integer> lastTable = buildLastTable(pattern);
178
179        int i = 0;
180
181        while (i <= n - m) {
182            int j = m - 1;
183            while ((j >= 0) && (comparator.compare(pattern.charAt(j), text.charAt(i + j)) == 0)) {
184                j--;
185            }
186
187            if (j == -1) {
188                result.add(i);
189                i++;
190            } else {
191                int shift = 0;
192                if (lastTable.get(text.charAt(i + j)) != null) {
193                    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();
```

```
243         for (int a = 0; a < m; a++) {
244             lastTable.put(pattern.charAt(a), a);
245         }
246 
247         return lastTable;
248     }
249 
250     /**
251      * Prime base used for Rabin-Karp hashing.
252      * DO NOT EDIT!
253      */
254     private static final int BASE = 113;
255 
256     /**
257      * Runs the Rabin-Karp algorithm. This algorithms generates hashes for the
258      * pattern and compares this hash to substrings of the text before doing
259      * character by character comparisons.
260      *
261      * When the hashes are equal and you do character comparisons, compare
262      * starting from the beginning of the pattern to the end, not from the end
263      * to the beginning.
264      *
265      * You must use the Rabin-Karp Rolling Hash for this implementation. The
266      * formula for it is:
267      *
268      * sum of: c * BASE ^ (pattern.length - 1 - i)
269      *   c is the integer value of the current character, and
270      *   i is the index of the character
271      *
272      * We recommend building the hash for the pattern and the first m characters
273      * of the text by starting at index (m - 1) to efficiently exponentiate the
274      * BASE. This allows you to avoid using Math.pow().
275      *
276      * Note that if you were dealing with very large numbers here, your hash
277      * will likely overflow; you will not need to handle this case.
278      * You may assume that all powers and calculations CAN be done without
279      * overflow. However, be careful with how you carry out your calculations.
280      * For example, if BASE^(m - 1) is a number that fits into an int, it's
281      * possible for BASE^m will overflow. So, you would not want to do
282      * BASE^m / BASE to calculate BASE^(m - 1).
283      *
284      * Ex. Hashing "bunn" as a substring of "bunny" with base 113
285      * = (b * 113 ^ 3) + (u * 113 ^ 2) + (n * 113 ^ 1) + (n * 113 ^ 0)
286      * = (98 * 113 ^ 3) + (117 * 113 ^ 2) + (110 * 113 ^ 1) + (110 * 113 ^ 0)
287      * = 142910419
288      *
289      * Another key point of this algorithm is that updating the hash from
290      * one substring to the next substring must be O(1). To update the hash,
291      * subtract the oldChar times BASE raised to the length - 1, multiply by
```

```
292     * BASE, and add the newChar as shown by this formula:
293     * (oldHash - oldChar * BASE ^ (pattern.length - 1)) * BASE + newChar
294     *
295     * Ex. Shifting from "bunn" to "unny" in "bunny" with base 113
296     * hash("unny") = (hash("bunn") - b * 113 ^ 3) * 113 + y
297     *              = (142910419 - 98 * 113 ^ 3) * 113 + 121
298     *              = 170236090
299     *
300     * Keep in mind that calculating exponents is not O(1) in general, so you'll
301     * need to keep track of what BASE^(m - 1) is for updating the hash.
302     *
303     * Do NOT use Math.pow() in this method.
304     * Do NOT implement your own version of Math.pow().
305     *
306     * @param pattern    a string you're searching for in a body of text
307     * @param text       the body of text where you search for pattern
308     * @param comparator you MUST use this to check if characters are equal
309     * @return list containing the starting index for each match found
310     * @throws java.lang.IllegalArgumentException if the pattern is null or has
311     *                                            length 0
312     * @throws java.lang.IllegalArgumentException if text or comparator is null
313     */
314    public static List<Integer> rabinKarp(CharSequence pattern,
315                                          CharSequence text,
316                                          CharacterComparator comparator) {
317        if (pattern == null || pattern.length() == 0) {
318            throw new IllegalArgumentException("The pattern given is invalid");
319        } else if (text == null) {
320            throw new IllegalArgumentException("The text given is invalid");
321        } else if (comparator == null) {
322            throw new IllegalArgumentException("The comparator given is invalid");
323        }
324
325        int m = pattern.length();
326        int n = text.length();
327
328        List<Integer> result = new ArrayList<>();
329        if (m > n) {
330            return result;
331        }
332
333        int patternHash = 0;
334        int textHash = 0;
335        int multiplier = 1;
336        for (int i = pattern.length() - 1; i >= 0; i--) {
337            patternHash += pattern.charAt(i) * multiplier;
338            textHash += text.charAt(i) * multiplier;
339            if (i > 0) {
340                multiplier *= BASE;
```

```
341             }
342         }
343 
344         int i = 0;
345         while (i <= n - m) {
346             if (patternHash == textHash) {
347                 int j = 0;
348                 while (j < pattern.length() && comparator.compare(pattern.charAt(j), text.charAt(i + j)) == 0) {
349                     j++;
350                 }
351 
352                 if (j == m) {
353                     result.add(i);
354                 }
355             }
356 
357             i++;
358             if (i <= n - m) {
359                 textHash =
360                         (textHash - text.charAt(i - 1) * multiplier) * BASE + text.charAt(i + pattern.length() - 1);
361 
362             }
363         }
364 
365         return result;
366     }
367 
368     /**
369      * This method is OPTIONAL and for extra credit only.
370      *
371      * The Galil Rule is an addition to Boyer Moore that optimizes how we shift the pattern
372      * after a full match. Please read Extra Credit: Galil Rule section in the homework pdf for details.
373      *
374      * Make sure to implement the buildLastTable() method and buildFailureTable() method
375      * before implementing this method.
376      *
377      * @param pattern    the pattern you are searching for in a body of text
378      * @param text       the body of text where you search for the pattern
379      * @param comparator you MUST use this to check if characters are equal
380      * @return list containing the starting index for each match found
381      * @throws java.lang.IllegalArgumentException if the pattern is null or has
382      *                                            length 0
383      * @throws java.lang.IllegalArgumentException if text or comparator is null
384      */
385     public static List<Integer> boyerMooreGalilRule(CharSequence pattern,
386                                                     CharSequence text,
387                                                     CharacterComparator comparator) {
388         if (pattern == null || pattern.length() == 0) {
389             throw new IllegalArgumentException("The pattern given is invalid");
```

```
390         } else if (text == null) {
391             throw new IllegalArgumentException("The text given is invalid");
392         } else if (comparator == null) {
393             throw new IllegalArgumentException("The comparator given is invalid");
394         }
395 
396         int m = pattern.length();
397         int n = text.length();
398 
399         List<Integer> result = new ArrayList<>();
400         if (m > n) {
401             return result;
402         }
403 
404         int periodK = m - buildFailureTable(pattern, comparator)[m - 1];
405         Map<Character, Integer> lastTable = buildLastTable(pattern);
406 
407         int i = 0;
408         int w = 0;
409 
410         while (i <= n - m) {
411             int j = m - 1;
412 
413             while (j >= w && comparator.compare(pattern.charAt(j), text.charAt(i + j)) == 0) {
414                 j--;
415             }
416 
417             if (j < w) {
418                 result.add(i);
419                 w = m - periodK;
420                 i += periodK;
421             } else {
422                 w = 0;
423 
424                 int shift = 0;
425                 if (lastTable.get(text.charAt(i + j)) != null) {
426                     shift = lastTable.get(text.charAt(i + j));
427                 } else {
428                     shift = -1;
429                 }
430 
431                 if (shift < j) {
432                     i = i + j - shift;
433                 } else {
434                     i++;
435                 }
436             }
437         }
438 
```

```
439         return result;
440     }
441 }
442
```