```java
public class Main {
    public static void main(String[] args) {
        test_suites();
    }


    private static void test_suites() {
        n_test_suite();
        m_test_suite();
        alphabet_test_suite();
    }

    //Below are the three primary test suites. Included with them are comments discussing what we can do with the data
    //IMPORTANT TERMINOLOGY: N = text size. M = pattern size.

    /**
     * Hypothesis: different n sizes will cause variation in time and # of comparisons when holding M and alphabet constant
     * Additional questions to ask based on data collected from this test suite:
     *      For both search algorithms, what happens when we have a larger N? smaller N?
     *      For both search algorithms, what is the relationship as N increases while holding M and alphabet constant?
     *          i.e., does it change on a O(N) scale? O(logN)?
     *      Is one particular search algorithm more sensitive than the other one to text size? Which one do you want to use with a larger/smaller N?
     *          Furthermore, does one search algorithm work better until N reaches a large enough size?
     *      When answering these questions, don't look at exact values. Look at changes between values, and difference between
     *      numbers based on the algorithm (as number variations will occur due to different Ms and Alphabets)
     * Testing:
     *      We run tests with several large N values (to reduce noise) AND small N values (to answer one of the above questions)
     *      We also try three different M values and alphabets (for 9 total combinations), to make sure that the data is credible
     *          for the large N tests, the M value is large to reduce noise; for the small N tests, M value is also small
     */
    private static void n_test_suite() {
        System.out.println("BEGINNING N TEST SUITE");
        System.out.println("----------------------------------------");
        final int trial_count = 100;
        final int[] large_m_values = {(int)Math.pow(2,6), (int)Math.pow(2,10), (int)Math.pow(2,13)};
        final String[] alphabets = {"abcdefghijklmnopqrstuvwxyz", "01", "actg"};
        System.out.println("Comparison testing for 9 (M,Alphabet) pairs for varying LARGE N's");
        System.out.println("----------------------------------------");
        for(int m: large_m_values) {
            for(String a: alphabets) {
                comparisons_vary_n(m,a,trial_count, true);
            }
        }
        final int[] small_m_values = {2, (int)Math.pow(2,3), (int)Math.pow(2,4)};
```

```
45          System.out.println("Comparison testing for 9 (M,Alphabet) pairs for
    varying SMALL N's");
46          System.out.println("-----------------------------------------");
47          for(int m: small_m_values) {
48              for(String a: alphabets) {
49                  comparisons_vary_n(m,a,trial_count, false);
50              }
51          }
52          System.out.println("Time testing for 9 (M,Alphabet) pairs for varying
    LARGE N's");
53          System.out.println("-----------------------------------------");
54          for(int m: large_m_values) {
55              for(String a: alphabets) {
56                  time_vary_n(m,a,trial_count, true);
57              }
58          }
59          System.out.println("Time testing for 9 (M,Alphabet) pairs for varying
    SMALL N's");
60          System.out.println("-----------------------------------------");
61          for(int m: small_m_values) {
62              for(String a: alphabets) {
63                  time_vary_n(m,a,trial_count, false);
64              }
65          }
66      }
67
68      /**
69       * Hypothesis: different m sizes will cause variation in time and # of
    comparisons when holding N and alphabet constant
70       * Additional questions to ask based on data collected from this test
    suite:
71       *      For both search algorithms, what happens when we have a larger M?
    smaller M?
72       *      For both search algorithms, what is the relationship as M
    increases while holding N and alphabet constant?
73       *          i.e., does it change on a O(N) scale? O(logN)?
74       *      Is one particular search algorithm more sensitive than the other
    one to pattern size? Which one do you want to use with a larger/smaller N?
75       *          Furthermore, does one search algorithm work better until M
    reaches a large enough size?
76       *      When answering these questions, don't look at exact values. Look
    at changes between values, and difference between
77       *      numbers based on the algorithm (as number variations will occur
    due to different N and Alphabets)
78       * Testing:
79       *      We run tests with several large M values (to reduce noise) AND
    small M values (to answer one of the above questions)
80       *      We also try three different N values and alphabets (for 9 total
    combinations), to make sure that the data is credible
81       *          for the large M tests, the N value is large to reduce noise;
    for the small M tests, N value is also small
82       */
83      private static void m_test_suite() {
84          System.out.println("BEGINNING M TEST SUITE");
85          System.out.println("-----------------------------------------");
86          final int trial_count = 100;
87          final int[] large_n_values = {(int)Math.pow(2,14),
    (int)Math.pow(2,18), (int)Math.pow(2,22)};
88          final String[] alphabets = {"abcdefghijklmnopqrstuvwxyz", "01",
    "actg"};
```

```
89          System.out.println("Comparison testing for 9 (N,Alphabet) pairs for
    varying LARGE M's");
90          System.out.println("-----------------------------------------");
91          for(int n: large_n_values) {
92              for(String a: alphabets) {
93                  comparisons_vary_m(n,a,trial_count, true);
94              }
95          }
96          final int[] small_n_values = {(int)Math.pow(2,6), (int)Math.pow(2,9),
    (int)Math.pow(2,12)};
97          System.out.println("Comparison testing for 9 (N,Alphabet) pairs for
    varying SMALL M's");
98          System.out.println("-----------------------------------------");
99          for(int n: small_n_values) {
100             for(String a: alphabets) {
101                 comparisons_vary_m(n,a,trial_count, false);
102             }
103         }
104         System.out.println("Time testing for 9 (N,Alphabet) pairs for varying
    LARGE M's");
105         System.out.println("-----------------------------------------");
106         for(int n: large_n_values) {
107             for(String a: alphabets) {
108                 time_vary_m(n,a,trial_count, true);
109             }
110         }
111         System.out.println("Time testing for 9 (N,Alphabet) pairs for varying
    SMALL M's");
112         System.out.println("-----------------------------------------");
113         for(int n: small_n_values) {
114             for(String a: alphabets) {
115                 time_vary_m(n,a,trial_count, false);
116             }
117         }
118     }
119
120     /**
121      * Hypothesis: different alphabet sizes will cause variations in time and
    # of comparisons when holding N and M constant
122      * Additional questions to ask based on data collected from this test
    suite:
123      *      For both search algorithms, what happens when we have a larger
    alphabet? smaller alphabet?
124      *      For both search algorithms, what is the relationship as alphabet
    size increases while holding M and N constant?
125      *          i.e., does it change on a O(N) scale? O(logN)?
126      *      Is one particular search algorithm more sensitive than the other
    one to alphabet? Which one do we want to use in what case?
127      *      When answering these questions, don't look at exact values. Look
    at changes between values, and difference between
128      *      numbers based on the algorithm (as number variations will occur
    due to different N and Ms)
129      * Testing:
130      *      We run tests with several alphabets of varying sizes (the
    alphabets will be in the code)
131      *      We also try three different N and M values (for 9 total
    combinations), to make sure that the data is credible
132      *          we use large values for N and M to reduce noise
133      */
134     private static void alphabet_test_suite() {
```

```
135         System.out.println("BEGINNING ALPHABET TEST SUITE");
136         System.out.println("----------------------------------------");
137         final int trial_count = 100;
138         final int[] n_values = {(int)Math.pow(2,14), (int)Math.pow(2,18),
(int)Math.pow(2,22)};
139         final int[] m_values = {(int)Math.pow(2,6), (int)Math.pow(2,10),
(int)Math.pow(2,13)};
140         System.out.println("Comparison testing for 9 (N,M) pairs for varying
alphabets");
141         System.out.println("----------------------------------------");
142         for(int n:n_values) {
143             for(int m:m_values) {
144                 comparisons_vary_alphabet(n,m,trial_count);
145             }
146         }
147         System.out.println("Time testing for 9 (N,M) pairs for varying
alphabets");
148         System.out.println("----------------------------------------");
149         for(int n:n_values) {
150             for(int m:m_values) {
151                 time_vary_alphabet(n,m,trial_count);
152             }
153         }
154     }
155     /*
156         What we want to test (aka values that we can vary):
157             M (pattern size)
158             N (text size)
159             Alphabet
160         **only vary one at a time
161 */
162 
163     /**
164      * test suite for variable m (pattern length)
165      * @param n N value (text size)
166      * @param alphabet the alphabet used for text and pattern
167      * @param trials the number of trials to be run
168      */
169     private static void comparisons_vary_m(int n, String alphabet, int
trials, boolean isLarge) {
170         final int[] m_values = new int[(isLarge ? 8 : 6)];
171         for(int i=(isLarge ? 7 : 1); i<=((isLarge ? 14 : 6)); i++)
m_values[i-(isLarge ? 7 : 1)] = (int)Math.pow(2,i);
172         System.out.println("CONSTANTS: N = " + n + ", alphabet = " + alphabet
+ "");
173         System.out.println("M\tKMP\tNaive");
174         final String[] random_texts = initializeRandomStrings(n, trials,
alphabet);
175         for(int i=0;i<m_values.length&&m_values[i]<n;i++) {
176             final String[] random_patterns =
initializeRandomStrings(m_values[i], trials, alphabet);
177             System.out.println(m_values[i] + "\t" +
kmp_tests(random_patterns, random_texts) + "\t" +
naive_tests(random_patterns, random_texts));
178         }
179         System.out.println("----------------------------------------");
180     }
181 
182     /**
183      * test suite for variable n (text length)
```

```
184      * @param m M value (pattern size)
185      * @param alphabet the alphabet used for text and pattern
186      * @param trials the number of trials to be run
187      */
188     private static void comparisons_vary_n(int m, String alphabet, int
trials, boolean isLarge) {
189         final int[] n_values = new int[(isLarge ? 9 : 7)];
190         for(int i=isLarge ? 14 : 5; i<=(isLarge ? 22 : 11); i++) n_values[i-
(isLarge ? 14 : 5)] = (int)Math.pow(2,i);
191         System.out.println("CONSTANTS: M = " + m + ", alphabet = " + alphabet
+ "");
192         System.out.println("N\tKMP\tNaive");
193         final String[] random_patterns = initializeRandomStrings(m, trials,
alphabet);
194         for(int i=0;i<n_values.length&&n_values[i]>m;i++) {
195             final String[] random_texts =
initializeRandomStrings(n_values[i], trials, alphabet);
196             System.out.println(n_values[i] + "\t" +
kmp_tests(random_patterns, random_texts) + "\t" +
naive_tests(random_patterns, random_texts));
197         }
198         System.out.println("----------------------------------------");
199     }
200
201     /**
202      * test suite for variable alphabet
203      * @param n N value (text size)
204      * @param m M value (pattern size)
205      * @param trials the number of trials to be run
206      */
207     private static void comparisons_vary_alphabet(int n, int m, int trials) {
208         final String[] alphabets = {"abcdefghijklmnopqrstuvwxyz", "01",
"actg", "0123456789", "abcdefghijklmnopqrstuvwxyz0123456789!@#$%^&*()`~,.
<>/?;:[{]}-_=+|"};
209         System.out.println("CONSTANTS: N = " + n + ", M = " + m + "");
210         System.out.println("A size\tKMP\tNaive");
211         for(String a:alphabets) {
212             final String[] random_patterns = initializeRandomStrings(m,
trials, a);
213             final String[] random_texts = initializeRandomStrings(n, trials,
a);
214             System.out.println(a.length() + "\t" + kmp_tests(random_patterns,
random_texts) + "\t" + naive_tests(random_patterns, random_texts));
215         }
216         System.out.println("----------------------------------------");
217     }
218
219     /**
220      * test suite for variable m (pattern length)
221      * @param n N value (text size)
222      * @param alphabet the alphabet used for text and pattern
223      * @param trials the number of trials to be run
224      */
225     private static void time_vary_m(int n, String alphabet, int trials,
boolean isLarge) {
226         final int[] m_values = new int[(isLarge ? 8 : 6)];
227         for(int i=(isLarge ? 7 : 1); i<=((isLarge ? 14 : 6)); i++)
m_values[i-(isLarge ? 7 : 1)] = (int)Math.pow(2,i);
228         System.out.println("CONSTANTS: N = " + n + ", alphabet = " + alphabet
+ "");
```

```
229         System.out.println("M\tKMP\tNaive");
230         final String[] random_texts = initializeRandomStrings(n, trials,
alphabet);
231         for(int i=0;i<m_values.length&&m_values[i]<n;i++) {
232            final String[] random_patterns =
initializeRandomStrings(m_values[i], trials, alphabet);
233             System.out.println(m_values[i] + "\t" +
kmp_test_time(random_patterns, random_texts) + "\t" +
naive_test_time(random_patterns, random_texts));
234         }
235         System.out.println("----------------------------------------");
236     }
237 
238     /**
239      * test suite for variable n (text length)
240      * @param m M value (pattern size)
241      * @param alphabet the alphabet used for text and pattern
242      * @param trials the number of trials to be run
243      */
244     private static void time_vary_n(int m, String alphabet, int trials,
boolean isLarge) {
245         final int[] n_values = new int[(isLarge ? 9 : 7)];
246         for(int i=isLarge ? 14 : 5; i<=(isLarge ? 22 : 11); i++) n_values[i-
(isLarge ? 14 : 5)] = (int)Math.pow(2,i);
247         System.out.println("CONSTANTS: M = " + m + ", alphabet = " + alphabet
+ "");
248         System.out.println("N\tKMP\tNaive");
249         final String[] random_patterns = initializeRandomStrings(m, trials,
alphabet);
250         for(int i=0;i<n_values.length&&n_values[i]>m;i++) {
251             final String[] random_texts =
initializeRandomStrings(n_values[i], trials, alphabet);
252             System.out.println(n_values[i] + "\t" +
kmp_test_time(random_patterns, random_texts) + "\t" +
naive_test_time(random_patterns, random_texts));
253         }
254         System.out.println("----------------------------------------");
255     }
256 
257     /**
258      * test suite for variable alphabet
259      * @param n N value (text size)
260      * @param m M value (pattern size)
261      * @param trials the number of trials to be run
262      */
263     private static void time_vary_alphabet(int n, int m, int trials) {
264         final String[] alphabets = {"abcdefghijklmnopqrstuvwxyz", "01",
"actg", "0123456789", "abcdefghijklmnopqrstuvwxyz0123456789!@#$%^&*()`~,.
<>/?;:[{]}-_=+|"};
265         System.out.println("CONSTANTS: N = " + n + ", M = " + m + "");
266         System.out.println("A Size\tKMP\tNaive");
267         for(String a:alphabets) {
268             final String[] random_patterns = initializeRandomStrings(m,
trials, a);
269             final String[] random_texts = initializeRandomStrings(n, trials,
a);
270             System.out.println(a.length() + "\t" +
kmp_test_time(random_patterns, random_texts) + "\t" +
naive_test_time(random_patterns, random_texts));
271         }
```

```
272         System.out.println("-----------------------------------------");
273     }
274 
275     //the below 1 functions are necessary so that I can run the exact same
    test strings for both searches to
276     //make our data as accurate as possible.
277     /**
278      * creates a list of random strings of size n, with the list being of
    length size.
279      * @param n size of strings
280      * @param size size of list
281      * @param alphabet the alphabet to use for the random strings
282      * @return the list
283      */
284     private static String[] initializeRandomStrings(int n, int size, String
    alphabet) {
285         String[] strs = new String[size];
286         for(int i=0; i<size; i++) {
287             strs[i] = Randomizer.numRandomizer(n, alphabet);
288         }
289         return strs;
290     }
291 
292 
293     /**
294      * Gets average number of comparisons KMP uses for a given text_length
    (n) and alphabet
295      * @param patterns  list of randomly generated patterns
296      * @param texts list of randomly generated texts
297      * @return average number of comparisons of KMP
298      */
299     private static double kmp_tests(String[] patterns, String[] texts) {
300         double sum = 0;
301         final int trial_kount = patterns.length;
302         for(int i=0; i<trial_kount; i++) {
303             String pattern = patterns[i];
304             String text = texts[i];
305             final KMP kmp = new KMP(pattern);
306             kmp.search(text);
307             sum+= kmp.NUM_COMPARISONS;
308         }
309         return sum/trial_kount;
310     }
311 
312     /**
313      * Gets average number of comparisons Naive implementation uses for a
    given text_length (n) and alphabet
314      * @param patterns  list of randomly generated patterns
315      * @param texts list of randomly generated texts
316      * @return average number of comparisons of Naive implementation
317      */
318     private static double naive_tests(String[] patterns, String[] texts) {
319         double sum = 0;
320         for(int i=0; i<patterns.length; i++) {
321             NaiveSearch.search(texts[i], patterns[i]);
322             sum+=NaiveSearch.numComp;
323         }
324         return sum/patterns.length;
325     }
326 
```

```
    /**
     * testing time, for kmp search algorithm
     * @param patterns list of random patterns
     * @param texts list of random texts
     * @return average running time of a trial, in milliseconds
     */
    private static double kmp_test_time(String[] patterns, String[] texts) {
        double sum = 0;
        for(int i=0; i<patterns.length; i++) {
            String pattern = patterns[i], text = texts[i];
            final long startTime = System.nanoTime();
            final KMP kmp = new KMP(pattern);
            kmp.search(text);
            final long endTime = System.nanoTime();
            sum += (endTime-startTime);
        }
        final int avg = (int)(sum/patterns.length);
        final double avg_temp = avg/1000000.0;
        return avg_temp;
    }

    /**
     * testing time, for naive search algorithm
     * @param patterns list of random patterns
     * @param texts list of random texts
     * @return average running time of a trial, in milliseconds
     */
    private static double naive_test_time(String[] patterns, String[] texts)
{
        double sum = 0;
        for(int i=0; i<patterns.length; i++) {
            final long startTime = System.nanoTime();
            NaiveSearch.search(texts[i], patterns[i]);
            final long endTime = System.nanoTime();
            sum += endTime-startTime;
        }
        final int avg = (int)(sum/patterns.length);
        final double avg_temp = avg/1000000.0;
        return avg_temp;
    }

}

```