COSC 320 – 001
Analysis of Algorithms
2020 Winter Term 2

Project Topic Number: #4

String Matching Problem - Milestone 1

Group Lead:

Matthew Borle - 42615435

Group Members:

Barry Feng - 12981156

Guy Kaminsky - 65234775

Ross Morrison - 36963411

Abstract.

We started this milestone by organizing everyone in the group into Github teams. This feature will allow us to prepare to work on code together and have a place to store all of the documentation. Secondly, we were able to formulate all three string matching algorithms into algorithmic problems. Based on the problem formulation, we then created pseudocode for each of the three algorithms. With the pseudo-code complete, we analyzed the algorithms. In particular, we provided the running time, and proof of correctness for each algorithm. Lastly, we started gathering resources to create a dataset to test the algorithms in the future.

Problem Formulation.

Master document A of character length a is compared to a set of documents B each of character length b. Each document is compared using LCSS then KMP and Rabin-Karp.

The LCSS comparison will compare by paragraph. A and B are split into n_{AP} and n_{BP} paragraphs of varying length. These paragraphs are compared for matches. This takes $n_{AP}{}^{nBP}$ comparisons to complete.

The KMP comparison will compare by sentence. A and B are split into n_{AS} and n_{BS} sentences of varying length. These sentences are compared for matches. This takes $n_{AS}{}^{nBS}$ comparisons to complete.

The Rabin-Karp comparison will compare by word. A and B are split into n_{AW} and n_{BW} words of varying length. These words are compared for matches. This takes $n_{AW}{}^{nBW}$ comparisons to complete.

All three algorithms have similar descriptions but will take drastically different amounts of time to complete. Rabin-Karp takes the most time. For that reason it is performed last as any paragraph or sentence match from the previous algorithms will likely be enough to flag the document for plagiarism.

Pseudo-code - LCSS, KMP, Rabin-Karp Algorithms

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2      return
3  if b[i, j] == "↖"
4      PRINT-LCS(b, X, i − 1, j − 1)
5      print x_i
6  elseif b[i, j] == "↑"
7      PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

```
KMP-MATCHER(T, P)
 1  n = T.length
 2  m = P.length
 3  π = COMPUTE-PREFIX-FUNCTION(P)
 4  q = 0                                  // number of characters matched
 5  for i = 1 to n                         // scan the text from left to right
 6      while q > 0 and P[q + 1] ≠ T[i]
 7          q = π[q]                       // next character does not match
 8      if P[q + 1] == T[i]
 9          q = q + 1                      // next character matches
10      if q == m                          // is all of P matched?
11          print "Pattern occurs with shift" i − m
12          q = π[q]                       // look for the next match
```

```
RABIN-KARP-MATCHER(T, P, d, q)
 1  n = T.length
 2  m = P.length
 3  h = d^(m−1) mod q
 4  p = 0
 5  t_0 = 0
 6  for i = 1 to m                         // preprocessing
 7      p = (dp + P[i]) mod q
 8      t_0 = (dt_0 + T[i]) mod q
 9  for s = 0 to n − m                     // matching
10      if p == t_s
11          if P[1..m] == T[s + 1..s + m]
12              print "Pattern occurs with shift" s
13      if s < n − m
14          t_(s+1) = (d(t_s − T[s + 1]h) + T[s + m + 1]) mod q
```

Source: Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (n.d.). Introduction to algorithms. An *Introduction to algorithms* (3rd ed.).

Algorithm Analysis.

RABIN-KARP - Worst Case: O(nm), average: O($n + m$)

https://algoparc.ics.hawaii.edu/~nodari/teaching/s15/Notes/Topic-23.html

Algorithm Complexity.

Preprocessing is Θ(m): the first loop executes m times and the work in the loop is O(1).

Worst case matching time is still Θ($(n - m + 1)m$), like brute force, because every hash code possibly matches and therefore every brute force comparison has to be done. This is highly unlikely in any realistic application. Otherwise, the running time of the algorithm is O(nm).

KMP - Worst Case: O(n)

https://www.ics.uci.edu/~eppstein/161/960227.html

Algorithm Complexity.

Search running time does not include the time to generate the Longest Prefix suffix (LPS) array. The word pointer i will only be moved backwards up to i times. This means it can only go as far backwards as it has gone forwards, making the running time 2n. At most you will loop through each branch in the **for** loop twice.

LCSS - Worst case: O(nm)

Algorithm Complexity.

The time complexity would be O(nm) because each sub-problem calculation takes O(1) time, and there are mn subproblems. The space complexity of this algorithm follows to be O(nm) as we take nm space. Additional optimizations make it possible to reduce the space complexity to O(min(n, m)) by only storing the two rows we are comparing in the dynamic programming table.

Unexpected Cases/Difficulties.

The first unexpected difficulty began with the problem formulation as the instructions seemed unclear. Next, it was difficult to analyze the algorithms. In particular, all of these are public algorithms with detailed information about them. In this case it is less about us analyzing the algorithms and more about making sure our reasoning was correct. A technical issue with this project was getting the pseudocode to fit within the documentation page limit. With three lengthy algorithms, we had to get creative in making the pseudocode fit in the document. We had to use an editing software in order to make them fit on one page. The most difficult task came in identifying the differences between all three algorithms. Since these are complicated algorithms it took time to find out the method each algorithm uses to compare strings.

Task Separation and Responsibilities.

In terms of task separation, we decided to implement peer programming techniques. Specifically, Matt Borle and Barry Feng started the first project milestone by formulating the string matching problem into mathematical notation. Next, Guy Kaminsky and Ross Morrison created pseudocode for all three algorithms. Afterward, Barry Feng and Ross Morrison were able to finish up the analysis of algorithms tasks. Lastly, Matt Borle and Guy Kaminsky started on data collection for the project.