Jon Rippe  
CSCE A365  
HW #3 Writeup  
December 4, 2020

String Matching

# Code Implementation

The provided test.c code was crudely modified to run all three algorithms in succession. Testing was done using a Python script that created and then executed test.exe several times with the created parameters. The C code implemented is a direct translation of the pseudo-code in the textbook (Cormen). Corresponding code lines are as follows:

## Naive Algorithm

|  |  |  |
| --- | --- | --- |
| C Implementation | Pseudo-code | Notes |
|  |  | Brought in through args |
|  | Brought in through args |
| for (int s = 0; s < n - m + 1; s++) { |  | Add +1 since we’re using <. |
| res[s] = 1; |  | Setting this to *1* makes it easy to tell if a match is still possible. |
| for (int t = 0; t < m && res[s] == 1; t++) {  if (haystack[s + t] != needle[t]) { |  | Reversed the logic. Use *res[s]* to see if we should continue check. |
| res[s] = 0; |  | Determined pattern does NOT occur. Breaks current *for* loop. *res[s]* remains *1* if string matched. |

## Rabin-Karp Algorithm

|  |  |  |
| --- | --- | --- |
| C Implementation | Pseudo-code | Notes |
|  |  | Brought in through args |
|  |  |
| uint64\_t p = 0, t = 0; |  |  |
|  |  |
| uint64\_t h = 1;  for (uint64\_t i = 0; i < m - 1; i++) {  h = mulModuloBase(h);  } |  | Use multiplication to get power. |
| for (uint64\_t i = 0; i < m; i++) { |  | Offset by 1 since we start at 0 |
| p = (addModuloChar(mulModuloBase(p), needle[i])); |  |  |
| t = (addModuloChar(mulModuloBase(t), haystack[i]));  } |  |  |
| for (uint64\_t s = 0; s < n - m + 1; s++) { |  | Add +1 since we’re using <. |
| res[s] = 0; |  | Ensures we’re initialized properly. |
| if (p == t) { |  | If our hashes match… |
| }  //Naive algorithm described above |  | …double check with Naive |
|  |  |
| if (s < n - m) { |  |  |
| t = addModuloChar(mulModuloBase(subModulo (t, mulModuloChar(h, haystack[s]))), haystack[s + m]);  }  } |  | Shift to the next hash in the haystack. |

## Knuth-Morris-Pratt Algorithm

#### Matcher

|  |  |  |
| --- | --- | --- |
| C Implementation | Pseudo-code | Notes |
|  |  | Brought in through args |
|  |  |
| int pi[m] |  | Create *pi* |
| computePrefixFunction(needle, m, pi); |  |  |
| int q = 0; |  |  |
| for (int i = 0; i < n; i++) { |  | There are several *±1* offsets due to our arrays starting at *0* instead of *1* |
| res[i] = 0; |  | Ensures we’re initialized properly. |
| while (q > 0 && needle[q] != haystack[i]) { |  |  |
| q = pi[q - 1];  } |  |  |
| if (needle[q] == haystack[i]) { |  |  |
| q++;  } |  |  |
| if (q == m) { |  |  |
| res[i - (m - 1)] = 1; |  |  |
| q = pi[q - 1];  }  } |  |  |

#### Compute Prefix

|  |  |  |
| --- | --- | --- |
| C Implementation | Pseudo-code | Notes |
|  |  | Brought in through args. |
|  |  | Created in main function and brought in through args. |
| pi[0] = 0; |  |  |
| int k = 0; |  |  |
| for (int q = 1; q < m; q++) { |  | There are several *±1* offsets due to our arrays starting at *0* instead of *1* |
| while (k > 0 && needle[k] != needle[q]) { |  |  |
| k = pi[k - 1];  } | k |  |
| if (needle[k] == needle[q]) { |  |  |
| k++;  } |  |  |
| pi[q] = k;  } |  |  |
|  |  | Our function modifies *pi* directly, so no need to return it. |

# Testing

## Test #1

This test consists of a haystack with a needle . It provides a worst-case scenario for the Naive Algorithm.

#### Naive Algorithm

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | m (x100) | | | | | |
|  | time (ms) | 1 | 2 | 3 | 4 | 5 |
| n (x1,000,000) | 1 | 282 | 535 | 809 | 1034 | 1288 |
| 2 | 546 | 1061 | 1566 | 2052 | 2546 |
| 3 | 803 | 1566 | 2362 | 3109 | 3839 |
| 4 | 1073 | 2083 | 3205 | 4170 | 5152 |
| 5 | 1346 | 2629 | 3940 | 5129 | 6406 |

Worst Case

This shows a clear worst-case scenario as the algorithm must check character for every character in . Processing time doubles as doubles and doubles as doubles. This reflects the theoretical worst-case.

#### Rabin-Karp Algorithm

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | m (x100) | | | | | |
|  | time (ms) | 1 | 2 | 3 | 4 | 5 |
| n (x1,000,000) | 1 | 58 | 59 | 60 | 62 | 56 |
| 2 | 111 | 117 | 118 | 118 | 102 |
| 3 | 188 | 172 | 176 | 173 | 173 |
| 4 | 228 | 263 | 247 | 230 | 231 |
| 5 | 292 | 287 | 288 | 287 | 300 |

This shows a best-case scenario, unaffected by the length of . Because our hash never changes and never equals (except for the very end), we only hit the Naive Algorithm once.

#### Knuth-Morris-Pratt Algorithm

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | m (x100) | | | | | |
|  | time (ms) | 1 | 2 | 3 | 4 | 5 |
| n (x1,000,000) | 1 | 9 | 12 | 10 | 10 | 10 |
| 2 | 19 | 20 | 20 | 21 | 20 |
| 3 | 30 | 30 | 30 | 30 | 30 |
| 4 | 40 | 41 | 41 | 41 | 40 |
| 5 | 51 | 51 | 51 | 51 | 52 |

The results show the expected complexity of . Completion time increases linearly with and is unaffected by .

## Test #2

This test consists of a haystack of random characters with a needle equal to the last characters of .

#### Naive Algorithm

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | m (x100) | | | | | |
|  | time (ms) | 1 | 2 | 3 | 4 | 5 |
| n (x1,000,000) | 1 | 16 | 9 | 7 | 7 | 7 |
| 2 | 16 | 9 | 18 | 16 | 17 |
| 3 | 16 | 23 | 18 | 10 | 11 |
| 4 | 30 | 30 | 30 | 31 | 31 |
| 5 | 44 | 40 | 38 | 47 | 38 |

Average Case

This shows an average scenario (close to best-case). Because the algorithm will likely terminate its inner loop relatively early, the entire length of will not need to be calculated for each value of .

#### Rabin-Karp Algorithm

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | m (x100) | | | | | |
|  | time (ms) | 1 | 2 | 3 | 4 | 5 |
| n (x1,000,000) | 1 | 58 | 59 | 60 | 62 | 56 |
| 2 | 111 | 117 | 118 | 118 | 102 |
| 3 | 188 | 172 | 176 | 173 | 173 |
| 4 | 228 | 263 | 247 | 230 | 231 |
| 5 | 292 | 287 | 288 | 287 | 300 |

Average Case

This also shows an average scenario (very close to the best-case). With a large enough prime number used for hashing, it’s very unlikely we’ll receive a false positive.

#### Knuth-Morris-Pratt Algorithm

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | m (x100) | | | | | |
|  | time (ms) | 1 | 2 | 3 | 4 | 5 |
| n (x1,000,000) | 1 | 5 | 5 | 6 | 5 | 7 |
| 2 | 11 | 11 | 1 | 3 | 11 |
| 3 | 17 | 16 | 7 | 18 | 17 |
| 4 | 22 | 22 | 22 | 23 | 22 |
| 5 | 29 | 28 | 28 | 28 | 31 |

The results again show the expected complexity of . Completion time increases linearly with and is unaffected by .

## Test #3

This test consists of a haystack of the text from several textbooks with a needle equal to the last characters of .

#### Naive Algorithm

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | m (x100) | | | | | |
|  | time (ms) | 1 | 2 | 3 | 4 | 5 |
| n (x1,000,000) | 1 | 16 | 7 | 8 | 8 | 8 |
| 2 | 16 | 18 | 20 | 13 | 9 |
| 3 | 32 | 16 | 16 | 17 | 16 |
| 4 | 31 | 16 | 16 | 7 | 16 |
| 5 | 16 | 17 | 19 | 16 | 16 |

#### Rabin-Karp Algorithm

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | m (x100) | | | | | |
|  | time (ms) | 1 | 2 | 3 | 4 | 5 |
| n (x1,000,000) | 1 | 56 | 58 | 58 | 57 | 56 |
| 2 | 115 | 104 | 114 | 105 | 102 |
| 3 | 131 | 139 | 134 | 139 | 134 |
| 4 | 134 | 126 | 130 | 139 | 160 |
| 5 | 130 | 142 | 142 | 141 | 146 |

#### Knuth-Morris-Pratt Algorithm

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | m (x100) | | | | | |
|  | time (ms) | 1 | 2 | 3 | 4 | 5 |
| n (x1,000,000) | 1 | 8 | 5 | 5 | 6 | 6 |
| 2 | 11 | 14 | 17 | 3 | 18 |
| 3 | 21 | 15 | 17 | 16 | 14 |
| 4 | 21 | 16 | 18 | 13 | 24 |
| 5 | 19 | 15 | 21 | 13 | 33 |

While I would expect this test to run similarly to Test #2, the results from all three algorithms resemble a complexity closer to than . I could speculate that there is a change in some calculations or inner loop run times due to the characters following patterns found in the English language. It’s also possible this test is flawed in a way I’m not seeing, but it would require further examination. I’ve kept the results in the writeup for informative purposes but will not be attempting to analyze them.

## Test #4

This test consists of a haystack with a needle . It provides a worst-case scenario for the Naïve and Rabin-Karp Algorithms.

#### Naive Algorithm

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | m (x100) | | | | | |
|  | time (ms) | 1 | 2 | 3 | 4 | 5 |
| n (x1,000,000) | 1 | 276 | 527 | 810 | 1057 | 1318 |
| 2 | 552 | 1062 | 1545 | 2063 | 2603 |
| 3 | 804 | 1576 | 2357 | 3114 | 3866 |
| 4 | 1104 | 2115 | 3142 | 4143 | 5215 |
| 5 | 1337 | 2645 | 3861 | 5133 | 6473 |

Worst Case

Same results as Test #1, which is to be expected.

#### Rabin-Karp Algorithm

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | m (x100) | | | | | |
|  | time (ms) | 1 | 2 | 3 | 4 | 5 |
| n (x1,000,000) | 1 | 341 | 577 | 872 | 1095 | 1337 |
| 2 | 633 | 1156 | 1673 | 2178 | 2705 |
| 3 | 963 | 1730 | 2490 | 3235 | 4002 |
| 4 | 1270 | 2318 | 3361 | 4340 | 5347 |
| 5 | 1661 | 2934 | 4122 | 5451 | 6760 |

Worst Case

This shows slightly worse results than the Naive Algorithm. This is expected as this algorithm is essentially running the entire Naive Algorithm plus it’s own preprocessing and modulo overhead.

#### Knuth-Morris-Pratt Algorithm

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | m (x100) | | | | | |
|  | time (ms) | 1 | 2 | 3 | 4 | 5 |
| n (x1,000,000) | 1 | 9 | 12 | 10 | 10 | 10 |
| 2 | 19 | 20 | 20 | 21 | 20 |
| 3 | 30 | 30 | 30 | 30 | 30 |
| 4 | 40 | 41 | 41 | 41 | 40 |
| 5 | 51 | 51 | 51 | 51 | 52 |

Same results as Test #1, which is to be expected.

# Notable Observations

* The Knuth-Morris-Pratt Algorithm seems to have a worst-case when is homogenous, a best/better case when followed a certain pattern (Test #3 with the English language), and an “in-the-middle” case when and were randomized.
* The Rabin-Karp Algorithm shows a fair amount of overhead. I would speculate this is due to the modulo arithmetic that needs to be calculated during every loop iteration. It still scales as expected.
* Test #3 was included, but not analyzed. The test material was less controlled, and the results were unexpected.