**COSC 320 – 001**

***Analysis of Algorithms***

2022/2023 Winter Term 2

**Project Topic Number: #2**

**String Matching for Plagiarism Detection**

**Group Lead:**

**Youssef Mahmoud**

**Group Members:**

**Esteban Martínez (22717805),**

**Youssef Mahmoud (37624970),**

**Khalid Mahmoud (28842458).**

**Problem Formulation**

Given an input string of size (without counting whitespace), and given an array of strings of length where and is of size :

The algorithm should output a list of key-value pairs where each key is the index in where plagiarism was detected, and the value is the length of the substring which has been plagiarised, such that length() = number of instances of plagiarism .

**Pseudo-code**

**Knuth Morris Pratt Algorithm:**

| 1. **LPS ← ComputeLPS(Pattern) {build LPS table function}** 2. **i ← 0** 3. **j ← 0** 4. **n ← string length** 5. **m ← pattern length** 6. **while i < n do** 7. **if pattern[j] = string[i] then {if the characters are a match}** 8. **i ← i + 1** 9. **j ← j + 1** 10. **if j = m then {j pointer has reached end of pattern}** 11. **return i - j {index of the match}** 12. **j ← LPS[j - 1]** 13. **else if i<n && pattern[j] != string[i] then {no match}** 14. **if j > 0** 15. **j ← LPS[j - 1]** 16. **else** 17. **i ← i + 1** 18. **return -1 {no match}** |
| --- |

**LCSS Algorithm:**

| **Takes X = < x\_1,...x\_m > and Y = < y\_1,...y\_n > as input. Stores c[i,j] into table c[0..m,0..n] in row-major order. The array b[i,j] points to the table entry for optimal subproblem solution when computing c[i,j].**  **LCS-Length(X, Y)**   1. **m <- length[X]** 2. **n <- length[Y]** 3. **for i <- 1 to m** 4. **c[i,0] <- 0** 5. **for j <- 1 to n** 6. **c[0,j] <- 0** 7. **for i <- 1 to m** 8. **for j <- 1 to n** 9. **if (x\_i == y\_j) {** 10. **c[i,j] <- c[i-1,j-1] + 1** 11. **b[i,j] <- NW** 12. **}** 13. **else if (c[i-1,j] >= c[i,j-1]) {** 14. **c[i,j] <- c[i-1,j]** 15. **b[i,j] <- N** 16. **}** 17. **else {** 18. **c[i,j] <- c[i,j-1]** 19. **b[i,j] <- W** 20. **}** |
| --- |

**Algorithm Analysis**

* **KMP Algorithm:**

The KMP (Knuth-Morris-Pratt) algorithm is a string matching algorithm that is used to search for a pattern within a larger text string. Unlike other string matching algorithms, the KMP algorithm uses a preprocessing step to construct a failure function, which helps the algorithm avoid unnecessary comparisons during the actual search. This results in a more efficient and faster algorithm compared to the naive string matching algorithm.

The failure function is stored in an auxiliary array and helps the algorithm determine how many characters in the pattern can be skipped in case of a mismatch. During the search, the algorithm compares the characters of the pattern to the characters of the text and, in case of a mismatch, uses the failure function to determine how many characters in the pattern can be skipped in the next iteration.

The time complexity of the KMP algorithm is O(m + n), where m is the length of the pattern and n is the length of the text. This is more efficient than the time complexity of the naive string matching algorithm, which is O(mn).

**Proof of Correctness:**

Let P be a pattern string of length m and T be a text string of length n. Let f(i) be the length of the longest proper prefix of P that is also a suffix of P[1 ... i]. Then, for any position j in T, if the KMP algorithm has matched P[1 ... i] to T[j - i + 1 ... j], then it will match P[1 ... i + 1] to T[j - i + 1 ... j + 1] if and only if P[i + 1] = T[j + 1].

The proof of this statement can be established by induction. The base case, i = 0, is straightforward. For the induction step, assume that the statement is true for some i. Then, if P[i + 1] = T[j + 1], we have P[1 ... i + 1] = T[j - i + 1 ... j + 1], so the KMP algorithm will match P[1 ... i + 1] to T[j - i + 1 ... j + 1]. If P[i + 1] ≠ T[j + 1], then we have P[f(i) + 1 ... i + 1] ≠ T[j - i + 1 ... j], so we can use the information stored in f(i) to skip some characters in T and move the matching process forward. In this case, the KMP algorithm will match P[1 ... f(i)] to T[j - i + 1 + (j - (j - i + 1) - f(i) + 1) + 1 ... j - (j - i + 1) + f(i)], and the induction hypothesis applies to f(i), so the KMP algorithm will match P[1 ... f(i) + 1] to T[j - i + 1 + (j - (j - i + 1) - f(i) + 1) + 1 ... j - (j - i + 1) + f(i) + 1].

Therefore, the statement is true for all i, so the KMP algorithm is correct. This induction proof shows that the KMP algorithm correctly uses the information stored in the partial match table (failure function) to skip characters in the text string and move the matching process forward, ensuring linear time complexity.

* **LCSS Algorithm:**

The Longest Common Subsequence (LCSS) algorithm is a dynamic programming approach used to find the longest common subsequence between two strings or sequences. The basic idea behind the algorithm is to compare characters of the two sequences and store the length of the LCS in a 2D table. The LCSS algorithm uses a bottom-up approach to fill the table by iterating through the characters of the two sequences and updating the length of the LCS whenever a common character is found.

The final result of the algorithm is the last cell of the 2D table which represents the length of the LCS. This length can be used to find the actual sequence by backtracking through the table, starting from the bottom-right cell and following the path of the longest common subsequence. The LCSS algorithm has a time complexity of O(m \* n), where m and n are the lengths of the two sequences.

**Proof of Correctness:**

Let X and Y be two strings of lengths m and n, respectively. Let c[i][j] be the length of the LCS of X[1...i] and Y[1...j]. Then, for any i and j, c[i][j] = max(c[i - 1][j], c[i][j - 1], c[i - 1][j - 1] + 1) if X[i] = Y[j], and c[i][j] = max(c[i - 1][j], c[i][j - 1]) otherwise.

The base case, i = 0 or j = 0, is straightforward. For the induction step, assume that the statement is true for some i and j. Then, if X[i] = Y[j], we have c[i][j] = max(c[i - 1][j], c[i][j - 1], c[i - 1][j - 1] + 1), so the statement is true for i + 1 and j + 1. If X[i] ≠ Y[j], then we have c[i][j] = max(c[i - 1][j], c[i][j - 1]), so the statement is true for i + 1 and j + 1.

Therefore, the statement is true for all i and j, so the LCS algorithm is correct. This induction proof shows that the LCS algorithm correctly computes the length of the LCS of two strings by considering all possible combinations of subproblems, ensuring that the correct answer is obtained.

**Unexpected Cases/Difficulties**

One of the issues we have considered with using these algorithms is to ensure that we are able to implement them in a way that makes them comparable to each other in terms of real world performance. For instance, the KMP algorithm would return the index of the first instance of plagiarism it detects, whereas the LCSS algorithm would return a string which must then be decided to be plagiarism or not.

To account for this issue, we decided to normalize the output for all our implementations to be a key value pair of instances of plagiarism. In this way we can quantify the percentage of the text which was plagiarised for each of the algorithms and be able to compare one to each other fairly. We should take care to make sure whichever way we decide to implement each algorithm does not affect the complexity of the procedure itself.

Another aspect which must be accounted for is the preprocessing of the data. We must make sure that whichever way we decide to process the data in each implementation, we must keep it consistent from one to the other so this does not add extra processing time to each of the algorithms. It might also be a challenge in itself to handle a large data set and be able to implement its use correctly.

**Task Separation and Responsibilities**

**First milestone: 1.1** Problem formulation for KMP and LCSS fingerprint algorithms **1.2** Pseudo code for KMP and LCSS algorithms, **1.3** Algorithmic analysis and Proof of correctness and running time of KMP and LCSS algorithms.

**Second milestone:** **2.1** Implementation of KMP and LCSS

**Third milestone: 3.1** Problem formulation for the Rabin Karp fingerprint algorithm **3.2** Pseudo code for the Rabin Karp fingerprint algorithm, **3.3** Algorithmic analysis and Proof of correctness and running time of the Rabin Karp fingerprint algorithm.

**Fourth milestone: 4.1** Implementation of Rabin Karp and integration of algorithms, **4.2** Full test cases and time plots with growing inputs **4.3** Video presentation which includes an explanation of choices and designed algorithms, and display of implementations.

| Name | Tasks |
| --- | --- |
| Khalid | 1.2, 2.1, 3.3, 4.1, 4.2, 4.3 |
| Youssef | 1.3, 2.1, 3.1, 4.1, 4.2, 4.3 |
| Esteban | 1.1, 2.1, 3.2, 4.1, 4.2, 4.3 |