**COSC 320 – 001**

***Analysis of Algorithms***

2022/2023 Winter Term 2

**Second Milestone**

**Project Topic Number: #2**

**String Matching for Plagiarism Detection**

**Group Lead:**

**Youssef Mahmoud**

**Group Members:**

**Esteban Martínez (22717805),**

**Youssef Mahmoud (37624970),**

**Khalid Mahmoud (28842458).**

**Abstract:**

For this milestone, we analysed and showed the correctness of the Rabin-Karp fingerprint algorithm for string matching, as well as a brief analysis of the naive implementation which will be used for comparison. This is the last algorithm we wish to implement, and leaves us with the implementation of all three algorithms for our next two milestones. Our original plan was to begin implementing the previous two algorithms we had discussed, but after meeting as a team and realizing our workloads or this timeframe would diminish the quality of our delivery, we decided to modify our plans for this milestone to best accommodate the team. We think this showed our ability to remain flexible and prioritize the needs of the project. Additionally, the exploration of the final algorithm we plan to implement has left us with a clear idea of how they all relate to one another, and has given us a solid understanding of how each of our implementations should compare. We believe this new approach has given us a better idea of the work ahead.

**Algorithm Analysis:**

**Rabin-Karp Fingerprint Algorithm**

The Rabin-Karp Fingerprint algorithm works by generating a fingerprint for the pattern and for each substring of the text, and then comparing these fingerprints to find matches. The algorithm uses a hash function to efficiently search for occurrences of the pattern within the text.

2. Proof of Correctness

Let T be the text and P be the pattern, and let h(T[i...i+m-1]) be the fingerprint of substring T[i...i+m-1], where m is the length of the pattern. Let h(P) be the fingerprint of the pattern.

Assuming that h(T[i...i+m-1]) and h(P) are computed using the same hash function, the algorithm works by comparing the fingerprints of the pattern and the substrings of the text. If the fingerprints match, the algorithm checks if the substring and pattern match character by character.

Let s be a substring of T that matches P, and let j be the index of the first character of s in T. Then, we have: h(T[j...j+m-1]) = h(P). By the definition of the hash function, we know that if two strings have the same hash value, then they are likely to be the same. Therefore, if the fingerprints match, we can assume that s and P are the same string. Moreover, if the fingerprints do not match, we can assume that s and P are different strings.

To prove that the Rabin-Karp algorithm correctly identifies all matches of P within T, we need to show that it does not miss any match and that it does not report any false positives.

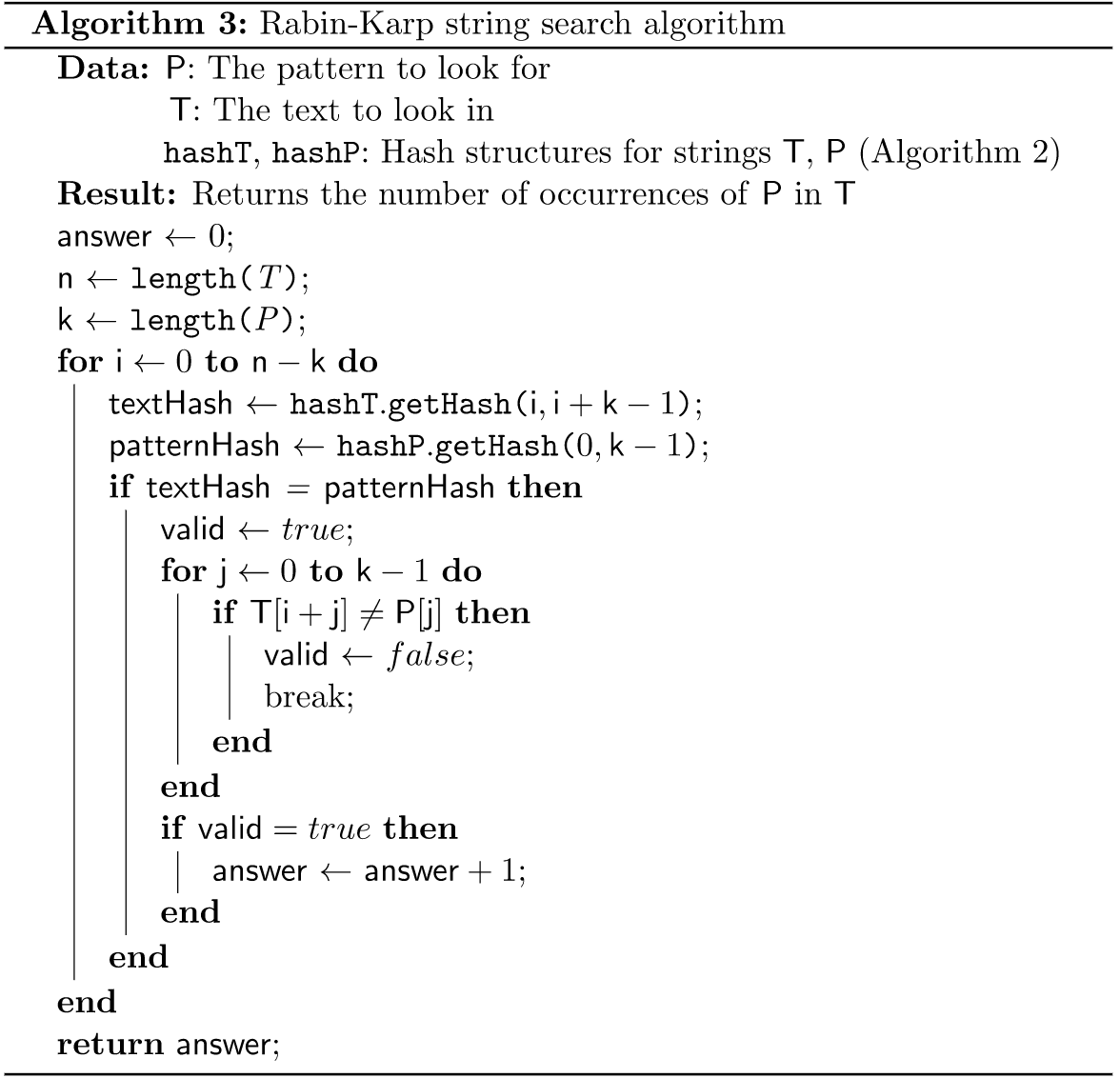
· The algorithm does not miss any match: Suppose there exists a substring s of T that matches P, but the algorithm does not report it. This implies that the fingerprint of s and P do not match. However, since s and P are the same string, the fingerprint of s and P must be the same. This is a contradiction, and therefore the algorithm does not miss any match.

· The algorithm does not report any false positive: Suppose the algorithm reports a substring s of T as a match, but s does not match P. This implies that the fingerprint of s and P match. However, since s and P are different strings, the fingerprint of s and P should not be the same. This is a contradiction, and therefore the algorithm does not report any false positives. Therefore, the Rabin-Karp algorithm correctly identifies all matches of P within T.

3. Best Average and Worst Case Running Times

The Rabin-Karp algorithm has a worst-case time complexity of O(mn), where m is the length of the pattern and n is the length of the text. However, in practice, the algorithm's performance is much better, with an average case time complexity of O(n+m) and a best-case time complexity of O(m) when there are no matches. The average and best-case time complexities depend on the distribution of the hash function and the number of matches. In practice, the algorithm is often faster than other string matching algorithms, especially for large texts or when the pattern length is variable. The algorithm's performance can be further improved by using a rolling hash function that updates the fingerprint value of each substring by reusing the hash value of the previous substring.

**Pseudo-code**



**Naive String Matching Algorithm**

1. Algorithm Analysis

This naive algorithm is the simplest method for pattern searching. It checks for all of the characters in the candidate string against the pattern. This algorithm does not need pre-processing or use extra memory by using a data structure. This approach checks for all possible placements of the pattern (which might be a sentence for example) in the candidate text. The procedure of the algorithm is to check the candidate text character by character; if the first letter of the pattern is found, the pointer for the text and the pattern is then moved one space and the procedure repeats. If a letter is not found, the pointer for the text is advanced by 1, and the pointer for the pattern is reset.

2. Proof of Correctness

The pseudo code code for the procedure is:

| 1. **Naive String Matching (T, P)** 2. **for i = 0 to n-m do** 3. **if P[1…m] == T[i+1…i+m] then** 4. **return 0 {match}** 5. **return -1 {no match}** |
| --- |

Since every character in the text is checked at least once against the pattern, the correctness of the algorithm is self-evident.

3. Best Average and Worst Case Running Times

The best case for the algorithm happens is the first character of the pattern is not in the candidate text. In this case, the number of comparisons will be n (the length of the candidate text). Therefore, the algorithm is .

The worst case for the algorithm happens in two scenarios. The first is when all characters of the text and pattern are the same. The second happens when only the last character of the text and the pattern are different. In this case, the number of comparisons is , and therefore the algorithm is .

**Data Structure:**

The Rabin-Karp algorithm uses a hash function to generate a fingerprint for the pattern and for each substring of the text, and then compares these fingerprints to find matches. The hash function is used to efficiently search for occurrences of the pattern within the text by generating a unique integer value for each possible pattern. This hash value can be stored in a variable or an array.In some implementations of the algorithm, a rolling hash function is used to update the fingerprint value of each substring by reusing the hash value of the previous substring. This reduces the number of hash function calculations required and improves the algorithm's performance even further.

**Unexpected Cases/Difficulties:**

One of our concerns with this algorithm will be the python implementation of the hash table. To address this, we will familiarize ourselves with said data structure and brush up on our Python concepts. We also believe that using the naive implementation of this algorithm will prove challenging in terms of the time it could potentially take to run it and acquire data, since it is significantly slower than our other algorithms.

**Task Separation and Responsibilities:**

**Second milestone:** **2.1** Problem formulation for the Rabin Karp Fingerprint Algorithm(RKA) **2.2** Pseudo code for RKA **2.3** Algorithmic analysis and Proof of correctness and running time of the RKA

| Khalid | 2.2 |
| --- | --- |
| Youssef | 2.3 |
| Esteban | 2.1 |