**Introduction**

The power of computing introduced the idea of solving problems that would otherwise take a significant amount of time in a matter of seconds. Let us posit for a moment the process of pattern matching. The problem of finding a matching string is a linear one. In other words, one would have to proceed word by word and stop once the match has been found. This is horribly inefficient. Because of this, we are here to introduce the Rabin-Karp string pattern matching algorithm. This algorithm offers many advantages over traditionally naïve string searching algorithms that are in use today.

Take the naïve string search for example. It needs to compare the key string and match character by character to the document text. Imagine trying to find a needle in a haystack by comparing a single piece of hay, one by one, to check whether it is the needle that you are looking for. The Rabin-Karp algorithm essentially uses hashing to significantly cut down the time required to find a pattern in a document. There are many practical applications of the Rabin-Karp algorithm a well. A common application that the Rabin-Karp is used for is detecting plagiarism due to its implementation of multiple searching. The algorithm can rapidly find instances of a “pattern” key thus offering an advantage over single-string searching algorithms. The algorithm itself is easy to implement and the justifications it has over many other searching algorithms should make it obvious why Rabin-Karp can be a powerful tool.

**Implementation**

The simple pattern search algorithm functions by comparing the document text of **n** characters to each character **m** of the pattern key. For every character in **n** that matches **m**, we proceed until the entire pattern **m** matches. If the character in **n** does not match the character in pattern **m**, we jump to the next iteration of character in **n** until a match is found. This process takes a staggering Θ(nm) matching time. Like the naïve string search, Rabin-Karp also functions by doing a comparison of each character **m**. However, what Rabin-Karp essentially does is takes the hash value of both the pattern and the current substring of the text and compares the two. In order to implement Rabin-Karp, we now need to calculate hash values for the pattern itself of **m** characters and all the substrings of the text of length **m**. A hashing function essentially maps strings to a unique numeric value. The algorithm utilizes the idea of hashing where if two strings match, they must also have the same hash value.

The hashing function used in Rabin-Karp returns an integer value representing the numeric value of a string. Using a large prime base and a given substring, we would take the ASCII value of each character in the substring and do modular arithmetic thereby returning the integer value representing that string. The challenge here is to implement said hash function. A naïve approach would require O(mn) time which would not reduce the runtime at all. As a result, we can implement a rolling hash as described above which basically adds the values of each character in the substring allowing the computation of the next hash value from the previous value. In other words, we start by calculating the hash value of the pattern **m**. As we are iterating through the text, we calculate the hash value of the first potential match of text of length **m**:

1. Start with the hash of some text “wxyz”: ‘w’ \* R3 + ‘x’ \* R2 + ‘y’\* R + ‘z’
2. Subtract ‘w’ \* R3 + ‘x’ \* R2 + ‘y’\* R + ‘z’
3. Multiply the result by R, dynamically adding the next character, and mod Q:

‘x’ \* R3 + ‘y’ \* R2 + ‘z’\* R + ‘a’ mod Q for the hash of “xyza”

The rolling hash essentially produces a new hash with a fixed number of operations and takes constant time. If the hash value matches, we then also can ensure the match by comparing the substring and pattern character by character. If they do not match, we simply “slide” the substring over to the next potential match of text of length **m**. Ultimately, this results in an average case of Θ(n+m) but still a worst case of Θ(nm). However, the worst case only occurs when all the characters of pattern **m** and the text **n** are same as the hash values of all substrings of the text. In other words, the worst case is extremely rare and can practically be ignored.

**Justification**

Two commonly used string pattern matching algorithms today are the KMP and Boyer-Moore mismatched character heuristic algorithms. These algorithms generally have a faster runtime than Rabin-Karp with similar preprocessing time. To quickly overview, KMP computes a deterministic finite automaton for the pattern **m** allowing the algorithm to dynamically compare the string pattern to the text. This allows the algorithm to essentially keep track at what point of the pattern it is being compared rather than having to start over for every mismatch. KMP has a worst-case runtime of Θ(n). On the other hand, the Boyer-Moore algorithm, like Rabin-Karp, seeks to lower the number of comparisons needed to be done to find a pattern match. Boyer-Moore works by starting the comparison with the last character of **m**. When the algorithm encounters a mismatch, the pattern slides forward as many characters depending on the mismatch heuristic. If the mismatched character does not appear in the pattern text, the pattern is slid forward **m** characters; however, if the mismatched character does appear in the pattern text, the pattern is slid forward to the next occurrence of that letter. This ensures that the algorithm does not accidently skip past a match of the pattern **m**. Boyer-Moore mismatched character heuristic has a best-case runtime of O(n/m) and worst-case runtime O(nm). While these benefits of runtime offer an advantage of single pattern string matching, Rabin-Karp is the algorithm of choice for multiple pattern searching. In other words, we can use Rabin-Karp to check multiple occurrences of whether a hash value of a given string belongs to a set of hash values of patterns thus allowing for a greater variety of practical applications.

Rabin-Karp does have its shortcomings. As with many hashing functions, collisions may arise when two different strings can match to the same hash value. To mitigate this, a good hashing function must be used as described earlier but even then, collisions might still occur. A solution to this is to match the pattern to the substring character by character when a hash value is matched. Depending on the pattern that is being compared, this can actually take quite some time. Interestingly enough, this introduces the concept of Las Vegas vs Monte Carlo algorithms. Monte Carlo is an algorithm whose output may be incorrect but guaranteed to be produced quickly whereas Las Vegas is an algorithm whose output is guaranteed correct at the cost of runtime. A possible avenue for improvement on Rabin-Karp is to utilize even better hashing functions such that this problem virtually never occurs. Despite this, collision can easily be mitigated by implementing a secondary string check at the cost of runtime. Also, Rabin-Karp can be implemented as a Monte Carlo algorithm since collisions are extremely rare in the first place. We see the advantages that the Rabin-Karp algorithm provides not only over the naïve string search but also two commonly used string pattern matching algorithms used today. The practical applications of multiple string searching introduce endless applications that other algorithms simply cannot provide.

In conclusion, we have shown how to implement Rabin-Karp and how it offers a much quicker runtime of Θ(n+m) over the naïve string search. We have also shown why Rabin-Karp is preferred over other generally quicker algorithms due to its specific applications for multiple string searches. Furthermore, despite the shortcomings it may have we have provided an avenue for future work and how these issues are currently mitigated in its implementation. We are proud to present this new algorithm to the computing world and hope that it aids you in your endeavors.