The Importance of Rabin-Karp

Imagine your every day case whereby on a web-page you need to look for a specific pattern using the find feature in your web browser. This feature is an invaluable tool which allows you to avoid skimming through large swathes of text.

Alas, this tool is based on the recognized idea of string-pattern matching. Formally, string-pattern matching can be well-defined as the following. Let's assume we have a pattern of characters we want to find in a text. Can we find a place in the text whereby, consecutively, characters match those exactly found in our pattern?

As you may expect, there are many different solutions to finding a pattern inside a sea of text. The most useful of all these methods is the one I present before you which is the Rabin-Karp pattern matching algorithm!

The revolutionary idea behind this algorithm is to take advantage of hashing a series of characters to compare whether two strings are equal and therefore confirm a "hit" on a query pattern. This is the grand picture of crux of this innovative algorithm!

Foremost, I will begin by presenting a detailed explanation of how to implement this algorithm in code, going step-by step. In tandem with this explanation, I will present the higher-level understanding of the algorithm. This new algorithm uses constant additional memory unlike the Knuth-Morris-Pratt (KMP) algorithm and Boyer Moore algorithm (with the mismatched character heuristic), works in O(n+m) time without the need for more than O(1) extra memory, and is made for all "cases". This and more will be explained in the ensuing paragraphs.

Nevertheless, let's jump into how we can implement this algorithm in code. Start off by making a generic class and give it a name you desire. Make a main method for this class where you accept the pattern and the text, that needs to be searched through, as command line arguments. Once you have this, pass these in to another helper function, giving it a proper name such as "Rabin-Karp".

This helper method will be the substance behind what allows you to return the indices of the positions whereby the pattern occurs in the given text. Initialize a global variable "hash" to be equal to 0. This variable will store our hash function value. We need this value since Rabin-Karp decides to compare hash-values between patterns and substrings of text to find whether they are equal. This value will help us when we move on to a future section of our algorithm that requires a "sliding window". Once again, this will be explained in-depth later. Make sure to also initialize a global variable "lrgprm" that stores an extremely large prime number. This is necessary for the hashing function since large prime numbers give a uniform probability distribution of values for random strings. This helps avoid collisions and gives us a better chance of "true" positives. This means that it is very unlikely, but not impossible, two different strings will hash to the same value.

Furthermore, to compute "hash", make a for loop that starts at 0 and goes until the length of the pattern minus 1. Increment the loop variable by 1 after each iteration. Within this specific for-loop, multiply "hash" by 256 and take this entire value modulus "lrgprm". Once this loop is finished iterating, the value of "hash" is stored and will be useful for the algorithm further on.

Additionally, make another for loop. Initialize the loop variable to start at 0 and run until it is less than the length of the pattern, while incrementing the loop variable by 1. Within this for loop, we are going to calculate the hash values of the given pattern and the "initial window" of

text. A “window” of text is defined as a length of text equivalent to the length of the pattern. The “initial window” is the first “x” characters of the text where “x” represents the length of the pattern. This is a necessary starting point for the algorithm before we have our “sliding window” of text. A “sliding window” is where we move one character at a time through our given text to see if our pattern and “sliding window” match. Logically, we need an “initial window” at first to have a “sliding window”. So, to calculate this initial window, we need to first initialize two more global variables “pathash” and “txthash” to zero. These will store our pattern and text hashes, respectively.

Within this “initial window”-calculating for loop, set “pathash” equal to “pathash” multiplied by our alphabet radix (256) plus the pattern’s character at the integer loop variable. Then, set “pathash” equal to “pathash” modulus “lrgprm”. After this, set “txthash” to be equal to “txthash” multiplied by our alphabet radix (256) plus the text’s character at the integer loop variable. Then, set “txthash” equal to “txthash” modulus “lrgprm”. After this statement, this is the end of the code within this for loop.

Once again, we need to create another for loop. This for loop will serve as our “sliding window” to help us match the new “window” with our pattern. For this for loop, initialize the loop variable to 0 and iterate it till the loop variable is less than or equals the length of the text minus the length of the pattern. This is done since we cannot have our final “sliding window” length be less than the length of the pattern. Make sure to increment the loop variable by 1.

Within this for-loop, first check if “pathash” equals “txthash”. If this is the case, then print out that the pattern has been found at the index of the current loop variable. This is the last statement within this if statement. Make a separate if statement that checks whether the loop variable is less than the value of the length of the text minus the length of the pattern. This is the

part of the code that allows us to implement our “sliding window”. If this is the case, then we must move our “window” and hash the new window.

However, since we are moving one character at a time, the new hash function is equivalent to the old one except we subtract the hash of the first character of our old string and hash the next new character to our current string. To do this in code, set a new initialized integer variable (“z”) to be equal to 256 multiplied by the value of “txthash” minus the value of the text’s character at the loop variable multiplied by “hash”. Now set a new variable (“y”) to be equal to the text at the character of the value of the loop variable plus the length of the pattern. Finally, set “txthash” to be equal to the value of “z” plus “y”, then modulus “lrgprm”.

Yet, sometimes our new value of “txthash” can be negative so we must check if “txthash” is less than 0. If this is the case, then we set “txthash” equal to “txthash” plus “lrgprm” to make it positive once again. At the end of this if statement, we are done. Our searching method is complete. The if-statement is checked. The for loop is done iterating and we reach the end of our searching method.

This ingenious algorithm is better than what we already have! In comparison to the KMP algorithm and Boyer-Moore mismatched character heuristic algorithm, this algorithm was designed for all cases. The KMP algorithm may have a runtime of O(n+m) but that’s potentially misleading! The KMP and Boyer-Moore algorithms were designed for the worst case specifically. The worst case in string pattern matching is where the pattern has high repetition but is different in its last character while the text it is being checked against is also repetitive but is different at the last character in multiples of the pattern’s length. This worst-case is very rare in practice and so these algorithms off-the-bat are not optimized for an everyday string-pattern search. The Rabin-Karp algorithm is!

Better yet, the Rabin-Karp algorithm is the only string-pattern matching algorithm out of itself, Boyer-Moore and KMP to use no more than constant additional memory. The Boyer-Moore algorithm, using the character mismatch heuristic, uses “R” extra memory where “R” is the radix of the alphabet. In this case, it is 256. The KMP algorithm is even worse and uses “m*R” memory where “m” represents the length of the pattern and “R” once again represents the radix (256).

Lastly, algorithms with extra memory usage should ideally guarantee that they can do better than Rabin-Karp! This is also not the case, especially with the Boyer-Moore algorithm. Boyer-Moore still has a worst-case runtime of O(n*m). This is the same worst-case runtime of Rabin-Karp. Why use extra memory to still get no guaranteed better asymptotic worst-case runtime performance?

To conclude, the Rabin-Karp algorithm is a landmark string-pattern matching procedure that uses hashing. Other common solutions such as Boyer-Moore and the Knuth-Morris-Pratt algorithm (KMP) use extra memory and do not take advantage of the constant time results of hashing a string. This algorithm can be used in all “cases” of string-pattern matching as an ideal way to find a random pattern in a sea of random text.

On the contrary, we must as well provide some criticism of this hallmark achievement. The Rabin-Karp algorithm does not do better than other algorithm’s worst-case time complexity of O(m*n) where “m” is the length of the pattern and “n” is the length of the text. This is one possible area of future research and improvement.

Likewise, another downside of this algorithm is our usage of the “Monte Carlo” version of this algorithm. We assumed that two hash values that match are equivalent strings without

checking characters sequentially between our substring of text and the pattern. Research should be done to find whether the “Monte Carlo” version of this algorithm gives too many false positive “matches” versus the “Las Vegas” version of this algorithm, in the average case. The “Las Vegas” version of this algorithm checks characters sequentially after a hash value match to guarantee that we never get false positives.

All in all, Rabin-Karp is a great algorithm that has changed the landscape of string-pattern matching procedures while introducing hashing as an efficient means of pattern-substring checking!