Rabin-Karp

String pattern matching is a very common problem. Many different solutions have been presented to solve it. For string pattern matching, a given string is searched to see if it contains a certain substring. This is useful for things such as running a search of documents on a computer, or for a “control f” find operation in a web browser. The Rabin-Karp algorithm is a solution to the problem of string pattern matching. This substring is called a pattern. The pattern can be located anywhere in the string, and Rabin-Karp will find it if it exists. The algorithm uses a hash-based comparison method. This means that the letters of the string are not directly compared, but rather their determined hash values. The runtime of Rabin-Karp is not better than other approaches such as Boyer-Moore on average but can beat it in the worst case. However, for multiple pattern matching Rabin-Karp beats Boyer-Moore.

To do this, first an alphabet must be established. Usually, it is ASCII, but any numbering system could be used for a possible letter, so long as no two letters have the same number. Next, a hash function must be created to take a string as an input and output a unique value. This hash function can be implemented different ways, but we will use Horner’s method to hash the strings. In Horner’s method, strings are treated like a number. Each letter is given a positional weight, like the ten’s or hundred’s place in a decimal number. With our algorithm, we will use an ASCII alphabet, which means there are 256 possible values. This gives us a 256^0 place, a 256^1 place, 256^2 place, all the way up to 256^(the length of the input string). For each place, the value of the character in that place is multiplied by the positional weight. Each place is added together, and then a modulus of the size of the data type storing the value is taken. This will yield a unique number for a very large number of strings. Strings that create a hash value that overflows the data type could result in collisions, which need to be either accounted for or ignored. A collision is when two unique values both map to the same hash value.

To begin the algorithm, first the pattern string is sent through the hash function, and its value is stored. Then, begin by taking the first substring with equal length as the pattern. If the length of the pattern is larger than the length of the string, then the pattern is not contained in the string and no further testing is needed. Otherwise, the substring is sent through the hash function. The value of the substring is then compared to the value of the pattern. If they are equal then the pattern is found (or not, more on that soon). If they are unequal, then the next substring must be considered. The next substring is found by using a sliding window, which moves down the string by one letter each time. To compute the hash value of the next substring, a few options exist. Most simply, the substring could be sent into the hash function, and the entire value is computed. However, this is inefficient. From one substring to the next, the first and last letter change, but the rest of the substring remains the same. Recomputing the numerical value of the letters that stay the same is unneeded. Instead, a few simple steps are taken. First, subtract out the value of the letter being removed. Next, multiply the current hash value by R, which is the number of letters in the alphabet. This has the effect of shifting the letters in the word to the left by one place. This is aligned with the sliding window of which substring is being considered. Lastly, add the value of the new letter to the value of the hash. Now, the hash value represents the new substring, and much less work is required compared to computing the entire new substring with Horner’s method.

Whenever the value of the pattern matches the value of the current substrings, it is potentially a match. It might not be a match because of overflow when computing the hash value, and a collision might have occurred. When a very large pattern and substring are being considered, it is possible to overflow the data type being used. A modulus is used to prevent a runtime exception, but this case must be accounted for. There are two ways to deal with collisions in Rabin-Karp: The Las Vegas approach and Monte Carlo approach. In the Las Vegas approach, after the pattern value matches the current substring value, compare each character of the strings to double check that it is a match. This approach is guaranteed correct, but not guaranteed fast. It is not guaranteed to be fast due to the work required to check each character. If the pattern is short, the work needed is practically not noticeable. However, for large patterns, this work is significant, which causes the algorithm to be slow in this case. In the Monte Carlo approach, assume that when the pattern value matches the current substring value, it is correct. This could return false positives but would be guaranteed fast due to ignoring the extra work needed for long patterns.

Other string pattern matching solutions exist, such as the Knuth-Morris-Pratt algorithm, and the Boyer-Moore algorithm. The runtime of these, where M is the length of the pattern and N is the length of the string that is searched, can be written as O(N) for KMP and O(N\*M) for Boyer-Moore. Rabin-Karp is also O(N\*M). Also, Knuth-Morris-Pratt has the same average case run time as Rabin-Karp of O(N+M). For Boyer-Moore, the runtime in the average case is a blazing O(N/M).

The Knuth-Morris-Pratt (KMP) string patching method works in a very different way than Rabin-Karp. In KMP, the idea that every letter in the string being searched only needs to be compared one time, because once you compare it you know what letter it is. This allows for moving down the string being searched to be more efficient. To do this, a deterministic finite state automaton is constructed based on the pattern. Constructing this DFA is based on what character is being looked at in the string being matched. If the current character is a match, move to the next state. If not, many options are available for where to move. If part of the pattern is still matched, move to a lower state, but not back to zero. If the pattern is not able to be matched from previous letters, move to state zero. After constructing the DFA, simply follow which state to move to for the current character, and if the last state is reached the pattern is matched.

In Boyer-Moore pattern matching, we will consider the mismatched character heuristic. In this algorithm, a window of the string being searched is compared to the pattern. Somewhat unintuitively, the comparison between the string and the pattern starts at the end. If the last letter does not match, the window is not a match, so the window needs to be moved. How far the window should be moved depends on a few things. If the character in the string is contained at another point inside the pattern, slide the window down to align those two characters. If multiple occurrences of that character exist in the pattern, align the rightmost one with the string. Try the comparison process again and repeat until the pattern is found or the window tries to move past the end of the string. If it tries to move past the end of the string, the pattern is not contained in the string. This algorithm proves to be very efficient at searching English text, as most of the time a character isn’t in the pattern, and the first comparison is a mismatch. This leads to the average case runtime of O(N/M). Also, large patterns have potential to skip long distances in the string and save lots of time compared to other algorithms.

Comparing these other algorithms to Rabin-Karp, they each have different use cases. Rabin-Karp can be a more versatile algorithm than KMP because when the alphabet of possible characters is large KMP has few opportunities to reset to a state other than zero. With these large alphabet cases, Rabin-Karp is preferred. Many more use cases exist for a large alphabet than with a small alphabet. With Boyer-Moore, the smaller the pattern, the slower the runtime. Boyer-Moore is preferred on patterns that are larger so that moving the window down large distances occurs. With smaller patterns, Rabin-Karp is again preferred. Rabin-Karp excels over both Boyer-Moore and KMP at multiple string pattern matching. In multiple string pattern matching, a set of strings are used as patterns, and an entire input string is searched for occurrences of any of the pattern strings. Optimized versions of Boyer-Moore and KMP exist for this problem, but Rabin-Karp still provides a better practical runtime. Multiple string pattern matching is very useful in practice, and having an algorithm as efficient as Rabin-Karp is very helpful.

In conclusion, Rabin-Karp is a versatile solution to the string pattern matching problem. Using hashing to compare substrings can be efficiently accomplished by using a rolling hash value. There is no need to compare substrings on a character by character basis, although it could be used as a safeguard against false positives. Picking between the Las Vegas and Monte Carlo approaches will depend on the type of problem being solved using Rabin-Karp. If speed is the primary factor, then use Monte Carlo. If not, it is best to use Las Vegas and ensure a correct solution is returned.

To improve Rabin-Karp, the worst case should be considered. Is it possible to combine the approach of Boyer-Moore with hashing? Is it possible to further reduce collisions, or to create better ways of collision handling other than Las Vegas and Monte Carlo? These will be investigated as future work.