Describing Rabin-Karp

The subject of computer science constantly presents problems for algorithm designers to solve. Specifically, the problem of string pattern matching is an extremely prevalent problem in this field. The ability to determine whether or not a given pattern of characters is included in a larger string is an important tool to have. Additionally, we can also determine the exact position in the string in which the given pattern is located. In the Java language, the indexOf function is the String class’ included string pattern matching method. However, the implementation of Java’s indexOf utilizes a “brute force” algorithm with poor asymptotical runtime and several other disadvantages. For this reason, my team and I have created a new pattern matching algorithm in an attempt to solve the inherent problems which plague other solutions to this problem. This algorithm, which we call the Rabin-Karp algorithm, utilizes hashing in order to efficiently find patterns in strings. In brief, it functions by computing a hash value for the pattern and comparing it to the hash value of a portion of the string with the same length of the pattern. If the hash values match, then the pattern is found. If the hash values do not match, the next hash value is computed and compared to the pattern. By utilizing a technique known as the rolling hash, this can be done with great efficiency compared to other pattern matching algorithms. Rabin-Karp is also useful for multiple pattern matching, giving it an advantage over other algorithms without that capability.

In order to fully understand the Rabin-Karp algorithm, you must familiarize yourself with the concept of hashing. Hashing is the act of taking a large piece of data and representing it using a smaller piece of data. In order to achieve this, we use something called a hash function. An efficient hash function is integral to the performance of string pattern matching with Rabin-Karp. The key to a good Rabin-Karp hash function is its compatibility with the rolling hash concept. A rolling hash function has the ability to calculate the hash value of a string without having to completely rehash the entire string (given certain conditions). For example, if we have already calculated the hash of the string “abcd”, we can utilize a rolling hash to quickly calculate the hash value of the string “bcde” since they are the same length, and the only characters that differ between them are the first character of the first string and the last character of the second string. There are many different implementations of rolling hashes, but one of the most well-known implementations utilizes Horner’s method. Horner’s hash function is defined as *h = t0 \* RM-1 + t1 \* RM-2+ … + tM-1 \* R0 (mod Q)* where *ti* is the *ith* character of the string (converted to its ASCII integer equivalent), R is the length of the radix, and M is the length of the string. We also utilize modular arithmetic in order to keep the numbers small and avoid overflow. Given that we know a certain string *x*, we can calculate *h(x+1)* by using *h(x)* and the rolling hash calculation. The calculation is defined as *hi+1 = (hi - t0 \* RM-1) \* R + tM* where *hi* is the current hash value, and *tM* is the new trailing character. In other words, you take the original hash value, subtract the leading character, multiply by the length of the radix, and then add the new trailing character. For example, we can calculate *h(“abc”)* = *97\*2562 + 98\*2561 + 99\*2560 = 6382179* assuming the radix is every ASCII character and neglecting the modulus. Then, we can calculate *h(“bcd”) = (6382179 - 97\*2562) \* 256 + 100 = 6447972.* The rolling hash effectively allowed us to calculate the hash value of “bcd” in constant time using the hash value of “abc”.

String pattern matching can be done using Rabin-Karp in a series of four steps. To exemplify the algorithm, assume that we are searching for a pattern *P* with length *L* in string *S*.First, run *P* through a hash function compatible with the rolling hash (such as Horner’s). Then, hash the substring *X* consisting of the first *L* characters of *S*. If *h(X) = h(L)*, then we have found a match. Otherwise, use the rolling hash to calculate the hash of the next *L* characters of *S,* starting from the second character of *X*. If a match is still not found, we start from the third character of X, and so on. Continue calculating and comparing hash values until either a match is found, or we reach the end of the string. This is known as the Monte Carlo implementation of Rabin-Karp because we are assuming that the string is a match if the hash values are equivalent. However, as mentioned before, we need to use modular arithmetic in our hash function in order to keep numbers small. If we do not, we risk the possibility of overflowing past the max value possible in certain programming languages. A drawback of modular arithmetic is the possibility of collisions between hash values. A collision occurs when two different strings compute the same hash value. Using the previous implementation of Rabin-Karp, we could produce false positive matches – returning a match when there is not one because of a collision between strings. To solve this, we can use the Las Vegas implementation of the algorithm which considers collisions and does an exhaustive comparison between the substring and the pattern to ensure that there are no false positives.

Since string pattern matching is such a ubiquitous problem in computer science, there are numerous other algorithms which attempt to solve it. In order to portray the merits of Rabin-Karp, it is important to compare it to other popular pattern matching algorithms. As mentioned before, Java’s indexOf method uses a “brute force” algorithm which checks for a given pattern by starting at the first character of the string, checking if the character matches with the first character of the pattern, then moving to the next character. If a match is not found, the second character of the string becomes the new starting point. Using this method, our worst-case runtime becomes *O(MN)* where *M* is the length of the pattern and *N* is the length of the string. In the worst case with brute force, we are effectively iterating through the string once, and iterating through the pattern for every character in the string. Algorithms such as the Knuth-Morris-Pratt (KMP) substring search algorithm, and the Boyer-Moore mismatched character heuristic attempt to improve the runtime of a substring search. In short, KMP avoids checking characters in the string which we have already iterated through. For example, if we are searching for *“abc”* in the string *“abababc”,* brute force would compare *“abc”* to *“aba”* and then compare *“abc”* to *“bab”.* KMP would eliminate the need for the second comparison because we already know that the 2nd letter in the string is a *‘b’* and would not be a match. Boyer-Moore takes a different approach to avoiding backup. In Boyer-Moore, when searching for a length *L* pattern *P* in string *S,* comparisons begin at the last character of *P* and the *Lth* character of *S*. If the character is mismatched, we can move the pattern down the string past the mismatch which skips unnecessary comparisons.

Both Boyer-Moore and KMP offer runtime improvements over brute force but come with other trade-offs which Rabin-Karp avoids. In order to fully implement KMP, it is necessary to iterate through the pattern beforehand and construct a table which is used to determine how many comparisons we can skip. This pre-processing can be done in *O(M)* time and searching can be done in *O(N)* time, bringing the total average-case runtime to *O(M+N).* This algorithm also requires *O(MR)* extra space to store this table. Boyer-Moore also requires preprocessing and *O(R)* extra storage, having a total runtime of *O(M+N).* In comparison, the Rabin-Karp algorithm uses no extra space and does not require any preprocessing. Additionally, assuming the use of a good hash function, Rabin-Karp boasts an *O(M+N)* average case asymptotical runtime. As mentioned previously, Rabin-Karp can also be modified to support multiple pattern matching. Instead of just hashing one pattern, you can hash a set of patterns and compare them to the string using the same method discussed before. Other algorithms such as Boyer-Moore and KMP can only check for one pattern at a time. Therefore, when searching a string for a set of *Y* patterns, Rabin-Karp holds the same *O(M+N)* average runtime while Boyer-Moore and KMP degrade to *O(Y(M+N)).*

Rabin-Karp is a revolutionary string pattern matching algorithm with the ability to hold an *O(M+N)* average time complexity while requiring no additional space. Utilizing the rolling hash allows for efficient pattern matching without needing to preprocess the pattern or string. However, this does not come without certain shortcomings. Rabin-Karp has an *O(MN)* worst case runtime which makes it equivalent to brute force. This case occurs when the use of a poorly implemented hash function causes multiple collisions between strings. Similarly, with the possibility of collisions, it is generally preferable to use the Las Vegas implementation of Rabin-Karp. Unfortunately, this causes the runtime to degrade even further because it requires brute force comparisons between the string and pattern when a possible match is found to avoid false positives. With the Monte Carlo implementation, we guarantee an acceptable runtime but risk returning a falsely positive match between string and pattern. As it currently stands, Rabin-Karp cannot guarantee an *O(M+N)* runtime without requiring extra space and avoiding false matches. In the future, we will be working towards providing an algorithm that can guarantee these things in order to create a universally accepted algorithm for both single and multiple string pattern matching.