Picture this, instead of hitting ctrl + F on your keyboard to find a specific word you are looking for, you have to manually search through all of the text to find this matching word. This would take you an insurmountable amount of time, especially if say, you were searching for that word in a textbook. However, with evolution in what is called string pattern matching, this process of searching for a word has become as simple as typing in that word and letting the computer do the rest. This basis of string pattern matching has become a fundamental operation for strings. It uses the idea that given a string of length N and pattern of length M, search for an occurrence of the pattern within the text. Like most things involving algorithm implementation, string pattern matching can be solved in a variety of ways. The early brute force algorithm, set a benchmark to beat in the string pattern matching community. While, there are many solutions to string pattern matching, my Rabin-Karp approach uses the basis of modular hashing to effectively and efficiently find matching patterns.

Before jumping into the implementation of Rabin-Karp it is important that, I first provide a detailed description of how it works. Given a pattern (string) of length M to search for in a text, for the purpose of hashing, it is important to note that this pattern M corresponds to an M-digit Base-R (R = radix) number. Using this number and a hash table of size Q, we can hash these values to longs ranging from 0 to Q-1. Utilizing the general idea behind Horner’s method, our hash function for converting the pattern of length M, from an M-digit base-R number, to a long allows us to use this hash function for M of any size. To do so, store each digit in the corresponding M-digit base R-number in a char array. Utilizing this character array my algorithm then takes each digit in the character array, multiplies it by R, and adds the value of the digit you are at in character array to this. With this value, I then perform modular hashing with a table size of Q to compute the pattern’s hash value. Although this same method could be used for hashing the text to search for our hashed pattern, I modified it in my algorithm because it becomes very costly having to do multiplication, addition, and remainder operations for each text character.

Given that the approach described above would diminish to brute-force runtime, for the text being searched, it’s important for my algorithm to efficiently compute the position I am currently searching at in the text, based on the value calculated for the position before it. To do so in an efficient way I exploit a special property of the modulo operator. Given a hash table of size Q, if I take the remainder when divided by Q after all arithmetic operations, this will yield the same value hash value as computing modulo after each arithmetic operation. Therefore, to compute the hash value from one position of the text, to the next position in the text I use a mathematical function which uses the hash value of the previous position to first subtract off the leading digit. Following this, my algorithm multiplies that hash value of the previous position by R and then adds the M-digit R-value for the digit at the next position. Once this value is computed my algorithm then does modulo Q to compute the hash value for the next position in the pattern and will repeat this process for all patterns in the text being a multiple of M. This algorithm will find a match if the hash value of the current substring in the text is equivalent to the hash value of the pattern being searched for. Ultimately, this gives my algorithm the ability to move right one position in the text being searched in constant, O(1), time regardless of the value for M (M = length of pattern).

Now that I have provided a detailed description of Rabin-Karp algorithm, it’s time to hone in on how to actually implement this algorithm. The first step is to create a new class with a main method that takes the pattern and text string as command line arguments. The global variables needed for this program consist of: a long variable that stores hash value of the pattern, an int variable that stores the length in digits of the pattern, another long variable stored as a large prime number to be the size of the hash table, an int variable to store the size of the alphabet (radix), finally another long variable to store the calculation for removing the leading digit when searching for a match in text (formula: R^(M-1) % Q, R = radix, M = size of pattern, Q = size of hash table). After establishing these global variables, I made a constructor for this new string pattern matching class. This constructor takes in the pattern string given at the command line and pre-processes that string. First by setting the size of the alphabet (radix) to 256, to represent the number of extended ASCII characters. Following this, I set the global variable that stores the length of the pattern and the size of the hash table to some large prime number. To set the size of this hash table to be a random prime number, I create another method which generates a large prime number and returns a long. Next, for this constructor I pre-compute the global variable to be used in removing leading digit for searching the text. To do so, I set the value of this global to 1 and create a loop that iterates through the length of the pattern. During each iteration of this loop I set this global mentioned above to the size of the alphabet \* the current value stored in the global. Then I modulo this value each time and store it back into this global used for removing the leading digit. The final step of my constructor consists of computing the hash value for my pattern by calling a new method to hash my pattern.

To start, my hashing method must return a long, representing the hash value of whatever string was passed in. My method for hashing is used for both the pattern being searched for and the text, therefore my method needs to take in two parameters: one which holds the string you are hashing and another for the length of the string being hashed. This second parameter allows this hash method to be used for the pattern and the text. Once in this method I first make a long variable to hold the value of the hash function, followed by a loop. This loop cycles for the length of the string passed in, during each iteration computing the hash value of the current digit by multiplying the (size of the alphabet \* the hash value of the previous digit + the character value at the next digit to be hashed) % the size of the hash table. To complete this method for hashing, once I have generated a hash value for the full string that was passed in, I return this value. Without worrying about doing a final check to see if a matching hash value from text actually matches the characters in a given pattern, I only have one more method to implement.

This final method is used for searching the text and returning the index of the first occurrence of the pattern string in the text string. Since this method must return the index of this matching occurrence, my method returns an integer value. This method takes a single parameter, denoting the text string being searched. First, I declare an integer to store the length of the text string and immediately want to return from this method if the length of the text string is less than the length of the pattern string (cause if length of text is less than pattern it obviously isn’t going to match). Following this, I declare a long variable which stores the result of a call to my hash method run on the text string, for a substring the size of the pattern being searched for. Then I check if the hash value of the pattern matches the hash value of the substring of the text at offset 0 (first string in text), if so I return the index 0. Otherwise, I begin a loop which continually searches the text string for multiples of the length of the pattern (M). For each iteration of this loop I move right one position in the text, remove the leading digit from the text and add the new trailing digit (this calculation was explained in detail when describing the algorithm). Before going to the next iteration of this loop I first create a variable to store the offset at which the pattern was matched or mismatched in the text. If the hash value of the pattern is equal to the hash value for the string in the text, then return this variable that holds the offset as the index of the first occurrence of the pattern string in the text string. However, if our whole text string has been read in and no hash values match the pattern string, then I exit the loop and return the length of the text string, meaning no match was found. This marks the end of my method for searching for a pattern string in the text string. It is already evident that my Rabin-Karp approach is superior to other string pattern matching approaches, but let me show some examples how.

While my Rabin-Karp approach, improves performance overall, both the Knuth-Morris-Pratt and Boyer-More (with mismatched character heuristic) approaches came as adaptations to the original brute force algorithm. These algorithms aimed to improve the worst case scenario, which occurs when both the pattern being searched for and the texted being searched are all repetitive, but the final character of the pattern is different that that of the final character of the text. In this case all of the characters in the pattern are checked against the text giving a worst-case runtime of O(NM) (N = text length, M = pattern length). However, this worst case scenario is a rare occurrence in modern day practice, so the application of these algorithms in modern day is limited. On the contrary, my Rabin-Karp algorithm provides a probabilistic linear runtime guarantee in all runtime cases, as well as improving memory overhead.

The KMP substring search approach accesses no more than M+N character to search for a pattern on length M, in text of length N. Therefore, KMP is guaranteed linear time and required no backup on the input. In the case of memory overhead, KMP utilizes space proportional to M\*R, where M is the length of the pattern and R is the size alphabet(radix). Boyer-Moore (with mismatched character heuristic) run on standard inputs, uses N/M (O(N/M) runtime) character compares to search for pattern of length M in a text of length N. However, in the worst case this algorithm degrades to θ(NM) runtime yielding the same worst-case runtime as brute force. The memory overhead for Boyer-Moore isn’t as brutal as that for KMP, using space proportional to R, where R is the size of the alphabet (radix). My Rabin-Karp algorithm provides an average and best case runtime of O(N+M) and a rare worst case runtime of O(NM). Concerning the issue of memory overhead, this algorithm uses space proportional to constant memory. Hence, Rabin-Karp provides a high probability of linear runtime without using a great deal of memory overhead, making it superior to both KMP and Boyer-Moore.

Rabin-Karp has provided plentiful contributions to the string pattern matching community. Its use of hashing, makes the algorithm relatively easy to follow and understand as opposed to KMP which constructs a DFA within its code. Also, Rabin-Karp takes advantage of the constant time operation for hashing strings, yielding a high probability of linear runtime. This algorithm can also be extended to 2D patterns and text, making it very useful for image processing. Even though Rabin-Karp has paved the way for a new method of string pattern matching, it still has downsides.

One downside to Rabin-Karp is the small probability of collisions, where a string’s hash value from the text matches its pattern’s hash value, but the content of the string does not truly match the pattern. Since the hash table utilized is never stored in memory, I take advantage of this, by making the size of the hash table enormous to make collision occurrence very rare. There are two approaches to solving the collision issue, one being Las Vegas and the other being Monte Carlo. The Las Vegas implementation does a character by character comparison after each hash match, which guarantees the string will match, but with a high probability of being linear time. On the other hand, the Monte Carlo version assumes that a hash match means a substring match, which yields a guaranteed linear runtime with a high probability of the string being correct. Both of these implementations have their tradeoffs, sacrificing runtime for accuracy or vice versa. Therefore, to improve on future string pattern matching algorithms I would still use the general idea behind Rabin-Karp, but decipher a way to obtain guaranteed linear runtime and guaranteed correct matches.