**Mangesh Raut**

**Professor Mark Boady**

**CS 502 Data Structures and Algorithms**

**February 5th, 2022**

**Karp-Rabin Algorithm**

**Abstract**

Karp-Rabin Algorithm is an average case analysis of string-matching algorithm. This algorithm uses hash technique for string searching. In other words, we can say that Karp-Rabin uses finger printing technique. The initial hashes are called Fingerprints. This algorithm updates these fingerprints in constant time. Hashing technique adds ASCII values for each letter and mod it by some prime number. Good hash function needs to look at every character in the string. After calculating the hash function, If the values are same then we need to do normal string comparison. If both the strings are same, then we need to do normal string comparison. If both strings are same then the search completed, Else we have to continue. Now shift over a character in the string and calculate the hash value of it, Continue the process as above until the actual string is found or we reach to the end of the string.

**Introduction**

Karp-Rabin Algorithm is one of the most effective string-matching algorithms. It uses a rolling hash to quickly filter out positions of the text that cannot match the pattern, and then checks for a match at the remaining positions. Generalizations of the same idea can be used to find more than one match of a single pattern, or to find matches for more than one pattern.

To find a single match of a single pattern, the expected time of the algorithm is linear in the combined length of the pattern and text. To find multiple matches, the expected time is linear in the input lengths, plus the combined length of all the matches, which could be greater than linear. Rabin and Karp proposed a string-matching algorithm that performs well in practice and that also generalizes to other algorithms for related problems, such as two-dimensional pattern matching. It uses technique called hash. Hash technique used to find any set of patterns in a string. In Rabin-Karp algorithm, we'll generate a hash of our pattern that we are looking for & check if the rolling hash of our text matches the pattern or not. If it doesn't match, we can guarantee that the pattern doesn't exist in the text. However, if it does match, the pattern can be present in the text.

**History**

In computer science, the Rabin- Karp algorithm or Karp-Rabin algorithm is created by Richard M. Karp and Michael O. Rabin (1987) that uses a hashing to find an exact match of a pattern string in a text. Richard Manning Karp is an American computer scientist and computational theorist at the University of California, Berkeley. He is most notable for his research in the theory of algorithms, for which he received a Turing Award in 1985, The Benjamin Franklin Medal in Computer and Cognitive Science in 2004, and the Kyoto Prize in 2008. Michael Oser Rabin is an Israeli mathematician and computer scientist and a recipient of the Turing Award. It has very fast speed to compare the longest string. This type of string compare algorithm is very useful for something like detecting plagiarism as it can quickly go through long sets of strings and see how many matches there are. As time this algorithm it came from string and data uses people use data in large amount and copy each other’s data.

**Why Rabin-Karp Algorithm Invented:**

The basic principle employed in Rabin Karp algorithm is hashing. In the given text every substring is converted to a hash value and compared with the hash value of the pattern. This is an algorithm that uses hashing to find patterns between two strings. Compared to other algorithms that look at large chunks of a string at a time to speed up the comparisons, such as Knuth-Morris-Pratt and Boyer-Moore string-search algorithm, the Rabin-Karp algorithm uses hashing to increase the speed of the comparison. This type of string compare algorithm is very useful for something like detecting plagiarism as it can quickly go through long sets of strings and see how many matches there are. The Rabin-Karp algorithm is one of the best plagiarism detection algorithms and can run very and simply if a strong hash function is used. After this algorithm the string-matching patterns are easy to compare and solve it. This algorithm helps the problems which are previously unsolvable and using this algorithm existing problems are get easy to solvable.

**Algorithm**

This algorithm makes use of elementary number-theoretic notions such as the equivalence of two numbers modulo a third number. For expository purposes, let us assume that Σ = {0,1, 2…,9}, so that each character is a decimal digit. We can then view a string of k consecutive characters as representing a length-k decimal number. The character string 3 1 4 1 5 thus corresponds to the decimal number 31,415. Because we interpret the input characters as both graphical symbols and digits, we find it convenient in this section to denote them as we would digits, in our standards text font. Given a pattern P[1..m], let p denote its corresponding decimal value. In a similar manner, given a text T[1..n], let Ts denote the decimal value of the length-m substring T[s+1..s+m], for s = 0, 1,…n-m. Certainly, Ts = p if and only if T[s+1..s+m] = P[1..m]; thus, s is a valid shift if and only if Ts = p. If we could computer p in time Θ(m) and all the Ts values in a total of Θ (n-m+10 = Θ(n) by comparing p with each of the Ts values.

(For the moment, let’s not worry about the possibility that p and the Ts values might be very large numbers. We can computer p in time Θ(m) using Horner’s rule:

P = P[m] + 10(P[m-1] +10(P[m-2] +…+10(P [2] +10P [1]) …))

Similarly, we can compute t0 from T[1..m] in time Θ(m).

If we precompute the constant 10m-1 (which we can do in time O (lg m) using the techniques of above example although for this application a straightforward O(m) time method suffices), then each execution of equation above example takes a constant number of arithmetic operations. Thus, we can compute p in time Θ(m), and we can compute all t0, t1, …, tn-m in time Θ(n-m+1). Therefore, we can find all occurrences of the pattern P[1..m] in the text T[1..n] with Θ(m) preprocessing time and Θ(n-m+1) matching time.

**The String-Matching problem the algorithm solve:**

A screenshot of a computer

Description automatically generated with low confidence

Al, Cormen Thomas H et. *Introduction to Algorithms*. MIT Press, 2009.

Above Figure the Rabin-Karp algorithm. Each character is a decimal digit, and we compute values modulo 13. (a) A text string. A window of length 5 is shown shaded. The numerical value of the shaded number, computed modulo 13, yields the value 7. (b) The same text string with values com- puted modulo 13 for each possible position of a length-5 window. Assuming the pattern P D 31415, we look for windows whose value modulo 13 is 7, since 31415 approximately 7 (mod 13). The algorithm finds two such windows, shown shaded in the figure. The first, beginning at text position 7, is indeed an occurrence of the pattern, while the second, beginning at text position 13, is a spurious hit. (c) How to compute the value for a window in constant time, given the value for the previous window. The first window has value 31415. Dropping the high-order digit 3, shifting left (multiplying by 10), and then adding in the low-order digit 2 gives us the new value 14152. Because all computations are performed modulo 13, the value for the first window is 7, and the value for the new window is 8.

**RABIN-KARP-MATCHER (T, P, d, q)**

1. n ← length [T]

2. m ← length [P]

3. h ← dm-1 mod q

4. p ← 0

5. t0 ← 0

6. for i ← 1 to m

7. do p ← (dp + P[i]) mod q

8. t0 ← (dt0+T [i]) mod q

9. for s ← 0 to n-m

10. do if p = ts

11. then if P [1.....m] = T [s+1.....s + m]

12. then "Pattern occurs with shift" s

13. If s < n-m

14. then ts+1 ← (d (ts-T [s+1]h)+T [s+m+1])mod q

The procedure RABIN-KARP-MATCHER works as follows. All characters are interpreted as radix-d digits. The subscripts on t are provided only for clarity; the program works correctly if all the subscripts are dropped. Line 3 initializes h to the value of the high-order digit position of an m-digit window. Lines 4–8 compute p as the value of P[1..m] mod q and t0 as the value of T[1..m] mod q. The for loop of lines 9–14 iterates through all possible shifts s, maintaining the following invariant: Whenever line10 is executed, ts = T[s+1..s+m] mod q. If p = ts in line 10 (a “hit”), then line 11 checks to see whether P[1..m] = T[s + 1..s+m] in order to rule out the possibility of a spurious hit. Line 12 prints out any valid shifts that are found. If s < n-m (checked in line 13), then the for loop will execute at least one more time, and so line 14 first executes to ensure that the loop invariant holds when we get back to line 10. Line 14 computes the value of ts+1 mod q from the value of ts mod q in constant time using equation directly.

**Hash function**

A hash function is any function that can be used to map data of arbitrary size to fixed-size values. The values returned by a hash function are called hash values, hash codes, digests, or simply hashes. The values are usually used to index a fixed-size table called a hash table. Use of a hash function to index a hash table is called hashing or scatter storage addressing.Hash functions and their associated hash tables are used in data storage and retrieval applications to access data in a small and nearly constant time per retrieval. They require an amount of storage space only fractionally greater than the total space required for the data or records themselves. Hashing is a computationally and storage space-efficient form of data access that avoids the non-linear access time of ordered and unordered lists and structured trees, and the often-exponential storage requirements of direct access of state spaces of large or variable-length keys.

Use of hash functions relies on statistical properties of key and function interaction: worst-case behavior is intolerably bad with a vanishingly small probability, and average-case behavior can be nearly optimal (minimal collision).

**Applications**

A practical application of the algorithm is detecting plagiarism. Given source material, the algorithm can rapidly search through a paper for instances of sentences from the source material, ignoring details such as case and punctuation. Because of the abundance of the sought strings, single string searching algorithms are impractical. Karp-Rabin algorithm is still in use for searching string or matching pattern. Rabin-Karp is another pattern searching algorithm to find the pattern in a more efficient way. It also checks the pattern by moving window one by one, but without checking all characters for all cases, it finds the hash value. When the hash value is matched, then only it tries to check each character. This procedure makes the algorithm more efficient.

**Complexity**

The time complexity is O(m+n), but for the worst case, it is O(mn). For text of length ‘n’ and ‘p’ patterns of combined length ‘m’, It’s average and best-case running times is O(n+m) in space O(p), but its worst case is O(nm). This type of string compare algorithm is very useful for something like detecting plagiarism as it can quickly go through long sets of strings and see how many matches there are. The current applications are for pattern matching and for searching string in a bigger text. This algorithm is still in use and some of the methods of this algorithm have updated with never functions that helps to get the string or pattern matching more powerful and shows the better and deep results. Thus the complexity will change if the system and methods are updated or newer which is depend upon all of this.

**Rabin-Karp Algorithm for Pattern Searching**

Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m.

**Example**: Input: txt[] = “My Name Is Mangesh Raut”

pat[] = “Mangesh”

Output: Pattern Found at index 11

Input: txt[] = “ GREATGRANDMOMGRANDMOM”

pat[] = “MOM”

Output: Pattern found at index 10

Pattern found at index 18

The Naive String-Matching algorithm slides the pattern one by one. After each slide, it one by one checks characters at the current shift and if all characters match then prints the match.

Like the Naive Algorithm, Rabin-Karp algorithm also slides the pattern one by one. But unlike the Naive algorithm, Rabin Karp algorithm matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters. So Rabin Karp algorithm needs to calculate hash values for following strings.

1) Pattern itself.

2) All the substrings of the text of length m.

Since we need to efficiently calculate hash values for all the substrings of size m of text, we must have a hash function which has the following property.

**How the rolling hash works:**

Hash at the next shift must be efficiently computable from the current hash value and next character in text or we can say hash(txt[s+1 .. s+m]) must be efficiently computable from hash(txt[s .. s+m-1]) and txt[s+m] i.e., hash(txt[s+1 .. s+m])= rehash(txt[s+m], hash(txt[s .. s+m-1])) and rehash must be O(1) operation.

The hash function suggested by Rabin and Karp calculates an integer value. The integer value for a string is the numeric value of a string.

For example, if all possible characters are from 1 to 10, the numeric value of “122” will be 122. The number of possible characters is higher than 10 (256 in general) and pattern length can be large. So, the numeric values cannot be practically stored as an integer. Therefore, the numeric value is calculated using modular arithmetic to make sure that the hash values can be stored in an integer variable (can fit in memory words). To do rehashing, we need to take off the most significant digit and add the new least significant digit for in hash value. Rehashing is done using the following formula.

hash( txt[s+1 .. s+m] ) = ( d ( hash( txt[s .. s+m-1]) – txt[s]\*h ) + txt[s + m] ) mod q

hash( txt[s .. s+m-1] ) : Hash value at shift s.

hash( txt[s+1 .. s+m] ) : Hash value at next shift (or shift s+1)

d: Number of characters in the alphabet

q: A prime number

h: d^(m-1)

This is simple mathematics, we compute decimal value of current window from previous window.

For example pattern length is 3 and string is “56789”

You compute the value of first window (which is “567”) as 567.

How will you compute value of next window “678”? You will do (567 – 5\*100)\*10 + 8 and get 678.

**Conclusion**

This algorithm is used in detecting plagiarism. Given source material, the algorithm can rapidly search through a paper for instances of sentences from the source material, ignoring details such as case and punctuation. Because of the abundance of the sought strings, single string searching algorithms are impractical here. If we want to find any of the large number, say k, fixed length patterns in a text, we can create a simple variant of the Rabin-Karp algorithm. The Rabin-Karp algorithm is a unique string-search algorithm because of its use of a rolling hashing function. It uses the window to implement each character and compare the hash values of the pattern wanted and the current window. This algorithm has lots of value for string comparisons and the way it uses a rolling hash function allows for it for the string differences.

**References**

Wikipedia November 15, 2021 Rabin–Karp algorithm Retrieved from URL <https://en.wikipedia.org/wiki/Rabin–Karp_algorithm>

GeeksforGeeks January 10, 2022 Rabin-Karp Algorithm for Pattern Searching Retrieved from URL <https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/>

Wikipedia January 07, 2022 Introduction to Algorithms Retrieved from URL <https://en.wikipedia.org/wiki/Introduction_to_Algorithms>

Wikipedia February 01, 2022 Hash function Retrieved from URL <https://en.wikipedia.org/wiki/Hash_function>