Rabin-Karp Algorithm for Pattern Searching

Given a text T**[0. . .n-1]** and a pattern P**[0. . .m-1]**, write a function search(char P[], char T[]) that prints all occurrences of P[] present in T[] using Rabin Karp algorithm. You may assume that **n > m**.

Examples:

Input: *T[] = "THIS IS A TEST TEXT", P[] = "TEST"*
Output: *Pattern found at index 10*
Input: *T[] = "AABAACAADAABAABA", P[] = "AABA"*
Output: *Pattern found at index 0*
Pattern found at index 9
Pattern found at index 12

Rabin-Karp Algorithm:

In the Naive String Matching algorithm, we check whether every substring of the text of the pattern's size is equal to the pattern or not one by one.

*Like the Naive Algorithm, the Rabin-Karp algorithm also check every substring. But unlike the Naive algorithm, the Rabin Karp algorithm matches the **hash value** of the **pattern** with the **hash value** of the 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 the following strings.*

- ***Pattern** itself*
- *All the substrings of the **text** of length **m** which is the size of pattern.*

How is Hash Value calculated in Rabin-Karp?

Hash value is used to efficiently check for potential matches between a **pattern** and substrings of a larger **text**. The hash value is calculated using a **rolling hash function**, which allows you to update the hash value for a new substring by efficiently removing the contribution of the old character and adding the contribution of the new character. This makes it possible to slide the pattern over the **text** and calculate the hash value for each substring without recalculating the entire hash from scratch.

Here's how the hash value is typically calculated in Rabin-Karp:

Step 1: Choose a suitable **base** and a **modulus**:

- Select a prime number '**p**' as the modulus. This choice helps avoid overflow issues and ensures a good distribution of hash values.

- Choose a base ‘**b**‘ (usually a prime number as well), which is often the size of the character set (e.g., 256 for ASCII characters).

Step 2: Initialize the hash value:

- Set an initial hash value ‘**hash**‘ to **0**.

Step 3: Calculate the initial hash value for the **pattern**:

- Iterate over each character in the **pattern** from **left** to **right**.
- For each character **‘c’** at position **‘i’**, calculate its contribution to the hash value as **‘c * ($b_{pattern\_length - i - 1}$) % p’** and add it to ‘**hash**‘.
- This gives you the hash value for the entire **pattern**.

Step 4: Slide the pattern over the **text**:

- Start by calculating the hash value for the first substring of the **text** that is the same length as the **pattern**.

Step 5: Update the hash value for each subsequent substring:

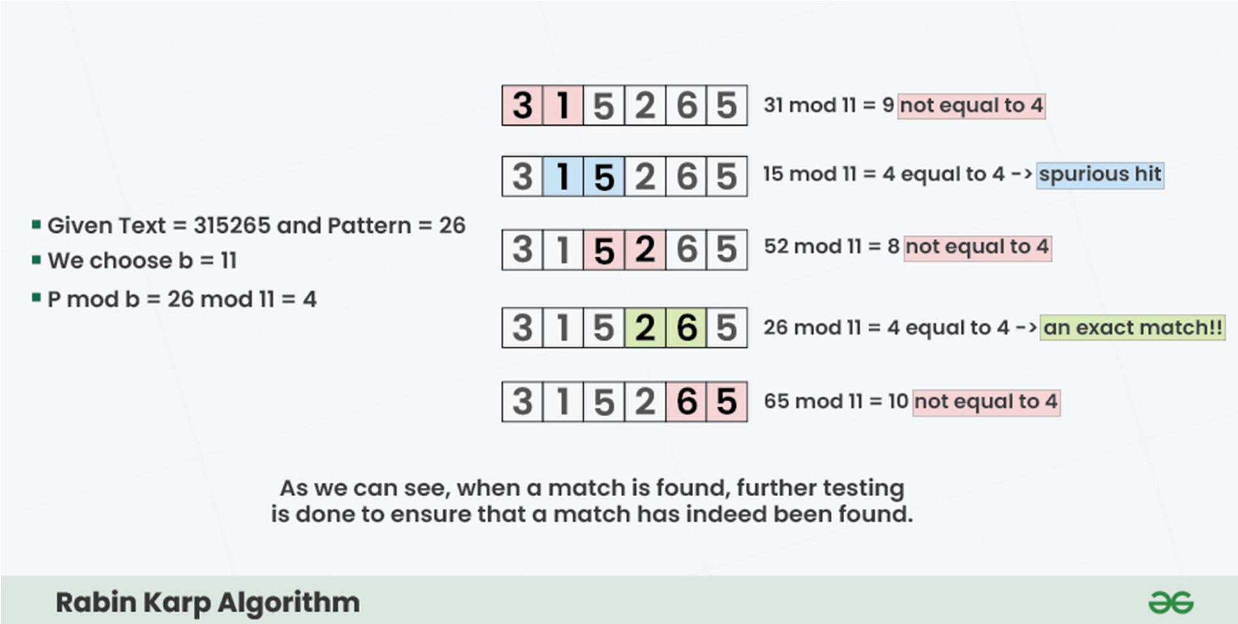
- To slide the **pattern** one position to the right, you remove the contribution of the leftmost character and add the contribution of the new character on the right.
- The formula for updating the hash value when moving from position **‘i’** to **‘i+1’** is:

`hash = (hash - (text[i - pattern_length] * (b`$_{pattern\_length - 1}$`)) % p) * b + text[i]`

Step 6: Compare hash values:

- When the hash value of a substring in the **text** matches the hash value of the **pattern**, it’s a **potential match**.
- If the hash values match, we should perform a character-by-character comparison to confirm the match, as hash collisions can occur.

Below is the Illustration of above algorithm:



Step-by-step approach:

- Initially calculate the hash value of the pattern.
- Start iterating from the starting of the string:
    - Calculate the hash value of the current substring having length **m**.
    - If the hash value of the current substring and the pattern are same check if the substring is same as the pattern.
    - If they are same, store the starting index as a valid answer. Otherwise, continue for the next substrings.
- Return the starting indices as the required answer.

Below is the implementation of the above approach:

C++

```
/* Following program is a C++ implementation of Rabin Karp
Algorithm given in the CLRS book */
#include <bits/stdc++.h>
using namespace std;

// d is the number of characters in the input alphabet
#define d 256
```

```
/* pat -> pattern
    txt -> text
    q -> A prime number
*/
void search(char pat[], char txt[], int q)
{
    int M = strlen(pat);
    int N = strlen(txt);
    int i, j;
    int p = 0; // hash value for pattern
    int t = 0; // hash value for txt
    int h = 1;

    // The value of h would be "pow(d, M-1)%q"
    for (i = 0; i < M - 1; i++)
        h = (h * d) % q;

    // Calculate the hash value of pattern and first
    // window of text
    for (i = 0; i < M; i++) {
        p = (d * p + pat[i]) % q;
        t = (d * t + txt[i]) % q;
    }

    // Slide the pattern over text one by one
    for (i = 0; i <= N - M; i++) {
```

```
        // Check the hash values of current window of text
        // and pattern. If the hash values match then only
        // check for characters one by one
        if (p == t) {
            /* Check for characters one by one */
            for (j = 0; j < M; j++) {
                if (txt[i + j] != pat[j]) {
                    break;
                }
            }

            // if p == t and pat[0...M-1] = txt[i, i+1,
            // ...i+M-1]

            if (j == M)
                cout << "Pattern found at index " << i
                     << endl;
        }

        // Calculate hash value for next window of text:
        // Remove leading digit, add trailing digit
        if (i < N - M) {
            t = (d * (t - txt[i] * h) + txt[i + M]) % q;

            // We might get negative value of t, converting
            // it to positive
```

```
            if (t < 0)
                t = (t + q);
        }
    }
}

/* Driver code */
int main()
{
    char txt[] = "GEEKS FOR GEEKS";
    char pat[] = "GEEK";

    // we mod to avoid overflowing of value but we should
    // take as big q as possible to avoid the collison
    int q = INT_MAX;

    // Function Call
    search(pat, txt, q);
    return 0;
}
```

Output

```
Pattern found at index 0
Pattern found at index 10
```

Time Complexity:

- The average and best-case running time of the Rabin-Karp algorithm is O(n+m), but its worst-case time is O(nm).

- The worst case of the Rabin-Karp algorithm occurs when all characters of pattern and text are the same as the hash values of all the substrings of T[] match with the hash value of P[].

Auxiliary Space: O(1)