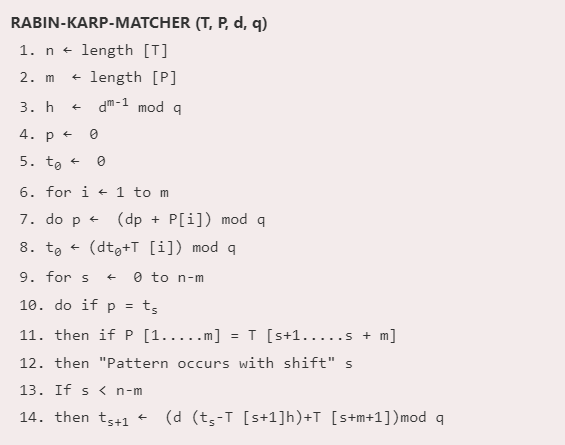
**NAME: ADITYA ABHIJIT PARULEKAR**

**DIV: S2-1**

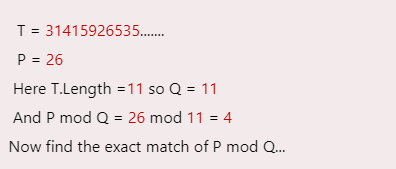
**ROLL.NO: 2201072**

* **Experiment 13:**
* **AIM: To study and implement Rabin-Karp Algorithm for Pattern Searching.**
* **THEORY:**

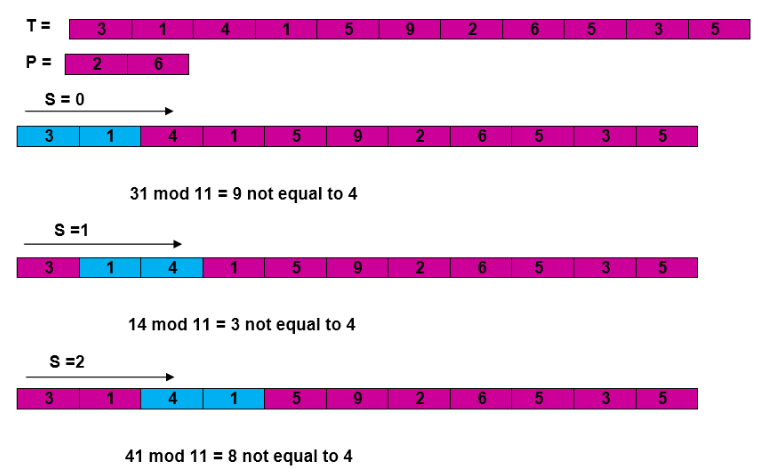
The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character subsequences of text to be compared. If the hash values are unequal, the algorithm will determine the hash value for next M-character sequence. If the hash values are equal, the algorithm will analyze the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and character matching is only required when the hash values match.

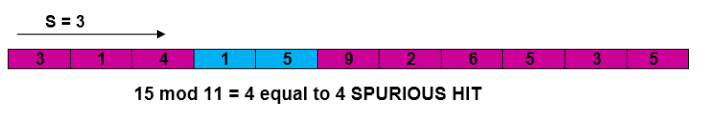


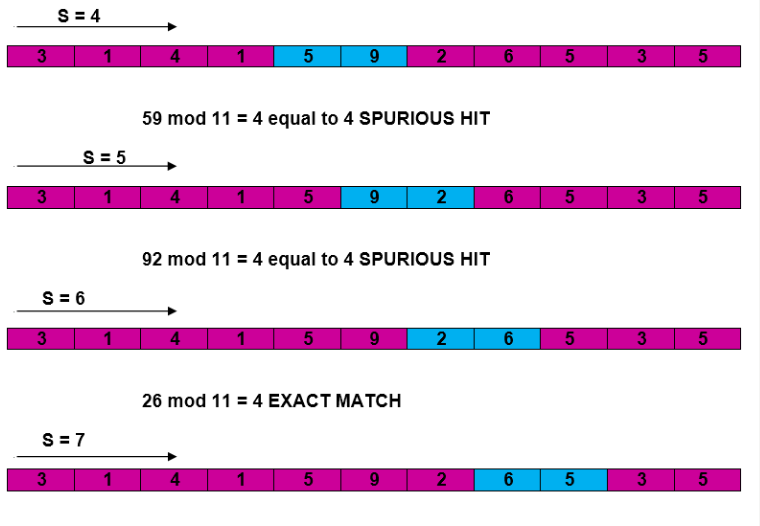
**Example:** For string matching, working module q = 11, how many spurious hits does the Rabin-Karp matcher encounters in Text T = 31415926535.......

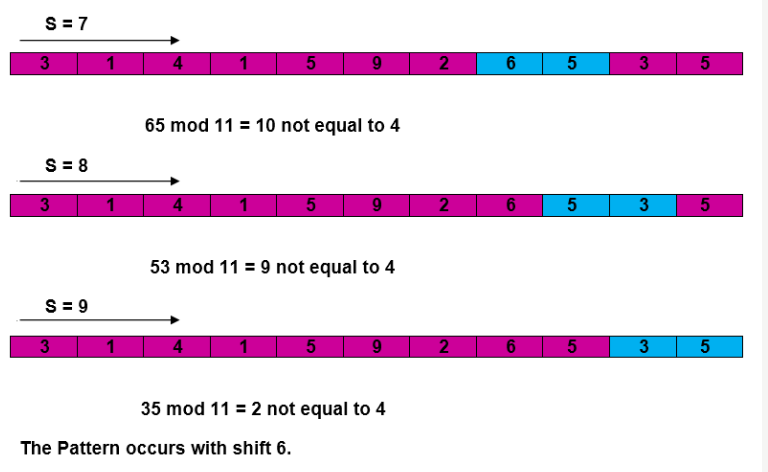


Solution:





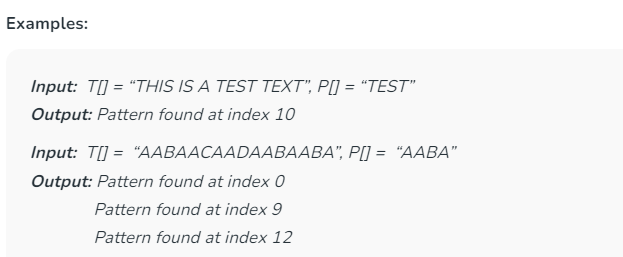




## Complexity:

The running time of **RABIN-KARP-MATCHER** in the worst case scenario **O ((n-m+1) m** but it has a good average case running time. If the expected number of strong shifts is small **O (1)** and prime q is chosen to be quite large, then the Rabin-Karp algorithm can be expected to run in time **O (n+m)** plus the time to require to process spurious hits.

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**.



**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**:

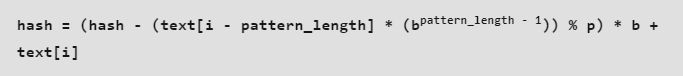
* Iterate over each character in the **pattern** from **left** to **right**.
* For each character **‘c’**at position **‘i’**, calculate its contribution to the hash value as **‘c \* (bpattern\_length – i – 1) % p’** and add it to ‘**hash**‘.
* This gives you the hash value for the entire **pattern**.

**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:

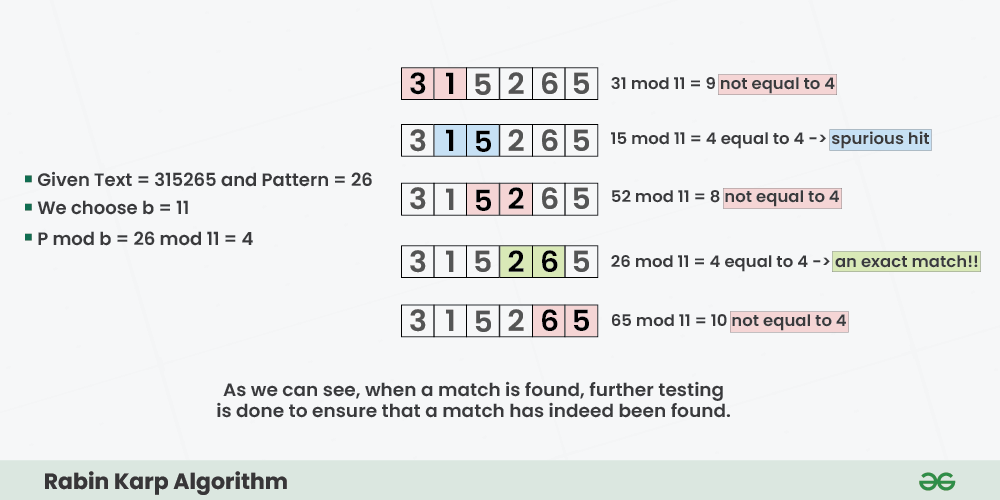
* To slide the **pattern** one position to the right, you remove the contribution of the leftmost character and add the contribution of the new character on the right.
* The formula for updating the hash value when moving from position**‘i’** to **‘i+1’**is:



**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](https://www.geeksforgeeks.org/what-is-hashing/) 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.
* **C PROGRAM:**

#include <stdio.h>

#include <string.h>

#define d 256

void search(char pat[], char txt[], int q)

{

int M = strlen(pat);

int N = strlen(txt);

int i, j;

int p = 0;

int t = 0;

int h = 1;

for (i = 0; i < M - 1; i++)

h = (h \* d) % q;

for (i = 0; i < M; i++) {

p = (d \* p + pat[i]) % q;

t = (d \* t + txt[i]) % q;

}

for (i = 0; i <= N - M; i++) {

if (p == t) {

for (j = 0; j < M; j++) {

if (txt[i + j] != pat[j])

break;

}

if (j == M)

printf("Pattern found at index %d\n", i);

}

if (i < N - M) {

t = (d \* (t - txt[i] \* h) + txt[i + M]) % q;

if (t < 0)

t = (t + q);

}

}

}

int main()

{

char txt[100];

char pat[100];

int q;

printf("ADITYA PARULEKAR-S21-2201072\n");

printf("------------- RABIN-KARP ALGORITHM -------------\n\n");

printf("Enter the text: ");

scanf("%s", txt);

printf("Enter the pattern: ");

scanf("%s", pat);

printf("Enter a prime number: ");

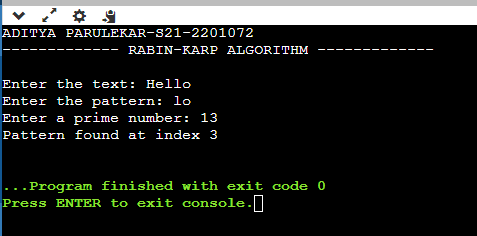
scanf("%d", &q);

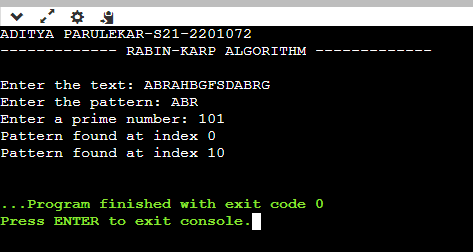
search(pat, txt, q);

return 0;

}

* **OUTPUT:**





* **CONCLUSION: Hence, we have successfully implemented Rabin-Karp Algorithm for Pattern Searching; LO 1, LO 2.**