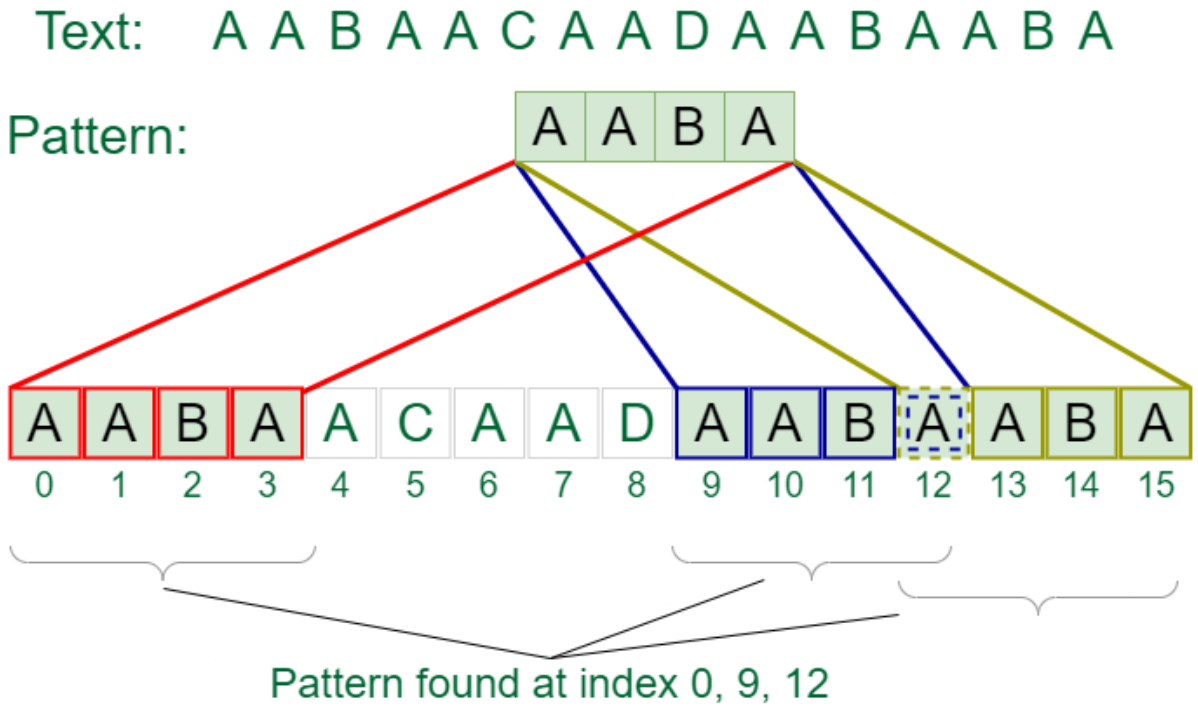
NAME:	Prakhar Gupta
UID:	2021300040
SUBJECT	Design and Analysis of Algorithm
EXPERIMENT NO :	10
DATE OF PERFORMANCE	10/04/2023
DATE OF SUBMISSION	17/04/2023
AIM:	Solve string matching problem by Rabin-Karp algorithm
PROBLEM STATEMENT 1:	String Pattern Matching
ALGORITHM and THEORY:	Given a text **txt[0. . .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**. **Examples:** ***Input:*** *txt[] = "THIS IS A TEST TEXT", pat[] = "TEST"* ***Output:*** *Pattern found at index 10* ***Input:*** *txt[] = "AABAACAADAABAABA", pat[] = "AABA"* ***Output:*** *Pattern found at index 0* *Pattern found at index 9* *Pattern found at index 12*



Rabin-Karp algorithm

Approach: To solve the problem follow the below idea:
The Naive String Matching algorithm slides the pattern one by one. After each slide, one by one checks characters at the current shift, and if all characters match then print the match
Like the Naive Algorithm, the Rabin-Karp algorithm also slides the pattern one by one. 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*

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

- *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.*

Note: 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 the 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:

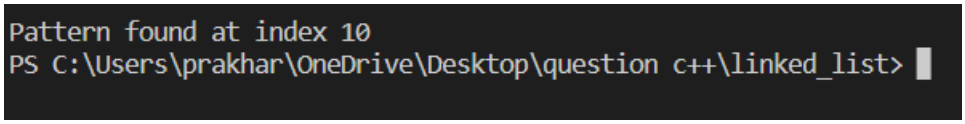
*hash(txt[s+1 .. s+m]) = (d (hash(txt[s .. s+m-1]) – txt[s]*h) + txt[s + m]) mod q*
hash(txt[s .. s+m-1]) : Hash value at shift s
hash(txt[s+1 .. s+m]) : Hash value at next shift (or shift s+1)
d: Number of characters in the alphabet
q: A prime number
h: $d^{(m-1)}$
How does the above expression work?
This is simple mathematics, we compute the decimal value of the current window from the previous window.
Example: pattern length is 3 and string is "23456"
You compute the value of the first window (which is "234") as 234.
How will you compute the value of the next window "345"? You will do (234 – 2*100)*10 + 5 and get 345.
Follow the steps mentioned here to implement the idea:

- 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:
PROGRAM:	

```cpp
/* 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;
}
```

	`// This is code is contributed by rathbhupendra`
OUTPUT:	
CONCLUSION:	By performing the above experiment I was able to implement the String Pattern Matching Problem and Print the index where it get matched for first time