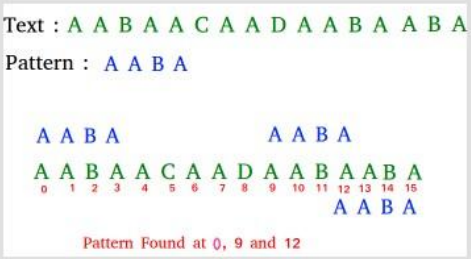
Rabin-Karp Algorithm for pattern Searching

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.

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
```



The *Naive String Matching* algorithm slides the pattern one by one. After each slide, one by one it checks characters at the current shift and if all characters match then it 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, that is of the length of pattern string.

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.

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 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*

Where,
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)*

C/C++ Java

```c
/* Following program is a C/C++ implementation of Rabin Karp Algorithm */
#include<stdio.h>
#include<string.h>
// 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++)
    {
```

C/C++ Java

```
        // Check the hash values of current window of text
        // and pattern. If the hash values match then only check for characters on 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)
                printf("Pattern found at index %d \n", i);
        }
// 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 program to test above function */
int main()
{
    char txt[] = "GEEKS FOR GEEKS";
    char pat[] = "GEEK";
    int q = 101; // A prime number
    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). Worst case of Rabin-Karp algorithm occurs when all characters of pattern and text are the same as the hash values of all the substrings of txt[] match with the hash value of pat[]. For example pat[] = "AAA" and txt[] = "AAAAAAA".