Name:	Prerna Sunil Jadhav
Sap Id:	60004220127
Class:	S. Y. B.Tech (Computer Engineering)
Course:	Analysis of Algorithm Laboratory
Course Code:	DJ19CEL404
Experiment No.:	10

AIM: **TO IMPLEMENT STRING MATCHING USING RABIN KARP AND KMP ALGORITHM.**

THEORY:

RABIN KARP STRING MATCHING

- 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
- **Algorithm:**

```
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.
```

CODE:

```
#include <stdio.h>
#include <string.h>
int d = 23;
void search(char P[], char T[], int q)
{
    int m = strlen(P);
    int n = strlen(T);
    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 + P[i]) % q;
        t = (d * t + T[i]) % q;
    }
    for (i = 0; i <= n - m; i++)
    {
        if (p == t)
        {
            for (j = 0; j < m; j++)
            {
                if (T[i + j] != P[j])
                    break;
            }
            if (j == m)
                printf("Pattern found at index %d \n", i);
        }
        if (i < n - m)
        {
            t = (d * (t - T[i] * h) + T[i + m]) % q;
            if (t < 0)
                t = (t + q);
        }
    }
}
int main()
{
    char P[200], T[200];
    printf("Enter the text: \n");
    gets(T);
    printf("The text is %s \n", T);
    printf("Enter the pattern: \n");
    gets(P);
    printf("The pattern is %s \n", P);

    int q = 13;

    search(P, T, q);
    return 0;
}
```

OUTPUT:

```
exe' '--interpreter=mi'
Enter the text:
preftgrhhpregffpremkg
The text is preftgrhhpregffpremkg
Enter the pattern:
pregff
The pattern is pregff
Pattern found at index 9
PS C:\Users\Jadhav\Desktop\BTech\4th sem\AOA\Prac\Code>
```

KMP STRING MATCHING

- The KMP matching algorithm uses degenerating property (pattern having the same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst-case complexity to O(n).
- The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window.
- We take advantage of this information to avoid matching the characters that we know will anyway match.
- **Algorithm:**

```
Pre-processing overview:
• KMP algorithm preprocesses pat[] and constructs an auxiliary lps[] of size m
(same as the size of the pattern) which is used to skip characters while
matching.
• name lps indicates the longest proper prefix which is also a suffix. A
proper prefix is a prefix with a whole string not allowed. For example,
prefixes of "ABC" are "", "A",
"AB" and "ABC". Proper prefixes are "", "A" and "AB". Suffixes of the string
are "", "C", "BC", and "ABC".
• We search for lps in sub-patterns. More clearly we focus on sub-strings of
patterns that are both prefix and suffix.
• For each sub-pattern pat[0..i] where i = 0 to m-1, lps[i] stores the length
of the maximum matching proper prefix which is also a suffix of the sub-
pattern pat[0..i]

Pre-processing:
• We calculate values in lps[]. To do that, we keep track of the length of the
longest prefix suffix value (we use len variable for this purpose) for the
previous index
• We initialize lps[0] and len as 0
• If pat[len] and pat[i] match, we increment len by 1 and assign the
incremented value to lps[i].
• If pat[i] and pat[len] do not match and len is not 0, we update len to
lps[len-1]
• See computeLPSArray () in the above code for details
```

String matching:
1. We start the comparison of pat[j] with j = 0 with characters of the current window of text.
2. We keep matching characters txt[i] and pat[j] and keep incrementing i and j while pat[j] and txt[i] keep matching.
3. When we see a mismatch
• We know that character's pat[0..j-1] match with txt[i-j...i-1] (Note that j starts with 0 and increments it only when there is a match).
• We also know (from the above definition) that lps[j-1] is the count of characters of pat[0...j-1] that are both proper prefix and suffix.
• From the above two points, we can conclude that we do not need to match these lps[j-1] characters with txt[i-j...i-1] because we know that these characters will anyway match. Let us consider the above example to understand this.

CODE:

```
#include <bits/stdc++.h>
using namespace std;
void computeLPSArray(char *pat, int M, int *lps);
void KMPSearch(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);
    int lps[M];
    int i = 0; // index for txt[]
    int j = 0; // index for pat[]
    while ((N - i) >= (M - j))
    {
        if (pat[j] == txt[i])
        {
            j++;
            i++;
        }
        if (j == M)
        {
            printf("Found pattern at index %d \n", i - j);
            j = lps[j - 1];
        }

        else if (i < N && pat[j] != txt[i])
        {
            if (j != 0)
                j = lps[j - 1];
            else
```

```
                i = i + 1;
        }
    }
}
void computeLPSArray(char *pat, int M, int *lps)
{
    int len = 0;
    lps[0] = 0; // lps[0] is always 0
    int i = 1;
    while (i < M)
    {
        if (pat[i] == pat[len])
        {
            len++;
            lps[i] = len;
            i++;
        }
        else // (pat[i] != pat[len])
        {
            if (len != 0)
            {
                len = lps[len - 1];
            }
            else // if (len == 0)
            {
                lps[i] = 0;
                i++;
            }
        }
    }
}
int main()
{
    char txt[25];
    char pat[5];
    cout << "Enter text:";
    cin >> txt;
    cout << "Enter pattern:";
    cin >> pat;
    KMPSearch(pat, txt);
    return 0;
}
```

OUTPUT:

```
Enter text:abdneif
Enter pattern:ne
Found pattern at index 3


...Program finished with exit code 0
Press ENTER to exit console.
```

CONCLUSION:

- Thus, we implemented the code for string matching using Rabin Karp algorithm and KMP algorithm.