Experiment 10

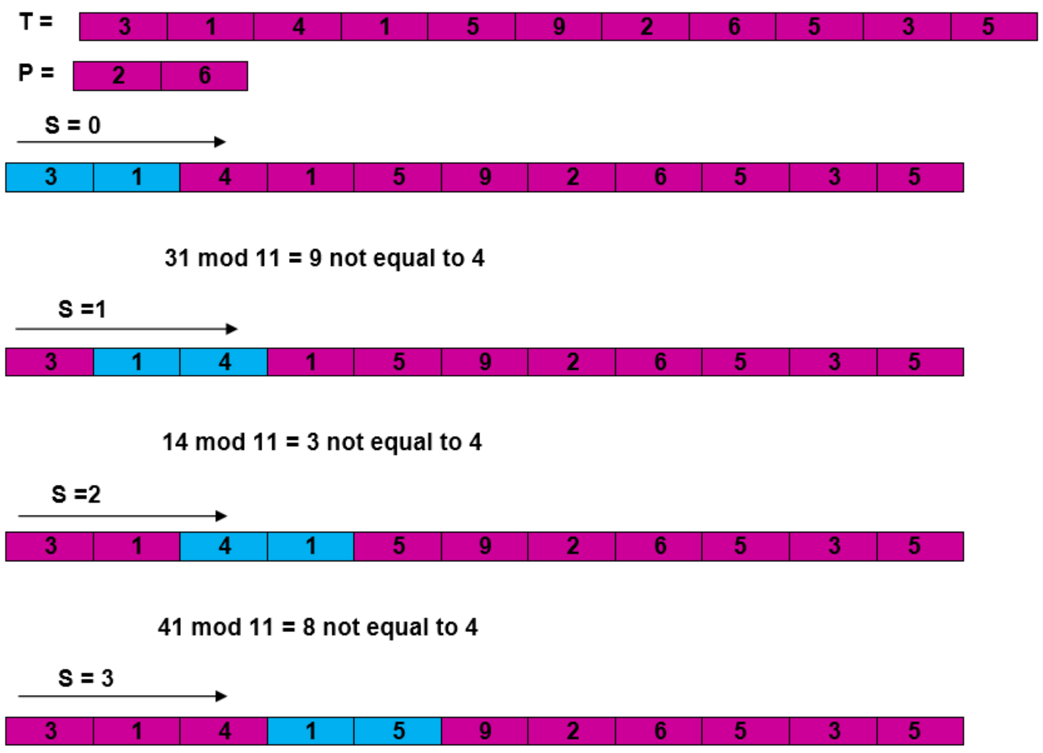
Aim: To implement string matching algorithms Rabin Karp and Knuth Morris Pratt.

Rabin Karp:

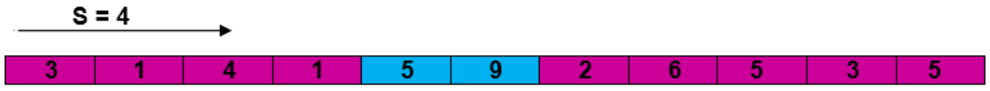
Theory: The Rabin-Karp algorithm is a string matching algorithm that searches for a given pattern in a text string by using hashing. The algorithm calculates a hash value for the pattern and for each substring of the text, and compares these hash values to determine whether the pattern matches the substring.

The Rabin-Karp algorithm is efficient for pattern matching because it avoids comparing each character of the pattern to each character of the text string. Instead, it compares the hash values, which are much faster to compute. However, the algorithm can have poor worst-case performance if the hash function produces many hash collisions.

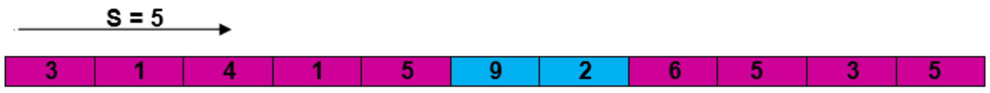
Example:


T = 3 1 4 1 5 9 2 6 5 3 5
P = 2 6
S = 0
3 1 4 1 5 9 2 6 5 3 5
31 mod 11 = 9 not equal to 4
S =1
3 1 4 1 5 9 2 6 5 3 5
14 mod 11 = 3 not equal to 4
S =2
3 1 4 1 5 9 2 6 5 3 5
41 mod 11 = 8 not equal to 4
S = 3
3 1 4 1 5 9 2 6 5 3 5

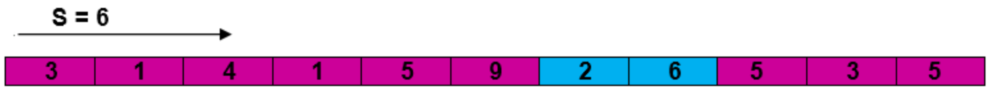
15 mod 11 = 4 equal to 4 SPURIOUS HIT



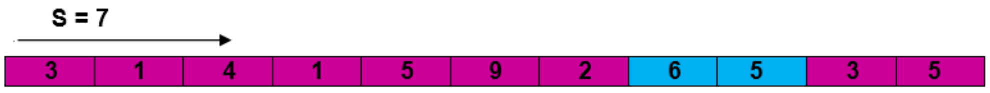
59 mod 11 = 4 equal to 4 SPURIOUS HIT



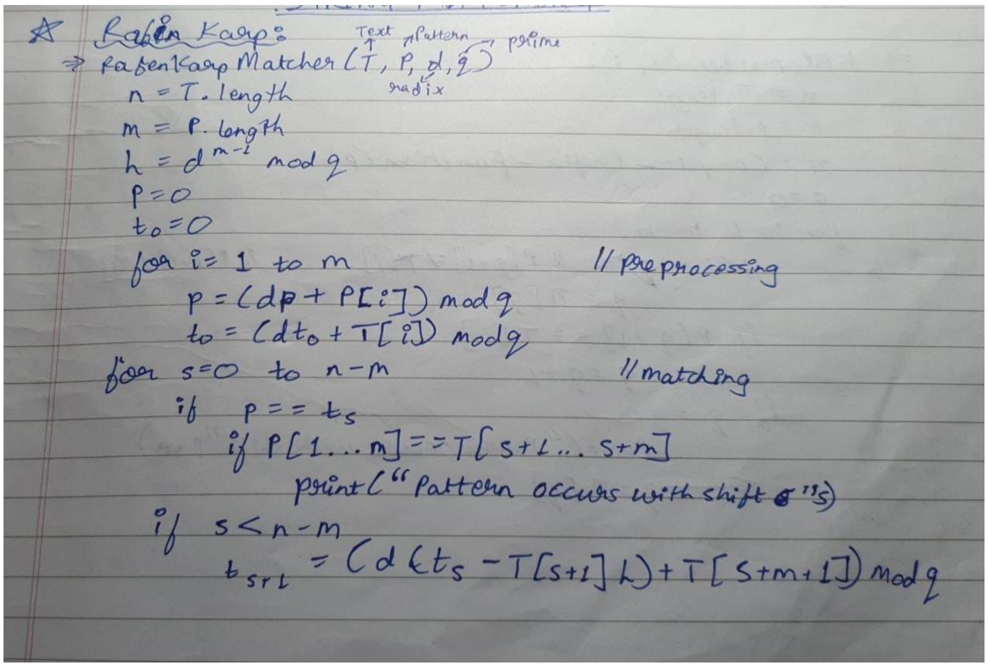
92 mod 11 = 4 equal to 4 SPURIOUS HIT



26 mod 11 = 4 EXACT MATCH



Algorithm:

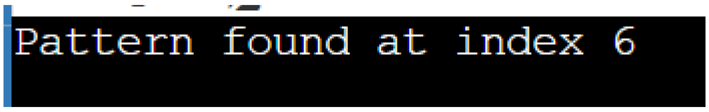


```
☆ Rabin Karp:          Text  Pattern  prime
⇒ RabinKarp Matcher (T, P, d, q)
                            radix
    n = T.length
    m = P.length
    h = d^{m-1} mod q
    p = 0
    t_0 = 0
    for i = 1 to m                           // preprocessing
        p = (dp + P[i]) mod q
        t_0 = (dt_0 + T[i]) mod q
  for s = 0 to n-m                            // matching
      if  p == t_s
          if P[1...m] == T[s+1... s+m]
              print("Pattern occurs with shift "s)
      if  s < n-m
          t_{s+1} = (d(t_s - T[s+1] h) + T[s+m+1]) mod q
```

Code:

```
#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);
    }
    }
}
void main()
{char txt[] = "hello good morning";
    char pat[] = "good";
    int q = 101;
    search(pat, txt, q);
}
```

Output:



```
Pattern found at index 6
```

Knuth Morris Pratt:

Theory: The Knuth-Morris-Pratt (KMP) algorithm is a string matching algorithm that searches for a given pattern in a text string by using a failure function. The algorithm calculates the failure function, which is an array of values that indicate the positions in the pattern where a prefix matches a suffix. The algorithm then uses the failure function to avoid unnecessary comparisons while searching for the pattern in the text.

The KMP algorithm has better worst-case performance than other string matching algorithms because it avoids comparing overlapping substrings of the pattern. It is particularly efficient when the pattern contains repeated prefixes.

Example:

P :

a	b	a	b	a	c	a

Step 1: q = 2, k = 0

Π [2] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0					

Step 2: q = 3, k = 0

Π [3] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1				

Step3: q =4, k =1

Π [4] = 2

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	A
π	0	0	1	2			

Step4: q = 5, k =2

Π [5] = 3

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3		

Step5: q = 6, k = 3

Π [6] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	0	

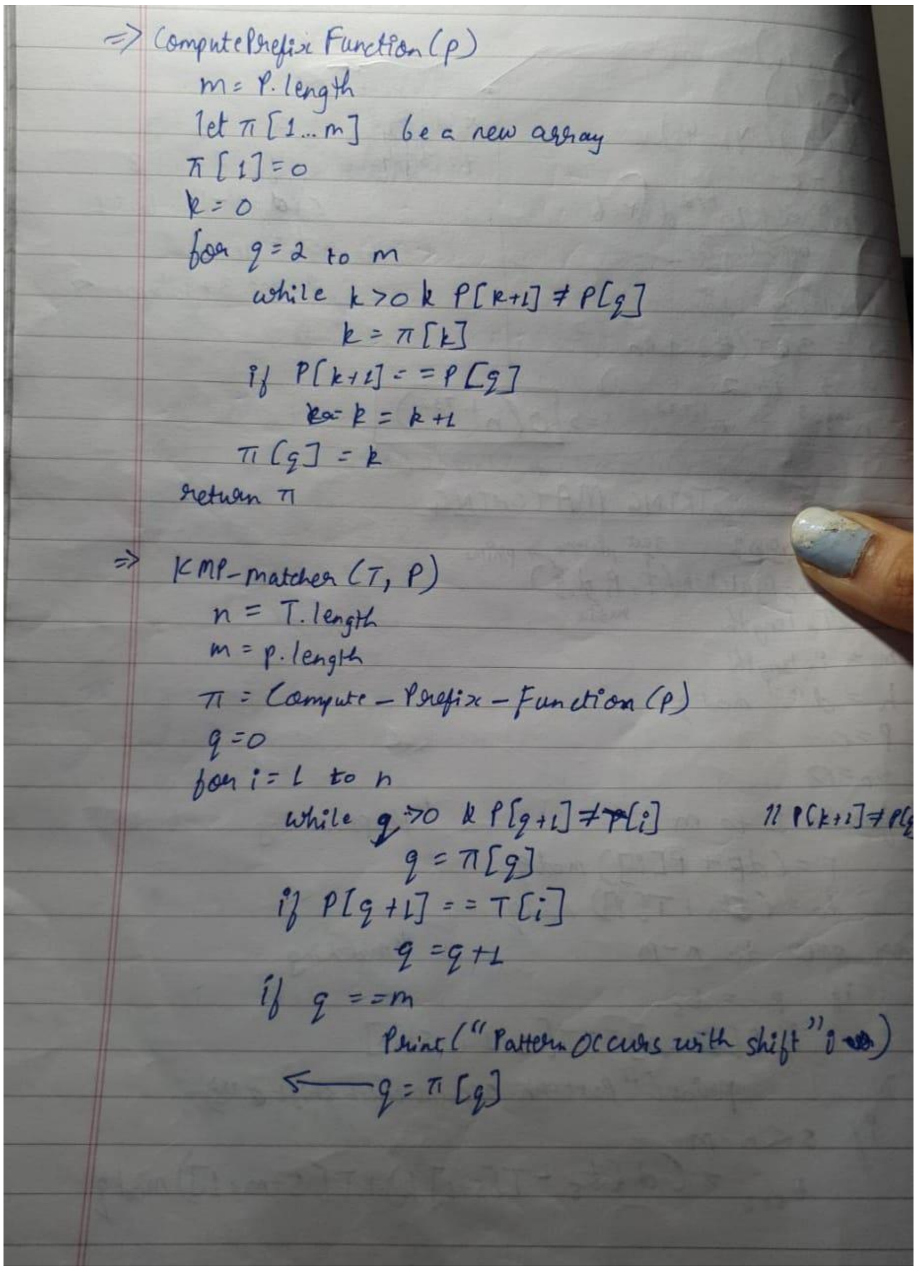
Step6: q = 7, k = 1

Π [7] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	0	1

After iteration 6 times, the prefix function computation is complete

Algorithm:



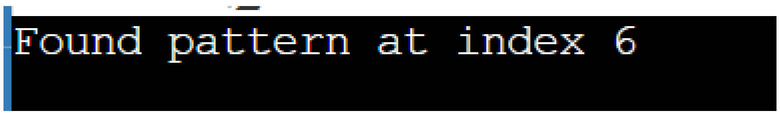
```
=> ComputePrefix Function(p)
     m = P.length
     let π[1...m] be a new array
     π[1] = 0
     k = 0
     for q = 2 to m
          while k > 0 & P[k+1] ≠ P[q]
               k = π[k]
          if P[k+1] = = P[q]
               k = k + 1
          π[q] = k
     return π

=> KMP-matcher(T, P)
     n = T.length
     m = p.length
     π = Compute-Prefix-Function(p)
     q = 0
     for i = 1 to n
          while q > 0 & P[q+1] ≠ [i]          // P[k+1] ≠ P[
               q = π[q]
          if P[q+1] = = T[i]
               q = q + 1
          if q = = m
               Print("Pattern occurs with shift" i)
     <——  q = π[q]
```

Code:

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
void computeLPSArray(char *pat, int M, int *lps);
void KMPSearch(char *pat, char *txt) {
   int M = strlen(pat);
   int N = strlen(txt);
   int *lps = (int *) malloc(sizeof(int) * M);
```

```
    int j = 0;
    computeLPSArray(pat, M, lps);
    int i = 0;
    while (i < N) {
        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;
        }
    }
    free(lps);
}
void computeLPSArray(char *pat, int M, int *lps) {
    int len = 0;
    int i;
    lps[0] = 0;
    i = 1;
    while (i < M){
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        }
        else
        {
            if (len != 0) {
                len = lps[len - 1];
            }
            else
            {
                lps[i] = 0;
                i++;
            }
        }
    }
}
void main() {
    char *txt = "hello good morning";
    char *pat = "good";
    KMPSearch(pat, txt);
}
```

Output:



```
Found pattern at index 6
```

Conclusion: Hence we have successfully implemented string matching algorithms Rabin Karp and Knuth Morris Pratt.