Sherwyn Fernandes
SE COMP-B
191105063

Experiment No: 15

AIM: Implementation of KMP pattern matching algorithm

THEORY:

In the classic pattern matching problem on strings, we are given a text string of length n and a pattern of length m and we want to find whether P is a substring of T. The notion of a "match" is that there is a substring of T starting from some index i that matches P, character by character, so that

$$T[i] = P[0], T[i+1] = P[1],\dots,T[i+m-1] = P[m-1]$$

That is

$$P = T[i\dots I + m -1]$$

Thus, the output from a pattern matching algorithm is either an indication that the pattern P does not exist int T or the starting index in T of a substring matches P. Two classic algorithms are discussed here, namely, the Brute Force algorithm and the Boyer Moore algorithm.

Knuth-Morris-Pratt Algorithm:

The main idea of the KMP algorithm is to pre-process the pattern string P so as to compute a failure function f that indicates the proper shift of P so that, to the largest extent possible, we can reuse previously performed comparisons.. Specifically, the failure function f(j) is defined as the length of the longest prefix of F that is a suffix of P[1..j]

We also use the convention that f(0) = O.

<u>The importance of this failure function is that it encodes repeated substrings inside the pattern itself</u>

Sherwyn Fernandes
SE COMP-B
191105063

The KMP pattern matching algorithm, increments and processes the text string T comparing it to the pattern string P. Each time there is a match, we increment the current indices. On the other hand, if there is a mismatch and we have previously made progress in P, then we consult the failure function to determine the new index in P where we need to continue checking P against T.

Otherwise (there was a mismatch, and we are at the beginning of P), we simply increment the index for T (and keep the index variable for P at its beginning and repeat this process until we find a match of P in T or the index for T reaches length of T (indicating that we did not find the pattern P in T)

The main part of the KMP algorithm is the while-loop, which performs a comparison between a character in T and a character in P in each iteration. Depending upon the outcome of this comparison, the algorithm either moves on to the characters in T and P, consults the failure function for a new candidate character in P, or starts over with the next index in T. The correctness of this algorithm follows from the definition of the failure function. The skipped comparisons are actually unnecessary, for the failure function guarantees that all the ignored comparisons are redundant, and they would involve comparing characters we already know match.

The KMP matching algorithm achieves a **running time of O(n+m)** which in the worst case is optimal. That is, in the worst case any pattern matching algorithm will have to examine all the characters of the text and all the characters of the pattern at least once.

KMP ALGORITHM:

```
Algorithm KMPMatch(T,P)
{
        f <- KMPFailureFunction(P) {construct the failure function f for P}
        i=0
        j=0
```

Sherwyn Fernandes
SE COMP-B
191105063

```
    while i<n do
        if P[j] = T[i] then
            if j=m-1 then
                return i-m+1        {a match!}

    else if j > 0 {no match, but we have advanced in P} then
            j = f(j-1)      {j indexes just after prefix of P that must match}
    else
        i = i+1

    return "There. is no substring of T matching P."

}
```

Algorithm KMPFailureFunction (P):

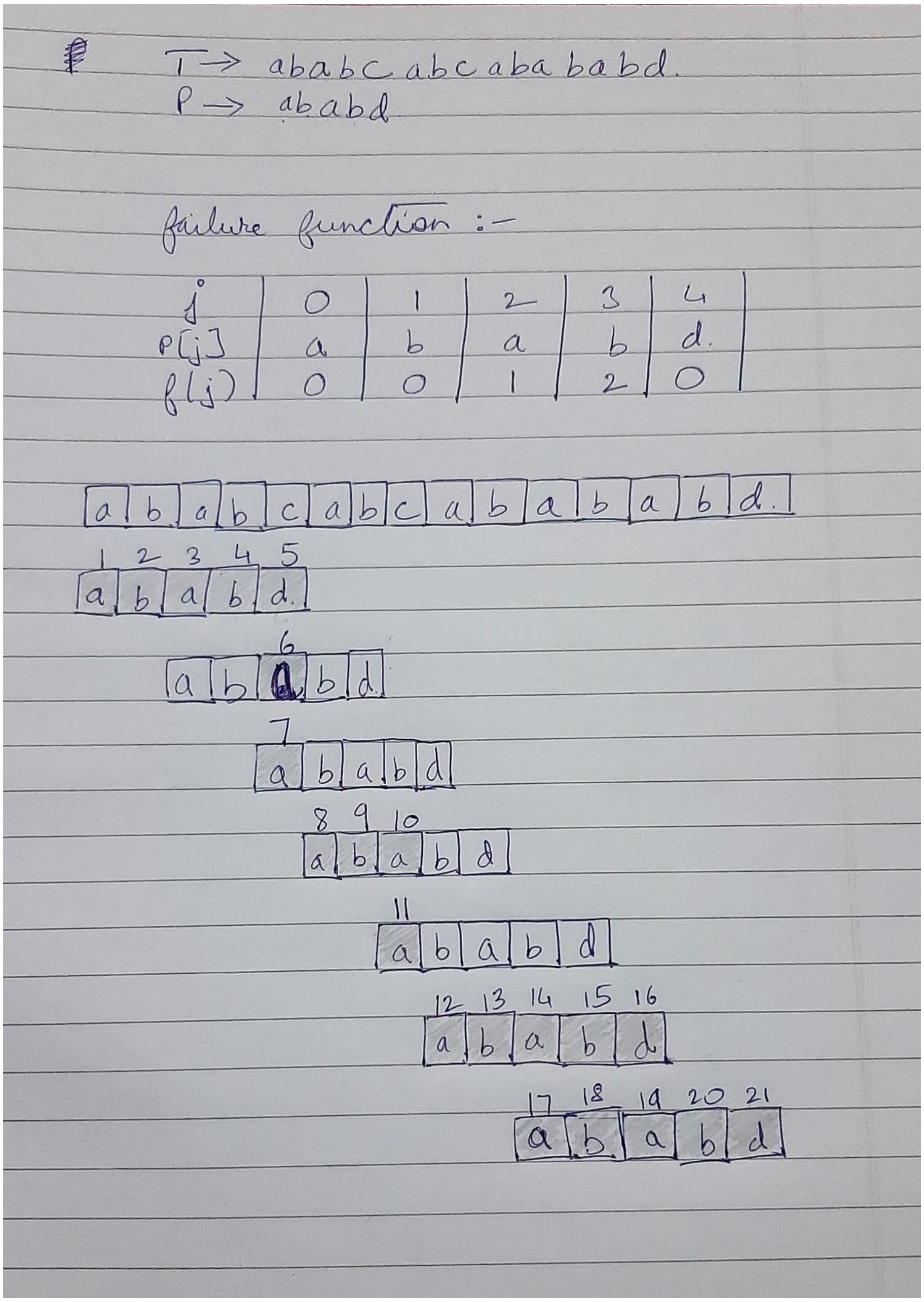
```
{
    f(0) = 0
    i=1
    j=0

    while i<m do '
        if P[j] =P[i] then            {we have matched j + 1 characters}
            f(i) = j+1
            j=j+1
            i=i+1

        else if j>0 then   {j indexes just after a prefix of P that must match}
            j = f(j-1)

        else                    {we have no match here}
            f(i) = 0
            i = i+1


}
```

Time Complexity

- The time complexity of this algorithm is O (m+n), where m is the size of the pattern and n is the size of the text

Sherwyn Fernandes
SE COMP-B
191105063

Problem Tracing



T → ababc abc aba babd.
P → ababd.

failure function :-

j	0	1	2	3	4
P[j]	a	b	a	b	d.
f(j)	0	0	1	2	0

| a | b | a | b | c | a | b | c | a | b | a | b | a | b | d. |

1 2 3 4 5
| a | b | a | b | d. |

6
| a | b | a | b | d |

7
| a | b | a | b | d |

8 9 10
| a | b | a | b | d |

11
| a | b | a | b | d |

12 13 14 15 16
| a | b | a | b | d |

17 18 19 20 21
| a | b | a | b | d |

Sherwyn Fernandes
SE COMP-B
191105063

PROGRAM IMPLEMENTATION:

```
#include<iostream>

using namespace std;

char p[20];


int * failure(char *p,int m)

{

        int *f = new int[m];


        int j=0,i=1;

        f[0] = 0;

        while(i<m)

        {

                if(p[j] == p[i])

                        {

                                f[i] = j+1;

                                j = j+1;

                                i= i+1;

                        }


                else if(j>0)

                        j = f[j-1];


                else
```

Sherwyn Fernandes
SE COMP-B
191105063

```
                {
                        i = i+1;
                        f[i] = 0;
                }
        }

        return f;
}



int KMP_match(char *t)
{
        int n = strlen(t);
        int m = strlen(p);

        int *f = failure(p,m);

        int i=0,j=0;

        while(i<n)
        {
                if(p[j] == t[i])
                {
                        if(j == m-1)
                                return i - m+1;
```

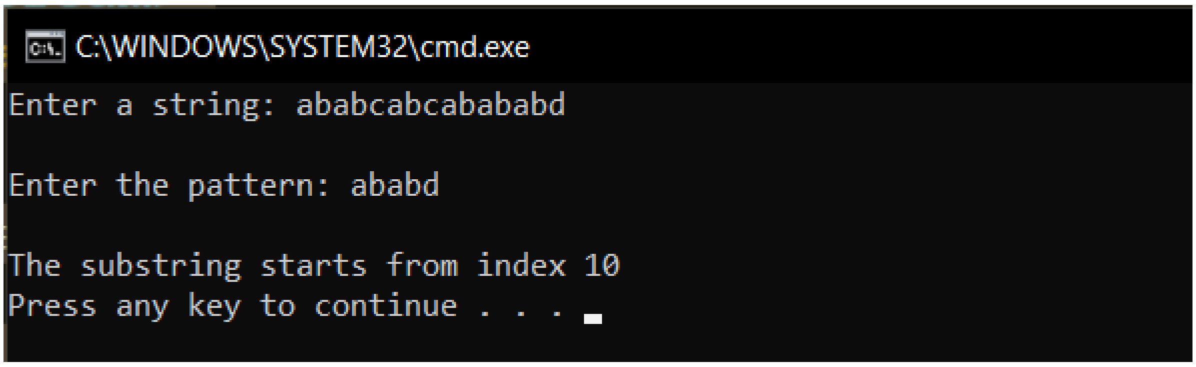
Sherwyn Fernandes
SE COMP-B
191105063

```
                    else

                    {

                              i = i+1;

                              j=j+1;

                    }

          }


          else if(j>0)

                    j = f[j-1];


          else

                    i=i+1;


     }


     return -1;


}


int main()

{

     char t[20];

     int i;

     cout<<"Enter a string: ";
```

Sherwyn Fernandes
SE COMP-B
191105063

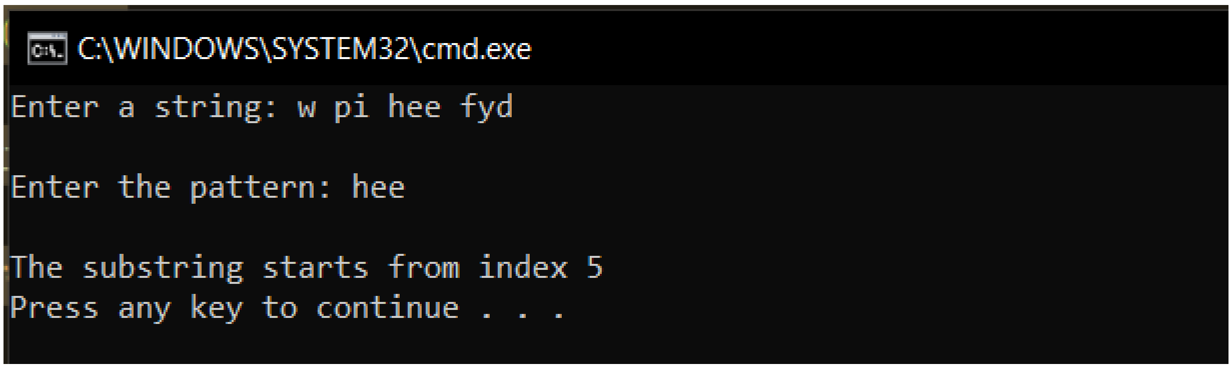
```
    cin.getline(t,20);

    cout<<"\nEnter the pattern: ";

    cin.getline(p,20);


    i = KMP_match(t);

    if(i!=-1)

        cout<<"\nThe substring starts from index "<<i<<endl;


    else

        cout<<"\nThere is no substring of T matching P\n";

    return 0;


}
```

OUTPUTS:



C:\WINDOWS\SYSTEM32\cmd.exe

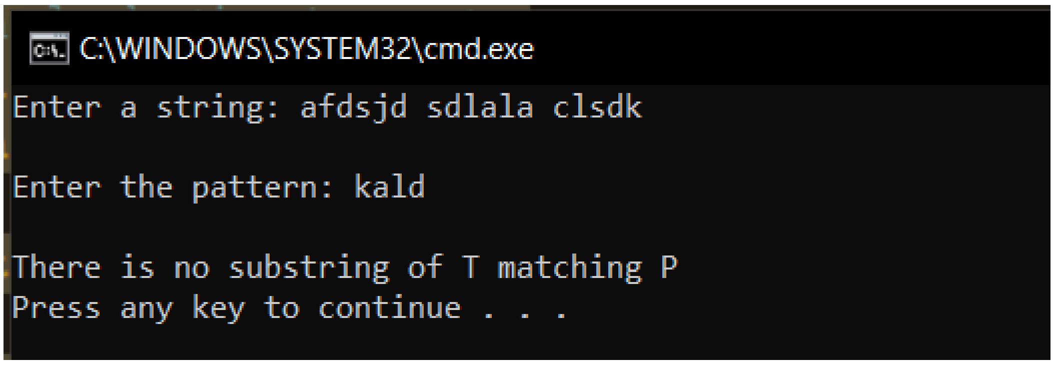
```
Enter a string: ababcabcabababd

Enter the pattern: ababd

The substring starts from index 10
Press any key to continue . . .
```



C:\WINDOWS\SYSTEM32\cmd.exe

```
Enter a string: w pi hee fyd

Enter the pattern: hee

The substring starts from index 5
Press any key to continue . . .
```

Sherwyn Fernandes
SE COMP-B
191105063



```
Enter a string: afdsjd sdlala clsdk

Enter the pattern: kald

There is no substring of T matching P
Press any key to continue . . .
```

Conclusion:

- Time complexity of the algorithm is of the order of O(m+n), where m is length of the pattern and n is the length of the text string. This run time is highly optimal for the worst-case scenario