Sherwyn Fernandes
SE COMP-B
191105063

Experiment No: 14

AIM: Implementation of Brute force and BM 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

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

That is

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

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.

Brute Force Pattern Matching:

In this algorithm, we simply test all the placements of P, relative to T.

It consists of two nested loops, with the outer loop indexing through all possible starting indices of the pattern in the text, and the inner loop indexing through each character of the pattern, comparing it to its potentially corresponding character, in the text. Thus, the correctness of the brute-force pattern matching algorithm follows immediately.

The running time of brute-force pattern matching in the worst case is not good, however, because, for each candidate index in T, we can perform up to m character comparisons to discover that P does not match T at the current index.

The outer for-loop is executed at most n – m + 1 times,

Sherwyn Fernandes
SE COMP-B
191105063

and the inner loop is executed at most m times. Thus, the running time of the bruteforce method is O((n -m + 1)m), which is O(nm). Thus, in the worst case, when n and m are roughly equal, this algorithm has a quadratic running time.

Brute Force ALGORITHM:

Algorithm BruteForce_PatternMatching(T,P)
    for i=0 to n - m {for each candidate index in T} **do**
        j=0
        while (j<m and T[i+j] = P[j]) **do**
        j=j+1

        if j=m **then**
        return i

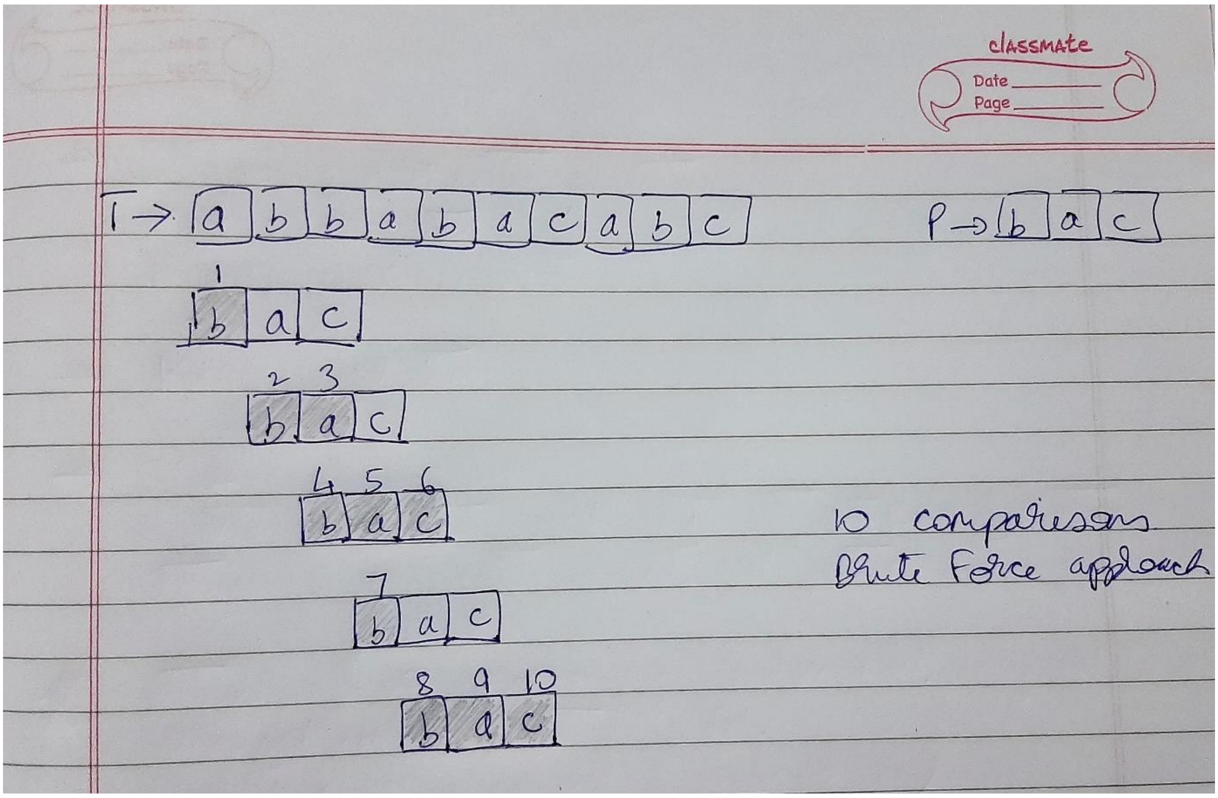
return "There is no substring of T matching P."

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



PROGRAM IMPLEMENTATION:

```
#include<iostream>

using namespace std;

int bruteforce(char *t, char *p)

{

	int j,n=strlen(t),m=strlen(p);


	for(int i=0;i<n-m;i++)

		{

			j=0;

			while(j<m && t[i+j] == p[j])
```

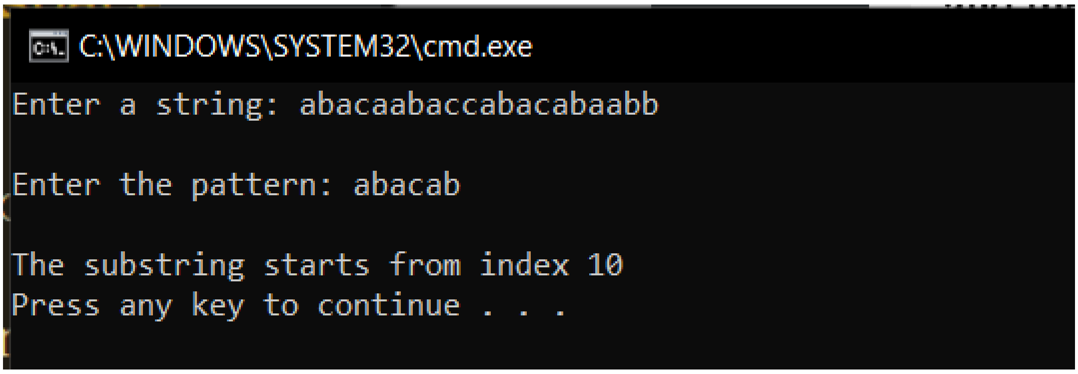
Sherwyn Fernandes
SE COMP-B
191105063

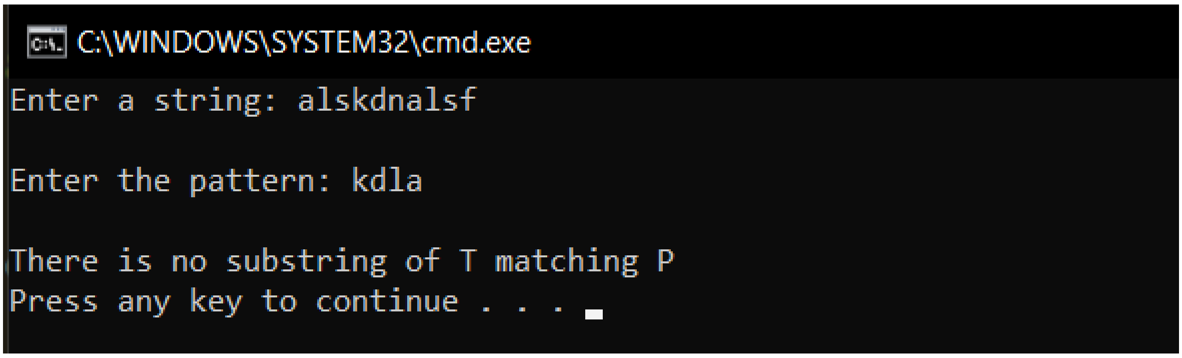
```
                    j+=1;


                if(j==m)

                    return i;


            }


            return -1;


}


int main()

{

        char t[20],p[20];

        int i;

        cout<<"Enter a string: ";

        cin>>t;

        cout<<"\nEnter the pattern: ";

        cin>>p;


        i = bruteforce(t,p);

        if(i!=-1)

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


        else
```

Sherwyn Fernandes
SE COMP-B
191105063

```
        cout<<"\nThere is no substring of T matching P\n";
    return 0;
}
```

OUTPUTS:





Boyer Moore Pattern Matching:

Boyer Moore Pattern Matching algorithm uses heuristics to simplify the approach to solve the problem dfsd

Looking-Glass Heuristic: *When testing a possible placement of P against T, begin the comparisons from the end of P and move backward to the front of P*

Character-Jump Heuristic: *During the testing of a possible placement of P against T, a mismatch of text character T [i] = c with the corresponding pattern character P[j] is handled as follows. If c is not contained anywhere in P, then shift P*

Sherwyn Fernandes
SE COMP-B
191105063

completely past T[i] (for it cannot match any character in P). Otherwise shift P until an occurrence of character c in P gets aligned with T [i]

Defining Character-jump heuristic:

To implement this heuristic, we define a function Iast(c) that takes a character c from the text and specifies how far we may shift the pattern P if a character equal to c is found in the text that does not match the pattern. In particular, we define Iast(c) as follows:

- If c is in P, last(c) is the index of the last (right-most) occurrence of c in P Otherwise, we conventionally define Iast(c) = 1.

The correctness of the BM pattern matching algorithm follows from the fact that each time the method makes a shift, it is guaranteed not to "skip" over any possible matches. For last(c) is the location of the last occurrence of c in P.

The worst-case running time of the BM algorithm is O(nm + ∑). Namely computation of the last function takes time O(m + ∑) and the actual search for the pattern takes O(nm) time in the worst case, the same as the brute-force algorithm. However, the comparisons are drastically reduced.

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

Boyer Moore ALGORITHM:

Algorithm BM Match(T,P):

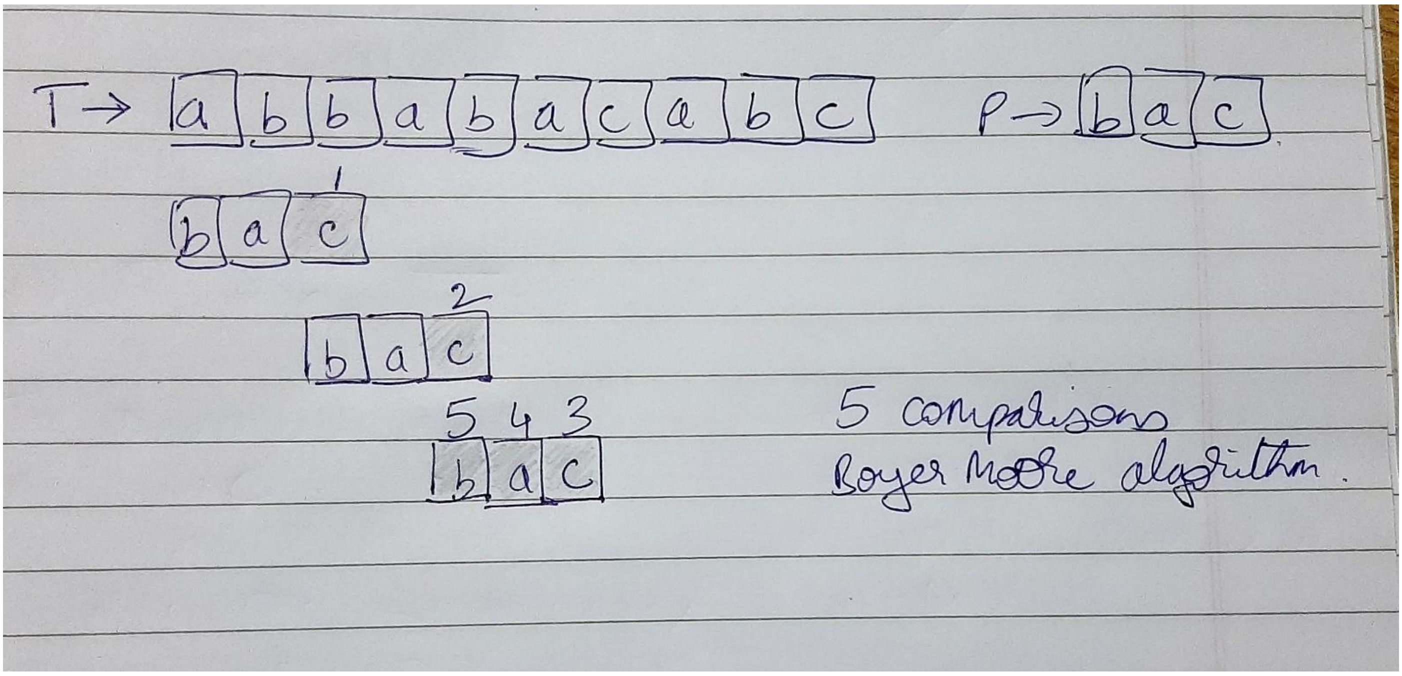
{

Sherwyn Fernandes
SE COMP-B
191105063

```
    i = m-1
    j = m-1
    repeat
    if P[j] = T[i] then
        if j=0 then
            return i      {a match!}
        else
            i=i-1
            j=j-1
    else
        i = i + m – min(j, 1+last(T[i])
        j = m-1

    until i>n-1
return "There is no substring in T matching P"
}
```

Sherwyn Fernandes
SE COMP-B
191105063

Problem Tracing



PROGRAM IMPLEMENTATION:

```
#include<iostream>

#include<map>

using namespace std;


typedef map<char,int> letterMap;        //last occurrence map

char p[20];


letterMap lastOcc;


int last(char c)

{
```

Sherwyn Fernandes
SE COMP-B
191105063

```
        auto iterator = lastOcc.find(c);


        if(iterator != lastOcc.end())

                return iterator->second;


        return -1;

}




int BM(char *t)

{


        int m=strlen(p),n=strlen(t);


        int i=m-1,j=m-1;


        for(int k=m-1;k>=0;k--)

                lastOcc[p[k]] = -1;




        for(int k=m-1;k>=0;k--)

                if(lastOcc[p[k]] == -1)

                        lastOcc[p[k]] = k;


        do
```

Sherwyn Fernandes
SE COMP-B
191105063

```
    {
        if(p[j] == t[i])
            {
                if(j==0)
                    return i;

                else
                {
                    j-=1;
                    i-=1;
                }
            }

        else
        {
            i = i + m - min(j, 1+last(t[i]));
            j=m-1;
        }

    }while(i<n);

    return -1;
}

int main()
```

Sherwyn Fernandes
SE COMP-B
191105063

```
{
	char t[20];
	int i;
	cout<<"Enter a string: ";
	cin.getline(t,20);
	cout<<"\nEnter the pattern: ";
	cin.getline(p,20);

	i = BM(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;
}
```

OUTPUT:

```
C:\WINDOWS\SYSTEM32\cmd.exe
Enter a string: lskdfnsod

Enter the pattern: dfns

The substring starts from index 3
Press any key to continue . . .
```

Sherwyn Fernandes
SE COMP-B
191105063

Conclusion:

- Time complexity of both the algorithms is of the order of O(m*n), where m is length of the pattern and n is the length of the text string.
- However, the number of comparisons needed by BM algorithm is much lesser than that needed by the brute force approach