**Practical-8**

**Aim: String Matching Algorithm**

**Software Requirement:** GNU Compiler Collection (GCC), MS Word.

**Hardware Requirement:** Desktop Computer.

**[8.1]**

**Aim:** Suppose you are given a source string S[0 ..n − 1] of length n, consisting of symbols a and b. Suppose that you are given a pattern string P[0 ..m − 1] of length m < n, consisting of symbols a, b, and \*, representing a pattern to be found in string S. The symbol \* is a “wild card” symbol, which matches a single symbol, either a or b. The other symbols must match exactly. The problem is to output a sorted list M of valid “match positions”, which are positions j in S such that pattern P matches the substring S [j..j + |P|− 1]. For example, if S = ababbab and P = ab\*, then the output M should be [0, 2].

Implement a straightforward, naive algorithm to solve the problem.

**Theory:**

Slide the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches. The naïve approach tests all the possible placement of Pattern P [1.......m] relative to text T [1......n]. We try shift s = 0, 1.......n-m, successively and for each shift s. Compare T [s+1.......s+m] to P [1......m].

The naïve algorithm finds all valid shifts using a loop that checks the condition P [1.......m] = T [s+1.......s+m] for each of the n - m +1 possible value of s.

**Algorithm:**

naiveAlgorithm(mainString, pattern)

M = [ ]

for s = 0 to n − m

valid = TRUE

for j = 0 to m − 1

if P[j] = ∗ and P[j] = S[s + j]

valid = FALSE

if valid

M.APPEND(s); return M

**Code:**

#include <iostream>

#include <vector>

using namespace std;

void naiveAlgorthm(vector<char> text, vector<char> pattern)

{

   int i, j;

   bool valid;

   int textSize = text.size();

   int pattSize = pattern.size();

   vector<int> M;

   for (i = 0; i < (textSize - pattSize); i++)

   {

      valid = true;

      for (j = 0; j < (pattSize - 1); j++)

      {

         if (pattern[j] != '\*' && pattern[j] != text[i + j])

            valid = false;

      }

      if (valid)

      {

         M.push\_back(i);

      }

   }

   cout << endl << "--------------------------------" << endl;

   for (i = 0; i < M.size(); i++)

   {

      cout << "Pattern found at index: " << M[i] << endl;

   }

}

int main()

{

   int sizeText, sizePattern;

   char temp, temp1;

   vector<char> text;

   vector<char> pattern;

   cout << "Enter Size of Text: ";

   cin >> sizeText;

   cout << "\nEnter Text: ";

   for (int i = 0; i < sizeText; i++)

   {

      cin >> temp;

      text.push\_back(temp);

   }

   cout << "\nEnter Size of Pattern: ";

   cin >> sizePattern;

   cout << "\nEnter Pattern: ";

   for (int i = 0; i < sizePattern; i++)

   {

      cin >> temp1;

      pattern.push\_back(temp1);

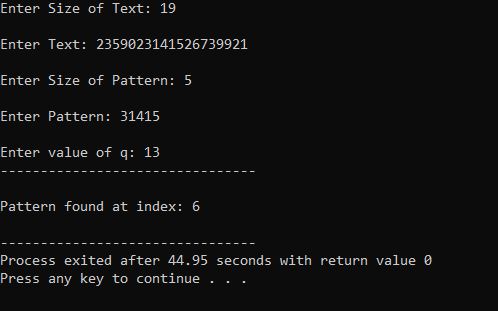
   }

   naiveAlgorthm(text, pattern);

   return 0;

}

**Output:**

****

**Conclusion:**

In this practical we performed Naïve String-Matching Algorithm and found out that the time complexity of this particular algorithm is O(m\*n) which is the worst-case complexity of String-Matching Algorithm.

**[8.2]**

# **Aim:** Implement Rabin karp algorithm and test it on the following test cases:

|  |  |
| --- | --- |
| **String** | **Pattern** |
|  |  |
| 2359023141526739921 | 31415 |
|  | q=13 |
|  | and |
|  | q=11 |
| ABAAABCDBBABCDDEBCABC | ABC |
|  | q=101 |

**Theory:**

Rabin-Karp is another pattern searching algorithm to find the pattern in a more efficient way. It also checks the pattern by moving window one by one, but without checking all characters for all cases, it finds the hash value. When the hash value is matched, then only it tries to check each character. This procedure makes the algorithm more efficient.

The time complexity is O(m+n), but for the worst case, it is O(mn).

Input:

Main String: “ABAAABCDBBABCDDEBCABC”, Pattern “ABC”

Output:

Pattern found at position: 4

Pattern found at position: 10

Pattern found at position: 18

**Algorithm:**

**Input −** The main text and the pattern. Another prime number of find hash location

**Output −** location where patterns are found

Begin

   patLen := pattern Length

   strLen := string Length

   patHash := 0 and strHash := 0, h := 1

   maxChar := total number of characters in character set

   for index i of all character in pattern, do

      h := (h\*maxChar) mod prime

   done

   for all character index i of pattern, do

      patHash := (maxChar\*patHash + pattern[i]) mod prime

      strHash := (maxChar\*strHash + text[i]) mod prime

   done

   for i := 0 to (strLen - patLen), do

      if patHash = strHash, then

         for charIndex := 0 to patLen -1, do

            if text[i+charIndex] ≠ pattern[charIndex], then

               break the loop

         done

         if charIndex = patLen, then

            print the location i as pattern found at i position.

      if i < (strLen - patLen), then

         strHash := (maxChar\*(strHash – text[i]\*h)+text[i+patLen]) mod prime, then

      if strHash < 0, then

         strHash := strHash + prime

   done

End

**Code:**

#include <iostream>

#include <vector>

#define MAXCHAR 256

using namespace std;

void rabinKarp(vector<char> text, vector<char> pattern, int q)

{

    int j;

    int textSize = text.size();

    int pattSize = pattern.size();

    int patternHash = 0, textHash = 0, h = 1;

    //calculating h = {d^(M-1)} mod prime

    for (int i = 0; i < pattSize - 1; i++)

    {

        h = (MAXCHAR \* h) % q;

    }

    for (int i = 0; i < pattSize; i++)

    {

        //pattern hash value

        patternHash = (MAXCHAR \* patternHash + pattern[i]) % q;

        //hash for first window

        textHash = (MAXCHAR \* textHash + text[i]) % q;

    }

    cout << "--------------------------------" << endl;

    for (int i = 0; i <= (textSize - pattSize); i++)

    {

        //when hash values are same check for matching

        if (patternHash == textHash)

        {

            int flag = 1;

            for (j = 0; j < pattSize; j++)

            {

                if (text[i + j] != pattern[j])

                {

                    flag = 0;

                    break;

                }

            }

            //the pattern is found

            if (j == pattSize)

                cout << endl << "Pattern found at index: " << i << endl;

        }

        // find hash value for next window

        if (i < textSize - pattSize)

        {

            textHash = (MAXCHAR \* (textHash - text[i] \* h) + text[i + pattSize]) % q;

            //when hash value is negative, make it positive

            if (textHash < 0)

                textHash += q;

        }

    }

}

int main()

{

    int sizeText, sizePattern;

    char temp, temp1;

    vector<char> text;

    vector<char> pattern;

    int q;

    cout << "Enter Size of Text: ";

    cin >> sizeText;

    cout << "\nEnter Text: ";

    for (int i = 0; i < sizeText; i++)

    {

        cin >> temp;

        text.push\_back(temp);

    }

    cout << "\nEnter Size of Pattern: ";

    cin >> sizePattern;

    cout << "\nEnter Pattern: ";

    for (int i = 0; i < sizePattern; i++)

    {

        cin >> temp1;

        pattern.push\_back(temp1);

    }

    cout << "\nEnter value of q: ";

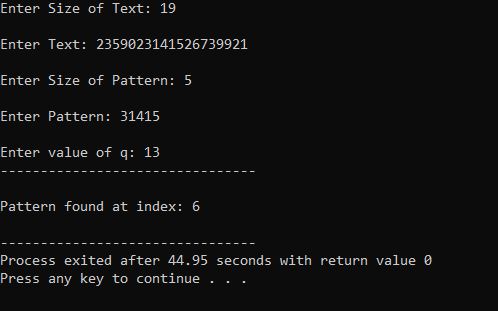
    cin >> q;

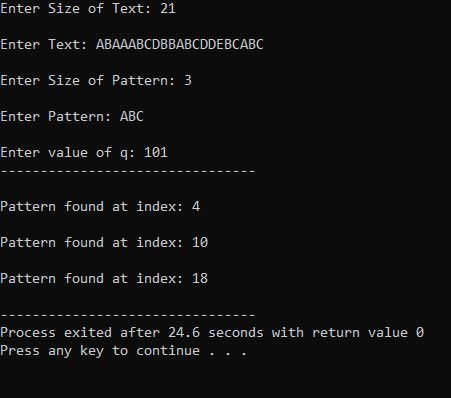
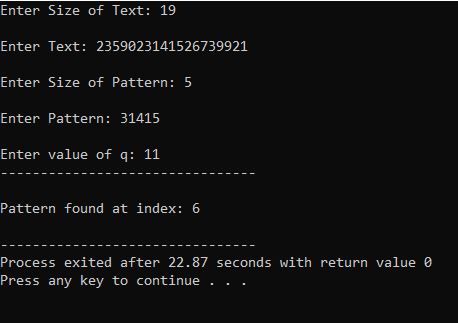
    rabinKarp(text, pattern, q);

    return 0;

}

**Output:**

****

****

**Conclusion:**

In this Practical we performed String Matching algorithm known as Rabin Karp Algorithm. And found out that the time complexity of this particular algorithm is O(m+n) and in worst case it is O(m\*n)

**[8.3]**

**Aim:** Boyers Moore Algorithm

**Theory:**

The Boyer Moore algorithm is a searching algorithm in which a string of length **n**and a pattern of length **m** is searched. It prints all the occurrences of the pattern in the Text.

Like the other string matching algorithms, this algorithm also preprocesses the pattern.

Boyer Moore uses a combination of two approaches – Bad character and good character heuristic. Each of them is used independently to search the pattern.

In this algorithm, different arrays are formed for both heuristics by pattern processing, and the best heuristic is used at each step. Boyer Moore starts to match the pattern from the last, which is a different approach from KMP and Naive.

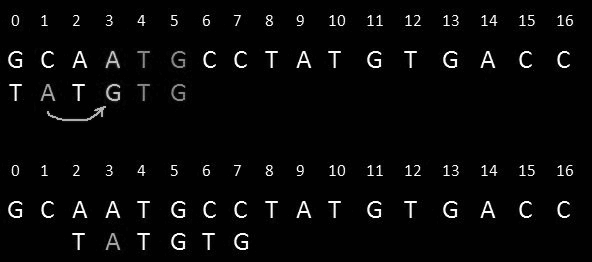
Let us now understand the bad character heuristic :

**Bad character:**A character in the Text that has no match with the pattern is called a bad character.

Now, when there is a case of mismatch, we shift the pattern until it becomes a match, and the pattern moves past the mismatched character.

**MISMATCH BECOMES A MATCH**

Find the position of the last character that mismatched in the pattern and, then shift the pattern until it gets aligned to the mismatched character in the Text.

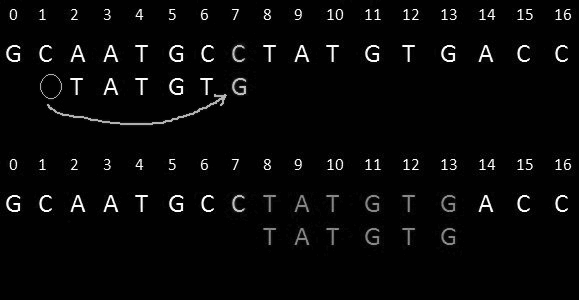


**Example:** In the above Text, there is a mismatch at position 3( A does not match with G). Now we find the last occurrence of A in our Text. At position 1, We get the A now we shift the

pattern to position 1.

**PATTERN MOVE PAST THE MISMATCH CHARACTER**

We look up the last occurrence of the character that does not match the pattern; if it does not exist in the pattern, we move the pattern past that last mismatched character.



**Algorithm:**

naiveAlgorithm(mainString, pattern)

M = [ ]

for s = 0 to n − m

valid = TRUE

for j = 0 to m − 1

if P[j] = ∗ and P[j] = S[s + j]

valid = FALSE

if valid

M.APPEND(s); return M

**Code:**

#include<iostream>

using namespace std;

void fullSuffixMatch(int shiftArr[], int borderArr[], string pattern) {

   int n = pattern.size();       //find length of pattern

   int i = n;

   int j = n+1;

   borderArr[i] = j;

   while(i > 0) {

      //search right when (i-1)th and (j-1)th item are not same

      while(j <= n && pattern[i-1] != pattern[j-1] ) {

         if(shiftArr[j] == 0)

            shiftArr[j] = j-i;     //shift pattern from i to j

         j = borderArr[j];       //update border

      }

      i--;

      j--;

      borderArr[i] = j;

   }

}

void partialSuffixMatch(int shiftArr[], int borderArr[], string pattern) {

   int n = pattern.size();    //find length of pattern

   int j;

   j = borderArr[0];

   for(int i = 0; i<n; i++) {

      if(shiftArr[i] == 0)

         shiftArr[i] = j;        //when shift is 0, set shift to border value

         if(i == j)

            j = borderArr[j];    //update border value

   }

}

void searchPattern(string mainString, string pattern, int array[], int \*index) {

   int patLen = pattern.size();

   int strLen = mainString.size();

   int borderArray[patLen+1];

   int shiftArray[patLen + 1];

   for(int i = 0; i<=patLen; i++) {

      shiftArray[i] = 0;     //set all shift array to 0

   }

   fullSuffixMatch(shiftArray, borderArray, pattern);

   partialSuffixMatch(shiftArray, borderArray, pattern);

   int shift = 0;

   while(shift <= (strLen - patLen)) {

      int j = patLen - 1;

      while(j >= 0 && pattern[j] == mainString[shift+j]) {

         j--;         //reduce j when pattern and main string character is matching

      }

      if(j < 0) {

         (\*index)++;

         array[(\*index)] = shift;

         shift += shiftArray[0];

      }else {

          shift += shiftArray[j+1];

      }

   }

}

int main() {

   cout<<"Boyer's Moore Algorithm\n";

   string mainString;

   cout<<"Enter Main String : ";

   cin>>mainString;

   string pattern;

   cout<<"Enter Pattern String : ";

   cin>>pattern;

   int locArray[mainString.size()];

   int index = -1;

   searchPattern(mainString, pattern, locArray, &index);

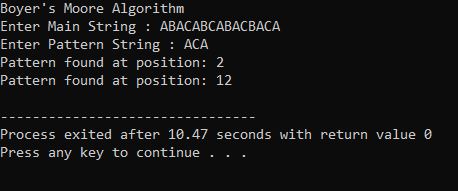
   for(int i = 0; i <= index; i++) {

      cout << "Pattern found at position: " << locArray[i]<<endl;

    }

}

**Output:**

****

**Conclusion:**

In this practical we performed Boyers Moore Algorithm and found out that the time complexity of this particular algorithm is O(m\*n) which is the worst-case complexity of String-Matching Algorithm.

**[8.4]**

**Aim:** KMP Algorithm.

**Theory:**

The KMP algorithm is a solution to the string search problem wherein we are required to find if a given pattern string occurs in another main string. It is one of the advanced string matching algorithm that was conceived by Donald Knuth, James H. Morris and Vaughan Pratt, hence the name **"KMP algorithm"**.

The algorithm **keeps a track of the comparison of characters between main text and pattern**, thereby ensuring that comparisons that have already been done are not repeated, i.e. backtracking of the main string never occurs. All this results in a linear matching time and a more optimized solution.

**Algorithm:**

**COMPUTE- PREFIX- FUNCTION (P)**

1. m ←length [P] //'p' pattern to be matched

2. Π [1] ← 0

3. k ← 0

4. for q ← 2 to m

5. do while k > 0 and P [k + 1] ≠ P [q]

6. do k ← Π [k]

7. If P [k + 1] = P [q]

8. then k← k + 1

9. Π [q] ← k

10. Return Π

**KMP-MATCHER (T, P)**

1. n ← length [T]

2. m ← length [P]

3. Π← COMPUTE-PREFIX-FUNCTION (P)

4. q ← 0 // numbers of characters matched

5. for i ← 1 to n // scan S from left to right

6. do while q > 0 and P [q + 1] ≠ T [i]

7. do q ← Π [q] // next character does not match

8. If P [q + 1] = T [i]

9. then q ← q + 1 // next character matches

10. If q = m // is all of p matched?

11. then print "Pattern occurs with shift" i - m

12. q ← Π [q] // look for the next match

**Code:**

#include <iostream>

using namespace std;

void computePrefix(string p, int m, int lps[])

{

    // length of the previous longest prefix suffix

    int l = 0;

    lps[0] = 0; // lps[0] is always 0

    //calculating lps[i] for i = 1 to M-1

    int i = 1;

    while (i < m) {

        if (p[i] == p[l]) {

            l++;

            lps[i] = l;

            i++;

        }

        else

        {

                if (l != 0) {

                l = lps[l - 1];

            }

            else

            {

                lps[i] = 0;

                i++;

            }

        }

    }

}

void kmpPatternSearch(string p, string S)

{

    int m = p.length();

    int n = S.length();

    // creating lps array

    int lps[m];

    // finding prefix table

    computePrefix(p, m, lps);

    int i = 0;

    int j = 0;

    while (i < n) {

        if (p[j] == S[i]) {

            j++;

            i++;

        }

        if (j == m) {

            cout<<"Pattern found at location: "<<i - j<<"\n";

            j = lps[j - 1];

        }

        // mismatch case

        else if (i < n && p[j] != S[i]) {

            // Skip matching lps[0..lps[j-1]] characters

            if (j != 0)

                j = lps[j - 1];

            else

                i = i + 1;

        }

    }

}

int main() {

    cout<<"KMP Algorithm\n";

    string text;

    cout<<"Enter Main String : ";

    cin>>text;

    string pat;

    cout<<"Enter Pattern String : ";

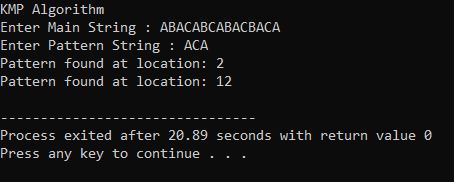
    cin>>pat;

    kmpPatternSearch(pat, text);

    return 0;

}

**Output:**

****

**Conclusion:**

In this practical we performed KMP Algorithm and found out that the worst case time complexity of this particular algorithm is O(n).