**BRUTE FORCE TECHNIQUE – NAIVE APPROACH**

i) It is the simplest method which uses brute force approach.

ii) It is a straight forward approach of solving the problem.

iii) It compares first character of pattern with searchable text. If match is found, pointersinboth strings are advanced. If match not found, pointer of text is incremented andpointerofpattern is reset. This process is repeated until the end of the text.

iv) It does not require any pre-processing. It directly starts comparing both strings character bycharacter.

v) Time Complexity = **O(m\* (n-m))**

**Algorithm for Naive string matching :**

Algorithm-NAVE\_STRING\_MATCHING (T, P)

for i←0 to n-m do

if P[1......m] == T[i+1.....i+m] then

print "Match Found"

end

end

Given a text of length **N** txt[0..N-1] and a pattern of length M pat[0..M-1], write a functionsearch(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assumethat N > M.

**Examples:**

***Input:*** *txt[] = “THIS IS A TEST TEXT”, pat[] = “TEST”*

***Output:*** *Pattern found at index 10*

***Input:*** *txt[] = “AABAACAADAABAABA”,*

*pat[] = “AABA”*

***Output:*** *Pattern found at index 0, Pattern found at index 9, Pattern found at index 12*

**

#include <iostream>

using namespace std;

#include<string.h>

int main()

{

 char mstr[100],pstr[100];

 int i, j, mslen, pslen, count=0;

 cout << "\nEnter the main String : ";

 gets(mstr);

 cout << "\nEnter the Pattern/Search String : ";

 gets(pstr);

 mslen = strlen(mstr);

 pslen = strlen(pstr);

 for(i=0; i<=mslen-pslen; i++)

 {

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

 if(mstr[i+j] != pstr[j])

 break;

 if(j==pslen)

 {

 cout << "\nPattern found at index " << i << " in the main string\n\n"; count++;

 }

 }

 if(count==0)

 cout << "\nSorry, Pattern not found in the main string\n\n"; return 0;

}

**RABIN-KARP PATTERN MATCHING ALGORITHM**

The Rabin-Karp-Algorithm

The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character subsequences of text to be compared. If the hash values are unequal, the algorithm will determine the hash value for next M-character sequence. If the hash values are equal, the algorithm will analyze the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and character matching is only required when the hash values match.

**RABIN-KARP-MATCHER (T, P, d, q)**

1. n ← length [T]

 2. m ← length [P]

 3. h ← dm-1 mod q

 4. p ← 0

 5. t0 ← 0

 6. for i ← 1 to m

 7. do p ← (dp + P[i]) mod q

 8. t0 ← (dt0+T [i]) mod q

 9. for s ← 0 to n-m

 10. do if p = ts

 11. then if P [1.....m] = T [s+1.....s + m]

 12. then "Pattern occurs with shift" s

 13. If s < n-m

 14. then ts+1 ← (d (ts-T [s+1]h)+T [s+m+1])mod q

**Example:** For string matching, working module q = 11, how many spurious hits does the Rabin-Karp matcher encounters in Text T = 31415926535.......

1. T = 31415926535.......

2. P = 26

3. Here T.Length =11 so Q = 11

4. And P mod Q = 26 mod 11 = 4

5. Now find the exact match of P mod Q...

**Solution:**

****

****

****Complexity:

The running time of **RABIN-KARP-MATCHER** in the worst case scenario **O ((n-m+1) m** but it has a good average case running time. If the expected number of strong shifts is small **O (1)** and prime q is chosen to be quite large, then the Rabin-Karp algorithm can be expected to run in time **O (n+m)** plus the time to require to process spurious hits

// Rabin-Karp algorithm implementation

#include <string.h>

#include <iostream>

using namespace std;

#define d 10

int rabinKarp(char pattern[100], char text[100], int q)

{

 int m = strlen(pattern);

 int n = strlen(text);

 int i, j;

 int p = 0;

 int t = 0;

 int h = 1;

 int count = -1;

 for (i = 0; i < m - 1; i++)

 h = (h \* d) % q;

 //cout << "\n h value is " << h << endl; // Calculate hash value for pattern and text for (i = 0; i < m; i++)

 {

 p = (d \* p + pattern[i]) % q; t = (d \* t + text[i]) % q;

 }

// Find the match

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

 if (p == t)

 {

 for (j = 0; j < m; j++) {

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

 }

 if (j == m)

 {

 cout << "Pattern is found at position: " << i + 1 << endl; count++;

 }

 }

 if (i < n - m)

 {

 t = (d \* (t - text[i] \* h) + text[i + m]) % q; if (t < 0)

 t = (t + q);

 }

 }

 return count;

}

int main()

{

 char text[100], pattern[100];

 cout << "\nEnter the main String : ";

 gets(text);

 cout << "\nEnter the Pattern/Search String : ";

 gets(pattern);

 int q = 13;

 int res = rabinKarp(pattern, text, q);

 if(res < 0)

 cout << "Pattern not found in the text" << endl; }

**KNUTH MORRIS PATTERN MATCHING ALGORITHM**

#include<iostream>

#include<string.h>

using namespace std;

void findPrefix(string pattern, int m, int prefArray[ ])

{

 int i = 0;

 prefArray[0] = 0; //first place is always 0 as no prefix for(int j = 1; j<m; j++)

 {

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

 {

 i++;

 prefArray[j] = i;

 }

 else

 {

 if(i != 0)

 {

 i = prefArray[i - 1];

 j--; //decrease i to avoid effect of increasing after iteration }

 else

 prefArray[j] = 0;

 }

 }

 printf("\n");

 for(i=0;i<m;i++)

 printf(" %d ", prefArray[i]);

}

void kmpPattSearch(string mainString, string pattern, int \*locArray, int &loc) {

 int n, m, i = 0, j = 0;

 n = mainString.size();

 m = pattern.size();

 int prefixArray[m]; //prefix array as same size of pattern findPrefix(pattern, m, prefixArray);

 loc = 0;

 while(i < n)

 {

 if(mainString[i] == pattern[j])

 {

 i++; j++;

 }

 if(j == m)

 {

 locArray[loc] = i-j; //item found at i-j position. loc++;

 j = prefixArray[j-1]; //get the prefix LENGTH from array }

 else if(i < n && pattern[j] != mainString[i])

 {

 if(j != 0)

 j = prefixArray[j-1];

 else

 i++;

 }

 }

}

int main()

{

 char str[100], patt[100];

 int locationArray[100], index;

 cout << "\nEnter the string: ";

 gets(str);

 cout << "\nEnter the patttern: ";

 gets(patt);

 kmpPattSearch(str, patt, locationArray, index); for(int i = 0; i<index; i++)

 {

 cout << "Pattern found at location: " <<locationArray[i] << endl; }

}