**Experiment No.6: Internet Algorithms**

**Date:**

**Aim:-** Write C program to implement the following using Internet Algorithm.

1. KMP Pattern Matching
2. BM Pattern Matching
3. Huffman Coding (Text Compression)
4. Longest common Subsequence. (Text Similarities)

**Theory :**

Internet algorithms are specifically designed algorithms used in various aspects of the internet to solve specific problems efficiently.These algorithms address challenges related to data transmission, routing, network optimization, information retrieval, and security, among others. Internet algorithms often consider factors such as latency, bandwidth, scalability, fault tolerance, and resource constraints to ensure efficient and reliable operations.

Knuth-Morris-Pratt (KMP) Algorithm:

The KMP algorithm is used for pattern matching in strings efficiently.

It avoids unnecessary comparisons by utilizing information from previous matches.

It pre-processes the pattern to construct a lookup table (also called the "failure function") that helps skip unnecessary comparisons during matching.The algorithm compares the pattern with the input text character by character, utilizing the lookup table to determine the next position for comparison in case of a mismatch.

Boyer-Moore (BM) Algorithm:

The Boyer-Moore algorithm is another efficient pattern matching algorithm, particularly suitable for searching in large texts.It pre-processes the pattern and utilizes two heuristics: the "bad character rule" and the "good suffix rule."The bad character rule skips comparisons by shifting the pattern to align the rightmost occurrence of a mismatched character in the text with the corresponding position in the pattern.The good suffix rule utilizes information about matching suffixes in the pattern to skip unnecessary comparisons.

Huffman Coding:

Huffman coding is a compression algorithm used to encode data efficiently by assigning shorter codes to frequently occurring characters and longer codes to less frequent characters.The algorithm builds a binary tree (Huffman tree) based on the frequency of characters in the input data.The characters with higher frequency are assigned shorter codes, and those with lower frequency are assigned longer codes, ensuring prefix-free codes.Huffman coding achieves compression by representing the input data using the generated Huffman codes, reducing the overall number of bits required for storage or transmission.

Longest Common Subsequence (LCS) Algorithm:

The LCS algorithm is used to find the longest common subsequence between two sequences, typically strings.It determines the longest subsequence that is present in both sequences but does not necessarily have to be contiguous.The algorithm utilizes dynamic programming to build a table that stores the lengths of the longest common subsequences for various subproblems.Starting from the end of the sequences, the algorithm fills the table by considering two cases: matching characters or non-matching characters.Finally, the algorithm traces back the table to reconstruct the LCS.

**A)KMP Pattern Matching**

**DATE :-**

**Problem Statement**

Write a C program to implement KMP pattern matching for

T = aabbbaababbbabab and P = bbaba

**Algorithm**

**Algorithm KMPMatch(T,P)**:

Input: Strings T (text) with n characters and P (pattern) with m characters

Output: Starting index of the first substring of T matching P, or an indication

that P is not a substring of T

f ← KMPFailureFunction(P) // construct the failure function f for P

i ← 0

j ← 0

while i<n do

if P[j] = T[i] then

if j = m − 1 then

return i − m + 1 // a match!

i ← i + 1

j ← j + 1

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):**

Input: String P (pattern) with m characters

Output: The failure function f for P, which maps j to the length of the longest

prefix of P that is a suffix of P[1..j]

i ← 1

j ← 0

f(0) ← 0

while i<m do

if P[j] = P[i] then

// we have matched j + 1 characters

f(i) ← j + 1

i ← i + 1

j ← j + 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 and Space Complexity**

Time Complexity = **O(m+n)**

Space complexity = **O(m)**

Where , n is the length of text and m is the length of pattern .

**Code:**

#include <stdio.h>

#include <string.h>

#define MAX 256

int f[MAX];

int Failure(char pattern[], int m)

{

int i = 1, j = 0;

f[0] = 0;

while (i < m)

{

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

{

f[i] = j + 1;

i++, j++;

}

else if (j > 0)

j = f[j - 1];

else

{

f[i] = 0;

i++;

}

}

}

void KMP(char test[], char pattern[], int n, int m)

{

Failure(pattern, m);

int i, j;

i = j = 0;

while (i < n)

{

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

{

printf("%d of test matches with %d of pattern.\n", i, j);

if (j == m - 1)

{

printf("Pattern %s found at index %d in %s.\n", pattern, i - m + 1, test);

return;

}

i++, j++;

}

else

{

printf("%d of test mismatches with %d of pattern.\n", i, j);

if (j > 0)

j = f[j - 1];

else

i++;

}

}

printf("Pattern %s not present in string %s!\n", pattern, test);

return;

}

int main()

{

char test[50], pattern[50];

printf("Enter test string.\n");

scanf("%s", test);

printf("Enter pattern string.\n");

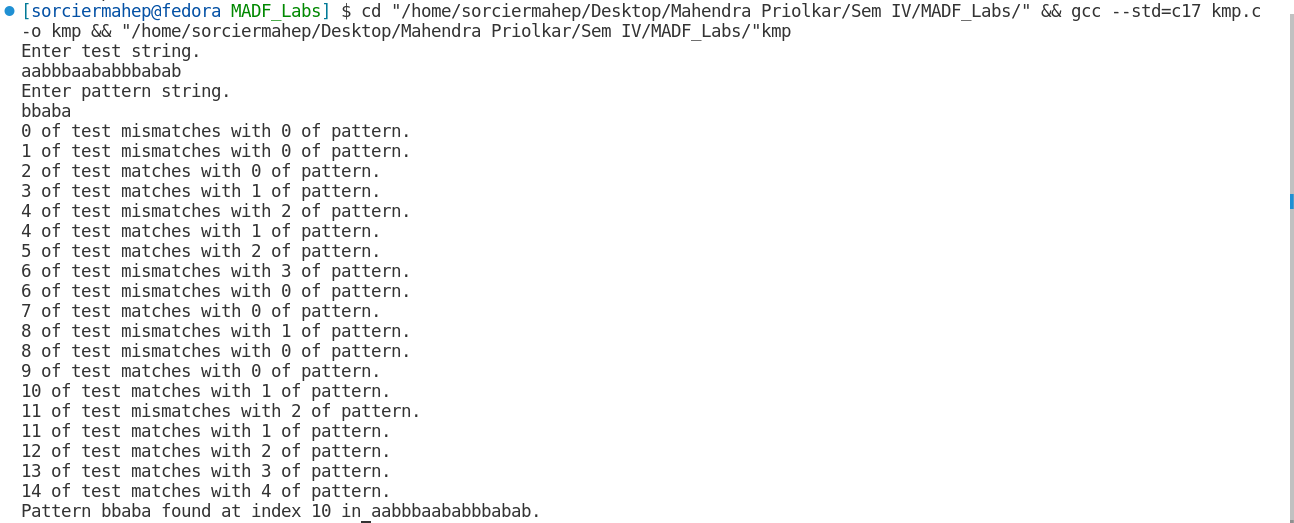
scanf("%s", pattern);

KMP(test, pattern, strlen(test), strlen(pattern));

return 0;

}

**Output:**



**B)** **Boyer Moore pattern Matching**

**DATE :-**

**Problem Statement**

Write a C program to implement Boyer Moore pattern matching for

T = 1123114234112113 and P = 4112113

**Algorithm**

**Algorithm BMMatch(T,P):**

Input: Strings T (text) with n characters and P (pattern) with m characters

Output: Starting index of the first substring of T matching P, or an indication

that P is not a substring of T

compute function last

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])) // jump step

j ← m − 1

until i>n − 1

return “There is no substring of T matching P.”

**Time and Space Complexity**

Time Complexity = **O(m+n)**

Space complexity = **O(m+k)**

Where , n is the length of text and m is the length of pattern .

**Code:**

#include <stdio.h>

#include <string.h>

#define MAX 256

int lastarr[MAX];

int min(int a, int b)

{

return a < b ? a : b;

}

void last(char pattern[MAX], int m)

{

// Index of the last (right most) occurrence of c in pattern.

// Otherwise we define last(c) = -1.

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

lastarr[i] = -1;

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

lastarr[(int)pattern[i]] = i;

}

void BM(char test[], char pattern[], int n, int m)

{

last(pattern, m);

int i, j;

i = m - 1;

j = m - 1;

do

{

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

{

printf("%d of test matches with %d of pattern.\n", i, j);

if (!j)

{

printf("Pattern %s found at index %d in string %s.\n", pattern, i, test);

return;

}

else

{

i--;

j--;

}

}

else

{

printf("%d of test mismatches with %d of pattern.\n", i, j);

i = i + m - min(j, 1 + lastarr[(int)test[i]]);

j = m - 1;

}

} while (i <= n - 1);

printf("Pattern %s not present in string %s!\n", pattern, test);

}

int main()

{

char test[50], pattern[50];

printf("Enter test string.\n");

scanf("%s", test);

printf("Enter pattern string.\n");

scanf("%s", pattern);

BM(test, pattern, strlen(test), strlen(pattern));

return 0; }

**Output:**



**C)Huffman Coding (Text Compression)**

**DATE :-**

**Problem Statement**

Write a C program to implement Huffman Encoding for

“j is the position of the partitioning element”.

**Algorithm**

**Algorithm Huffman(C):**

Input: A set, C, of d characters, each with a given weight, f(c)

Output: A coding tree, T, for C, with minimum total path weight

Initialize a priority queue Q.

for each character c in C do

Create a single-node binary tree T storing c.

Insert T into Q with key f(c).

while Q.size() > 1 do

f1 ← Q.minKey()

T1 ← Q.removeMin()

f2 ← Q.minKey()

T2 ← Q.removeMin()

Create a new binary tree T with left subtree T1 and right subtree T2.

Insert T into Q with key f1 + f2.

return tree Q.removeMin()

**Time and Space Complexity**

Time Complexity = **O(nlogn+m)**

Space complexity = **O(n+mlogn)**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX 200

int greatest = 0;

struct node

{

int freq;

char ch;

struct node \*next, \*left, \*right;

} \*start = NULL;

void add\_at\_end(char c[], int f[], int n)

{

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

{

if (start == NULL)

{

start = (struct node \*)malloc(sizeof(struct node));

start->ch = c[i], start->freq = f[i], start->next = NULL, start->left = NULL, start->right = NULL;

}

else

{

struct node \*ptr = start;

while (ptr->next != NULL)

ptr = ptr->next;

struct node \*temp = (struct node \*)malloc(sizeof(struct node));

temp->ch = c[i], temp->freq = f[i];

temp->next = temp->left = temp->right = NULL, ptr->next = temp;

} } }

void show()

{

struct node \*ptr = start;

while (ptr != NULL)

{

printf("(%c, %d), ", ptr->ch, ptr->freq);

ptr = ptr->next;

}

printf("\n");

}

struct node \*del\_front()

{

struct node \*temp = start;

start = start->next;

return temp;

}

void add\_after(struct node \*temp)

{

if (start == NULL)

start = temp;

else if (start->next == NULL)

{

if (temp->freq < start->freq)

{

temp->next = start;

start = temp;

}

else

{

temp->next = start->next;

start->next = temp;

} }

else

{

int found = 0;

struct node \*posn, \*ptr = start;

while (ptr != NULL)

{

if (ptr->freq == temp->freq)

{

found = 1;

posn = ptr;

break;

}

ptr = ptr->next;

}

ptr = start;

if (found)

{

struct node \*ptr = posn;

while (ptr->next->freq == temp->freq && ptr->next != NULL)

ptr = ptr->next;

temp->next = ptr->next;

ptr->next = temp;

}

else

{

ptr = start;

while (ptr->next->freq <= temp->freq && ptr != NULL)

ptr = ptr->next;

if (ptr != NULL)

{

temp->next = ptr->next;

ptr->next = temp;

}

else

{

ptr = start;

while (ptr->next != NULL)

ptr = ptr->next;

temp->next = ptr->next;

ptr->next = temp;

} } }}

void insert\_end(struct node \*temp)

{

if (start == NULL)

temp->next = NULL, start = temp;

else

{

struct node \*ptr = start;

while (ptr->next != NULL)

ptr = ptr->next;

temp->next = ptr->next;

ptr->next = temp;

}}

void greatest\_freq()

{

struct node \*ptr = start;

while (ptr->next != NULL)

ptr = ptr->next;

greatest = ptr->freq;

}

void Huffman()

{

if (!start)

return;

while (start->next != NULL)

{

struct node \*a = del\_front(), \*b = del\_front(), \*temp;

temp = (struct node \*)malloc(sizeof(struct node));

temp->ch = '$';

temp->freq = a->freq + b->freq;

temp->left = a, temp->right = b, temp->next = NULL;

if (temp->freq >= greatest)

{

greatest = temp->freq;

insert\_end(temp);

}

else

add\_after(temp);

show();

printf("\n");

} }

void printcodes(struct node \*root, int path[], int stringlen)

{

if (root == NULL)

return;

if (root->left == NULL && root->right == NULL)

{

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

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

printf("\t%c\n", root->ch);

}

// left child

path[stringlen] = 0;

printcodes(root->left, path, stringlen + 1);

// right child

path[stringlen] = 1;

printcodes(root->right, path, stringlen + 1);

}

int main()

{

char arr[17] = {'j', 'f', 'a', 'r', 'g', 'l', 'm', 's', 'h', 'p', 'o', 'n', 'e', 'i', 't', ' '};

int freq[17] = {1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 4, 4, 5, 6, 6, 7};

printf("The string is:\n");

printf("j is the position of the partitioning element");

printf("\n\n");

add\_at\_end(arr, freq, strlen(arr));

show();

greatest\_freq();

Huffman();

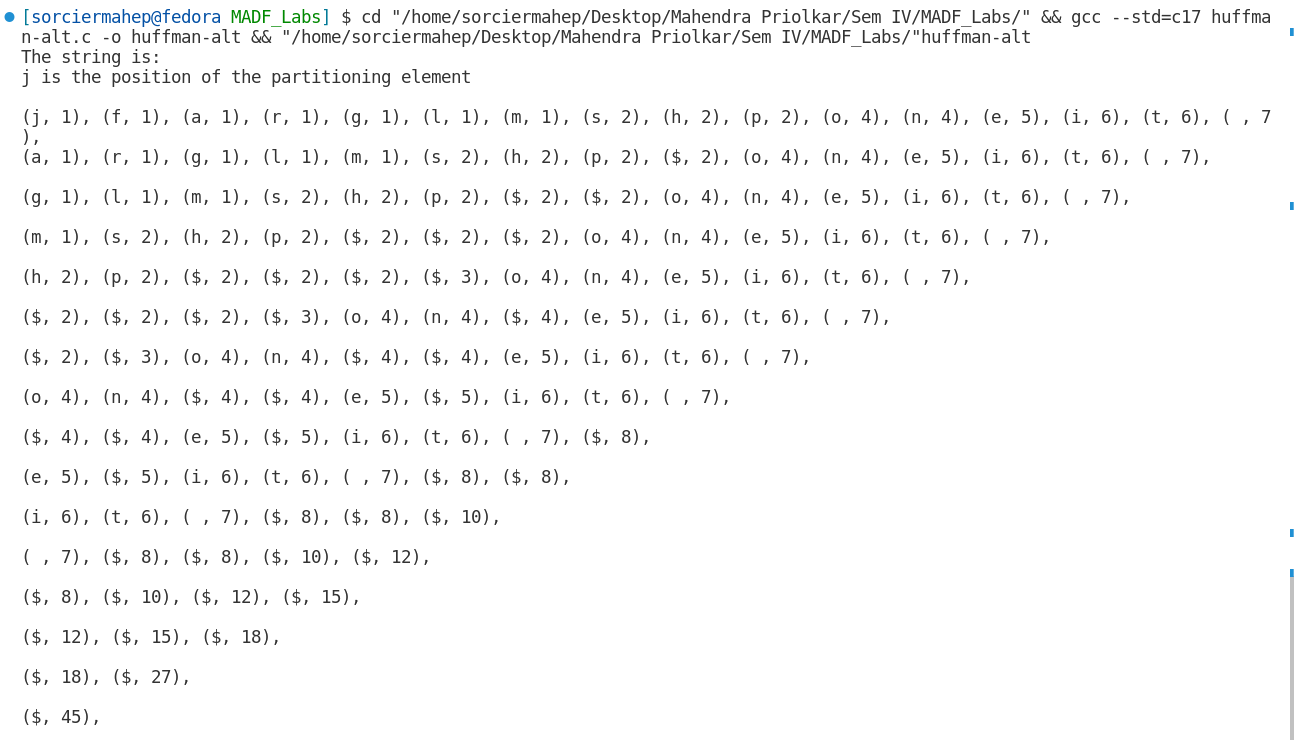
int path[MAX];

printf("Huffman Codes:\n");

printcodes(start, path, 0);

return 0; }

**Output:**





**D)LCS (Text Similarity)**

**DATE :-**

**Problem Statement**

Write a C program to implement LCS for X= KLOKMKNKLOK and Y= KLLKNKLLKNYY

**Algorithm**

**Algorithm LCS(X, Y ):**

Input: Strings X and Y with n and m elements, respectively

Output: For i = 0,...,n − 1, j = 0,...,m − 1, the length L[i, j] of a longest

common subsequence of X[0..i] and Y [0..j]

for i ← −1 to n − 1 do

L[i, −1] ← 0

for j ← 0 to m − 1 do

L[−1, j] ← 0

for i ← 0 to n − 1 do

for j ← 0 to m − 1 do

if X[i] = Y [j] then

L[i, j] ← L[i − 1, j − 1] + 1

else

L[i, j] ← max{L[i − 1, j] , L[i, j − 1]}

return array L

**Time and Space Complexity**

Time Complexity = **O(mn)**

Space complexity = **O(mn)**

Where , n and m are the lengths of the strings.

**Code:**

#include <stdio.h>

#include <string.h>

#define MAX 100

int max(int a, int b)

{

return a > b ? a : b;

}

int LCS(char s1[], char s2[], int m, int n, int matrix[m + 1][n + 1])

{

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

matrix[i][0] = 0;

for (int j = 0; j <= n; j++)

matrix[0][j] = 0;

for (int j = 1; j <= m; j++)

{

for (int i = 1; i <= n; i++)

{

if (s1[i - 1] == s2[j - 1])

matrix[j][i] = matrix[j - 1][i - 1] + 1;

else

matrix[j][i] = max(matrix[j][i - 1], matrix[j - 1][i]);

}

}

for (int i = 0; i <= n + 1; i++)

{

printf("%.2d\t", i);

}

printf("\n--------------------------------------\n");

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

{

printf("%.2d|\t", j);

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

{

printf("%d\t", matrix[j][i]);

}

printf("\n");

}

return matrix[m][n];

}

void sequence(int ss[], int m, int n, int matrix[m + 1][n + 1])

{

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

while (i > 0)

{

if (matrix[i][j] == matrix[i - 1][j])

ss[k] = 0, k--, i--;

else

ss[k] = 1, i--, j--, k--;

}

}

void print(char str[], int ss[], int m)

{

for (int i = 1, j = 0; i <= m, j < m; i++, j++)

{

if (ss[i])

printf("%c", str[j]);

}

printf("\n");

}

int main()

{

int matrix[MAX][MAX];

char str1[] = "KLOKMKNKLOK";

char str2[] = "KLLKNKLLKNYY";

int n = strlen(str1);

int m = strlen(str2);

int ss[m];

int k = LCS(str1, str2, m, n, matrix);

printf("String 1: %s\n", str1);

printf("String 2: %s\n", str2);

printf("Answer: %d\n", k);

sequence(ss, m, n, matrix);

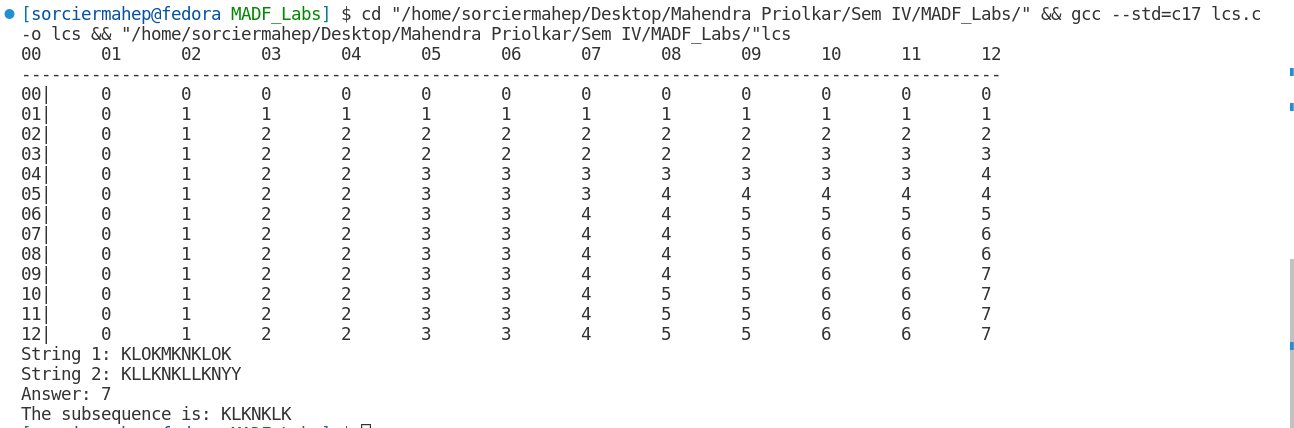
printf("The subsequence is: ");

print(str2, ss, m);

return 0;

}

**Output:**



**Conclusion:**

Internet algorithms were studied . The programs for

1. KMP Pattern Matching
2. BM Pattern Matching
3. Huffman Coding
4. Longest Common Subsequence

were studied and implemented successfully.