**VI: INTERNET ALGORITHMS Date:**

**Aim: -**Write algorithm and C program to implement the following problems using internet

algorithms

A) Boyer Moore Pattern Matching Algorithm

B) Knuth Morris Pratt Pattern Matching Algorithm

C) Huffman Encoding

D) Longest common subsequence

THEORY:

Text processing algorithms operate primarily on character strings. They involve interesting

methods for string pattern matching.

In the classic pattern matching problem on strings, we are given a text string of length n and a

pattern string P of length in, and want to find whether P is a sub-string of T. The notion of a

"match" is that there is a sub-string of T starting some index i that matches P, character by

character, so that

T[i]=P{O], T[i+1] =P[1], ..., T{i+in.1I =P[m i]

That is,

P=T[i..i+tn-1].

Thus, the output from a pattern matching algorithm is either an indication

pattern P does not exist in T or the starting index in T of a substring matching P.

Components of Pattern Matching

1. Patterns: A pattern represents a particular arrangement or sequence of elements that

we seek to identify within a larger dataset. These elements could be symbols,

numbers, shapes, or any type of data.

2. Target Data: This is the dataset or source where we want to locate instances of our

specified pattern.

3. Matching Algorithm: The algorithm defines the process by which patterns are

recognized or identified within the target data. It typically involves defining rules or

criteria for what constitutes a match.

Types of Pattern Matching

1. Exact Matching: The simplest form where the task is to locate occurrences of an exact

pattern within the target data. This is analogous to searching for a specific string of

characters within a text.

2. Approximate Matching: Involves finding patterns that are similar to, but not

necessarily identical to, a specified pattern. This can include tasks like finding similar

sequences in DNA or matching similar images.

Theoretical Perspectives

1. Formal Language Theory: Pattern matching can be studied through formal language

theory, which deals with the properties of languages and structures in terms of rules

and grammars. Regular expressions and context-free grammars are often used to

define patterns.

2. Algorithmic Complexity: Pattern matching algorithms can be analyzed in terms of

their computational complexity (e.g., time complexity and space complexity).

Efficient algorithms like the Knuth-Morris-Pratt (KMP) algorithm and the Boyer

Moore algorithm have been developed for string pattern matching.

3. Information Theory: Pattern matching can also be viewed from an information theory

perspective, where the goal is to quantify the amount of information gained or

reduced by identifying a particular pattern within a dataset.

Applications

1. Text Processing: Searching for specific words or phrases within documents.

2. Image and Signal Processing: Recognizing objects or signals based on predefined

templates.

3. Bioinformatics: Identifying similarities in genetic sequences.

4. Data Mining: Discovering recurring patterns in large datasets.

Challenges and Future Directions

1. Scalability: Efficient pattern matching in massive datasets remains a challenge.

2. Noise and Variability: Dealing with noisy or incomplete data where patterns may not

be exact.

3. Machine Learning Integration: Exploring how pattern matching can be enhanced or

automated using machine learning techniques.

4. Cross-domain Applications: Translating insights and techniques from one domain

(e.g., text) to another (e.g., images).

In summary, pattern matching is a rich area of study with diverse theoretical foundations and

practical applications. Its development continues to be driven by advancements in algorithm

design, computational resources, and interdisciplinary collaborations.

Problem Statement

Given:

 A text string 𝑇T of length 𝑛n.

 A pattern string 𝑃P of length 𝑚m.

The goal is to determine whether 𝑃 appears as a substring within 𝑇. A "match" occurs when

there exists an index 𝑖i such that 𝑇[𝑖]=𝑃[0], 𝑇[𝑖+1]=𝑃[1], and so on up to 𝑇[𝑖+𝑚−1]=𝑃[𝑚−1].

Output of the Pattern Matching Algorithm

The pattern matching algorithm's output can be:

 The starting index 𝑖 in 𝑇 where 𝑃 is found as a substring.

 An indication that 𝑃 does not exist as a substring in 𝑇.

Character Set and Alphabet

 We typically assume that the characters in 𝑇 and 𝑃 come from a well-defined

character set or alphabet denoted as Σ.

 Σ can be a finite set like ASCII, Unicode, or any custom-defined character set.

 Σ could also be more general and potentially infinite, although finite character sets are

common in practical applications like document processing.

Alphabet Size ∣Σ∣

 The size of the alphabet , Σ denoted as ∣Σ∣, represents the number of distinct

characters in Σ.

 ∣Σ∣ is typically considered a fixed constant in most scenarios.

The pattern matching problem is fundamental in computer science and is utilized in various

applications, such as text search, data mining, and string manipulation. Efficient algorithms

exist for solving this problem, and the choice of algorithm often depends on the

characteristics of the input strings and the size of the alphabet Σ. The problem is well-studied

and has many practical implementations that leverage various algorithmic techniques to

achieve optimal performance.

A] Boyer-Moole pattern matching algorithm

#include<stdio.h>

#include<string.h>

#define MAX 100

char p[MAX];                // pattern

char t[MAX];                // text

int cmp[MAX]={0};

int store;

int comparison\_count = 0;

int lastoccurence(char a) {

    int m = strlen(p);

    for(int i = m - 1; i >= 0; i--)

        if(p[i] == a)

            return i;

    return -1;

}

int min(int a, int b) {

    return (a <= b) ? a : b;

}

void printstring(int a, int i, int j) {

    if (a == 1) { // Printing comparison matrix

        int m = strlen(p);

        int n = strlen(t);

        for (int k = 1; k <= (i - j); k++)

            printf("  ");

        for (int k = (i - j) + 1; k <= n; k++) printf(" ");

        printf("\n");

        for(int k = 1; k <= (i - j); k++)

            printf("  ");

        printf("|");

        for (int idx = 0; idx < m; idx++) {

            if (idx >= j) printf("\e[1m%c\e[m|", p[idx]);

            else printf("%c|", p[idx]);

        }

        for(int idx = 1; idx <= store && idx <= n; idx++) printf("  ");

        printf("     i = %d", i);

        printf("\n");

        for (int k = 1; k <= (i - j); k++)

            printf("  ");

        printf("|");

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

            printf("%d|", cmp[idx]);

        for (int idx = 1; idx <= store && idx <= n; idx++) printf("  ");

        printf("     j = %d", j);

        printf("\n");

    }

    if (a == 0) { // Printing the text row

        int n = strlen(t);

        printf("|");

        for (int idx = 0; idx < n - 1; idx++)

            printf("%c|", t[idx]);

        printf("\n");

        for (int idx = 0; idx < n - 1; idx++) printf("  ");

        printf("\n");

    }

}

int BoyerMoole() {

    printstring(0,0,0);

    int m = strlen(p);

    int n = strlen(t);

    int i = m - 1;

    int j = m - 1;

    int flag=1;

    do {

        comparison\_count++;

        if(p[j] == t[i]) {

            cmp[j]++;

            if(j == 0){

                return i;

            }

            else {

                i--;

                j--;

            }

        } else {

            cmp[j]++;

            store=n-i-(m-j);

            printstring(1,i,j);

            i = i + m - min(j, lastoccurence(t[i]) + 1);

            j = m - 1;

        }

    } while(i <=n - 1);

    return -1;

}

int main() {

    FILE \*input = fopen("boyermoole.txt", "r");

    fgets(t, MAX, input);

    fgets(p, MAX, input);

    printf("Text = %sPattern = %s\n", t, p);

    int result = BoyerMoole();

    printstring(1,result,0);

    if (result != -1)

        printf("\nPattern found at: %d\n", result);

    else printf("\nPattern not found\n");

    printf("Number of comparisons made: %d\n", comparison\_count);

    return 0;

}

OUTPUT:

Text = abbcdbbabacabdabad

Pattern = abdab

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

| a | b | d | a | b | i=4 j=4 l=2

1

| a | b | d | a | b | i=5 j=3 l=4

3 2

| a | b | d | a | b | i=7 j=4 l=3

4

| a | b | d | a | b | i=6 j=2 l=4

7 6 5

| a | b | d | a | b | i=8 j=3 l=4

9 8

| a | b | d | a | b | i=10 j=4 l=3

10

| a | b | d | a | b | i=11 j=4 l=-1

11

| a | b | d | a | b | i=12 j=0

16 15 14 13 12

Pattern found at: 11

Number of comparisons made: 14

B] Knuth-Morris-Pratt pattern matching

#include <stdio.h>

#include <string.h>

#define MAX 50

int store;

int f[MAX];          // failure function

char p[MAX];         // pattern

char t[MAX];         // text

int cmp[MAX] = {0};  // comparison table

int comparisons = 0; // Total comparisons

void printt() {

    int n = strlen(t);

    printf("|");

    for (int i = 0; i < n - 1; i++) {

        printf("%c|", t[i]);

    }

    printf("\n");

}

void printp(int i, int j) {

    int m = strlen(p);

    int n = strlen(t);

    int l = (j == m) ? (n - m - i) : n - m - (i - j);

    if (j == m)

        for (int k = 0; k < i; k++)

            printf(" ");

    else

        for (int k = 0; k < (i - j); k++)

            printf(" ");

    /\*printf(":");

    if (j == m)

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

            printf(" :");

    else

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

            printf(" :");\*/

    printf("\n");

    if (j == m)

        for (int k = 0; k < i; k++)

            printf("  ");

    else

        for (int k = 0; k < (i - j); k++)

            printf("  ");

    printf("|");

    for (int k = 0; k < m; k++) {

        if (k < j)

            printf("\e[1m%c\e[m|", p[k]);

        else

            printf("%c|", p[k]);

    }

    int num\_spaces = (j == m) ? (l) : (l - (i - j));;

    //for (int k = 0; k < num\_spaces; k++) printf("  ");

    printf(" i= %d", i);

    printf("\n");

    if (j == m)

        for (int k = 0; k < i; k++)

            printf("  ");

    else

        for (int k = 0; k < (i - j); k++)

            printf("  ");

    printf("|");

    for (int k = 0; k < m; k++) {

        printf("%d|", cmp[k]);

    }

    num\_spaces = (j == m) ? (l) : (l - (i - j));

    //for (int k = 0; k < num\_spaces; k++) printf("  ");

    printf(" j= %d", j);

    printf("\n");

}

void failureFunction(int m) {

    f[0] = 0;

    int i = 1, j = 0;

    printf("Failure Function (f[]):\n|");

    for (int k = 0; k < m; k++)

        printf("%c|", p[k]);

    while (i < m) {

        if (p[i] == p[j]) {

            f[i] = j + 1;

            i++; j++;

        } else if (j > 0) {

            j = f[j - 1];

        } else {

            f[i] = 0;

            i++;

        }

    }

    printf("\n|");

    for (int k = 0; k < m; k++)

        printf("%d|", f[k]);

    printf("\n\n");

}

int KMP(int m, int n) {

    failureFunction(m);

    printt();

    int i = 0, j = 0;

    while (i < n) {

        comparisons++;

        if (t[i] == p[j]) {

            cmp[j]++;

            if (j == m - 1)

                return i - m + 1;

            i++;

            j++;

        } else if (j > 0) {

            store = i;

            cmp[j]++;

            printp(i, j);

            j = f[j - 1];

        } else {

            cmp[j]++;

            printp(i, j);

            i++;

        }

    }

    return -1;

}

int main() {

    int n, m;

    FILE \*input = fopen("knp.txt", "r");

    fgets(t, MAX, input);

    fgets(p, MAX, input);

    n = strlen(t);

    m = strlen(p);

    printf("Text = %sPattern = %s\n", t, p);

    int result = KMP(m, n);

    printp(result, m);

    if (result != -1)

        printf("\nPattern found at: %d\n", result);

    else printf("\nPattern not found\n");

    printf("Number of comparisons made: %d\n", comparisons);

    return 0;

}

Output:

Text = 110123110022311312012

Pattern = 13120

Failure Function (f[]):

| 1 | 3 | 1 | 2 | 0 |

| 0 | 0 | 1 | 0 | 0 |

| 1 | 1 | 0 | 1 | 2 | 3 | 1 | 1 | 0 | 0 | 2 | 2 | 3 | 1 | 1 | 3 | 1 | 2 | 0 | 1 | 2 |

| 1 | 3 | 1 | 2 | 0 |

1 2

| 1 | 3 | 1 | 2 | 0 |

3 4

| 1 | 3 | 1 | 2 | 0 |

5

| 1 | 3 | 1 | 2 | 0 |

6 7

| 1 | 3 | 1 | 2 | 0 |

8

| 1 | 3 | 1 | 2 | 0 |

9

| 1 | 3 | 1 | 2 | 0 |

10 11

| 1 | 3 | 1 | 2 | 0 |

12 13

| 1 | 3 | 1 | 2 | 0 |

14

| 1 | 3 | 1 | 2 | 0 |

15

| 1 | 3 | 1 | 2 | 0 |

16

| 1 | 3 | 1 | 2 | 0 |

17

| 1 | 3 | 1 | 2 | 0 |

18

| 1 | 3 | 1 | 2 | 0 |

19 20

| 1 | 3 | 1 | 2 | 0 |

21 22 23 24 25

Pattern found at: 14

Number of comparisons made: 25

C] Huffman Encoding

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX 100

struct node {

    int freq;

    char c;

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

};

char text[MAX];             // Text

struct node \*start = NULL;  // Queue

int greatest = 0;           // this holds the greatest freq

int treeCount = 1;          // To keep track of the number of trees formed

void addatend(char c[], int f[], int n) {

    for (int i = 0; i <= n; i++) {

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

        temp->c = c[i];

        temp->freq = f[i];

        temp->next = NULL;

        temp->left = NULL;

        temp->right = NULL;

        struct node \*ptr = start;

        struct node \*prev = NULL;

        while (ptr != NULL && (ptr->freq < temp->freq || (ptr->freq == temp->freq && ptr->c < temp->c))) {

            prev = ptr;

            ptr = ptr->next;

        }

        if (prev == NULL) {

            temp->next = start;

            start = temp;

        } else {

            temp->next = ptr;

            prev->next = temp;

        }

    }

}

void showlist() {

    printf("\n");

    printf("Queue    :");

    struct node \*ptr = start;

    while (ptr != NULL) {

        printf(" %c ", ptr->c);

        ptr = ptr->next;

    }

    printf("\n");

    printf("Frequency:");

    ptr = start;

    while (ptr != NULL) {

        printf(" %d ", ptr->freq);

        ptr = ptr->next;

    }

    printf("\n");

}

void findgreatest() {

    if (start == NULL)

        return;

    struct node \*ptr = start;

    greatest = ptr->freq;

    while (ptr != NULL) {

        if (ptr->freq > greatest)

            greatest = ptr->freq;

        ptr = ptr->next;

    }

}

struct node \*delfront() {

    if (start == NULL)

        return NULL;

    struct node \*temp = start;

    start = start->next;

    return temp;

}

void addafterliketerms(struct node \*temp) {

    struct node \*ptr = start;

    struct node \*prev = NULL;

    while (ptr != NULL && (ptr->freq < temp->freq || (ptr->freq == temp->freq && ptr->c < temp->c))) {

        prev = ptr;

        ptr = ptr->next;

    }

    if (prev == NULL) {

        temp->next = start;

        start = temp;

    } else {

        temp->next = ptr;

        prev->next = temp;

    }

}

void HuffmanEncoding() {

    if (start == NULL) return;

    int iteration = 1;

    while (start->next != NULL) {

        printf("Iteration %d: ", iteration++);

        showlist();

        struct node \*a = delfront();

        struct node \*b = delfront();

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

        temp->c = '#';

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

        temp->left = a;

        temp->right = b;

        temp->next = NULL;

        if (temp->freq >= greatest)

            greatest = temp->freq;

        printf("t%d (%c%d) + t%d (%c%d) -> t%d (%c%d)\n", treeCount, a->c, a->freq, treeCount + 1, b->c, b->freq, treeCount + 2, temp->c, temp->freq);

        treeCount += 2;

        addafterliketerms(temp);

    }

}

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->c);

    }

    path[stringlen] = 0;

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

    path[stringlen] = 1;

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

}

void printTree(struct node \*root, int level) {

    if (root == NULL) return;

    printTree(root->right, level + 1);

    printf("\n");

    for (int i = 0; i < level; i++)

        printf("    ");

    if (root->c == '#')

        printf("\*");

    else printf("%c", root->c);

    printf("(%d)", root->freq);

    printTree(root->left, level + 1);

}

int main() {

    printf("Enter the text: ");

    fgets(text, MAX, stdin);

    text[strcspn(text, "\n")] = '\0';

    int freq[256] = {0};

    int n = strlen(text);

    for (int i = 0; i < n; i++)

        freq[(int)text[i]]++;

    char arr[MAX];

    int unique\_freq[MAX];

    int unique\_count = 0;

    for (int i = 0; i < 256; i++) {

        if (freq[i] > 0) {

            arr[unique\_count] = (char)i;

            unique\_freq[unique\_count] = freq[i];

            unique\_count++;

        }

    }

    addatend(arr, unique\_freq, unique\_count - 1);

    printf("Initial list: ");

    showlist();

    findgreatest();

    HuffmanEncoding();

    printf("\nHuffman Tree:\n");

    printTree(start, 0);

    printf("\n");

    printf("\nHuffman Codes:\n");

    int path[MAX];

    printcodes(start, path, 0);

    return 0;

}

Output:

Enter the text: three missionaries and three cannibals

Initial list:

Queue    : b  c  d  l  m  o  h  t  r     a  i  n  s  e

Frequency: 1  1  1  1  1  1  2  2  3  4  4  4  4  4  5

Iteration 1:

Queue    : b  c  d  l  m  o  h  t  r     a  i  n  s  e

Frequency: 1  1  1  1  1  1  2  2  3  4  4  4  4  4  5

t1 (b1) + t2 (c1) -> t3 (#2)

Iteration 2:

Queue    : d  l  m  o  #  h  t  r     a  i  n  s  e

Frequency: 1  1  1  1  2  2  2  3  4  4  4  4  4  5

t3 (d1) + t4 (l1) -> t5 (#2)

Iteration 3:

Queue    : m  o  #  #  h  t  r     a  i  n  s  e

Frequency: 1  1  2  2  2  2  3  4  4  4  4  4  5

t5 (m1) + t6 (o1) -> t7 (#2)

Iteration 4:

Queue    : #  #  #  h  t  r     a  i  n  s  e

Frequency: 2  2  2  2  2  3  4  4  4  4  4  5

t7 (#2) + t8 (#2) -> t9 (#4)

Iteration 5:

Queue    : #  h  t  r     #  a  i  n  s  e

Frequency: 2  2  2  3  4  4  4  4  4  4  5

t9 (#2) + t10 (h2) -> t11 (#4)

Iteration 6:

Queue    : t  r     #  #  a  i  n  s  e

Frequency: 2  3  4  4  4  4  4  4  4  5

t11 (t2) + t12 (r3) -> t13 (#5)

Iteration 7:

Queue    :    #  #  a  i  n  s  #  e

Frequency: 4  4  4  4  4  4  4  5  5

t13 ( 4) + t14 (#4) -> t15 (#8)

Iteration 8:

Queue    : #  a  i  n  s  #  e  #

Frequency: 4  4  4  4  4  5  5  8

t15 (#4) + t16 (a4) -> t17 (#8)

Iteration 9:

Queue    : i  n  s  #  e  #  #

Frequency: 4  4  4  5  5  8  8

t17 (i4) + t18 (n4) -> t19 (#8)

Iteration 10:

Queue    : s  #  e  #  #  #

Frequency: 4  5  5  8  8  8

t19 (s4) + t20 (#5) -> t21 (#9)

Iteration 11:

Queue    : e  #  #  #  #

Frequency: 5  8  8  8  9

t21 (e5) + t22 (#8) -> t23 (#13)

Iteration 12:

Queue    : #  #  #  #

Frequency: 8  8  9  13

t23 (#8) + t24 (#8) -> t25 (#16)

Iteration 13:

Queue    : #  #  #

Frequency: 9  13  16

t25 (#9) + t26 (#13) -> t27 (#22)

Iteration 14:

Queue    : #  #

Frequency: 16  22

t27 (#16) + t28 (#22) -> t29 (#38)

Huffman Tree:

a(4)

\*(8)

(4)

\*(13)

r(3)

\*(5)

o(1)

\*(2)

m(1)

\*(22)

e(5)

\*(9)

l(1)

\*(2)

d(1)

\*(4)

c(1)

\*(2)

b(1)

\*(38)

t(2)

\*(4)

h(2)

\*(8)

s(4)

\*(16)

n(4)

\*(8)

i(4)

Huffman Codes:

000 i

001 n

010 s

0110 h

0111 t

10000 b

10001 c

10010 d

10011 l

101 e

11000 m

11001 o

1101 r

1110

1111 a

D] Longest Common Subsequence

#include <stdio.h>

#include <string.h>

#include <locale.h>

#define MAX 20

#define UP\_ARROW '^'

#define LEFT\_ARROW '<'

#define DIAGONAL\_ARROW '\\'

char X[MAX]; //Pattern to be searched to

char Y[MAX]; //Pattern to be searched against

int L[MAX][MAX];//Matrix to store the DP result

int sub1[MAX]; //Solution for left method

int sub2[MAX]; //Solution for right method

int leftSequence[MAX][MAX] = {0}; //Computes if left arrow needed

int upSequence[MAX][MAX] = {0}; //Computes if up arrow needed

int maxleft(int a, int b){

    return (a > b) ? a : b;

}

int maxup(int a, int b){

    return (a >= b) ? a : b;

}

void LCS(int flag) {

    int n = strlen(X);

    int m = strlen(Y);

    for (int i = 0; i <= n; i++)

        L[i][0] = 0;

    for (int j = 0; j <= m; j++)

        L[0][j] = 0;

    if(flag == 0){

        for(int i = 1; i <= n; i++) {

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

                if (X[i - 1] == Y[j - 1])

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

                else

                    L[i][j] = maxleft(L[i - 1][j], L[i][j - 1]);

            }

        }

    }

    else{

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

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

                if (X[i - 1] == Y[j - 1])

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

                else

                    L[i][j] = maxup(L[i - 1][j], L[i][j - 1]);

            }

        }

    }

}

void traverse(int flag){

    int n = strlen(X);

    int m = strlen(Y);

    if (flag == 0) {

        int i = n;

        int j = m;

        int c = L[n][m];

        while (c > 0) {

            if (X[i - 1] == Y[j - 1]) {

                leftSequence[i][j] = DIAGONAL\_ARROW;

                sub1[c] = X[i - 1];

                c--; i--; j--;

            } else {

                if (L[i - 1][j] > L[i][j - 1]) {

                    leftSequence[i][j] = UP\_ARROW;

                    i--;

                } else {

                    leftSequence[i][j] = LEFT\_ARROW;

                    j--;

                }

            }

        }

    } else {

        int i = n;

        int j = m;

        int c = L[n][m];

        while (c > 0) {

            if (X[i - 1] == Y[j - 1]) {

                upSequence[i][j] = DIAGONAL\_ARROW;

                sub2[c] = X[i - 1];

                c--; i--; j--;

            } else {

                if (L[i - 1][j] >= L[i][j - 1]) {

                    i--;

                    upSequence[i][j] = UP\_ARROW;

                } else {

                    j--;

                    upSequence[i][j] = LEFT\_ARROW;

                }

            }

        }

    }

}

int main() {

    printf("Enter 2 strings :\n");

    scanf(" %s",X);

    scanf(" %s",Y);

    int n = strlen(X);

    int m = strlen(Y);

    printf("X = %s\nY = %s\n", X, Y);

    LCS(0);

    traverse(0);

    printf("Left Method:\n");

    for (int i = 0; i <= n; i++) {

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

            if (leftSequence[i][j] == DIAGONAL\_ARROW) {

                printf(" %c", DIAGONAL\_ARROW); printf("\e[1m%d\e[m",L[i][j]);

            } else if (leftSequence[i][j] == UP\_ARROW) {

                printf(" %c", UP\_ARROW);printf("\e[1m%d\e[m",L[i][j]);

            } else if (leftSequence[i][j] == LEFT\_ARROW) {

                printf("%c ", LEFT\_ARROW);printf("\e[1m%d\e[m",L[i][j]);

            } else {

                printf("  %d", L[i][j]);

            }

        }

        printf("\n");

    }

    printf("Longest Subsequence: ");

    for (int i = 1; i <= L[n][m]; i++)

        printf("%c ", sub1[i]);

    printf("\n\nUp Methods:\n");

    traverse(1);

    for (int i = 0; i <= n; i++) {

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

            if (upSequence[i][j] == DIAGONAL\_ARROW) {

                printf(" %c", DIAGONAL\_ARROW);printf("\e[1m%d\e[m",L[i][j]);

            } else if (upSequence[i][j] == UP\_ARROW) {

                printf(" %c", UP\_ARROW);printf("\e[1m%d\e[m",L[i][j]);

            } else if (upSequence[i][j] == LEFT\_ARROW) {

                printf("%c ", LEFT\_ARROW);printf("\e[1m%d\e[m",L[i][j]);

            } else {

                printf("  %d", L[i][j]);

            }

        }

        printf("\n");

    }

    printf("Longest Subsequence: ");

    for (int i = 1; i <= L[n][m]; i++)

        printf("%c ", sub2[i]);

    return 0;

}

Output:

Enter 2 strings :

ABBACADABCA

BBACABADDABBCA

X = ABBACADABCA

Y = BBACABADDABBCA

Left Method:

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 1 1 1 1 1 1 1 1 1 1 1 1

0 \1 1 1 1 1 2 2 2 2 2 2 2 2 2

0 1 \2 2 2 2 2 2 2 2 2 3 3 3 3

0 1 2 \3 3 3 3 3 3 3 3 3 3 3 4

0 1 2 3 \4< 4< 4 4 4 4 4 4 4 4 4

0 1 2 3 4 5 5 \5< 5 5 5 5 5 5 5

0 1 2 3 4 5 5 5 6 \6 6 6 6 6 6

0 1 2 3 4 5 5 6 6 6 \7< 7 7 7 7

0 1 2 3 4 5 6 6 6 6 7 8 \8 8 8

0 1 2 3 4 5 6 6 6 6 7 8 8 \9 9

0 1 2 3 4 5 6 7 7 7 7 8 8 9 \10

Longest Subsequence: B B A C A D A B C A

Up Methods:

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 1 1 1 1 1 1 1 1 1 1 1 1

0 \1 1 1 1 1 2 2 2 2 2 2 2 2 2

0 1 \2 2 2 2 2 2 2 2 2 3 3 3 3

0 1 2 \3 3 3 3 3 3 3 3 3 3 3 4

0 1 2 3 \4< 4 4 4 4 4 4 4 4 4 4

0 1 2 3 4 5 5 \5 5 5 5 5 5 5 5

0 1 2 3 4 5 5 5 6 \6 6 6 6 6 6

0 1 2 3 4 5 5 6 6 6 \7 7 7 7 7

0 1 2 3 4 5 6 6 6 6 7 8 \8 8 8

0 1 2 3 4 5 6 6 6 6 7 8 8 \9 9

0 1 2 3 4 5 6 7 7 7 7 8 8 9 \10

Longest Subsequence: B B A C A D A B C A