Program: 4

To implement Longest Common Subsequence problem and analyse its time complexity.

#include <bits/stdc++.h>

**using** **namespace** std;

**int** max(**int** a, **int** b);

/\* Returns length of LCS for X[0..m-1], Y[0..n-1] \*/

**int** lcs( **char** \*X, **char** \*Y, **int** m, **int** n )

{

**if** (m == 0 || n == 0)

**return** 0;

**if** (X[m-1] == Y[n-1])

**return** 1 + lcs(X, Y, m-1, n-1);

**else**

**return** max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));

}

**int** max(**int** a, **int** b)

{

**return** (a > b)? a : b;

}

/\* Driver code \*/

**int** main()

{

**char** X[] = "AGGTAB";

**char** Y[] = "GXTXAYB";

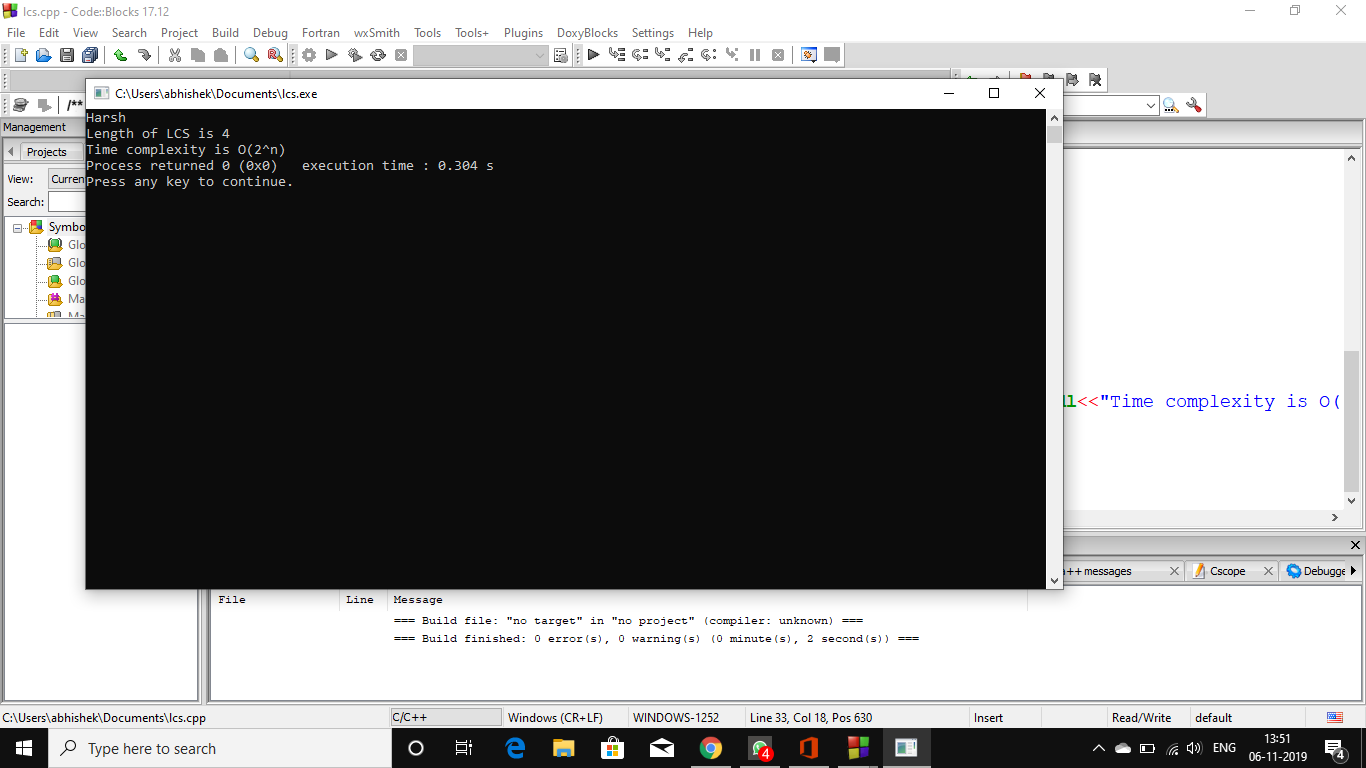
**int** m = **strlen**(X);

**int** n = **strlen**(Y);

cout<<”Abhishek mishra”<<"Length of LCS is "<< lcs( X, Y, m, n ) ;

**return** 0; }

Output:



PROGRAM-5

To implement Huffman Coding and analyse its time complexity.

// C++ program for Huffman Coding

#include <bits/stdc++.h>

using namespace std;

// A Huffman tree node

struct MinHeapNode {

// One of the input characters

char data;

// Frequency of the character

unsigned freq;

// Left and right child

MinHeapNode \*left, \*right;

MinHeapNode(char data, unsigned freq)

{

left = right = NULL;

this->data = data;

this->freq = freq;

}

};

// For comparison of

// two heap nodes (needed in min heap)

struct compare {

bool operator()(MinHeapNode\* l, MinHeapNode\* r)

{

return (l->freq > r->freq);

}

};

// Prints huffman codes from

// the root of Huffman Tree.

void printCodes(struct MinHeapNode\* root, string str)

{

if (!root)

return;

if (root->data != '$')

cout << root->data << ": " << str << "\n";

printCodes(root->left, str + "0");

printCodes(root->right, str + "1");

}

// The main function that builds a Huffman Tree and

// print codes by traversing the built Huffman Tree

void HuffmanCodes(char data[], int freq[], int size)

{

struct MinHeapNode \*left, \*right, \*top;

// Create a min heap & inserts all characters of data[]

priority\_queue<MinHeapNode\*, vector<MinHeapNode\*>, compare> minHeap;

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

minHeap.push(new MinHeapNode(data[i], freq[i]));

// Iterate while size of heap doesn't become 1

while (minHeap.size() != 1) {

// Extract the two minimum

// freq items from min heap

left = minHeap.top();

minHeap.pop();

right = minHeap.top();

minHeap.pop();

// Create a new internal node with

// frequency equal to the sum of the

// two nodes frequencies. Make the

// two extracted node as left and right children

// of this new node. Add this node

// to the min heap '$' is a special value

// for internal nodes, not used

top = new MinHeapNode('$', left->freq + right->freq);

top->left = left;

top->right = right;

minHeap.push(top);

}

// Print Huffman codes using

// the Huffman tree built above

printCodes(minHeap.top(), "");

}

// Driver program to test above functions

int main()

{

char arr[] = { 'a', 'b', 'c', 'd', 'e', 'f' };

int freq[] = { 5, 9, 12, 13, 16, 45 };

int size = sizeof(arr) / sizeof(arr[0]);

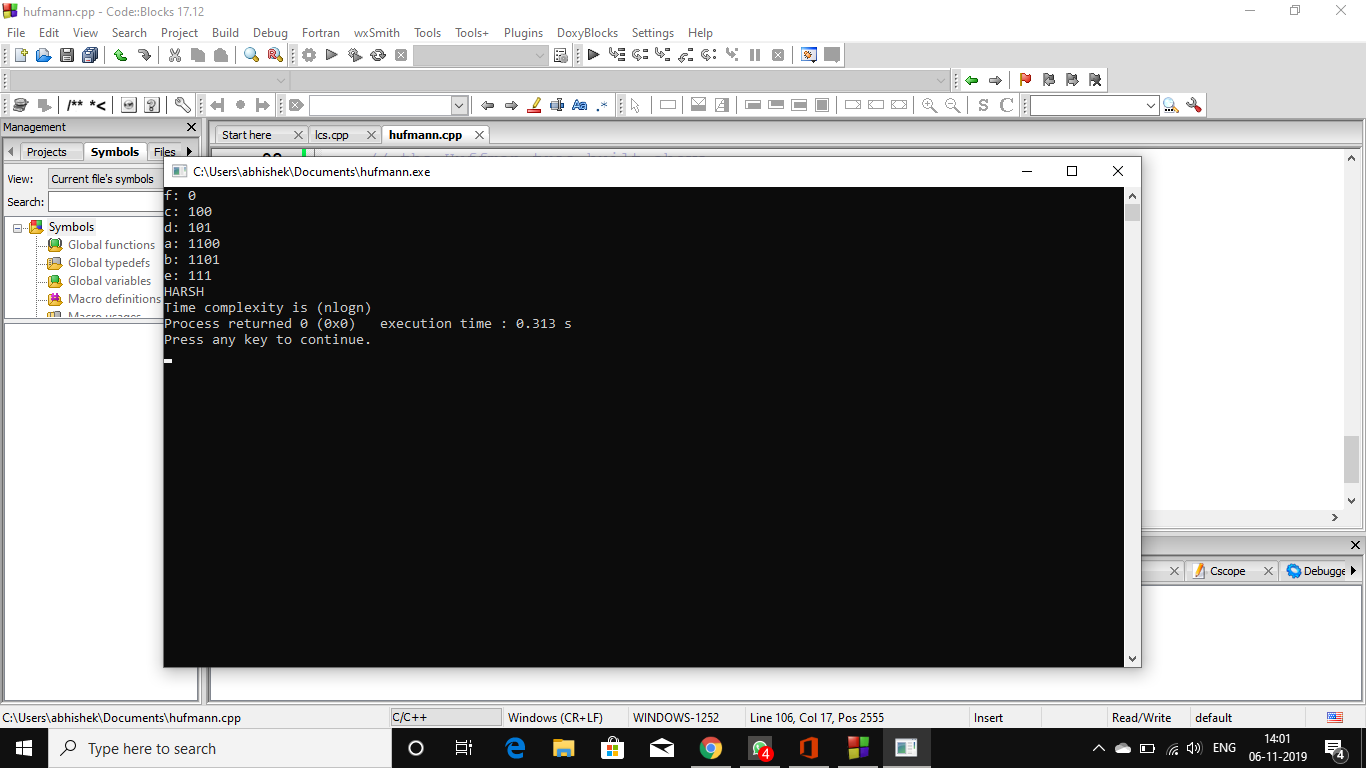
HuffmanCodes(arr, freq, size);

cout<<"DHANANJAY" <<endl<<"Time complexity is (nlogn)" ;

return 0;

}

OUTPUT:



PROGRAM –8

To implement naïve String Matching algorithm, Rabin Karp algorithm and Knuth Morris Pratt algorithm and analyse its time complexity.

-------------Naïve string matching---------------

#include<stdio.h>

#include<string.h>

void search(char \*pat, char \*txt)

{

int M = strlen(pat);

int N = strlen(txt);

/\* A loop to slide pat[] one by one \*/

for (int i = 0; i <= N - M; i++)

{

int j;

/\* For current index i, check for pattern match \*/

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

if (txt[i+j] != pat[j])

break;

if (j == M) // if pat[0...M-1] = txt[i, i+1, ...i+M-1]

printf("Pattern found at index %d \n", i);

}

}

/\* Driver program to test above function \*/

int main()

{

char txt[] = "AABAACAADAABAAABAA";

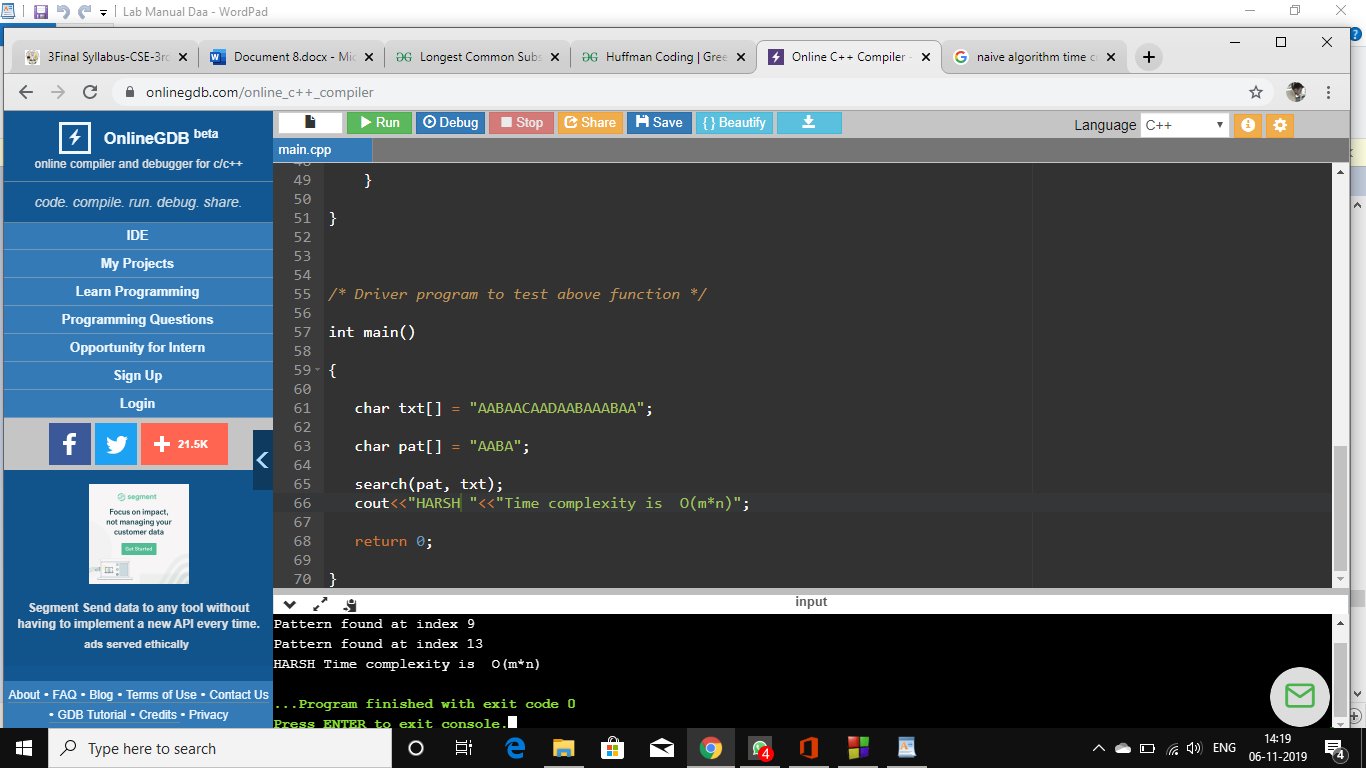
char pat[] = "AABA";

search(pat, txt);

return 0;

}

Output



-----------------Rabin carp-----------------------

#include<stdio.h>

#include<string.h>

// d is the number of characters in input alphabet

#define d 256

/\* pat -> pattern

txt -> text

q -> A prime number

\*/

void search(char \*pat, char \*txt, int q)

{

int M = strlen(pat);

int N = strlen(txt);

int i, j;

int p = 0; // hash value for pattern

int t = 0; // hash value for txt

int h = 1;

// The value of h would be "pow(d, M-1)%q"

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

h = (h\*d)%q;

// Calculate the hash value of pattern and first window of text

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

{

p = (d\*p + pat[i])%q;

t = (d\*t + txt[i])%q;

}

// Slide the pattern over text one by one

for (i = 0; i <= N - M; i++)

{

// Check the hash values of current window of text and pattern

// If the hash values match then only check for characters on by one

if ( p == t )

{

/\* Check for characters one by one \*/

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

{

if (txt[i+j] != pat[j])

break;

}

if (j == M) // if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]

{

printf("Pattern found at index %d \n", i);

}

}

// Calculate hash value for next window of text: Remove leading digit,

// add trailing digit

if ( i < N-M )

{

t = (d\*(t - txt[i]\*h) + txt[i+M])%q;

// We might get negative value of t, converting it to positive

if(t < 0)

t = (t + q);

}

}

}

/\* Driver program to test above function \*/

int main()

{

char \*txt = "PILLOWS FOR PILLOWS";

char \*pat = "PILLOW";

int q = 101; // A prime number

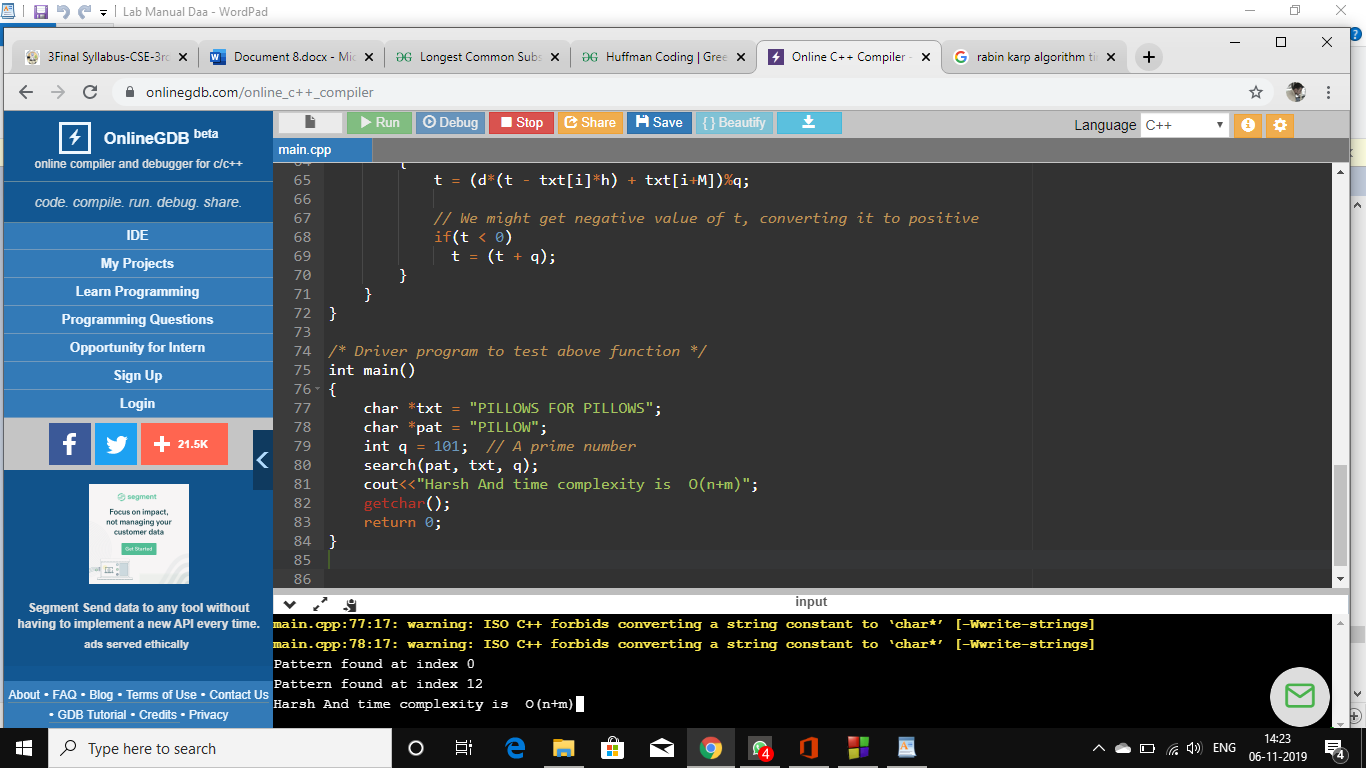
search(pat, txt, q);

getchar();

return 0;

}

Output



-------------Knuth morris pratt--------------

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

void computeLPSArray(char \*pat, int M, int \*lps);

void KMPSearch(char \*pat, char \*txt)

{

int M = strlen(pat);

int N = strlen(txt);

// create lps[] that will hold the longest prefix suffix

// values for pattern

int \*lps = (int \*)malloc(sizeof(int)\*M);

int j = 0; // index for pat[]

// Preprocess the pattern (calculate lps[] array)

computeLPSArray(pat, M, lps);

int i = 0; // index for txt[]

while (i < N)

{

if (pat[j] == txt[i])

{

j++;

i++;

}

if (j == M)

{

printf("Found pattern at index %d \n", i-j);

j = lps[j-1];

}

// mismatch after j matches

else if (i < N && pat[j] != txt[i])

{

// Do not match lps[0..lps[j-1]] characters,

// they will match anyway

if (j != 0)

j = lps[j-1];

else

i = i+1;

}

}

free(lps); // to avoid memory leak

}

void computeLPSArray(char \*pat, int M, int \*lps)

{

int len = 0; // length of the previous longest prefix suffix

int i;

lps[0] = 0; // lps[0] is always 0

i = 1;

// the loop calculates lps[i] for i = 1 to M-1

while (i < M)

{

if (pat[i] == pat[len])

{

len++;

lps[i] = len;

i++;

}

else // (pat[i] != pat[len])

{

if (len != 0)

{

// This is tricky. Consider the example

// AAACAAAA and i = 7.

len = lps[len-1];

// Also, note that we do not increment i here

}

else // if (len == 0)

{

lps[i] = 0;

i++;

}

}

}

}

// Driver program to test above function

int main()

{

char \*txt = "ABABDABACDABABCABAB";

char \*pat = "ABABCABAB";

KMPSearch(pat, txt);

return 0;

}

Output

