**String**

### Introduction of string

#include <bits/stdc++.h>

using namespace std;

int main()

{

    string str="geeksforgeeks";

    int count[26]={0};

    for(int i=0;i<str.length();i++){

        count[str[i]-'a']++;

    }

    for(int i=0;i<26;i++){

        if(count[i]>0){

            cout<<char(i+'a')<<" "<<count[i]<<endl;

        }

    }

}

### OUTPUT :

e 4

f 1

g 2

k 2

o 1

r 1

s 2

### String in C++

### Program 1:

#include<iostream>

using namespace std;

int main()

{

    char str[]="gfg";

    cout<<str;

    return 0;

}

OUTPUT : gfg

### Program 2:

#include<iostream>

using namespace std;

int main()

{

    char str[]="gfg";

    cout<<sizeof(str);

    return 0;

}

OUTPUT : 4

### Program 3 :

#include<iostream>

using namespace std;

int main()

{

    char str[]={'g','f','g'};

    cout<<str;

    return 0;

}

OUTPUT : gfg

### Program 4 :

#include<iostream>

using namespace std;

int main()

{

    char str[]={'g','f','g','\0'};

    cout<<str;

    return 0;

}

OUTPUT : gfg

### Program 5 :

#include<iostream>

#include<cstring>

using namespace std;

int main()

{

    char s1[]="bcd";

    char s2[]="abc";

    int res=strcmp(s1,s2);

    // cout<<res;

    if(res>0)

        cout<<"Greater";

    else if(res==0)

        cout<<"Same";

    else

        cout<<"Smaller";

}

OUTPUT : Greater

### Program 6 :

#include<iostream>

#include<cstring>

using namespace std;

int main()

{

    char s1[]="gfg";

    char s2[]="bcd";

    int res=strcmp(s1,s2);

    // cout<<res;

    if(res>0)

        cout<<"Greater";

    else if(res==0)

        cout<<"Same";

    else

        cout<<"Smaller";

}

OUTPUT : Greater

### Program 7 :

#include<iostream>

#include<cstring>

using namespace std;

int main()

{

    char s1[]="bcd";

    char s2[]="bcd";

    int res=strcmp(s1,s2);

    if(res>0)

        cout<<"Greater";

    else if(res==0)

        cout<<"Same";

    else

        cout<<"Smaller";

}

OUTPUT : Same

### Program 9 :

#include<iostream>

#include<cstring>

using namespace std;

int main()

{

    char str[5];

    strcpy(str,"gfg");

    cout<<str;

    return 0;

}

OUTPUT : gfg

### Program 10 :

#include<iostream>

#include<cstring>

using namespace std;

int main()

{

    string str ="geeksforgeeks";

    cout<<str.length()<<" ";

    str=str + "xyz";

    cout<<str<<" ";

    cout<<str.substr(1,3)<<" ";

    cout<<str.find("eek")<<" ";

    return 0;

}

OUTPUT : 13 geeksforgeeksxyz eek 1

### Program 11 :

#include<iostream>

#include<cstring>

using namespace std;

int main()

{

    char s1[]="abc";

    char s2[]="bcd";

    if(s1==s2)

        cout<<"Same";

    else if(s1<s2)

        cout<<"Smaller";

    else

        cout<<"Greater";

}

OUTPUT : Greater

### Program 12 :

#include<iostream>

#include<cstring>

using namespace std;

int main()

{

    string str;

    cout<<"Enter Your name ";

    cin>>str;

    cout<<"\nYour name is "<<str;

    return 0;

}

OUTPUT :

Enter Your name Ganesh

Your name is Ganesh

### Program 13 :

#include<iostream>

#include<cstring>

using namespace std;

int main()

{

    string str;

    cout<<"Enter Your name ";

    cin>>str;

    cout<<"\nYour name is "<<str;

    return 0;

}

OUTPUT : Enter Your name Ganesh

Your name is Ganesh

### Program 15 :

#include<iostream>

#include<cstring>

using namespace std;

int main()

{

    string str;

    cout<<"Enter Your name ";

    getline(cin,str,'a');

    cout<<"\nYour name is "<<str;

    return 0;

}

OUTPUT : Enter Your name Ganesh

Your name is G

### Program 16 :

#include<iostream>

#include<cstring>

using namespace std;

int main()

{

    string str="geeksforgeeks";

    for(int i=0;i<str.length();i++)

        cout<<str[i];

    cout<<endl;

    for(char x:str)

        cout<<x;

}

OUTPUT :

geeksforgeeks

geeksforgeeks

### Palindrome Check

### Naïve :

#include<bits/stdc++.h>

using namespace std;

//it require theta(n) time and theta (n) auxiliary space

bool isPal(string str)

{

    string rev=str;

    reverse(rev.begin(),rev.end());

    return rev==str;

}

int main()

{

    string str="aba";

    cout<<isPal(str);

    return 0;

}

### OUTPUT :

### 1

### Efficient for Palindrome check :

#include<bits/stdc++.h>

using namespace std;

//it require o(n) time and o(1) extra space

bool isPal(string str)

{

    int begin=0;

    int end=str.length()-1;

    while(begin<end)

    {

        if(str[begin]!=str[end])

            return false;

        begin++;

        end--;

    }

    return true;

}

int main()

{

    string str="aba";

    cout<<isPal(str);

    return 0;

}

### OUTPUT :

### 1

### Check if string is Subsequence of other

### A String is a subsequence of a given String, that is generated by deleting some character of a given string without changing its order. Examples: Input : abc Output : a, b, c, ab, bc, ac, abc Input : aaa Output : a, aa, aaa.

### Iterative solution :

#include<bits/stdc++.h>

using namespace std;

bool isSubSeq(string s1, string s2, int n, int m)

{

    int j=0;

    for(int i=0; i<n && j<m; i++)

    {

        if(s1[i]==s2[j])

        j++;

    }

    return j==m;

}

int main()

{

    int n,m;

    string s1,s2;

    cin>>n>>m;

    cin>>s1>>s2;

    cout<<boolalpha<<isSubSeq(s1,s2,n,m);

    return 0;

}

### OUTPUT :

### False

### Efficient Recursive for if string is subsequence of other :

//time complexity is O(m+n) and auxilary space is O(m+n)

#include<iostream>

using namespace std;

bool isSubSeq(string s1, string s2, int n, int m)

{

    if(m==0)

        return true;

    if(n==0)

        return false;

    if(s1[n-1]==s2[m-1])

        return isSubSeq(s1,s2,n-1,m-1);

    else

        return isSubSeq(s1,s2,n-1,m);

}

int main()

{

    int n,m;

    string s1,s2;

    cin>>n>>m;

    cin>>s1>>s2;

    cout<<boolalpha<<isSubSeq(s1,s2,n,m);

    return 0;

}

### OUTPUT :

### false

### Check for Anagram :

### An anagram of a string is another string that contains the same characters, only the order of characters can be different. For example, “abcd” and “dabc” are an anagram of each other.

### Naïve :

#include<bits/stdc++.h>

using namespace std;

bool areAnagram(string &s1, string &s2)

{

    int n1=s1.length();

    int n2=s2.length();

    if(n1!=n2)

        return false;

    sort(s1.begin(),s1.end());

    sort(s2.begin(),s2.end());

    return (s1==s2);

}

int main()

{

    string str1="abaac";

    string str2="aacba";

    if(areAnagram(str1,str2))

        cout<<"The Two String are anagram of each other";

    else

        cout<<"the two string are not anagram of each other ";

    return 0;

}

### OUTPUT :

### The Two String are anagram of each other

### Efficient for check for anagram :

#include<bits/stdc++.h>

using namespace std;

const int CHAR=256;

bool areAnagram(string &s1, string &s2)

{

    int n1=s1.length();

    int n2=s2.length();

    if(n1!=n2)

        return false;

    int count[CHAR]={0};

    for(int i=0;i<s1.length();i++)

    {

        count[s1[i]]++;

        count[s2[i]]--;

    }

    for(int i=0;i<CHAR;i++)

        if(count[i]!=0)

            return false;

    return true;

}

int main()

{

    string str1 = "abaac";

    string str2 = "aacba";

    if (areAnagram(str1, str2))

        cout << "The two strings are anagram of each other";

    else

        cout << "The two strings are not anagram of each other";

    return 0;

}

### OUTPUT :

### The two strings are anagram of each other

### Leftmost repeating character :

### Given a string, the task is to find the first character (whose leftmost appearance is first) that repeats

### Naïve :

//the task is to find the first character (whose leftmost appearance is first) that repeats

#include<bits/stdc++.h>

using namespace std;

int leftMost(string &str)

{

    for(int i=0;i<str.length();i++)

    {

        for(int j=i+1;j<str.length();j++)

        {

            if(str[i]==str[j])

                return i;

        }

    }

    return -1;

}

int main()

{

    string str="geeksforgeeks";

    cout<<"Index of leftmost repeating character : "<<endl;

    cout<<leftMost(str)<<endl;

    return 0;

}

### OUTPUT :

### 0

### Better Approach for leftmost repeating character :

// the task is to find the first character (whose leftmost appearance is first) that repeats

#include<bits/stdc++.h>

using namespace std;

const int CHAR=256;

int leftMost(string &str)

{

    int count[CHAR]={0};

    for(int i=0;i<str.length();i++)

        count[str[i]]++;

    for(int i=0;i<str.length();i++)

        if(count[str[i]]>1)

            return i;

    return -1;

}

int main()

{

    string str = "geeksforgeeks";

    cout<<"Index of leftmost repeating character:"<<endl;

    cout<<leftMost(str)<<endl;

    return 0;

}

### OUTPUT :

### 0

### Efficient approach -1 for leftmost repeating character :

//the task is to find the first character (whose leftmost appearance is first) that repeats

#include<bits/stdc++.h>

using namespace std;

const int CHAR=256;

int leftMost(string &str)

{

    int fIndex[CHAR]={0};

    fill(fIndex,fIndex+CHAR,-1);

    int res=INT32\_MAX;

    for(int i=0;i<str.length();i++)

    {

        int fi=fIndex[str[i]];

        if(fi==-1)

            fIndex[str[i]]=i;

        else

            res=min(res,fi);

    }

    return (res==INT32\_MAX)? -1:res;

}

int main()

{

    string str = "geeksforgeeks";

    cout<<"Index of leftmost repeating character:"<<endl;

    cout<<leftMost(str)<<endl;

    return 0;

}

### OUTPUT :

### Index of leftmost repeating character:

### 0

### Efficient approach -2 for leftmost repeating character :

//the task is to find the first character (whose leftmost appearance is first) that repeats.

#include<bits/stdc++.h>

using namespace std;

const int CHAR=256;

int leftMost(string &str)

{

    bool visited[CHAR];

    fill(visited,visited+CHAR,false);

    int res=-1;

    for(int i=str.length()-2;i>=0;i--)

    {

        if(visited[str[i]])

            res=i;

        else

            visited[str[i]]=true;

    }

    return res;

}

int main()

{

    string str = "geeksforgeeks";

    cout<<"Index of leftmost repeating character:"<<endl;

    cout<<leftMost(str)<<endl;

    return 0;

}

### OUTPUT :

### Index of leftmost repeating character:

### 0

### Leftmost non repeating element :

### Given a string, the task is to find the leftmost character that does not repeat.

### Naïve :

#include<iostream>

using namespace std;

int nonRep(string &str)

{

    for(int i=0;i<str.length();i++)

    {

        bool flag=false;

        for(int j=0;j<str.length();j++)

        {

           if(i!=j && str[i]==str[j])

           {

                flag=true;

                break;

           }

        }

        if(flag==false)

            return i;

    }

    return -1;

}

int main()

{

    string str = "geeksforgeeks";

    cout<<"Index of leftmost non-repeating element:"<<endl;

    cout<<nonRep(str)<<endl;

    return 0;

}

### OUTPUT :

### 5

### better approach for leftmost non repeating element :

#include<iostream>

using namespace std;

const int CHAR=256;

int nonRep(string &str)

{

    int count[CHAR]={0};

    for(int i=0;i<str.length();i++)

        count[str[i]]++;

    for(int i=0;i<str.length();i++)

        if(count[str[i]]==1)

            return i;

    return -1;

}

int main()

{

    string str = "geeksforgeeks";

    cout<<"Index of leftmost non-repeating element:"<<endl;

    cout<<nonRep(str)<<endl;

    return 0;

}

### OUTPUT :

### 5

### efficient approach for leftmost non repeating element :

//using one traversal

//time complexity is theta(n)

#include<bits/stdc++.h>

using namespace std;

const int CHAR=256;

int nonRep(string &str)

{

    int fI[CHAR];

    fill(fI,fI+CHAR,-1);

    for(int i=0;i<str.length();i++)

    {

        if(fI[str[i]]==-1)

            fI[str[i]]=i;

        else

            fI[str[i]]=-2;

    }

    int res=INT32\_MAX;

    for(int i=0;i<CHAR;i++)

        if(fI[i]>=0)

            res=min(res,fI[i]);

    return (res==INT32\_MAX)? -1:res;

}

int main()

{

    string str = "geeksforgeeks";

    cout<<"Index of leftmost non-repeating element:"<<endl;

    cout<<nonRep(str)<<endl;

    return 0;

}

### OUTPUT :

### 5

### Reverse Word in String :

### Efficient :

//auxilary space o(1)

#include<bits/stdc++.h>

using namespace std;

void reverse(char str[], int low, int high)

{

    while(low<=high)

    {

        swap(str[low],str[high]);

        low++;

        high--;

    }

}

void reverseWords(char str[], int n)

{

    int start=0;

    for(int end=0;end<n;end++)

    {

        if(str[end]==' ')

        {

            reverse(str,start,end-1);

            start=end+1;

        }

    }

    reverse(str,start,n-1);

    reverse(str,0,n-1);

}

int main()

{

    string s="Welcome to Gfg";

    int n=s.length();

    char str[n];

    strcpy(str,s.c\_str());

    cout<<"After reversing word in the string:"<<endl;

    reverseWords(str,n);

    // cout<<str;

    for(int i=0;i<n;i++)

        cout<<str[i];

    return 0;

}

### OUTPUT :

### After reversing word in the string:

### Gfg to Welcome

### Overview of Pattern searching :

### Niave pattern searching :

#include<bits/stdc++.h>

using namespace std;

void patSearching(string &txt,string &pat)

{

    int m=pat.length();

    int n=txt.length();

    for(int i=0;i<=(n-m);i++)

    {

        int j;

        for(j=0;j<m;j++)

            if(pat[j]!=txt[i+j])

                break;

        if(j==m)

            cout<<i<<" ";

    }

}

int main()

{

    string txt = "ABCABCD";string pat="ABCD";

    cout<<"All index numbers where pattern found:"<<" ";

    patSearching(txt,pat);

    return 0;

}

### OUTPUT : All index numbers where pattern found: 3

### Improved Naïve pattern searching for distinct element :

// Given a pattern with distinct characters and a text,

// we need to print all occurrences of the pattern in the text.

// This video talks about improved Naive pattern searching with Theta(n) time complexity

#include<bits/stdc++.h>

using namespace std;

void patSearching(string &txt, string &pat)

{

    int m=pat.length();

    int n=txt.length();

    for(int i=0;i<=(n-m);)

    {

        int j;

        for(j=0;j<m;j++)

            if(pat[j]!=txt[i+j])

                break;

        if(j==m)

            cout<<i<<" ";

        if(j==0)

        {

            i++;

        }

        else

        {

            i=(i+j);

        }

    }

}

int main()

{

    string txt = "ABCABCD";string pat="ABCD";

    cout<<"All index numbers where pattern found:"<<" ";

    patSearching(txt,pat);

    return 0;

}

### OUTPUT :

### All index numbers where pattern found: 3

### Rabin Karp Algorithm :

### In computer science, the Rabin–Karp algorithm or Karp–Rabin algorithm is a string-searching algorithm created by Richard M. Karp and Michael O. Rabin (1987) that uses hashing to find an exact match of a pattern string in a text. ... A practical application of the algorithm is detecting plagiarism

#include <bits/stdc++.h>

using namespace std;

#define d 256

const int q=101;  //any prime number

void RBSearch(string pat,string txt,int M, int N){

    //Compute (d^(M-1))%q

    int h=1;

    for(int i=1;i<=M-1;i++)

        h=(h\*d)%q;

    //Compute p and to

    int p=0,t=0;

    for(int i=0;i<M;i++){

        p=(p\*d+pat[i])%q;

        t=(t\*d+txt[i])%q;

    }

    for(int i=0;i<=(N-M);i++){

       //Check for hit

       if(p==t){

           bool flag=true;

           for(int j=0;j<M;j++)

                if(txt[i+j]!=pat[j]){flag=false;break;}

            if(flag==true)cout<<i<<" ";

       }

       //Compute ti+1 using ti

       if(i<N-M){

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

        if(t<0)t=t+q;

       }

    }

}

int main()

{

    string txt = "GEEKS FOR GEEKS";string pat="GEEK";

    cout<<"All index numbers where pattern found:"<<" ";

    RBSearch(pat,txt,4,15);

    return 0;

}

### OUTPUT :

### All index numbers where pattern found: 0 10

### KMP Algorithm part1 (Constructing LPS array) :

### Naïve solution :

// KMP Agorithm (Part 1 : Constructing LPS Array)

// Two methods of LPS (Longest Proper Prefx Suffix) Array are

// discussed. One method has time complexity O(n^3) and other method is O(n).

#include <bits/stdc++.h>

using namespace std;

int longPropPreSuff(string str, int n)

{

    for(int len=n-1;len>0;len--)

    {

        bool flag=true;

        for(int i=0;i<len;i++)

            if(str[i]!=str[n-len+i])

                flag=false;

        if(flag==true)

            return len;

    }

    return 0;

}

void fillLPS(string str, int \*lps)

{

    for(int i=0;i<str.length();i++)

        lps[i]=longPropPreSuff(str,i+1);

}

int main()

{

    string txt="abacabad";

    int lps[txt.length()];

    fillLPS(txt,lps);

    for(int i=0;i<txt.length();i++)

        cout<<lps[i]<<" ";

    return 0;

}

### OUTPUT : 0 0 1 0 1 2 3 0

### KMP Algorithm part1 (Constructing LPS array) :

### Efficient solution :

// KMP Agorithm (Part 1 : Constructing LPS Array)

// Two methods of LPS (Longest Proper Prefx Suffix) Array are

// discussed. One method has time complexity O(n^3) and other method is O(n).

#include<bits/stdc++.h>

using namespace std;

void fillLPS(string str,int \*lps)

{

    int n=str.length(), len=0;

    lps[0]=0;

    int i=1;

    while (i<n)

    {

        if(str[i]==str[len])

        {

            len++;

            lps[i]=len;

            i++;

        }

        else

        {

            if(len==0)

            {

                lps[i]=0;

                i++;

            }

            else

            {

                len=lps[len-1];

            }

        }

    }

}

int main()

{

    string txt="abacabad";

    int lps[txt.length()];

    fillLPS(txt,lps);

    for(int i=0;i<txt.length();i++)

        cout<<lps[i]<<" ";

    return 0;

}

### OUTPUT :

### 0 0 1 0 1 2 3 0

### KMP Algorithm part2 (Complete algorithm) :

#include <bits/stdc++.h>

using namespace std;

void fillLPS(string str, int \*lps){

    int n=str.length(),len=0;

    lps[0]=0;

    int i=1;

    while(i<n){

        if(str[i]==str[len])

        {len++;lps[i]=len;i++;}

        else

        {if(len==0){lps[i]=0;i++;}

            else{len=lps[len-1];}

        }

    }

}

void KMP(string pat,string txt){

    int N=txt.length();

    int M=pat.length();

    int lps[M];

    fillLPS(pat,lps);

    int i=0,j=0;

    while(i<N){

        if(pat[j]==txt[i]){i++;j++;}

        if (j == M) {

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

            j = lps[j - 1];

        }

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

            if (j == 0)

                i++;

            else

                j = lps[j - 1];

        }

    }

}

int main()

{

    string txt = "ababcababaad",pat="ababa";

    KMP(pat,txt);

    return 0;

}

### OUTPUT : Found pattern at index 5

### Check if string are Rotations

// It is a constant static member value with the highest

// possible value for an element of type size\_t.

// It actually means until the end of the string.

// It is used as the value for a length parameter in the string’s

// member functions.

// As a return value, it is usually used to indicate no matches.

#include<bits/stdc++.h>

using namespace std;

bool areRotations(string s1,string s2)

{

    if(s1.length()!=s2.length()) return false;

    return ((s1+s2).find(s2)!=string::npos);

}

int main()

{

    string s1="ABCD";

    string s2="CDAB";

    if(areRotations(s1,s2)){

        cout<<"String are rotations of each other"<<endl;

    }

    else{

        cout<<"String are not rotation of each other"<<endl;

    }

    return 0;

}

### OUTPUT :

### String are rotations of each other

### Anagram Search

### Naïve :

#include <bits/stdc++.h>

using namespace std;

const int CHAR=256;

bool areAnagram(string &pat, string &txt,int i)

{

    int count[CHAR]={0};

    for(int j=0;j<pat.length();j++){

        count[pat[j]]++;

        count[txt[i+j]]--;

    }

    for(int j=0;j<CHAR;j++){

        if(count[j]!=0)return false;

    }

    return true;

}

bool isPresent(string &txt, string &pat){

    int n=txt.length();

    int m=pat.length();

    for(int i=0;i<=n-m;i++){

        if(areAnagram(pat,txt,i))return true;

    }

    return false;

}

int main()

{

    string txt = "geeksforgeeks";

    string pat = "frog";

    if (isPresent(txt,pat))

        cout << "Anagram search found";

    else

        cout << "Anagram search not found";

    return 0;

}

### OUTPUT :

### Anagram search found

### Anagram Search

### Efficient:

#include<bits/stdc++.h>

using namespace std;

const int CHAR=256;

bool areSame(int CT[],int CP[])

{

    for(int i=0;i<CHAR;i++)

    {

        if(CT[i]!=CP[i])

            return false;

    }

    return true;

}

bool isPresent(string &txt, string &pat)

{

    int CT[CHAR]={0},CP[CHAR]={0};

    for(int i=0;i<pat.length();i++)

    {

        CT[txt[i]]++;

        CP[pat[i]]++;

    }

    for(int i=pat.length();i<txt.length();i++)

    {

        if(areSame(CT,CP))return true;

        CT[txt[i]]++;

        CT[txt[i-pat.length()]]--;

    }

    return false;

}

int main()

{

    string txt="geeksforgeeks";

    string pat="frog";

    if(isPresent(txt,pat))

        cout<<"Anagram Search Found";

    else

        cout<<"Anagram Search not found";

    return 0;

}

### OUTPUT :

### Anagram Search Found

### Lexicographic rank of string

#include <bits/stdc++.h>

using namespace std;

const int CHAR=256;

int fact(int n)

{

    return (n <= 1) ? 1 : n \* fact(n - 1);

}

int lexRank(string &str)

{

    int res = 1;

    int n=str.length();

    int mul= fact(n);

    int count[CHAR]={0};

    for(int i=0;i<n;i++)

        count[str[i]]++;

    for(int i=1;i<CHAR;i++)

        count[i]+=count[i-1];

    for(int i=0;i<n-1;i++){

        mul=mul/(n-i);

        res=res+count[str[i]-1]\*mul;

        for(int j=str[i];j<CHAR;j++)

            count[j]--;

    }

    return res;

}

int main()

{

    string str = "STRING";

    cout << lexRank(str);

    return 0;

}

### OUTPUT :

### 598

### Longest substring with distinct character

### Naïve :

//time complexity O(n^3)

#include<iostream>

#include<vector>

using namespace std;

bool areDistinct(string str, int i, int j)

{

    vector<bool>visited(256);

    for(int k=i;k<=j;k++)

    {

        if(visited[str[k]]==true)

            return false;

        visited[str[k]]=true;

    }

    return true;

}

int longestDistinct(string str)

{

    int n=str.length();

    int res=0;

    for(int i=0;i<n;i++)

        for(int j=i;j<n;j++)

            if(areDistinct(str,i,j))

                res=max(res,j-i+1);

    return res;

}

int main()

{

    string str="geeksforgeeks";

    int len=longestDistinct(str);

    cout<<"The length of the longest distinct character substring is "<<len;

    return 0;

}

### OUTPUT :

### The length of the longest distinct character substring is 7

### Longest substring with distinct character

### Better:

//time complexity O(n^2)

#include<iostream>

#include<vector>

using namespace std;

int longestDistinct(string str)

{

    int n=str.size();

    int res=0;

    for(int i=0;i<n;i++)

    {

        vector<bool>visited(256);

        for(int j=i;j<n;j++)

        {

            if(visited[str[j]]==true)

                break;

            else

            {

                res=max(res,j-i+1);

                visited[str[j]]=true;

            }

        }

    }

    return res;

}

int main()

{

    string str="geeksforgeeks";

    int len=longestDistinct(str);

    cout<<"The length of the longest distinct character substring is "<<len;

    return 0;

}

### OUTPUT :

### The length of the longest distinct character substring is 7

### Longest substring with distinct character

### Efficient :

#include<iostream>

#include<vector>

using namespace std;

int longestDistinct(string str)

{

    int n=str.length();

    int res=0;

    vector<int>prev(256,-1);

    int i=0;

    for(int j=0;j<n;j++)

    {

        i=max(i,prev[str[j]]+1);

        int maxEnd=j-i+1;

        res=max(res,maxEnd);

        prev[str[j]]=j;

    }

    return res;

}

int main()

{

    string str="geeksforgeeks";

    int len=longestDistinct(str);

    cout<<"The length of the longest distinct characters substring is "<<len;

    return 0;

}

### OUTPUT :

### The length of the longest distinct characters substring is 7