**Practical-1**

**Aim: Implement and analyze algorithms given below.**

* 1. **Fibonacci Series Iterative**

**Program:**

#include <iostream>

using namespace std;

void fib(int num) {

int x = 0, y = 1, z = 0;

for (int i = 0; i < num; i++) {

cout << x << " ";

z = x + y;

x = y;

y = z;

}

}

int main() {

int num;

cout << "Enter the number : ";

cin >> num;

cout << "\nThe fibonacci series : " ;

fib(num);

return 0;

}

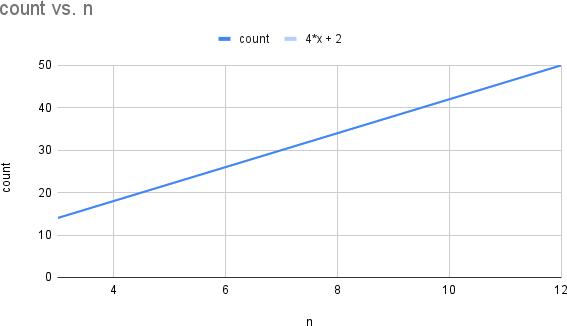
# Output:

****

**Analysis Table:**

|  |  |
| --- | --- |
| N | count |
| 3 | 14 |
| 4 | 18 |
| 5 | 22 |
| 6 | 26 |
| 7 | 30 |
| 8 | 34 |
| 9 | 38 |
| 10 | 42 |
| 11 | 46 |
| 12 | 50 |

**Graph:**



**Complexity: O(n)**

Conclusion: Y = 4x + 6 is the equation as represented above in the graph. As the highest degree is 1, therefore order is O(n).

# Recursive

**Program:**

#include <iostream>

using namespace std;

int fib(int x) {

if((x==1)||(x==0)) {

return(x);

}else {

return(fib(x-1)+fib(x-2));

}

}

int main() {

int x , i=0;

cout << "Enter the number of terms of series : ";

cin >> x;

cout << "\nFibonnaci Series : ";

while(i < x) {

cout << " " << fib(i);

i++;

}

return 0;

}

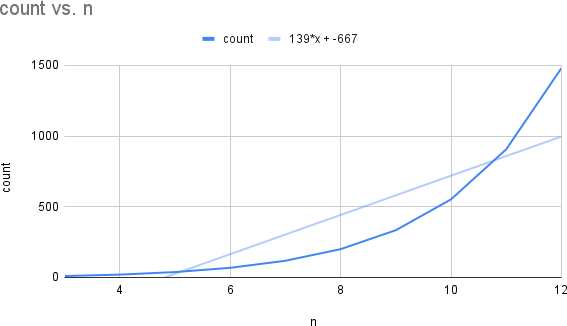
# Output:



**Analysis Table:**

|  |  |
| --- | --- |
| n | count |
| 3 | 10 |
| 4 | 20 |
| 5 | 38 |
| 6 | 68 |
| 7 | 118 |
| 8 | 200 |
| 9 | 334 |
| 10 | 552 |
| 11 | 906 |
| 12 | 1480 |

**Graph:**



**Complexity: O(n^2)**

**Conclusion:** As the Fibonacci iterative has order O(n) which is linear and Fibonacci recursive has exponential order therefore the Fibonacci iterative is optimal rather than Fibonacci recursive

# 1.2 Factorial of a given number

# Iterative

**Program:**

# #include<iostream>

# using namespace std;

# int fact(int n)

# { int result=1;

# 

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

# {

# result=result\*i;

# }

# return result;

# }

# int main()

# {

# int num;

# cout<<"Please Enter a Number to find its Factorial : ";

# cin>>num;

# if(num<0)

# {

# cout<<"\ninpossible to find the factorial, Please Try again"; }

# else

# {

# cout<<"\nFactorial is :"<<fact(num);

# } return 0;

# }

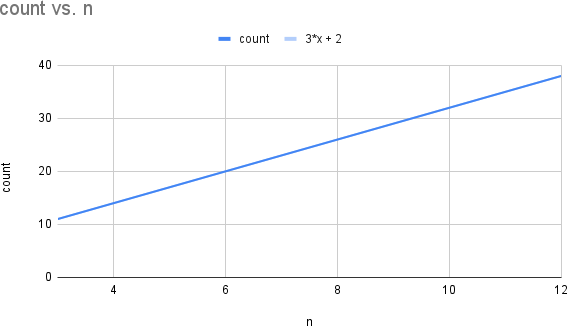
# Output:

****

**Analysis Table:**

|  |  |
| --- | --- |
| n | count |
| 3 | 11 |
| 4 | 14 |
| 5 | 17 |
| 6 | 20 |
| 7 | 23 |
| 8 | 26 |
| 9 | 29 |
| 10 | 32 |
| 11 | 35 |
| 12 | 38 |

**Graph:**



**Complexity: O(n)**

**Conclusion:** Y = 3x + 2 is the equation as represented in the graph above. And as the highest degree is 1, therefore order is O(n).

# Recursive

**Program:**

# #include<iostream>

# using namespace std;

# int factorial(int n);

# int main() {

# int n;

# cout << "Enter a positive integer: ";

# cin >> n;

# cout << "Factorial of " << n << " = " << factorial(n);

# return 0;

# }

# int factorial(int n) {

# if(n > 1)

# return n \* factorial(n - 1);

# else

# return 1;

# }

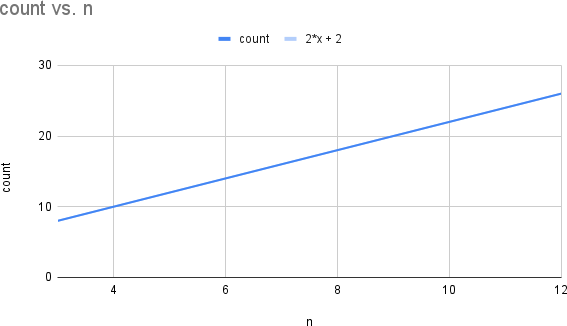
# Output:



**Analysis Table:**

|  |  |
| --- | --- |
| n | count |
| 3 | 8 |
| 4 | 10 |
| 5 | 12 |
| 6 | 14 |
| 7 | 16 |
| 8 | 18 |
| 9 | 20 |
| 10 | 22 |
| 11 | 24 |
| 12 | 26 |

**Graph:**



**Complexity:O{n)**

**Conclusion: :** As for both the methods i.e., Factorial iterative and factorial recursive the Order is same i.e., O(n)

# 1.3 Matrix Addition and Matrix Multiplication (Iterative)

# Matrix Addition

**Program:**

#include <iostream>

using namespace std;

int main()

{

int r, c, a[100][100], b[100][100], sum[100][100], i, j;

cout << "Enter number of rows (between 1 and 100): ";

cin >> r;

cout << "Enter number of columns (between 1 and 100): ";

cin >> c;

cout << endl << "Enter elements of 1st matrix: " << endl;

// Storing elements of first matrix entered by user.

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

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

{

cout << "Enter element a" << i + 1 << j + 1 << " : ";

cin >> a[i][j];

}

// Storing elements of second matrix entered by user.

cout << endl << "Enter elements of 2nd matrix: " << endl;

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

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

{

cout << "Enter element b" << i + 1 << j + 1 << " : ";

cin >> b[i][j];

}

// Adding Two matrices

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

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

sum[i][j] = a[i][j] + b[i][j];

// Displaying the resultant sum matrix.

cout << endl << "Sum of two matrix is: " << endl;

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

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

{

cout << sum[i][j] << " ";

if(j == c - 1)

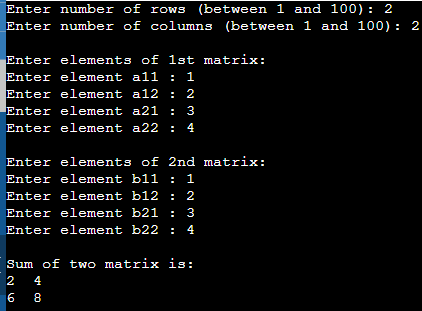
cout << endl;

}

return 0;

}

# Output:



**Multiplication:**

**Program:**

#include<iostream>

using namespace std;

int main() {

int product[10][10], r1=3, c1=3, r2=3, c2=3, i, j, k;

int a[3][3] = { {2, 4, 1} , {2, 3, 9} , {3, 1, 8} };

int b[3][3] = { {1, 2, 3} , {3, 6, 1} , {2, 4, 7} };

if (c1 != r2) {

cout<<"Column of first matrix should be equal to row of second matrix";

} else {

cout<<"The first matrix is:"<<endl;

for(i=0; i<r1; ++i) {

for(j=0; j<c1; ++j)

cout<<a[i][j]<<" ";

cout<<endl;

}

cout<<endl;

cout<<"The second matrix is:"<<endl;

for(i=0; i<r2; ++i) {

for(j=0; j<c2; ++j)

cout<<b[i][j]<<" ";

cout<<endl;

}

cout<<endl;

for(i=0; i<r1; ++i)

for(j=0; j<c2; ++j) {

product[i][j] = 0;

}

for(i=0; i<r1; ++i)

for(j=0; j<c2; ++j)

for(k=0; k<c1; ++k) {

product[i][j]+=a[i][k]\*b[k][j];

}

cout<<"Product of the two matrices is:"<<endl;

for(i=0; i<r1; ++i) {

for(j=0; j<c2; ++j)

cout<<product[i][j]<<" ";

cout<<endl;

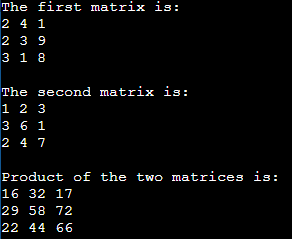
}

}

return 0;

}

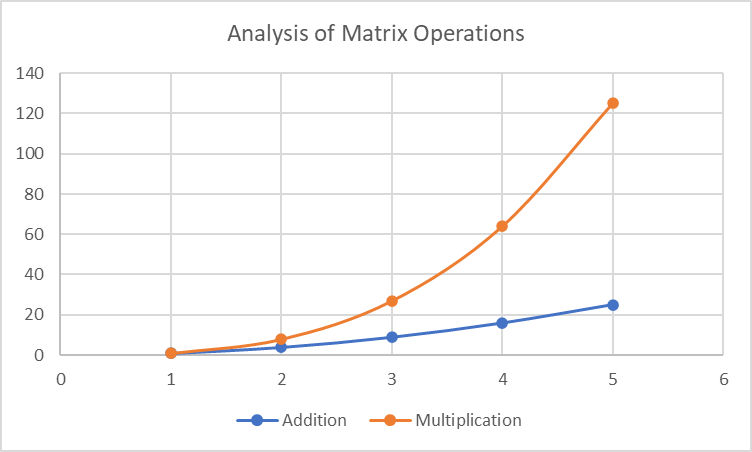
**Output:**

****

**INPUT vs COUNT TABLE:**

|  |  |  |
| --- | --- | --- |
| Dimensio n (Input) | Addit ion | Multiplic ation |
| 1 | 1 | 1 |
| 2 | 4 | 8 |
| 3 | 9 | 27 |
| 4 | 16 | 64 |
| 5 | 25 | 125 |
| 1 | 1 | 1 |
| 2 | 4 | 8 |

**Graph:**



**1.4**

Find a subset of a given set S = {s1,s2,. ,sn} of n positive integers whose sum is equal to a given

positive integer d. For example, if S= {1, 2, 5, 6, 8} and d = 9 there are two solutions {1,2,6} and

{1,8}. A suitable message is to be displayed if the given problem instance doesn't have a solution. Also, analyse the solution algorithm.

# Program:

#include<iostream> using namespace std; int counter=0;

bool isSubsetSum(int set[], int n, int sum)

{

counter++;

if (sum == 0)

return true;

if (n == 0 && sum != 0)

return false;

if (set[n-1] > sum)

return isSubsetSum(set, n-1, sum);

return isSubsetSum(set, n-1, sum) || isSubsetSum(set, n-1, sum-set[n-1]);

}

int main()

{

int set[] = {1,2,7,8,3,5,9,99};

int sum = 14; cout<<"\nSum is : "<<sum;

int n = sizeof(set)/sizeof(set[0]);

if (isSubsetSum(set, n, sum) == true) cout<<"\nFound a subset with given sum\n";

else

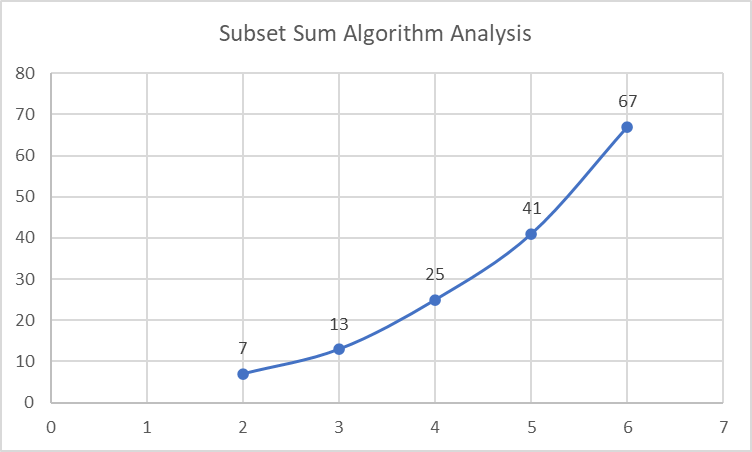
cout<<"No subset with given sum"; cout<<"\nCounter : "<<counter;\ return 0;

}

**INPUT vs COUNT TABLE:**

|  |  |
| --- | --- |
| **Input Size** | **Countin g** |
| **2** | **7** |
| **3** | **13** |
| **4** | **25** |
| **5** | **41** |
| **6** | **67** |

**GRAPH:**



**Practical 2**

**Aim: Implement and analyze algorithms given below. (Compare them)**

**2.1 Bubble sort**

**Program:**

#include <iostream> using namespace std; int counter = 0;

void bubbleSort(int arr[], int size){ int temp;

for(int i=0; i<size; i++){ for(int j=0; j<size-i-1; j++){

if(arr[j] > arr[j+1]){ temp = arr[j]; arr[j] = arr[j+1]; arr[j+1] = temp; counter++;

}

}

}

}

int main(){ int n;

cout<<"Please enter the size of array : "; cin>>n;

int arr[n];

cout<<endl<<"Please enter the array elements : "; for(int i=0; i<n; i++)

{

cin>>arr[i];

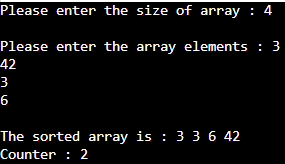
}

bubbleSort(arr, n);

cout<<endl<<"The sorted array is : "; for(int i=0;i<n;i++)

cout<<arr[i]<<" "; cout<<endl<<"Counter : "<<counter<<endl;

# Output:



**Analysis Table:**

|  |  |
| --- | --- |
| n | count |
| 4 | 2 |
| 6 | 5 |
| 7 | 17 |
| 9 | 11 |
| 11 | 20 |

**Graph:**

**Average case**

**Complexity: Best case O( 1)**

**Worst case (n^2) Average case(n^2)**

**Aim: Implement and analyze algorithms given below. (Compare them)**

* 1. **SELECTION SORT**

**PROGRAM CODE**

#include <iostream> using namespace std; int counter=0;

void selectionSort(int arr[], int size){ int temp;

int min\_index = 0; for(int i=0;i<size;i++){

min\_index=i;

for(int j=i; j<size; j++){ if(arr[min\_index] > arr[j]){

min\_index = j;

}}

temp = arr[i];

arr[i] = arr[min\_index]; arr[min\_index] = temp; counter++;

}

}

int main(){ int n;

cout<<"\nEnter size of array : "; cin>>n;

int arr[n];

cout<<"Enter array elements : "; for(int i=0;i<n;i++)

cin>>arr[i]; selectionSort(arr, n);

cout<<endl<<"Sorted array is : "; for(int i=0; i<n; i++)

cout<<arr[i]<<" ";

cout<<endl<<"Counter : "<<counter<<endl;

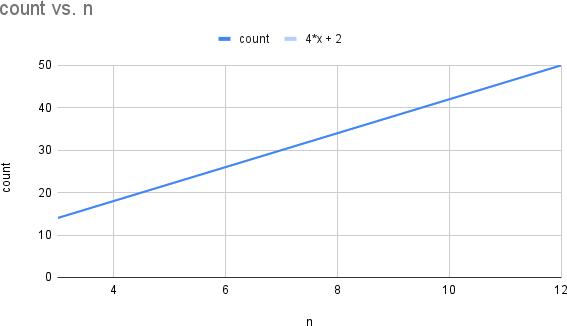
}

# Output:

**Analysis Table:**

|  |  |
| --- | --- |
| n | count |
| 3 | 14 |
| 4 | 18 |
| 5 | 22 |
| 6 | 26 |
| 7 | 30 |
| 8 | 34 |
| 9 | 38 |
| 10 | 42 |
| 11 | 46 |
| 12 | 50 |

**Graph:**



**Complexity: O(n)**

Conclusion: Y = 4x + 6 is the equation as represented above in the graph. As the highest degree is 1, therefore order is O(n).

**2.3 INSERTION SORT**

**PROGRAM CODE**

#include <iostream> using namespace std; int counter=0;

void insertionSort(int arr[], int size){ int temp;

for(int i=0;i<size-1;i++){ for(int j=i+1; j>0; j--){

if(arr[j] < arr[j-1]){ temp = arr[j]; arr[j] = arr[j-1]; arr[j-1] = temp; counter ++ ;

}

}

}

}

int main(){ int n;

cout<<"\nEnter the size of the array : "; cin>>n;

int arr[n];

cout<<endl<<"Please enter the array elements : "; for(int i=0; i<n; i++)

cin>>arr[i]; insertionSort(arr, n);

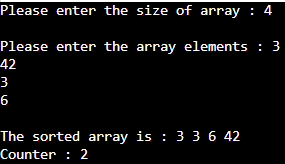
cout<<endl<<"The sorted array is : "<<endl; for(int i=0;i<n;i++)

cout<<arr[i]<<" ";

cout<<endl<<"Counter : "<<counter<<endl; return 0;

}

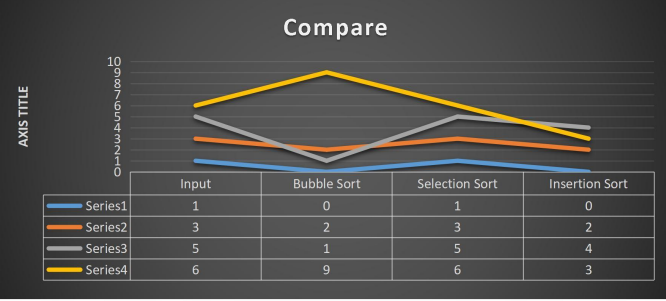
**Output:**



**Analysis Table:**

|  |  |
| --- | --- |
| n | count |
| 4 | 14 |
| 6 | 18 |
| 7 | 22 |
| 9 | 26 |
| 11 | 20 |

**Graph:**



**Complexity: O(n)**

In this practical I learn the comparison between selection sort, bubble sort and insertion sort. We also studied their worst, best and average case scenarios and accordingly we plotted the graph.

**PRACTICAL -3**

**3.1 AIM: Implement and perform analysis of worst case of Merge Sort and Quick sort. Compare both algorithms.**

**MERGE SORT**

**PROGRAM CODE**:

#include <iostream>

using namespace std;

int counter = 0;

void merge(int arr[], int p, int q, int r)

{

int n1 = q - p + 1;

int n2 = r - q;

int L[n1], M[n2];

for (int i = 0; i < n1; i++)

L[i] = arr[p + i];

for (int j = 0; j < n2; j++)

M[j] = arr[q + 1 + j];

int i, j, k;

i = 0;

j = 0;

k = p;

while (i < n1 && j < n2)

{

if (L[i] <= M[j])

{

arr[k] = L[i];

i++;

}

else

{

arr[k] = M[j];

j++;

}

k++;

}

while (i < n1)

{

arr[k] = L[i];

i++;

k++;

}

while (j < n2)

{

arr[k] = M[j];

j++;

k++;

}

}

void mergeSort(int arr[], int l, int r)

{

counter++;

if (l < r)

{

int m = l + (r - l) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

int main()

{

int size;

cout << "Enter the size of the array: ";

cin >> size;

int arr[size];

cout << "Enter the values in the array: " << endl;

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

{

cin >> arr[i];

}

mergeSort(arr, 0, size - 1);

cout << "The sorted array will be: " << endl;

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

cout << arr[i] << " ";

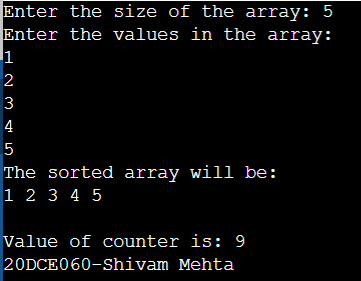
cout << endl;

cout << "\nValue of counter is: " << counter;

cout << "\n20DCE060-Shivam Mehta";

return 0;}

**Output:**



**QUICK SORT**

**PROGRAM CODE**:

#include <iostream>

using namespace std;

int counter = 0;

int partition(int arr[], int low, int high)

{

int pivot = arr[high];

int i = low - 1;

for (int j = low; j <= high - 1; j++)

{

if (arr[j] <= pivot)

{

i++;

swap(arr[i], arr[j]);

}

}

swap(arr[i + 1], arr[high]);

return i + 1;

}

void quickSort(int arr[], int low, int high)

{

counter++;

if (low < high)

{

int pivotIndex = partition(arr, low, high);

quickSort(arr, low, pivotIndex - 1);

quickSort(arr, pivotIndex + 1, high);

}

}

int main()

{

int size;

cout << "Enter the size of the array: ";

cin >> size;

int arr[size];

cout << "Enter the values in the array: " << endl;

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

{

cin >> arr[i];

}

quickSort(arr, 0, size - 1);

cout << "The sorted array will be: " << endl;

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

{

cout << arr[i] << " ";

}

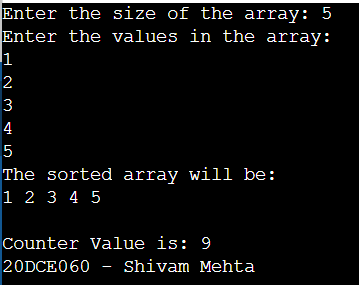
cout << "\n\nCounter Value is: " << counter;

cout << "\n20DCE060 – Shivam Mehta";

return 0;

}

**Output:**



**ANALYSIS TABLE**:

|  |  |  |
| --- | --- | --- |
| No. of Element | Iteration(merge sort) | Iteration(quick sort) |
| 3 | 5 | 10 |
| 4 | 7 | 7 |
| 5 | 9 | 9 |
| 10 | 19 | 13 |

**GRAPH**:

**3.2 AIM:** **Implement the program to perform Linear Search and Binary Search. Also compare Time complexity of both.**

**BINARY SEARCH**

**PROGRAM CODE:**

#include <stdio.h>

int count=0;

int main(){

int num;

int i, keynum, found = 0;

printf("Enter the number of elements: ");

scanf("%d", &num);

int array[num];

printf("Enter the elements one by one: \n");

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

{

scanf("%d", &array[i]);

}

printf("Enter the element to be searched: ");

scanf("%d", &keynum);

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

{count++;

if (keynum == array[i] )

{count++;

found = 1;

break;

}}

if (found == 1)

printf("Element is present in the array at position %d",i+1);

else

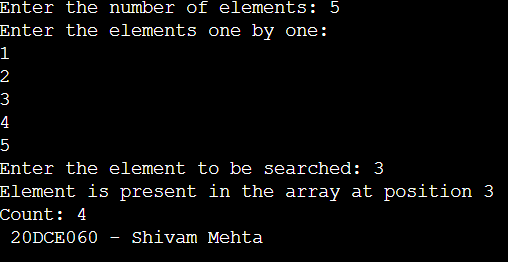
printf("Element is not present in the array\n");

printf("\nCount: %d",count);

printf("\n 20DCE060 – Shivam Mehta");

}

**Output:**



**LINEAR SEARCH**

**PROGRAME CODE:**

#include <stdio.h>

int count=0;

int main()

{

int i, low, high, mid, n, key, array[100];

printf("Enter number of elements: ");

scanf("%d",&n);

printf("Enter %d integers: ", n);

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

scanf("%d",&array[i]);

printf("Enter value to find: ");

scanf("%d", &key);

low = 0;

high = n - 1;

mid = (low+high)/2;

while (low <= high) {

count++;

if(array[mid] < key){

low = mid + 1;

count++;

}

else if (array[mid] == key) {

count++;

printf("%d found at location %d: ", key, mid+1);

break;

}

Else{

count++;

high = mid - 1;

mid = (low + high)/2;

}

if(low > high)

printf("Not found! %d isn't present in the list: ", key);

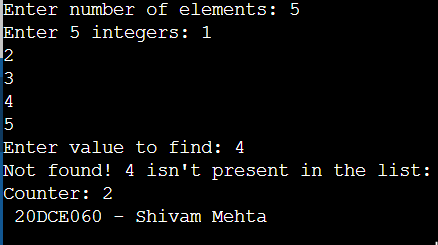
printf("\nCounter: %d",count);

printf("\n 20DCE060 – Shivam Mehta");

return 0;

}

**Output:**



**ANALYSIS TABLE:**

**Linear Search:**

**Best Case:**

|  |  |
| --- | --- |
| **Number of Elements** | **Counter Value** |
| 3 | 2 |
| 4 | 2 |
| 5 | 2 |
| 6 | 2 |
| 7 | 2 |

**Average Case:**

|  |  |
| --- | --- |
| **Number of Elements** | **Counter Value** |
| 3 | 3 |
| 4 | 4 |
| 5 | 4 |
| 6 | 5 |
| 7 | 5 |

**Worst Case:**

|  |  |
| --- | --- |
| **Number of Elements** | **Counter Value** |
| 3 | 4 |
| 4 | 5 |
| 5 | 6 |
| 6 | 7 |
| 7 | 8 |

**Binary Search:**

**Best Case:**

|  |  |
| --- | --- |
| **Number of Elements** | **Counter Value** |
| 3 | 2 |
| 4 | 2 |
| 5 | 2 |
| 6 | 2 |
| 7 | 2 |

**Average Case:**

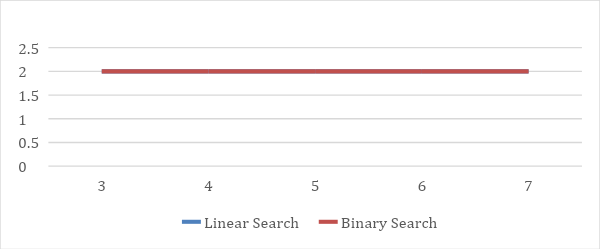
|  |  |
| --- | --- |
| **Number of Elements** | **Counter Value** |
| 3 | 2 |
| 4 | 4 |
| 5 | 4 |
| 6 | 4 |
| 7 | 6 |

**Worst Case:**

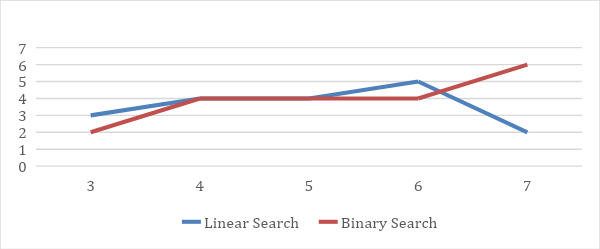
|  |  |
| --- | --- |
| **Number of Elements** | **Counter Value** |
| 3 | 2 |
| 4 | 2 |
| 5 | 4 |
| 6 | 4 |
| 7 | 4 |

**GRAPH:**

**Best Case:**

****

**Average Case:**

****

**Worst Case:**

**Chart, line chart

Description automatically generated**

**CONCLUSION:**

In this practical we have learnt the basics of divide and conquer strategy. We have also tried to optimize the power of element problem using divide and conquer strategy

# Practical – 5

**Aim:** Dynamic Programming

# Practical 5.1

## A cashier at any mall needs to give change of an amount to customers many times in a day. Cashier has multiple number of coins available with different denominations which is described by a set C. Implement the program for a cashier to find the minimum number of coins required to find a change of a particular amount A. Output should be the total number of coins required of given denominations. Check the program for following test cases:

|  |  |  |
| --- | --- | --- |
| **Test Case** | **n** | **k** |
| **1** | **₹1, ₹2, ₹3** | **₹ 5** |
| **2** | **₹18, ₹17,**  **₹5, ₹1** | **₹ 22** |
| **3** | **₹100, ₹25,**  **₹10, ₹5, ₹1** | **₹ 289** |

**Program Code**:

#include<iostream>

using namespace std;

int MakeChange(int d[], int n, int sum)

{

int i,j;

int x,y;

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

c[0][j]=INT\_MAX;

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

c[i][0]=0;

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

{

for(j=1;j<=sum;j++)

{

if(j>=d[i-1])

{

c[i][j]=min(c[i-1][j],1+c[i][j-d[i-1]]);

}

else

{

c[i][j]=c[i-1][j];

}

}

}

return c[n][sum];

}

int main()

{

int n,sum;

cout<<"Enter the amount whose change is required: ";

cin>>sum;

cout<<"Enter the number of coins available: ";

cin>>n;

int d[n];

cout<<"Enter the values of coins: ";

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

{

cin>>d[i];

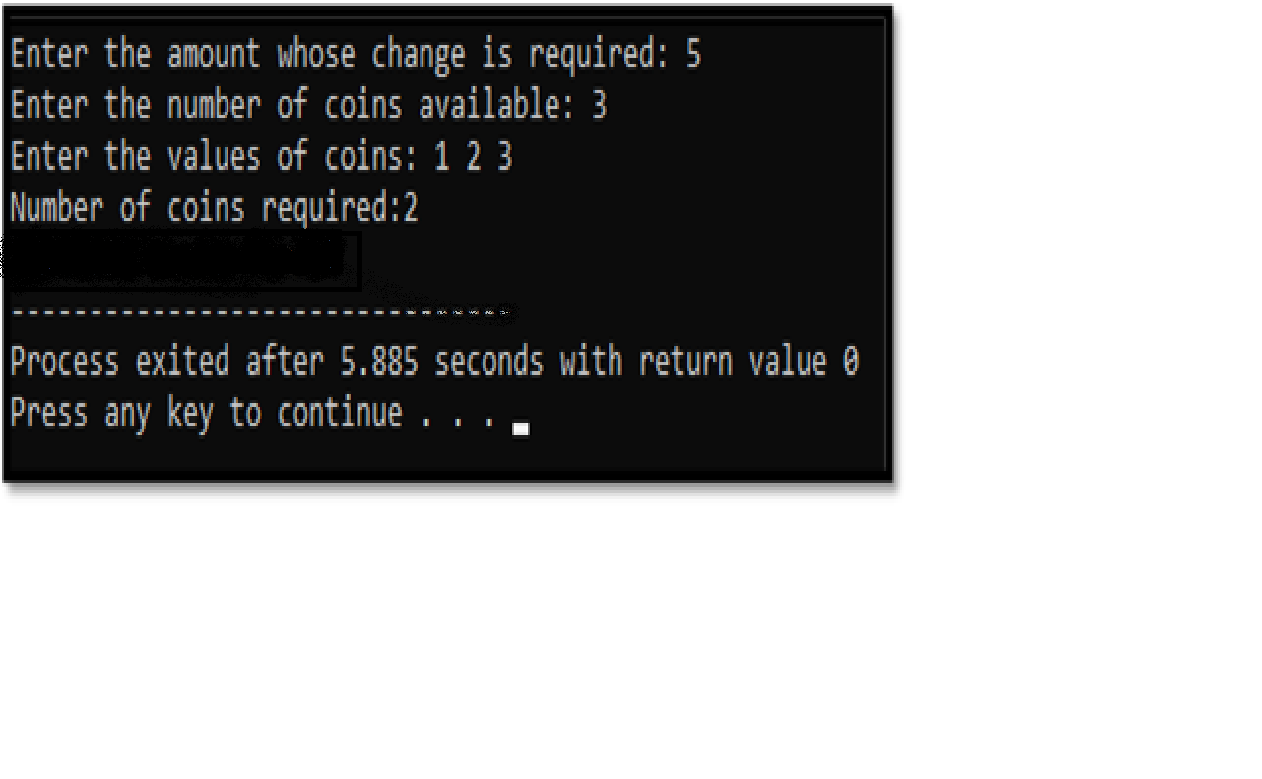
}

cout<<"Number of coins required:"<<MakeChange(d,n,sum);

return 0;

}

## **Output:**



**Time Complexity: O(NC)**

**Justification:** If the coin is greater than 1 the inner loop will run (m - coin) iterations instead m, but here we're considering the worst case and ignoring the constant factors.

# Practical 5.2

## Implement the program 4.2 using Dynamic Programing. Compare Greedy and Dynamic approach.

**Program Code**:

#include<iostream> using namespace std;

int KnapSack(int W, int w[], int v[], int n)

{

int i, j;

int V[n + 1][W + 1];

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

{

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

{

if (i == 0 || j == 0) V[i][j] = 0;

else if (w[i - 1] <= j)

V[i][j] = max(v[i - 1] + V[i - 1][j - w[i - 1]], V[i -

1][j]);

else

V[i][j] = V[i - 1][j];

}

}

return V[n][W];

}

int main()

{

int W,n;

cout<<"Enter Knapsack Capacity:"; cin>>W;

cout<<"Enter number of items:"; cin>>n;

int v[n],w[n];

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

{

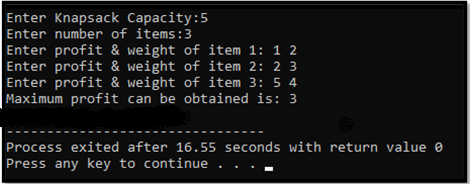
cout<<"Enter profit & weight of item "<<i<<": "; cin>>v[i]>>w[i];

}

cout << "Maximum profit can be obtained is: "<<KnapSack(W, w, v, n);

}

## **Output:**



**Time Complexity: O(NW)**

**Justification:** We can have a maximum of N\*W different subproblems that we may encounter in the recursion. Since a particular subproblem is calculated only once, the time complexity for the approach is O(N\*W).

# Practical 5.3

## Given a chain < A1, A2,…,An> of n matrices, where for i=1,2,…,n matrix Ai with dimensions. Implement the program to fully parenthesize the product A1,A2,…,An in a way that minimizes the number of scalar multiplications. Also calculate the number of scalar multiplications for all possible combinations of matrices.

|  |  |  |
| --- | --- | --- |
| **Test Case** | **n** | **Matrices with dimensions** |
| 1 | 3 | A1: 3\*5, A2: 5\*6, A3: 6\*4 |
| 2 | 6 | A1: 30\*35, A2: 35\*15, A3:  15\*5, A4: 5\*10, A5: 10\*20, A6: 20\*25 |

**Program Code**:

#include<iostream> #include<climits> using namespace std;

int matrixChain(int n, int order[])

{

int i,j,k; int temp;

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

{

dp[i][i]=0;

}

for(int size=2;size<=n;size++)

{

for(i=1;i<=(n-size+1);i++)

{

j=i+size-1; dp[i][j]=INT\_MAX;

for(k=i;k<j;k++)

{

temp=dp[i][k]+dp[k+1][j]+order[i- 1]\*order[k]\*order[j];

if(temp<dp[i][j])

{

dp[i][j]=temp;

}

}

}

}

return dp[1][n];

}

int main()

{

int i,j; int n;

cout<<"Enter the number of matrices in the chain: "; cin>>n;

int p[n+1];

cout<<"Enter the order array of the matrix chain ("<<n+1<<" elements)"<<endl;

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

{

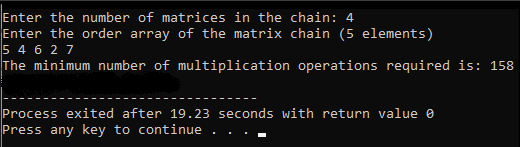
cin>>p[i];

}

cout<<"The minimum number of multiplication operations required is: "<<matrixChain(n,p);

}

## Output:



**Time Complexity: O(N3)**

**Justification:** There could be O(n^2) unique sub-problems to any MCM given problem and for every such sub-problem there could be O(n) splits possible.

# Practical 5.4

## Program to implement all pairs shortest path.

## **Program Code:**

#include <iostream>

#include <cstdlib>

#define max 10

#define infi 99

using namespace std; int p[max][max];

void allpairshort(int a[max][max], int n)

{

int k, i, j;

for (k = 0; k < n; k++)

{

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

{

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

{

if (a[i][k] + a[k][j] < a[i][j])

{

a[i][j] = a[i][k] + a[k][j]; p[i][j] = k;

}

}

}

}

}

void shortest(int i, int j)

{

int k = p[i][j]; if (k > 0)

{

shortest(i, k); cout<<" "<<k<<" "; shortest(k, j);

}

}

void findpath(int a[max][max], int i, int j, int n)

{

cout<<"Path from " << i <<" to "<< j << ":"; if (a[i][j] < infi)

{

cout<<" "<<i<<" ";

shortest(i, j); cout<<" "<<j<<" ";

}

}

int main()

{

int i, j;

int a[][10] = {{0, 10, infi, 30, 100},

{infi, 0 , 50, infi, infi},

{infi, infi , 0, infi, 10},

{infi, infi , 20, 0, 60},

{infi, infi , infi, infi, 0},

};

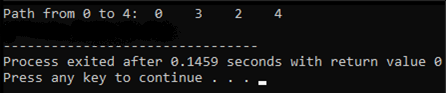
allpairshort(a, 5);

findpath(a, 0, 4, 5);

return 0;

}

**Output**:



## Time Complexity: O(N3)

**Justification:** There are three loops. Each loop has constant complexities.

# Practical – 6

**Aim:** Graph

# Practical 6.1

## Write a program to implement BFS and DFS in Graph. Compare Time Complexity of both algorithms.

**BFS:**

#include<iostream>

#include<queue>

#define NODE 6

using namespace std;

typedef struct node{

int val;

int state; //status

}node;

int graph[NODE][NODE] = {

{0, 1, 1, 1, 0, 0},

{1, 0, 0, 1, 1, 0},

{1, 0, 0, 1, 0, 1},

{1, 1, 1, 0, 1, 1},

{0, 1, 0, 1, 0, 1},

{0, 0, 1, 1, 1, 0}

};

void bfs(node \*vert, node s){ node u;

int i, j; queue<node> que;

for(i = 0; i<NODE; i++){ vert[i].state = 0;

}

vert[s.val].state = 1; que.push(s); while(!que.empty()){

u = que.front(); que.pop();

cout << char(u.val+'A') << " "; for(i = 0; i<NODE; i++){

if(graph[i][u.val]){ if(vert[i].state == 0){

vert[i].state = 1; que.push(vert[i]);

}

}

}

u.state = 2;

}

}

int main(){

node vertices[NODE]; node start;

char s;

for(int i = 0; i<NODE; i++){ vertices[i].val = i;

}

s = 'B';

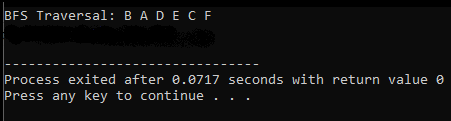
start.val = s-'A';

cout << "BFS Traversal: "; bfs(vertices, start);

cout << endl;

}

## Output:



**DFS:**

#include<iostream>

#include<stack>

using namespace std;

#define NODE 6

typedef struct node{

int val; int state;

}node;

int graph[NODE][NODE] = {

{0, 1, 1, 1, 0, 0},

{1, 0, 0, 1, 1, 0},

{1, 0, 0, 1, 0, 1},

{1, 1, 1, 0, 1, 1},

{0, 1, 0, 1, 0, 1},

{0, 0, 1, 1, 1, 0}

};

void dfs(node \*vertex, node start){

node u;

stack<node> myStack;

for(int i = 0; i<NODE; i++){

vertex[i].state = 0;

}

myStack.push(start);

while(!myStack.empty()){

u = myStack.top(); myStack.pop();

cout << char(u.val+'A') << " "; if(u.state != 1){

u.state = 1;

vertex[u.val].state = 1; for(int i = 0; i<NODE; i++){ if(graph[i][u.val]){

if(vertex[i].state == 0){ myStack.push(vertex[i]); vertex[i].state = 1;

}

}

}

}}}

int main(){

node vertices[NODE]; node start;

char s;

for(int i = 0; i<NODE; i++){ vertices[i].val = i;

}

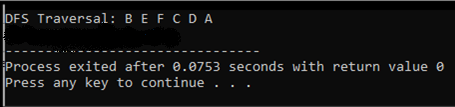
s = 'B';

start.val = s-'A';

cout << "DFS Traversal: "; dfs(vertices, start);

}

## Output:



**Time Complexity:**

## BFS: O(V+E) DFS: O(V+E)

**Justification:** Time complexity of both algorithms are same because in both the cases we have to traverse all the vertex and edges once which takes O(V) + O(E) time.

# Practical 6.2

## From a given vertex in a weighted graph, implement a program to find shortest paths to other vertices using Dijkstra’s algorithm.

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Adjacency Matrix of graph** | **Start Vertex** |
| **1** |  | **1** |
| **2** |  | **3** |

**Program Code**:

#include<iostream>

#include<climits>

using namespace std;

int minimumDist(int dist[], bool Tset[])

{

int min=INT\_MAX,index; for(int i=0;i<8;i++)

{

if(Tset[i]==false && dist[i]<=min)

{

min=dist[i]; index=i;

}

}

return index;

}

void Dijkstra(int graph[8][8],int src)

{

int dist[8]; bool Tset[8

for(int i = 0; i<8; i++)

{

dist[i] = INT\_MAX; Tset[i] = false;

}

dist[src] = 0;

for(int i = 0; i<8; i++)

{

int m=minimumDist(dist,Tset); Tset[m]=true;

for(int i = 0; i<8; i++)

{

if(!Tset[i] && graph[m][i] && dist[m]!=INT\_MAX && dist[m]+graph[m][i]<dist[i])

dist[i]=dist[m]+graph[m][i];

}

}

cout<<"Vertex\t\tDistance from source"<<endl; for(int i = 0; i<8; i++)

{

char str=65+i; cout<<str<<"\t\t\t"<<dist[i]<<endl;

}

}

int main()

{

int graph[8][8]={

{0, 0, 2, 0, 0, 0, 0, 0},

{6, 0, 0, 0, 0, 0, 0, 0},

{3, 8, 0, 0, 5, 0, 0, 0},

{0, 9, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 1, 0},

{0, 0, 0, 7, 0, 0, 0, 0},

{0, 0, 9, 0, 4, 0, 0, 3},

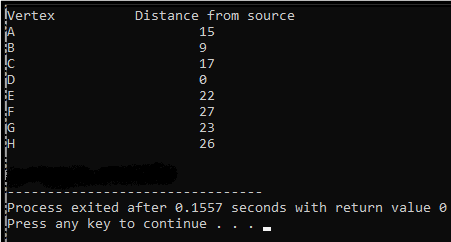
{0, 0, 0, 0, 0, 1, 6, 0}};

Dijkstra(graph,3);

return 0;

}

## Output:



**Time Complexity: O(V2)**

**Justification:** Time for visiting all vertices =O(V). Time required for processing one vertex=O(V). Time required for visiting and processing all the vertices = O(V)\*O(V) =O(V^2).

# Practical – 7

**Aim:** Backtracking

# Practical 7.1

Program to implement 8-Queen’s problem using Backtracking.

## **Program Code:**

#include<iostream>

using namespace std;

#define N 8

void printBoard(int board[N][N]) {

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++) cout << board[i][j] << " "; cout << endl;

}

}

bool isValid(int board[N][N], int row, int col) {

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

if (board[row][i]) return false;

for (int i=row, j=col; i>=0 && j>=0; i--, j--) if (board[i][j])

return false;

for (int i=row, j=col; j>=0 && i<N; i++, j--) if (board[i][j])

return false; return true;

}

bool solveNQueen(int board[N][N], int col) { if (col >= N)

return true;

for (int i = 0; i < N; i++) {

if (isValid(board, i, col) ) { board[i][col] = 1;

if ( solveNQueen(board, col + 1)) return true;

board[i][col] = 0;

}

}

return false;

}

bool checkSolution() {

int board[N][N];

for(int i = 0; i<N; i++) for(int j = 0; j<N; j++)

board[i][j] = 0; //set all elements to 0

if ( solveNQueen(board, 0) == false ) { cout << "Solution does not exist";

return false;

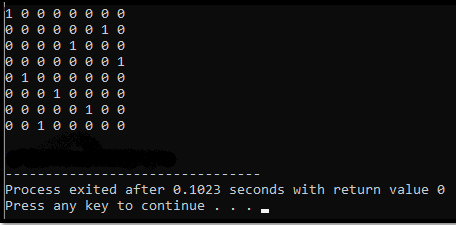
}

printBoard(board); return true;

}

int main() { checkSolution();}

**Output**:



## Time Complexity: O(n!)

**Justification:** The first queen has N placements, the second queen must not be in the same column as the first as well as at an oblique angle, so the second queen has N-1 possibilities, and so on, with a time complexity of O(N!).

# Practical – 8

**Aim:** String Matching Algorithm

# Practical 8.1

## **Suppose you are given a source string S[0 ..n − 1] of length n, consisting of symbols a and b. Suppose that you are given a pattern string P[0 ..m − 1] of length m < n, consisting of symbols a, b, and \*, representing a pattern to be found in string S. The symbol \* is a “wild card” symbol, which matches a single symbol, either a or b. The other symbols must match exactly. The problem is to output a sorted list M of valid “match positions”, which are positions j in S such that pattern P matches the substring S [j..j + |P|− 1]. For example, if S**

**= ababbab and P = ab\*, then the output M should be [0, 2]. Implement a straightforward, naive algorithm to solve the problem.**

## **Program Code:**

#include <iostream>

#include <string.h>

using namespace std;

int main()

{

char t[100], p[100];

int tn, pn, shift[20] = {0}, s = 0, i, j = 0, count

= 0, m = 0;

cout << "\nEnter The Text : "; cin >> t;

cout << "\nEnter The Pattern : "; cin >> p;

tn = strlen(t); pn = strlen(p);

while (s != (tn - pn + 1))

{

j = 0;

for (i = s; i < pn + s; i++)

{

if (p[j] == t[i])

{

count++;

if (count == pn)

{

count = 0; shift[m] = s; m++;

}

}

else

{

count = 0; break;

} j++;

} s++;

}

if (m > 0)

{

cout << "\n\nValid Shifts: \n"; for (i = 0; i < m; i++)

cout << shift[i] << "\n";

}

else

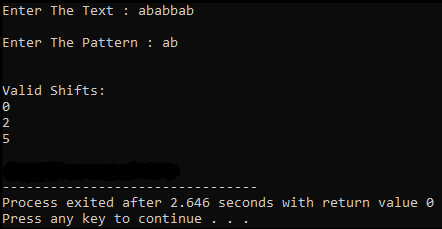
{

cout << "\n\nNo Valid Shifts.";

}

}

**Output**:



## Time Complexity: O((n-m+1)m)

**Justification:** The worst case occurs when the pattern is at last position and there are spurious hits all the way. To move pattern one position right, m comparisons are made. Searchable text in T has a length (n – m). Hence, in worst case algorithm runs in O(m\*(n – m)) time.

# Practical 8.2

## Implement Rabin Karp algorithm and test it on the following test cases:

|  |  |  |
| --- | --- | --- |
| **Test Case** | **String** | **Pattern** |
| 1 | 2359023141526739921 | 31415 q=13 |
| 2 | ABAAABCDBBABCDDEBCABC | ABC q=101 |

**Program Code**:

#include <string.h>

#include <iostream>

using namespace std;

#define d 10

void rabinKarp(char pattern[], char text[], int q) {

int m = strlen(pattern);

int n = strlen(text);

int i, j;

int c = 0; int p = 0; int t = 0; int h = 1;

for (i = 0; i < m - 1; i++) h = (h \* d) % q;

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

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

}

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

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

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

c++;

}

if (j == m)

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

}

if (i < n - m) { c++;

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

t = (t + q);}}

cout<<"\ncounter:"<<c;}

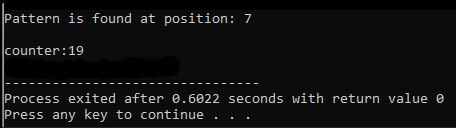
int main() {

char text[] = "2359023141526739921";

char pattern[] = "31415"; int q = 13; rabinKarp(pattern, text, q);

}

## Output:



**Time Complexity: O(nm)**

**Justification:** When all characters of pattern and text are the same as the hash values of all the substrings of text match with the hash value of pattern. Then we have to traverse all m characters of pattern as well as all n characters of text.