# Practical-1

## Aim: Implement and analyse the given algorithms.

[1.1] You are given two machines with capability to solve given problems (10 ^ (-4) *(2^n) Seconds and (10^ (-6) *(2^n) Seconds respectively. Implement a program that proves which algorithm is better for input instance size n=10, 15, 20, 30 and 45. Also print time taken by each machine for different values on n. Instead of machine A if we choose to use machine C with capability (10^ (-2) *(n^3), what kind of significant improvement we may get. Show the detail analysis of it using appropriate tool.

[1.2] Fibonacci Series (Iterative and Recursive).

[1.3] Matrix Addition and Matrix Multiplication (Iterative).

[1.4] Linear Search and Binary Search.

[1.5] Identify, Explain and Analyse [Time complexity] the code shared with you.

**Software Requirement:** GNU Compiler Collection (GCC), MS Excel.

**Hardware Requirement:** Desktop Computer.

# [1.1]

**Aim:** You are given two machines with capability to solve given problems (10 ^ (-4) *(2^n) Seconds and (10^ (-6) *(2^n) Seconds respectively. Implement a program that proves which algorithm is better for input instance size n=10, 15, 20, 30 and 45. Also print time taken by each machine for different values on n. Instead of machine A if we choose to use machine C with capability (10^ (-2) *(n^3), what kind of significant improvement we may get. Show the detail analysis of it using appropriate tool.

## Theory:

Here we have given three machines, each machine has capability to solve given problems

(10 ^ (-4) *(2^n) Seconds and (10^ (-6) *(2^n) Seconds and (10^ (-2) *(n^3) seconds respectively.

Let's say Machine A has capability to solve given problems is (10 ^ (-4) *(2^n) Seconds,

Machine B has capability to solve given problems is (10^ (-6) *(2^n) Seconds,

Machine C has capability to solve given problems is (10^ (-2) *(n^3) Seconds.

As we can see Machine B has better Hardware configuration than other two but it is performing 2^n complexity problem which will take time exponentially.

Machine A is also performing 2^n complexity problem so, here both Machine A and Machine B are performing same time complexity problem theatrically Machine B will be faster than Machine A.

Machine C is performing n^3 complexity problem So, theatrically for big value of n Machine C will be faster than Machine A and B.

## Algorithm:

Function Machine-A (int I)

return double (pow (10, -4)) * double (pow (2, I));

Function Machine-B (int I)

return double (pow (10, -6)) * double (pow (2, I));

Function Machine-C (int I)

return double (pow (10, -2)) * double (pow (I, 3));

Let n [5] = {10,15,20,30,45}

Comparison between machine A and B:

For I=0 to 5

Do machine-A = Machine-A(n[I])

machine-B = Machine-A(n[I])

Do if machine-A > machine-B

Then print “Machine B is better”

Else print “Machine A is better”

Comparison between machine B and C:

For i=0 to 5

Do machine-B = Machine-A(n[I])

machine-C = Machine-A(n[I])

Do if machine-C > machine-B

Then print “Machine B is better”

Else print “Machine C is better”

## Code:

```
#include <bits/stdc++.h>
using namespace std;
double MachineA(int n)
{
    double result = pow(10, -4) * pow(2, n);
    return result;
}
double MachineB(int n)
{
    double result = pow(10, -6) * pow(2,n);
    return result;
}
double MachineC(int n)
{
    double result = pow(10, -2) * pow(n, 3);
    return result;
}
```

```
int main()
{

    const char separator = ' ';
    const int nameWidth = 15;
    const int numWidth = 15;
    int a[5] = {10, 15, 20, 30, 45};

    cout << left << setw(nameWidth) << setfill(separator) << "Number";
    cout << left << setw(nameWidth) << setfill(separator) << "Machine A
";
    cout << left << setw(nameWidth) << setfill(separator) << "Machine B
" << endl;
    for (int i : a)
    {
        double A = MachineA(i);
        double B = MachineB(i);
        cout << left << setw(nameWidth) << setfill(separator) << i;
        cout << left << setw(nameWidth) << setfill(separator) << A;
        cout << left << setw(nameWidth) << setfill(separator) << B;
        cout << left << setw(nameWidth) << setfill(separator) << ((A <
B) ? ("Machine A is Faster then Machine B") : ("Machine B is Faster tha
n Machine A")) << endl;
    }
    cout << endl;
    cout << left << setw(nameWidth) << setfill(separator) << "Number";
    cout << left << setw(nameWidth) << setfill(separator) << "Machine B
";
    cout << left<< setw(nameWidth)<< setfill(separator)<< "Machine C" <
< endl;
    for (int i : a)
    {
        double B = MachineB(i);
        double C = MachineC(i);
        cout << left << setw(nameWidth) << setfill(separator) << i;
        cout << left << setw(nameWidth) << setfill(separator) << B;
        cout << left << setw(nameWidth) << setfill(separator) << C;
        cout << left << setw(nameWidth) << setfill(separator) << ((C <
B) ? ("Machine C is Faster then Machine B") : ("Machine B is Faster tha
n Machine C")) << endl;
    }
    return 0;
}
```

## Output:

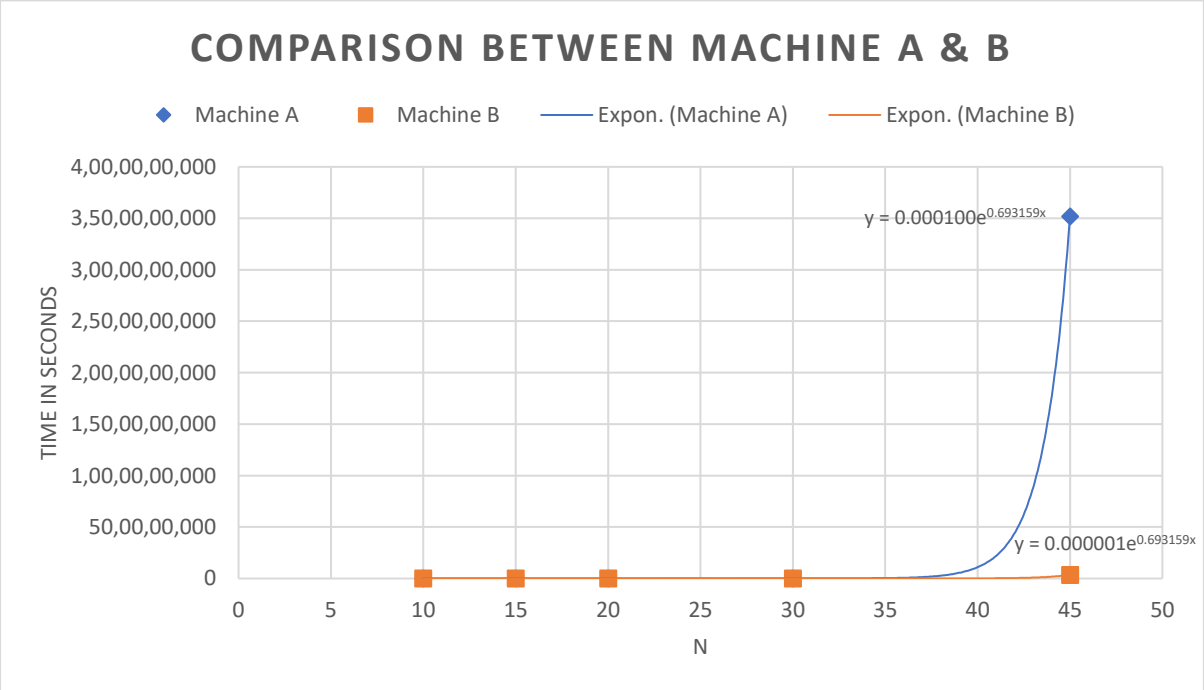
```
Number          Machine A       Machine B
10              0.1024          0.001024        Machine B is Faster than Machine A
15              3.2768          0.032768        Machine B is Faster than Machine A
20              104.858         1.04858         Machine B is Faster than Machine A
30              107374          1073.74         Machine B is Faster than Machine A
45              3.51844e+009    3.51844e+007    Machine B is Faster than Machine A

Number          Machine B       Machine C
10              0.001024        10              Machine B is Faster than Machine C
15              0.032768        33.75           Machine B is Faster than Machine C
20              1.04858         80              Machine B is Faster than Machine C
30              1073.74         270             Machine C is Faster then Machine B
45              3.51844e+007    911.25          Machine C is Faster then Machine B
```

## Analysis:

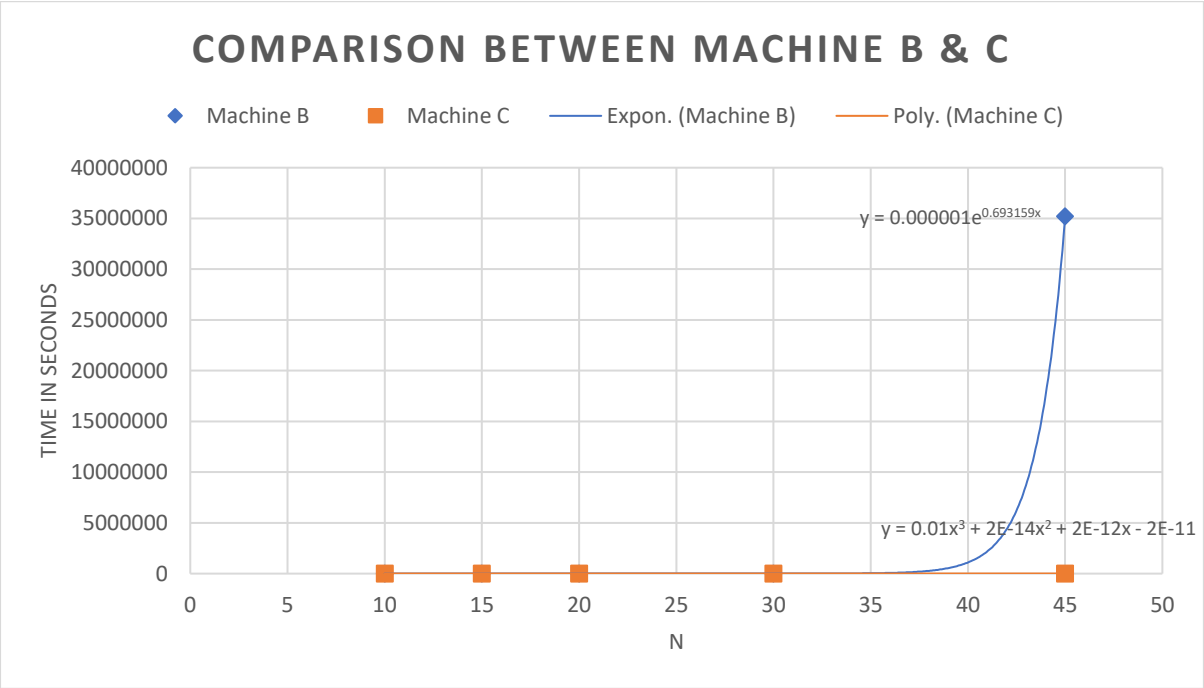
**Comparison between Machine A & B:**

| N | Machine A | Machine B |
|---|---|---|
| 10 | 0.1024 | 0.001024 |
| 15 | 3.2768 | 0.032768 |
| 20 | 104.858 | 1.04858 |
| 30 | 107374 | 1073.74 |
| 45 | 3.52E+09 | 3.52E+07 |



**Comparison between Machine B & C:**

| N | Machine B | Machine C |
|---|---|---|
| 10 | 0.001024 | 10 |
| 15 | 0.032768 | 33.75 |
| 20 | 1.04858 | 80 |
| 30 | 1073.74 | 270 |
| 45 | 3.52E+07 | 911.25 |




**Conclusion:** After this Posteriori Analysis, from the both graph I conclude that finding Less Time Complexity of an algorithm or optimization of an algorithm is very much important.
As we can see that Machine C has less hardware configuration than other two but it was performing n^3($n^3$)-time complexity problem while other two machines are performing 2^n($2^n$) time complexity problem till Machine C is Performing better for larger value of N. So, if we can optimize an algorithm and can reduce its time complexity, we must do it.

**[1.2]**

**Aim:** Fibonacci Series (Iterative and Recursive).

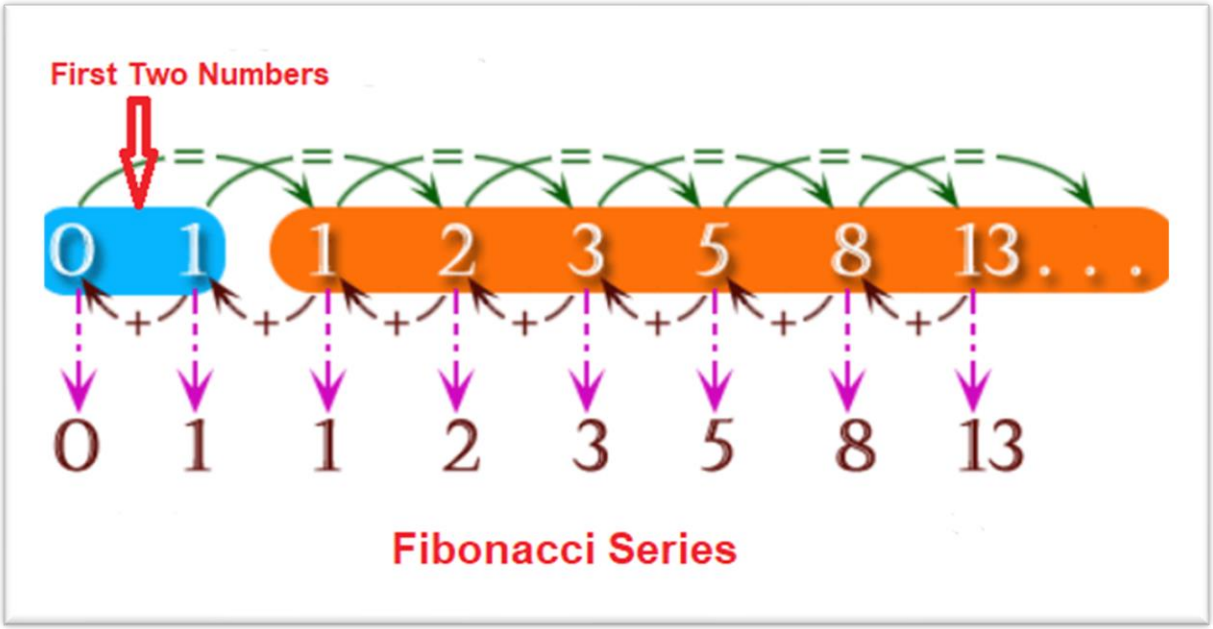
## Theory:

The Fibonacci sequence is a set of numbers that starts with a one or a zero, followed by a one, and proceeds based on the rule that each number (called a Fibonacci number) is equal to the sum of the preceding two numbers. If the Fibonacci sequence is denoted F (n), where n is the first term in the sequence, the following equation obtains for n = 0, where the first two terms are defined as 0 and 1 by convention:

F (0) = 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ...

In some texts, it is customary to use n = 1. In that case, the first two terms are defined as 1 and 1 by default, and therefore:

F (1) = 1, 1, 2, 3, 5, 8, 13, 21, 34 ...



## Algorithm:

1) **Using Iterative Approach:**

Function Fibonacci (int n)
    Let a = 0, b = 1
    For i=1 to n
        Do print a
        c = a + b
        a = b
        c = a
    End for

2) **Using Recursive Approach:**

Function Fibonacci (int n)
    If n == 0
        Then return 0
    If n == 1
        Then return 1
    Else return Fibonacci (n-1) + Fibonacci (n-2)

## Code:

1) **Using Iterative Approach:**

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int n, count1 = 0;
    cout<<"Enter Number:";
    cin >> n;
    int *a = new int(n);
    a[0] = 0;
    a[1] = 1;
    cout << a[0] << " " << a[1] << " ";
    for (int i = 2; i < n; i++)
    {
        count1++;
        a[i] = a[i - 1] + a[i - 2];
        count1++;
        cout << a[i] << " ";
    }
    cout << endl
         << "Count:" << count1;
    return 0;
}
```

**2) Using Recursive Approach:**

```cpp
#include <bits/stdc++.h>
using namespace std;
int count1 = 0;

int fibonacci(int n)
{
    count1++;
    if (n <= 1)
        return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main()
{

    int n;
    cout<<"Enter Number:";
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        cout << fibonacci(i) << " ";
    }
    cout << endl
         << "Steps:" << count1;
    return 0;
}
```

## Output:

**1) Using Iterative Approach:**

```
Enter Number:25
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368
Count:46
```

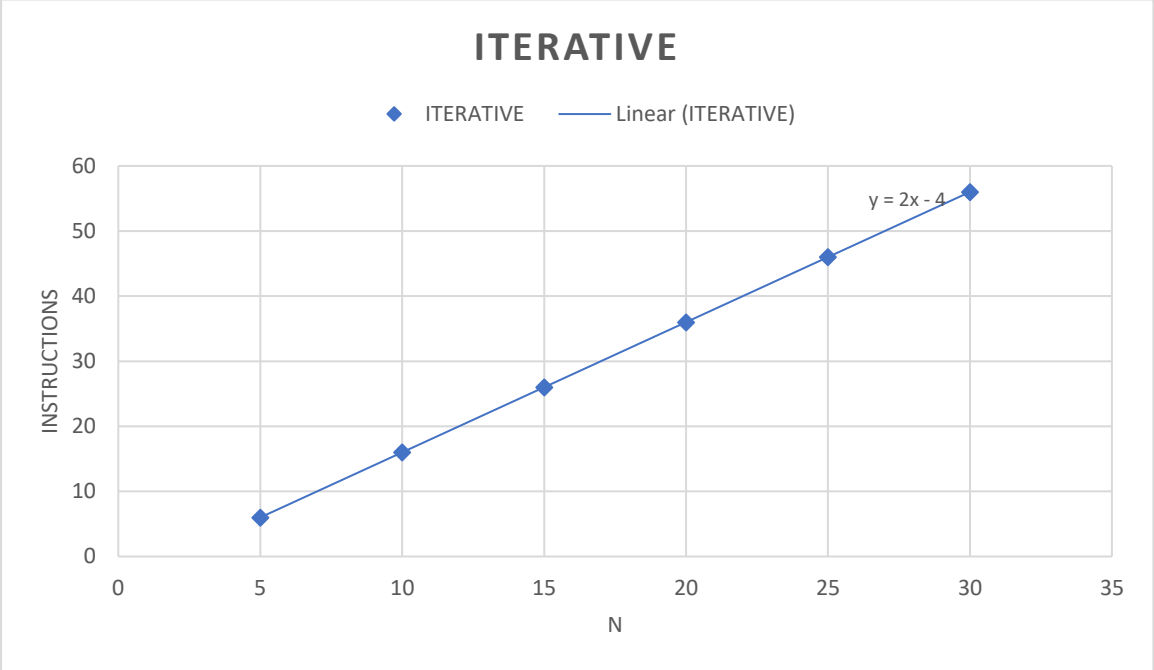
**2) Using Recursive Approach:**

```
Enter Number:25
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368
Steps:392809
```

## Analysis:

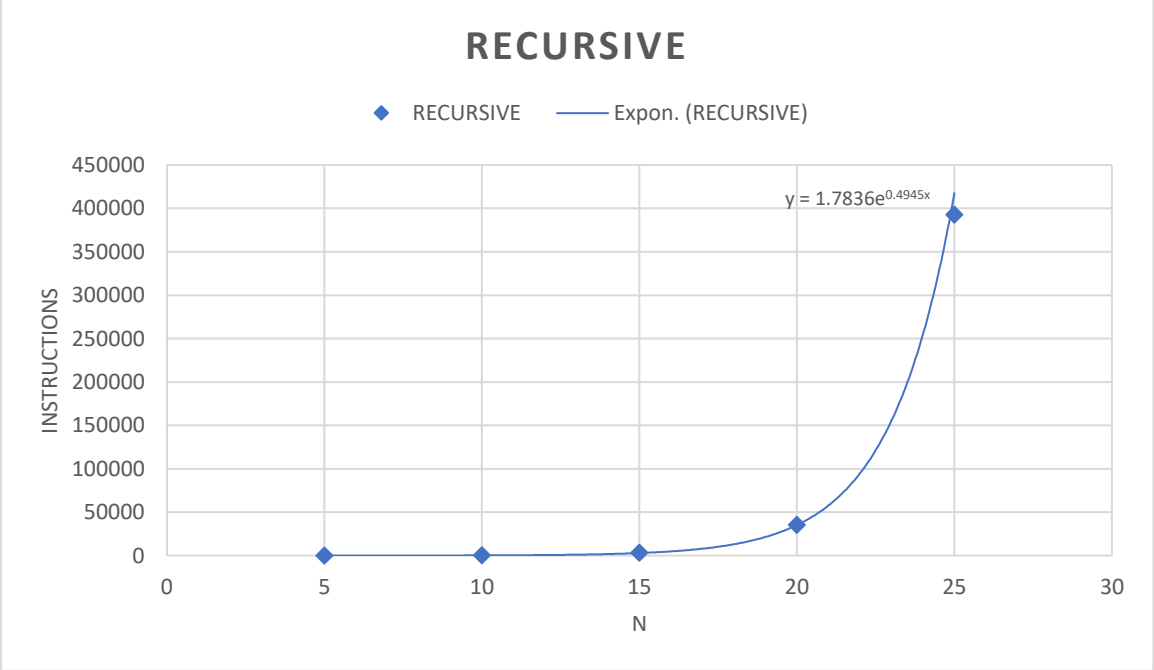
### 1) Iterative Approach:

| N | ITERATIVE |
|---|---|
| 5 | 6 |
| 10 | 16 |
| 15 | 26 |
| 20 | 36 |
| 25 | 46 |
| 30 | 56 |



**2) Recursive Approach:**

| N | RECURSIVE |
|---|---|
| 5 | 19 |
| 10 | 276 |
| 15 | 3177 |
| 20 | 35400 |
| 25 | 392809 |
| 30 | 4356586 |




**Conclusion:** After this Posteriori Analysis, I conclude that single problem 'Finding Fibonacci Series' which has two approaches 1) Iterative and 2) Recursive both have different time complexities.

From the graph I can say Time Complexity of

1) Iterative Algorithms is O(n).
2) Recursive Algorithm is $O(2^n)$.

## [1.3]

**Aim:** Matrix Addition and Matrix Multiplication (Iterative).

## Theory:

### 1) Matrix Addition:

Matrix Addition is the operation of adding two matrices by adding the corresponding entries together.

Two matrices must have an equal number of rows and columns to be added. In which case, the sum of two matrices **A** and **B** will be a matrix which has the same number of rows and columns as **A** and **B**. The sum of **A** and **B**, denoted **A** + **B**, is computed by adding corresponding elements of **A** and **B**:

$$\mathbf{A}+\mathbf{B} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \cdots & a_{1n}+b_{1n} \\ a_{21}+b_{21} & a_{22}+b_{22} & \cdots & a_{2n}+b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}+b_{m1} & a_{m2}+b_{m2} & \cdots & a_{mn}+b_{mn} \end{bmatrix}$$

### 2) Matrix Multiplication:

Matrix Multiplication is a binary operation that produces a matrix from two matrices. For matrix multiplication, the number of columns in the first matrix must be equal to the number of rows in the second matrix. The resulting matrix, known as the **matrix product**, has the number of rows of the first and the number of columns of the second matrix. The product of matrices **A** and **B** is denoted as **AB**.

If **A** is an $m \times n$ matrix and **B** is an $n \times p$ matrix,

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

the matrix product **C** = **AB** (denoted without multiplication signs or dots) is defined to be the m × p matrix such that,

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj},$$

for $i = 1, ..., m$ and $j = 1, ..., p$.

That is, the entry of the product is obtained by multiplying term-by-term the entries of the $i^{th}$ row of **A** and the $j^{th}$ column of **B**, and summing these $n$ products. In other words, is the dot product of the $i^{th}$ row of **A** and the $j^{th}$ column of **B**.

Therefore, **AB** can also be written as,

$$\mathbf{C} = \begin{pmatrix} a_{11}b_{11} + \cdots + a_{1n}b_{n1} & a_{11}b_{12} + \cdots + a_{1n}b_{n2} & \cdots & a_{11}b_{1p} + \cdots + a_{1n}b_{np} \\ a_{21}b_{11} + \cdots + a_{2n}b_{n1} & a_{21}b_{12} + \cdots + a_{2n}b_{n2} & \cdots & a_{21}b_{1p} + \cdots + a_{2n}b_{np} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{11} + \cdots + a_{mn}b_{n1} & a_{m1}b_{12} + \cdots + a_{mn}b_{n2} & \cdots & a_{m1}b_{1p} + \cdots + a_{mn}b_{np} \end{pmatrix}$$

## Algorithm:

**1) Matrix Addition:**

If row1 == row2 && col1 == col2

For i=0 to row1 - 1

For j=0 to col1 – 1

Do Result [i][j] = Matrix-1 [i][j] + Matrix-2 [i][j]

Print Result [i][j]

End for

End for

Else print “Matrix addition is not possible”

**2) Matrix Multiplication:**

If col1 == row2

For i=0 to row1 - 1

For j=0 to col2 – 1

Let sum = 0

For k=0 to col1 – 1

sum = sum + Matrix-1 [i][k] * Matrix-2 [k][j]

Result [i][j] = sum

End for

End for

Else print “Matrix multiplication is not possible”

## Code:

### 1) Matrix Addition:

```
#include <bits/stdc++.h>
using namespace std;
int **makeMatrix(int m, int n)
{
    int **a = new int *[m];
    for (int i = 0; i < m; i++)
    {
        a[i] = new int[n];
    }

    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < n; j++)
        {
            cin >> a[i][j];
        }
    }
    return a;
}
void printArray(int **a, int m, int n)
{
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < n; j++)
        {
            cout << a[i][j] << " ";
        }
        cout << endl;
    }
}

int main()
{
    int n, m, p, q;
    cout << "Enter dimension of Matrix A(m n):";
    cin >> m >> n;
    cout << "Enter dimension of Matrix B(m n):";
    cin >> p >> q;
    if (m == p && n == q)
```

```
    {
        cout << "Enter A:" << endl;
        int **a1 = makeMatrix(m, n);
        cout << "Enter B:" << endl;
        int **a2 = makeMatrix(p, q);

        int **c = new int *[m], count1 = 0;
        for (int i = 0; i < m; i++)
        {
            c[i] = new int[n];
            count1++;
        }

        for (int i = 0; i < m; i++)
        {
            count1++;
            for (int j = 0; j < n; j++)
            {
                count1++;
                c[i][j] = a1[i][j] + a2[i][j];
                count1++;
            }
        }
        cout << "C:" << endl;
        printArray(c, m, n);
        cout << "\nCount:" << count1;
    }
    else
        cout << "Can't sum this two Matrices";
    return 0;
}
```

### 3) Matrix Multiplication (Iterative):

```
#include <bits/stdc++.h>
using namespace std;
int count1=0;

int **makeMatrix(int m, int n)
{
    int **a = new int *[m];
    for (int i = 0; i < m; i++)
    {
        a[i] = new int[n];
    }

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

```
        {
            cin >> a[i][j];
        }
    }
    return a;
}
void printArray(int **a, int m, int n)
{
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < n; j++)
        {
            cout << a[i][j] << " ";
        }
        cout << endl;
    }
}

int main()
{
    int m, n, p, q, sum = 0;
    cout << "Enter dimension of matrix a (m n):";
    cin >> m >> n;
    cout << "Enter dimension of matrix B (p q):";
    cin >> p >> q;

    if (n == p)
    {
        cout<<"Enter matrix A:\n";
        int **a1 = makeMatrix(m, n);
        cout<<"Enter matrix B:\n";
        int **a2 = makeMatrix(p, q);

        int **a3 = new int *[m];
        for (int i = 0; i < m; i++)
        {
            a3[i] = new int[q];
        }

        for (int i = 0; i < m; i++)
        {
            count1++;
            for (int j = 0; j < q; j++)
            {
                count1++;
                for (int k = 0; k < p; k++)
                {
                    count1++;
```

```
                    sum += a1[i][k] * a2[k][j];
                }
                count1++;
                a3[i][j] = sum;
                count1++;
                sum = 0;
                count1++;
            }
        }
        cout<<"Multiplied Matrix:\n";
        printArray(a3, m, q);
    }
    else
        cout << "Can't multiply this two matrices";


    cout<<"\n Count:"<<count1;

    return 0;
}
```

## Output:

1) **Matrix Addition:**

```
Enter dimension of Matrix A(m n):4 4
Enter dimension of Matrix B(m n):4 4
Enter A:
7 0 2 6
8 9 4 0
9 0 0 0
1 9 3 3
Enter B:
2 4 3 4
9 0 8 2
7 8 6 1
3 1 4 8
C:
9 4 5 10
17 9 12 2
16 8 6 1
4 10 7 11

Count:40
```

**2) Matrix Multiplication (Iterative):**

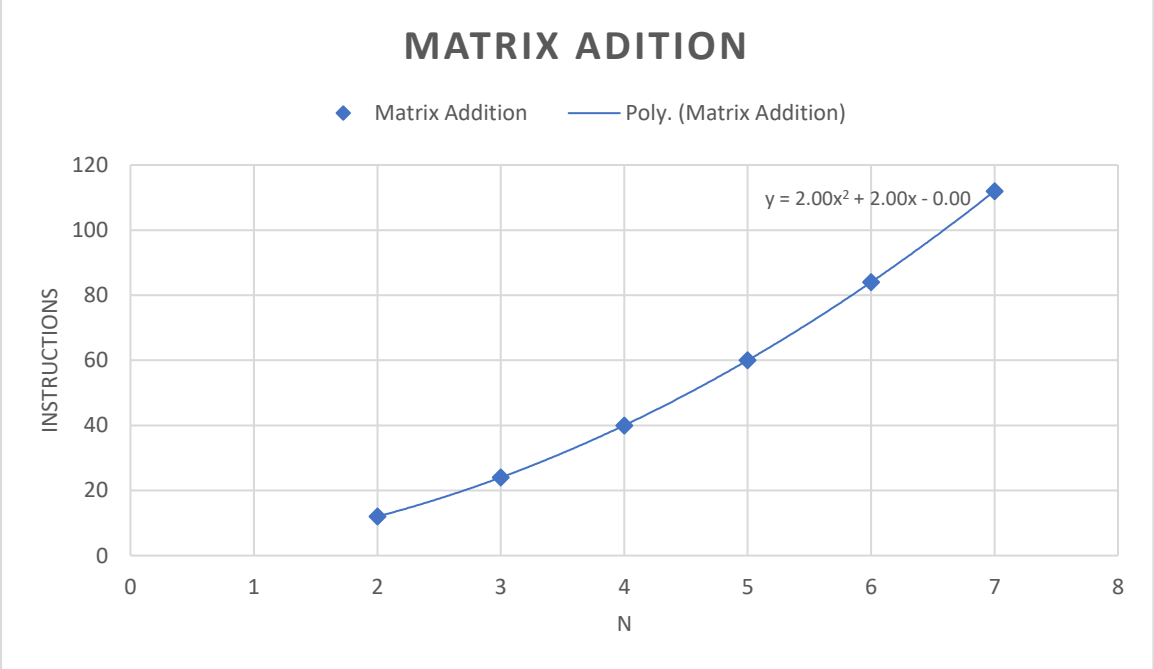
```
Enter dimension of matrix a (m n):4 5
Enter dimension of matrix B (p q):5 3
Enter matrix A:
4 9 3 6 0
2 5 6 2 0
7 5 7 3 2
3 3 2 0 4
Enter matrix B:
9 9 9
0 1 7
0 1 2
3 0 2
2 2 0
Multiplied Matrix:
54 48 117
24 29 69
76 79 118
35 40 52

 Count:112
```

## Analysis:

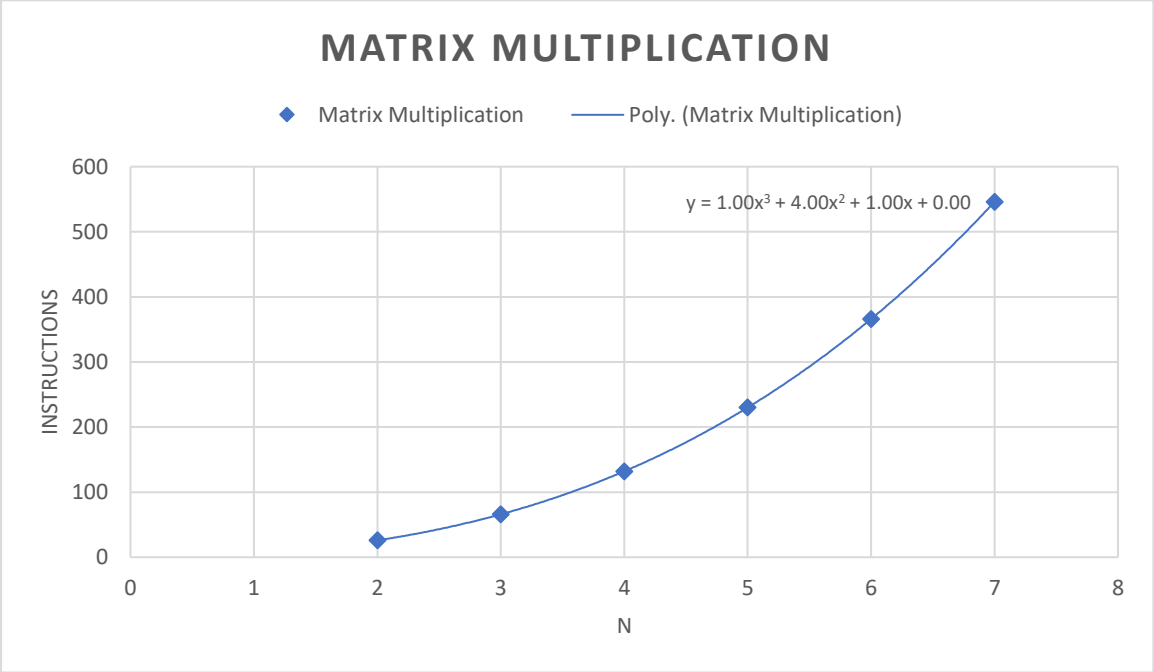
### 1) Matrix Addition:

| MxN | Matrix Addition |
|---|---|
| 2 | 12 |
| 3 | 24 |
| 4 | 40 |
| 5 | 60 |
| 6 | 84 |
| 7 | 112 |



**2) Matrix Multiplication (Iterative):**

| MxN | Matrix Multiplication |
|---|---|
| 2 | 26 |
| 3 | 66 |
| 4 | 132 |
| 5 | 230 |
| 6 | 366 |
| 7 | 546 |




**Conclusion:** After this Posteriori Analysis from the graphs, I conclude that Time Complexity of

1) Matrix Addition Algorithms is $O(n^2)$.
2) Matrix Multiplication Iterative Algorithm is $O(n^3)$.

## [1.4]

**Aim:** Linear Search and Binary Search.

## Theory:

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful and the process returns the location of that element, otherwise the search is called unsuccessful.

There are two popular search methods that are widely used in order to search some item into the list. However, choice of the algorithm depends upon the arrangement of the list.

### 1) Linear Search:

Linear search is the simplest search algorithm and often called sequential search. In this type of searching, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match found then location of the item is returned otherwise the algorithm return NULL.

Linear search is mostly used to search an unordered list in which the items are not sorted.

### 2) Binary Search:

Binary search is the search technique which works efficiently on the sorted lists. Hence, in order to search an element into some list by using binary search technique, we must ensure that the list is sorted.

Binary search follows divide and conquer approach in which, the list is divided into two halves and the item is compared with the middle element of the list. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves depending upon the result produced through the match.

## Algorithm:

1) **Linear Search:**

```
Let count=0

For i=0 to n-1
        Do if array[i] == element
                Print "The element found at index i"
        count++
Check if count > n
        print "element not found"
```

2) **Binary Search:**

```
Let minimum = 0
Do maximum = n -1

While min <= max
        Do mid = (minimum + maximum) / 2

        If array[mid] < element
                minimum = mid + 1
        else If array[mid] > element
                maximum = mid -1
        else print "The element found at index mid"

Check if min > max
        Print "element not found"
```

## Code:

### 1) Linear Search:

```
#include<bits/stdc++.h>
using namespace std;
int count1=0;
int search(int *a,int n,int f){
    for(int i=0;i<n;i++)
    {
        count1++;
        if(a[i]==f) return i;
        count1++;
    }
    return -1;

}
int main(){
    int n,f;
    cout<<"Enter the elements number:";
    cin>>n;
    int *a=new int(n);
    cout<<"Enter Elements:";
    for(int i=0;i<n;i++){
        cin>>a[i];
    }
    cout<<"Enter element to be searched";
    cin>>f;
    int find=search(a,n,f);
    (find==-
1) ? cout<< "Element is not present in array" : cout << "Element is present
 at index " << find;

    cout<<"\nCount:"<<count1;
    return 0;
}
```

**2) Binary Search:**

```
#include<bits/stdc++.h>
using namespace std;
int count1=0;
int Bsearch(int *a,int l,int r,int f){
    while(l<=r){
        count1++;

        int m=(l+r)/2; count1++;

        if(a[m]==f){
             count1++;
            return m;
        }


        if(a[m]<f){
             count1++;
            l=m+1;
        }

        else{
             count1++;
            r=m-1;
        }
    }
     count1++;
      return -1;
}
int main(){
    int n,f;
    cout<<"Enter the elements number:";
    cin>>n;
    int *a=new int(n);
    cout<<"Enter Elements:";
    for(int i=0;i<n;i++){
        cin>>a[i];
    }
    cout<<"Enter element to be searched";
    cin>>f;
    int find=Bsearch(a,0,n-1,f);
    (find==-
1) ? cout<< "Element is not present in array" : cout << "Element is present
 at index " << find;
    cout<<"\nCount:"<<count1;
    return 0;
}
```

## Output:

1) **Linear Search:**
   a) **Best Case:**

```
Enter the elements number:10
Enter Elements:1 2 3 4 5 6 7 8 9 10
Enter element to be searched:1
Element is present at index 0
Count:1
```

   b) **Average Case:**

```
Enter the elements number:10
Enter Elements:1 2 3 4 5 6 7 8 9 10
Enter element to be searched:4
Element is present at index 3
Count:7
```

   c) **Worst Case:**

```
Enter the elements number:10
Enter Elements:1 2 3 4 5 6 7 8 9 10
Enter element to be searched:10
Element is present at index 9
Count:19
```

2) **Binary Search:**
   a) **Best Case:**

```
Enter the elements number:10
Enter Elements:1 2 3 4 5 6 7 8 9 10
Enter element to be searched:5
Element is present at index 4
Count:3
```

   b) **Average Case:**

```
Enter the elements number:10
Enter Elements:1 2 3 4 5 6 7 8 9 10
Enter element to be searched:2
Element is present at index 1
Count:6
```

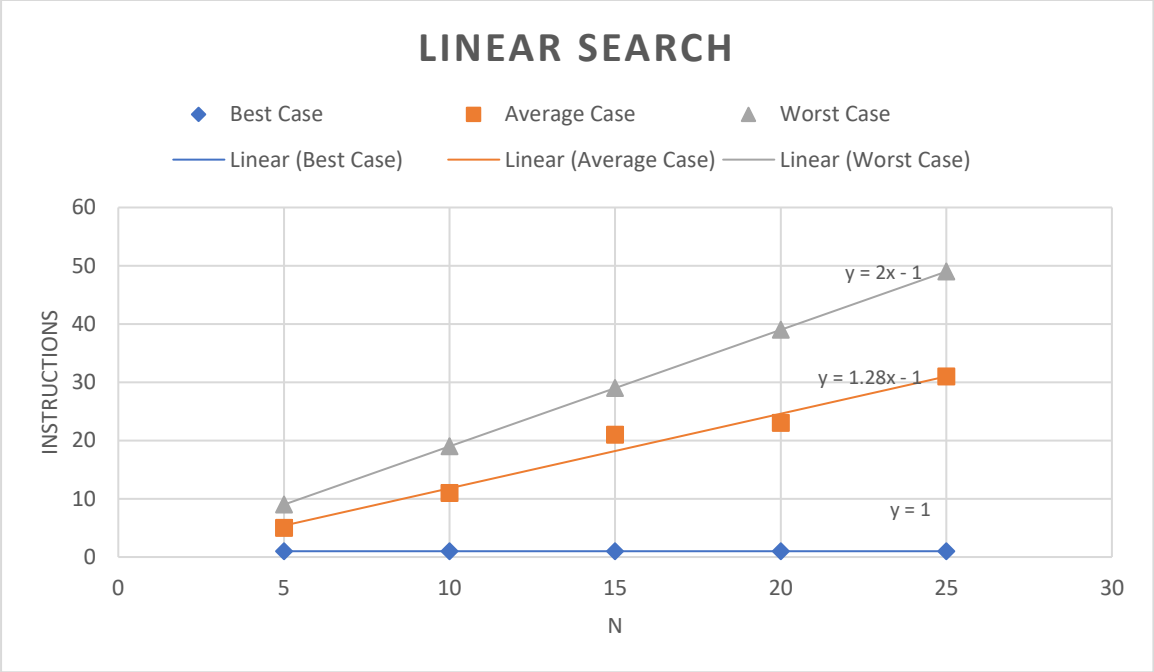
   c) **Worst Case:**

```
Enter the elements number:10
Enter Elements:1 2 3 4 5 6 7 8 9 10
Enter element to be searched:4
Element is present at index 3
Count:12
```

## Analysis:

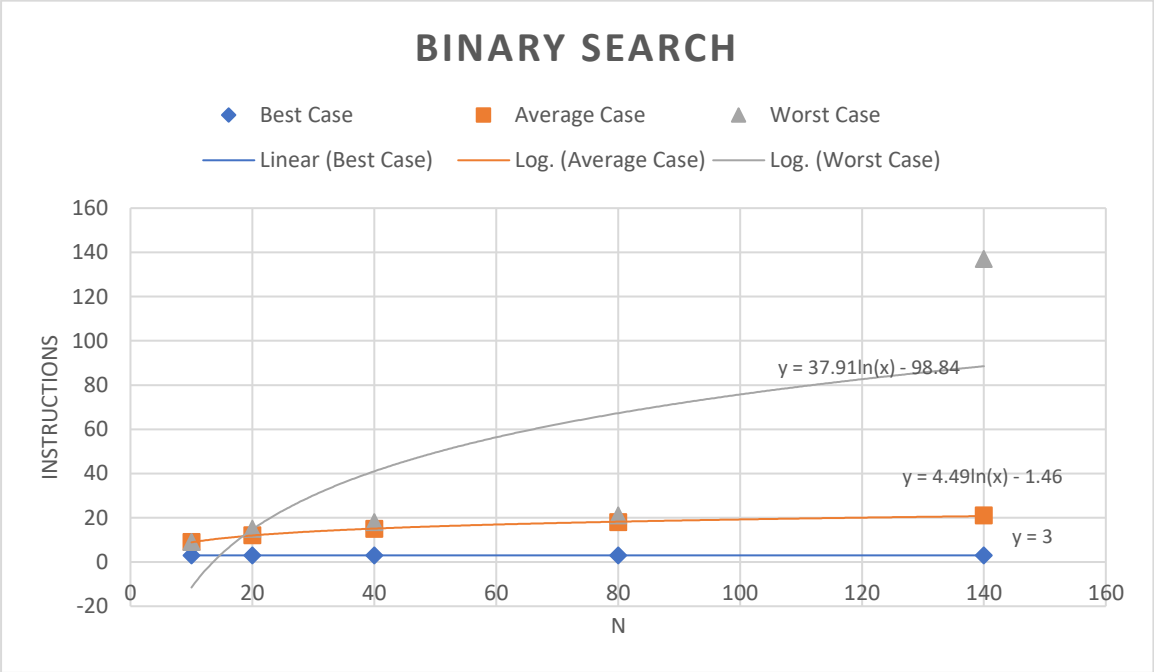
### 1) Linear Search:

| N | Best Case | Average Case | Worst Case |
|---|---|---|---|
| 5 | 1 | 5 | 9 |
| 10 | 1 | 11 | 19 |
| 15 | 1 | 21 | 29 |
| 20 | 1 | 23 | 39 |
| 25 | 1 | 31 | 49 |



**2) Binary Search:**

| N | Best Case | Average Case | Worst Case |
|---|---|---|---|
| 10 | 3 | 9 | 9 |
| 20 | 3 | 12 | 15 |
| 40 | 3 | 15 | 18 |
| 80 | 3 | 18 | 21 |
| 140 | 3 | 21 | 137 |




**Conclusion:** After this Posteriori Analysis from the graphs, I conclude that Time Complexity of

1) Linear Search Algorithm for
   a) Best Case is O (1)
   b) Average Case is O (n)
   c) Worst Case is O (n).
2) Binary Search Algorithm for
   a) Best Case is O (1)
   b) Average Case is O (log n)
   c) Worst Case is O (log n).

## [1.5]

**Aim:** Identify, Explain and Analyse [Time complexity] the code shared with you (Insertion Sort).

## Theory:

I have identified that the shared code is of Insertion Sorting.

Insertion Sort:

Insertion sort is the simple sorting algorithm which is commonly used in the daily lives while ordering a deck of cards. In this algorithm, we insert each element onto its proper place in the sorted array. This is less efficient than the other sort algorithms like quick sort, merge sort, etc.

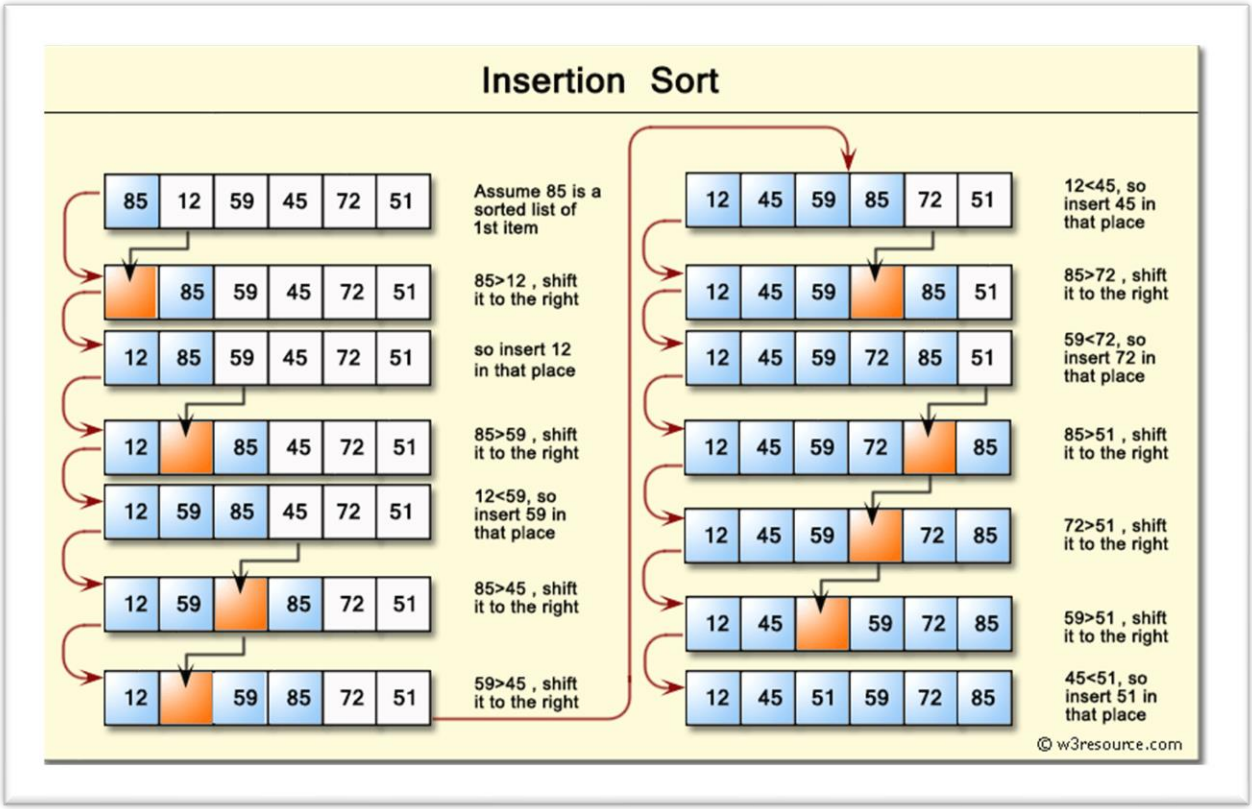
Technique:

Consider an array A whose elements are to be sorted. Initially, A[0] is the only element on the sorted set. In pass 1, A[1] is placed at its proper index in the array.

In pass 2, A[2] is placed at its proper index in the array. Likewise, in pass n-1, A[n-1] is placed at its proper index into the array.

To insert an element A[k] to its proper index, we must compare it with all other elements i.e. A[k-1], A[k-2], and so on until we find an element A[j] such that, A[j]<=A[k].

All the elements from A[k-1] to A[j] need to be shifted and A[k] will be moved to A[j+1].



## Algorithm:

Insertion_sort (int p[],int n)

1. Repeat through step 4 for pass i=1, 2, ……., n
2. Store the element of array in some variable temp=p[i] and j=i-1
3. while (j>0) and (p[j]>temp)
4. p[j+1]=p[j]
5. J=j-1
6. p[j+1]=temp
7. Then print sorted array
8. Return

## Code:

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 10
// #define SIZE 15
//  #define SIZE 12

int count1=0;
void swap(int *a, int *b);
void sort(int *a,const int size);

int main()
{
    // int a[SIZE] = {3, 4 , 6, 1, 5, 8, 7, 9, 0, 2};
    // int a[SIZE] = {0,1,2,3,4,5,6,7,8,9};
     int a[SIZE] = {9,8,7,6,5,4,3,2,1,0};
    // int a[SIZE] = {3, 4 , 6, 1, 5, 8, 7, 9, 0, 2, 10,12};
    sort(a, SIZE);
    int i;
     printf("Array before sorting:\n");
     printf("3 4 6 1 5 8 7 9 0 2 \n");
   // display(a,SIZE);
printf("Array After sorting:\n");
    for(i = 0; i < SIZE; i++)
    {

        printf("%d ",a[i]);
    }

    printf("\n Count:: %d",count1);
    return 0;
```

```
}

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
void sort(int *a,const int size)
{

    int i,j, k;
    for (i = 1; i < size; ++i)
    {
        k = a[i]; count1++;
        j = i - 1;  count1++;
        while ((j >= 0) && (k < a[j]))
        {
            count1++;
            a[j + 1] = a[j];
            count1++;
            --j;
            count1++;
        }
        count1++;
        a[j + 1] = k;
    }
}
```

## Output:

**a) Best Case:**

```
Array before sorting:
0 1 2 3 4 5 6 7 8 9
Array After sorting:
0 1 2 3 4 5 6 7 8 9
 Count:: 27
```

**b) Average Case:**

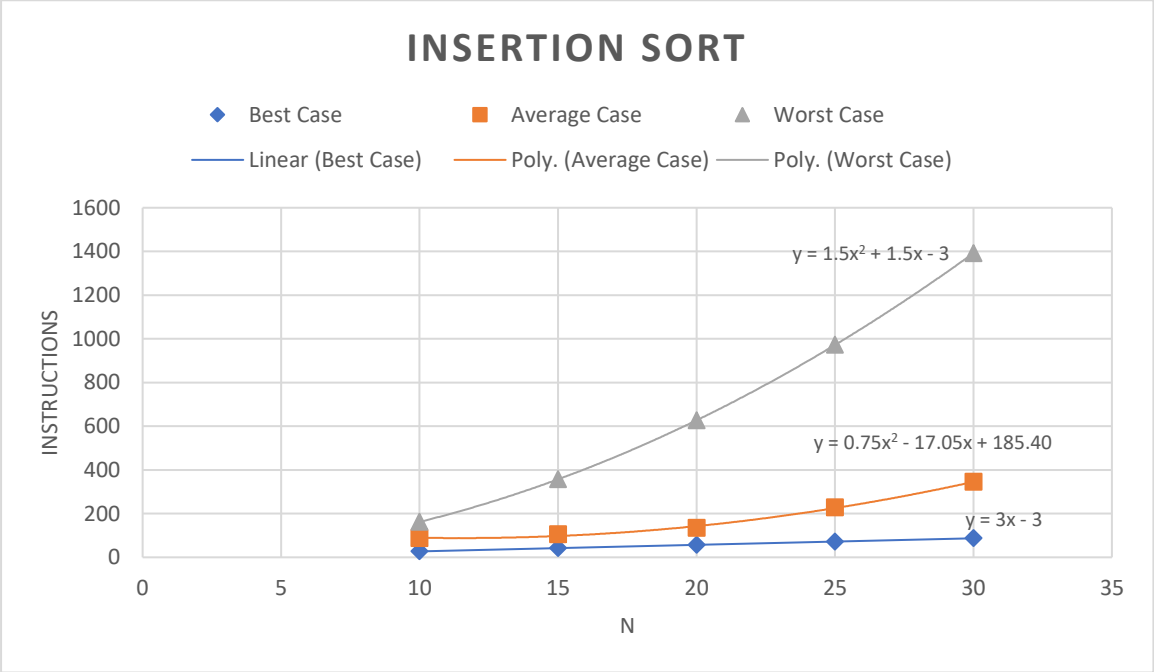
```
Array before sorting:
3 4 6 1 5 8 7 9 0 2
Array After sorting:
0 1 2 3 4 5 6 7 8 9
 Count:: 87
```

**c) Worst Case:**

```
Array before sorting:
9 8 7 6 5 4 3 2 1 0
Array After sorting:
0 1 2 3 4 5 6 7 8 9
 Count:: 162
```

## Analysis:

| N | Best Case | Worst Case |
|---|---|---|
| 10 | 27 | 162 |
| 15 | 42 | 357 |
| 20 | 57 | 627 |
| 25 | 72 | 972 |
| 30 | 87 | 1392 |




**Conclusion:** After this Posteriori Analysis from the graph, I conclude that Time Complexity of Insertion Sort in
a) Best Case is: O (n)
b) Worst Case is: O ($n^2$).

## Final Conclusion:

| Practical 1: Implement and analyse algorithms | | | | | |
|---|---|---|---|---|---|
| **Sr. No** | **Problem Definition** | **I/P Quality** | **Practical Time Complexity with Equation** | | **Theoretical Complexity** |
| 1.1 (Machine Complexities are given) | Machine A | -- | $y = 0.000100e^{0.693159x}$ | $O(2^n)$ | $O(2^n)$ |
| | Machine B | -- | $y = 0.000001e^{0.693159x}$ | $O(2^n)$ | $O(2^n)$ |
| | Machine C | -- | $y = 0.01x^3 + 2E\text{-}14x^2 + 2E\text{-}12x - 2E\text{-}11$ | $O(n^3)$ | $O(n^3)$ |
| 1.2 | Fibonacci Series Iterative | -- | $y = 2x - 4$ | O(n) | O(n) |
| | Fibonacci Series Recursive | -- | $y = 1.7836e^{0.4945x}$ | $O(2^n)$ | $O(2^n)$ |
| 1.3 | Matrix Addition | -- | $y = 2.00x^2 + 2.00x - 0.00$ | $O(n^2)$ | $O(n^2)$ |
| | Matrix Multiplication | -- | $Y = 1x^3 + 8x^2 + 21x + 19$ | $O(n^3)$ | $O(n^3)$ |
| 1.4 (1) | Iterative Search | Best Case | $y = 1$ | O (1) | O (1) |
| | | Avg. Case | $y = 1.28x - 1$ | O(n) | O(n) |
| | | Worst Case | $y = 2x - 1$ | O(n) | O(n) |
| 1.4 (2) | Binary Search | Best Case | $y = 3$ | O (1) | O (1) |
| | | Avg. Case | $y = 4.49\ln(x) - 1.46$ | O (log n) | O (log n) |
| | | Worst Case | $y = 37.91\ln(x) - 98.84$ | O (log n) | O (log n) |
| 1.5 | Insertion Sort | Best Case | $y = 3x - 3$ ` | O(n) | O(n) |
| | | Avg. Case | $y = 0.75x^2 - 17.05x + 185.40$ | $O(n^2)$ | $O(n^2)$ |
| | | Worst Case | $y = 1.5x^2 + 1.5x - 3$ | $O(n^2)$ | $O(n^2)$ |