**NAVRACHANA UNIVERSITY**

*a UGC recognized University*

**School of engineering and technology**

**Computer science and engineering department**

SY B Tech CSE - IV semester

Academic Year: 2022-23

**Course name and code**

Design and Analysis of Algorithms (CS313)

# Laboratory Manual

**Student name and Enrolment Number**

SAHIL PANDYA (21124097)

**Course In-charge**

Prof. KRITI JAISWAL

## Table of Contents:

## Experiment No. 1

**Aim:** Write a program to implement array addition, array multiplication and matrix addition**.**

## Introduction:

Array addition: Array addition involves adding corresponding elements from two arrays to create a new array with the same dimensions.

Array multiplication: In array multiplication, each element of an array is multiplied by a scalar value to create a new array. The resulting array has the same dimensions as the original array, and each element in the new array is obtained by multiplying the scalar value with the corresponding element in the original array.

Matrix addition: Matrix addition is the process of adding corresponding elements of two matrices having the same dimensions to obtain a new matrix, where each element is the sum of the corresponding elements from the original matrices.

## Example:

ARRAY ADDITION -

A = {1,2,3} and B = {3,2,1} then A + B = {4,4,4}

ARRAY MULTIPLICATION

MATRIX ADDITION -

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} + \begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1+9 & 2+8 & 3+7 \\ 4+6 & 5+5 & 6+4 \\ 7+3 & 8+2 & 9+1 \end{bmatrix}$$

$$= \begin{bmatrix} 10 & 10 & 10 \\ 10 & 10 & 10 \\ 10 & 10 & 10 \end{bmatrix}$$

## Algorithm:

Array Addition Algorithm:

1.Define two arrays of equal size, A and B.

2.Create a new array C that has the same size as A and B.

3.For each index i from 0 to the length of A or B, follow these steps:

a. Calculate the sum of A[i] and B[i].

b. Assign the resulting sum to C[i].

4.C now holds the element-wise addition of A and B.

Array Multiplication Algorithm:

1.Define an array A of size n and a scalar value s.

2.Create a new array B that has the same size as A.

3.For each index i from 0 to the length of A, follow these steps:

a. Multiply the value of s by the element A[i].

b. Assign the resulting product to B[i].

4.B now holds the element-wise product of A and the scalar s.

Matrix Addition Algorithm:

1.Define two matrices A and B of size n x m.

2.Create a new matrix C of size n x m.

3.For each row i and column j of matrices A and B, follow these steps:

a. Calculate the sum of A[i][j] and B[i][j].

b. Assign the resulting sum to C[i][j].

4.C now holds the element-wise addition of matrices A and B.

## Source code:

Array Addition

```
import time

st=time.time()
```

```
arr1=[1,2,3,4]
sum=0
for i in arr1:
    sum=sum+i
print("Sum of array is: ",sum)
et=time.time()
elapsed_time = et - st
print('Execution time of array additon:', elapsed_time, 'seconds')
```

<u>Array Multiplication</u>

```
import time
st=time.time()
arr1=[1,2,3,4]
mul=1
for i in arr1:
    mul=mul*i
print("Multiplication of array is: ",mul)
et=time.time()
elapsed_time = et - st
print('Execution time of array additon:', elapsed_time, 'seconds')
```

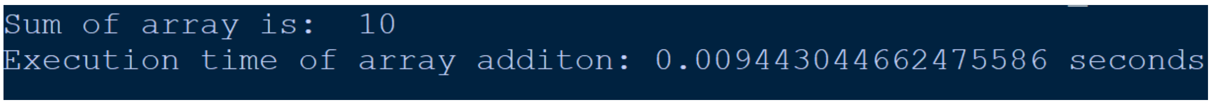
<u>Matrix Addition</u>

```
import time
a=[[1,2,3],[4,5,6],[7,8,9]]
b=[[1,2,3],[4,5,6],[7,8,9]]
```

```
c=[[0,0,0],[0,0,0],[0,0,0]]

n=int(input("Enter order of matrix: "))

st1 = time.time()


for i in range(n):

    for j in range(n):

        c[i][j]=a[i][j]+b[i][j]

print("Matrix A:\n",a)

print("Matrix B:\n",b)

print("Matrix Addition is:\n",c)

et1 = time.time()

elapsed_time1 = et1 - st1

print('Execution time of matrix addition:', elapsed_time1, 'seconds')
```

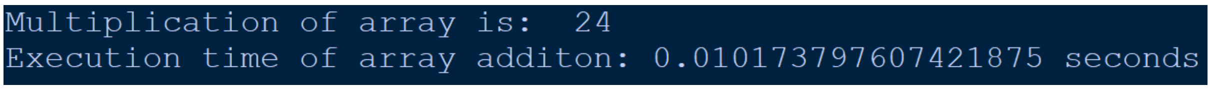
## Output:

<u>Array Addition</u>



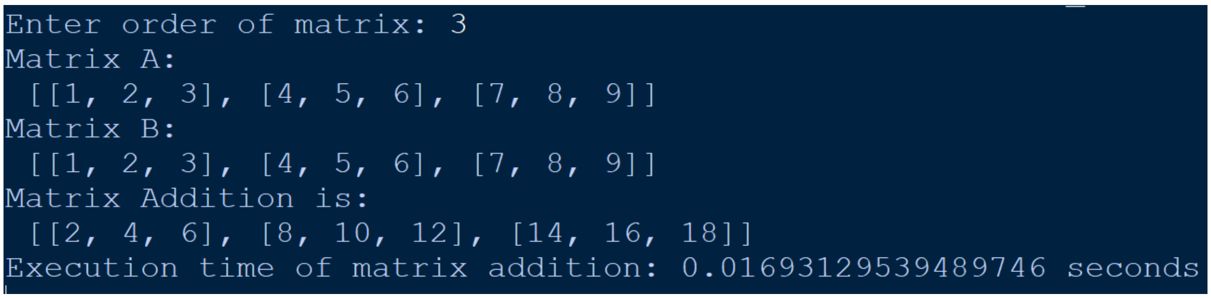
```
Sum of array is:  10
Execution time of array additon: 0.009443044662475586 seconds
```

Array Multiplication



```
Multiplication of array is:  24
Execution time of array additon: 0.010173797607421875 seconds
```

Matrix Addition



```
Enter order of matrix: 3
Matrix A:
 [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Matrix B:
 [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Matrix Addition is:
 [[2, 4, 6], [8, 10, 12], [14, 16, 18]]
Execution time of matrix addition: 0.01693129539489746 seconds
```

## Analysis / Observations:

Array addition:
Worst-case time complexity: O(n)
Best-case time complexity: O(n)
Average-case time complexity: O(n)

Array multiplication:
Worst-case time complexity: O(n^3)
Best-case time complexity: O(n^2)
Average-case time complexity: O(n^3)

Matrix addition:
Worst-case time complexity: O(n^2)
Best-case time complexity: O(n^2)
Average-case time complexity: O(n^2)

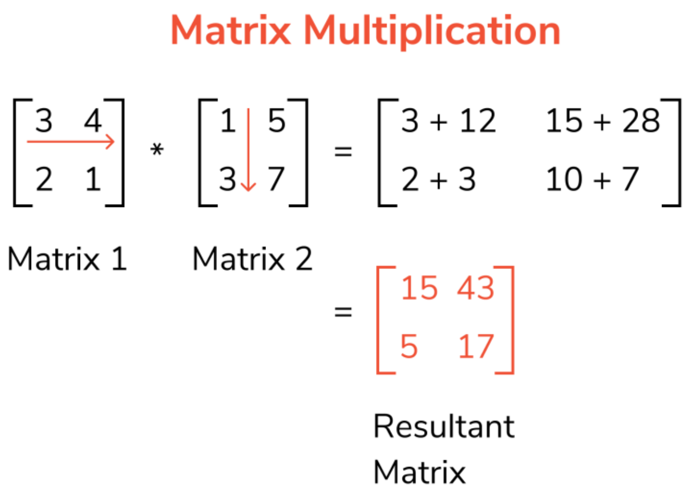
## Experiment No. 2

**Aim:** Write a program to perform matrix multiplication of two matrices of order 'n'. Perform the analysis of implementation for worst case.

## Introduction:

To perform matrix multiplication in C, nested loops are used. The outer loop handles the rows of the first matrix, while the inner loop deals with the columns of the second matrix. During each iteration of the loops, the dot product of the corresponding row and column is computed and saved in the resulting matrix.

## Example:


Matrix Multiplication
$\begin{bmatrix}3 & 4\\2 & 1\end{bmatrix} * \begin{bmatrix}1 & 5\\3 & 7\end{bmatrix} = \begin{bmatrix}3+12 & 15+28\\2+3 & 10+7\end{bmatrix}$
Matrix 1
Matrix 2
$= \begin{bmatrix}15 & 43\\5 & 17\end{bmatrix}$
Resultant Matrix

## Algorithm:

<u>Matrix Multiplication Algorithm:</u>

a. Define two matrices, A and B, where A has n rows and m columns and B has m rows and p columns.

b. Create a new matrix C with n rows and p columns.

c. For each row i in matrix A and each column j in matrix B, follow these steps:

1. Calculate the dot product of the ith row of A and the jth column of B.
2. Assign the resulting dot product to C[i][j].

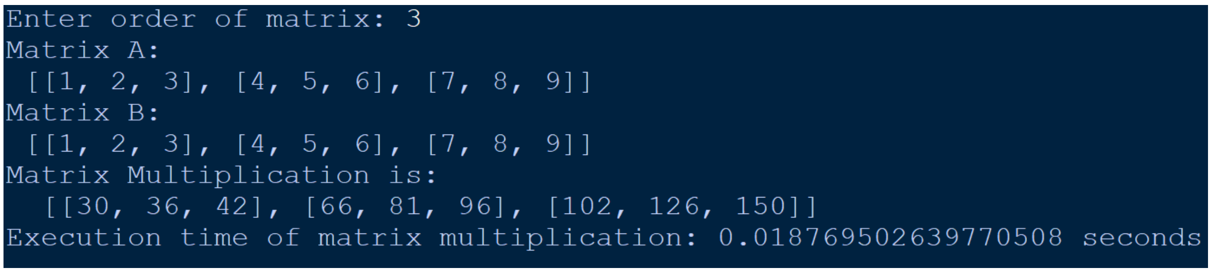
d. Matrix C now holds the product of A and B.

## Source code:

```
import time

a=[[1,2,3],[4,5,6],[7,8,9]]

b=[[1,2,3],[4,5,6],[7,8,9]]
```

```
c=[[0,0,0],[0,0,0],[0,0,0]]

n=int(input("Enter order of matrix: "))

st2 = time.time()

for i in range(n):

    for j in range(n):

        for k in range(n):

            c[i][j]=a[i][k]*b[k][j]+c[i][j]

print("Matrix A:\n",a)

print("Matrix B:\n",b)

print("Matrix Multiplication is:\n ",c)

et2 = time.time()

elapsed_time2 = et2 - st2

print('Execution time of matrix multiplication:', elapsed_time2, 'seconds')
```

## Output:



```
Enter order of matrix: 3
Matrix A:
 [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Matrix B:
 [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Matrix Multiplication is:
  [[30, 36, 42], [66, 81, 96], [102, 126, 150]]
Execution time of matrix multiplication: 0.018769502639770508 seconds
```

## Analysis / Observations:

(n) = O(n^3), As two for loop are required

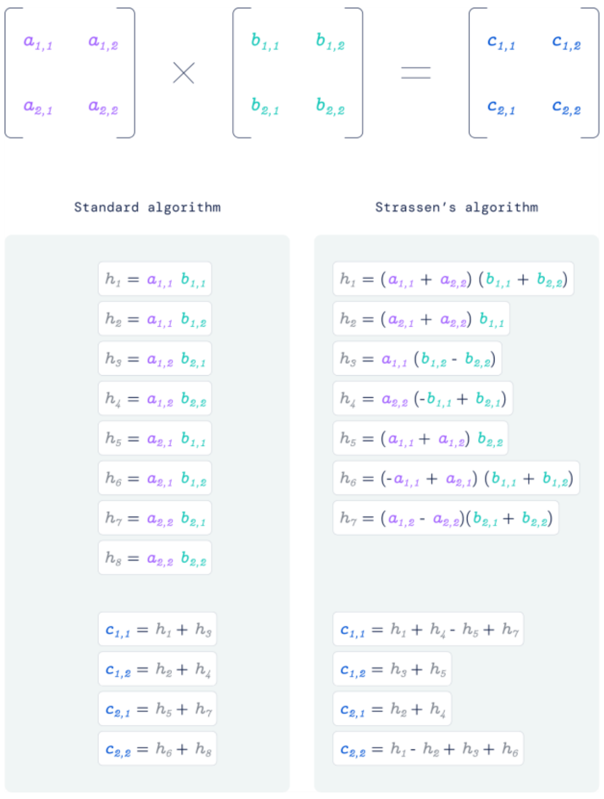
## Experiment No. 3

**Aim:** Write a program to implement Strassen's method to perform matrix multiplication. Perform the analysis of implementation for worst case.

## Introduction:

The Strassen algorithm is a faster way to multiply matrices than the standard multiplication algorithm, particularly for large matrices. It also has a better asymptotic complexity.

## Example:



$$\begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \times \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} = \begin{bmatrix} c_{1,1} & c_{1,2} \\ c_{2,1} & c_{2,2} \end{bmatrix}$$

Standard algorithm

$h_1 = a_{1,1}\ b_{1,1}$

$h_2 = a_{1,1}\ b_{1,2}$

$h_3 = a_{1,2}\ b_{2,1}$

$h_4 = a_{1,2}\ b_{2,2}$

$h_5 = a_{2,1}\ b_{1,1}$

$h_6 = a_{2,1}\ b_{1,2}$

$h_7 = a_{2,2}\ b_{2,1}$

$h_8 = a_{2,2}\ b_{2,2}$

$c_{1,1} = h_1 + h_3$

$c_{1,2} = h_2 + h_4$

$c_{2,1} = h_5 + h_7$

$c_{2,2} = h_6 + h_8$

Strassen's algorithm

$h_1 = (a_{1,1} + a_{2,2})\ (b_{1,1} + b_{2,2})$

$h_2 = (a_{2,1} + a_{2,2})\ b_{1,1}$

$h_3 = a_{1,1}\ (b_{1,2} - b_{2,2})$

$h_4 = a_{2,2}\ (-b_{1,1} + b_{2,1})$

$h_5 = (a_{1,1} + a_{1,2})\ b_{2,2}$

$h_6 = (-a_{1,1} + a_{2,1})\ (b_{1,1} + b_{1,2})$

$h_7 = (a_{1,2} - a_{2,2})(b_{2,1} + b_{2,2})$

$c_{1,1} = h_1 + h_4 - h_5 + h_7$

$c_{1,2} = h_3 + h_5$

$c_{2,1} = h_2 + h_4$

$c_{2,2} = h_1 - h_2 + h_3 + h_6$

## Algorithm:

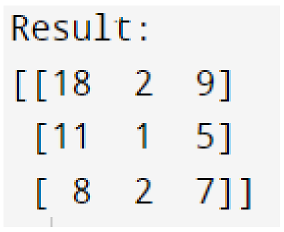
1. Divide the input matrices A and B into four equally sized submatrices, each of size n/2 x n/2.
2. Define 10 new matrices, S1 through S10, each of size n/2 x n/2.
3. Compute the 7 products required for Strassen's Method:
    a. S1 = (A11 + A22) x (B11 + B22)
    b. S2 = (A21 + A22) x B11
    c. S3 = A11 x (B12 - B22)
    d. S4 = A22 x (B21 - B11)
    e. S5 = (A11 + A12) x B22
    f. S6 = (A21 - A11) x (B11 + B12)
    g. S7 = (A12 - A22) x (B21 + B22)
4. Compute the resulting submatrices C11, C12, C21, and C22 as follows:
    a. C11 = S1 + S4 - S5 + S7
    b. C12 = S3 + S5
    c. C21 = S2 + S4
    d. C22 = S1 - S2 + S3 + S6

5. Combine the resulting submatrices into the resulting matrix C.

## Source code:

```
import numpy as np

def strassen_algorithm(x, y):

    if x.size == 1 or y.size == 1:

        return x * y

    n = x.shape[0]

    if n % 2 == 1:

        x = np.pad(x, (0, 1), mode='constant')

        y = np.pad(y, (0, 1), mode='constant')

    m = int(np.ceil(n / 2))

    a = x[: m, : m]

    b = x[: m, m:]

    c = x[m:, : m]

    d = x[m:, m:]

    e = y[: m, : m]

    f = y[: m, m:]

    g = y[m:, : m]

    h = y[m:, m:]

    p1 = strassen_algorithm(a, f - h)

    p2 = strassen_algorithm(a + b, h)

    p3 = strassen_algorithm(c + d, e)

    p4 = strassen_algorithm(d, g - e)

    p5 = strassen_algorithm(a + d, e + h)

    p6 = strassen_algorithm(b - d, g + h)
    p7 = strassen_algorithm(a - c, e + f)
    result = np.zeros((2 * m, 2 * m), dtype=np.int32)
    result[: m, : m] = p5 + p4 - p2 + p6
    result[: m, m:] = p1 + p2
    result[m:, : m] = p3 + p4
    result[m:, m:] = p1 + p5 - p3 - p7
    return result[: n, : n]
if __name__ == "__main__":
    x = np.array([[1, 2, 3], [0, 1, 2], [1, 2, 1]])
    y = np.array([[1, 0, 0], [1, 1, 3], [5, 0, 1]])
    print('Result: ')
    print(strassen_algorithm(x, y))
```

**Output:**



```
Result:
[[18  2  9]
 [11  1  5]
 [ 8  2  7]]
```

## Analysis / Observations:

By using the divide and conquer technique the overall complexity for the multiplication of two matrices has been reduced. This happens by decreasing the total number of multiplications performed at the expense of a slight increase in the number of additions.

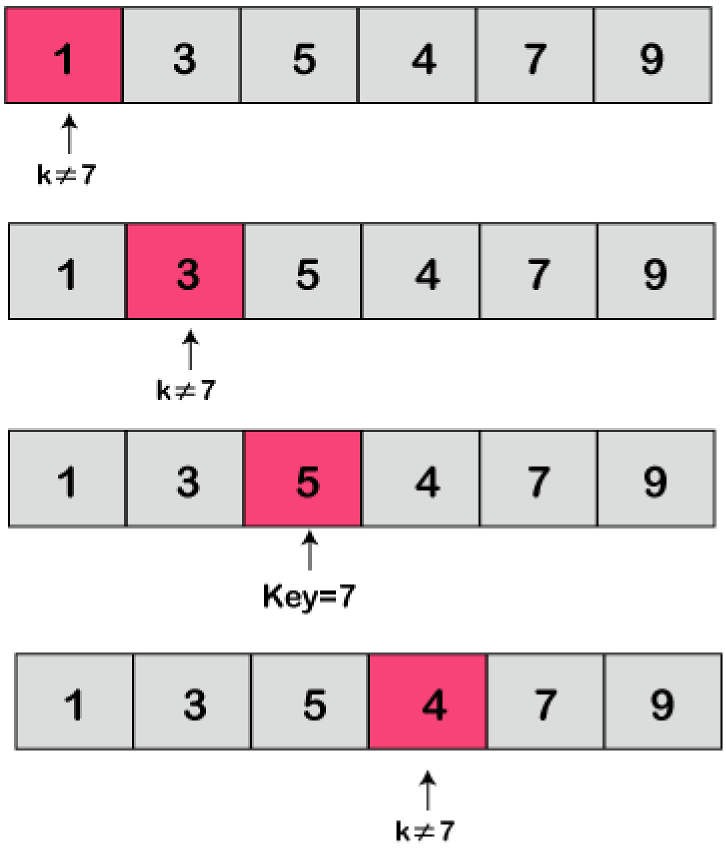
## Experiment No. 4

**Aim:** Write a program to implement linear search algorithm. Perform the analysis of implementation for worst case.

### Introduction:

Linear search is an uncomplicated searching algorithm used to find a specific value in a list or array by checking each element one by one until either a match is found or the entire list has been searched.

### Example:



### Algorithm:

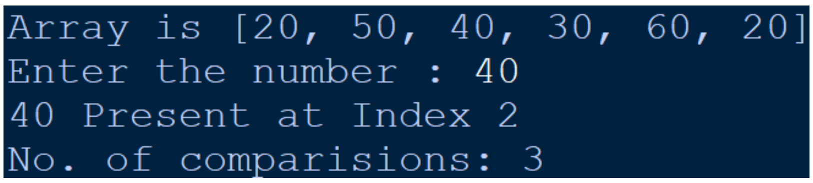
1. Begin at the first element in a list.
2. Check if the current element matches the target element.
3. If the current element matches the target element, return the index of the current element.

4. If the end of the list is reached and no match is found, return -1 to indicate that the target element is not in the list.
5. If the current element does not match the target element, move to the next element and repeat from step 2.

## Source code:

```
def linear_search(arr, x):
   global compare
   compare=0
   for i in range(len(arr)):
      compare=compare+1
      if arr[i] == x:
         return i
arr=[20,50,40,30,60,20]
print("Array is",arr)
num=int(input("Enter the number : "))
print(num,"Present at Index",linear_search(arr,num))
print("No. of comparisions:",compare)
```

## Output:



```
Array is [20, 50, 40, 30, 60, 20]
Enter the number : 40
40 Present at Index 2
No. of comparisions: 3
```

## Analysis / Observations:

Time complexity: O(n)

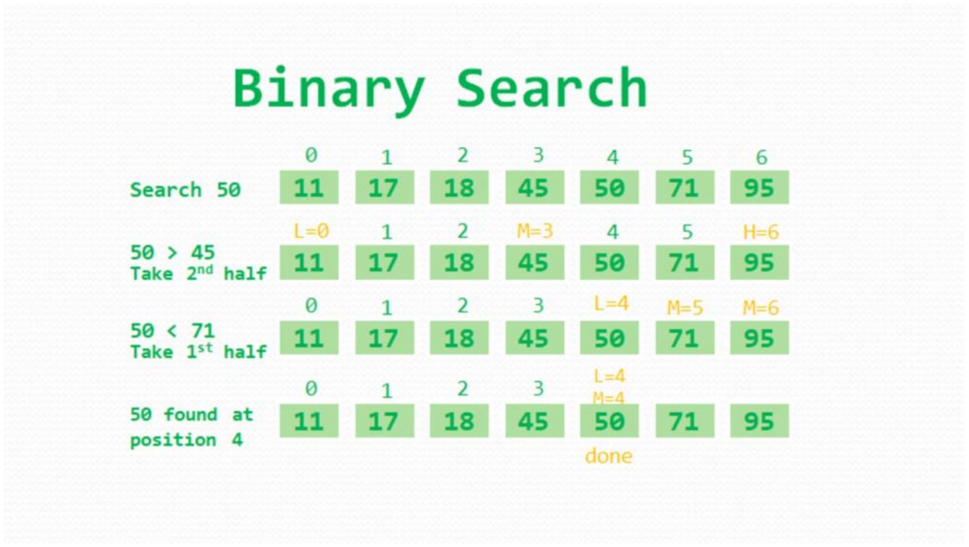
## Experiment No. 5

**Aim:** Write a program to implement binary search algorithm. Perform the analysis of implementation for worst case.

### Introduction:

Binary search is a searching algorithm that quickly and effectively finds the target value by dividing the search space in half repeatedly until the value is found or the search space is empty. It is often used with sorted lists or arrays.

### Example:



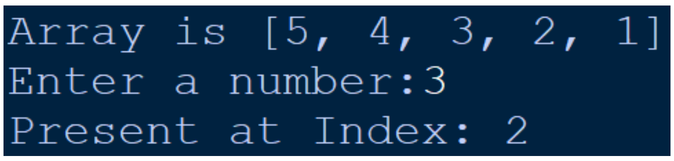
### Algorithm:

1. Set the search interval to the entire array by initializing low and high indices.
2. While the search interval is not empty:
    a. Find the middle index of the search interval.
    b. If the value at the middle index matches the target value, return its index.
    c. If the value at the middle index is less than the target value, set the new search interval to the right half of the current search interval.
    d. If the value at the middle index is greater than the target value, set the new search interval to the left half of the current search interval.
3. If the target value is not found in the array, return -1 to indicate failure.

### Source code:

```
def binary_search(arr, low, high, x):

   if high >= low:

      mid = (high + low) // 2

      if arr[mid] == x:

         return mid

      elif arr[mid] > x:

         return binary_search(arr, low, mid - 1, x)

      else:

         return binary_search(arr, mid + 1, high, x)

   else:

      return -1


arr=[5,4,3,2,1]

print("Array is",arr)

num=int(input("Enter a number:"))

print("Present at Index:",binary_search(arr,0,len(arr),num))
```

## Output:



```
Array is [5, 4, 3, 2, 1]
Enter a number:3
Present at Index: 2
```

## Analysis / Observations:

Worst-case time complexity: O(log n)
Best-case time complexity: O(1)
Average-case time complexity: O(log n)

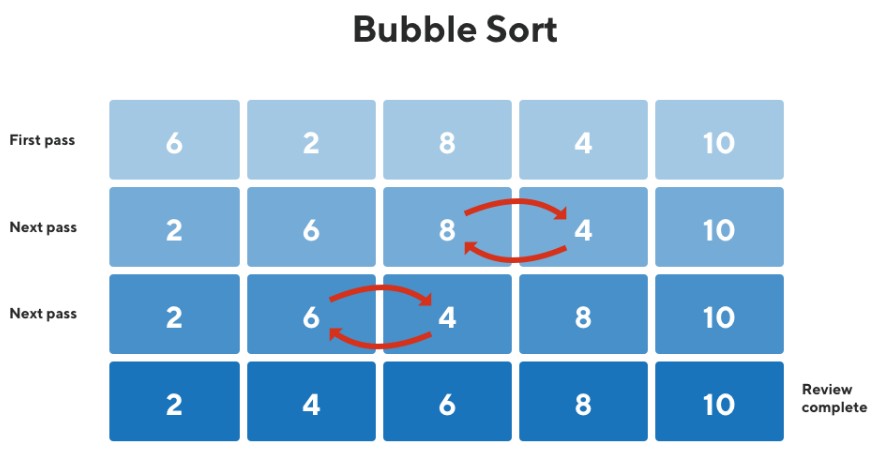
## Experiment No. 6

**Aim:** Write a program to implement Bubble sort algorithm. Perform the analysis of implementation for best case, average case and worst case.

## Introduction:

Bubble sort is an uncomplicated and easy-to-understand sorting algorithm that functions by swapping adjacent elements if they are in the incorrect order. This causes the largest elements to move towards the end of the list during each iteration. Although it has a worst-case time complexity of O(n^2), it is effective for small input sizes or partially sorted lists.

## Example:



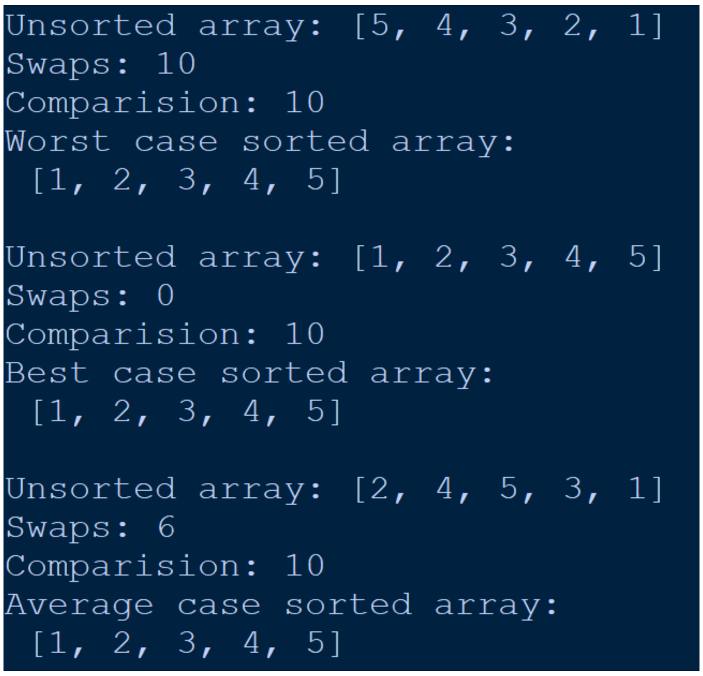
## Algorithm:

1. Start at the beginning of the array.
2. Loop through each element of the array.
3. Compare the current element to the next element.
4. If the current element is greater than the next element, swap them.
5. Continue looping through the array until no more swaps are needed.
6. Return the sorted array.

## Source code:

```
def bubbleSort(a):
    swap=0
    compare=0
    for i in range(len(a)):
        for j in range(len(a)-i-1):#0,1,2,3
            compare=compare+1
            if (a[j]>a[j+1]):
                (a[j],a[j+1])=(a[j+1],a[j])
                swap=swap+1
    print("Swaps:",swap)
    print("Comparision:",compare)
a=[5,4,3,2,1]
print("Unsorted array:",a)
bubbleSort(a)
print("Worst case sorted array:\n",a)
print()
b=[1,2,3,4,5]
print("Unsorted array:",b)
bubbleSort(b)
print("Best case sorted array:\n",b)
print()
c=[2,4,5,3,1]
print("Unsorted array:",c)
bubbleSort(c)
print("Average case sorted array:\n",c)
```

**Output:**



```
Unsorted array: [5, 4, 3, 2, 1]
Swaps: 10
Comparision: 10
Worst case sorted array:
 [1, 2, 3, 4, 5]

Unsorted array: [1, 2, 3, 4, 5]
Swaps: 0
Comparision: 10
Best case sorted array:
 [1, 2, 3, 4, 5]

Unsorted array: [2, 4, 5, 3, 1]
Swaps: 6
Comparision: 10
Average case sorted array:
 [1, 2, 3, 4, 5]
```

## Analysis / Observations:

1. Best Case Complexity: O(n) If the array is already sorted, then there is no need for sorting.
2. Average Case Complexity: O(n^2) It occurs when the elements of the array are in jumbled order.
3. Worst Case Complexity: O(n^2) If we want to sort in ascending order and the array is in descending order then the worst case occurs.

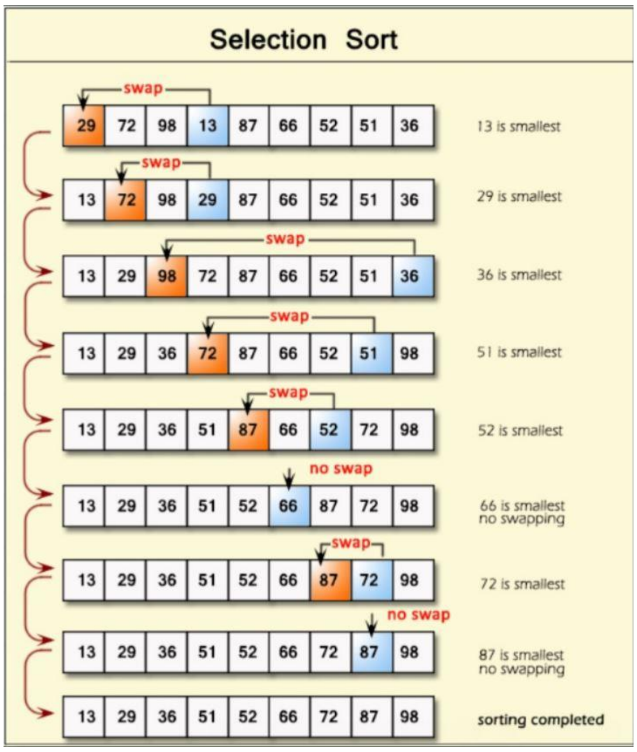
## Experiment No. 7

**Aim:** Write a program to implement Selection sort algorithm. Perform the analysis of implementation for best case, average case and worst case.

### Introduction:

Selection sort is a sorting algorithm that works by repeatedly searching for the smallest element in the unsorted part of an array and moving it to the beginning of the sorted part of the array. It has a time complexity of O(n^2) and may not be efficient for large arrays, but it is simple to comprehend and execute.

### Example:



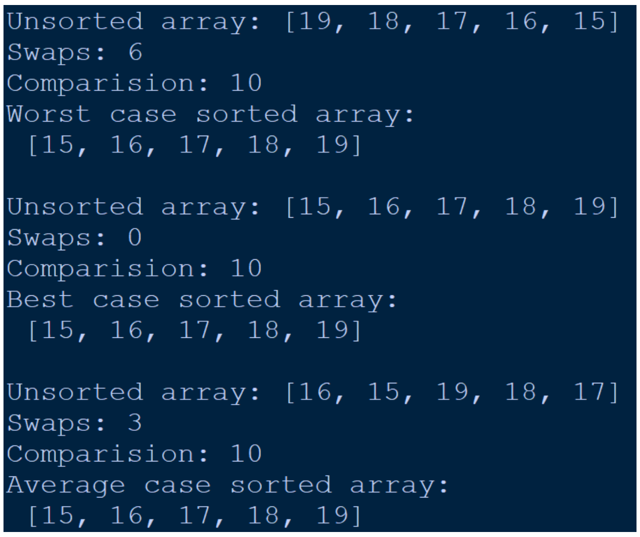
## Algorithm:

1. Start at the beginning of the array.
2. Loop through each element of the array.
3. Find the minimum element in the unsorted portion of the array.
4. Swap the minimum element with the first element of the unsorted portion of the array.
5. Repeat steps 3-4 until the array is sorted.
6. Return the sorted array.

## Source code:

```
def selectionSort(a):
  swap=0
  compare=0
  for i in range(len(a)):
    ind=i
    for j in range(i+1,len(a)):
            compare=compare+1
            if (a[j]<a[ind]):
               swap=swap+1
               ind=j
        (a[i],a[ind])=(a[ind],a[i])

    print("Swaps:",swap)
    print("Comparision:",compare)

a=[19,18,17,16,15]
print("Unsorted array:",a)
selectionSort(a)
print("Worst case sorted array:\n",a)
print()

b=[15,16,17,18,19]
print("Unsorted array:",b)
selectionSort(b)
print("Best case sorted array:\n",b)
print()

c=[16,15,19,18,17]
print("Unsorted array:",c)
selectionSort(c)
print("Average case sorted array:\n",c)
```

**Output:**



```
Unsorted array: [19, 18, 17, 16, 15]
Swaps: 6
Comparision: 10
Worst case sorted array:
 [15, 16, 17, 18, 19]

Unsorted array: [15, 16, 17, 18, 19]
Swaps: 0
Comparision: 10
Best case sorted array:
 [15, 16, 17, 18, 19]

Unsorted array: [16, 15, 19, 18, 17]
Swaps: 3
Comparision: 10
Average case sorted array:
 [15, 16, 17, 18, 19]
```

## Analysis / Observations:

1. Best Case Complexity: O(n^2) If the array is already sorted, then there is no need for sorting.
2. Average Case Complexity: O(n^2) It occurs when the elements of the array are in jumbled order.
3. Worst Case Complexity: O(n^2) If we want to sort in ascending order and the array is in descending order then the worst case occurs

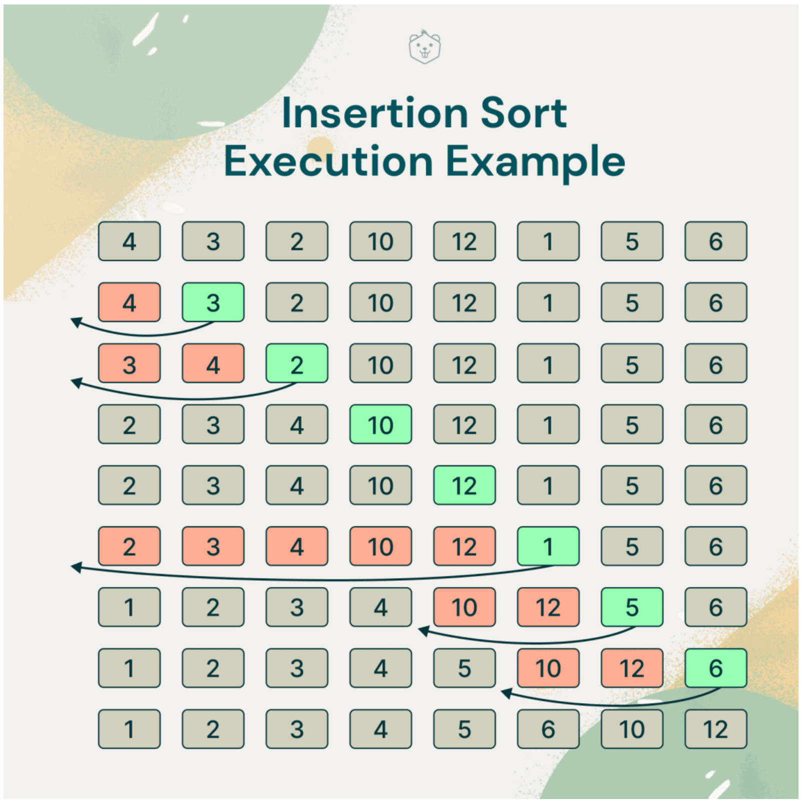
## Experiment No. 8

**Aim:** Write a program to implement Insertion sort algorithm. Perform the analysis of implementation for best case, average case and worst case.

## Introduction:

Insertion sort is a simple sorting algorithm that builds the final sorted array one element at a time. It works by repeatedly taking an unsorted element and inserting it into the correct position in the sorted part of the array. It has a time complexity of O(n^2), but it is efficient for small data sets or partially sorted arrays.

## Example:



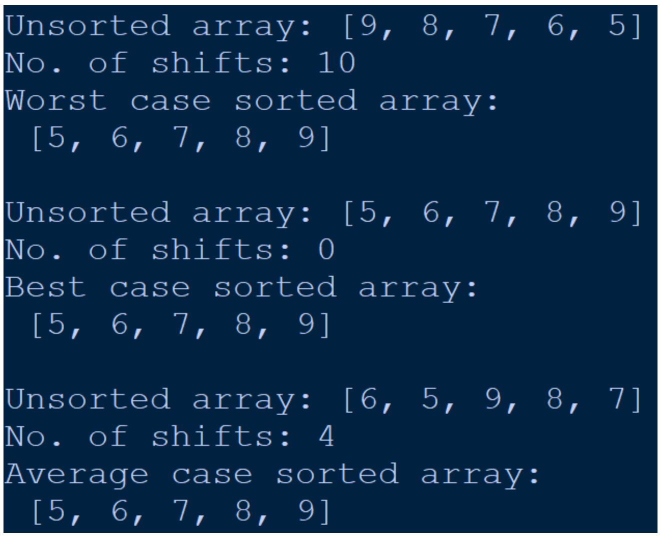
## Algorithm:

1. Iterate over the array from the second element to the last element.
2. For each element, compare it with the elements before it and find the correct position to insert it in the sorted part of the array.
3. Shift the elements that are greater than the current element to the right by one position.
4. Insert the current element into the correct position in the sorted part of the array.
5. Continue the iteration until the entire array is sorted

## Source code:

```
def insertionSort(a):
    shift=0
    for i in range(1,len(a)):


        temp=a[i]
        j=i-1
        while(j>=0 and temp<a[j]):
            shift=shift+1
            a[j+1]=a[j]
            j=j-1
        a[j+1]=temp

    print("No. of shifts:",shift)

a=[9,8,7,6,5]
print("Unsorted array:",a)
insertionSort(a)
print("Worst case sorted array:\n",a)
print()

b=[5,6,7,8,9]
print("Unsorted array:",b)
insertionSort(b)
print("Best case sorted array:\n",b)
print()

c=[6,5,9,8,7]
print("Unsorted array:",c)
insertionSort(c)
print("Average case sorted array:\n",c)
```

## Output:



```
Unsorted array: [9, 8, 7, 6, 5]
No. of shifts: 10
Worst case sorted array:
 [5, 6, 7, 8, 9]

Unsorted array: [5, 6, 7, 8, 9]
No. of shifts: 0
Best case sorted array:
 [5, 6, 7, 8, 9]

Unsorted array: [6, 5, 9, 8, 7]
No. of shifts: 4
Average case sorted array:
 [5, 6, 7, 8, 9]
```

## Analysis / Observations:

1. The worst-case time complexity of Insertion sort is O(N^2). Each element has to be compared with each of the other elements so, for every nth element, (n-1) number of comparisons are made.
2. The average case time complexity of Insertion sort is O(N^2). When the array is already sorted, the outer loop runs for n number of times whereas the inner loop does not run at all. So, there are only n number of comparisons. Thus, complexity is linear.
3. The time complexity of the best case is O(N).

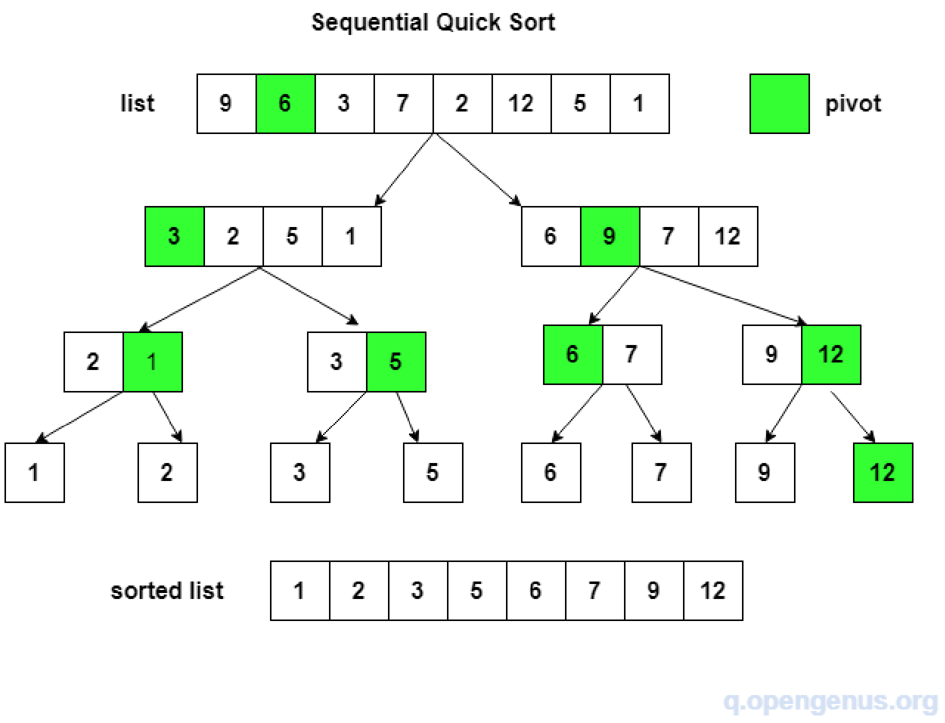
## Experiment No. 9

**Aim:** Write a program to implement Quick sort algorithm. Perform the analysis of implementation for best case, average case and worst case.

## Introduction:

Quick Sort is an effective and widely used sorting algorithm that divides and conquers to sort an array or list. It chooses a pivot element, divides the remaining elements into two sub-arrays based on whether they are less than or greater than the pivot, and sorts these sub-arrays recursively.

## Example:



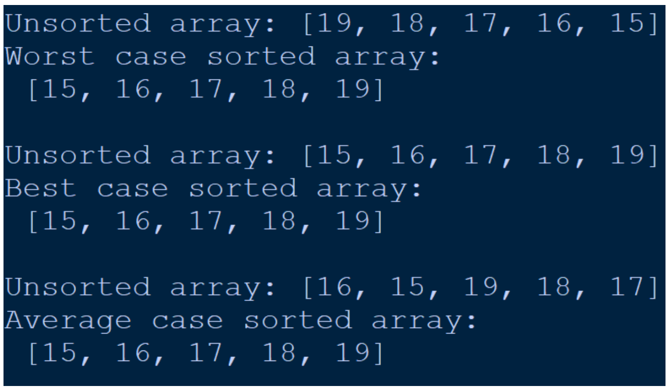
## Algorithm:

1. Choose a pivot element from the array. The pivot element can be chosen in various ways, such as selecting the first element, the last element, or a random element.
2. Partition the array around the pivot element by rearranging the elements such that all elements smaller than the pivot are on the left, and all elements larger than the pivot are on the right.
3. Recursively apply steps 1-2 to the sub-arrays created by the partition until the sub-arrays have only one element, which is already sorted.
4. Concatenate the sorted sub-arrays to get the final sorted array.
5. Return the sorted array.

## Source code:

```
def quickSort(a):
    if len(a)<=1:
        return a
    pivot=a[-1]
    left,right=[],[]
    for i in range(len(a)-1):
        if (a[i]<pivot):
            left.append(a[i])
        else:
            right.append(a[i])
    return quickSort(left)+[pivot]+quickSort(right)


a=[19,18,17,16,15]
print("Unsorted array:",a)
print("Worst case sorted array:\n",quickSort(a))
print()
b=[15,16,17,18,19]
print("Unsorted array:",b)
print("Best case sorted array:\n",quickSort(b))
print()
c=[16,15,19,18,17]
print("Unsorted array:",c)
print("Average case sorted array:\n",quickSort(c))
```

**Output:**



```
Unsorted array: [19, 18, 17, 16, 15]
Worst case sorted array:
 [15, 16, 17, 18, 19]

Unsorted array: [15, 16, 17, 18, 19]
Best case sorted array:
 [15, 16, 17, 18, 19]

Unsorted array: [16, 15, 19, 18, 17]
Average case sorted array:
 [15, 16, 17, 18, 19]
```

## Analysis / Observations:

The time complexity of Quick Sort in the average case is O(n*log n), where n is the number of elements in the array. However, in the worst case, when the input array is already sorted or almost sorted, the time complexity of Quick Sort becomes O(n^2).

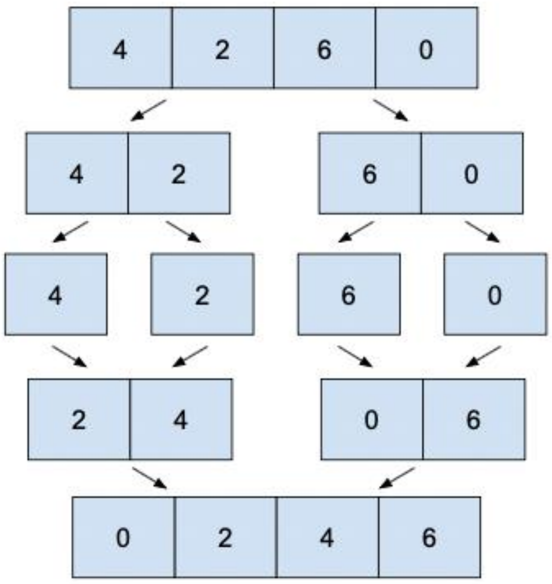
## Experiment No. 10

**Aim:** Write a program to implement Merge sort algorithm. Perform the analysis of implementation for best case, average case and worst case.

### Introduction:

Merge sort is a sorting algorithm that follows the divide-and-conquer approach. It starts by dividing the input array into two halves, sorting each of these halves recursively, and then merging them back together in a sorted manner. Because it has a time complexity of O(n log n), Merge sort is an efficient sorting algorithm for large arrays or lists.

### Example:



## Algorithm:

1. Divide the input array into two halves: left and right.
2. Recursively sort the left half using Merge Sort.
3. Recursively sort the right half using Merge Sort.
4. Merge the two sorted halves back together by repeatedly comparing the smallest unmerged element in each half, and copying the smaller one to the temporary array.
5. Copy the remaining elements of the left and right halves to the temporary array.
6. Copy the merged elements from the temporary array back to the original array.

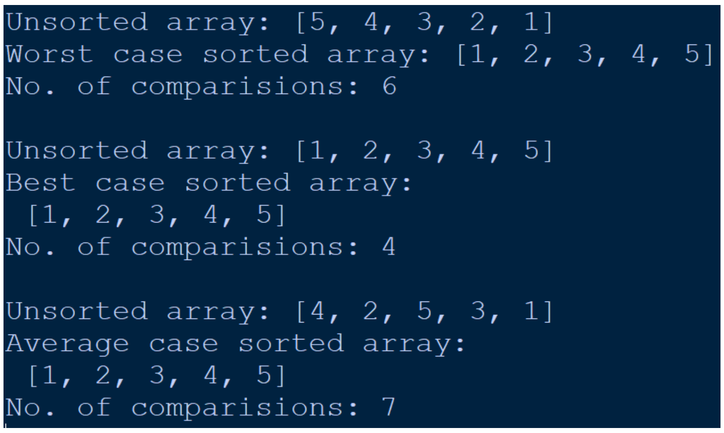
## Source code:

```
def mergeSort(a):

   global compare

   compare=0

   if (len(a)>1):

      mid=len(a)//2

      L=a[:mid]

      R=a[mid:]

      mergeSort(L)
```

```
    mergeSort(R)
    i=j=k=0
    while i<len(L) and j<len(R):
      if L[i]<=R[j]:
        a[k]=L[i]
        i+=1
      else:
        a[k]=R[j]
        j+=1
      k+=1
      compare=compare+1
    while i<len(L):
      a[k]=L[i]
      i+=1
      k+=1
    while j<len(R):
      a[k]=R[j]
      j+=1
      k+=1
a=[5,4,3,2,1]
print("Unsorted array:",a)
mergeSort(a)
print("Worst case sorted array:",a)
print("No. of comparisions:",compare)
print()
b=[1,2,3,4,5]
```

```
print("Unsorted array:",b)

mergeSort(b)

print("Best case sorted array:\n",b)

print("No. of comparisions:",compare)

print()

c=[4,2,5,3,1]

print("Unsorted array:",c)

mergeSort(c)

print("Average case sorted array:\n",c)

print("No. of comparisions:",compare)
```

## Output:



```
Unsorted array: [5, 4, 3, 2, 1]
Worst case sorted array: [1, 2, 3, 4, 5]
No. of comparisions: 6

Unsorted array: [1, 2, 3, 4, 5]
Best case sorted array:
 [1, 2, 3, 4, 5]
No. of comparisions: 4

Unsorted array: [4, 2, 5, 3, 1]
Average case sorted array:
 [1, 2, 3, 4, 5]
No. of comparisions: 7
```

## Analysis / Observations:

Worst-case time complexity: O(n log n)
Best-case time complexity: Ω(n log n)
Average-case time complexity: Θ(n log n)

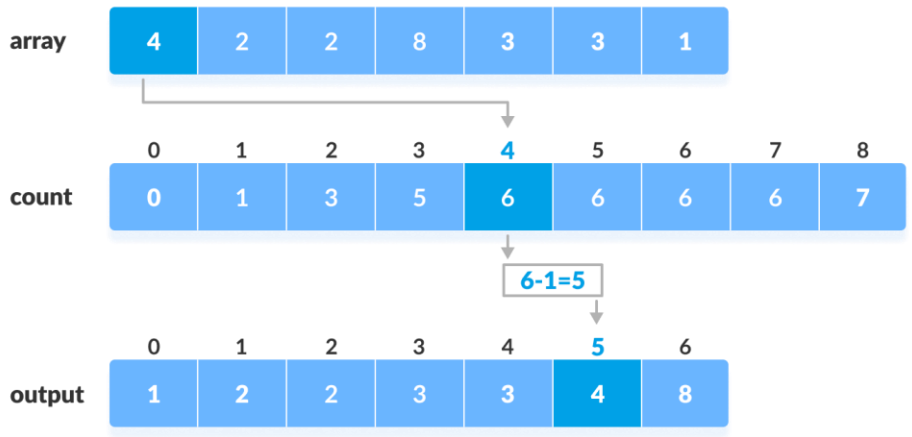
## Experiment No. 11

**Aim:** Write a program to implement Counting sort algorithm. Perform the analysis of implementation for best case, average case and worst case.

## Introduction:

Counting Sort is a sorting algorithm that works by counting how often each element appears in the input array and then using this information to sort the elements into the appropriate position in an output array. It has a time complexity of O(n+k), where n is the input array size and k is the range of input elements. Counting Sort is frequently used when the range of input elements is small compared to the size of the input array.

## Example:



## Algorithm:

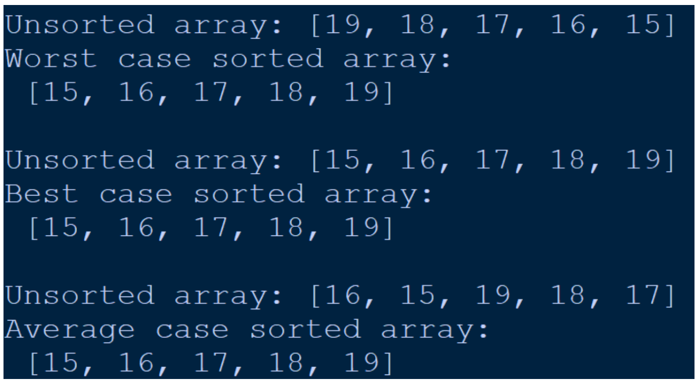
1. Find the maximum element in the array to be sorted.
2. Create an empty array of counts with a length equal to the maximum element plus one.
3. Iterate through the array to be sorted and count the occurrences of each element by incrementing the corresponding count in the counts array.
4. Calculate the cumulative sum of the counts in the counts array.
5. Create an empty output array of the same length as the input array.
6. Iterate through the input array in reverse order, and for each element, place it in its correct position in the output array based on its count and the cumulative sum of the counts.
7. Return the sorted output array.

## Source code:

```
def countingSort(array):
    size = len(array)
    output = [0] * size
    count = [0] * (max(array)+1)
    for i in range(0, size):
        count[array[i]] += 1
    for i in range(1, len(count)):
        count[i] += count[i - 1]
    i = size-1
    while i >= 0:
        output[count[array[i]] - 1] = array[i]
        count[array[i]] -= 1
        i -= 1
    for i in range(0, size):
        array[i] = output[i]
a=[19,18,17,16,15]
print("Unsorted array:",a)
countingSort(a)
print("Worst case sorted array:\n",a)
print()
b=[15,16,17,18,19]
print("Unsorted array:",b)
countingSort(b)
print("Best case sorted array:\n",b)
print()
```

```
c=[16,15,19,18,17]

print("Unsorted array:",c)

countingSort(c)

print("Average case sorted array:\n",c)
```

## Output:



```
Unsorted array: [19, 18, 17, 16, 15]
Worst case sorted array:
 [15, 16, 17, 18, 19]

Unsorted array: [15, 16, 17, 18, 19]
Best case sorted array:
 [15, 16, 17, 18, 19]

Unsorted array: [16, 15, 19, 18, 17]
Average case sorted array:
 [15, 16, 17, 18, 19]
```

## Analysis / Observations:

Worst-case time complexity: O(n + k)
Best-case time complexity: O(n + k)
Average-case time complexity: O(n)

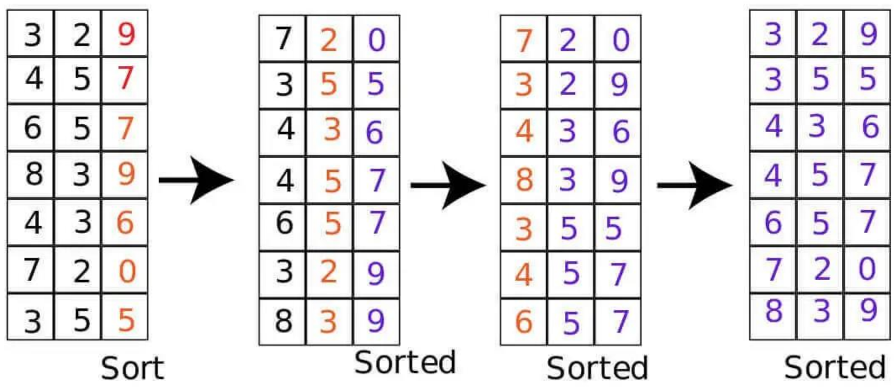
## Experiment No. 12

**Aim:** Write a program to implement Radix sort algorithm. Perform the analysis of implementation for best case, average case and worst case.

## Introduction:

Radix sort is a type of sorting algorithm that sorts data based on integer keys by grouping them by their individual digits in the same significant position and value. This method is useful for sorting numbers or strings that have varying lengths. The algorithm's time complexity is O(nk), where n is the number of items to be sorted, and k is the number of digits in the largest element.

## Example:



## Algorithm:

1. Find the maximum element in the array, and determine the number of digits in it.
2. Starting from the least significant digit, sort the array using a stable counting sort algorithm.
3. Repeat step 2 for each digit, moving towards the most significant digit.
4. After sorting for all digits, the array will be sorted in non-decreasing order.

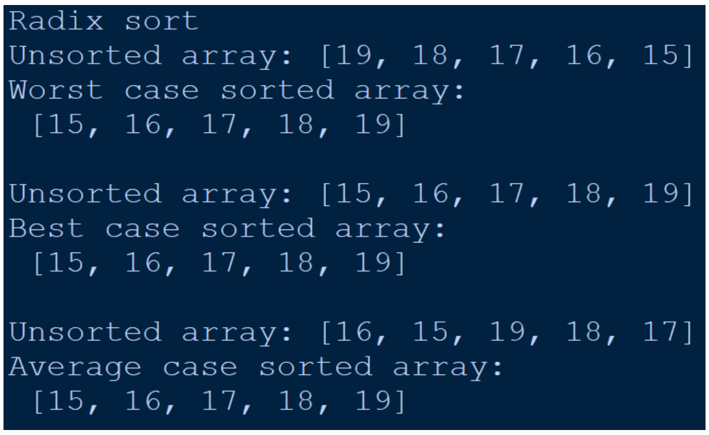
## Source code:

```
def countingSort(array, place):
    size = len(array)
    output = [0] * size
    count = [0] * (max(array)+1)
    for i in range(0, size):
        index = array[i] // place
        count[index % 10] += 1
    for i in range(1, len(count)):
        count[i] += count[i - 1]
    i = size - 1
    while i >= 0:
        index = array[i] // place
        output[count[index % 10] - 1] = array[i]
        count[index % 10] -= 1
        i -= 1
    for i in range(0, size):
        array[i] = output[i]


def radixSort(array):
    max_element = max(array)


    place = 1
    while max_element // place > 0:
        countingSort(array, place)
        place *= 10
print("Radix sort ")
a=[19,18,17,16,15]
print("Unsorted array:",a)
radixSort(a)
print("Worst case sorted array:\n",a)
print()
b=[15,16,17,18,19]
print("Unsorted array:",b)
radixSort(b)
print("Best case sorted array:\n",b)
print()
c=[16,15,19,18,17]
print("Unsorted array:",c)
```

radixSort(c)

print("Average case sorted array:\n",c)

## Output:



```
Radix sort
Unsorted array: [19, 18, 17, 16, 15]
Worst case sorted array:
 [15, 16, 17, 18, 19]

Unsorted array: [15, 16, 17, 18, 19]
Best case sorted array:
 [15, 16, 17, 18, 19]

Unsorted array: [16, 15, 19, 18, 17]
Average case sorted array:
 [15, 16, 17, 18, 19]
```

## Analysis / Observations:

Radix sort is a non-comparative sorting algorithm that is better than the comparative sorting algorithms. It has linear time complexity that is better than the comparative algorithms with complexity O(nlogn).

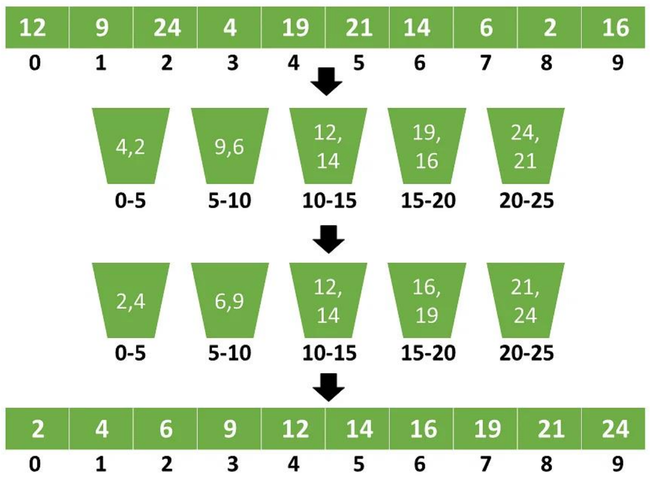
## Experiment No. 13

**Aim:** Write a program to implement Bucket sort algorithm. Perform the analysis of implementation for best case, average case and worst case.

## Introduction:

Bucket sort is a sorting technique that sorts an array by distributing its elements into several buckets, each of which is then sorted separately. It is highly effective when the input is uniformly distributed over a range. The algorithm's time complexity is O(n+k), where n represents the number of elements to be sorted and k represents the number of buckets used.

## Example:



## Algorithm:

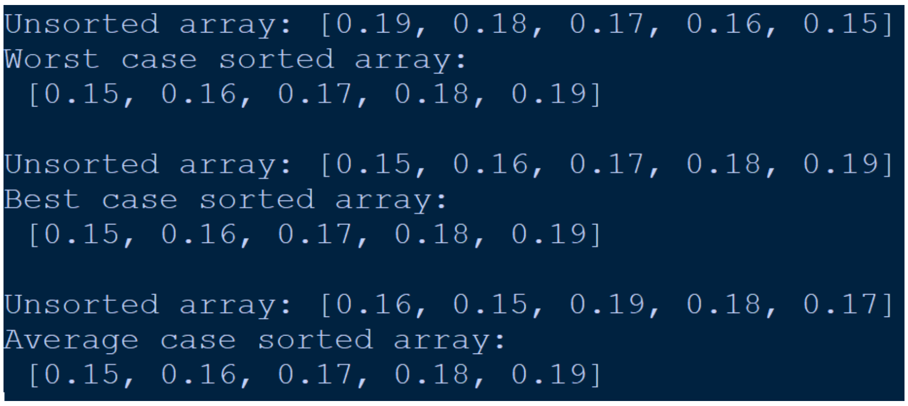
1. Create an empty array of buckets.
2. Iterate through the array to be sorted and distribute each element into its corresponding bucket based on its value.
3. Sort each bucket using a simple sorting algorithm, such as insertion sort or selection sort.
4. Concatenate the sorted buckets to get the final sorted array.
5. Return the sorted array.

## Source code:

```
def bucketSort(array):
    bucket = []
    for i in range(len(array)):
        bucket.append([])
    for j in array:
        index_b = int(10 * j)
        bucket[index_b].append(j)
    for i in range(len(array)):
        bucket[i] = sorted(bucket[i])
    k = 0
    for i in range(len(array)):
        for j in range(len(bucket[i])):
            array[k] = bucket[i][j]
            k += 1
    return array
a=[.19,.18,.17,.16,.15]
print("Unsorted array:",a)
bucketSort(a)
print("Worst case sorted array:\n",a)
print()
b=[.15,.16,.17,.18,.19]
print("Unsorted array:",b)
bucketSort(b)
print("Best case sorted array:\n",b)
print()
```

```
c=[.16,.15,.19,.18,.17]

print("Unsorted array:",c)

bucketSort(c)

print("Average case sorted array:\n",c)
```

## Output:



```
Unsorted array: [0.19, 0.18, 0.17, 0.16, 0.15]
Worst case sorted array:
 [0.15, 0.16, 0.17, 0.18, 0.19]

Unsorted array: [0.15, 0.16, 0.17, 0.18, 0.19]
Best case sorted array:
 [0.15, 0.16, 0.17, 0.18, 0.19]

Unsorted array: [0.16, 0.15, 0.19, 0.18, 0.17]
Average case sorted array:
 [0.15, 0.16, 0.17, 0.18, 0.19]
```

## Analysis / Observations:

Worst-case time complexity: O(n^2)
Best-case time complexity: O(n)
Average-case time complexity: O (n + k)

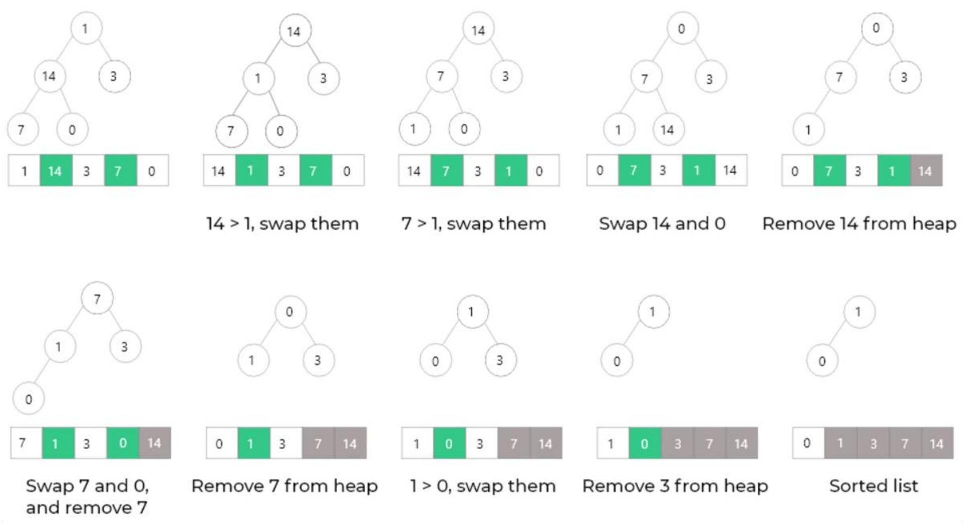
## Experiment No. 14

**Aim:** Write a program to implement heap sort algorithm. Perform the analysis of implementation for best case, average case and worst case.

## Introduction:

Heap sort is a sorting algorithm that uses a binary heap to sort an array. It works by repeatedly extracting the largest element from the heap until the array is sorted. This algorithm has a time complexity of O(n log n), which is faster than bubble sort and insertion sort.

## Example:



## Algorithm:

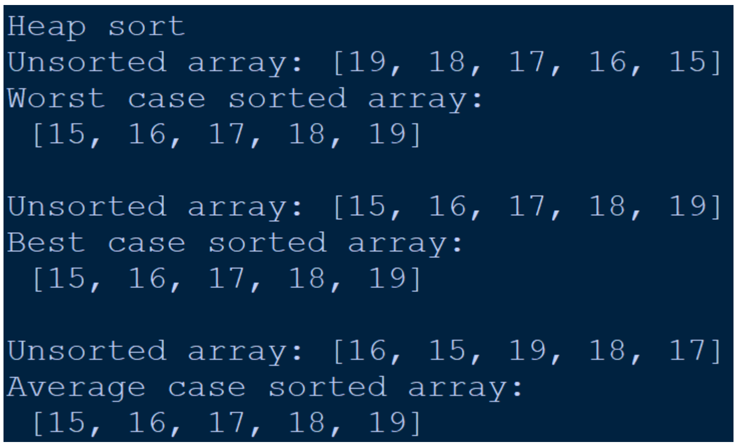
1. Build a max-heap from the given array.
2. The maximum element of the heap will be at the root node. Swap the root node with the last element of the heap, and decrease the heap size by 1.
3. Heapify the remaining elements of the heap to restore the heap property.
4. Repeat steps 2 and 3 until the heap size becomes 1.
5. The array is now sorted in ascending order.

## Source code:

```
def heapify(arr, n, i):

    largest = i

    l = 2 * i + 1 # left = 2*i + 1

    r = 2 * i + 2 # right = 2*i + 2

    if l < n and arr[i] < arr[l]:

        largest = l

    if r < n and arr[largest] < arr[r]:

        largest = r
    if largest != i:
        (arr[i], arr[largest]) = (arr[largest], arr[i]) # swap

        heapify(arr, n, largest)
def heapSort(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    for i in range(n - 1, 0, -1):
        (arr[i], arr[0]) = (arr[0], arr[i])
        heapify(arr, i, 0)


print("Heap sort ")
a=[19,18,17,16,15]
print("Unsorted array:",a)
heapSort(a)
print("Worst case sorted array:\n",a)
print()
b=[15,16,17,18,19]
print("Unsorted array:",b)
heapSort(b)
print("Best case sorted array:\n",b)
print()
c=[16,15,19,18,17]
print("Unsorted array:",c)
```

```
heapSort(c)

print("Average case sorted array:\n",c)
```

## Output:



```
Heap sort
Unsorted array: [19, 18, 17, 16, 15]
Worst case sorted array:
 [15, 16, 17, 18, 19]

Unsorted array: [15, 16, 17, 18, 19]
Best case sorted array:
 [15, 16, 17, 18, 19]

Unsorted array: [16, 15, 19, 18, 17]
Average case sorted array:
 [15, 16, 17, 18, 19]
```

## Analysis / Observations:

Worst-case time complexity: O(nlog n)
Best-case time complexity: O(nlog n)
Average-case time complexity: O(nlog n)

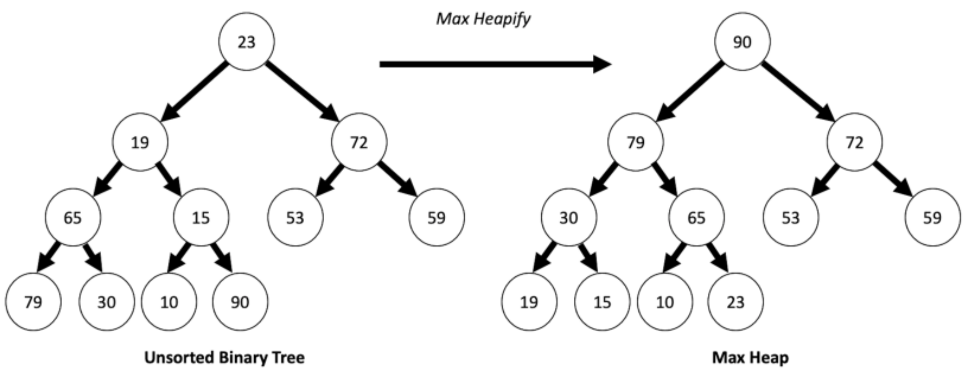
## Experiment No. 15

**Aim:** Write a program to implement Heapify function to create a binary heap.

## Introduction:

Heap is a special tree-based data structure. A binary tree is said to follow a heap data structure if it is a complete binary tree. All nodes in the tree follow the property that they are greater than their children i.e. the largest element is at the root and both its children and smaller than the root and so on. Such a heap is called a max-heap. If instead, all nodes are smaller than their children, it is called a min-heap.

## Example:



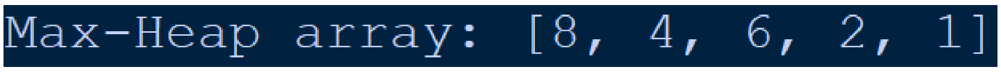
## Algorithm:

1. First increase the heap size by 1, so that it can store the new element.
2. Insert the new element at the end of the Heap.
3. This newly inserted element may distort the properties of Heap for its parents.
4. So, in order to keep the properties of Heap, heapify this newly inserted element following a bottom-up approach.

## Source code:

```
def heapify(arr, n, i):

  largest = i

  l = 2 * i + 1

  r = 2 * i + 2

  if l < n and arr[i] < arr[l]:

    largest = l
    if r < n and arr[largest] < arr[r]:
        largest = r
    if largest != i:
        arr[i],arr[largest] = arr[largest],arr[i]
        heapify(arr, n, largest)
def insert (array, newNum):
    size = len(array)
    if size == 0:
        array.append(newNum)
    else:
        array.append(newNum);
        for i in range((size//2)-1, -1, -1):
            heapify(array, size, i)
arr = []
insert(arr, 8)
insert(arr, 2)
insert(arr, 6)
insert(arr, 4)
insert(arr, 1)
print ("Max-Heap array: " + str(arr))
```

## Output:



```
Max-Heap array: [8, 4, 6, 2, 1]
```

## Analysis / Observations:

Worst-case time complexity: O(nlog n)
Best-case time complexity: O(1)
Average-case time complexity: O(nlog n)