



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

■
Design & Analysis of Algorithm - LAB MANUAL

Subject Code: **25UIT502P**
Academic Year : 2025-2026
Semester: V

KNOW ABOUT YOUR LAB





Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

INDEX

Sr No.	Contents
1	Vision & Mission of the Institute
2	Vision & Mission of the Department
3	Program Educational Objectives
4	Program Outcomes
5	Program Specific Outcomes
6	Course Outcomes & CO-PO & PSO Mapping
7	List of Experiments & Content Beyond Syllabus
8	Description of Experiments



LOKMANYA TILAK JANKALYAN SHIKSHAN SANSTHA'S

PRIYADARSHINI COLLEGE OF ENGINEERING

An Autonomous Institute Affiliated to R.T.M. Nagpur University, Nagpur

Accredited with Grade 'A+' by NAAC

Near C.R.P.F. Campus, Hingna Road, Nagpur - 440 019 (Maharashtra) India

Phone : 07104 - 299648, Fax : 07104-299681

E-mail : principal.pce.ngp@gmail.com • Website: www.pcenagpur.edu.in

AICTE ID No. 1-5435581; DTE CODE No. 4123,

UNIVERSITY CODE No. : 278



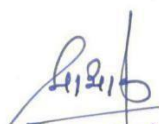
Vision

To become one of the India's leading Engineering Institutes in both education and research. We are committed to provide quality and state-of-the-art technical education to our students so that they become Technologically competent and in turn contribute for creating a great society.

Mission

1. Fostering a dynamic learning environment that equips students with Technical expertise, problem-solving skills and a deep commitment to ethical practices.
2. To cultivate a culture of innovation, incubation, research and entrepreneurship that drives technological advancements.
3. To uphold the spirit of mutual excellence while interacting with stake holders of our Institutional ecosystem.
4. Promoting lifelong learning, professional growth and ensuring holistic development of students and the well being of society.




Principal
Priyadarshini College of Engg.
Nagpur.

LOKMANYA TILAK JANKALYAN SHIKSHAN SANSTHA

Lokmanya Tilak Bhavan, Laxmi Nagar, Nagpur - 440 022. Maharashtra, INDIA. Tel : +91-712-2230665, 2245121. Fax No. : +91-712 2221430. Website : www.ltjss.net



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

PROGRAM EDUCATIONAL OBJECTIVES :

PEOs	Program Educational Objectives Statements
PEO1	Demonstrate strong technical expertise and uphold ethical values to excel in their professional careers.
PEO2	Apply advanced skills and knowledge in Information Technology to address real-world challenges and contribute to industry success.
PEO3	Pursue lifelong learning and engage in research and innovation to address societal needs and advance the field of Information Technology.



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

PROGRAM OUTCOMES :

Engineering Graduates will be able to:

POs	Program Outcomes
PO1	Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.
PO2	Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO3	Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO4	Conduct investigations of complex problems: Use research-based knowledge and research methods Including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO6	The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO7	Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8	Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO9	Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

PO10	Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO11	Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.



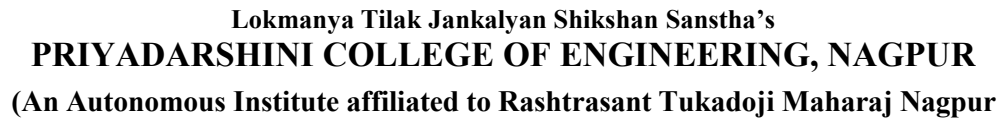
Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

PROGRAM SPECIFIC OUTCOMES :

PSOs	Program Specific Outcomes
PSO1	An ability to apply mathematical foundations, algorithmic principles and computer science theory in the modeling and design of software systems of varying complexity.
PSO2	An ability to work with Open Source Software and use off the shelf utilities for program integration.



Students will able to:

CO-PO & PSO MAPPING:

[illegible]



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

List of Experiments:

Sr. No.	Name of Practical
1	Write a program to perform operation count for a given pseudo code.
2	Write a program to perform Bubble sort for any given list of numbers.
3	Write a program to perform Insertion sort for any given list of numbers.
4	Write a program to perform Quick Sort for the given list of integer values.
5	Write a program to find Maximum and Minimum of the given set of integer values.
6	Write a Program to perform Merge Sort on the given two lists of integer values.
7	Write a Program to perform Binary Search for a given set of integer values recursively and non recursively.
8	Write a program to find solution for knapsack problem using greedy method.
9	Write a program to find minimum cost spanning tree using Prim's Algorithm.
10	Write a program to find minimum cost spanning tree using Kruskal's Algorithm.
Content Beyond Syllabus	
11	
12	



PRACTICAL NO. 1

Aim: Write a program to perform operation count for a given pseudo code.

Theory:

What is Design Algorithm and analysis?

Design and Analysis of Algorithms (DAA) is the branch of computer science that deals with:

1 Designing efficient algorithms to solve computational problems.

2 Analyzing the performance of these algorithms in terms of time and space complexity.

It helps ensure that programs not only work correctly, but also perform efficiently even for large input sizes.

Operation Count:

In the Design and Analysis of Algorithms (DAA), analyzing how an algorithm performs with varying input sizes is crucial. One fundamental technique used is operation counting, which refers to counting the number of basic operations (like comparisons, assignments, additions, etc.) that an algorithm performs.

This analysis helps us understand the efficiency and time complexity of an algorithm.

Why is Operation Count Important?

To understand how efficient an algorithm is

To compare different algorithms solving the same problem

To estimate how an algorithm behaves as the input size increases

To derive asymptotic complexities like $O(n)$, $O(n^2)$, $O(\log n)$, etc.

Types of Operations Usually Counted:

Arithmetic operations: +, -, *, /

Assignment operations: =

Comparison operations: ==, <, >, !=

Logical operations: AND, OR, NOT

Array access or indexing: $A[i]$

Each of these is considered a basic operation, typically counted as a single unit.

Pseudo Code:

Consider the following pseudo code:

for $i = 1$ to n

 for $j = 1$ to n

$A[i][j] = i + j$

Operation Count Explanation:

This pseudo code consists of:

Two nested loops, each running from 1 to n .

The body of the inner loop performs:

One addition operation: $i + j$

One assignment operation: $A[i][j] = \dots$

► Total number of iterations:

Outer loop: n times

Inner loop (for each iteration of i): n times

So, total iterations of the inner statement = $n \times n = n^2$

► Operations:



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

Addition ($i + j$): Executed n^2 times
Assignment ($A[i][j] =$): Executed n^2 times
Total operations = $2 \times n^2$

Time Complexity:

As the number of basic operations grows proportionally with , the time complexity of the algorithm is:

$\boxed{O(n^2)}$

This is called quadratic time complexity.

Purpose of Operation Count:

Helps analyze how an algorithm scales

Useful to compare performance of different algorithms

Provides foundation for understanding asymptotic analysis

Gives an idea of the best, worst, and average-case behavior of an algorithms

Algorithm:

Step 1: Start

Step 2: Initialize addition $\leftarrow 0$, assignment $\leftarrow 0$

Step 3: Create a 2D array A of size $n \times n$

Step 4: Repeat for $i = 0$ to $n - 1$

Step 5: Repeat for $j = 0$ to $n - 1$

Step 6: $A[i][j] \leftarrow i + j$

Step 7: addition \leftarrow addition + 1 // Counting addition operation

Step 8: assignment \leftarrow assignment + 1 // Counting assignment operation

[End of inner loop]

[End of outer loop]

Step 9: Print addition

Step 10: Print assignment

Step 11: Print total operations \leftarrow addition + assignment

Step 12: Stop

Program:

```
#include<stdio.h>
//#include<conio.h>
void main()
{
    int count=0,sum=0,n,i,a[50];
    //clrscr();
    count=count+1;
    printf("\n Enter the n value:");
    scanf("%d",&n);
```



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

```
count=count+1;
printf("\n Enter %d values to sum:",n);
for(i=0;i<n;i++)
{
count=count+1;
scanf("%d",&a[i]);
}
count=count+1;
for(i=0;i<n;i++)
{
count=count+1;
sum=sum+a[i];
count=count+1;
}
count=count+1;
printf("\n The of %d values is:%d and count is=%d",n,sum,count);
//getch();
}
```

Output:-

Output

Enter the n value:5

Enter 5 values to sum:4

3
2
6
1

The of 5 values is:16 and count is=19



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

■

Conclusion: Operation count is a fundamental method to evaluate and understand an algorithm's efficiency. It provides a quantitative measure of how an algorithm will perform as input size increases. In our example, we observed that a nested loop resulted in $O(n^2)$ time complexity, making the algorithm inefficient for large values of n .

By learning operation count, we gain the ability to predict performance, optimize code, and make informed decisions when choosing or designing algorithms.

Viva Questions and Answers:

1. What is operation count?

Answer:

Operation count refers to the total number of basic or primitive operations (like addition, assignment, comparison, etc.) performed by an algorithm during its execution. It helps in determining the efficiency of an algorithm.

2. Why do we calculate the operation count for an algorithm?

Answer:

We calculate the operation count to analyze the actual cost of an algorithm in terms of performance, especially for small input sizes or when comparing multiple algorithms with the same time complexity.

3. What do you mean by primitive operations?

Answer:

Primitive operations are basic low-level operations like:

Assignment (=)

Arithmetic (+, -, *, /)

Comparison (<, ==)

Array indexing (A[i])

Function calls

Each of these is considered to take constant time ($O(1)$).

◇ 4. Which operations are generally considered in operation count?

Answer:

Commonly considered operations are:

Assignments

Comparisons

Increments/Decrements

Arithmetic operations

Array accesses

Loop control (initialization, condition check, increment)

5. How does operation count help in analyzing an algorithm?

Answer:

It provides a more accurate picture of algorithm efficiency, especially for small inputs or real-world performance, and helps in choosing between similar algorithms.



PRACTICAL NO. 2

Aim: Write a program to perform Bubble sort for any given list of numbers.

Theory:

Bubble Sort is a simple and widely known comparison-based sorting algorithm. It works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order. The process is repeated until the list becomes sorted.

The algorithm gets its name because smaller elements "bubble" to the top of the list (beginning), while larger elements sink to the bottom (end) after each pass.

This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where n is the number of items.

How it works :-

1. We sort the array using multiple passes. After the first pass, the maximum element goes to end (its correct position). Same way, after second pass, the second largest element goes to second last position and so on.
2. In every pass, we process only those elements that have already not moved to correct position. After k passes, the largest k elements must have been moved to the last k positions.
3. In a pass, we consider remaining elements and compare all adjacent and swap if larger element is before a smaller element. If we keep doing this, we get the largest (among the remaining elements) at its correct position.

Algorithm:

```
Step 1 :- Start
Step 2 :- for i = 0 to n - 2 do
Step 3 :-   for j = 0 to n - i - 2 do
Step 4 :-   if A[j] > A[j + 1] then
Step 5 :-     swap A[j] and A[j + 1]
               end if
           (End of step 3)end for
       (End of step 2)end for
Step 6 :- return A
Step 7 :- Stop
```

Or Algorithm

```
Step 1 :- Start

Step 2 :- Repeat for i = 0 to n-1
(where n is the number of elements in the array)
```

Step 3 :- For each pass, compare adjacent elements:

If the left element is greater than the right element → swap them

Step 4 :- After each pass, the largest unsorted element bubbles to its correct position.

Step 5 :- Repeat the process until the array is sorted.



Step 6 :- End

Program:

```
#include<stdio.h>
#include<conio.h>
void bubblesort(int[],int);
void display(int[],int);
int main()
{
    int a[20],n,i;
    clrscr();
    printf("\n Enter the number of elements in array are:");
    scanf("%d",&n);
    printf("\n Enter %d elements in the array:",n);
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    bubblesort(a,n);
    printf("\n The sorted elements in the array are:");
    display(a,n);
    getch();
    return 0;
}
void bubblesort(int a[],int n)
{
    int i,j,temp,excg=0;
    int last=n-1;
    for(i=0;i<n;i++)
    {
        excg=0;
        for(j=0;j<last;j++)
        {
            if(a[j]>a[j+1])
            {
                temp=a[j];
                a[j]=a[j+1];
                a[j+1]=temp;
                excg++;
            }
        }
        if(excg==0)
            return ;
        else
            last=last-1;
    }
}
DESIGN AND ANALYSIS OF ALGORITHMS LAB
Prepared by: Dept. of CSE, RGM CET Page 4
}
```



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

```
void display(int a[],int n)
{
    int i;
    for(i=0;i<n;i++)
        printf("%d \t",a[i]);
}
```

Output:-

```
Output

Enter the number of elements in array: 5

Enter 5 elements in the array: 5 4 3 2 1

The sorted elements in the array are:
1  2  3  4  5
```

□

Conclusion: In the study of Design and Analysis of Algorithms (DAA), Bubble Sort is a fundamental algorithm that helps in understanding basic concepts of algorithm design, performance analysis, and time complexity.

Viva Questions and Answers :



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

1. What is Bubble Sort?

Answer:

Bubble Sort is a simple comparison-based sorting algorithm. It repeatedly compares and swaps adjacent elements if they are in the wrong order, and this process continues until the entire list is sorted.

2. Why is it called "Bubble" Sort?

Answer:

It is called "Bubble" Sort because during the sorting process, the largest (or smallest) elements "bubble up" to their correct positions at the end of the list in each iteration.

3. What is the time complexity of Bubble Sort?

Answer:

Best Case (Already Sorted): $O(n)$

Average Case: $O(n^2)$

Worst Case (Reversed List): $O(n^2)$

4. Is Bubble Sort a stable sorting algorithm?

Answer:

Yes, Bubble Sort is a stable sorting algorithm. It does not change the relative order of equal elements during sorting.

5. When should Bubble Sort be preferred?

Answer:

Bubble Sort should only be used for educational purposes or very small datasets, as it is simple to understand but inefficient for large lists compared to other algorithms like Quick Sort or Merge Sort.



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur
DEPARTMENT OF INFORMATION TECHNOLOGY



PRACTICAL NO. 3

Aim: Write a program to perform Insertion sort for any given list of numbers.

Theory: Insertion Sort is one of the most fundamental and intuitive sorting algorithms in computer science. It mimics the way humans often sort playing cards in their hands. Starting with an empty hand, we pick up one card at a time and insert it into its correct position relative to the already sorted cards. Similarly, Insertion Sort works by building a sorted portion of the array one element at a time, inserting each new element into its correct position.

Despite being simple and inefficient for large datasets, Insertion Sort plays an important role in understanding sorting logic and is still used in practical scenarios involving small or nearly sorted arrays. Its stability, low overhead, and simplicity make it ideal for embedded systems, education, and hybrid sorting algorithms (like Timsort).

Algorithm:

- Step 1: For $i \leftarrow 1$ to $n - 1$
 - a. $key \leftarrow arr[i]$
 - b. $j \leftarrow i - 1$
- Step 2: Repeat while $j \geq 0$ and $arr[j] > key$
 - a. $arr[j + 1] \leftarrow arr[j]$
 - b. $j \leftarrow j - 1$
- Step 3: $arr[j + 1] \leftarrow key$
- Step 4: Repeat Steps 1 to 3 for all elements
- Step 5: Return array
- Step 6: Stop

Program:

```
#include<stdio.h>
#include<conio.h>
void inssort(int[],int);
void display(int[],int);
int main()
{
    int a[20],n,i;
    clrscr();
    printf("\n Enter the number of elements in array are:");
```



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

```
scanf("%d",&n);
printf("\n Enter %d elements in the array:",n);
for(i=0;i<n;i++)
scanf("%d",&a[i]);
inssort(a,n);
printf("\n The sorted elements in the array are:");
display(a,n);
getch();
return 0;
}
void inssort(int a[],int n)
{
int i,j,index=0;
for(i=1;i<n;i++)
{
index=a[i];
j=i;
while((j>0)&&(a[j-1]>index))
{
a[j]=a[j-1];
j--;
}
a[j]=index;
}
}
void display(int a[],int n)
{
int i;
for(i=0;i<n;i++)
{
printf("%d\t",a[i]);
}
}
```



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

■
Output:

```
Output

Enter the number of elements in array: 5

Enter 5 elements in the array: 9 6 4 3 1

The sorted elements in the array are:
1 3 4 6 9
```



Conclusion: Insertion Sort is a basic yet powerful sorting technique best suited for small or nearly sorted datasets. It is intuitive, stable, and easy to implement. While it is not efficient for large data due to its quadratic time complexity, its simplicity and minimal memory usage make it ideal for specific use cases. Insertion Sort also helps in understanding the fundamentals of sorting, comparisons, and algorithmic design

Viva questions :-

Q1: What is the basic idea behind Insertion Sort?

Answer:

Insertion Sort works by building a sorted list one element at a time. At each step, it picks the next element from the unsorted part and inserts it into the correct position in the sorted part.

Q2: What is the time complexity of Insertion Sort?

Answer:

Best Case (Already Sorted): $O(n)$

Average Case: $O(n^2)$

Worst Case (Reverse Sorted): $O(n^2)$

It is efficient only for small datasets.

Q3: Is Insertion Sort stable and in-place?

Answer:

Yes.

Stable: It does not change the relative order of equal elements.

In-place: It uses only a constant amount of additional memory ($O(1)$).

Q4: When is Insertion Sort preferred over other algorithms?

Answer:



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

It is preferred when:

The dataset is small

The list is already partially sorted

Stability is required

Memory usage must be minimal

Q5: How does Insertion Sort differ from Bubble Sort?

Answer:

Insertion Sort inserts elements into their correct position in the sorted part.

Bubble Sort repeatedly swaps adjacent elements to "bubble" the largest value to the end.

Insertion Sort is generally faster than Bubble Sort for most cases.

PRACTICAL NO. 4

Aim: Write a program to perform Quick Sort for the given list of integer values.

Theory:- Quick Sort is a highly efficient sorting algorithm that follows the divide-and-conquer technique. It was developed by Tony Hoare in 1959 and is widely used due to its excellent performance on large datasets. The algorithm works by selecting a pivot element from the array and partitioning the other elements into two sub-arrays—those less than the pivot and those greater than or equal to it. These sub-arrays are then recursively sorted using the same process. This method places each pivot in its correct position in the sorted array. Quick Sort has an average-case time complexity of $O(n \log n)$, which is significantly better than simpler algorithms like Bubble Sort and Insertion Sort. Although its worst-case complexity is $O(n^2)$ (when poor pivot choices are made), using strategies like randomized pivot or median-of-three helps avoid this. Quick Sort is preferred in real-world applications for its speed, efficiency, and low memory usage.

Algorithm:

QUICK_SORT(arr, low, high)

Step 1: If low < high then

 Step 2: pivotIndex \leftarrow PARTITION(arr, low, high)

 Step 3: QUICK_SORT(arr, low, pivotIndex - 1)

 Step 4: QUICK_SORT(arr, pivotIndex + 1, high)

End If



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

OR
PARTITION(arr, low, high)

Step 1: pivot \leftarrow arr[low]

Step 2: i \leftarrow low + 1

Step 3: j \leftarrow high

Step 4: Repeat while i \leq j

a. While i \leq high AND arr[i] \leq pivot:

i \leftarrow i + 1

b. While arr[j] > pivot:

j \leftarrow j - 1

c. If i < j then

Swap arr[i] and arr[j]

Step 5: Swap arr[low] and arr[j] (pivot with element at j)

Step 6: Return j (pivot index)

Program:

```
#include<stdio.h>
#include<conio.h>
void qsort(int [],int,int);
int partition(int [],int,int);
void qsort(int a[],int first,int last)
{
    int j;
    if(first<last)
    {
        j=partition(a,first,last+1);
        qsort(a,first,j-1);
        qsort(a,j+1,last);
    }
}
int partition(int a[],int first,int last)
{
    int v=a[first];
    int i=first;
    int j=last;
    int temp=0;
    do
    {
        do
        {
            i++;
        }while(a[i]<v);
        do
        {
```



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

```
j--;  
}while(a[j]>v);  
if(i<j)  
{  
temp=a[i];  
a[i]=a[j];  
a[j]=temp;  
}  
}while(i<j);  
a[first]=a[j];  
a[j]=v;  
return j;  
}
```

DESIGN AND ANALYSIS OF ALGORITHMS LAB

Prepared by: Dept. of CSE, RGM CET Page 8

```
int main()  
{  
int a[40],i,n;  
clrscr();  
printf("\n Enter the no of elements (size):");  
scanf("%d",&n);  
printf("\n Enter the ELeMents to sort:");  
for(i=0;i<n;i++)  
scanf("%d",&a[i]);  
qsort(a,0,n-1);  
printf("\n The ELeMents after sorting are:");  
for(i=0;i<n;i++)  
{  
printf("%d\t",a[i]);  
}  
getch();  
return 0;  
}
```

Output:

```
Output  
  
Enter the number of elements: 6  
  
Enter the elements to sort: 11 23 45 63 7 20  
  
The elements after sorting are:  
7 11 20 23 45 63
```



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

■

Conclusion:- Quick Sort is an efficient, in-place, and widely used sorting algorithm. It performs better than other sorting techniques like Bubble Sort and Insertion Sort for large datasets. However, its performance depends on the choice of pivot. With good pivot selection (like random pivot or median-of-three), Quick Sort gives optimal performance in most practical scenarios.

Viva questions :-

Q1: What is the basic idea behind the Quick Sort algorithm?

Ans: Quick Sort uses the divide-and-conquer approach. It selects a pivot, partitions the array, and recursively sorts the sub-arrays.

Q2: What is the worst-case time complexity of Quick Sort?

Ans: The worst-case time complexity is $O(n^2)$, which happens when the smallest or largest element is always chosen as the pivot.

Q3: Why is Quick Sort considered faster than other simple sorting algorithms like Bubble Sort or Insertion Sort?

A: Because its average-case time complexity is $O(n \log n)$ and it performs fewer comparisons and swaps in practice.

Q4: Is Quick Sort a stable sorting algorithm?

Ans: No, Quick Sort is not stable because it may change the relative order of equal elements.

Q5: What is the space complexity of Quick Sort?

Ans: The space complexity is $O(\log n)$ due to the recursive call stack, as it is an in-place sorting algorithm.

PRACTICAL NO. 5

Aim: Write a program to find Maximum and Minimum of the given set of integer values.

Theory:

1. Introduction

In algorithm design, finding the maximum and minimum values from a given set of integers is a basic but essential problem. It introduces students to the concepts of problem-solving through linear algorithms and understanding how performance is affected by input size.

This problem is typically solved using a linear search approach. The algorithm scans the entire list or array of numbers and uses simple comparisons to find:

The maximum value (largest number)

The minimum value (smallest number)



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

■

This process involves initializing the first element as both max and min, and then iterating through the rest of the array while updating the values whenever a larger or smaller number is found.

2. Set of Integers

A set of integers is a group or collection of whole numbers which can be:

Positive numbers (e.g., 10, 23)

Negative numbers (e.g., -5, -8)

Zero (0)

These numbers do not have any decimal or fractional part.

In programming, we store and manipulate such a set of integers using arrays, which make it easy to access each number using an index.

3. Array

An array is a linear data structure used to store multiple values of the same type in a single variable. Arrays are widely used in programming because they allow:

Efficient access to elements via indexing

Fixed-size memory allocation

Sequential storage in contiguous memory

Each element in an array can be accessed directly using its index. Indexing starts from 0 in most programming languages like C, C++, and Java.

Example:

c

CopyEdit

```
int arr[5] = {12, -4, 0, 99, 33};
```

Here, arr is an integer array containing 5 elements.

```
arr[0] = 12
```

```
arr[1] = -4
```

```
arr[2] = 0
```

```
arr[3] = 99
```

```
arr[4] = 33
```

This structure is ideal for performing operations like searching, sorting, finding max/min, and data analysis.

Input Representation

In this problem, the input consists of:

A number n, representing the number of integers

An array arr[n], which stores the n integer values

The array allows us to store and traverse the input set easily using loops. During traversal, we can apply logic to update the maximum and minimum values.

5. Time and Space Complexity

Understanding algorithm performance is key in DAA. We analyze how the algorithm behaves with different input sizes:

Time Complexity:

Best Case: $O(n)$



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

Worst Case: $O(n)$

→ Every element is visited once, and each comparison takes constant time.

Space Complexity:

$O(1)$

→ Only a few variables (max, min, loop counters) are used. No extra space is needed apart from the input array.

This makes the approach space-efficient, especially suitable for large datasets.

Algorithm:

Input: An array of integers `arr[]` of size `n`.

Output: The maximum and minimum values from the array.

Step 1- Start

Step 2-Input the number of elements `n`

Step 3- Input `n` integer values into array `arr[]`

Step 4-Initialize:

`max = arr[0]`

`min = arr[0]`

Step 5- Repeat for `i = 1` to `n - 1`

If `arr[i] > max` then `max = arr[i]`

If `arr[i] < min` then `min = arr[i]`

Step 6-Print max and min

Step 7- Stop

Program:

```
#include<stdio.h>
#include<conio.h>
void minmax(int,int,int,int);
int i,j,a[50],n,fmax,fmin;
int main()
{
    clrscr();
    printf("\n Enter the number of elements in array are:");
    scanf("%d",&n);
    printf("\n Enter %d elements in the array:",n);
```



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

```

    for(i=0;i<n;i++)
    scanf("%d",&a[i]);
    printf("\n The Elements in the array are:");
    for(i=0;i<n;i++)
    printf("%d\n",a[i]);
    //fmax=fmin=a[0];
    minmax(0,n-1,a[0],a[0]);
    printf("\n The minimum Element of the list of elements is:%d",fmin);
    printf("\n The maximum Element of the list of elements is:%d",fmax);
    getch();
    return 0;
}
void minmax(int i,int j,int max,int min)
{
    int gmax,gmin,hmax,hmin;
    gmax=hmax=max;
    gmin=hmin=min;
    if(i==j)
    {
        fmax=fmin=a[i];
    }
    else if(i==(j-1))
    {
        if(a[i]>a[j])
        {
            fmax=a[i];
            fmin=a[j];
        }
        else
        {
            fmax=a[j];
            fmin=a[i];
        }
    }
    else
    {
        int mid=(i+j)/2;
        minmax(i,mid,a[i],a[i]);
        gmax=fmax;
        gmin=fmin;
        minmax(mid+1,j,a[mid+1],a[mid+1]);
        hmax=fmax;
        hmin=fmin;
        if(gmax>hmax)
        {
            fmax=gmax;
        }
    }
}
```

DESIGN AND ANALYSIS OF ALGORITHMS LAB

Prepared by: Dept. of CSE, RGM CET Page 10



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

```
else
{
fmax=hmax;
}
if(gmin>hmin)
{
fmin=hmin;
}
else
{
fmin=gmin;
}
}
```

Output:

```
Enter the number of elements in array: 4

Enter 4 elements in the array:
11
23
45
1

The elements in the array are:
11
23
45
1

The minimum element is: 1
The maximum element is: 45
```



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

■

Conclusion: In this practical, we implemented a simple and efficient algorithm to find the maximum and minimum values from a set of integers. Using a linear approach, we achieved a time complexity of $O(n)$. This task strengthens the understanding of basic algorithm design and performance analysis in DAA.

Viva voice Question:

1. What is the time complexity of the algorithm used to find maximum and minimum in an array?

Answer: The time complexity is $O(n)$, because each element in the array is visited exactly once during the linear traversal.

2. How many comparisons are made in the linear approach?

Answer: In the linear approach, for each element from index 1 to $n-1$, two comparisons can be made—one with max and one with min. So, in the worst case, $2(n-1)$ comparisons are made. However, some comparisons can be skipped depending on the value. This shows that while the approach is simple, it can be optimized further.

3. What changes would you make to handle both float and integer arrays in the same program?

Answer: To handle both float and integer arrays, we can use function overloading (in C++ or Java) or use generic programming (like templates). In C, we would need to write separate functions for each data type or use void pointers and typecasting. This would make the program more flexible and reusable for different types of numeric inputs.

4. How would this program behave if all array elements are the same?

Answer: If all elements are the same, then the first value itself will remain both the maximum and minimum throughout the loop. The program will still work correctly, and all comparisons will result in no change to max or min. This confirms the correctness and stability of the linear algorithm even in edge cases.

5. How can this algorithm be improved using Divide and Conquer?

Answer: The Divide and Conquer method reduces the number of comparisons by dividing the array into two halves, recursively finding the max and min of each half, and then comparing the results. This approach uses $3n/2 - 2$ comparisons, which is better than the linear approach's $2(n-1)$ comparisons in terms of efficiency, especially for large inputs. However, it involves recursion and slightly more complex code.



PRACTICAL NO. 6

Aim: Write a Program to perform Merge Sort on the given two lists of integer values.

Theory: Merge Sort is a divide and conquer sorting algorithm that divides the list into two halves recursively until each sublist contains one element. Then, it merges these sublists to produce new sorted sublists until there is only one sorted list. It is a stable sorting algorithm, which means it preserves the order of equal elements. Merge Sort works by repeatedly splitting the array and then merging the sorted parts. It is efficient with a time complexity of $O(n \log n)$ in all cases. The algorithm uses extra space for temporary arrays during merging, so it is not an in-place algorithm. It performs well on large datasets and guarantees consistent performance. Merge Sort is preferred when stability and predictable time complexity are required. Due to its recursive nature, it can be implemented easily using a simple divide-and-conquer strategy. Overall, Merge Sort is a powerful and reliable sorting method widely used in computer science.

Algorithm:

Step 1: $n1 \leftarrow \text{mid} - \text{left} + 1$

Step 2: $n2 \leftarrow \text{right} - \text{mid}$

Step 3: Create temporary arrays:

$\text{Left}[0 \dots n1 - 1]$ and $\text{Right}[0 \dots n2 - 1]$

Step 4: Copy data into temporary arrays:

For $i = 0$ to $n1 - 1$:

$\text{Left}[i] \leftarrow \text{arr}[\text{left} + i]$

For $j = 0$ to $n2 - 1$:

$\text{Right}[j] \leftarrow \text{arr}[\text{mid} + 1 + j]$

Step 5: Initialize $i \leftarrow 0, j \leftarrow 0, k \leftarrow \text{left}$

Step 6: Repeat while $i < n1$ AND $j < n2$

a. If $\text{Left}[i] \leq \text{Right}[j]$ then

$\text{arr}[k] \leftarrow \text{Left}[i]$

$i \leftarrow i + 1$

b. Else

$\text{arr}[k] \leftarrow \text{Right}[j]$

$j \leftarrow j + 1$

c. $k \leftarrow k + 1$



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

Step 7: Copy remaining elements of Left[] if any:

While $i < n1$:

arr[k] \leftarrow Left[i]

$i \leftarrow i + 1$

$k \leftarrow k + 1$

Step 8: Copy remaining elements of Right[] if any:

While $j < n2$:

arr[k] \leftarrow Right[j]

$j \leftarrow j + 1$

$k \leftarrow k + 1$

Program:

```
#include<stdio.h>
#include<conio.h>
void merge(int[],int,int,int);
void mergesort(int[], int,int);
void merge(int a[25], int low, int mid, int high)
{
    int b[25],h,i,j,k;
    h=low;
    i=low;
    j=mid+1;
    while((h<=mid)&&(j<=high))
    {
        if(a[h]<a[j])
        {
            b[i]=a[h];
            h++;
        }
        else
        {
            b[i]=a[j];
            j++;
        }
        i++;
    }
    if(h>mid)
    {
        for(k=j;k<=high;k++)
        {
            b[i]=a[k];
            i++;
        }
    }
    else
    {

```



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

```

    for(k=h;k<=mid;k++)
    {
        b[i]=a[k];
        i++;
    }
    for(k=low;k<=high;k++)
DESIGN AND ANALYSIS OF ALGORITHMS LAB
Prepared by: Dept. of CSE, RGM CET Page 12
    {
        a[k]=b[k];
    }
}
void mergesort(int a[25],int low,int high)
{
    int mid;
    if(low<high)
    {
        mid=(low+high)/2;
        mergesort(a,low,mid);
        mergesort(a,mid+1,high);
        merge(a, low,mid,high);
    }
}
void main()
{
    int a[25],i,n;
    clrscr();
    printf("\n Enter the size of the elements to be sorted:");
    scanf("%d",&n);
    printf("\n Enter the elements to sort:");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    printf("\n The Elements before sorting are:");
    for(i=0;i<n;i++)
        printf("%d\t",a[i]);
    mergesort(a, 0,n-1);
    printf("\n The elements after sorting are:");
    for(i=0;i<n;i++)
        printf("%d\t",a[i]);
    getch();
}
```




■

Output:

```
Output

Enter the size of the elements to be sorted: 4

Enter the elements to sort: 3 56 32 21

The elements before sorting are:
3  56 32 21
The elements after sorting are:
3  21 32 56
```

Conclusion: Merge Sort efficiently sorts lists by dividing and merging them recursively. It guarantees $O(n \log n)$ time complexity in all cases. The algorithm is stable and maintains order of equal elements. Overall, Merge Sort is reliable for sorting large datasets with consistent performance.

Viva Question:



Q1. What is the main principle behind Merge Sort?

A1:

Merge Sort is based on the divide and conquer technique. First, it divides the input list into two halves recursively until each sublist contains only one element. After dividing, it merges these small sublists back together in a sorted manner. The merging process combines two sorted lists into one sorted list by comparing elements. This process continues until the entire list is merged and sorted. Thus, the main principle is splitting and merging to achieve sorting.

Q2. Is Merge Sort a stable sorting algorithm? Explain.

A2:

Yes, Merge Sort is a stable sorting algorithm. Stability means that if two elements are equal, their original order is maintained after sorting. This is useful when the order of equal elements carries meaning. Merge Sort achieves this by carefully merging elements from sublists without swapping equal elements unnecessarily. Because it merges sorted sublists in order, the stability is preserved. This makes Merge Sort preferred when stability is required.

Q3. What is the time complexity of Merge Sort in the best, average, and worst cases?

A3:

The time complexity of Merge Sort is $O(n \log n)$ for best, average, and worst cases. This is because it always divides the list into halves ($\log n$ divisions). Then, it merges all elements back, which takes linear time (n) for each level of recursion. Multiplying these gives $O(n \log n)$. Unlike other algorithms, Merge Sort guarantees this time complexity consistently. This predictable performance is a key advantage of Merge Sort.

Q4. Does Merge Sort perform sorting in-place? Why or why not?



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

A4:

Merge Sort does not perform sorting in-place. It requires extra temporary arrays to hold the divided parts during the merge process. These extra arrays use additional memory equal to the size of the input list. Because of this, it needs more space compared to in-place algorithms like Quick Sort. The extra space usage is the trade-off for its stable and predictable performance. So, Merge Sort is not memory-efficient but is reliable for large datasets.

Q5. Mention one advantage of using Merge Sort over other sorting algorithms.

A5:

One major advantage of Merge Sort is its consistent time complexity of $O(n \log n)$. Unlike Quick Sort, which can degrade to $O(n^2)$ in the worst case, Merge Sort guarantees stable performance regardless of input. It is also stable, meaning it maintains the relative order of equal elements. Merge Sort works well with large datasets and external sorting where data cannot fit into memory. These benefits make Merge Sort a reliable and widely used sorting algorithm.

PRACTICAL NO. 7

Aim: Write a Program to perform Binary Search for a given set of integer values recursively and non recursively.

Theory: Binary Search is a widely used searching algorithm that efficiently locates the position of a target element within a sorted list or array. Unlike Linear Search, which examines each element sequentially, Binary Search works on the principle of divide and conquer. It repeatedly divides the search interval into two halves, reducing the problem size significantly at each step.

To perform Binary Search:

Identify the middle element of the sorted array.

Compare the middle element with the target value:

If equal, the search is successful.

If the target is smaller, search continues in the left half.

If the target is larger, search continues in the right half.



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

■
This process repeats until the element is found or the subarray size reduces to zero.

Binary Search can be implemented using:

Non-Recursive (Iterative) Method: Uses a loop to search within the array.

Recursive Method: Calls itself to search in subarrays.

Both methods require the array to be sorted in ascending order.

The time complexity of Binary Search in the best, average, and worst-case scenarios is $O(\log n)$ due to the halving of the search space after each comparison.

The space complexity is:

$O(1)$ in the iterative approach.

$O(\log n)$ in the recursive approach (due to recursion stack).

Binary Search is more efficient than Linear Search, especially for large datasets.

It is commonly used in searching problems, dictionaries, database indexing, and other applications requiring fast lookup in sorted collections.

Algorithm:

BINARY_SEARCH_ITERATIVE(arr, n, key)

Step 1: low \leftarrow 0

Step 2: high \leftarrow n - 1

Step 3: Repeat while low \leq high

 a. mid \leftarrow (low + high) / 2

 b. If arr[mid] == key then

 Return mid (element found)

 c. Else If arr[mid] < key then

 low \leftarrow mid + 1

 d. Else

 high \leftarrow mid - 1

Step 4: End While

Step 5: Return -1 (element not found)

◇ Algorithm: **BINARY_SEARCH_RECURSIVE**(arr, low, high, key)

Step 1: If low > high then

 Return -1 (element not found)

End If



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

Step 2: $\text{mid} \leftarrow (\text{low} + \text{high}) / 2$

Step 3: If $\text{arr}[\text{mid}] == \text{key}$ then
Return mid (element found)

Else If $\text{arr}[\text{mid}] < \text{key}$ then
Return `BINARY_SEARCH_RECURSIVE(arr, mid + 1, high, key)`

Else
Return `BINARY_SEARCH_RECURSIVE(arr, low, mid - 1, key)`

Step 4: End If

Program:

```
#include<stdio.h>
#include<conio.h>
void bubblesort(int[],int);
int binsrch(int[],int,int,int);
void display(int[],int);
int i,j;
int main()
{
    int a[20],n,key,pos=-1;
    clrscr();
    printf("\n Enter the number of elements in array are:");
    scanf("%d",&n);
    printf("\n Enter %d elements in the array:",n);
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    printf("\n Enter the element to be searched:");
    scanf("%d",&key);
    bubblesort(a,n);
    printf("\n The sorted elements in the array are:");
    display(a,n);
    pos=binsrch(a,key,0,n-1);
    if(pos!=-1)
        printf("\n The Element %d is found in position %d",key,pos);
    else
        printf("\n Element not found");
    getch();
    return 0;
}
int binsrch(int a[],int key,int low,int high)
{
    int mid;
    while(low<=high)
    {
        mid=(low+high)/2;
        if(key<a[mid])
```



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

```
high=mid-1;
else if(key>a[mid])
low=mid+1;
DESIGN AND ANALYSIS OF ALGORITHMS LAB
Prepared by: Dept. of CSE, RGM CET Page 14
else
return mid;
}
return -1;
}
void bubblesort(int a[],int n)
{
int i,j,temp,excg=0;
int last=n-1;
for(i=0;i<n;i++)
{
excg=0;
for(j=0;j<last;j++)
{
if(a[j]>a[j+1])
{
temp=a[j];
a[j]=a[j+1];
a[j+1]=temp;
excg++;
}
}
}
if(excg==0)
return ;
else
last=last-1;
}
void display(int a[],int n)
{
int i;
for(i=0;i<n;i++)
printf("%d\t",a[i]);
}
```



Output:

```
Output

Enter the number of elements in array: 6

Enter 6 elements in the array: 11 12 17 25 8 3

Enter the element to be searched: 8

The sorted elements in the array are:
3   8   11  12  17  25

The element 8 is found at position 2
```

Conclusion: Binary Search quickly finds elements in a sorted array by dividing the search space repeatedly. The iterative method uses less memory, while the recursive method is simpler to implement. Both have a time complexity of $O(\log n)$, making them efficient for large datasets.

Viva questions :-

Q1. What is the basic working principle of Binary Search?

A1:

Binary Search works on the divide and conquer approach. It repeatedly divides the sorted array into two halves. At each step, it compares the middle element with the target value. If the middle element matches the target, the search is successful. If the target is smaller, the search continues in the left half. If the target is larger, the search continues in the right half until the element is found or the subarray becomes empty.

Q2. What are the two main ways to implement Binary Search?

A2:

Binary Search can be implemented in two ways: iterative and recursive. The iterative method uses a loop to narrow down the search range by updating low and high indices. The recursive method calls itself with updated low and high indices to search subarrays. Both methods require the array to be sorted. The iterative method uses less memory, while the recursive method is often simpler to write and understand.

Q3. What is the time complexity of Binary Search, and why?

A3:

The time complexity of Binary Search is $O(\log n)$ in best, average, and worst cases. This is because at each step, the algorithm halves the search space by comparing the middle element. By halving the array repeatedly, the



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

number of comparisons grows logarithmically with input size. This makes Binary Search very efficient compared to linear search. It quickly reduces the problem size, resulting in fast lookup times for large datasets.

Q4. How does the space complexity differ between iterative and recursive Binary Search?

A4:

The iterative Binary Search has a space complexity of $O(1)$ because it uses only a few variables for indexes and does not use additional memory. In contrast, the recursive Binary Search has a space complexity of $O(\log n)$ due to the recursion stack frames created for each recursive call. Each recursive call adds a new frame to the stack, so the depth of recursion affects space usage. Thus, iterative is more memory-efficient, while recursive can be easier to implement.

Q5. Why is Binary Search preferred over Linear Search for large datasets?

A5:

Binary Search is preferred because it is much faster than Linear Search on sorted data. Linear Search checks every element one by one, making its time complexity $O(n)$. Binary Search reduces the search space by half each time, achieving $O(\log n)$ time complexity. This leads to significant performance improvement on large datasets. Binary Search is widely used in databases, dictionaries, and search problems requiring quick lookups.



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur
DEPARTMENT OF INFORMATION TECHNOLOGY



PRACTICAL NO. 8

Aim: Write a program to find solution for knapsack problem using greedy method.

Theory: Greedy Algorithm for Fractional Knapsack

The Knapsack Problem is a classic optimization problem where you have to select items with given weights and values to maximize the total value in a knapsack of fixed capacity.

The Fractional Knapsack Problem allows taking fractions of items, unlike the 0/1 knapsack problem where items are either taken completely or not at all.

Greedy Strategy:

The greedy method works by choosing items with the highest value-to-weight ratio first, as these give the most value per unit weight. Items are sorted based on this ratio, and selected until the knapsack is full.

This strategy does not work for 0/1 knapsack, but gives optimal result for fractional knapsack. This approach is efficient and has a time complexity of $O(n \log n)$ due to sorting.

Example:

Let:

Items:

Value = [60, 100, 120]

Weight = [10, 20, 30]

Capacity = 50

Ratios:

$60/10 = 6$



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

■

$$100/20 = 5$$

$$120/30 = 4$$

Sorted order: Item 1, Item 2, Item 3

Greedy picks:

Item 1: Full (10kg) → Value = 60

Item 2: Full (20kg) → Value = 100

Item 3: Only $20/30 = 2/3$ of it → Value = 80

$$\text{Total Value} = 60 + 100 + 80 = 240$$

Algorithm:

Fractional Knapsack Using Greedy Method

Input:

n: Number of items

W: Maximum weight capacity of the knapsack

values[]: Array of item values

weights[]: Array of item weights

☞ Steps:

1. For each item, calculate value-to-weight ratio:

$$\text{ratio}[i] = \text{values}[i] / \text{weights}[i]$$

2. Sort items in descending order of their ratio[i].

3. Initialize:

$$\text{totalValue} = 0$$

$$\text{remainingWeight} = W$$



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur
DEPARTMENT OF INFORMATION TECHNOLOGY



4. For each item in sorted order:

If $\text{weights}[i] \leq \text{remainingWeight}$:

Take the whole item

$\text{totalValue} += \text{values}[i]$

$\text{remainingWeight} -= \text{weights}[i]$

Else:

Take fraction of the item

$\text{totalValue} += \text{values}[i] * (\text{remainingWeight} / \text{weights}[i])$

$\text{remainingWeight} = 0$

Break the loop

5. Return totalValue

Program:

```
#include<stdio.h>
#include<conio.h>
void readf();
void knapsack(int,int);
void dsort(int n);
void display(int);
int p[20],w[20],n,m;
double x[20],d[20],temp,res=0.0,sum=0.0;
void readf()
{
    int m,n,i;
    printf("\n Enter the no of Profits and weights:");
    scanf("%d",&n);
    printf("\n Enter the Maximum Capacity of the Knapsack:");
    scanf("%d",&m);
    printf("\n Enter %d profits of the weights:",n);
```



```
for(i=0;i<n;i++)
scanf("%d",&p[i]);
printf("\n Enter %d Weights:",n);
for(i=0;i<n;i++)
scanf("%d",&w[i]);
for(i=0;i<n;i++)
d[i]=(double)p[i]/w[i];
dsort(n);
knapsack(m,n);
display(n);
}
void dsort(int n)
{
int i,j,t;
for(i=0;i<n;i++)
{
for(j=0;j<n-1;j++)
{
if(d[j]<d[j+1])
{
temp=d[j];
d[j]=d[j+1];
d[j+1]=temp;
t=p[j];
DESIGN AND ANALYSIS OF ALGORITHMS LAB
Prepared by: Dept. of CSE, RGM CET Page 17
p[j]=p[j+1];
p[j+1]=t;
t=w[j];
w[j]=w[j+1];
w[j+1]=t;
}
}
}
}
void display(int n)
{
int i,m;
printf("\n The Required Optimal solution is:\n");
printf("Profits Weights Xvalue\n");
for(i=0;i<n;i++)
{
printf("%d\t%d\t%f\n",p[i],w[i],x[i]);
```



```
sum=sum+(p[i]*x[i]);
res=res+(w[i]*x[i]);
}
printf("\n The Total Resultant Profit is:%f",sum);
printf("\n The total resultant Weight into the knapsack is:%f",res);
}
void knapsack(int m,int n)
{
    int i,cu=m;
    for(i=0;i<n;i++)
    {
        x[i]=0.0;
    }
    for(i=0;i<n;i++)
    {
        if(w[i]<cu)
        {
            x[i]=1.0;
            cu=cu-w[i];
        }
        else
            break;
    }
    if(i<=n)
    {
        x[i]=(double)cu/w[i];
    }
}
```

DESIGN AND ANALYSIS OF ALGORITHMS LAB

Prepared by: Dept. of CSE, RGM CET Page 18

```
int main()
{
    clrscr();
    readf();
    getch();
    return 0;
}
```

Output:



Output

```
Enter the number of items: 3
Enter the Maximum Capacity of the Knapsack: 35
Enter 3 profits: 12 18 7
Enter 3 weights: 28 8 19

The Required Optimal solution is:
Profit  Weight  Xvalue
18      8      1.00
12      28     0.96
7       19     0.00

Total Profit = 29.57
Total Weight in Knapsack = 35.00
```

Conclusion: The greedy algorithm provides an efficient and optimal solution for the Fractional Knapsack Problem by selecting items based on the highest value-to-weight ratio. It works effectively when items can be divided, ensuring maximum profit with minimal computation. However, for problems like the 0/1 Knapsack, where items cannot be broken, the greedy method may fail to find the best solution. Thus, while the greedy approach is simple and fast, its applicability is limited to certain types of problems.

Viva questions :-

Q1. What is the main difference between 0/1 knapsack and fractional knapsack problems?

Ans: In 0/1 knapsack, you can take the whole item or leave it. In fractional knapsack, you can take any fraction of an item.

Q2. Why is the greedy algorithm not suitable for 0/1 knapsack?

Ans: Because greedy choices may miss the optimal combination of items in 0/1 knapsack. It only works optimally when fractional parts are allowed.

Q3. What is the value-to-weight ratio and how is it used in the greedy approach?

Ans: It is the ratio of an item's value to its weight. Items with higher ratios are selected first to maximize total value.



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

■

Q4. What is the time complexity of the greedy solution for fractional knapsack?

Ans: The time complexity is $O(n \log n)$ due to sorting items based on value-to-weight ratio.

--

Q5. How does the greedy algorithm fill the knapsack in the fractional problem?

Ans: It picks the item with the highest ratio and fills the knapsack fully or partially until the capacity is reached.

PRACTICAL NO. 9

Aim: Write a program to find minimum cost spanning tree using Prim's Algorithm.

Theory: What is a Spanning Tree?

A spanning tree of a graph is a subgraph that:

1. Connects all vertices.
2. Has no cycles.
3. Uses $V - 1$ edges, where V is the number of vertices.

What is a Minimum Spanning Tree (MST)?

:A Minimum Spanning Tree (MST) is a spanning tree with the lowest total weight among all possible spanning trees.

Prim's Algorithm - Explanation:

1. Prim's Algorithm is a greedy algorithm that:

1. Starts from any vertex.
2. At each step, selects the lowest weight edge connecting the already included part of the MST to an outside vertex.
3. Repeats until all vertices are included in the MST.



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

Algorithm:

Step 1: Initialize:

key[] = ∞ (for all vertices)

visited[] = false (to track included vertices)

parent[] = -1 (to store MST)

key[0] = 0 (starting vertex)

Step 2: Repeat V - 1 times:

Pick the unvisited vertex with the smallest key[].

Mark it as visited.

Update the key[] and parent[] for adjacent vertices if a smaller weight is found.

Step 3: Print the edges using parent[].

Program:

```
#include<stdio.h>
#include<conio.h>
int n,cost[10][10],temp,nears[10];
void readv();
void primsalg();
void readv()
{
    int i,j;
    printf("\n Enter the No of nodes or vertices:");
    scanf("%d",&n);
    printf("\n Enter the Cost Adjacency matrix of the given graph:");
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            scanf("%d",&cost[i][j]);
            if((cost[i][j]==0) && (i!=j))
            {
                cost[i][j]=999;
            }
        }
    }
}
void primsalg()
{
    int k,l,min,a,t[10][10],u,i,j,mincost=0;
    min=999;
```



```
for(i=1;i<=n;i++) //To Find the Minimum Edge E(k,l)
{
for(u=1;u<=n;u++)
{
if(i!=u)
{
if(cost[i][u]<min)
{
min=cost[i][u];
k=i;
l=u;
}
}
}
}
```

DESIGN AND ANALYSIS OF ALGORITHMS LAB

Prepared by: Dept. of CSE, RGM CET Page 20

```
}
t[1][1]=k;
t[1][2]=l;
printf("\n The Minimum Cost Spanning tree is...");
printf("\n(%d,%d)-->%d",k,l,min);
for(i=1;i<=n;i++)
{
if(i!=k)
{
if(cost[i][l]<cost[i][k])
{
nears[i]=l;
}
else
{
nears[i]=k;
}
}
}
nears[k]=nears[l]=0;
mincost=min;
for(i=2;i<=n-1;i++)
{
j = findnextindex(cost,nears);
t[i][1]=j;
t[i][2]=nears[j];
printf("\n(%d,%d)-->%d",t[i][1],t[i][2],cost[j][nears[j]]);
```




```
mincost=mincost+cost[j][nears[j]];
nears[j]=0;
for(k=1;k<=n;k++)
{
if(nears[k]!=0 && cost[k][nears[k]]>cost[k][j])
{
nears[k]=j;
}
}
}
printf("\n The Required Mincost of the Spanning Tree is:%d",mincost);
}
int findnextindex(int cost[10][10],int nears[10])
{
int min=999,a,k,p;
for(a=1;a<=n;a++)
{
p=nears[a];
if(p!=0)
DESIGN AND ANALYSIS OF ALGORITHMS LAB
Prepared by: Dept. of CSE, RGM CET Page 21
{
if(cost[a][p]<min)
{
min=cost[a][p];
k=a;
}
}
}
return k;
}
void main()
{
clrscr();
readv();
primsalg();
getch();
}
```

Output:



Output

```
Enter the number of nodes or vertices: 4

Enter the Cost Adjacency Matrix of the given graph:
1 22 3 4
11 2 46 34
21 32 5 8
15 64 9 10

The Minimum Cost Spanning Tree is:
(1, 3) --> 3
(4, 3) --> 9
(2, 1) --> 11

The Required Min-Cost of the Spanning Tree is: 23
```

Conclusion: Prim's Algorithm is used to find a Minimum Cost Spanning Tree (MST). It ensures that the total weight of the selected edges is minimum, while still connecting all vertices.

Viva questions:

1. What is a Minimum Spanning Tree (MST)?

Answer:

A Minimum Spanning Tree is a subgraph of a connected, undirected graph that includes all the vertices with the minimum possible total edge weight and contains no cycles.

2. What is the difference between Prim's and Kruskal's Algorithm?

Answer:

Prim's Algorithm	Kruskal's Algorithm
Starts from one vertex	Starts from all edges
Adds the nearest vertex	Adds the smallest edge
Uses a priority queue	Uses Disjoint Set / Union-Find
Works better for dense graphs	Works better for sparse graphs

3. What data structures are used in Prim's Algorithm?

Answer:

Prim's Algorithm typically uses:

An array for tracking visited vertices,



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

■
A key[] array for minimum weights,
A parent[] array to store MST edges,
In advanced versions, a priority queue (min heap) is used.

4. What is the time complexity of Prim's Algorithm?

Answer:

Using adjacency matrix: $O(V^2)$

Using adjacency list + min heap: $O(E \log V)$

Where V = number of vertices, E = number of edges.

5. What happens if the graph is disconnected?

Answer:

If the graph is disconnected, no spanning tree exists, and Prim's Algorithm cannot generate an MST. The algorithm assumes the graph is connected.

PRACTICAL NO. 10

Aim: Write a program to find minimum cost spanning tree using Kruskal's Algorithm.

Theory:

Kruskal's Algorithm

Kruskal's Algorithm is a greedy algorithm that finds an MST by always picking the edge with the smallest weight that doesn't form a cycle.

Key Concepts:

Spanning Tree: A subgraph that connects all vertices with the minimum number of edges ($n - 1$ edges for n vertices) and no cycles.

Minimum Spanning Tree (MST): A spanning tree with the smallest possible total edge weight.

Greedy Approach: Choose the lowest weight edge that connects two different trees (components).

Algorithm:

Step 1:- Sort all edges in non-decreasing order of their weights.

Step 2:- Initialize MST as empty.



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

Step 3:-Use Disjoint Set (Union-Find) to keep track of connected components.

Step 4:-For each edge in sorted list:

If including the edge does not form a cycle, add it to the MST.

Union the sets of the two vertices.

Step 5:-Repeat until MST has $(V - 1)$ edges.

KRUSKAL(G):

MST = []

sort edges of G by weight

initialize disjoint set DSU for all vertices

for edge (u, v) in sorted edges:

if DSU.find(u) != DSU.find(v):

MST.add((u, v))

DSU.union(u, v)

return MST

Program:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
int i,j,k,a,b,u,v,n,ne=1;
int min,mincost=0,cost[9][9],parent[9];
int find(int);
int uni(int,int);
void main()
{
    clrscr();
    printf("\n\tImplementation of Kruskal's algorithm\n");
    printf("\nEnter the no. of vertices:");
    scanf("%d",&n);
    printf("\nEnter the cost adjacency matrix:\n");
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            scanf("%d",&cost[i][j]);
            if(cost[i][j]==0)
                cost[i][j]=999;
        }
    }
    printf("The edges of Minimum Cost Spanning Tree are\n");
    while(ne < n)
    {
        for(i=1,min=999;i<=n;i++)
        {
            for(j=1;j <= n;j++)
            {
```



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

```

    if(cost[i][j] < min)
    {
        min=cost[i][j];
        a=u=i;
        b=v=j;
    }
    }
    }
    u=find(u);
    v=find(v);
    if(uni(u,v))
DESIGN AND ANALYSIS OF ALGORITHMS LAB
Prepared by: Dept. of CSE, RGM CET Page 23
    {
        printf("%d edge (%d,%d) =%d\n",ne++,a,b,min);
        mincost +=min;
    }
    cost[a][b]=cost[b][a]=999;
    }
    printf("\n\tMinimum cost = %d\n",mincost);
    getch();
    }
    int find(int i)
    {
        while(parent[i])
            i=parent[i];
        return i;
    }
    int uni(int i,int j)
    {
        if(i!=j)
        {
            parent[j]=i;
            return 1;
        }
        return 0;
    }
}
```



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur
DEPARTMENT OF INFORMATION TECHNOLOGY



Output:

Output

Implementation of Kruskal's algorithm

Enter the no. of vertices: 3

Enter the cost adjacency matrix:

2 3 4

11 8 5

15 1 7

The edges of Minimum Cost Spanning Tree are:

1 edge (3,2) = 1

2 edge (1,2) = 3

Minimum cost = 4



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur



DEPARTMENT OF INFORMATION TECHNOLOGY

□

Conclusion: Thus, Kruskal's Algorithm is an efficient way to construct the Minimum Spanning Tree using a greedy strategy and disjoint set (union-find) for cycle detection.

Viva questions :-

Q1:- Why use Union-Find in Kruskal's algorithm?

A: Union-Find efficiently tracks connected components to detect cycles. find() determines if two vertices share the same root; union() merges components only if they're separate—ensuring we add edges only when they don't form a cycle

Q2: What's the time complexity of this implementation?

A:

Sorting edges: $O(E \log E)$

Union-Find operations: roughly $O(E \alpha(V))$, where α is inverse-Ackermann (nearly constant).

Overall: $O(E \log E)$, which simplifies to $O(E \log V)$ since $E \approx V^2$ in dense graphs.

Q3: How does Kruskal differ from Prim's algorithm?

A:

Kruskal: Edge-centric—sorts all edges, picks lowest that doesn't form a cycle—works well on sparse graphs.

Prim: Starts from a root vertex and grows MST by adding nearest adjacent edge—more efficient on dense graphs with adjacency representations

Q4: What happens with disconnected graphs?

A: Kruskal's will produce a minimum spanning forest—a spanning tree for each connected component—by simply ending when no more usable edges exist

Q5: Can MST be non-unique? Under what conditions?

A: Yes. If the graph has multiple edges with the same weight, there could be more than one valid MST with equal total weight. The choice depends on how ties are broken during sorting



Lokmanya Tilak Jankalyan Shikshan Sanstha's
PRIYADARSHINI COLLEGE OF ENGINEERING, NAGPUR
(An Autonomous Institute affiliated to Rashtrasant Tukadoji Maharaj Nagpur
DEPARTMENT OF INFORMATION TECHNOLOGY



■
Aim:

Theory:

Algorithm:

Program:

Output:

Screenshot of Output

Conclusion: