PRACTICAL NO. 1

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

Theory:
Design and Analysis of Algorithms (DAA):
Design and Analysis of Algorithms is a branch of computer science concerned with:
1. Designing efficient algorithms to solve computational problems.
2. Analyzing their performance in terms of **time** and **space complexity**.

It ensures that algorithms not only work correctly but also perform efficiently for large input sizes.

Operation Count:
In DAA, analyzing how an algorithm performs with different input sizes is crucial.
Operation counting is a fundamental method used to measure performance by counting the number of basic operations such as:
- Arithmetic operations: +, -, *, /
- Assignment operations: =
- Comparison operations: ==, <, >, !=
- Logical operations: AND, OR, NOT
- Array access or indexing: A[i]

Each of these is treated as a **basic operation** with a cost of 1.

Importance of Operation Count:
- To understand how efficient an algorithm is.
- To compare different algorithms solving the same problem.
- To estimate how performance changes as input size increases.
- To derive **asymptotic complexities** like *$O(n)$*, *$O(n^2)$*, *$O(\log n)$*, etc.

Pseudo Code:
```
for i = 1 to n
   for j = 1 to n
      A[i][j] = i + j
```

Operation Count Explanation:
- The algorithm has two nested loops, each running from **1 to n**.
- Inside the inner loop, two basic operations occur:
  - One addition operation: i + j
  - One assignment operation: A[i][j] = ...

Total Iterations:
- Outer loop runs n times.
- Inner loop runs n times for each iteration of the outer loop.
- Total inner loop executions = $n \times n = n^2$.

Total Operations:
- Addition operations: n^2 times
- Assignment operations: n^2 times

✓ **Total basic operations = $2 \times n^2$**

Time Complexity:

Since the number of basic operations increases proportionally with n^2,
the **time complexity** is:

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

This is known as **quadratic time complexity**.

Purpose of Operation Count:

- Helps analyze how an algorithm scales with input size.
- Useful for comparing the performance of different algorithms.
- Provides a foundation for understanding **asymptotic analysis**.
- Helps predict the **best**, **worst**, and **average-case** behavior of algorithms.

Algorithm:

```
1.  Start
2.  Initialize addition ← 0, assignment ← 0
3.  Create a 2D array A of size n × n
4.  Repeat for i = 0 to n - 1
5.  Repeat for j = 0 to n - 1
6.  A[i][j] ← i + j
7.  Increment addition ← addition + 1 // Counting addition operation
8.  Increment assignment ← assignment + 1 // Counting assignment operation
9.  Print addition, assignment
10. Print total_operations ← addition + assignment
11. Stop
```

Program (C Language):

```
#include <stdio.h>

void main() {
  int count = 0, sum = 0, n, i, a[50];

  count = count + 1;
  printf("\nEnter the number of elements (n): ");
  scanf("%d", &n);
  count = count + 1;

  printf("\nEnter %d values to sum:\n", 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("\nThe sum of %d values is: %d", n, sum);
    printf("\nTotal operation count = %d\n", count);
}
```

Output:

```
Enter the number of elements (n): 3
Enter 3 values to sum:
1
2
3
The sum of 3 values is: 6
Total operation count = 12
```

Conclusion:

Operation count is a fundamental method to **evaluate and understand the efficiency** of an algorithm.
It provides a quantitative measure of performance as input size increases.
In this example, the nested loop resulted in a **quadratic time complexity (O(n^2))**, showing that the algorithm becomes slower for large values of *n*.
By learning operation count, we gain the ability to:

- Predict algorithm performance,
- Optimize code, and
- Make informed decisions when choosing or designing 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 well-known **comparison-based sorting algorithm**. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list becomes completely sorted.
The algorithm is called *Bubble Sort* because smaller elements "bubble up" to the top (beginning of the array), while larger elements gradually "sink" to the bottom (end of the array) after each pass.
Although Bubble Sort is easy to understand and implement, it is inefficient for large datasets since both its **average** and **worst-case time complexities** are **$O(n^2)$**, where *n* is the number of elements.

Working of Bubble Sort:

1. The algorithm works in multiple passes.
   - After the first pass, the largest element moves to the end (its correct position).
   - After the second pass, the second-largest element moves to the second-last position, and so on.
2. In each pass, only the unsorted part of the array is processed.
   After *k* passes, the last *k* elements are already sorted.
3. During a pass, adjacent elements are compared and swapped if the left element is greater than the right element.
   Repeating this ensures that the largest unsorted element moves to its correct position in each iteration.

Algorithm:
```
Step 1: Start
Step 2: Repeat for i = 0 to n - 1
Step 3: Compare adjacent elements:
    If the left element > right element → swap them
Step 4: After each pass, the largest unsorted element moves to its correct position
Step 5: Repeat until the array is sorted
Step 6: Stop
```

Program (C Language):
```c
#include <stdio.h>

void bubbleSort(int[], int);
void display(int[], int);

int main() {
  int a[20], n, i;

  printf("\nEnter the number of elements in the array: ");
  scanf("%d", &n);

  printf("\nEnter %d elements:\n", n);
  for (i = 0; i < n; i++)
    scanf("%d", &a[i]);

  bubbleSort(a, n);

  printf("\nThe sorted elements in the array are:\n");
  display(a, n);
```

```
    return 0;
}

void bubbleSort(int a[], int n) {
    int i, j, temp, excg;

    for (i = 0; i < n - 1; i++) {
        excg = 0;
        for (j = 0; j < n - i - 1; j++) {
            if (a[j] > a[j + 1]) {
                temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
                excg = 1;
            }
        }
        // If no exchange occurs, array is already sorted
        if (excg == 0)
            break;
    }
}

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

Output:

```
Enter the number of elements in the array: 5
Enter 5 elements:
23
15
42
8
16

The sorted elements in the array are:
8	15	16	23	42
```

Conclusion:

In the study of **Design and Analysis of Algorithms (DAA)**, Bubble Sort serves as a fundamental example of a simple sorting algorithm. It helps to understand basic algorithm design concepts, performance analysis, and time complexity behaviour. Though not efficient for large datasets, it is useful for learning the foundational principles of sorting algorithms.

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 sort playing cards in their hands — starting with an empty hand and inserting one card at a time into its correct position among the already sorted cards.
Similarly, **Insertion Sort** builds the sorted portion of an array one element at a time, placing each new element in its proper position relative to the sorted section.
Although not efficient for large datasets, Insertion Sort is very useful for:

- **Small or nearly sorted datasets**
- **Embedded systems** (due to low memory use)
- **Educational purposes** (to understand sorting logic)
- **Hybrid algorithms** like *Timsort* (used in Python and Java), where it's applied on small subarrays

Key Features:

- Simple and intuitive
- Stable sorting algorithm
- Works efficiently on small or nearly sorted data
- Performs sorting **in-place** (requires no extra memory)

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

Program (C Language):
```
#include <stdio.h>

void insertionSort(int[], int);
void display(int[], int);

int main() {
   int a[20], n, i;

   printf("\nEnter the number of elements in the array: ");
   scanf("%d", &n);

   printf("\nEnter %d elements in the array:\n", n);
   for (i = 0; i < n; i++)
      scanf("%d", &a[i]);

   insertionSort(a, n);

   printf("\nThe sorted elements in the array are:\n");
```

```
    display(a, n);

    return 0;
}

void insertionSort(int a[], int n) {
    int i, j, key;

    for (i = 1; i < n; i++) {
        key = a[i];
        j = i - 1;

        // Move elements of a[0..i-1], that are greater than key,
        // to one position ahead of their current position
        while (j >= 0 && a[j] > key) {
            a[j + 1] = a[j];
            j--;
        }

        a[j + 1] = key;
    }
}

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

Output:

```
Enter the number of elements in the array: 6
Enter 6 elements in the array:
12
4
9
3
15
7
The sorted elements in the array are:
3	4	7	9	12	15
```

Conclusion:

Insertion Sort is a simple yet powerful sorting technique best suited for **small** or **nearly sorted datasets**. It is **intuitive**, **stable**, and **space-efficient**, though not ideal for large datasets due to its **quadratic time complexity**. Its simplicity and minimal overhead make it valuable for understanding **basic sorting logic**, **comparisons**, and **algorithmic design** in computer science.

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 **divide-and-conquer** sorting algorithm developed by **Tony Hoare in 1959**. It works by selecting a **pivot element** from the array and partitioning the other elements into two subarrays:

- One containing elements **less than the pivot**
- Another containing elements **greater than or equal to the pivot**

These subarrays are then **recursively sorted** using the same approach.
As a result, each pivot element is placed in its correct sorted position in the array.
Quick Sort is extremely fast in practice and is often used in standard libraries and real-world applications due to its:

- **Average time complexity:** O(n log n)
- **In-place sorting nature** (requires little extra memory)
- **High efficiency on large datasets**

However, its **worst-case time complexity** is **O(n^2)**, which occurs when poor pivot choices lead to highly unbalanced partitions.
This can be avoided by using techniques like **randomized pivot selection** or the **median-of-three** method.

Algorithm:

QUICK_SORT(arr, low, high)

```
Step 1: If low < high then
Step 2:     pivotIndex ← PARTITION(arr, low, high)
Step 3:     QUICK_SORT(arr, low, pivotIndex - 1)
Step 4:     QUICK_SORT(arr, pivotIndex + 1, high)
End If
```

PARTITION(arr, low, high)

```
Step 1: pivot ← arr[low]
Step 2: i ← low + 1
Step 3: j ← high
Step 4: Repeat while i ≤ j
    a. While i ≤ high AND arr[i] ≤ pivot → i ← i + 1
    b. While arr[j] > pivot → j ← 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 (C Language):

```c
#include <stdio.h>

void quickSort(int[], int, int);
int partition(int[], int, int);

void quickSort(int a[], int first, int last) {
   int j;
   if (first < last) {
      j = partition(a, first, last + 1);
      quickSort(a, first, j - 1);
      quickSort(a, j + 1, last);
   }
}

int partition(int a[], int first, int last) {
    int pivot = a[first];
    int i = first;
    int j = last;
    int temp;

    do {
        do {
            i++;
        } while (a[i] < pivot);

        do {
            j--;
        } while (a[j] > pivot);

        if (i < j) {
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    } while (i < j);

    a[first] = a[j];
    a[j] = pivot;
    return j;
}

int main() {
    int a[40], i, n;

    printf("\nEnter the number of elements: ");
    scanf("%d", &n);

    printf("\nEnter %d elements:\n", n);
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);

    quickSort(a, 0, n - 1);

    printf("\nThe elements after sorting are:\n");
    for (i = 0; i < n; i++)
        printf("%d\t", a[i]);

    printf("\n");
    return 0;
}
```

Output:

```
Enter the number of elements: 6
Enter 6 elements:
```

```
45
12
67
23
9
50

The elements after sorting are:
9    12    23    45    50    67
```

Conclusion:
Quick Sort is an **efficient**, **in-place**, and **widely used** sorting algorithm.
It outperforms simpler algorithms like **Bubble Sort** and **Insertion Sort**, especially on **large datasets**.
However, its efficiency depends heavily on **pivot selection**.
Using strategies like **randomized pivot** or **median-of-three pivot selection**, Quick Sort achieves **optimal performance** in most practical applications.
It remains one of the most important and commonly used sorting algorithms in computer science.

PRACTICAL NO: 5

Aim:

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

Theory:

1. **Introduction:**
   Finding the maximum and minimum elements in a list is one of the most basic yet essential problems in algorithm design.
   It introduces the concept of **linear search** and helps in understanding **time complexity** and **efficiency** in algorithm analysis.

The approach involves:

- Initializing the first element as both max and min.
- Scanning the rest of the array.
- Updating max when a larger element is found.
- Updating min when a smaller element is found.

---

2. **Set of Integers:**
   A set of integers consists of whole numbers which can be:
   - Positive numbers (e.g., 10, 23)
   - Negative numbers (e.g., -5, -8)
   - Zero (0)

These are stored in **arrays** for easy processing in programming.

---

3. **Array:**
   An **array** is a linear data structure used to store multiple values of the same type.
   It provides **efficient element access** and **sequential storage**.

Example:

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

Here:

- arr[0] = 12
- arr[1] = -4
- arr[2] = 0
- arr[3] = 99
- arr[4] = 33

Arrays make it easier to perform operations such as finding **maximum** and **minimum** values.

---

4. **Input Representation:**
   - Input number n → number of elements in the array.
   - Input arr[n] → list of integer values.

The algorithm scans through the array and updates the maximum and minimum values accordingly.

---

5. **Time and Space Complexity:**

Complexity Type	**Explanation**	**Value**
Time Complexity	Every element is checked once	**O(n)**
Space Complexity	Only a few variables are used	**O(1)**

This makes it **efficient** for large datasets and memory-constrained systems.

Algorithm:
Input: Array arr[] of size n
Output: Maximum and Minimum values

```
1. Start
2. Input number of elements n
3. Input n integers into array arr[]
4. Initialize
      o  max = arr[0]
      o  min = arr[0]
5. Repeat for i = 1 to n - 1:
      o  If arr[i] > max, then max = arr[i]
      o  If arr[i] < min, then min = arr[i]
6. Print max and min
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 the array: ");
   scanf("%d", &n);

   printf("\n Enter %d elements in the array:\n", n);
   for(i = 0; i < n; i++)
      scanf("%d", &a[i]);

   printf("\n The Elements in the array are:\n");
   for(i = 0; i < n; i++)
      printf("%d\n", a[i]);

   // Call the recursive minmax function
   minmax(0, n-1, a[0], a[0]);

   printf("\n The Minimum Element in the list is: %d", fmin);
   printf("\n The Maximum Element in the list 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;
        else
            fmax = hmax;

        if(gmin < hmin)
            fmin = gmin;
        else
            fmin = hmin;
    }
}
```

Sample Output:

```
Enter the number of elements in array: 5
Enter 5 elements in the array:
10
4
25
```

```
7
15

The Elements in the array are:
10
4
25
7
15

The Minimum Element of the list is: 4
The Maximum Element of the list is: 25
```

Conclusion:
In this practical, we implemented an algorithm to find the **maximum** and **minimum** elements in a given set of integers.
The algorithm efficiently traverses the array only once, achieving **O(n)** time complexity and **O(1)** space complexity.
This exercise strengthens the understanding of **divide and conquer**, **recursion**, and **performance analysis** in algorithm design.

PRACTICAL NO: 6

Aim:
Write a program to perform **Merge Sort** on the given list of integer values.

Theory:
Merge Sort is an efficient, stable, and comparison-based sorting algorithm that follows the **divide and conquer** approach.
The algorithm works in three main steps:

1. **Divide:** Split the array into two halves until each subarray contains only one element.
2. **Conquer:** Sort the subarrays recursively.
3. **Combine:** Merge the sorted subarrays to form a single sorted array.

It maintains stability (preserving the order of equal elements) and provides **O(n log n)** performance in all cases — best, average, and worst.
Although Merge Sort requires additional memory for temporary arrays, its predictable performance makes it suitable for **large datasets** and applications requiring **stable sorting**.
Key Points:

- **Developed using:** Divide and Conquer strategy
- **Time Complexity:** O(n log n)
- **Space Complexity:** O(n)
- **Stability:** Stable
- **In-place:** No (uses auxiliary space)

Merge Sort is widely used in sorting large data efficiently and forms the foundation of many advanced sorting algorithms (like **Timsort**).

Algorithm:
Input: Array arr[] of size n
Output: Sorted array in ascending order
Steps:

```
1. n1 ← mid - left + 1
2. n2 ← right - mid
3. Create two temporary arrays:
     o Left[0 … n1 - 1]
     o Right[0 … n2 - 1]
4. Copy data into the temporary arrays:
     o For i = 0 to n1 - 1: Left[i] ← arr[left + i]
     o For j = 0 to n2 - 1: Right[j] ← arr[mid + 1 + j]
5. Initialize i = 0, j = 0, k = left
6. Repeat while i < n1 and j < n2
   a. If Left[i] ≤ Right[j], then arr[k] ← Left[i], i ← i + 1
   b. Else arr[k] ← Right[j], j ← j + 1
   c. k ← k + 1
7. Copy remaining elements of Left[], if any
8. Copy remaining elements of Right[], if any
9. Stop
```

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
   {
      for(k = h; k <= mid; k++)
      {
         b[i] = a[k];
         i++;
      }
   }

   for(k = low; k <= high; k++)
   {
      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:\n");
  for(i = 0; i < n; i++)
    scanf("%d", &a[i]);

  printf("\n The Elements before sorting are:\n");
  for(i = 0; i < n; i++)
    printf("%d\t", a[i]);

  mergesort(a, 0, n - 1);

  printf("\n\n The Elements after sorting are:\n");
  for(i = 0; i < n; i++)
    printf("%d\t", a[i]);

  getch();
}
```

Sample Output:

```
Enter the size of the elements to be sorted: 6
Enter the elements to sort:
45
12
67
23
9
34

The Elements before sorting are:
45   12   67   23   9   34
```

```
The Elements after sorting are:
9   12   23   34   45   67
```

Conclusion:

In this experiment, we implemented the **Merge Sort** algorithm, which efficiently sorts lists by recursively dividing and merging sublists.
The algorithm ensures **O(n log n)** time complexity in all cases and maintains **stability**, preserving the order of equal elements.
Although it requires extra space, it provides **predictable performance** and is preferred for **large datasets** and applications requiring **stable sorting behavior**.
Merge Sort demonstrates the power of the **divide-and-conquer** strategy in algorithm design and analysis.

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 an efficient searching algorithm used to find the position of a target element in a **sorted array**.
Unlike **Linear Search**, which checks each element sequentially, Binary Search follows the **divide-and-conquer** principle — it divides the array into halves and reduces the search space by half in each step.

Working Principle:

1. Start with the entire sorted array.
2. Compare the **middle element** with the target value:
    - If equal → element found.
    - If the target is smaller → search the **left half**.
    - If the target is larger → search the **right half**.
3. Repeat until the element is found or the search range becomes empty.

Binary Search can be implemented in two ways:

- **Iterative (Non-Recursive)** — uses a loop.
- **Recursive** — uses function calls until the base condition is met.

Advantages:

- Significantly faster than Linear Search for large datasets.
- Time complexity: **O(log n)**.
- Commonly used in **databases, dictionaries, and indexing systems**.

Complexity Analysis:

Approach	Time Complexity	Space Complexity
Iterative	O(log n)	O(1)
Recursive	O(log n)	O(log n) (due to recursion stack)

Algorithm:
Iterative Binary Search
Input: Sorted array arr[], number of elements n, and search key key
Output: Index of element if found, else -1

```
1. low ← 0
2. high ← n - 1
3. Repeat while low ≤ high
   a. mid ← (low + high) / 2
   b. If arr[mid] == key → return mid
   c. Else if arr[mid] < key → low ← mid + 1
   d. Else → high ← mid - 1
4. Return -1 (element not found)
```

Recursive Binary Search
Input: Sorted array arr[], low, high, key
Output: Index of element if found, else -1

```
1. If low > high, return -1
2. mid ← (low + high) / 2
3. If arr[mid] == key, return mid
4. Else if arr[mid] < key, call BINARY_SEARCH_RECURSIVE(arr, mid + 1, high, key)
5. Else, call BINARY_SEARCH_RECURSIVE(arr, low, mid - 1, key)
```

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: ");
   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:\n");
   display(a, n);

   pos = binsrch(a, key, 0, n - 1);

   if(pos != -1)
      printf("\n\n The Element %d is found at position %d.", key, pos);
   else
      printf("\n\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])
         high = mid - 1;
```

```
        else if(key > a[mid])
            low = mid + 1;
        else
            return mid; // Element found
    }
    return -1; // Element not found
}

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)
            break;
        else
            last = last - 1;
    }
}

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

Sample Output:

```
Enter the number of elements in array: 6
Enter 6 elements in the array: 15 42 8 23 4 19
Enter the element to be searched: 23

The sorted elements in the array are:
4   8   15   19   23   42

The Element 23 is found at position 4.
```

Conclusion:

In this experiment, we implemented **Binary Search** using both **recursive** and **non-recursive** approaches.
Binary Search efficiently locates an element in a sorted array by repeatedly dividing the search interval in half.

- **Iterative Binary Search** uses loops, offering **O(1)** space complexity.
- **Recursive Binary Search** is easier to understand but uses **O(log n)** stack space.

Both provide **O(log n)** time complexity, making Binary Search significantly faster than Linear Search for large datasets.

PRACTICAL NO: 8

Aim:
Write a program to find the solution for the **Knapsack problem using the Greedy method**.

Theory:
The **Knapsack Problem** is a classic **optimization problem** in which we aim to maximize the total profit/value of items that can be placed in a knapsack with a fixed weight capacity.
There are two types of Knapsack problems:

1. **0/1 Knapsack:** Each item can be taken **completely or not at all**.
2. **Fractional Knapsack:** Items can be **divided** into fractions, allowing partial selection.

In this experiment, we focus on the **Fractional Knapsack Problem**, which can be efficiently solved using the **Greedy Algorithm**.

Greedy Strategy:
The **Greedy method** selects items based on the **highest value-to-weight ratio (profit per unit weight)** first.
Steps:

1. Compute the **ratio = profit/weight** for each item.
2. Sort items in **descending order** of this ratio.
3. Take items one by one until the knapsack is full:
    - If the full item can fit, take it entirely.
    - If not, take a **fraction** of it that fits the remaining capacity.
4. Stop when the knapsack is full.

This method works perfectly for the **fractional case**, giving an **optimal solution**.
For **0/1 Knapsack**, however, it may not give the best result.

Example:

Item	Value	Weight	Value/Weight
1	60	10	6
2	100	20	5
3	120	30	4

Knapsack Capacity = 50
Step-by-step selection:

- Take **Item 1 (10kg)** → Value = 60
- Take **Item 2 (20kg)** → Value = 100
- Take **2/3 of Item 3 (20/30kg)** → Value = 80

✅ **Total Value = 60 + 100 + 80 = 240**

Algorithm:
Fractional Knapsack using Greedy Method
Input:

- n: Number of items
- W: Maximum capacity of knapsack
- values[]: Profits/values of items
- weights[]: Weights of items

Steps:

1. For each item, calculate ratio:
   ratio[i] = values[i] / weights[i]
2. Sort items in descending order of ratio[i].

```
3. Initialize:
   totalValue = 0, remainingWeight = W
4. For each item (in sorted order):
      o If weights[i] <= remainingWeight → take full item.
      o Else → take fractional part:
        totalValue += values[i] * (remainingWeight / weights[i])
        and break.
5. Return totalValue.
```

Time Complexity: O(n log n) (due to sorting)
Space Complexity: O(n)

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 i;
  printf("\n Enter the number of items: ");
  scanf("%d", &n);

  printf("\n Enter the maximum capacity of the knapsack: ");
  scanf("%d", &m);

  printf("\n Enter %d profits of the items: ", n);
  for(i = 0; i < n; i++)
    scanf("%d", &p[i]);

  printf("\n Enter %d weights of the items: ", n);
  for(i = 0; i < n; i++)
    scanf("%d", &w[i]);

  // Calculate value/weight ratio
  for(i = 0; i < n; i++)
    d[i] = (double)p[i] / w[i];

  // Sort by descending ratio
  dsort(n);

  // Apply knapsack
  knapsack(m, n);
```

```
    // Display results
    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])
            {
                // Swap ratios
                temp = d[j];
                d[j] = d[j + 1];
                d[j + 1] = temp;

                // Swap profits
                t = p[j];
                p[j] = p[j + 1];
                p[j + 1] = t;

               // Swap weights
                t = w[j];
                w[j] = w[j + 1];
                w[j + 1] = t;
            }
        }
    }
}
void knapsack(int m, int n)
{
    int i;
    int cu = m;  // Remaining capacity

    for(i = 0; i < n; i++)
        x[i] = 0.0;  // Initialize fraction array

    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];
}
void display(int n)
{
    int i;
    printf("\n The Required Optimal Solution is:\n");
    printf("Profit\tWeight\tFraction Taken\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 Total Profit obtained: %f", sum);
    printf("\n Total Weight in Knapsack: %f", res);
}
int main()
{
    clrscr();
    readf();
    getch();
    return 0;
}
```

Sample Output:

```
Enter the number of items: 3
Enter the maximum capacity of the knapsack: 50
Enter 3 profits of the items: 60 100 120
Enter 3 weights of the items: 10 20 30

The Required Optimal Solution is:
Profit  Weight  Fraction Taken
60      10      1.000000
100     20      1.000000
120     30      0.666667

Total Profit obtained: 240.000000
Total Weight in Knapsack: 50.000000
```

Conclusion:

The **Greedy Algorithm** efficiently solves the **Fractional Knapsack Problem** by prioritizing items with the **highest value-to-weight ratio**, ensuring maximum profit.
It provides an **optimal solution** when items can be divided into fractions.
However, this approach **fails for 0/1 Knapsack**, where partial items are not allowed.
Thus, the **Greedy method** is ideal for problems where choices can be made in parts and an **optimal local decision leads to a globally optimal solution**.

PRACTICAL NO: 9

Aim:

Write a program to find the **Minimum Cost Spanning Tree (MST)** using **Prim's Algorithm**.

Theory:

What is a Spanning Tree?

A **spanning tree** of a connected graph is a subgraph that:

1. Connects all vertices together.
2. Contains **no cycles**.
3. Has exactly **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 **smallest possible total edge weight** among all spanning trees of the graph.

It ensures that:

- All vertices are connected.
- The total cost (sum of edge weights) is minimized.
- No cycles are formed.

Prim's Algorithm — Explanation

Prim's Algorithm is a **greedy algorithm** that builds the MST step by step.

It starts with any vertex and repeatedly selects the **minimum weight edge** that connects a vertex in the MST to a vertex outside it.

Steps of Prim's Algorithm:

1. **Initialize:**
   - key[] = ∞ for all vertices
   - visited[] = false
   - parent[] = -1
   - Start from vertex 0 → key[0] = 0
2. **Repeat (V - 1) times:**
   - Select the unvisited vertex with the **minimum key value**.
   - Mark this vertex as **visited**.
   - Update key values for adjacent vertices:
     If the edge weight is smaller than the existing key, update the key and parent.
3. **After all iterations:**
   - The parent[] array represents the edges of the MST.
4. **Compute total cost** by summing the edge weights included in the MST.

Time Complexity:

- **O(V^2)** for adjacency matrix implementation
- **O(E log V)** for adjacency list with min-heap (optimized version)

Program:

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

int n, cost[10][10], temp, nears[10];
```

```
void readv();
void primsalg();
int findnextindex(int cost[10][10], int nears[10]);

void readv()
{
   int i, j;
   printf("\n Enter the number of vertices: ");
   scanf("%d", &n);

   printf("\n Enter the Cost Adjacency Matrix of the graph:\n");
   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;  // Large value to represent no direct edge
      }
   }
}

void primsalg()
{
   int k, l, min, a, t[10][10], u, i, j, mincost = 0;

   // Find the first minimum edge (starting point)
   min = 999;
   for(i = 1; i <= n; i++)
   {
      for(u = 1; u <= n; u++)
      {
         if(i != u && cost[i][u] < min)
         {
            min = cost[i][u];
            k = i;
            l = u;
         }
      }
   }

   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 += 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\n The Minimum Cost 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)
        {
            if(cost[a][p] < min)
            {
                min = cost[a][p];
                k = a;
            }
        }
    }
    return k;
}
```

```
void main()
{
   clrscr();
   readv();
   primsalg();
   getch();
}
```

Sample Input:

```
Enter the number of vertices: 4

Enter the Cost Adjacency Matrix:
0 10 6 5
10 0 0 15
6 0 0 4
5 15 4 0
```

Sample Output:

```
The Minimum Cost Spanning Tree is:
(3, 4) --> 4
(1, 3) --> 6
(1, 2) --> 10

The Minimum Cost of the Spanning Tree is: 20
```

Explanation (Dry Run):

1. Start with the smallest edge (3,4) → cost = 4
2. Add next smallest edge that connects a new vertex → (1,3) → cost = 6
3. Add (1,2) → cost = 10
4. Total cost = 4 + 6 + 10 = **20**

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.

Here's your formatted and polished **PRACTICAL NO. 10** for your **Design and Analysis of Algorithms Lab Record** 👇

PRACTICAL NO. 10

Aim:

Write a program to find the Minimum Cost Spanning Tree using **Kruskal's Algorithm**.

Theory:

Kruskal's Algorithm:

Kruskal's Algorithm is a **greedy algorithm** used to find a **Minimum Spanning Tree (MST)** for a connected, undirected, weighted graph. It ensures that the total cost of the spanning tree is minimized while avoiding cycles.

Key Concepts:

- **Spanning Tree:**
  A subgraph that connects all vertices together without forming any cycles and has exactly **(V - 1)** edges for **V** vertices.
- **Minimum Spanning Tree (MST):**
  A spanning tree whose sum of edge weights is the smallest possible among all spanning trees.
- **Greedy Approach:**
  Kruskal's algorithm picks the **smallest weight edge** that does not create a cycle, repeating the process until all vertices are connected.

Algorithm (KRUSKAL(G)):

1. Sort all edges in **non-decreasing order** of their weights.
2. Initialize an empty MST and a disjoint set (Union-Find) to track connected components.
3. For each edge (u, v) in sorted order:
   - If including the edge does not form a cycle, **add it to MST**.
   - Perform **Union(u, v)** to merge sets.
4. Repeat until MST contains **V - 1** edges.
5. 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 number 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;  // Represent infinity
        }
    }

    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++)
            {
                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))
        {
            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;
}
```

Output:

```
Implementation of Kruskal's Algorithm

Enter the number of vertices: 4
Enter the cost adjacency matrix:
0 5 8 0
5 0 10 15
8 10 0 20
0 15 20 0

The edges of Minimum Cost Spanning Tree are:
1 edge (1,2) = 5
2 edge (1,3) = 8
3 edge (2,4) = 15

Minimum Cost = 28
```

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.