College of Technology and Engineering, MPUAT, Udaipur

Name – Abhishek Pandey Class – B.Tech III yr

Subject – Design & Analysis of Algo. (CS- 362) Semester – VI

Program – 1

Object - Write a program to implement Linear Search.

Code :

```
#include<bits/stdc++.h>
using namespace std;
void input_array(vector<int> &v, int size){
    for(int i=0; i<size; i++)
        cin >> v[i];
}

int linear_search(vector<int> v, int key){
    //linearly traverse each element of array and check if it is equal to search value if yes return the particular index
    for(int i=0; i<v.size(); i++)
        if(v[i] == key)
            return i;
    //after complete traversal, means search value not present return -1
    return -1;
}

int main()
{
    int n;
    cout << "Enter size of array : ";
    cin >> n;
    vector<int> arr(n);
    cout << "Enter array elements : ";
    input_array(arr, n);
    int target;
    cout << "Enter the element you want to search : ";
    cin >> target;
    int ans = linear_search(arr, target);
    if(ans == -1)
        cout << target << " is not present in the array." << endl;
    else
        cout << target << " is present in array at index " << ans << endl;

    return 0;
}
```

Output :

```
Enter size of array : 8
Enter array elements : 10 34 1 89 3 89 76 4
Enter the element you want to search : 3
3 is present in array at index 4
ap-73@AP:/mnt/A2A25781A257593D/Practical6th/DAA$
```

Program – 2

Object - Write a program to implement Binary Search.

Code :

```
#include<bits/stdc++.h>
using namespace std;
void input_array(vector<int> &v, int size)
{
    for(int i=0; i<size; i++)
        cin >> v[i];
}
int binary_search(vector<int> v, int key)
{
    int lo = 0;
    int hi = v.size()-1;
    while(lo <= hi)
    {
        int mid = lo + (hi - lo)/2;
        if(v[mid] == key) //element found
            return mid;
        if(v[mid] > key)//discard right of mid, move to left part of array
            hi = mid - 1;
        else
            lo = mid + 1;
    }
    return -1;//if element not found return -1
}

int main()
{
    int n;
    cout << "Enter size of array : ";
    cin >> n;
    vector<int> arr(n);
    cout << "Enter array elements (in sorted order) : ";
    input_array(arr, n);
    int target;
    cout << "Enter the element you want to search : ";
    cin >> target;
    int ans = binary_search(arr, target);
    if(ans == -1)
        cout << target << " is not present in the array." << endl;
```

```
    else
        cout << target << " is present in array at index " << ans << endl;

    return 0;
}
```

Output :

```
Enter size of array : 7
Enter array elements (in sorted order) : 1 7 9 10 14 21 33
Enter the element you want to search : 7
7 is present in array at index 1
ap-73@AP:/mnt/A2A25781A257593D/Practical6th/DAA$
```

Program – 3

Object - Write a program to implement BubbleSort.

Code :

```
#include<bits/stdc++.h>
using namespace std;

void bubble_sort(vector<int> &v){
   int size = v.size();
   // Loop through all array elements except last -> because the last element will always be in its correct position after each iteration of the inner loop.
   for(int i=0; i<size-1; i++){
      bool swapped = false; // to optimize if no swap was there in any one iteration of inner loop then already sorted
      for(int j=0; j<size-i-1; j++){
         if(v[j] > v[j+1]){
            swapped = true;
            swap(v[j], v[j+1]);
         }
      }
      if(swapped == false)
         break;
   }
}

int main(){
   vector<int> arr;
   int n;
   cout << "Enter the no of elements : ";
   cin >> n;
   cout << "Enter array elements : ";
   for(int i=0; i<n; i++){
      int x;
      cin>>x;
      arr.push_back(x);
   }
   cout << "\nArray before sorting - > ";
   for(int i=0; i<arr.size(); i++)
      cout << arr[i] << " ";
   //calling bubble sort function
   bubble_sort(arr);
   cout << "\n\nArray after sorting -> ";
```

```
    for(int i=0; i<arr.size(); i++)
        cout << arr[i] << " ";
    cout << endl;
    return 0;
}
```

Output :

```
Enter the no of elements : 8
Enter array elements : 2 8 4 90 12 78 0 34

Array before sorting - > 2 8 4 90 12 78 0 34

Array after sorting -> 0 2 4 8 12 34 78 90
ap-73@AP:/mnt/A2A25781A257593D/Practical6th/DAA$
```

Program – 4

Object - Write a program to implement Quick Sort.

Code :

```
#include<bits/stdc++.h>
using namespace std;

void quickSort(vector<int>& arr, int left, int right) {
    if (left >= right) {
        // Base case: the sub-array has zero or one elements
        return;
    }
    // Choose a pivot element from the sub-array
    int pivotIndex = left + (right - left) / 2;
    int pivot = arr[pivotIndex];

    // Partition the sub-array into two sub-arrays
    int i = left;
    int j = right;
    while (i <= j) {
        while (arr[i] < pivot) {
            i++;
        }
        while (arr[j] > pivot) {
            j--;
        }
        if (i <= j) {
            swap(arr[i], arr[j]);
            i++;
            j--;
        }
    }

    // Recursively apply quickSort to the sub-arrays
    if (left < j)
        quickSort(arr, left, j);
    if (i < right)
        quickSort(arr, i, right);
}

int main() {
    // Example usage of the quickSort function
```

```
    vector<int> arr = {5, 2, 9, 1, 5, 6};
    cout << "\nBefore Sorting -> ";
    for(int i=0; i<6; i++)
        cout << arr[i] << " ";
    cout << endl;
    quickSort(arr, 0, arr.size() - 1);
    cout << "\nAfter Sorting -> ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;
    return 0;
}
```

Output :

```
Before Sorting -> 5 2 9 1 5 6

After Sorting -> 1 2 5 5 6 9
ap-73@AP:/mnt/A2A25781A257593D/Practical6th/DAA$
```

Program – 5

Object - Write a program to implement merge sort.

Code :

```
#include <bits/stdc++.h>
using namespace std;

// Function to merge two sorted subarrays
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1; // Size of left subarray
    int n2 = r - m;     // Size of right subarray
    // Create temporary arrays for left and right subarrays
    int L[n1], R[n2];

    // Copy data to temporary arrays
    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    // Merge the temporary arrays back into the original array
    int i = 0;  // Index of left subarray
    int j = 0;  // Index of right subarray
    int k = l;  // Index of merged array

    while (i < n1  && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    // Copy any remaining elements of the left subarray
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
```

```
    // Copy any remaining elements of the right subarray
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
// Function to perform merge sort on an array
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2; // Calculate the middle index
        // Recursively sort the left and right subarrays
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        // Merge the two sorted subarrays
        merge(arr, l, m, r);
    }
}

// Main function to test the mergeSort function
int main() {
    int n;
    cout << "Enter the size of the array: ";
    cin >> n;
    int arr[n];
    cout << "Enter the array elements: ";
    for (int i = 0; i < n; i++)
        cin >> arr[i];
    // Call mergeSort function to sort the array
    mergeSort(arr, 0, n - 1);
    // Print the sorted array
    cout << "Sorted array: ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;

    return 0;
}
```

Output :

```
Enter the size of the array: 7
Enter the array elements: 8 2 56 12 0 9 123
Sorted array: 0 2 8 9 12 56 123
ap-73@AP:/mnt/A2A25781A257593D/Practical6th/DAA$
```

Program – 6

Object - Write a program to implement Insertion Sort.

Code :

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

void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;
        /* Move elements of arr[0..i-1], that are greater than key,
           to one position ahead of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

int main() {
    int n;
    cout << "Enter the size of the array: ";
    cin >> n;
    int arr[n];
    cout << "Enter the elements of the array: ";
    for (int i = 0; i < n; i++)
        cin >> arr[i];
    cout << "Given array is \n";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    insertionSort(arr, n);
    cout << "\nSorted array is \n";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
    return 0;
}
```

Output :

```
Enter the size of the array: 7
Enter the elements of the array: 5 87 2 0 12 76 89
Given array is
5 87 2 0 12 76 89
Sorted array is
0 2 5 12 76 87 89
ap-73@AP:/mnt/A2A25781A257593D/Practical6th/DAA$
```

Program – 7

Object - Write a program to implement Heap sort.

Code :

```
#include <bits/stdc++.h>
using namespace std;

// Function to heapify a subtree rooted at index i
void heapify(int arr[], int n, int i) {
    int largest = i;  // Initialize largest as root
    int left = 2 * i + 1;  // Index of left child
    int right = 2 * i + 2;  // Index of right child

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // If right child is larger than largest so far
    if (right < n && arr[right] > arr[largest])
        largest = right;

    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// Function to perform heap sort on an array
void heapSort(int arr[], int n) {
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }
    // Extract elements from heap one by one
    for (int i = n - 1; i > 0; i--) {
        // Move current root to end
        swap(arr[0], arr[i]);
        // Call heapify on the reduced heap
        heapify(arr, i, 0);
```

```
    }
}

// Main function to test the heapSort function
int main() {
    int n;
    cout << "Enter the size of the array: ";
    cin >> n;

    int arr[n];
    cout << "Enter the array elements: ";
    for (int i = 0; i < n; i++)
        cin >> arr[i];
    // Call heapSort function to sort the array
    heapSort(arr, n);
    // Print the sorted array
    cout << "Sorted array: ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
    return 0;
}
```

Output :

```
Enter the size of the array: 7
Enter the array elements: 8 7 90 54 23 0 12
Sorted array: 0 7 8 12 23 54 90
ap-73@AP:/mnt/A2A25781A257593D/Practical6th/DAA$
```

Program – 8

Object - Write a program to implement Counting Sort.

Code :

```
#include <bits/stdc++.h>
using namespace std;

void counting_sort(int arr[], int n) {
  // Find the range of the input array
  int max_val = *max_element(arr, arr + n);
  int min_val = *min_element(arr, arr + n);
  int range = max_val - min_val + 1;

  // Create a count array to store the frequency of each element
  int count[range] = {0};
  for (int i = 0; i < n; ++i) {
    ++count[arr[i] - min_val];
  }
  // Modify the count array to store the positions of each element
  for (int i = 1; i < range; ++i) {
    count[i] += count[i - 1];
  }
  // Create a new array to store the sorted output
  int sorted_arr[n];
  for (int i = n - 1; i >= 0; --i) {
    sorted_arr[count[arr[i] - min_val] - 1] = arr[i];
    --count[arr[i] - min_val];
  }

  // Copy the sorted array back to the original array
  copy(sorted_arr, sorted_arr + n, arr);
}

int main() {
  int n;
  cout << "Enter the number of elements: ";
  cin >> n;
  int arr[n];
  cout << "Enter the elements:\n";
  for (int i = 0; i < n; ++i)
    cin >> arr[i];
  counting_sort(arr, n);
```

```
  cout << "Sorted array: ";
  for (int i = 0; i < n; ++i)
    cout << arr[i] << " ";
  cout << "\n";

  return 0;
}
```

Output :

```
Enter the number of elements: 7
Enter the elements:
8 7 90 54 23 0 12
Sorted array: 0 7 8 12 23 54 90
ap-73@AP:/mnt/A2A25781A257593D/Practical6th/DAA$
```

Program – 9

Object - Write a program to implement Radix Sort.

Code :

```
#include <bits/stdc++.h>
using namespace std;

// Function to find the maximum value in an array
int findMax(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}

// Function to perform the counting sort algorithm
void countingSort(int arr[], int n, int exp) {
    int output[n];
    int count[10] = {0};
    // Counting the frequency of digits
    for (int i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    // Updating the count array to contain the actual position of digits
    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // Building the output array
    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }
    // Copying the output array to the original array
    for (int i = 0; i < n; i++)
        arr[i] = output[i];
}
// Function to perform radix sort on an array
void radixSort(int arr[], int n) {
    int max = findMax(arr, n);
```

```
    // Performing counting sort for each digit
    for (int exp = 1; max / exp > 0; exp *= 10) {
        countingSort(arr, n, exp);
    }
}
// Main function to take input and call the radix sort function
int main() {
    int n;
    cout << "Enter the size of the array: ";
    cin >> n;

    int arr[n];
    cout << "Enter the elements of the array: ";
    for (int i = 0; i < n; i++)
        cin >> arr[i];

    radixSort(arr, n);
    // Printing the sorted array
    cout << "Sorted array: ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
    return 0;
}
```

Output :

```
Enter the size of the array: 7
Enter the elements of the array: 8 90 7 54 23 0 12
Sorted array: 0 7 8 12 23 54 90
ap-73@AP:/mnt/A2A25781A257593D/Practical6th/DAA$
```

Program – 10

Object - Write a program to implement Bucket Sort.

Code :

```
#include<bits/stdc++.h>
using namespace std;

void bucketSort(float arr[], int n){
    // Create n empty buckets
    vector<float> b[n];
    // Put array elements in different buckets
    for (int i = 0; i < n; i++) {
        int bi = n * arr[i];
        b[bi].push_back(arr[i]);
    }
    // Sort individual buckets
    for (int i = 0; i < n; i++)
        sort(b[i].begin(), b[i].end());
    // Concatenate all buckets into arr[]
    int index = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < b[i].size(); j++)
            arr[index++] = b[i][j];
}

int main(){
    int n;
    cout << "Enter the number of elements in the array: ";
    cin >> n;
    float arr[n];
    cout << "Enter the elements of the array:\n";
    for(int i = 0; i < n; i++)
        cin >> arr[i];
    bucketSort(arr, n);
    cout << "Array after bucket sort:\n";
    for(int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
    return 0;
}
```

Output :

```
Enter the number of elements in the array: 8
Enter the elements of the array:
0.23 0.78 0.95 0.43 0.56 0.87 0.23 0.67
Array after bucket sort:
0.23 0.23 0.43 0.56 0.67 0.78 0.87 0.95
ap-73@AP:/mnt/A2A25781A257593D/Practical6th/DAA$
```

Program – 11

Object - Write a program to implement Strassen's Matrix Multiplication.

Code :

```
#include<bits/stdc++.h>
using namespace std;
vector<vector<int>> matrix_add(vector<vector<int>> A, vector<vector<int>> B) {
   int n = A.size();
   int m = A[0].size();
   vector<vector<int>> C(n, vector<int>(m));
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
         C[i][j] = A[i][j] + B[i][j];
      }
   }
   return C;
}
vector<vector<int>> matrix_subtract(vector<vector<int>> A, vector<vector<int>> B) {
   int n = A.size();
   int m = A[0].size();
   vector<vector<int>> C(n, vector<int>(m));
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
         C[i][j] = A[i][j] - B[i][j];
      }
   }
   return C;
}
vector<vector<int>> strassen(vector<vector<int>> A, vector<vector<int>> B) {
   int n = A.size();
   if (n == 1) {
      vector<vector<int>> C(1, vector<int>(1));
      C[0][0] = A[0][0] * B[0][0];
      return C;
   }

   // split matrices into submatrices
   int mid = n / 2;
   vector<vector<int>> A11(mid, vector<int>(mid));
   vector<vector<int>> A12(mid, vector<int>(mid));
   vector<vector<int>> A21(mid, vector<int>(mid));
   vector<vector<int>> A22(mid, vector<int>(mid));
```

```
vector<vector<int>> B11(mid, vector<int>(mid));
vector<vector<int>> B12(mid, vector<int>(mid));
vector<vector<int>> B21(mid, vector<int>(mid));
vector<vector<int>> B22(mid, vector<int>(mid));
for (int i = 0; i < mid; i++) {
    for (int j = 0; j < mid; j++) {
        A11[i][j] = A[i][j];
        A12[i][j] = A[i][j + mid];
        A21[i][j] = A[i + mid][j];
        A22[i][j] = A[i + mid][j + mid];
        B11[i][j] = B[i][j];
        B12[i][j] = B[i][j + mid];
        B21[i][j] = B[i + mid][j];
        B22[i][j] = B[i + mid][j + mid];
    }
}

// calculate 7 products using recursion
vector<vector<int>> P1 = strassen(matrix_add(A11, A22), matrix_add(B11, B22));
vector<vector<int>> P2 = strassen(matrix_add(A21, A22), B11);
vector<vector<int>> P3 = strassen(A11, matrix_subtract(B12, B22));
vector<vector<int>> P4 = strassen(A22, matrix_subtract(B21, B11));
vector<vector<int>> P5 = strassen(matrix_add(A11, A12), B22);
vector<vector<int>> P6 = strassen(matrix_subtract(A21, A11), matrix_add(B11, B12));
vector<vector<int>> P7 = strassen(matrix_subtract(A12, A22), matrix_add(B21, B22));

// calculate submatrices of result matrix
vector<vector<int>> C11(mid, vector<int>(mid));
vector<vector<int>> C12(mid, vector<int>(mid));
vector<vector<int>> C21(mid, vector<int>(mid));
vector<vector<int>> C22(mid, vector<int>(mid));
C11 = matrix_add(matrix_subtract(matrix_add(P1, P4), P5), P7);
C12 = matrix_add(P3, P5);
C21 = matrix_add(P2, P4);
C22 = matrix_add(matrix_add(matrix_subtract(P1, P2), P3), P6);

// combine submatrices into result matrix
vector<vector<int>> C(n, vector<int>(n));
for (int i = 0; i < mid; i++) {
    for (int j = 0; j < mid; j++) {
        C[i][j] = C11[i][j];
        C[i][j + mid] = C12[i][j];
        C[i + mid][j] = C21[i][j];
        C[i + mid][j + mid] = C22[i][j];
```

```
        }
    }
    return C;
}

int main() {
int n;
cout << "Enter the size of the matrix: ";
cin >> n;
vector<vector<int>> A(n, vector<int>(n));
vector<vector<int>> B(n, vector<int>(n));
cout << "Enter elements of matrix A: " << endl;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        cin >> A[i][j];
    }
}
cout << "Enter elements of matrix B: " << endl;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        cin >> B[i][j];
    }
}
vector<vector<int>> C = strassen(A, B);

cout << "Result matrix C: " << endl;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        cout << C[i][j] << " ";
    }
    cout << endl;
}
return 0;
}
```

Output :

```
Enter the size of the matrix: 4
Enter elements of matrix A:
1 2 3 4 4 5 6 7 7 8 9 10 10 11 12 13
Enter elements of matrix B:
1 2 3 4 4 5 6 7 7 8 9 10 10 11 12 13
Result matrix C:
70 80 90 100
136 158 180 202
202 236 270 304
268 314 360 406
ap-73@AP:/mnt/A2A25781A257593D/Practical6th/DAA$
```

Program – 12

Object - Write a program to find Longest Common Sequence.

Code :

```
#include<bits/stdc++.h>
using namespace std;

// Function to find the length of longest common subsequence
int lcs(string X, string Y, int m, int n){

    int L[m + 1][n + 1];  // Create a 2D array to store the LCS lengths

    // Loop through the two strings
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            // If one of the strings is empty, then there can be no common subsequence
            if (i == 0 || j == 0)
                L[i][j] = 0;

            // If the last characters of the two strings match, then the length of LCS is 1 + LCS of
the remaining strings
            else if (X[i - 1] == Y[j - 1])
                L[i][j] = L[i - 1][j - 1] + 1;

            // If the last characters of the two strings do not match, then the LCS is the maximum of
LCS of the two strings with the last character removed
            else
                L[i][j] = max(L[i - 1][j], L[i][j - 1]);
        }
    }
    return L[m][n];  // Return the length of LCS
}

int main()
{
    string X, Y;  // Declare two string variables for user input

    cout << "Enter first string: ";
    getline(cin, X);  // Read the first string from user input

    cout << "Enter second string: ";
    getline(cin, Y);  // Read the second string from user input
```

```
    int m = X.length();  // Length of first string
    int n = Y.length();  // Length of second string

    cout << "Length of LCS is " << lcs(X, Y, m, n) << endl;  // Call the lcs() function and print the result

    return 0;  // Exit the program
}
```

Output :

```
Enter first string: AXGTY
Enter second string: AXZBTYR
Length of LCS is 4
ap-73@AP:/mnt/A2A25781A257593D/Practical6th/DAA$
```

Program – 13

Object - Write a program to implement Matrix Chain Multiplication.

Code :

```
#include<bits/stdc++.h>
using namespace std;

// Function to find the minimum number of scalar multiplications required to
// multiply the given sequence of matrices
void matrixChainOrder(int p[], int n) {
    int m[n][n]; // to store the minimum number of scalar multiplications
    int s[n][n]; // to store the optimal split position
    for (int i = 1; i < n; i++) {
        m[i][i] = 0; // a single matrix requires no multiplication
    }
    for (int L = 2; L < n; L++) {
        for (int i = 1; i < n - L + 1; i++) {
            int j = i + L - 1; // set the end index of current subsequence
            m[i][j] = INT_MAX; // initialize to maximum value
            for (int k = i; k <= j - 1; k++) {
                // calculate the number of scalar multiplications required
                int q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
                if (q < m[i][j]) {
                    // update the minimum number of scalar multiplications
                    m[i][j] = q;
                    // record the optimal split position
                    s[i][j] = k;
                }
            }
        }
    }
    cout << "Minimum number of scalar multiplications: " << m[1][n - 1] << endl;
}

int main() {
    int n;
    cout << "Enter the number of matrices: ";
    cin >> n;
    int p[n + 1]; // to store the dimensions of the matrices
    cout << "Enter the dimensions of the matrices: ";
    for (int i = 0; i < n + 1; i++) {
        cin >> p[i];
```

```
  }
  matrixChainOrder(p, n + 1); // find the minimum number of scalar multiplications
  return 0;
}
```

Output :

```
Enter the number of matrices: 4
Enter the dimensions of the matrices: 2 3 4 5 1
Minimum number of scalar multiplications: 38
ap-73@AP:/mnt/A2A25781A257593D/Practical6th/DAA$
```

Program – 14

Object - Write a program to implement Depth First Search.

Code :

```
#include<bits/stdc++.h>
using namespace std;

// Function to perform DFS
void DFS(vector<vector<int>> graph, int start) {
    // Create a stack for DFS
    stack<int> s;
    // Create a visited array to keep track of visited nodes
    vector<bool> visited(graph.size(), false);

    // Push the starting node onto the stack
    s.push(start);
    while (!s.empty()) {
        // Pop a node from the stack and mark it as visited
        int node = s.top();
        s.pop();
        visited[node] = true;
        // Print the node
        cout << node << " ";
        // Push all unvisited neighbors of the node onto the stack
        for (int neighbor : graph[node]) {
            if (!visited[neighbor]) {
                s.push(neighbor);
            }
        }
    }
}

// Driver code
int main() {
    int n;
    cout << "Enter the number of nodes in the graph: ";
    cin >> n;

    // Create a graph
    vector<vector<int>> graph(n);

    int m;
```

```
    cout << "Enter the number of edges in the graph: ";
    cin >> m;

    for (int i = 0; i < m; i++) {
        int u, v;
        cout << "Enter an edge (u v): ";
        cin >> u >> v;
        graph[u].push_back(v);
        graph[v].push_back(u);
    }

    int start;
    cout << "Enter the starting node for DFS: ";
    cin >> start;

    // Perform DFS starting from the given node
    DFS(graph, start);
    cout <<endl;

    return 0;
}
```

Output :

```
Enter the number of nodes in the graph: 5
Enter the number of edges in the graph: 6
Enter an edge (u v): 0 1
Enter an edge (u v): 0 2
Enter an edge (u v): 2 3
Enter an edge (u v): 3 4
Enter an edge (u v): 1 3
Enter an edge (u v): 4 2
Enter the starting node for DFS: 0
0 2 4 3 1 3 1
ap-73@AP:/mnt/A2A25781A257593D/Practical6th/DAA$
```

Program – 15

Object - Write a program to implement Breadth First Search.

Code :

```
#include <bits/stdc++.h>
using namespace std;

void bfs(vector<vector<int>>& graph, int start) {  // pass graph by reference
    // Initialize an empty queue and a vector to store visited nodes
    queue<int> q;
    vector<bool> visited(graph.size(), false);

    // Mark the starting node as visited and push it to the queue
    visited[start] = true;
    q.push(start);

    // Continue searching until the queue is empty
    while (!q.empty()) {
        // Get the first node in the queue and remove it from the queue
        int node = q.front();
        q.pop();
        cout << node << " ";  // print the node

        // Iterate through the neighbors of the current node
        for (int i = 0; i < graph[node].size(); i++) {
            int neighbor = graph[node][i];
            // If the neighbor has not been visited, mark it as visited and add it to the queue
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
}

int main() {
    // Prompt the user to input the number of nodes and edges
    cout << "Enter the number of nodes and edges: ";
    int n, m;
    cin >> n >> m;

    // Create an adjacency list to represent the graph
```

```
    vector<vector<int>> graph(n);
    // Read each edge and add it to the adjacency list
    cout << "Enter the edges (as pairs of integers representing the endpoints):" << endl;
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        if (u >= n || v >= n) {  // check if node number is valid
            cout << "Invalid node number!" << endl;
            return 1;
        }
        graph[u].push_back(v);
        graph[v].push_back(u);
    }

    // Read the starting node from the input
    cout << "Enter the starting node: ";
    int start;
    cin >> start;
    if (start >= n) {  // check if node number is valid
        cout << "Invalid starting node!" << endl;
        return 1;
    }

    // Call the BFS function with the graph and starting node
    bfs(graph, start);
    cout << endl;

    return 0;
}
```

Output :

```
Enter the edges (as pairs of integers representing the
0 1
0 2
1 2
2 3
3 4
4 5
5 3
Enter the starting node: 0
0 1 2 3 4 5
ap-73@AP:/mnt/A2A25781A257593D/Practical6th/DAA$
```

Program – 16

Object - Write a program to implement Kruskal Algorithm.

Code :

```
#include <bits/stdc++.h>
using namespace std;

// Define a structure to represent a weighted edge in the graph
struct Edge {
    int src, dest, weight;
};

// Define a structure to represent a subset for union-find algorithm
struct Subset {
    int parent, rank;
};

// Comparator function to sort edges in non-decreasing order of their weight
bool cmp(const Edge& e1, const Edge& e2) {
    return e1.weight < e2.weight;
}

// Function to find the parent of a subset using path compression
int find(Subset subsets[], int i) {
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}

// Function to perform union of two subsets using rank
void Union(Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
```

```
}

// Function to implement Kruskal's algorithm
void Kruskal(vector<Edge> edges, int V) {
    // Sort the edges in non-decreasing order of their weight
    sort(edges.begin(), edges.end(), cmp);

    // Allocate memory for subsets
    Subset *subsets = new Subset[V];

    // Initialize subsets
    for (int i = 0; i < V; i++) {
        subsets[i].parent = i;
        subsets[i].rank = 0;
    }

    // Initialize variables
    vector<Edge> MST;
    int edge_count = 0;

    // Iterate through all edges in non-decreasing order
    for (auto it = edges.begin(); it != edges.end() && edge_count < V - 1; it++) {
        // Find the parent of the source and destination vertices
        int x = find(subsets, it->src);
        int y = find(subsets, it->dest);

        // If including this edge does not form a cycle, include it in MST and increment edge_count
        if (x != y) {
            MST.push_back(*it);
            Union(subsets, x, y);
            edge_count++;
        }
    }

    // Print MST
    cout << "Edges in MST:\n";
    for (auto it = MST.begin(); it != MST.end(); it++)
        cout << it->src << " - " << it->dest << " : " << it->weight << endl;

    // Free memory
    delete[] subsets;
}

int main() {
```

```
    // Get input from user
    int V, E;
    cout << "Enter the number of vertices: ";
    cin >> V;
    cout << "Enter the number of edges: ";
    cin >> E;

    vector<Edge> edges;
    cout << "Enter the source, destination and weight of each edge:\n";
    for (int i = 0; i < E; i++) {
        Edge edge;
        cin >> edge.src >> edge.dest >> edge.weight;
        edges.push_back(edge);
    }

    // Find MST using Kruskal's algorithm
    Kruskal(edges, V);

    return 0;
}
```

Output :

```
Enter the number of edges: 9
Enter the source, destination and weight of each edge:
0 1 4
0 2 3
1 2 1
1 3 2
2 3 4
3 4 2
4 5 6
3 5 3
2 4 5
Edges in MST:
1 - 2 : 1
1 - 3 : 2
3 - 4 : 2
0 - 2 : 3
3 - 5 : 3
```

Program – 17

Object - Write a program to implement Prims Algorithm.

Code :

```
#include<bits/stdc++.h>
using namespace std;

const int MAXN = 1e5+5;

vector<pair<int,int>> adj[MAXN]; // adjacency list of the graph
bool visited[MAXN]; // array to keep track of visited vertices

// function to add a directed edge from vertex u to vertex v with weight w
void add_edge(int u, int v, int w) {
    adj[u].push_back({v, w});
}

int prim(int start) {
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> pq;
    int minCost = 0;
    pq.push({0, start}); // start with vertex 0 and cost 0

    while (!pq.empty()) {
        int u = pq.top().second;
        int w = pq.top().first;
        pq.pop();
        if (visited[u]) continue; // if vertex is already visited, skip it
        visited[u] = true;
        minCost += w;

        for (auto v : adj[u]) {
            if (!visited[v.first]) {
                pq.push({v.second, v.first});
            }
        }
    }

    return minCost;
}

int main() {
    int n, m; // number of vertices and edges
```

```
    cout << "Enter the number of vertices and edges: ";
    cin >> n >> m;

    // read in the edges of the graph
    cout << "Enter the edges of the graph in the format 'u v w':\n";
    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        add_edge(u, v, w);
    }

    int start = 0; // start with vertex 0
    int minCost = prim(start);

    cout << "Minimum cost of spanning tree: " << minCost << endl;

    return 0;
}
```

Output :

```
Enter the number of vertices and edges: 4 5
Enter the edges of the graph in the format 'u v w':
0 1 2
0 2 3
1 2 1
1 3 4
2 3 5
Minimum cost of spanning tree: 7
ap-73@AP:/mnt/A2A25781A257593D/Practical6th/DAA$
```

Program – 18

Object - Write a program to implement Bellmon Ford Algorithm.

Code :

```
#include<bits/stdc++.h>
using namespace std;

// A structure to represent a weighted edge in the graph
struct Edge {
    int src, dest, weight;
};

// A structure to represent the graph
struct Graph {
    int V, E;
    vector<Edge> edges;
};

// A utility function to print the solution
void printSolution(vector<int> dist) {
    cout << "Vertex \t Distance from Source\n";
    for (int i = 0; i < dist.size(); i++) {
        cout << i << "\t\t" << dist[i] << "\n";
    }
}

// The main function that finds the shortest distance from the source vertex to all other vertices
void BellmanFord(Graph graph, int src) {
    int V = graph.V;
    int E = graph.E;
    vector<int> dist(V, numeric_limits<int>::max()); // Initialize distance from the source vertex to all vertices as infinite
    dist[src] = 0; // Initialize distance from the source vertex to itself as 0

    // Relax all edges V - 1 times to find the shortest path from the source vertex to all other vertices
    for (int i = 0; i < V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = graph.edges[j].src;
            int v = graph.edges[j].dest;
            int weight = graph.edges[j].weight;
            if (dist[u] != numeric_limits<int>::max() && dist[u] + weight < dist[v]) {
```

```
                dist[v] = dist[u] + weight;
            }
        }
    }

    // Check for negative-weight cycles
    for (int i = 0; i < E; i++) {
        int u = graph.edges[i].src;
        int v = graph.edges[i].dest;
        int weight = graph.edges[i].weight;
        if (dist[u] != numeric_limits<int>::max() && dist[u] + weight < dist[v]) {
            cout << "Graph contains negative-weight cycle\n";
            return;
        }
    }

    // Print the solution
    printSolution(dist);
}

int main() {
    int V, E;
    cout << "Enter the number of vertices in the graph: ";
    cin >> V;
    cout << "Enter the number of edges in the graph: ";
    cin >> E;

    Graph graph = {V, E, {}};

    for (int i = 0; i < E; i++) {
        Edge edge;
        cout << "Enter the source vertex, destination vertex, and weight of edge " << i+1 << ": ";
        cin >> edge.src >> edge.dest >> edge.weight;
        graph.edges.push_back(edge);
    }

    int src;
    cout << "Enter the source vertex: ";
    cin >> src;

    BellmanFord(graph, src);

    return 0;
}
```

Output :

```
Enter the number of vertices in the graph: 5
Enter the number of edges in the graph: 8
Enter the source vertex, destination vertex, and weight of edge 1: 0 1 6
Enter the source vertex, destination vertex, and weight of edge 2: 0 3 2
Enter the source vertex, destination vertex, and weight of edge 3: 1 2 1
Enter the source vertex, destination vertex, and weight of edge 4: 2 4 4
Enter the source vertex, destination vertex, and weight of edge 5: 2 1 3
Enter the source vertex, destination vertex, and weight of edge 6: 3 1 3
Enter the source vertex, destination vertex, and weight of edge 7: 3 4 6
Enter the source vertex, destination vertex, and weight of edge 8: 4 3 3
Enter the source vertex: 0
Vertex   Distance from Source
0             0
1             5
2             6
3             2
4             8
ap-73@AP:/mnt/A2A257814257593D/Practical6th/DAA$
```

Program – 19

Object - Write a program to implement Dijkstra Algorithm.

Code :

```
#include<bits/stdc++.h>
using namespace std;

const int MAX_SIZE = 100;

void dijkstra(int graph[MAX_SIZE][MAX_SIZE], int n, int start) {
   int dist[MAX_SIZE]; // array to store shortest distances from start to all vertices
   bool visited[MAX_SIZE]; // array to track visited vertices
   for (int i = 0; i < n; i++) {
      dist[i] = INT_MAX; // initialize distances to "infinity"
      visited[i] = false; // initialize visited array to false
   }
   dist[start] = 0; // distance from start to itself is 0

   // iterate n-1 times (once for each vertex)
   for (int i = 0; i < n-1; i++) {
      // find the vertex with the minimum distance from start
      int min_dist = INT_MAX;
      int min_vertex;
      for (int j = 0; j < n; j++) {
         if (!visited[j] && dist[j] < min_dist) {
            min_dist = dist[j];
            min_vertex = j;
         }
      }
      visited[min_vertex] = true; // mark vertex as visited

      // update distances of adjacent vertices
      for (int k = 0; k < n; k++) {
         int weight = graph[min_vertex][k];
          if (weight != 0 && dist[min_vertex] != INT_MAX && dist[min_vertex] + weight <
dist[k]) {
            dist[k] = dist[min_vertex] + weight;
         }
      }
   }

   // print shortest distances
```

```
    cout << "Shortest distances from vertex " << start << ":\n";
    for (int i = 0; i < n; i++) {
        cout << i << ": " << dist[i] << "\n";
    }
}

int main() {
    int n; // number of vertices
    int graph[MAX_SIZE][MAX_SIZE]; // adjacency matrix
    int start; // starting vertex

    // get user input
    cout << "Enter the number of vertices: ";
    cin >> n;
    cout << "Enter the adjacency matrix (0 for no edge):\n";
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cin >> graph[i][j];
        }
    }
    cout << "Enter the starting vertex: ";
    cin >> start;

    // run Dijkstra's algorithm
    dijkstra(graph, n, start);

    return 0;
}
```

Output :

```
Enter the number of vertices: 5
Enter the adjacency matrix (0 for no edge):
0 1 1 1 0
1 0 0 1 0
1 0 0 0 1
1 1 0 0 0
0 0 1 0 0
Enter the starting vertex: 1
Shortest distances from vertex 1:
0: 1
1: 0
2: 2
3: 1
4: 3
```

Program – 20

Object - Write a program to implement Floyd Warshall Algorithm.

Code :

```
#include<bits/stdc++.h>
using namespace std;

const int INF = 1e9; // a large value to represent infinity

int main() {
    int n, m;
    cout << "Enter the number of vertices (n) and edges (m) in the graph: ";
    cin >> n >> m;

    // initialize adjacency matrix
    vector<vector<int>> adj(n+1, vector<int>(n+1, INF));
    for (int i = 1; i <= n; i++) {
        adj[i][i] = 0; // set diagonal elements to 0
    }

    // take input for edges and their weights
    cout << "Enter the edges and their weights in the format (u, v, w):\n";
    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        adj[u][v] = w;
    }

    // Floyd-Warshall algorithm
    for (int k = 1; k <= n; k++) {
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                adj[i][j] = min(adj[i][j], adj[i][k] + adj[k][j]);
            }
        }
    }

    // print the final shortest distances
    cout << "Shortest distances between all pairs of vertices:\n";
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            if (adj[i][j] == INF) {
```

```
                cout << "INF ";
            } else {
                cout << adj[i][j] << " ";
            }
        }
        cout << "\n";
    }

    return 0;
}
```

Output :

```
Enter the number of vertices (n) and edges (m) in the graph: 4 5
Enter the edges and their weights in the format (u, v, w):
1 2 5
1 4 9
2 3 2
3 1 7
3 4 1
Shortest distances between all pairs of vertices:
0 5 7 8
9 0 2 3
7 12 0 1
INF INF INF 0
```

Program – 21

Object - Write a program to implement Ford Fulkerson Algorithm.

Code :

```
#include <bits/stdc++.h>  // Include all standard libraries
using namespace std;
// A structure to represent a directed edge between two vertices
struct Edge {
    int u, v, w;  // 'u' is the source vertex, 'v' is the destination vertex and 'w' is the weight of the edge
};
// A function to find the maximum flow from source 's' to sink 't' in a graph represented by adjacency matrix 'graph'
int fordFulkerson(vector<vector<int>>& graph, int s, int t) {
    int V = graph.size();  // Get the number of vertices in the graph
    // Initialize the residual graph with the same capacities as the original graph
    vector<vector<int>> residualGraph(V, vector<int>(V));
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            residualGraph[i][j] = graph[i][j];

    // Initialize the parent array for BFS
    vector<int> parent(V);
    int maxFlow = 0;  // Initialize the maximum flow to 0
    // Augment the flow while there is a path from source to sink in the residual graph
    while (true) {
        // Use BFS to find a path from source to sink in the residual graph
        fill(parent.begin(), parent.end(), -1);
        queue<int> q;
        q.push(s);
        parent[s] = -2;
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int v = 0; v < V; v++) {
                if (parent[v] == -1 && residualGraph[u][v] > 0) {
                    parent[v] = u;
                    q.push(v);
                }
            }
        }
```

```
        // If there is no path from source to sink in the residual graph, we have reached the maximum flow
        if (parent[t] == -1) {
            break;
        }
        // Find the bottleneck capacity of the augmenting path
        int bottleneckCapacity = INT_MAX;
        for (int v = t; v != s; v = parent[v]) {
            int u = parent[v];
            bottleneckCapacity = min(bottleneckCapacity, residualGraph[u][v]);
        }
        // Update the residual capacities of the edges and reverse edges along the augmenting path
        for (int v = t; v != s; v = parent[v]) {
            int u = parent[v];
            residualGraph[u][v] -= bottleneckCapacity;
            residualGraph[v][u] += bottleneckCapacity;
        }
        // Add the bottleneck capacity to the maximum flow
        maxFlow += bottleneckCapacity;
    }
    // Return the maximum flow
    return maxFlow;
}

int main() {
    int V, E, s, t;
    cout << "Enter the number of vertices: ";
    cin >> V;
    cout << "Enter the number of edges: ";
    cin >> E;
    // Initialize the graph as an adjacency matrix
    vector<vector<int>> graph(V, vector<int>(V));
    // Take input for all the edges
    for (int i = 0; i < E; i++) {
        int u, v, w;
        cout << "Enter the source, destination and weight of edge " << i+1 << ": ";
        cin >> u >> v >> w;
        graph[u][v] = w;
    }
    // Take input for source and sink
    cout << "Enter the source vertex: ";
    cin >> s;
    cout << "Enter the sink vertex: ";
    cin >> t;
```

```
    int maxFlow = fordFulkerson(graph, s, t);  // Calculate the maximum flow using the Ford-Fulkerson Algorithm
    cout << "The maximum flow from vertex " << s << " to vertex " << t << " is: " << maxFlow << endl;  // Output the maximum flow

    return 0;  // Return 0 to indicate successful completion of the program
}
```

Output :

```
Enter the number of vertices: 6
Enter the number of edges: 10
Enter the source, destination and weight of edge 1: 0 1 16
Enter the source, destination and weight of edge 2: 0 2 13
Enter the source, destination and weight of edge 3: 1 3 12
Enter the source, destination and weight of edge 4: 2 1 4
Enter the source, destination and weight of edge 5: 2 4 14
Enter the source, destination and weight of edge 6: 3 2 9
Enter the source, destination and weight of edge 7: 3 5 20
Enter the source, destination and weight of edge 8: 4 3 7
Enter the source, destination and weight of edge 9: 4 5 4
Enter the source, destination and weight of edge 10: 5 1 8
Enter the source vertex: 0
Enter the sink vertex: 5
The maximum flow from vertex 0 to vertex 5 is: 23
```