1.

(i) Implement Insertion Sort (The program should report the number of comparisons)

Code :-

```
 #include <bits/stdc++.h>
using namespace std;
void insertionSort(int A[], int n)
{
   int i, key, j;
   for (j = 1; j < n; j++)
   {
      key = A[j];
      i = j - 1;
      while (i >= 0 && A[i] > key)
      {
         A[i + 1] = A[i];
         i = i - 1;
      }
      A[i + 1] = key;
   }
}

void printArray(int A[], int n)
{
   int i;
   for (i = 0; i < n; i++)
```



```
      cout << A[i] << " ";
   cout << endl;
}


int main()
{
   cout<<"---------INSERTION SORT---------\n";
   int A[100],i,j,n;
   cout<<"\nPlease enter no. of elements:- ";
   cin>>n;
   cout<<"\nEnter element:\n";
   for (i=0;i<n;i++){
      cin>>A[i];
   }
   insertionSort(A, n);
   cout<<"Array after sorted:-\n";
   printArray(A, n);
   return 0;
}
```


(ii) Implement Merge Sort (The program should report the number of comparisons)

Code :-

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

void swapping(int &a, int &b) {     //swap the content of a and b
   int temp;
   temp = a;
   a = b;
   b = temp;
}

void merge(int *array, int l, int m, int r) {
   int i, j, k, nl, nr;
   //size of left and right sub-arrays
   nl = m-l+1; nr = r-m;
   int larr[nl], rarr[nr];
   //fill left and right sub-arrays
   for(i = 0; i<nl; i++)
      larr[i] = array[l+i];
   for(j = 0; j<nr; j++)
      rarr[j] = array[m+1+j];
   i = 0; j = 0; k = l;
   //merge temp arrays to real array
   while(i < nl && j<nr) {
```



```
      if(larr[i] <= rarr[j]) {
         array[k] = larr[i];
         i++;
      }else{
         array[k] = rarr[j];
         j++;
      }
      k++;
   }
   while(i<nl) {       //extra element in left array
      array[k] = larr[i];
      i++; k++;}
   while(j<nr) {     //extra element in right array
      array[k] = rarr[j];
      j++; k++;
}
}
void mergeSort(int *array, int l, int r) {
   int m;
   if(l < r) {
      int m = l+(r-l)/2;
      // Sort first and second arrays
      mergeSort(array, l, m);
      mergeSort(array, m+1, r);
      merge(array, l, m, r); }}
```



```
void display(int *array, int size) {
   for(int i = 0; i<size; i++)
      cout << array[i] << " ";
   cout << endl;
}

int main() {
   int n;
   cout << "--------MERGE SORT---------\n";
   cout << "\nEnter the number of elements: ";
   cin >> n;
   int arr[n];     //create an array with given number of elements
   cout << "Enter elements:" << endl;
   for(int i = 0; i<n; i++) {
      cin >> arr[i];
   }
   cout << "Array before Sorting: ";
   display(arr, n);
   mergeSort(arr, 0, n-1);     //(n-1) for last index
   cout << "Array after Sorting: ";
   display(arr, n);
}
```


2. Implement Heap Sort(The program should report the number of comparisons)

Code :-

```
#include <iostream>
#include<iostream>

using namespace std;

// To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
void heapify(int arr[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int l = 2 * i + 1; // left = 2*i + 1
    int r = 2 * i + 2; // right = 2*i + 2

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

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

    // 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 do heap sort
void heapSort(int arr[], int n)
{
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i = n - 1; i > 0; i--) {
        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```



```
void display(int *array, int size) {
   for(int i = 0; i<size; i++)
      cout << array[i] << " ";
   cout << endl;
}


int main() {
   cout << "--------HEAP SORT---------\n";
   int n;
   cout << "\nEnter the number of elements: ";
   cin >> n;
   int arr[n];
   cout << "Enter elements:" << endl;
   for(int i = 0; i<n; i++) {
      cin >> arr[i];
   }
   cout << "Array before Sorting: ";
   display(arr, n);
   heapSort(arr, n);
   cout << "Array after Sorting: ";
   display(arr, n);
}
```


3. Implement Randomized Quick sort (The program should report the number of comparisons)

Code :-

```
#include <cstdlib>
#include <time.h>
#include <iostream>
using namespace std;

/* This function takes last element as pivot, places the pivot element at its correct position in sorted array, and
 places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot */
int partition(int arr[], int low, int high)
{
        // pivot
        int pivot = arr[high];
        // Index of smaller element
        int i = (low - 1);
        for (int j = low; j <= high - 1; j++)
        {
                // If current element is smaller than or equal to pivot
                if (arr[j] <= pivot) {
                        // increment index of smaller element
                        i++;
                        swap(arr[i], arr[j]);
```



```
                }
        }
        swap(arr[i + 1], arr[high]);
        return (i + 1);
}

// Generates Random Pivot, swaps pivot with end element and calls the partition function
int partition_r(int arr[], int low, int high)
{
        // Generate a random number in between low to high
        srand(time(NULL));
        int random = low + rand() % (high - low);
        // Swap A[random] with A[high]
        swap(arr[random], arr[high]);
        return partition(arr, low, high);
}

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
{
        if (low < high) {
```



```
                /* pi is partitioning index,
                arr[p] is now
                at right place */
                int pi = partition_r(arr, low, high);
                // Separately sort elements before partition and after partition
                quickSort(arr, low, pi - 1);
                quickSort(arr, pi + 1, high);
        }
}

/* Function to print an array */
void display(int *array, int size) {
   for(int i = 0; i<size; i++)
      cout << array[i] << " ";
   cout << endl;
}


int main()
{
   int n;
   cout << "--------RANDOMIZED QUICK SORT---------\n";
   cout << "\nEnter the number of elements: ";
   cin >> n;
   int arr[n];
   //create an array with given number of elements
```



```
   cout << "Enter elements:" << endl;
   for(int i = 0; i<n; i++) {
      cin >> arr[i];
   }
   cout << "Array before Sorting: ";
   display(arr, n);
   quickSort(arr, 0, n - 1);
   cout << "Array after Sorting: ";
   display(arr, n);
   return 0;
}
```

4. Implement Radix Sort .

Code :-

```
#include <iostream>
using namespace std;
// A utility function to get maximum value in arr[]
int getMax(int arr[], int n)
{
        int mx = arr[0];
        for (int i = 1; i < n; i++)
                if (arr[i] > mx)
                        mx = arr[i];
        return mx;
}
// A function to do counting sort of arr[] according to the digit represented by exp.
void countSort(int arr[], int n, int exp)
{
        int output[n]; // output array
        int i, count[10] = { 0 };
        // Store count of occurrences in count[]
        for (i = 0; i < n; i++)
                count[(arr[i] / exp) % 10]++;
        // Change count[i] so that count[i] now contains actual position of this digit in output[]
        for (i = 1; i < 10; i++)
                count[i] += count[i - 1];
```



```
        // Build the output array
        for (i = n - 1; i >= 0; i--) {
                output[count[(arr[i] / exp) % 10] - 1] = arr[i];
                count[(arr[i] / exp) % 10]--;
        }
// Copy the output array to arr[], so that arr[] now contains sorted numbers according to current digit
        for (i = 0; i < n; i++)
                arr[i] = output[i];
    }
// Radix Sort
void radixsort(int arr[], int n)
{
        // Find the maximum number to know number of digits
        int m = getMax(arr, n);

        /* Do counting sort for every digit. Note that instead of passing digit number, exp is passed. exp is 10^i
        where i is current digit number */
        for (int exp = 1; m / exp > 0; exp *= 10)
                countSort(arr, n, exp);
}
/* Function to print an array */
void display(int *array, int size) {
  for(int i = 0; i<size; i++)
    cout << array[i] << " ";
```



```
  cout << endl;
}

int main()
{
  int n;
  cout << "--------RADIX SORT---------\n";
  cout << "\nEnter the number of elements: ";
  cin >> n;
  int arr[n];
  //create an array with given number of elements
  cout << "Enter elements:" << endl;
  for(int i = 0; i<n; i++) {
    cin >> arr[i];
  }
  cout << "Array before Sorting: ";
  display(arr, n);
  radixsort(arr, n);
  cout << "\nArray after Sorting: ";
  display(arr, n);
  return 0;
}
```


5. Implement Bucket Sort

Code :-

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

int getMax(int a[], int n) // function to get maximum element from the given array
{
  int max = a[0];
  for (int i = 1; i < n; i++)
    if (a[i] > max)
      max = a[i];
  return max;
}
void bucket(int a[], int n) // function to implement bucket sort
{
  int max = getMax(a, n); //max is the maximum element of array
  int bucket[max], i;
  for (int i = 0; i <= max; i++)
  {
    bucket[i] = 0;
  }
  for (int i = 0; i < n; i++)
  {
```



```
    bucket[a[i]]++;
  }
  for (int i = 0, j = 0; i <= max; i++)
  {
    while (bucket[i] > 0)
    {
      a[j++] = i;
      bucket[i]--;
    }
  }
}

/* Function to print an array */
void display(int *array, int size) {
  for(int i = 0; i<size; i++)
    cout << array[i] << " ";
  cout << endl;
}
/* Driver program to test above function */
int main()
{
  int n;
  cout << "--------BUCKET SORT---------\n";
  cout << "\nEnter the number of elements: ";
  cin >> n;
  int arr[n];
```



```
  //create an array with given number of elements
  cout << "Enter elements:" << endl;
  for(int i = 0; i<n; i++) {
    cin >> arr[i];
  }
  cout << "Array before Sorting: ";
  display(arr, n);
  bucket(arr, n);
  cout << "Array after Sorting: ";
  display(arr, n);
  return 0;
}
```

6. Implement Randomized Select.

Code :-

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

void swap(int *xp, int *yp)
{
        int temp = *xp;
        *xp = *yp;
        *yp = temp;
}

void selectionSort(int arr[], int n)
{
        int i, j, min_idx;
        // One by one move boundary of unsorted sub array
        for (i = 0; i < n-1; i++)
        {       // Find the minimum element in unsorted array
                min_idx = i;
                for (j = i+1; j < n; j++)
                if (arr[j] < arr[min_idx])
                        min_idx = j;
                // Swap the found minimum element with the first element
                swap(&arr[min_idx], &arr[i]);
        }
```



```
}
/* Function to print an array */
void display(int *array, int size) {
  for(int i = 0; i<size; i++)
    cout << array[i] << " ";
  cout << endl;

int main()
{
  int n;
  cout << "--------RANDOMIZED SELECTION---------\n";
  cout << "\nEnter the number of elements: ";
  cin >> n;
  int arr[n];
  //create an array with given number of elements
  cout << "Enter elements:" << endl;
  for(int i = 0; i<n; i++) {
    cin >> arr[i];  }
  cout << "Array before Sorting: ";
  display(arr, n);
  selectionSort(arr, n);
  cout << "Array after Sorting: ";
  display(arr, n);
  return 0;}
```


7. Implement Breadth-First Search in a graph

Code :-

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

// This class represents a directed graph using adjacency list representation
class Graph
{
        int V;
        // No. of vertices Pointer to an array containing adjacency lists
        list<int> *adj;
public:
        Graph(int V); // Constructor
        // function to add an edge to graph
        void addEdge(int v, int w);
        // prints BFS traversal from a given source s
        void BFS(int s);
};

Graph::Graph(int V)
{
        this->V = V;
        adj = new list<int>[V];
}
```



```
void Graph::addEdge(int v, int w)
{
        adj[v].push_back(w); // Add w to v's list.
}

void Graph::BFS(int s)
{
        // Mark all the vertices as not visited
        bool *visited = new bool[V];
        for(int i = 0; i < V; i++)
                visited[i] = false;
        // Create a queue for BFS
        list<int> queue;
        // Mark the current node as visited and enqueue it
        visited[s] = true;
        queue.push_back(s);
        // 'i' will be used to get all adjacent
        // vertices of a vertex
        list<int>::iterator i;
        while(!queue.empty())
        {
                // Dequeue a vertex from queue and print it
                s = queue.front();
                cout << s << " ";
                queue.pop_front();
```



```
                // Get all adjacent vertices of the dequeued vertex s. If a adjacent has not
been visited,
                // then mark it visited and enqueue it
                for (i = adj[s].begin(); i != adj[s].end(); ++i)
                {
                        if (!visited[*i])
                        {
                                visited[*i] = true;
                                queue.push_back(*i);
                        }
                }
        }
}

// Driver program to test methods of graph class
int main()
{
        // Create a graph given in the above diagram
  cout << "--------Breadth First Search---------\n";
        Graph g(13);
        g.addEdge(0, 1);
  g.addEdge(0, 2);
  g.addEdge(0, 3);
  g.addEdge(1, 3);
  g.addEdge(2, 4);
```



```
  g.addEdge(3, 5);
  g.addEdge(3, 6);
  g.addEdge(4, 7);
  g.addEdge(4, 5);
  g.addEdge(5, 2);
  g.addEdge(6, 5);
  g.addEdge(7, 5);
  g.addEdge(7, 8);
        cout << "\nFollowing is Breadth First Traversal "<< "(starting from vertex 2) \n";
        g.BFS(2);
        return 0;
}
```

8. Implement Depth-First Search in a graph.

Code :-

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

// Graph class represents a directed graphusing adjacency list representation
class Graph {
public:
	map<int, bool> visited;
	map<int, list<int> > adj;

	// function to add an edge to graph
	void addEdge(int v, int w);

	// DFS traversal of the vertices reachable from v
	void DFS(int v);
};

void Graph::addEdge(int v, int w)
{
	adj[v].push_back(w); // Add w to v's list.
}

void Graph::DFS(int v)
{
```



```
	// Mark the current node as visited and print it
	visited[v] = true;
	cout << v << " ";

	// Recur for all the vertices adjacent to this vertex
	list<int>::iterator i;
	for (i = adj[v].begin(); i != adj[v].end(); ++i)
		if (!visited[*i])
			DFS(*i);}

int main()
{
   cout << "--------Depth First Search---------\n";
	Graph g;
	g.addEdge(0, 1);
	g.addEdge(0, 2);
	g.addEdge(1, 2);
	g.addEdge(2, 0);
	g.addEdge(2, 3);
	g.addEdge(3, 3);
	cout << "\nFollowing is Depth First Traversal (starting from vertex 2) \n";
	g.DFS(2);
	return 0;
}
```


9. Write a program to determine the minimum spanning tree of a graph using both algorithm

(i) Prims (ii) Kruskals

Code :-

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

// Number of vertices in the graph
#define V 5
// A utility function to find the vertex with minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
{
	// Initialize min value
	int min = INT_MAX, min_index;
	for (int v = 0; v < V; v++)
		if (mstSet[v] == false && key[v] < min)
			min = key[v], min_index = v;
	return min_index;
}

// A utility function to print the constructed MST stored in parent[]
void printMST(int parent[], int graph[V][V])
{
	cout<<"Edge \tWeight\n";
	for (int i = 1; i < V; i++)
```



```
		cout<<parent[i]<<" - "<<i<<" \t"<<graph[i][parent[i]]<<" \n";
}

// Function to construct and print MST for a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
	// Array to store constructed MST
	int parent[V];
	// Key values used to pick minimum weight edge in cut
	int key[V];
	// To represent set of vertices included in MST
	bool mstSet[V];
	// Initialize all keys as INFINITE
	for (int i = 0; i < V; i++)
		key[i] = INT_MAX, mstSet[i] = false;

	// Always include first 1st vertex in MST.
	// Make key 0 so that this vertex is picked as first vertex.

	key[0] = 0;
	parent[0] = -1; // First node is always root of MST
	// The MST will have V vertices
	for (int count = 0; count < V - 1; count++)
	{
```



```
		// Pick the minimum key vertex from the set of vertices not yet included in MST
		int u = minKey(key, mstSet);

		// Add the picked vertex to the MST Set
		mstSet[u] = true;

		// Update key value and parent index of the adjacent vertices of the picked vertex.
		// Consider only those vertices which are not yet included in MST
		for (int v = 0; v < V; v++)

			// graph[u][v] is non zero only for adjacent vertices of m
			// mstSet[v] is false for vertices not yet included in MST
			// Update the key only if graph[u][v] is smaller than key[v]

			if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
				parent[v] = u, key[v] = graph[u][v];
	}
	// print the constructed MST
	printMST(parent, graph);
}

int main()
{
	cout << "--------Prim's Minimum Spanning Tree (MST)---------\n";
```



```
	/* Let us create the following graph
		2 3
	(0)--(1)--(2)
	| / \ |
	6| 8/ \5 |7
	| / \ |
	(3)-------(4)
			9 	*/
	int graph[V][V] = { { 0, 2, 0, 6, 0 },
						{ 2, 0, 3, 8, 5 },
						{ 0, 3, 0, 0, 7 },
						{ 6, 8, 0, 0, 9 },
						{ 0, 5, 7, 9, 0 } };

	primMST(graph);
	return 0;
}
```

(ii) Kruskal's

Code :-

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

// DSU data structure path compression + rank by union
class DSU {
    int* parent;
    int* rank;
public:
    DSU(int n)
    {
        parent = new int[n];
        rank = new int[n];

        for (int i = 0; i < n; i++) {
            parent[i] = -1;
            rank[i] = 1;
        }
    }

    // Find function
    int find(int i)
    {
        if (parent[i] == -1)
            return i;
```



```
            return parent[i] = find(parent[i]);
    }
    // union function
    void unite(int x, int y)
    {
        int s1 = find(x);
        int s2 = find(y);

        if (s1 != s2) {
            if (rank[s1] < rank[s2]) {
                parent[s1] = s2;
                rank[s2] += rank[s1];
            }
            else {
                parent[s2] = s1;
                rank[s1] += rank[s2];
            } } }
};

class Graph {
    vector<vector<int> > edgelist;
    int V;
public:
    Graph(int V) { this->V = V; }
```



```
    void addEdge(int x, int y, int w)
    {
        edgelist.push_back({ w, x, y });}

    void kruskals_mst()
    {
        // 1. Sort all edges
        sort(edgelist.begin(), edgelist.end());
        // Initialize the DSU
        DSU s(V);
        int ans = 0;
        cout << "Following are the edges in the "
                "constructed MST"
            << endl;
        for (auto edge : edgelist) {
            int w = edge[0];
            int x = edge[1];
            int y = edge[2];
            // take that edge in MST if it does form a cycle
            if (s.find(x) != s.find(y)) {
                s.unite(x, y);
                ans += w;
                cout << x << " -- " << y << " == " << w
                    << endl;
            }
        }
```



```
        cout << "Minimum Cost Spanning Tree: " << ans;
    }};
int main()
{
    cout << "-------- Kruskals's Minimum Spanning Tree (MST)---------\n";
    cout << "\n";
    /* Let us create following weighted graph
                10
            0--------1
            | \      |
            6| 5\ |15
            |       \ |
            2--------3
                4       */
    Graph g(4);
    g.addEdge(0, 1, 10);
    g.addEdge(1, 3, 15);
    g.addEdge(2, 3, 4);
    g.addEdge(2, 0, 6);
    g.addEdge(0, 3, 5);

    g.kruskals_mst();
    cout << "\n";
    return 0;
}
```


10. Write a program to solve the weighted interval scheduling problem.

Code :-

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

struct Job
{
    int start, finish, profit;
};
// A utility function that is used for sorting events according to finish time
bool myfunction(Job s1, Job s2)
{
    return (s1.finish < s2.finish);
}
int binarySearch(Job jobs[], int index)
{
    int lo = 0, hi = index - 1;
    while (lo <= hi)
    {
        int mid = (lo + hi) / 2;
        if (jobs[mid].finish <= jobs[index].start)
        {
            if (jobs[mid + 1].finish <= jobs[index].start)
                lo = mid + 1;
```



```
            else
                return mid;}
        else
            hi = mid - 1;}
    return -1;
}

// The main function that returns the maximum possible profit from given array of jobs
int findMaxProfit(Job arr[], int n)
{
    // Sort jobs according to finish time
    sort(arr, arr+n, myfunction);

    // Create an array to store solutions of subproblems. table[i]
    // stores the profit for jobs till arr[i] (including arr[i])
    int *table = new int[n];
    table[0] = arr[0].profit;

    // Fill entries in table[] using recursive property
    for (int i=1; i<n; i++)
    {
        // Find profit including the current job
        int inclProf = arr[i].profit;
        int l = binarySearch(arr, i);
        if (l != -1)
            inclProf += table[l];
```

```
            // Store maximum of including and excluding
            table[i] = max(inclProf, table[i-1]);
        }

        // Store result and free dynamic memory allocated for table[]
        int result = table[n-1];
        delete[] table;
        return result;
}

int main()
{
   cout << "--------Weighted Interval Scheduling---------\n";
   cout << "\n";
   cout << "Job arr[] = {{3, 10, 20}, {1, 2, 50}, {6, 19, 100}, {2, 100, 200}}";
        Job arr[] = {{3, 10, 20}, {1, 2, 50}, {6, 19, 100}, {2, 100, 200}};
        int n = sizeof(arr)/sizeof(arr[0]);
        cout << "\n";
        cout << "\n Optimal profit is = " << findMaxProfit(arr, n);
        cout << "\n";
        return 0;
}
```


11. Write a program to solve the 0-1 knapsack problem.

Code :-

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

//Function that returns maximum of two integers
int max(int a, int b)
{
   return (a > b) ? a : b;
   }

// Returns the maximum value that can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{
        // Base Case
        if (n == 0 || W == 0)
                return 0;

        // If weight of the nth item is more than Knapsack capacity W, then this item
cannot be included
        // in the optimal solution
        if (wt[n - 1] > W)
                return knapSack(W, wt, val, n - 1);

        // Return the maximum of two cases:
        // (1) nth item included
```



```
        // (2) not included
        else
                return max(
                        val[n - 1]
                                + knapSack(W - wt[n - 1],
                                                wt, val, n - 1),
                        knapSack(W, wt, val, n - 1));
}

int main()
{
   cout << "[--------0-1 Knapsack---------]\n";
        int val[] = { 60, 100, 120,114 };
        int wt[] = { 10, 20, 30, 40, };;
        int W = 40;
        int n = sizeof(val) / sizeof(val[0]);
   cout << "\nKnapsack value is =" <<knapSack(W, wt, val, n);
   cout << "\n";
        return 0;
}
```