1.//BubbleSort

// Optimized implementation of Bubble sort

#include<iostream>

#include <stdio.h>

using namespace std;

void swap(int \*xp, int \*yp)

{

int temp = \*xp;

\*xp = \*yp;

\*yp = temp;

}

// An optimized version of Bubble Sort

void bubbleSort(int arr[], int n)

{

int i, j;

bool swapped;

for (i = 0; i < n-1; i++)

{

swapped = false;

for (j = 0; j < n-i-1; j++)

{

if (arr[j] > arr[j+1])

{

swap(&arr[j], &arr[j+1]);

swapped = true;

}

}

// IF no two elements were swapped by inner loop, then break

if (swapped == false)

break;

}

}

/\*Counting no of comparisons

void bubbleSort(int arr[], int n)

{

bool sorted = false;

for(int pass = 1; pass < n.size() && !sorted; pass++)

{

sorted = true;

int i;

for(i = 0; i < n.size() - pass; i++)

{

if(\*n[i + 1] < \*n[i])

{

swap(n, i, i + 1);

sorted = false;

}

}

CountbubbleSort += i;

}

cout<<"bubbleSort comparison is "<<CountbubbleSort<<endl;

}\*/

/\* Function to print an array \*/

void printArray(int arr[], int size)

{

int i;

cout<<"[";

for (i=0; i < size; i++)

cout<<arr[i]<<"][";

}

// Driver program to test above functions

int main()

{

int arr[100];

int n;

cout<<"Enter the number of elements in array: ";

cin>>n;

for(int i = 0; i < n; i++){

cout<<"Enter the number-"<<i+1<<": ";

cin>>arr[i];

}

bubbleSort(arr, n);

cout<<"\n\nSorted array: \n";

printArray(arr, n);

return 0;

}

2.//Insertion Sort

// C++ program for insertion sort

#include<iostream>

#include <math.h>

#include <stdio.h>

using namespace std;

/\* Function to sort an array using insertion sort\*/

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 = j - 1;

}

arr[j + 1] = key;

}

}

// A utility function to print an array of size n

void printArray(int arr[], int n)

{

int i;

cout<<"\n\nSorted array is:\n";

for (i = 0; i < n; i++)

cout<<"["<<arr[i]<<"]";

cout<<"\n";

}

/\* Driver program to test insertion sort \*/

int main()

{

int arr[100];

int n;

cout<<"Enter numbers in array: ";

cin>>n;

for(int i = 0; i < n; i++){

cout<<"Enter element-"<<i+1<<": ";

cin>>arr[i];

}

insertionSort(arr, n);

printArray(arr, n);

return 0;

}

3.//Selection Sort

#include<iostream>

using namespace std;

//swap the content of a and b

void swapping(int &a, int &b){

int temp;

temp = a;

a = b;

b = temp;

}

void display(int \*array, int size){

for(int i = 0; i<size; i++)

cout << array[i] << " ";

cout << endl;

}

void selectionSort(int \*array, int size){

int i, j, imin;

for(i = 0; i<size-1; i++) {

imin = i; //get index of minimum data

for(j = i+1; j<size; j++)

if(array[j] < array[imin])

imin = j;

//placing in correct position

swap(array[i], array[imin]);

}

}

int main(){

int n;

cout << "Enter 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);

}

4.//Merge Sort

/\* C program for Merge Sort \*/

#include<iostream>

#include<stdlib.h>

#include<stdio.h>

using namespace std;

// Merges two subarrays of arr[].

// First subarray is arr[l..m]

// Second subarray is arr[m+1..r]

void merge(int arr[], int l, int m, int r)

{

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

/\* create temp arrays \*/

int L[n1], R[n2];

/\* Copy data to temp arrays L[] and R[] \*/

for (i = 0; i < n1; i++)

L[i] = arr[l + i];

for (j = 0; j < n2; j++)

R[j] = arr[m + 1+ j];

/\* Merge the temp arrays back into arr[l..r]\*/

i = 0; // Initial index of first subarray

j = 0; // Initial index of second subarray

k = l; // Initial index of merged subarray

while (i < n1 && j < n2)

{

if (L[i] <= R[j])

{

arr[k] = L[i];

i++;

}

else

{

arr[k] = R[j];

j++;

}

k++;

}

/\* Copy the remaining elements of L[], if there

are any \*/

while (i < n1)

{

arr[k] = L[i];

i++;

k++;

}

/\* Copy the remaining elements of R[], if there

are any \*/

while (j < n2)

{

arr[k] = R[j];

j++;

k++;

}

}

/\* l is for left index and r is right index of the

sub-array of arr to be sorted \*/

void mergeSort(int arr[], int l, int r)

{

if (l < r)

{

// Same as (l+r)/2, but avoids overflow for

// large l and h

int m = l+(r-l)/2;

// Sort first and second halves

mergeSort(arr, l, m);

mergeSort(arr, m+1, r);

merge(arr, l, m, r);

}

}

/\* UTILITY FUNCTIONS \*/

/\* Function to print an array \*/

void printArray(int arr[], int n)

{

for (int i=0; i < n; i++)

cout<<"["<<arr[i]<<"]";

cout<<"\n";

}

/\* Driver program to test above functions \*/

int main()

{

int arr[100];

int n;

cout<<"Given array should have elements: ";

cin>>n;

for(int i = 0; i < n; i++){

cout<<"Provide element-"<<i+1<<": ";

cin>>arr[i];

}

mergeSort(arr, 0, n - 1);

printf("\nSorted array is \n");

printArray(arr, n);

return 0;

}

5.//Quick Sort

/\* C implementation QuickSort \*/

#include<iostream>

#include<stdio.h>

using namespace std;

// A utility function to swap two elements

void swap(int\* a, int\* b)

{

int t = \*a;

\*a = \*b;

\*b = t;

}

/\* 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)

{

int pivot = arr[high]; // pivot

int i = (low - 1); // Index of smaller element

for (int j = low; j <= high- 1; j++)

{

// If current element is smaller than or

// equal to pivot

if (arr[j] <= pivot)

{

i++; // increment index of smaller element

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}

/\* 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(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 printArray(int arr[], int size)

{

int i;

for (i=0; i < size; i++)

cout<<"["<<arr[i]<<"]";

}

// Driver program to test above functions

int main()

{

int arr[100];

int n;

cout<<"Numbers in array should be: ";

cin>>n;

for(int i = 0; i < n; i++){

cout<<"Enter element-"<<i+1<<": ";

cin>>arr[i];

}

quickSort(arr, 0, n-1);

cout<<"\n\nSorted array:\n";

printArray(arr, n);

return 0;

}

6.//Heap Sort

// C++ program for implementation of Heap Sort

#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);

}

}

// main 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);

}

}

/\* A utility function to print array of size n \*/

void printArray(int arr[], int n)

{

for (int i = 0; i < n; ++i)

cout<<"["<<arr[i] <<"]";

cout << "\n";

}

// Driver program

int main()

{

int arr[100];

int n;

cout<<"Array should have the number in: ";

cin>>n;

for(int i = 0; i < n; i++){

cout<<"Provide element-"<<i+1<<": ";

cin>>arr[i];

}

heapSort(arr, n);

cout << "\nSorted array is \n";

printArray(arr, n);

}

7.//Radix Sort

// C++ implementation of Radix Sort

#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];

}

// The main function to that sorts arr[] of size n using

// 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);

}

// A utility function to print an array

void print(int arr[], int n)

{

for (int i = 0; i < n; i++)

cout<<"["<<arr[i]<<"]";

cout<<"\n";

}

// Driver program to test above functions

int main()

{

int arr[100];

int n;

cout<<"Array should encounter the numbers: ";

cin>>n;

for(int i = 0; i < n; i++){

cout<<"Counter element-"<<i+1<<": ";

cin>>arr[i];

}

radixsort(arr, n);

cout<<"\nSorted array is:\n";

print(arr, n);

return 0;

}

8.//DFT

// C++ program to print DFS traversal from

// a given vertex in a given graph

#include<iostream>

#include<list>

using namespace std;

// Graph 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;

// A recursive function used by DFS

void DFSUtil(int v, bool visited[]);

public:

Graph(int V); // Constructor

// 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);

};

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::DFSUtil(int v, bool visited[])

{

// 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])

DFSUtil(\*i, visited);

}

// DFS traversal of the vertices reachable from v.

// It uses recursive DFSUtil()

void Graph::DFS(int v)

{

// Mark all the vertices as not visited

bool \*visited = new bool[V];

for (int i = 0; i < V; i++)

visited[i] = false;

// Call the recursive helper function

// to print DFS traversal

DFSUtil(v, visited);

}

int main()

{

// Create a graph given in the above diagram

Graph g(4);

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 << "Following is Depth First Traversal\n"

" (starting from vertex 2) \n";

g.DFS(2);

return 0;

}

9.//BFT

// Program to print BFS traversal from a given

// source vertex. BFS(int s) traverses vertices

// reachable from s.

#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

Graph g(4);

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 << "Following is Breadth First Traversal\n"

<< "(starting from vertex 2) \n";

g.BFS(2);

return 0;

}