**Explanation of the code:**

The code is fully commented on and explained.

**Complexity analysis of the algorithm:**

To get the time complexity of this program isn’t straightforward. Since we compare multiple sorting algorithms we have to first calculate all of the worst-case scenarios for all of them individually. And then we can pick out the more dominant result out of all of them.

Heap Sort: O(nlogn)

Selection Sort: O(n^2)

Bubble Sort: O(n^2)

Insertion Sort: O(n^2)

So based of the results we can say that the entire program has a time complexity of O(n^2) since selection, bubble, and insertion all dominate the O(nlogn) from the heap sort algorithm.

**Code**:

#include <iostream>

#include <time.h>

#include <cstdlib>

#include <chrono>

#include <iomanip>

#include <fstream>

using namespace std;

using namespace std::chrono;

// Global counter for comparisons made

long long comparisons;

// I won't comment all of the sorting functions since I have already explained most of them in previous tasks

// Function to swap two elements in the array

void swap(int arr[], int i, int j) {

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

// Function to print the array

void printArray(int arr[], int n) {

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

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

}

cout << "\n";

}

// Function to heapify a subtree rooted with node `i`

void heapify(int arr[], int n, int i) {

int smallest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

if (left < n) {

comparisons++;

if (arr[left] < arr[smallest]) {

smallest = left;

}

}

if (right < n) {

comparisons++;

if (arr[right] < arr[smallest]) {

smallest = right;

}

}

if (smallest != i) {

swap(arr, i, smallest);

heapify(arr, n, smallest);

}

}

// Function to build a min heap from the array

void buildHeap(int arr[], int n) {

for (int i = n / 2 - 1; i >= 0; i--) {

heapify(arr, n, i);

}

}

// Function to sort the array using heap sort

void heapSort(int arr[], int n) {

// Reseting the comparisons counter each time

comparisons = 0;

buildHeap(arr, n);

for (int i = n - 1; i > 0; i--) {

swap(arr, 0, i);

heapify(arr, i, 0);

}

}

// Function for selection sort

void selectionSort(int arr[], int n) {

comparisons = 0;

int i, j, min\_idx;

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

min\_idx = i;

for (j = i + 1; j < n; j++) {

comparisons++;

if (arr[j] < arr[min\_idx])

min\_idx = j;

}

swap(arr[min\_idx], arr[i]);

}

}

// Bubble Sort function

void bubbleSort(int arr[], int n) {

comparisons = 0;

int i, j;

bool swapped;

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

swapped = false;

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

comparisons++;

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

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

swapped = true;

}

}

if (!swapped)

break;

}

}

// Insertion Sort function

void insertionSort(int arr[], int n) {

comparisons = 0;

int i, key, j;

for (i = 1; i < n; i++) {

key = arr[i];

j = i - 1;

while (j >= 0 && arr[j] > key) {

comparisons++;

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

j = j - 1;

}

if (j >= 0) comparisons++;

arr[j + 1] = key;

}

}

void regenerateArrays(int randomArray[], int sortedArray[], int inverselySortedArray[], int n) {

// Generate random array

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

randomArray[j] = rand() % 100;

}

// Generate sorted array

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

sortedArray[j] = j;

}

// Generate inversely sorted array

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

inverselySortedArray[j] = n - j - 1;

}

}

void test\_comparisons(ofstream& csvFile) {

// This is our random number generator

srand(time(NULL));

// We initialize a variable to set a size for our array

const int n = 30;

// Now we initialize a variable to set the number of test cases

const int testCases = 30;

// Print headers for the output in the console

cout << left << setw(25) << "Algorithm" << setw(20) << "Array Type" << setw(15) << "Comparisons" << setw(15) << "Time(ns)" << endl;

cout << string(75, '-') << endl;

// Here we create a loop for each test case

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

int randomArray[n], sortedArray[n], inverselySortedArray[n];

// Initial array generation

regenerateArrays(randomArray, sortedArray, inverselySortedArray, n);

// Creating and opening our csv file that we will later on write to (I am pretty sure I can remove this line since I already call it in the main but I receive an error in the next line)

std::ofstream outFile("sorting\_performance.csv");

// Writing at the top of the csv file our headers

outFile << "Algorithm,Comparisons,Time(nanoseconds)\n";

// Starting our timer for this execution

auto start = high\_resolution\_clock::now();

// Calling the function

heapSort(randomArray, n);

// As soon as our function is called and finished executing we stop the timer

auto stop = high\_resolution\_clock::now();

// Saving our time total time in nanoseconds in a variable called durationHeapRandom

auto durationHeapRandom = duration\_cast<nanoseconds>(stop - start);

// Now we write to our csv file our data with the number of comparison seperated by a comma with the calculated time

csvFile << "\nHeapSort Random Array," << comparisons << "," << durationHeapRandom.count() << "\n";

// And now we also write to the console our results

cout << left << setw(25) << "HeapSort" << setw(20) << "Random Array" << setw(15) << comparisons << setw(15) << durationHeapRandom.count() << "ns" << endl;

// For the next few lines of code the process is exactly the same as shown above so it would be redundant to re-explain everything over and over again.

auto start2 = high\_resolution\_clock::now();

regenerateArrays(randomArray, sortedArray, inverselySortedArray, n);

heapSort(sortedArray, n);

auto stop2 = high\_resolution\_clock::now();

auto durationHeapSorted = duration\_cast<nanoseconds>(stop2 - start2);

csvFile << "HeapSort Sorted Array," << comparisons << "," << durationHeapSorted.count() << "\n";

cout << left << setw(25) << "HeapSort" << setw(20) << "Sorted Array" << setw(15) << comparisons << setw(15) << durationHeapSorted.count() << "ns" << endl;

auto start3 = high\_resolution\_clock::now();

regenerateArrays(randomArray, sortedArray, inverselySortedArray, n);

heapSort(inverselySortedArray, n);

auto stop3 = high\_resolution\_clock::now();

auto durationHeapInverse = duration\_cast<nanoseconds>(stop3 - start3);

csvFile << "HeapSort Inverse Array," << comparisons << "," << durationHeapInverse.count() << "\n";

cout << left << setw(25) << "HeapSort" << setw(20) << "Inverse Array" << setw(15) << comparisons << setw(15) << durationHeapInverse.count() << "ns" << endl;

// InsertionSort

auto start4 = high\_resolution\_clock::now();

regenerateArrays(randomArray, sortedArray, inverselySortedArray, n);

insertionSort(randomArray, n);

auto stop4 = high\_resolution\_clock::now();

auto durationInsertionRandom = duration\_cast<nanoseconds>(stop4 - start4);

csvFile << "Insertion Random Array," << comparisons << "," << durationInsertionRandom.count() << "\n";

cout << left << setw(25) << "Insertion" << setw(20) << "Random Array" << setw(15) << comparisons << setw(15) << durationInsertionRandom.count() << "ns" << endl;

auto start5 = high\_resolution\_clock::now();

insertionSort(sortedArray, n);

auto stop5 = high\_resolution\_clock::now();

auto durationInsertionSorted = duration\_cast<nanoseconds>(stop5 - start5);

csvFile << "Insertion Sorted Array," << comparisons << "," << durationInsertionSorted.count() << "\n";

cout << left << setw(25) << "Insertion" << setw(20) << "Sorted Array" << setw(15) << comparisons << setw(15) << durationInsertionSorted.count() << "ns" << endl;

auto start6 = high\_resolution\_clock::now();

insertionSort(inverselySortedArray, n);

auto stop6 = high\_resolution\_clock::now();

auto durationInsertionInverse = duration\_cast<nanoseconds>(stop6 - start6);

csvFile << "Insertion Inverse Array," << comparisons << "," << durationInsertionInverse.count() << "\n";

cout << left << setw(25) << "Insertion" << setw(20) << "Inverse Array" << setw(15) << comparisons << setw(15) << durationInsertionInverse.count() << "ns" << endl;

// BubbleSort

auto start7 = high\_resolution\_clock::now();

regenerateArrays(randomArray, sortedArray, inverselySortedArray, n);

bubbleSort(randomArray, n);

auto stop7 = high\_resolution\_clock::now();

auto durationBubbleRandom = duration\_cast<nanoseconds>(stop7 - start7);

csvFile << "Bubble Random Array," << comparisons << "," << durationBubbleRandom.count() << "\n";

cout << left << setw(25) << "Bubble" << setw(20) << "Random Array" << setw(15) << comparisons << setw(15) << durationBubbleRandom.count() << "ns" << endl;

auto start8 = high\_resolution\_clock::now();

bubbleSort(sortedArray, n);

auto stop8 = high\_resolution\_clock::now();

auto durationBubbleSorted = duration\_cast<nanoseconds>(stop8 - start8);

csvFile << "Bubble Sorted Array," << comparisons << "," << durationBubbleSorted.count() << "\n";

cout << left << setw(25) << "Bubble" << setw(20) << "Sorted Array" << setw(15) << comparisons << setw(15) << durationBubbleSorted.count() << "ns" << endl;

auto start9 = high\_resolution\_clock::now();

bubbleSort(inverselySortedArray, n);

auto stop9 = high\_resolution\_clock::now();

auto durationBubbleInverse = duration\_cast<nanoseconds>(stop9 - start9);

csvFile << "Bubble Inverse Array," << comparisons << "," << durationBubbleInverse.count() << "\n";

cout << left << setw(25) << "Bubble" << setw(20) << "Inverse Array" << setw(15) << comparisons << setw(15) << durationBubbleInverse.count() << "ns" << endl;

// SelectionSort

auto start10 = high\_resolution\_clock::now();

regenerateArrays(randomArray, sortedArray, inverselySortedArray, n);

selectionSort(randomArray, n);

auto stop10 = high\_resolution\_clock::now();

auto durationSelectionRandom = duration\_cast<nanoseconds>(stop10 - start10);

csvFile << "Selection Random Array," << comparisons << "," << durationSelectionRandom.count() << "\n";

cout << left << setw(25) << "Selection" << setw(20) << "Random Array" << setw(15) << comparisons << setw(15) << durationSelectionRandom.count() << "ns" << endl;

auto start11 = high\_resolution\_clock::now();

selectionSort(sortedArray, n);

auto stop11 = high\_resolution\_clock::now();

auto durationSelectionSorted = duration\_cast<nanoseconds>(stop11 - start11);

csvFile << "Selection Sorted Array," << comparisons << "," << durationSelectionSorted.count() << "\n";

cout << left << setw(25) << "Selection" << setw(20) << "Sorted Array" << setw(15) << comparisons << setw(15) << durationSelectionSorted.count() << "ns" << endl;

auto start12 = high\_resolution\_clock::now();

selectionSort(inverselySortedArray, n);

auto stop12 = high\_resolution\_clock::now();

auto durationSelectionInverse = duration\_cast<nanoseconds>(stop12 - start12);

csvFile << "Selection Inverse Array," << comparisons << "," << durationSelectionInverse.count() << "\n";

cout << left << setw(25) << "Selection" << setw(20) << "Inverse Array" << setw(15) << comparisons << setw(15) << durationSelectionInverse.count() << "ns" << endl;

}

}

int main() {

// Create and opening a csv file for writing

ofstream csvFile("sorting\_performance.csv");

// Write a header line to the CSV file.

csvFile << "Algorithm,Array Type,Test Case,Comparisons,Time(ms)" << endl;

// Here we simply call our function to execute

test\_comparisons(csvFile);

// Closing the file after we are done

csvFile.close();

}