Sorting Algorithms

Prepared by:
Ahmed Abbady Mohamed 22P0308
Ezzeldin Ismail 22P0141
Omar Mohamed Mostafa 22P0197
Ahmed Wael Raafat 22P0221
Anas Mansour 22U0005

Course Coordinator:
Dr. Hesham Farag
Eng. Zead Hani

Data Structures and Algorithms
CSE331
December 28, 2024

Anas Manosur doesn’t have an id as he is a single degree student

Table of Contents

1. Introduction

Sorting algorithms are essential tools in computer science, used to arrange data in a specific order. The efficiency of these algorithms is measured by how quickly they can sort data, which can vary depending on the algorithm and the data set.

This project creates an application to compare the efficiency of different sorting algorithms. The application will allow users to choose an algorithm and compare its performance with another algorithm or with its expected behavior.

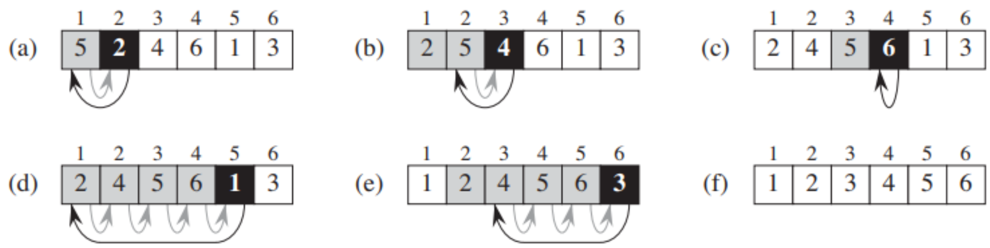
Users can generate test data, run the algorithms, and visualize the results in a graph. This project aims to help users better understand how different sorting algorithms perform and how their efficiency compares in practice.

1. Algorithms

1.1. Insertion Sort

Insertion Sort builds the sorted array one element at a time by repeatedly taking the next unsorted element and inserting it into its correct position within the sorted portion of the array.

It starts with the second element in the array as the first element is trivially sorted after it picks the element it compares it to the ones before. If the element is smaller than the one it is compared to the larger element is shifted one position to the right and after all the elements are compared our element is inserted in its right position then move the index to the next element and repeat



Code:

```cpp
vector<int> insertionsort(vector<int> v, int l, int r, vector<pair<int, int>> &insertioncsv) {
    int t = 0;

    for (int i = l + 1; i <= r; i++) {
        int key = v[i];
        t++;
        int j = i - 1;
        t++;
        while (j >= l && v[j] > key) {
            v[j + 1] = v[j];
            t++;
            j -= 1;
            t++;
        }
        v[j + 1] = key;
        t++;
    }
    int n = r - l + 1;
    insertioncsv.push_back({ n, t });
    return v;
}
```

Time complexity:

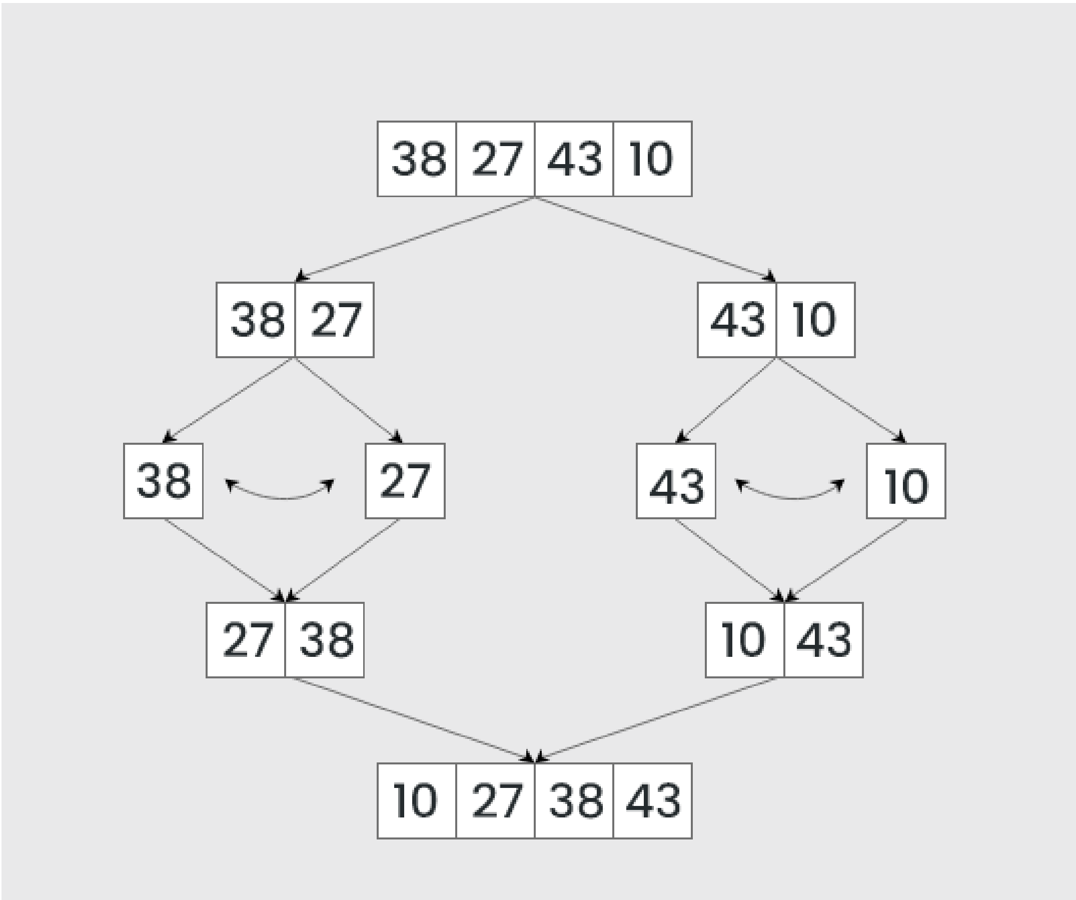
- Best Case = O(n)
- Average Case = O(n^2)
- Worst Case = O(n^2)

1.2. Merge Sort

Merge sort is a recursive sorting algorithm that follows the divide-and-conquer principle. It works by:

1. Dividing the input array into two halves recursively until each subarray contains one element
2. Merging these subarrays back together in sorted order by comparing elements
3. Continuing to merge until the entire array is sorted

The Sorting process:

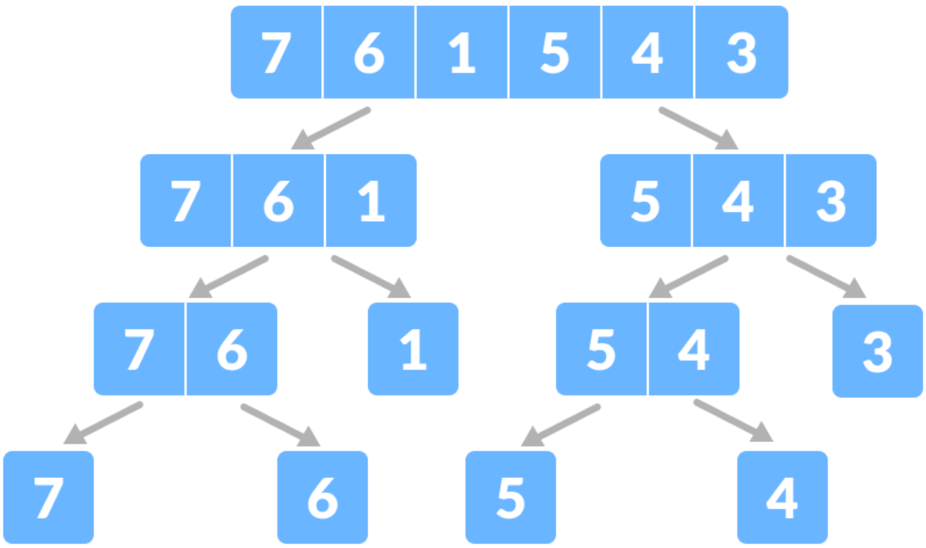


Code:

Merge Sort code is divided into 2 codes the first code (Merge_Sort) is responsible for dividing the array and then calling the other code

```
void MergeSort(vector<int>& arr, int left, int right, int& t_merge) {
    if (left >= right) {
        t_merge++;
        return;
    }

    int mid = left + (right - left) / 2;
    t_merge++;
    MergeSort(arr, left, mid, t_merge);
    t_merge++;

    MergeSort(arr, mid + 1, right, t_merge);
    t_merge++;

    Merge(arr, left, mid, right, t_merge);
    t_merge++;
}
```

Merge_Sort code process



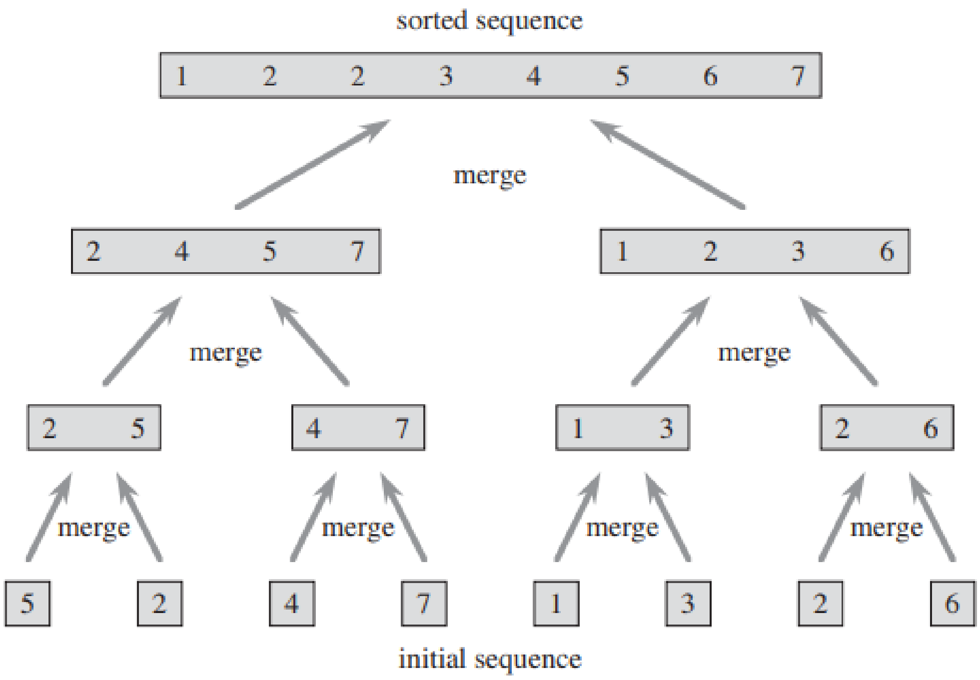
The second function (Merge) is responsible for combining the divided elements while sorting them

```
void Merge(vector<int>& arr, int left, int mid, int right, int& t_merge) {
    int n = mid - left + 1;
    t_merge++;
    int m = right - mid;
    t_merge++;
    vector<int> L(n), R(m);

    for (int i = 0; i < n; i++) {
        t_merge++;
        L[i] = arr[left + i];
        t_merge++;
    }

    for (int j = 0; j < m; j++) {
        t_merge++;
        R[j] = arr[mid + 1 + j];
        t_merge++;
    }
```

```
    int i = 0, j = 0, k = left;
    t_merge++;
    while (i < n && j < m) {
        t_merge++;
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            t_merge++;
            i++;
            t_merge++;
        }
        else {
            arr[k] = R[j];
            t_merge++;
            j++;
            t_merge++;
        }
        k++;
        t_merge++;
    }
```

```
    while (i < n) {
        arr[k] = L[i];
        t_merge++;
        i++;
        t_merge++;
        k++;
        t_merge++;
    }

    while (j < m) {
        arr[k] = R[j];
        t_merge++;
        j++;
        t_merge++;
        k++;
        t_merge++;
    }
}
```

Merge code process:



Then the main function that calls the merge sort, initialize the time and is responsible for publishing the values into the excel sheet

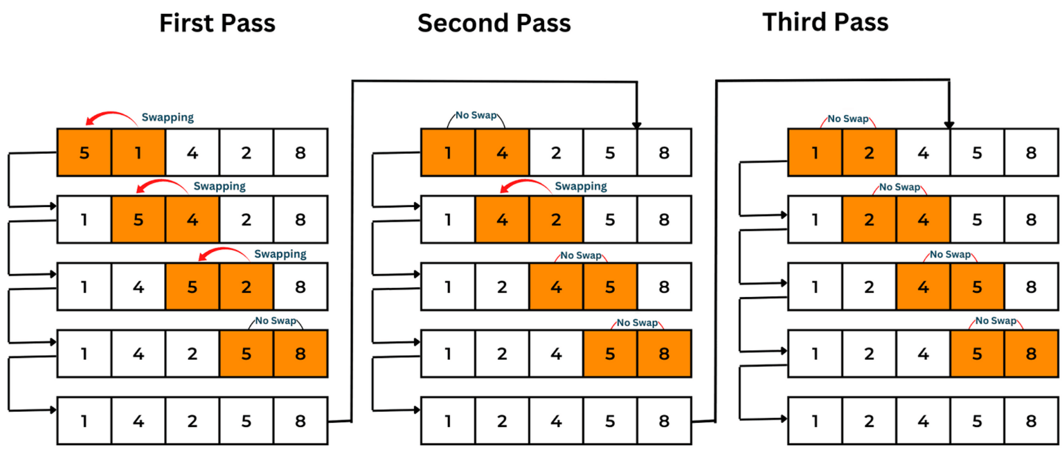
```
void mymerge(vector<int>& arr, int l, int r, vector<pair<int, int>>& mergecsv, int& t_merge) {
    t_merge = 0;
    MergeSort(arr, l, r, t_merge);
    mergecsv.push_back({ r - l + 1, t_merge });
}
```

Time complexity = O(nlgn) for all cases

1.3. Bubble Sort

Bubble Sort algorithm is a simple algorithm that compares every 2 adjacent elements and swaps them if they are in the wrong order and repeats this loop again for n times. At the end of every loop the largest element is at its right position at the end.

The Sorting process:



In our code we started from index 1 (the second element) so we looped for n (size of the array) times

```
vector<int> bubblesort(vector<int> v, int l, int r, vector<pair<int, int>>& bubblecsv) {
    int t = 0;

    for (int i = l; i < r; i++) {
        t++;
        for (int j = l; j < r - i; j++) {
            t++;
            if (v[j] > v[j + 1]) {
                swap(v[j], v[j + 1]);
                t++;
            }
        }
    }
    int n = r - l + 1;
    bubblecsv.push_back({ n, t });
    return v;
}
```

Time complexity = $O(n^2)$ for all cases

1.4. Quick Sort

Quick Sort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

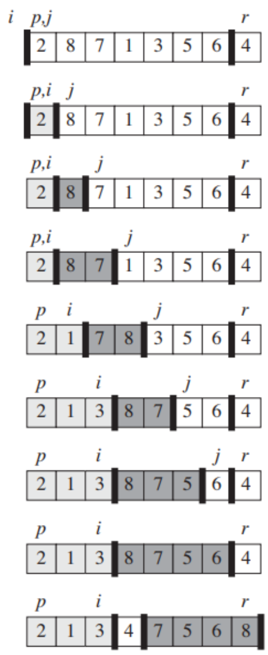
There are mainly three steps in the algorithm:

1. Choose a Pivot: Select an element from the array as the pivot. In our code we choose the last element.

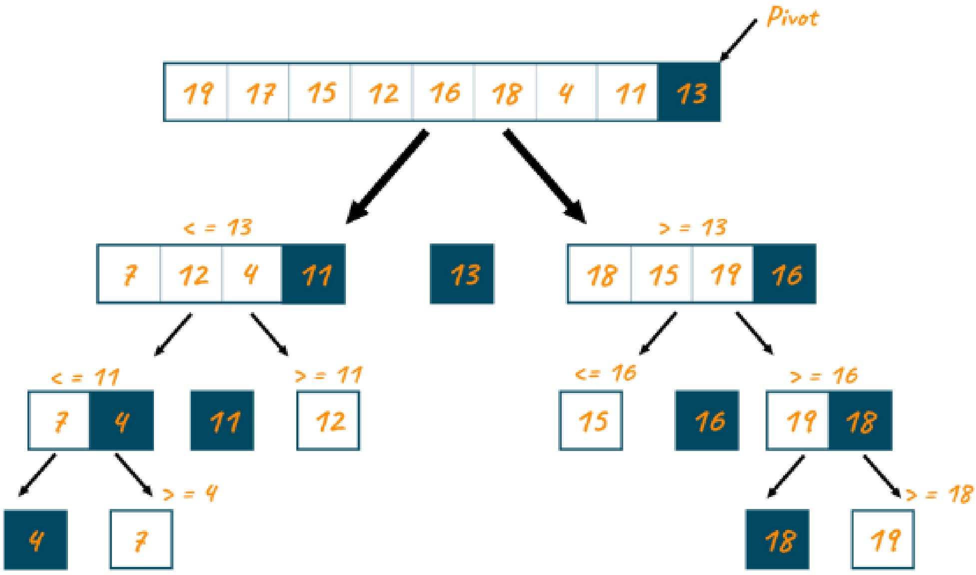
2. Partition the Array: Rearrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. The pivot is then in its correct position, and we obtain the index of the pivot.

3. Recursively Call: Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).

Example for the first partitioning on the last element in the array:



The whole sorting process:



Code:

The code is divided into the parts the first part (QuickSort) is responsible for dividing the array and calling the second code

```
void QuickSort(vector<int>& arr, int low, int high, int& t_quick) {
    t_quick++;
    if (low < high) {
        int pi = Partition(arr, low, high, t_quick);
        t_quick++;
        QuickSort(arr, low, pi - 1, t_quick);
        t_quick++;
        QuickSort(arr, pi + 1, high, t_quick);
        t_quick++;
    }
}
```

The second code is the partitioning code that performs the partitioning process we talked about above

```
int Partition(vector<int>& arr, int low, int high, int& t_quick) {
    int pivot = arr[high];
    t_quick++;
    int i = (low - 1);
    t_quick++;
    for (int j = low; j <= high - 1; j++) {
        t_quick++;
        if (arr[j] <= pivot) {
            i++;
            t_quick++;
            swap(arr[i], arr[j]);
            t_quick++;
        }
    }
    swap(arr[i + 1], arr[high]);
    t_quick++;
    return (i + 1);
    t_quick++;
}
```

Then the main function that calls the quick sort algorithm, initializes the time and publish the values into the excel sheet

```
void myquick(vector<int>& arr, int l, int r, vector<pair<int, int>>& quickcsv, int& t_quick) {
    QuickSort(arr, l, r, t_quick);
    quickcsv.push_back({ r - l + 1, t_quick });
}
```

Time complexity:

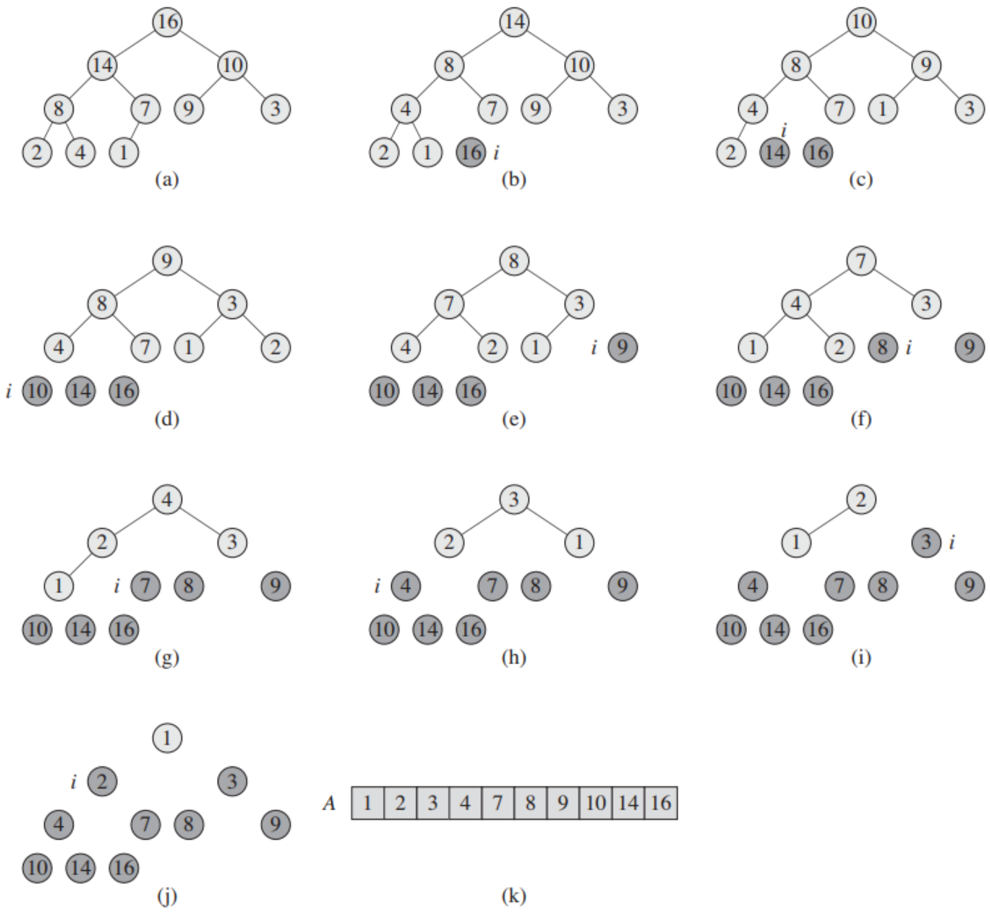
- Best Case = O(nlgn)
- Average Case = O(nlgn)
- Worst Case = $O(n^2)$

1.5. Heap Sort

The heap sort starts by taking an array and converting it to max heap then since the maximum element of the array is stored at the root (index 1) we can put it into its correct final position by exchanging it with the last element at index n.

After that decrement the heap size to eliminate the last element from the heap and restore the max heap property and repeat this operation until they are all sorted

The sorting process:



Code:

The code is divided into 2 codes. The first one (heapsort) is responsible for calling the second code (Heapify) to create max heap from the array then exchanges the last element with the index and calls the Heapify code on the new root to restore the max heap property as discussed above.

```
void heapsort(vector<int>& arr, int l, int r, vector<pair<int, int>>& heapcsv, int& t_heap) {
    int n = r - l + 1;
    vector<int> subarr(arr.begin() + l, arr.begin() + r + 1);

    for (int i = n / 2 - 1; i >= 0; i--) {
        Heapify(subarr, n, i, t_heap);
    }

    for (int i = n - 1; i > 0; i--) {
        swap(subarr[0], subarr[i]);
        t_heap++;
        Heapify(subarr, i, 0, t_heap);
    }

    for (int i = l; i <= r; i++) {
        arr[i] = subarr[i - l];
    }

    heapcsv.push_back({ n, t_heap });
}
```

The second code (Heapify) is responsible for creating the max heap at the start and restoring the max heap property after each iteration

```
void Heapify(vector<int>& arr, int n, int i, int& t_heap) {
    int largest = i;
    t_heap++;
    int left = 2 * i + 1;
    t_heap++;
    int right = 2 * i + 2;
    t_heap++;
    if (left < n && arr[left] > arr[largest]) {
        largest = left;
        t_heap++;
    }

    if (right < n && arr[right] > arr[largest]) {
        largest = right;
        t_heap++;
    }

    if (largest != i) {
        swap(arr[i], arr[largest]);
        t_heap++;
        Heapify(arr, n, largest, t_heap);
        t_heap++;
    }
}
```

Time Complexity = O(nlgn) for all cases

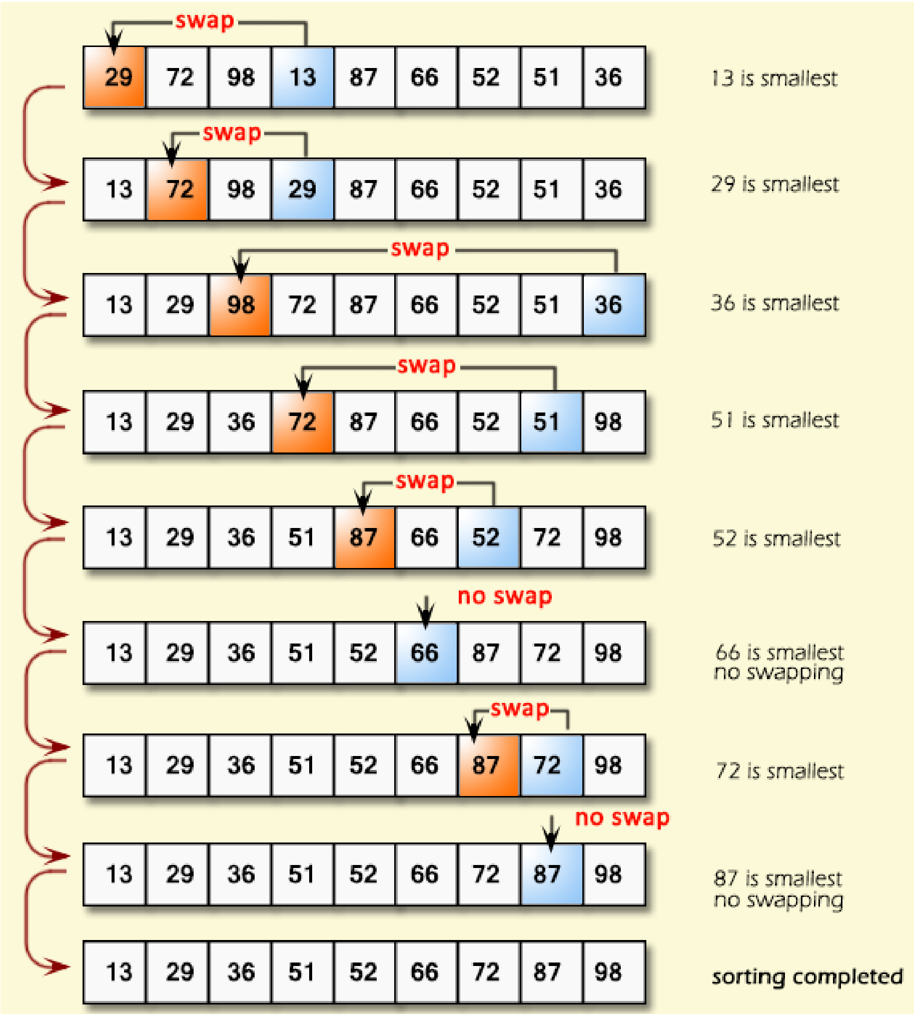
1.6. Selection Sort

Selection Sort is a comparison-based sorting algorithm. It sorts an array by repeatedly selecting the smallest element from the unsorted portion and swapping it with the first unsorted element. This process continues until the entire array is sorted.

First, we find the smallest element and swap it with the first element. This way we get the smallest element at its correct position.

Then we find the smallest among the remaining elements and swap it with the second element. We keep doing this until we get all elements moved to the correct position.

The Sorting process:



Code:

```
vector<int> SelectionSort(vector<int> arr, int l, int r, vector<pair<int, int>>& selectioncsv) {
    int t = 0;

    for (int i = l; i < r - 1; i++) {
        int MinIn = i;
        t++;
        for (int j = i + 1; j < r; j++) {
            t++;
            if (arr[j] < arr[MinIn]) {
                MinIn = j;
                t++;
            }
            t++;
        }
        swap(arr[i], arr[MinIn]);
        t++;
    }
    int n = r - l;
    selectioncsv.push_back({ n, t });
    return arr;
}
```

Time Complexity = $O(n^2)$ for all cases

1.7. Counting Sort

Counting Sort is a non-comparison-based sorting algorithm. It is particularly efficient when the range of input values is small compared to the number of elements to be sorted. The basic idea behind Counting Sort is to count the frequency of each distinct element in the input array and use that information to place the elements in their correct sorted positions.

1- Find out the maximum element from the given array

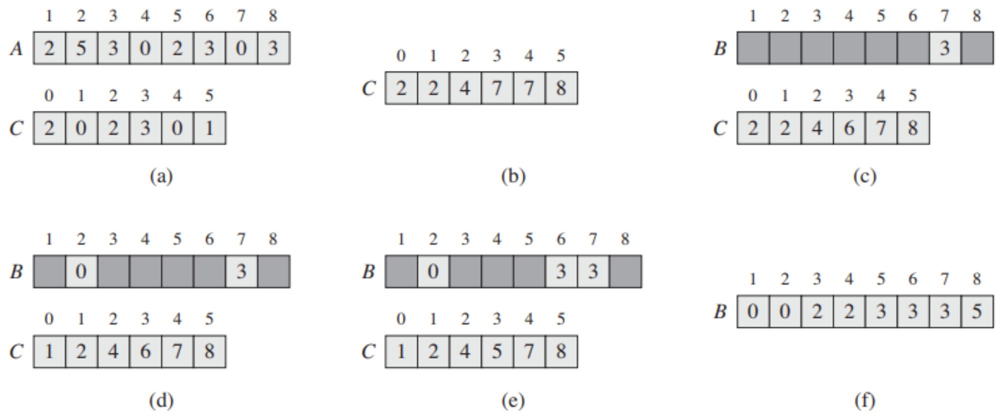
2- Initialize a count array of length of the maximum element then initialize all the array elements with 0. This array will be used for storing the occurrences of the elements of the input array.

3- In the count array, store the count of each unique element of the input array at their respective indices for example if there are 2 elements of number 3 in the main array the index 3 in the count array will be equal to 2

4- Store the cumulative sum or prefix sum of the elements of the count array by doing countArray[i] = countArray[i – 1] + countArray[i]

5- Make a new array to put the sorted array and iterate from the end of the main array and for each element search for its index in the count array and take the number in this index for example if it was 6 so go index 6 in the new array and put the number from the main array in this index then decrement this 6 in the count array

The sorting process:



Code:

```cpp
vector<int> countingSort(vector<int> A, int l, int r, vector<pair<int, int>>& countingcsv) {
    int t = 0;
    vector <int> B(A.size() + 1);
    t++;
    int maxA = A[0];
    t++;
    for (int i = 0; i < r + 1; i++) {
        if (A[i] > maxA) {
            t++;
            maxA = A[i];
        }
    }

    vector <int> C(maxA + 1, 0);
    t++;

    for (int i = 0; i < r + 1; i++) {
        t++;
        C[A[i]]++;
    }

    for (int i = 1; i < C.size(); i++) {
        t++;
        C[i] = C[i] + C[i - 1];
    }

    for (int i = r; i >= 0; i--) {
        t++;
        B[C[A[i]]] = A[i];
        t++;
        C[A[i]] = C[A[i]] - 1;
    }
    int n = r - l + 1;
    countingcsv.push_back({ n, t });

    return B;
}
```

Time Complexity = O(n + k) for all cases

2. Cases

2.1. Best Case

For the best case we initialized an array with n (the number inputted) and then iterate from the beginning of the array to the end and put the numbers in order

```
vector<int> best(int size) {
    vector<int> v;
    for (int i = 0; i < size; i++) {
        v.push_back(i);
    }
    return v;
}
```

2.2. Worst case

For the worst case we initialized an array with n (the number inputted) and then iterate from the end of the array to the beginning and put the numbers in reverse order (descending)

```
vector<int> worst(int size) {
    vector<int> v;
    for (int i = size; i > 0; i--) {
        v.push_back(i);
    }
    return v;
}
```

2.3. Average Case

For the average case we initialized an array with n (the number inputted) and then iterate from the beginning of the array to the end and put the random numbers in every index, besides we made sure that no number is repeated by using a flag

```
vector<int> avg(int size) {
    vector<int> v;
    vector<bool> taken(size, false);
    while (v.size() < size) {
        int x = rand() % size;
        if (!taken[x]) {
            v.push_back(x);
        }
        taken[x] = true;
    }
    return v;
}
```

3. Excel File

As seen in every of the algorithms above that they all have common inputs which are ‘l’, ‘r’ and a csv file.

For the ‘l’ it is the starting index which is always 0 and the ‘r’ is the end index which varies every iteration as when we input an array size so to output the timing we calculate the time for each n elements in the array then increase them every loop with a specific step for example if the size of the array is 500 and the step is 5 we start from r = 5 then the next loop r = 10 and continue till 500 so we calculate the time for every number of elements and accumulate them every loop.

For the csv file this is the excel file that is a pair vector type so each loop in the algorithm we can push inside the size (n) and the time (t).

This criteria works for every of our 3 cases and the step is calculated by size/100 and if the size/100 is a number below 1 then the step will be 1.

```
int step= max(1, (int)ceil(choices[2]/100.0));
```

Worst case excel:

```
vector<int> w = worst(choices[2]);
for (int i = step-1; i <= choices[2]; i += step) {
    vector<int> v = w;
    insertionsort(v, 0, i, insertioncsv);

    v = w;
    t_merge = 0;
    mymerge(v, 0, i, mergecsv, t_merge);

    v = w;
    bubblesort(v, 0, i, bubblecsv);

    v = w;
    t_quick = 0;
    myquick(v, 0, i, quickcsv, t_quick);

    v = w;
    t_heap = 0;
    heapsort(v, 0, i, heapcsv, t_heap);

    v = w;
    SelectionSort(v, 0, i, selectioncsv);
    v = w;
    countingSort(v, 0, i, countingcsv);
}
```

Best case excel:

```
vector<int> b = best(choices[2]);
for (int i = step-1; i <= choices[2]; i += step) {
    vector<int> v = b;
    insertionsort(v, 0, i, insertioncsv);

    v = b;
    t_merge = 0;
    mymerge(v, 0, i, mergecsv, t_merge);

    v = b;
    bubblesort(v, 0, i, bubblecsv);

    v = b;
    t_quick = 0;
    myquick(v, 0, i, quickcsv, t_quick);

    v = b;
    t_heap = 0;
    heapsort(v, 0, i, heapcsv, t_heap);

    v = b;
    SelectionSort(v, 0, i, selectioncsv);
    v = b;
    countingSort(v, 0, i, countingcsv);
}
```

Average case excel:

```
vector<int> a = avg(choices[2]);
for (int i = step-1; i <= choices[2]; i += step) {
    vector<int> v = a;
    insertionsort(v, 0, i, insertioncsv);

    v = a;
    t_merge = 0;
    mymerge(v, 0, i, mergecsv, t_merge);

    v = a;
    bubblesort(v, 0, i, bubblecsv);

    v = a;
    t_quick = 0;
    myquick(v, 0, i, quickcsv, t_quick);

    v = a;
    t_heap = 0;
    heapsort(v, 0, i, heapcsv, t_heap);

    v = a;
    SelectionSort(v, 0, i, selectioncsv);
    v = a;

    countingSort(v, 0, i, countingcsv);
}
```

4. Output

4.1. Comparing 2 algorithms

To compare 2 algorithms an array is made (choices) which stores 4 things which are the first algorithm, second algorithm, the size and finally the case.

We have 4 cases which compare the best case only, worst case only, average case only and compare all of them.

This is why above we put the size(n) as choices[2].

```
// Create a vector to store choices
std::vector<int> choices; // choices[alg1(0),alg2(1),n(2),case(3)]

// Get selected algorithms
for (int i = 0; i < algorithmCheckboxes.size(); ++i) {
    if (algorithmCheckboxes[i]->isChecked()) {
        choices.push_back(i + 1);
    }
}

// Add number of elements
choices.push_back(n);

// Determine case type
int caseType = 0;
if (ui->AverageCase->isChecked()) caseType = 1;
else if (ui->BestCase->isChecked()) caseType = 2;
else if (ui->WorstCase->isChecked()) caseType = 3;
else if (ui->AllCases->isChecked()) caseType = 4;

choices.push_back(caseType);
```

Choosing the algorithms:

```
switch (choices[0]) {
case 1:
    sort1_name = "Insertion Sort";
    vectorcomp1 = insertioncsv;
    break;
case 2:
    sort1_name = "Merge Sort";
    vectorcomp1 = mergecsv;
    break;
case 3:
    sort1_name = "Bubble Sort";
    vectorcomp1 = bubblecsv;
    break;
case 4:
    sort1_name = "Quick Sort";
    vectorcomp1 = quickcsv;
    break;
case 5:
    sort1_name = "Heap Sort";
    vectorcomp1 = heapcsv;
    break;
case 6:
    sort1_name = "Selection Sort";
    vectorcomp1 = selectioncsv;
    break;
case 7:
    sort1_name = "Counting Sort";
    vectorcomp1 = countingcsv;
    break;
default:
    sort1_name = "Insertion Sort";
    vectorcomp1 = insertioncsv;
}
```

```
switch (choices[1]) {
case 1:
    sort2_name = "Insertion Sort";
    vectorcomp2 = insertioncsv;
    break;
case 2:
    sort2_name = "Merge Sort";
    vectorcomp2 = mergecsv;
    break;
case 3:
    sort2_name = "Bubble Sort";
    vectorcomp2 = bubblecsv;
    break;
case 4:
    sort2_name = "Quick Sort";
    vectorcomp2 = quickcsv;
    break;
case 5:
    sort2_name = "Heap Sort";
    vectorcomp2 = heapcsv;
    break;
case 6:
    sort2_name = "Selection Sort";
    vectorcomp2 = selectioncsv;
    break;
case 7:
    sort2_name = "Counting Sort";
    vectorcomp2 = countingcsv;
    break;
default :
    sort2_name = "Insertion Sort";
    vectorcomp2 = insertioncsv;
}
```

Outputting the excel and graph:

```
std::pair<QString, QString> paths;
if (choices[3] == 1) {
    paths=csvavg(vectorcomp1, vectorcomp2, sort1_name, sort2_name);
}
else if (choices[3] == 2) {
    paths=csvbest(vectorcomp1, vectorcomp2, sort1_name, sort2_name);
}
else if (choices[3] == 3) {
    paths=csvworst(vectorcomp1, vectorcomp2, sort1_name, sort2_name);
}
else if (choices[3] == 4) {
    paths=csvall(vectorcomp1, vectorcomp2, sort1_name, sort2_name);
}
```

Functions csvavg, csvbest, csvworst and csvall are responsible for generating the excel sheets and graphs and getting their paths

csvavg:

```
std::pair<QString, QString> csvavg(vector<pair<int, int>> sort1, vector<pair<int, int>> sort2, string sort1_name, string sort2_name) {
    ofstream outFile("compare_avg_case.csv");
    ofstream htmlFile("avg_chart.html");
    pair<QString, QString> paths;

    QString currentPath = QDir::currentPath();
    QDir dir(currentPath);

    if (!dir.exists()) {
        if (!dir.mkpath(".")) {
            currentPath = QDir::tempPath();
            dir = QDir(currentPath);
        }
    }

    if (!QFileInfo(currentPath).isWritable()) {
        currentPath = QDir::tempPath();
        dir = QDir(currentPath);
    }

    QString csvPath = dir.filePath("compare_avg_case.csv");
    QString htmlPath = dir.filePath("avg_chart.html");

    outFile << "n,Average_" << sort1_name << "t,Average" << sort2_name << "_t" << el;

    // Determine the size based on the smallest input size
    int size = std::min(sort1.size(), sort2.size());

    // Dynamically calculate part size for proper distribution
    int part_size = size / 3;
```

```
    // Ensure rows correspond exactly to the input size
    for (int i = 0; i < part_size; i++) {
        outFile << sort1[i].first << ","; // Write n
        outFile << sort1[i].second << "," << sort2[i].second << el; // Write timings
    }

    // Write HTML content
    htmlFile << "<!DOCTYPE html><html><head><script src=\"https://cdn.jsdelivr.net/npm/chart.js\"></script></head><body>";
    htmlFile << "<canvas id=\"avgChart\" width=\"800\" height=\"400\"></canvas>";
    htmlFile << "<script>var ctx = document.getElementById('avgChart').getContext('2d');";
    htmlFile << "var myChart = new Chart(ctx, {type: 'line',data: {labels: [";

    for (int i = 0; i < part_size; i++) {
        htmlFile << sort1[i].first;
        if (i != part_size - 1) htmlFile << ",";
    }

    htmlFile << "],datasets: [";

    // Add first dataset
    htmlFile << "{label: '" << sort1_name << " Average',data: [";
    for (int i = 0; i < part_size; i++) {
        htmlFile << sort1[i].second;
        if (i != part_size - 1) htmlFile << ",";
    }
    htmlFile << "],borderColor: 'blue',fill: false},";

    // Add second dataset
    htmlFile << "{label: '" << sort2_name << " Average',data: [";
    for (int i = 0; i < part_size; i++) {
        htmlFile << sort2[i].second;
        if (i != part_size - 1) htmlFile << ",";
    }
    htmlFile << "],borderColor: 'green',fill: false}";
```

```
    htmlFile << "]},options: {scales: {y: {beginAtZero: true}}}});</script></body></html>";

    outFile.close();
    htmlFile.close();

    cout << "CSV and HTML for Average Case comparison generated: compare_avg_case.csv, avg_chart.html" << el;
    paths.first = csvPath;
    paths.second = htmlPath;
    return paths;
}
```

csvbest:

```
std::pair<QString, QString> csvbest(vector<pair<int, int>> sort1, vector<pair<int, int>> sort2, string sort1_name, string sort2_name) {
    ofstream outFile("compare_best_case.csv");
    ofstream htmlFile("best_chart.html");
    pair<QString, QString> paths;

    QString currentPath = QDir::currentPath();
    QDir dir(currentPath);

    if (!dir.exists()) {
        if (!dir.mkpath(".")) {
            currentPath = QDir::tempPath();
            dir = QDir(currentPath);
        }
    }

    if (!QFileInfo(currentPath).isWritable()) {
        currentPath = QDir::tempPath();
        dir = QDir(currentPath);
    }

    QString csvPath = dir.filePath("compare_best_case.csv");
    QString htmlPath = dir.filePath("best_chart.html");

    outFile << "n,Best_" << sort1_name << "t,Best" << sort2_name << "_t" << el;

    // Determine the size based on the smallest input size
    int size = std::min(sort1.size(), sort2.size());

    // Dynamically calculate part size for proper distribution
    int part_size = size / 3;
```

```cpp
    // Ensure rows correspond exactly to the input size
    for (int i = 0; i < part_size; i++) {
        outFile << sort1[i].first << ","; // Write n
        outFile << sort1[i].second << "," << sort2[i].second << el; // Write timings
    }

    // Write HTML content
    htmlFile << "<!DOCTYPE html><html><head><script src=\"https://cdn.jsdelivr.net/npm/chart.js\"></script></head><body>";
    htmlFile << "<canvas id=\"avgChart\" width=\"800\" height=\"400\"></canvas>";
    htmlFile << "<script>var ctx = document.getElementById('avgChart').getContext('2d');";
    htmlFile << "var myChart = new Chart(ctx, {type: 'line',data: {labels: [";

    for (int i = 0; i < part_size; i++) {
        htmlFile << sort1[i].first;
        if (i != part_size - 1) htmlFile << ",";
    }

    htmlFile << "],datasets: [";

    // Add first dataset
    htmlFile << "{label: '" << sort1_name << " Best',data: [";
    for (int i = 0; i < part_size; i++) {
        htmlFile << sort1[i].second;
        if (i != part_size - 1) htmlFile << ",";
    }
    htmlFile << "],borderColor: 'blue',fill: false},";

    // Add second dataset
    htmlFile << "{label: '" << sort2_name << " Best',data: [";
    for (int i = 0; i < part_size; i++) {
        htmlFile << sort2[i].second;
        if (i != part_size - 1) htmlFile << ",";
    }
    htmlFile << "],borderColor: 'green',fill: false}";
```

```cpp
    htmlFile << "]},options: {scales: {y: {beginAtZero: true}}}});</script></body></html>";

    outFile.close();
    htmlFile.close();

    cout << "CSV and HTML for Best Case comparison generated: compare_best_case.csv, best_chart.html" << el;
    paths.first = csvPath;
    paths.second = htmlPath;
    return paths;
}
```

csvworst:

```cpp
std::pair<QString, QString> csvworst(vector<pair<int, int>> sort1, vector<pair<int, int>> sort2, string sort1_name, string sort2_name) {
    ofstream outFile("compare_worst_case.csv");
    ofstream htmlFile("worst_chart.html");
    pair<QString, QString> paths;

    QString currentPath = QDir::currentPath();
    QDir dir(currentPath);

    if (!dir.exists()) {
        if (!dir.mkpath(".")) {
            currentPath = QDir::tempPath();
            dir = QDir(currentPath);
        }
    }

    if (!QFileInfo(currentPath).isWritable()) {
        currentPath = QDir::tempPath();
        dir = QDir(currentPath);
    }

    QString csvPath = dir.filePath("compare_worst_case.csv");
    QString htmlPath = dir.filePath("worst_chart.html");

    outFile << "n,Worst_" << sort1_name << "t,Worst" << sort2_name << "_t" << el;

    // Determine the size based on the smallest input size
    int size = std::min(sort1.size(), sort2.size());

    // Dynamically calculate part size for proper distribution
    int part_size = size / 3;
```

```cpp
    // Ensure rows correspond exactly to the input size
    for (int i = 0; i < part_size; i++) {
        outFile << sort1[i].first << ","; // Write n
        outFile << sort1[i].second << "," << sort2[i].second << el; // Write timings
    }

    // Write HTML content
    htmlFile << "<!DOCTYPE html><html><head><script src=\"https://cdn.jsdelivr.net/npm/chart.js\"></script></head><body>";
    htmlFile << "<canvas id=\"worstChart\" width=\"800\" height=\"400\"></canvas>";
    htmlFile << "<script>var ctx = document.getElementById('worstChart').getContext('2d');";
    htmlFile << "var myChart = new Chart(ctx, {type: 'line',data: {labels: [";

    for (int i = 0; i < part_size; i++) {
        htmlFile << sort1[i].first;
        if (i != part_size - 1) htmlFile << ",";
    }

    htmlFile << "],datasets: [";

    // Add first dataset
    htmlFile << "{label: '" << sort1_name << " Worst',data: [";
    for (int i = 0; i < part_size; i++) {
        htmlFile << sort1[i].second;
        if (i != part_size - 1) htmlFile << ",";
    }
    htmlFile << "],borderColor: 'blue',fill: false},";

    // Add second dataset
    htmlFile << "{label: '" << sort2_name << " Worst',data: [";
    for (int i = 0; i < part_size; i++) {
        htmlFile << sort2[i].second;
        if (i != part_size - 1) htmlFile << ",";
    }
    htmlFile << "],borderColor: 'green',fill: false}";
```

```cpp
    htmlFile << "]},options: {scales: {y: {beginAtZero: true}}}});</script></body></html>";

    outFile.close();
    htmlFile.close();

    cout << "CSV and HTML for Worst Case comparison generated: compare_worst_case.csv, worst_chart.html" << el;
    paths.first = csvPath;
    paths.second = htmlPath;
    return paths;
}
```

csvall:

```cpp
std::pair<QString, QString> csvall(vector<pair<int, int>> sort1, vector<pair<int, int>> sort2, string sort1_name, string sort2_name) {
    ofstream outFile("compare_all_cases.csv");
    ofstream htmlFile("all_cases_chart.html");
    pair<QString, QString> paths;

    QString currentPath = QDir::currentPath();
    QDir dir(currentPath);

    if (!dir.exists()) {
        if (!dir.mkpath(".")) {
            currentPath = QDir::tempPath();
            dir = QDir(currentPath);
        }
    }

    if (!QFileInfo(currentPath).isWritable()) {
        currentPath = QDir::tempPath();
        dir = QDir(currentPath);
    }

    QString csvPath = dir.filePath("compare_all_cases.csv");
    QString htmlPath = dir.filePath("all_cases_chart.html");

    outFile << "n,Worst_" << sort1_name << "t,Worst" << sort2_name << "t,Best" << sort1_name << "t,Best"
    << sort2_name << "t,Average" << sort1_name << "t,Average" << sort2_name << "_t" << el;

    int size = std::min(sort1.size(), sort2.size());
    int part_size = size / 3; // Dynamically calculate size for each case
```

```cpp
for (int i = 0; i < part_size; i++) {
    outFile << sort1[i].first << ",";
    outFile << sort1[i].second << "," << sort2[i].second << ",";
    outFile << (i + part_size < size ? sort1[i + part_size].second : 0) << ",";
    outFile << (i + part_size < size ? sort2[i + part_size].second : 0) << ",";
    outFile << (i + 2 * part_size < size ? sort1[i + 2 * part_size].second : 0) << ",";
    outFile << (i + 2 * part_size < size ? sort2[i + 2 * part_size].second : 0) << el;
}

htmlFile << "<!DOCTYPE html><html><head><script src=\"https://cdn.jsdelivr.net/npm/chart.js\"></script></head><body>";
htmlFile << "<canvas id=\"allCasesChart\" width=\"800\" height=\"400\"></canvas>";
htmlFile << "<script>var ctx = document.getElementById('allCasesChart').getContext('2d');";
htmlFile << "var myChart = new Chart(ctx, {type: 'line',data: {labels: [";

for (int i = 0; i < part_size; i++) {
    htmlFile << sort1[i].first;
    if (i != part_size - 1) htmlFile << ",";
}

htmlFile << "],datasets: [";

htmlFile << "{label: '" << sort1_name << " Worst',data: [";
for (int i = 0; i < part_size; i++) {
    htmlFile << sort1[i].second;
    if (i != part_size - 1) htmlFile << ",";
}
htmlFile << "],borderColor: 'red',fill: false},";

htmlFile << "{label: '" << sort2_name << " Worst',data: [";
for (int i = 0; i < part_size; i++) {
    htmlFile << sort2[i].second;
    if (i != part_size - 1) htmlFile << ",";
}
htmlFile << "],borderColor: 'blue',fill: false},";
```

```cpp
htmlFile << "{label: '" << sort1_name << " Best',data: [";
for (int i = 0; i < part_size; i++) {
    htmlFile << (i + part_size < size ? sort1[i + part_size].second : 0);
    if (i != part_size - 1) htmlFile << ",";
}
htmlFile << "],borderColor: 'green',fill: false},";

htmlFile << "{label: '" << sort2_name << " Best',data: [";
for (int i = 0; i < part_size; i++) {
    htmlFile << (i + part_size < size ? sort2[i + part_size].second : 0);
    if (i != part_size - 1) htmlFile << ",";
}
htmlFile << "],borderColor: 'yellow',fill: false},";

htmlFile << "{label: '" << sort1_name << " Average',data: [";
for (int i = 0; i < part_size; i++) {
    htmlFile << (i + 2 * part_size < size ? sort1[i + 2 * part_size].second : 0);
    if (i != part_size - 1) htmlFile << ",";
}
htmlFile << "],borderColor: 'purple',fill: false},";

htmlFile << "{label: '" << sort2_name << " Average',data: [";
for (int i = 0; i < part_size; i++) {
    htmlFile << (i + 2 * part_size < size ? sort2[i + 2 * part_size].second : 0);
    if (i != part_size - 1) htmlFile << ",";
}
htmlFile << "],borderColor: 'orange',fill: false}";

htmlFile << "]},options: {scales: {y: {beginAtZero: true}}}});</script></body></html>";

outFile.close();
htmlFile.close();
```

```cpp
    cout << "CSV and HTML for All Cases comparison generated: compare_all_cases.csv, all_cases_chart.html" << el;
    paths.first = csvPath;
    paths.second = htmlPath;
    return paths;
}
```

4.2. Comparing algorithm with its asymptotic efficiency

We calculated the big oh for every algorithm

```
for (int k = step-1; k < n; k+=step) {
    insertionO.push_back((k+1) * (k+1));
    mergeO.push_back((k+1)*(log2(k+1)));
    bubbleO.push_back((k+1) * (k+1));
    quickO.push_back((k+1) * (k+1));
    heapO.push_back((k+1) * (log2(k+1)));
    selectionO.push_back((k+1) * (k+1));
    countingO.push_back((k+1)+(k+1));
}
```

Then we used an integer (Alg) that changed according to the input user to specify the algorithm chosen

```
if(ui->InsertionSort->isChecked()) Alg=1;
if(ui->MergeSort->isChecked()) Alg=2;
if(ui->BubbleSort->isChecked()) Alg=3;
if(ui->QuickSort->isChecked()) Alg=4;
if(ui->HeapSort->isChecked()) Alg=5;
if(ui->SelectionSort->isChecked()) Alg=6;
if(ui->CountingSort->isChecked()) Alg=7;
```

The cases for each algorithm is calculated exactly as the first case

Then according to the integer (Alg) we output the results of a specific algorithm with its big oh

```
switch (Alg) {
case 1:
    sortO_name = "insertion sort";
    path=csvbigO(insertioncsv, insertionO,sortO_name);
    BigO="O(n^2)";
    break;
case 2:
    sortO_name = "merge sort";
    path=csvbigO(mergecsv, mergeO, sortO_name);
    BigO="O(nlg(n))";
    break;
case 3:
    sortO_name = "bubble sort";
    path=csvbigO(bubblecsv, bubbleO, sortO_name);
    BigO="O(n^2)";
    break;
case 4:
    sortO_name = "quick sort";
    path=csvbigO(quickcsv, quickO, sortO_name);
    BigO="O(n^2)";
    break;
case 5:
    sortO_name = "heap sort";
    path=csvbigO(heapcsv, heapO, sortO_name);
    BigO="O(nlg(n))";
    break;
```

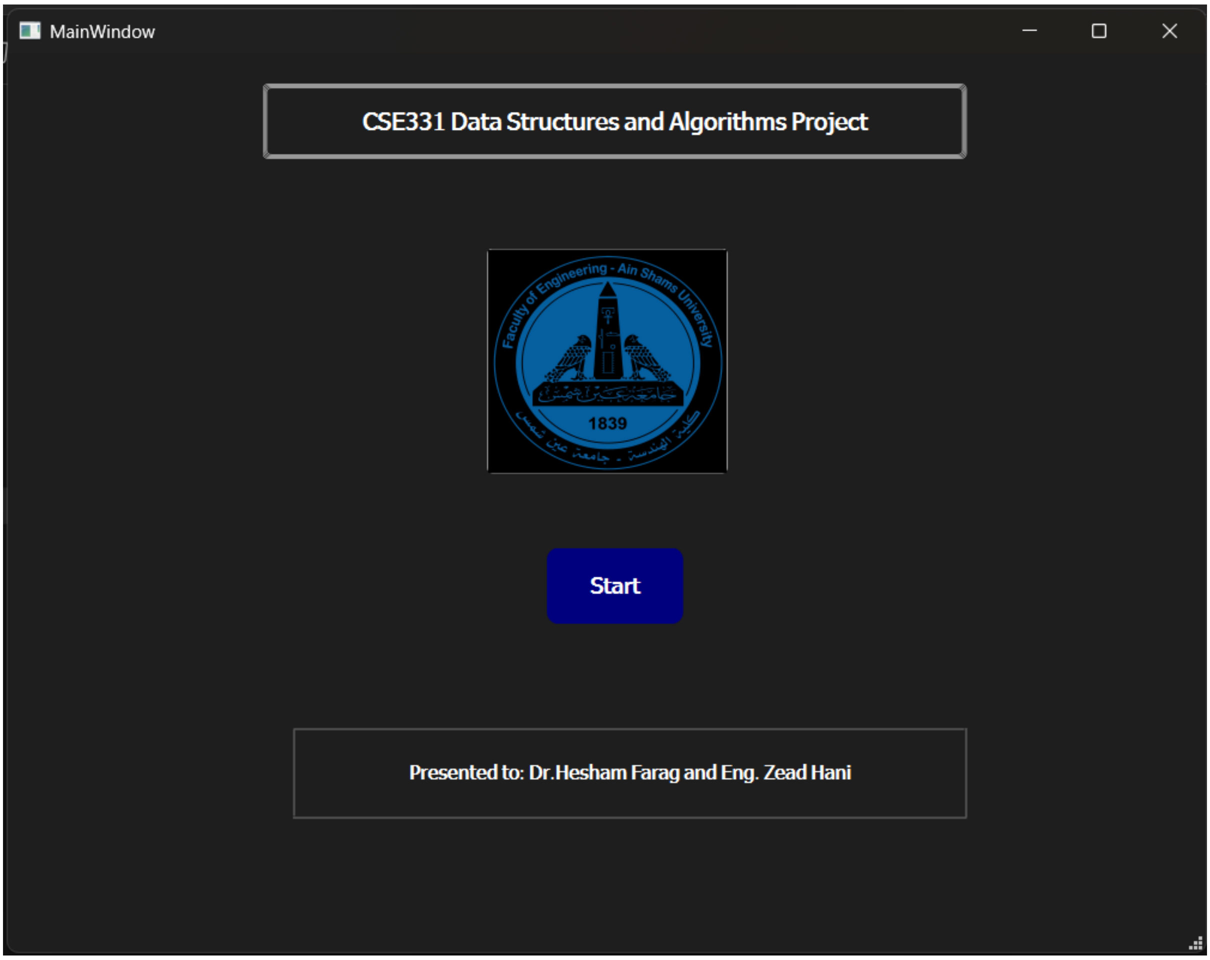
```
case 6:
    sortO_name = "selection sort";
    path=csvbigO(selectioncsv, selectionO, sortO_name);
    BigO="O(n^2)";
    break;
case 7:
    sortO_name = "counting";
    path=csvbigO(countingcsv, countingO, sortO_name);
    BigO="O(n+k); k being range of data";
    break;
default:
    path=csvbigO(insertioncsv, insertionO, sortO_name);
}
```

Function csvbigO is responsible for generating the excel sheets and graphs and getting their paths

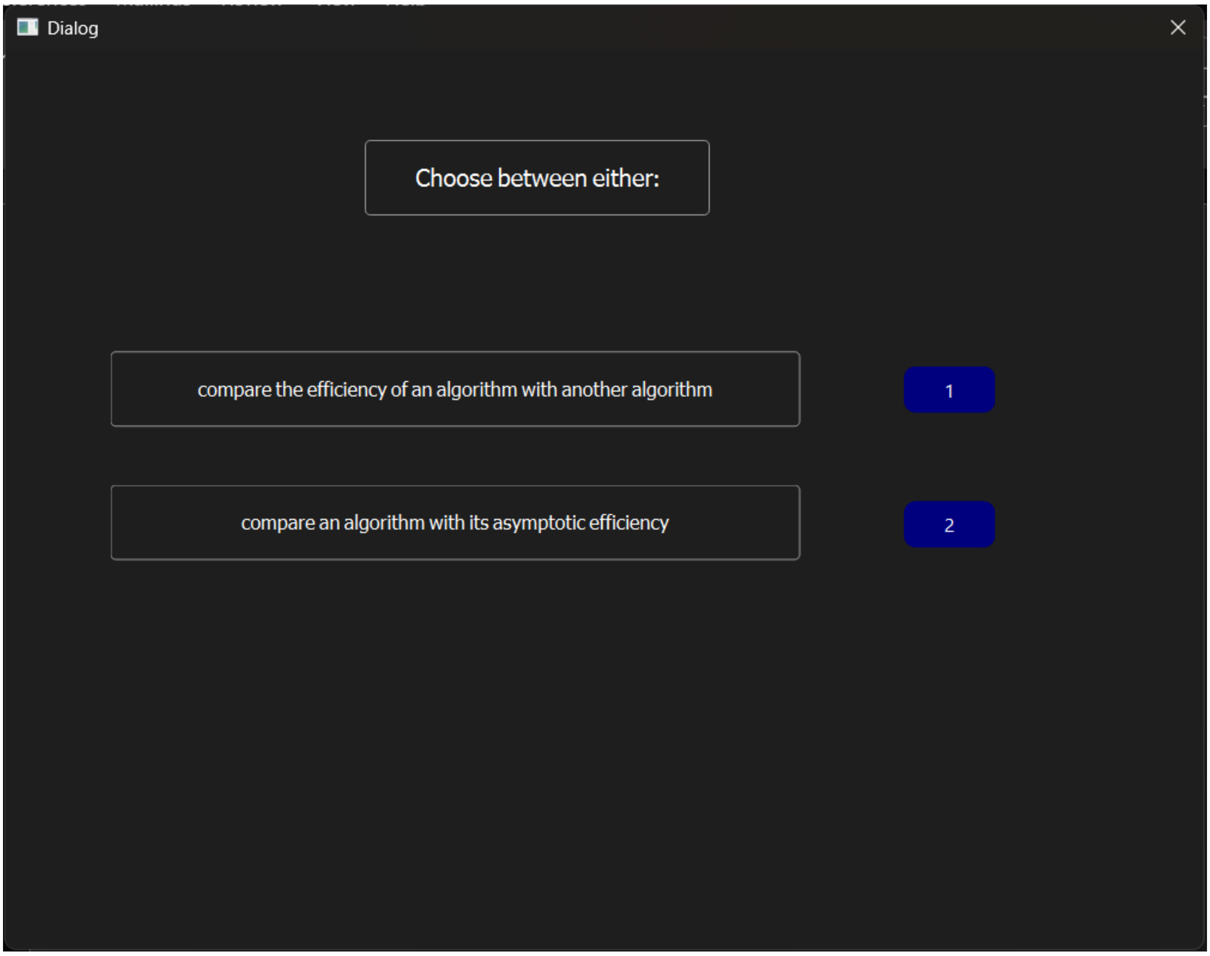
```
std::pair<QString, QString> csvbigO(vector<pair<int, int>> sortData, vector<int> bigOData, string sortO_name) {
    ofstream outFile("compare_sort_bigO.csv");
    ofstream htmlFile("bigO_chart.html");

    pair<QString, QString> paths;

    QString currentPath = QDir::currentPath();
    QDir dir(currentPath);

    if (!dir.exists()) {
        if (!dir.mkpath(".")) {
            currentPath = QDir::tempPath();
            dir = QDir(currentPath);
        }
    }

    if (!QFileInfo(currentPath).isWritable()) {
        currentPath = QDir::tempPath();
        dir = QDir(currentPath);
    }

    QString csvPath = dir.filePath("compare_sort_bigO.csv");
    QString htmlPath = dir.filePath("bigO_chart.html");

    outFile << "n," << sortO_name << "_t,BigO_t" << el;
    htmlFile << "<!DOCTYPE html><html><head><script src=\"https://cdn.jsdelivr.net/npm/chart.js\"></script></head><body>";
    htmlFile << "<canvas id=\"bigOChart\" width=\"800\" height=\"400\"></canvas>";
    htmlFile << "<script>var ctx = document.getElementById('bigOChart').getContext('2d');";
    htmlFile << "var myChart = new Chart(ctx, {type: 'line',data: {labels: [";
    // Adjust loop to handle dynamic size
    int size = min(sortData.size(), bigOData.size());
    for (int i = 0; i < size; i++) {
        outFile << sortData[i].first << "," << sortData[i].second << "," << bigOData[i] << el;
        htmlFile << sortData[i].first;
        if (i != size - 1) htmlFile << ",";
    }

    htmlFile << "],datasets: [{label: '" << sortO_name << "',data: [";

    for (int i = 0; i < size; i++) {
        htmlFile << sortData[i].second;
        if (i != size - 1) htmlFile << ",";
    }

    htmlFile << "],borderColor: 'blue',fill: false},{label: 'Big-O',data: [";

    for (int i = 0; i < size; i++) {
        htmlFile << bigOData[i];
        if (i != size - 1) htmlFile << ",";
    }

    htmlFile << "],borderColor: 'red',fill: false}]},options: {scales: {y: {beginAtZero: true}}}});</script></body></html>";

    outFile.close();
    htmlFile.close();
    cout << "CSV and HTML for " << sortO_name << " vs. Big-O generated." << el;

    paths.first = csvPath;
    paths.second = htmlPath;
    return paths;
}
```

5. Graphical User Interface

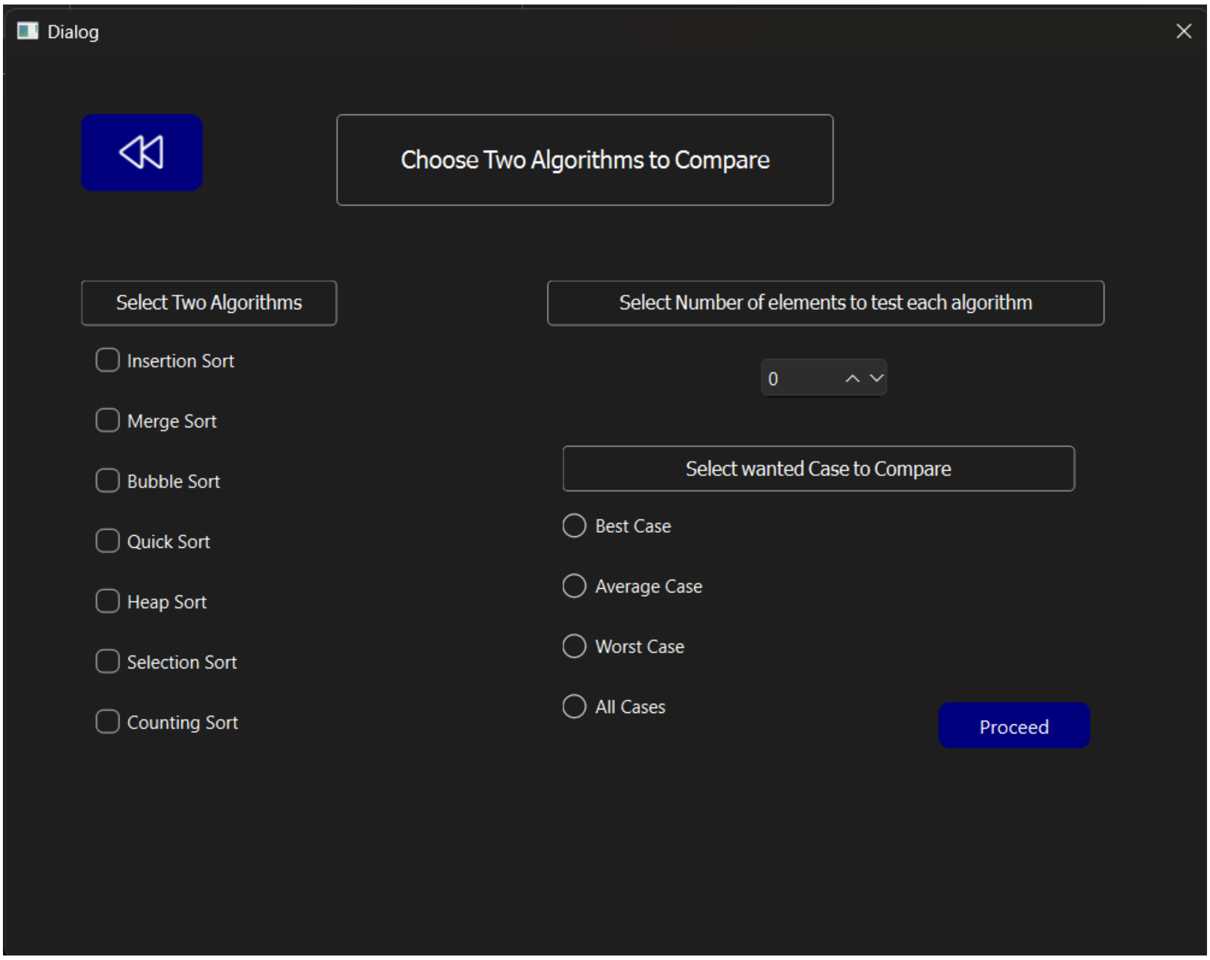
5.1. Home Screen



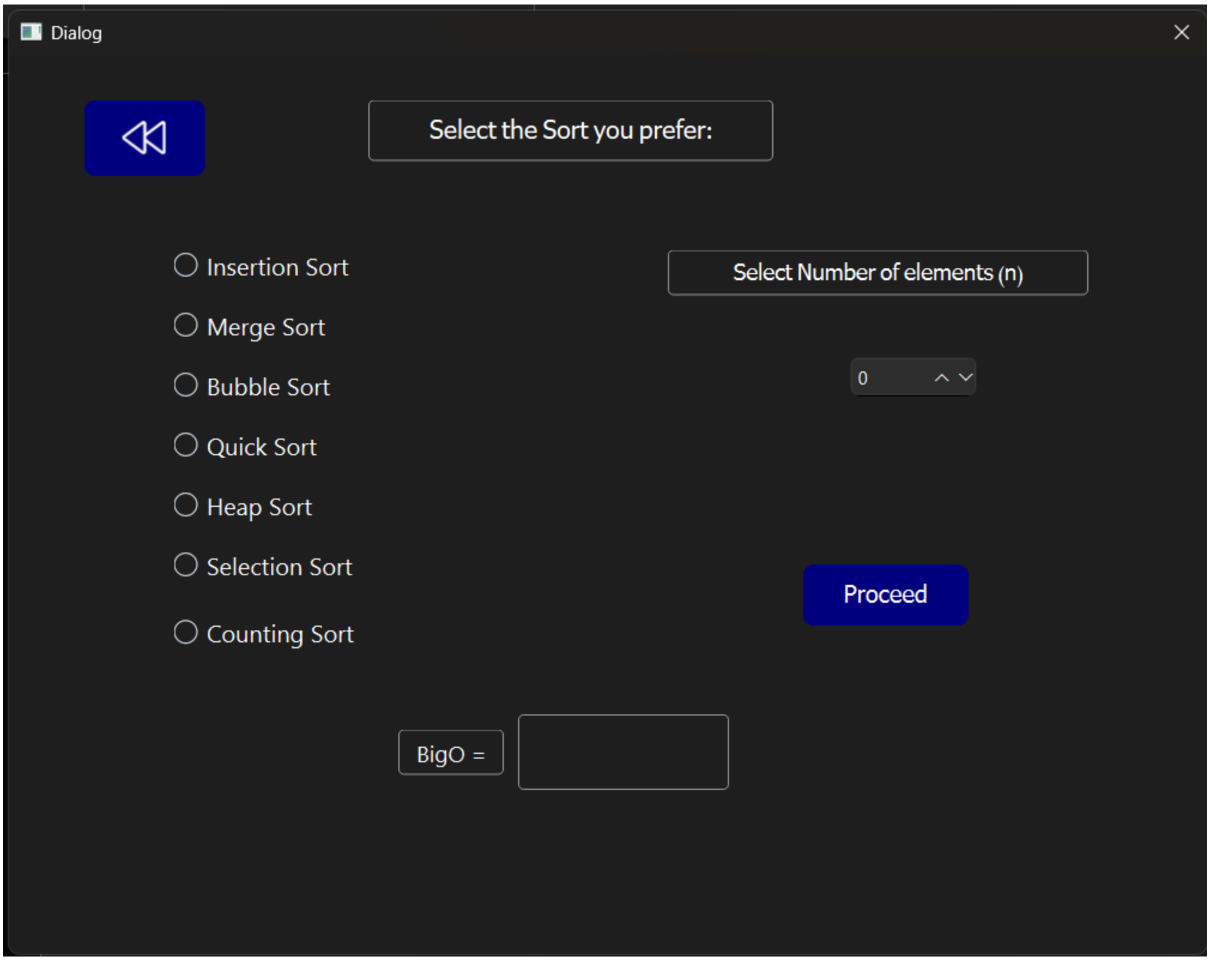
5.2. Choices


Dialog
Choose between either:
compare the efficiency of an algorithm with another algorithm
1
compare an algorithm with its asymptotic efficiency
2

5.3. Case 1

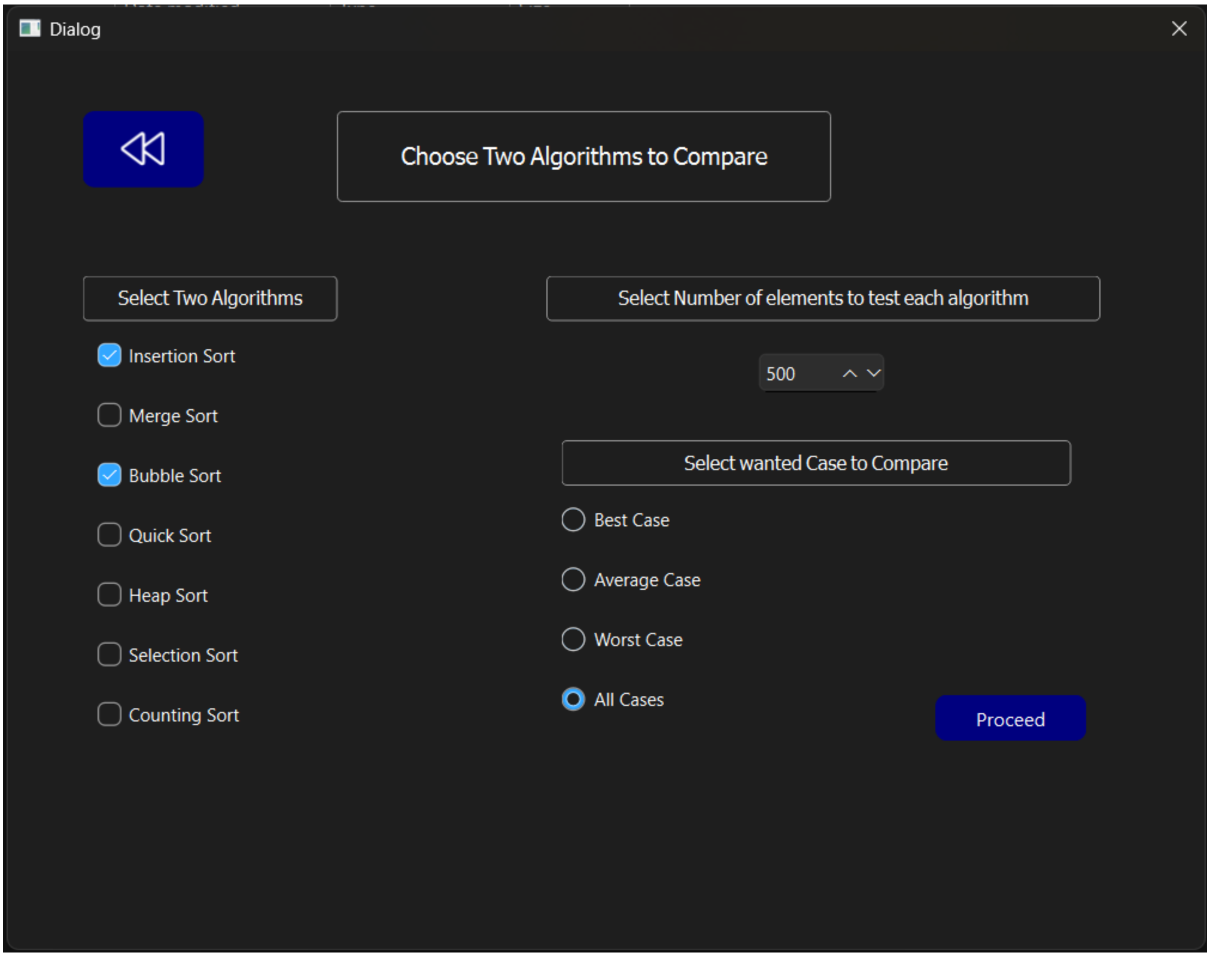

Dialog
Choose Two Algorithms to Compare
Select Two Algorithms
Insertion Sort
Merge Sort
Bubble Sort
Quick Sort
Heap Sort
Selection Sort
Counting Sort
Select Number of elements to test each algorithm
0
Select wanted Case to Compare
Best Case
Average Case
Worst Case
All Cases
Proceed

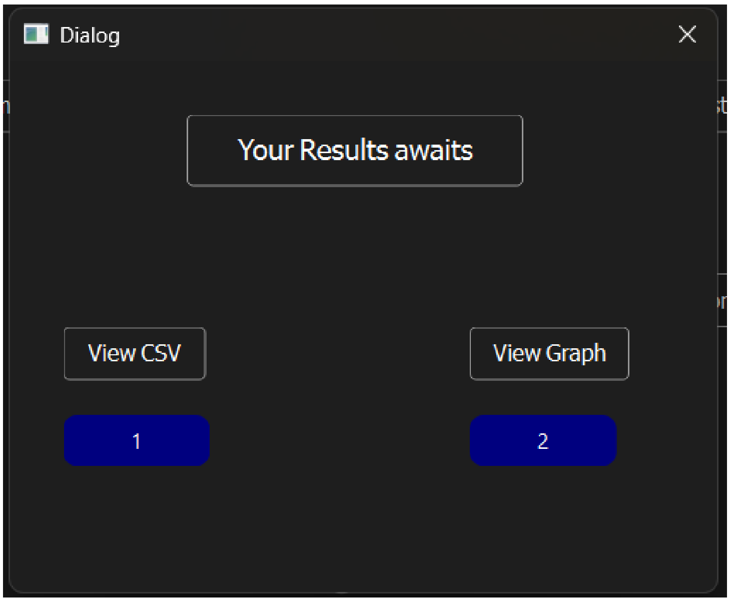
5.4. Case 2


Dialog
Select the Sort you prefer:
Insertion Sort
Merge Sort
Bubble Sort
Quick Sort
Heap Sort
Selection Sort
Counting Sort
Select Number of elements (n)
0
Proceed
BigO =

6. Test

6.1. Case 1



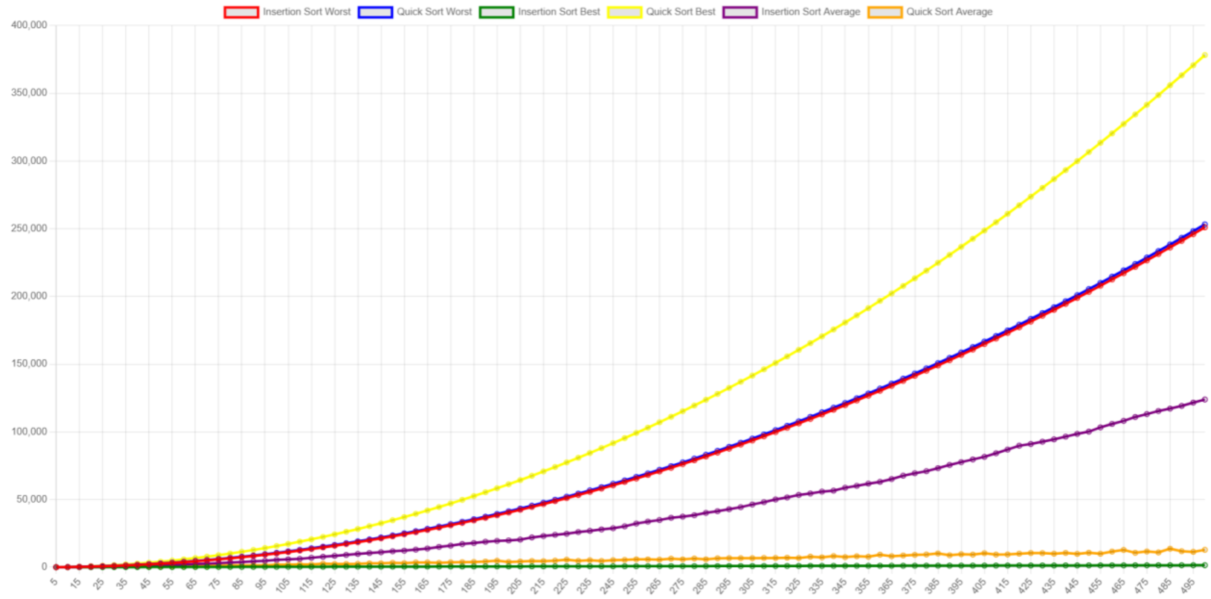


CSV:

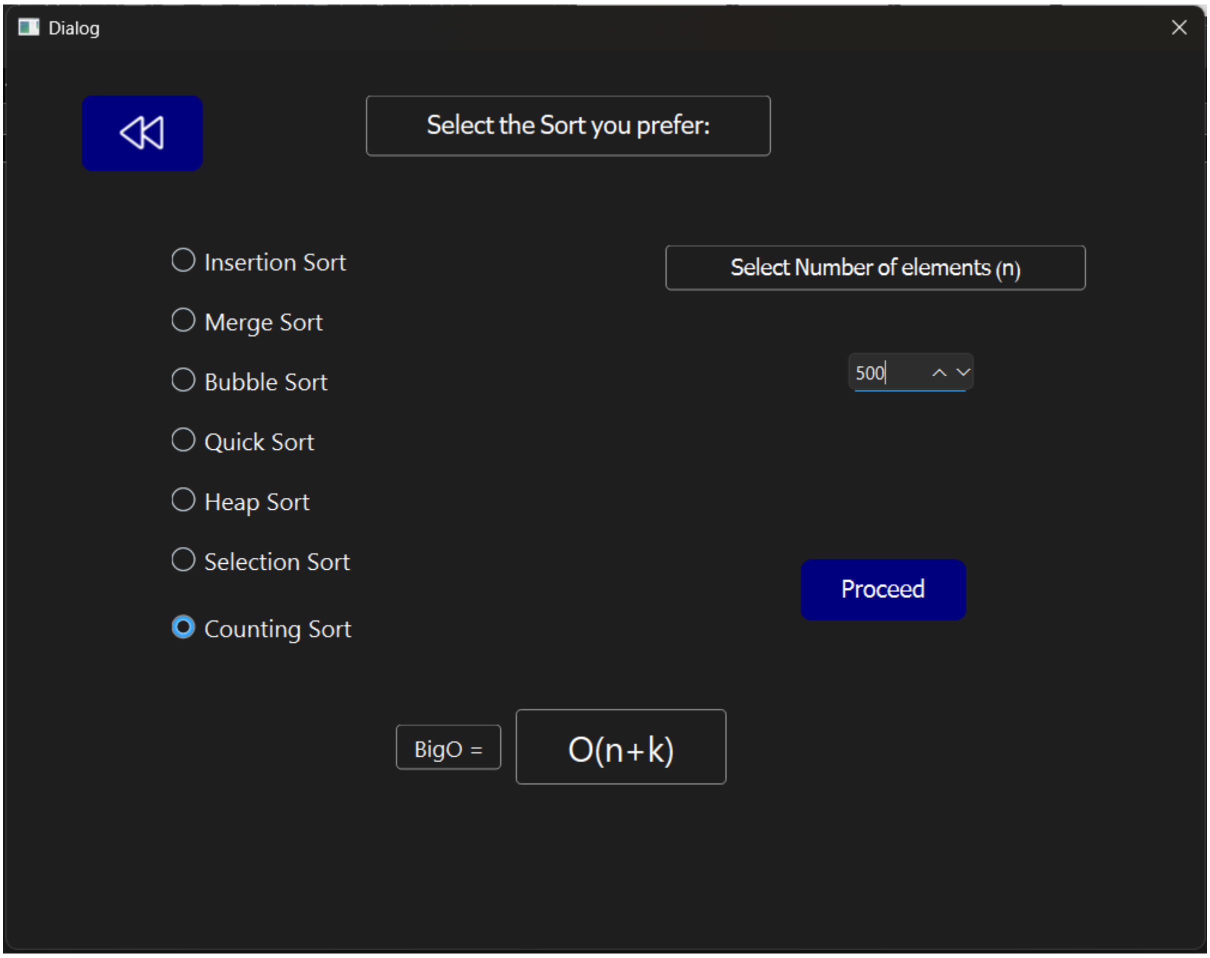
	A	B	C	D	E	F	G
1	n	Worst_Insertion Sortt	WorstBubble Sortt	BestInsertion Sortt	BestBubble Sortt	AverageInsertion Sortt	AverageBubble Sort_t
2	5	32	24	12	14	24	20
3	10	117	99	27	54	79	80
4	15	252	224	42	119	148	172
5	20	437	399	57	209	233	297
6	25	672	624	72	324	326	451
7	30	957	899	87	464	447	644
8	35	1292	1224	102	629	518	837
9	40	1677	1599	117	819	637	1079
10	45	2112	2024	132	1034	942	1439
11	50	2597	2499	147	1274	1241	1821
12	55	3132	3024	162	1539	1510	2213
13	60	3717	3599	177	1829	1947	2714
14	65	4352	4224	192	2144	2212	3154
15	70	5037	4899	207	2484	2577	3669
16	75	5772	5624	222	2849	2928	4202
17	80	6557	6399	237	3239	3339	4790
18	85	7392	7224	252	3654	3762	5409
19	90	8277	8099	267	4094	4123	6022
20	95	9212	9024	282	4559	4628	6732
21	100	10197	9999	297	5049	5059	7430
22	105	11232	11024	312	5564	5686	8251
23	110	12317	12099	327	6104	6237	9059
24	115	13452	13224	342	6669	6900	9948
25	120	14637	14399	357	7259	7627	10894
26	125	15872	15624	372	7874	8200	11788
27	130	17157	16899	387	8514	8663	12652
28	135	18492	18224	402	9179	9246	13601
29	140	19877	19599	417	9869	9947	14634

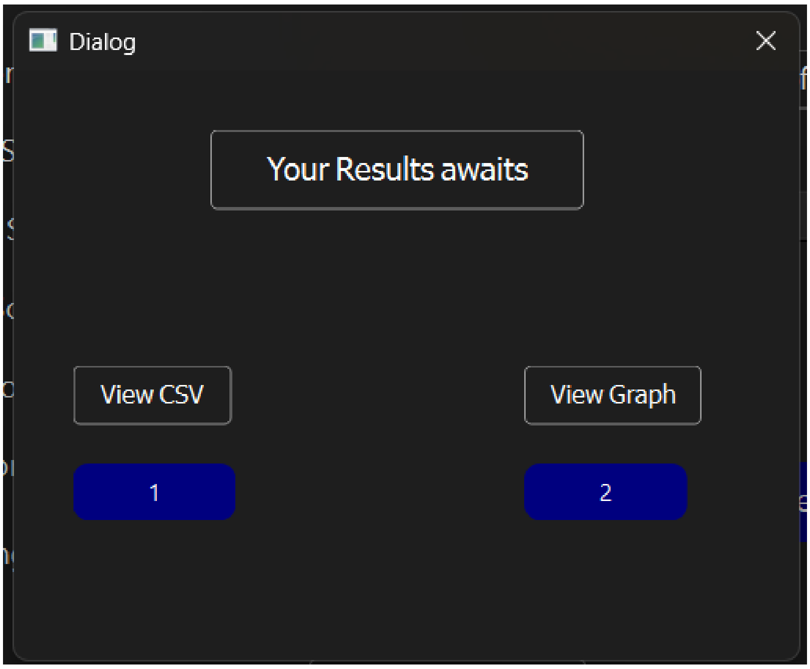
	A	B	C	D	E	F	G	H
76	375	141372	140624	1122	70499	71218	105547	
77	380	145157	144399	1137	72389	73877	108759	
78	385	148992	148224	1152	74304	76046	111751	
79	390	152877	152099	1167	76244	78511	114916	
80	395	156812	156024	1182	78209	80356	117796	
81	400	160797	159999	1197	80199	81565	120383	
82	405	164832	164024	1212	82214	83632	123424	
83	410	168917	168099	1227	84254	85381	126331	
84	415	173052	172224	1242	86319	87550	129473	
85	420	177237	176399	1257	88409	90385	132973	
86	425	181472	180624	1272	90524	91650	135713	
87	430	185757	184899	1287	92664	94087	139064	
88	435	190092	189224	1302	94829	95178	141767	
89	440	194477	193599	1317	97019	97253	144987	
90	445	198912	198024	1332	99234	98952	148044	
91	450	203397	202499	1347	101474	101165	151383	
92	455	207932	207024	1362	103739	102470	154293	
93	460	212517	211599	1377	106029	104425	157553	
94	465	217152	216224	1392	108344	107602	161449	
95	470	221837	220899	1407	110684	109067	164514	
96	475	226572	225624	1422	113049	113140	168908	
97	480	231357	230399	1437	115439	115279	172360	
98	485	236192	235224	1452	117854	118864	176560	
99	490	241077	240099	1467	120294	121087	180104	
100	495	246012	245024	1482	122759	123898	183967	
101	500	250997	249999	1497	125249	124907	186954	
102								
103								
104								

Graph:


Insertion Sort Worst
Quick Sort Worst
Insertion Sort Best
Quick Sort Best
Insertion Sort Average
Quick Sort Average
400,000
350,000
300,000
250,000
200,000
150,000
100,000
50,000
0

6.2. Case 2


Dialog
Select the Sort you prefer:
Insertion Sort
Merge Sort
Bubble Sort
Quick Sort
Heap Sort
Selection Sort
Counting Sort
Select Number of elements (n)
500
Proceed
BigO =
O(n+k)

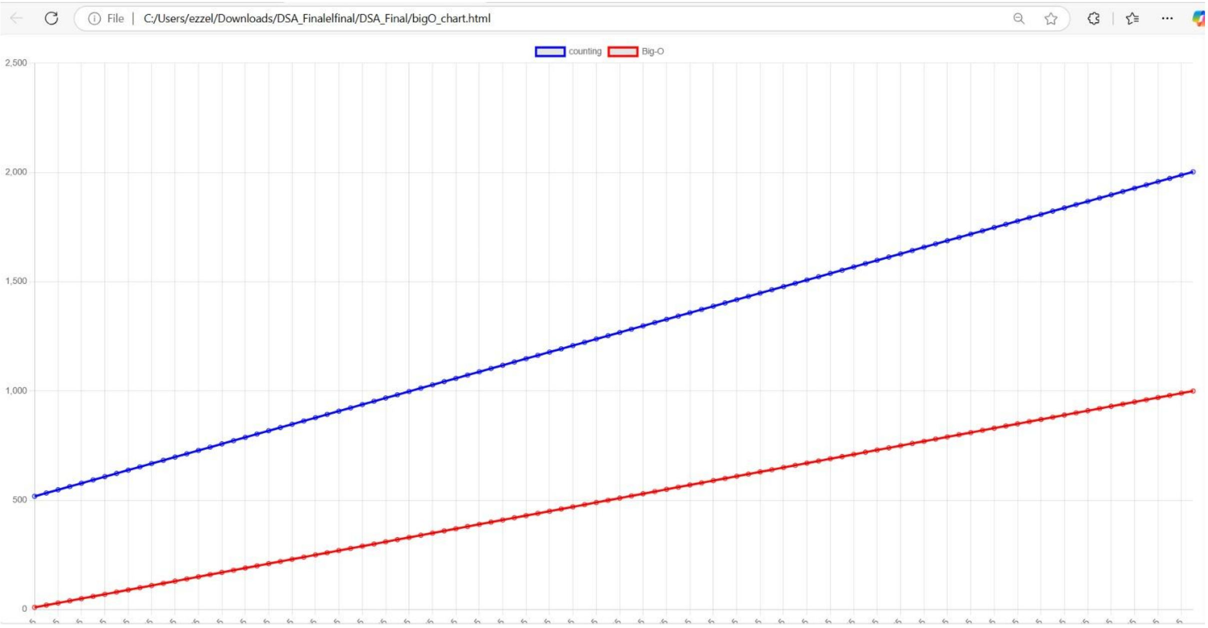

Dialog
Your Results awaits
View CSV
View Graph
1
2

CSV:

1	n	counting_t	BigO_t
2	5	518	10
3	10	533	20
4	15	548	30
5	20	563	40
6	25	578	50
7	30	593	60
8	35	608	70
9	40	623	80
10	45	638	90
11	50	653	100
12	55	668	110
13	60	683	120
14	65	698	130
15	70	713	140
16	75	728	150
17	80	743	160
18	85	758	170
19	90	773	180
20	95	788	190
21	100	803	200
22	105	818	210
23	110	833	220
24	115	848	230
25	120	863	240
26	125	878	250

	A	B	C
82	405	1718	810
83	410	1733	820
84	415	1748	830
85	420	1763	840
86	425	1778	850
87	430	1793	860
88	435	1808	870
89	440	1823	880
90	445	1838	890
91	450	1853	900
92	455	1868	910
93	460	1883	920
94	465	1898	930
95	470	1913	940
96	475	1928	950
97	480	1943	960
98	485	1958	970
99	490	1973	980
100	495	1988	990
101	500	2003	1000
102			
103			
104			
105			
106			

Graph:

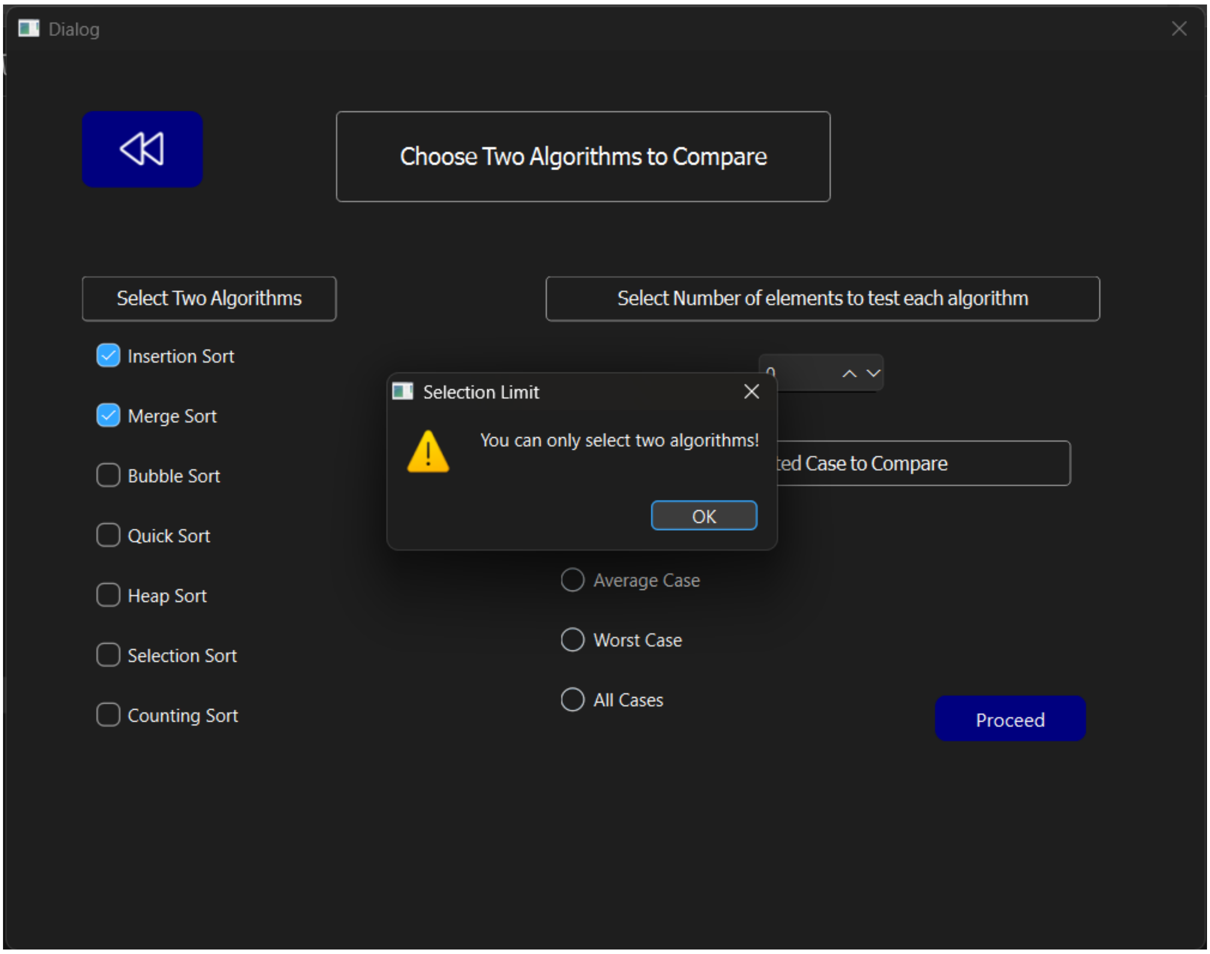

File
C:/Users/ezzel/Downloads/DSA_Finalelfinal/DSA_Final/bigO_chart.html
counting
Big-O
2,500
2,000
1,500
1,000
500
0

6.3. Output files

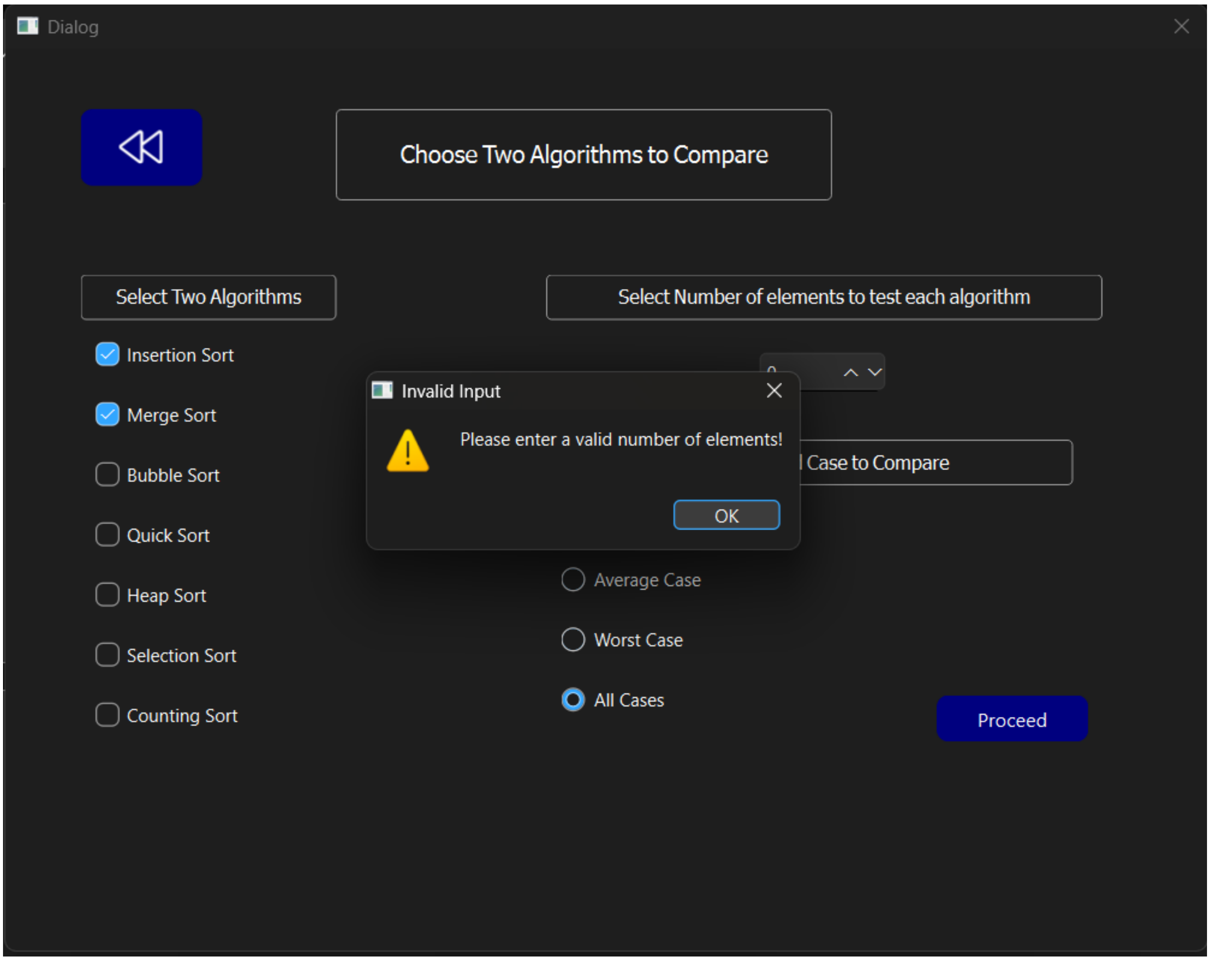
all_cases_chart.html	12/27/2024 1:41 AM	Microsoft Edge HT...	5 KB
bigO_chart.html	12/27/2024 1:39 AM	Microsoft Edge HT...	1 KB
compare_all_cases.csv	12/27/2024 1:39 AM	Microsoft Excel Co...	4 KB
compare_sort_bigO.csv	12/27/2024 1:39 AM	Microsoft Excel Co...	1 KB
DSA_PROJECT.exe	12/27/2024 1:39 AM	Application	340 KB

7. Error test

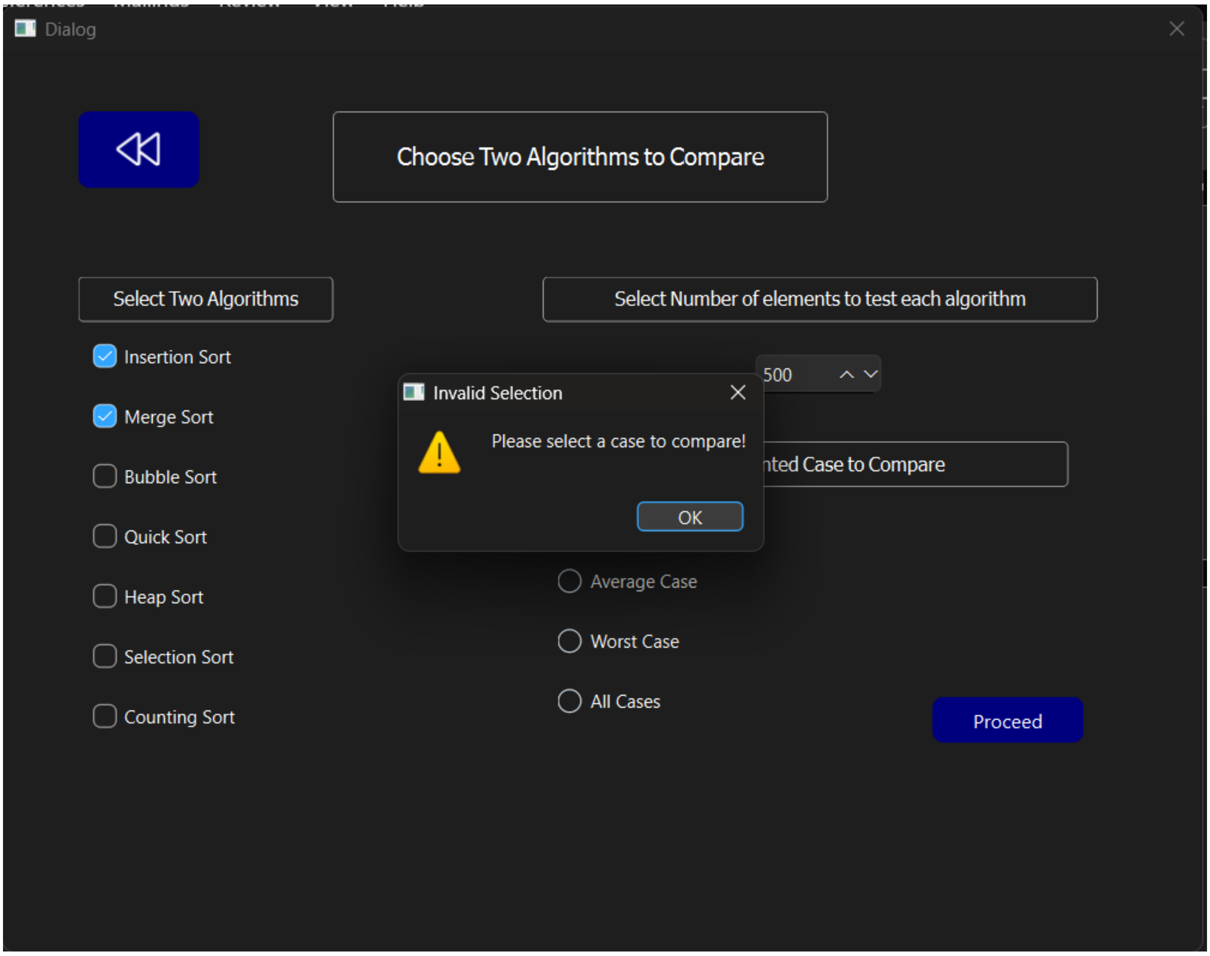
7.1. Selecting more than 2 cases



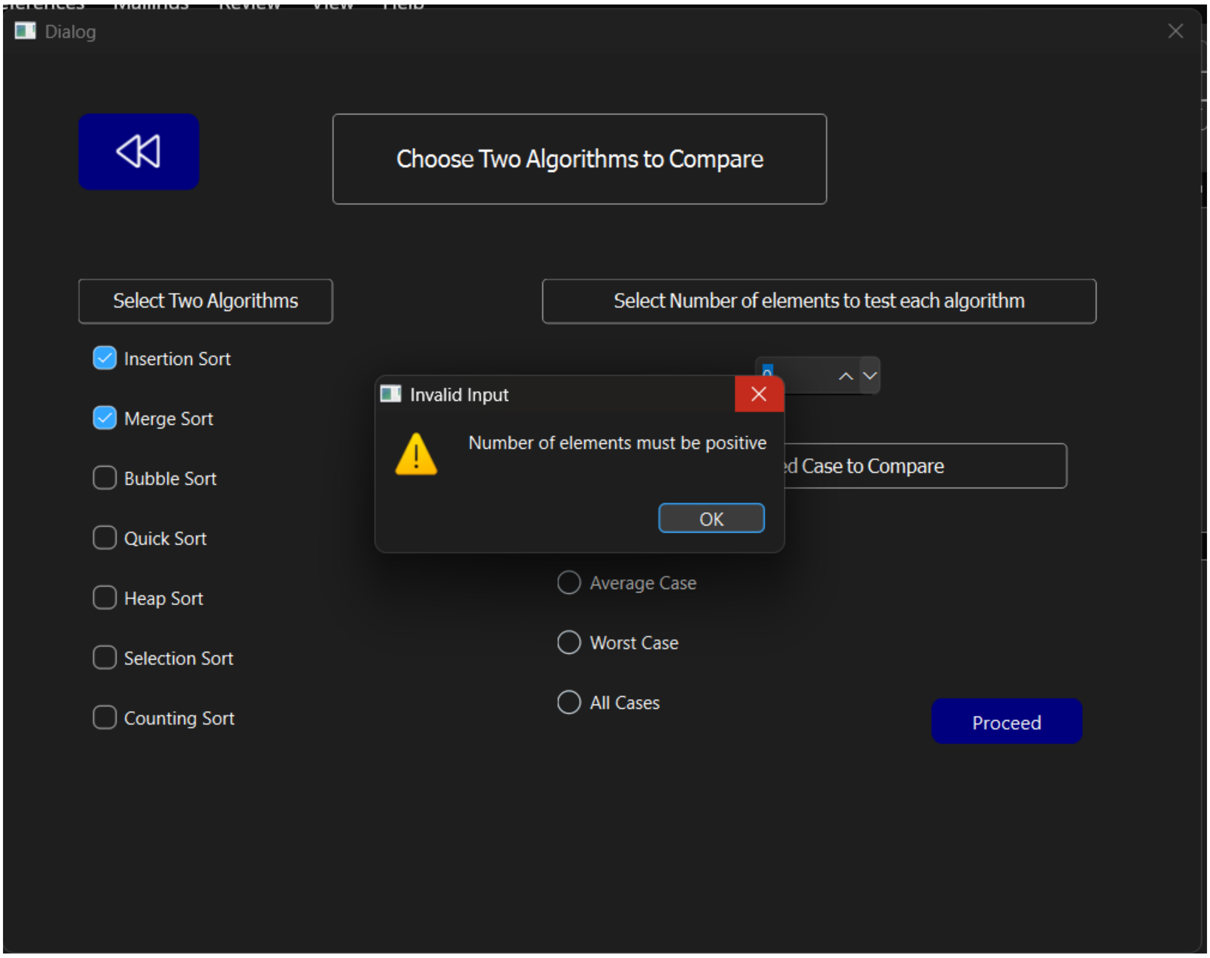
7.2. Entering n with 0



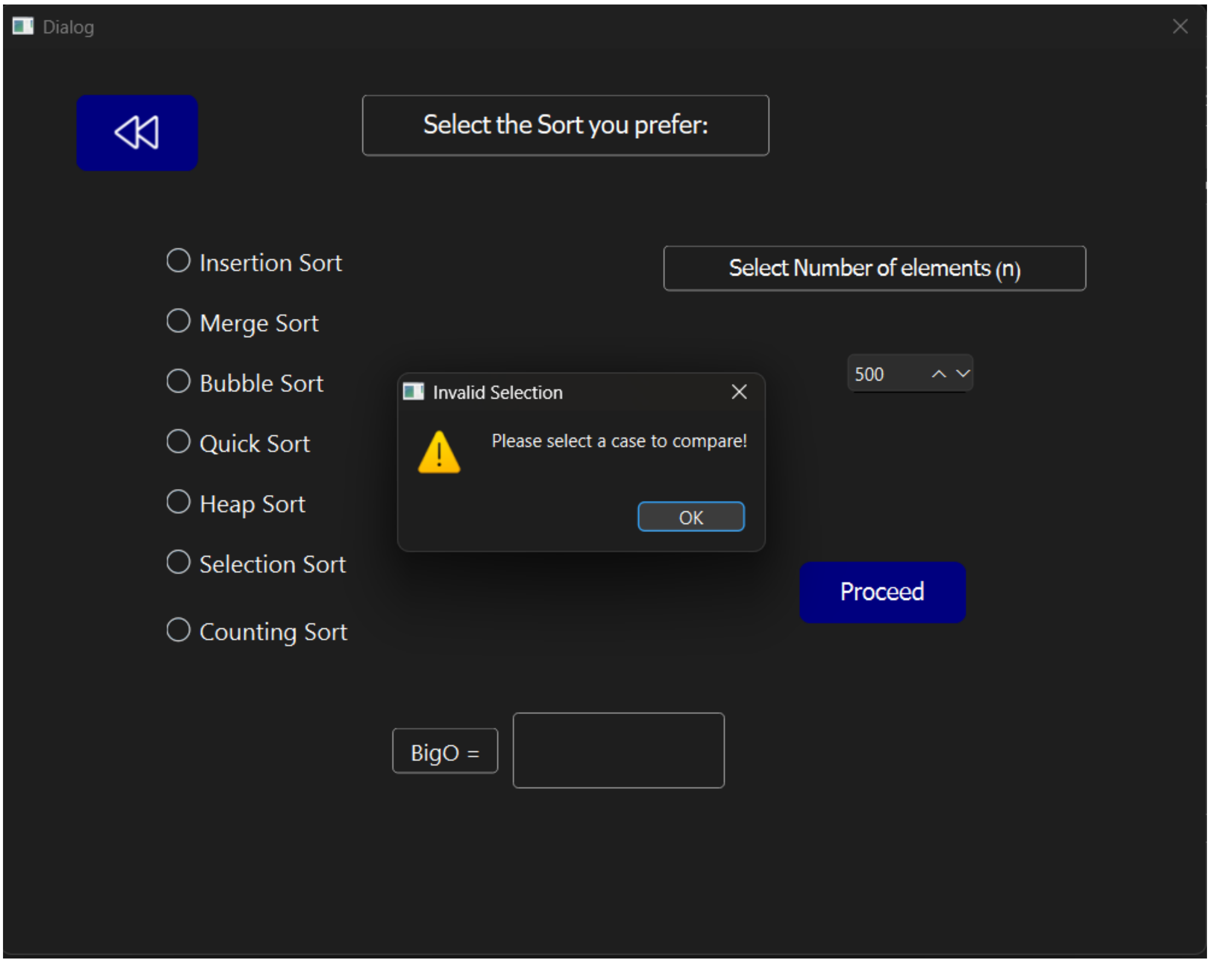
7.3. Didn’t select case



7.4. Entering n with negative number



7.5. Didn’t select algorithm in case 2



8. Contribution

Name	Contribution
Ahmed Abbady Mohamed	Front End & Back End
Ezzeldin Ismail	Back End
Omar Mohamed Mostafa	Front & Back End
Ahmed Wael Raafat	Back End
Anas Mansour	Back End