SORTING ALGORITHMS

By Furkan Yakkan
and Berkay Gökçay

Analysis of Algorithms

Introduction

Sorting algorithms that arrange the items in a list or array according to certain rules, these rules can be alphabetical order, numerical ascending and descending order. These algorithms serve certain purposes, some of these purposes are that to increase the efficiency of search algorithms and to make analysis possible by given people.

The efficiency of algorithms is measured with the big o notation. Some sorting algorithms may perform better than others. For example, counting sort can have smaller "big o notation "than based on comparison algorithm sort.

This article will try to analyze the following six different sorting algorithms: Merge, Quick, Insertion, Shell, Bubble, Selection Run times will be measured in different programming languages in peer environment and with same dataset. Then it will be analyzed and visualized through graphics.

Test Environment

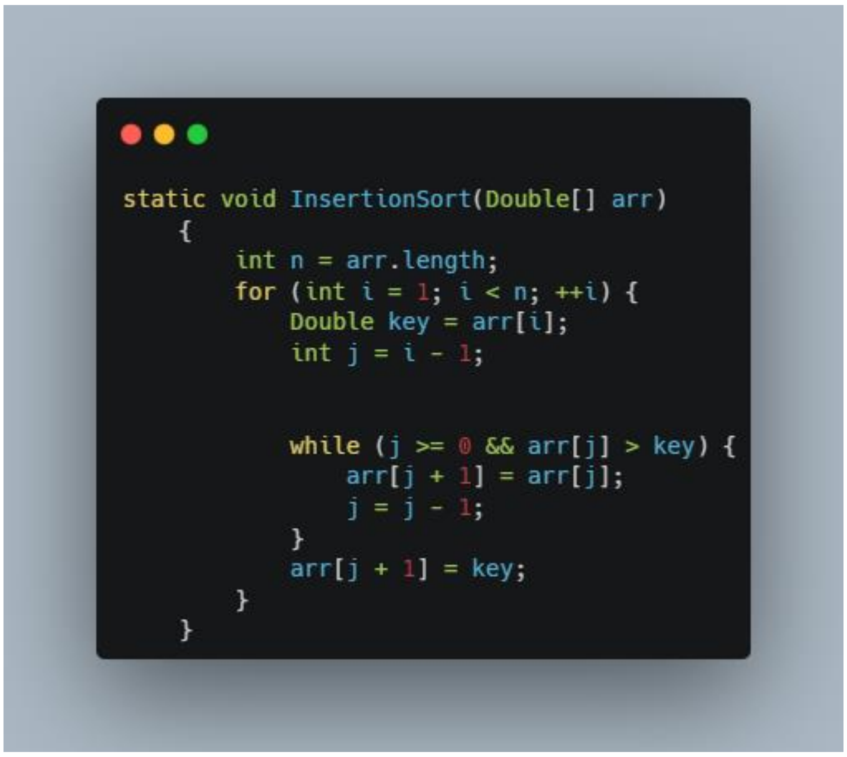
Half of this research was done in Java programming language on intel i7 9750H processor and 8gb ram. IntelliJ IDEA 2021.3.2 IDE was used to run the Java code in the research. Second half of this research was done in C++ programming language on intel i7 10750H processor and 16gb ram. Visual Studio 2019 IDE was used to run the C++ code in the research.

Implementation

Insertion Sort

The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part. Time complexity: Best O(n), Worst O(n^2), Average O(n^2). Space Complexity O (1).

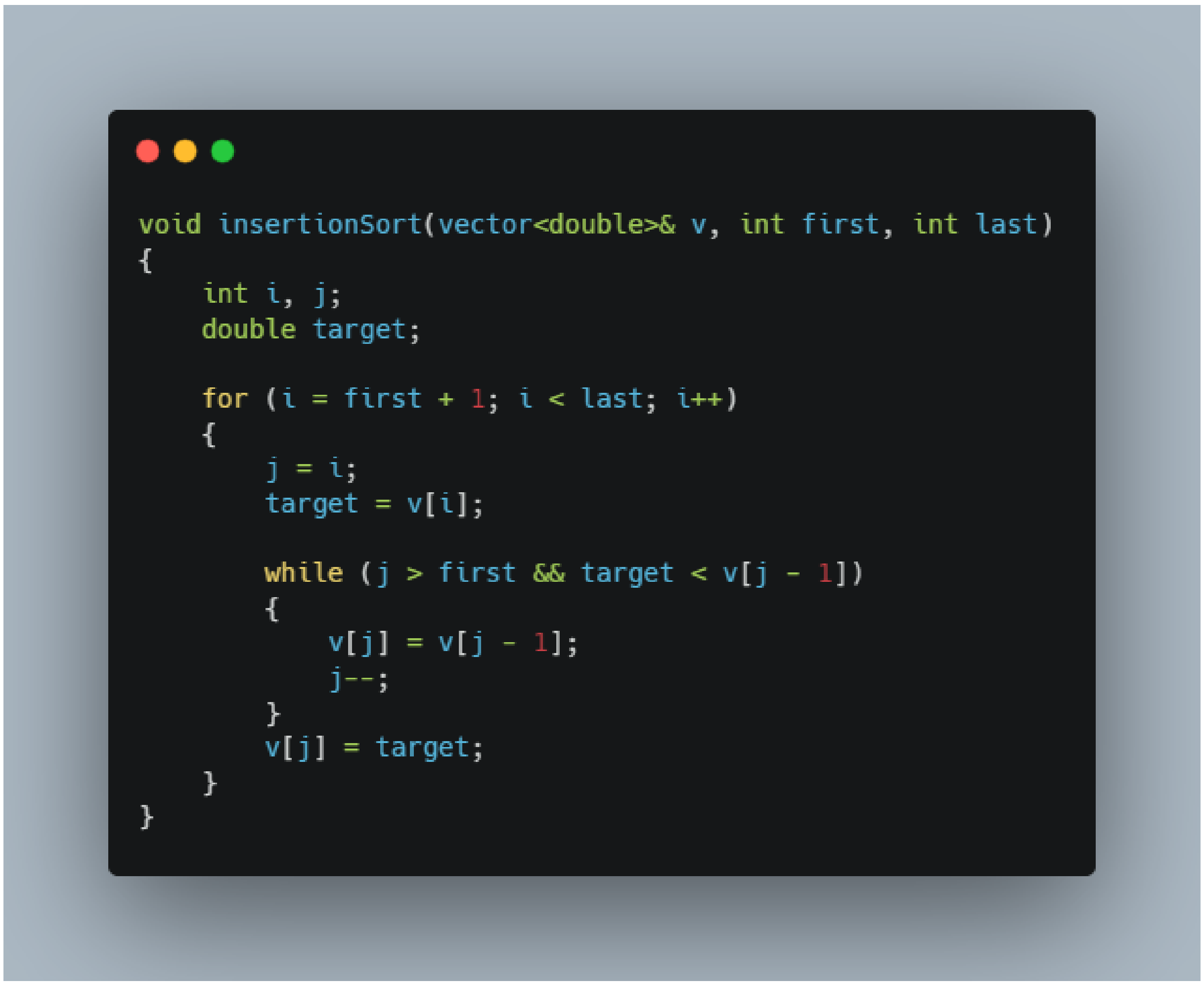
The implementation of the algorithm in **java** is below:



```java
static void InsertionSort(Double[] arr)
    {
        int n = arr.length;
        for (int i = 1; i < n; ++i) {
            Double key = arr[i];
            int j = i - 1;


            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j = j - 1;
            }
            arr[j + 1] = key;
        }
    }
```

Figure 1: Insertion Sort Implemented in Java

The implementation of the algorithm in **C++** is below:



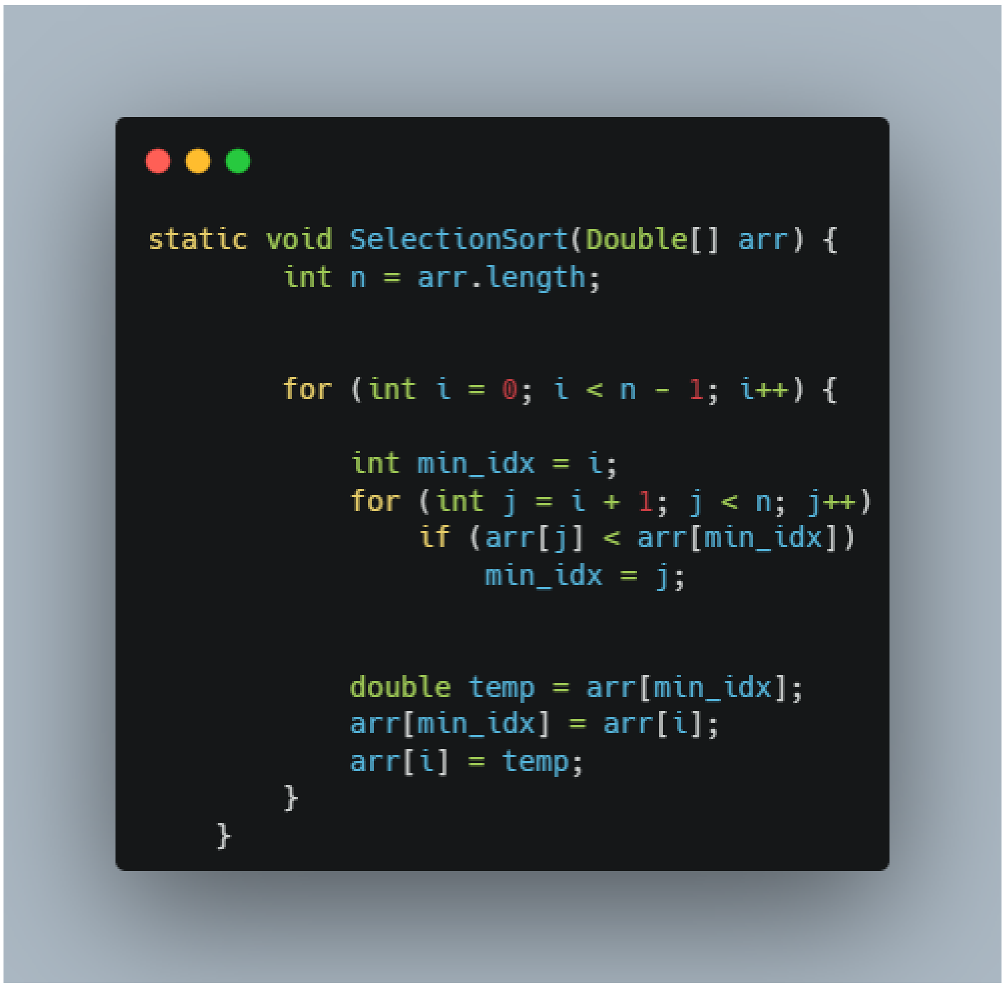
```cpp
void insertionSort(vector<double>& v, int first, int last)
{
    int i, j;
    double target;

    for (i = first + 1; i < last; i++)
    {
        j = i;
        target = v[i];

        while (j > first && target < v[j - 1])
        {
            v[j] = v[j - 1];
            j--;
        }
        v[j] = target;
    }
}
```

Figure 2: Insertion Sort Implemented in C++

Selection Sort

The algorithm divides the input list into two parts: a sorted sub list of items which is built up from left to right at the front (left) of the list and a sub list of the remaining unsorted items that occupy the rest of the list. Initially, the sorted sub list is empty and the unsorted sub list is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sub list, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sub list boundaries one element to the right. Time complexity: Best O(n^2), Worst O(n^2), Average O(n^2). Space Complexity O (1).

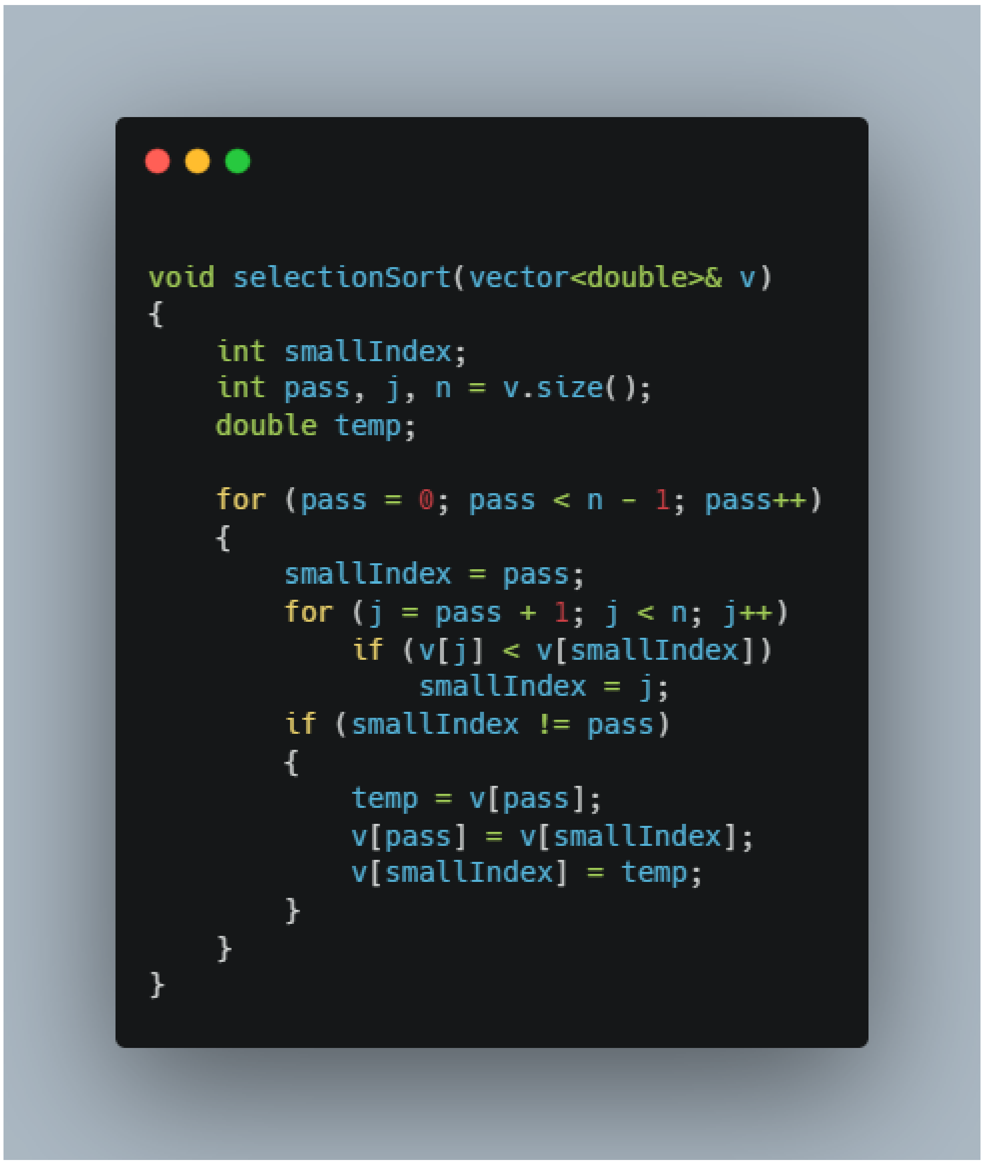
The implementation of the algorithm in **java** is below:



```java
static void SelectionSort(Double[] arr) {
        int n = arr.length;


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

            int min_idx = i;
            for (int j = i + 1; j < n; j++)
                if (arr[j] < arr[min_idx])
                    min_idx = j;


            double temp = arr[min_idx];
            arr[min_idx] = arr[i];
            arr[i] = temp;
        }
    }
```

Figure 3: Selection Implemented in Java

The implementation of the algorithm in **C++** is below:



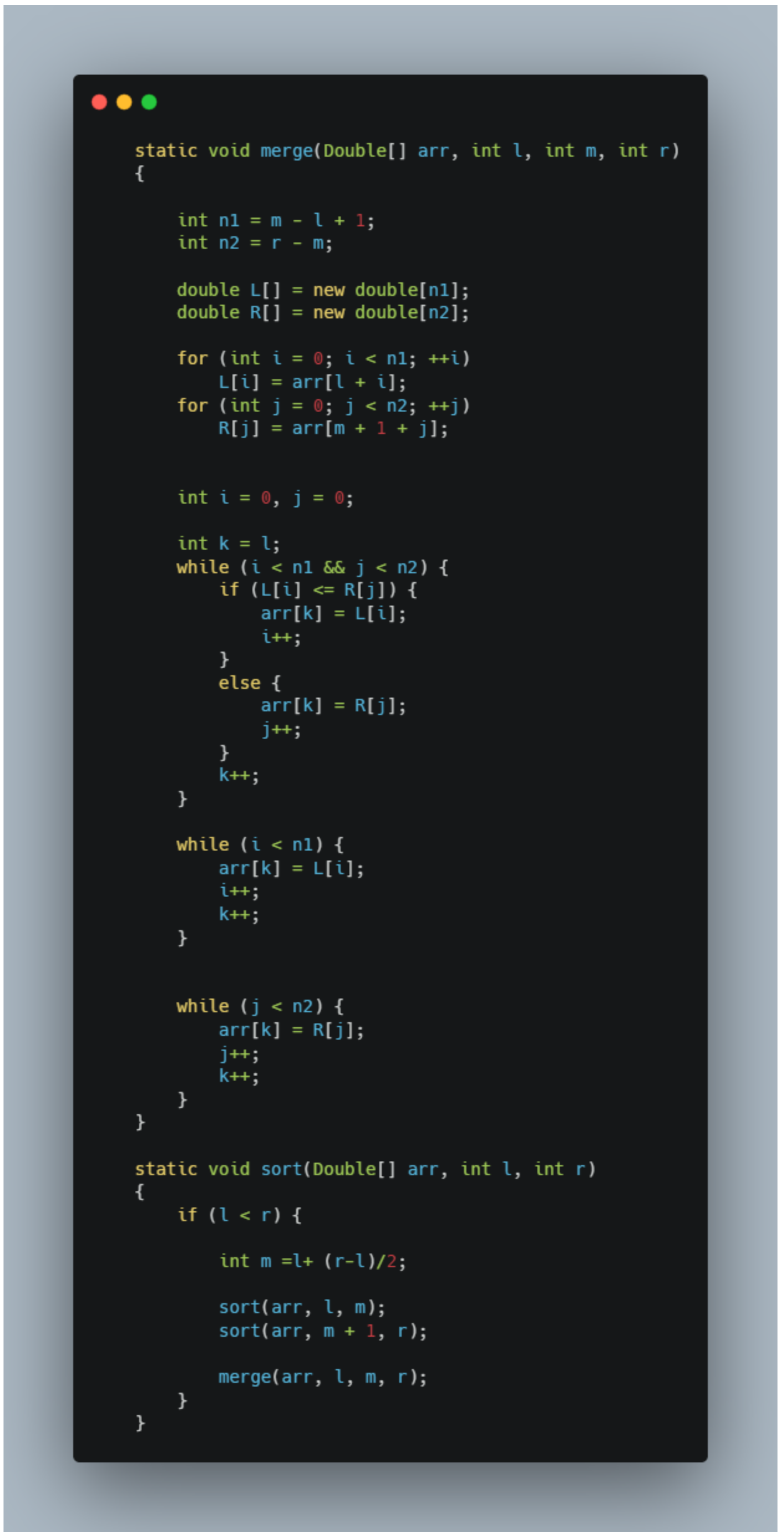
```
void selectionSort(vector<double>& v)
{
    int smallIndex;
    int pass, j, n = v.size();
    double temp;

    for (pass = 0; pass < n - 1; pass++)
    {
        smallIndex = pass;
        for (j = pass + 1; j < n; j++)
            if (v[j] < v[smallIndex])
                smallIndex = j;
        if (smallIndex != pass)
        {
            temp = v[pass];
            v[pass] = v[smallIndex];
            v[smallIndex] = temp;
        }
    }
}
```

Figure 4: Selection Sort Implemented in C++

Merge Sort

Merge Sort works on the principle of Divide and Conquer. Merge sort repeatedly breaks down a list into several sub lists until each sub list consists of a single element and merging those sub lists in a manner that results into a sorted list. Time complexity: Best O (n*log n), Worst O (n*log n), Average O (n*log n). Space Complexity O(n)

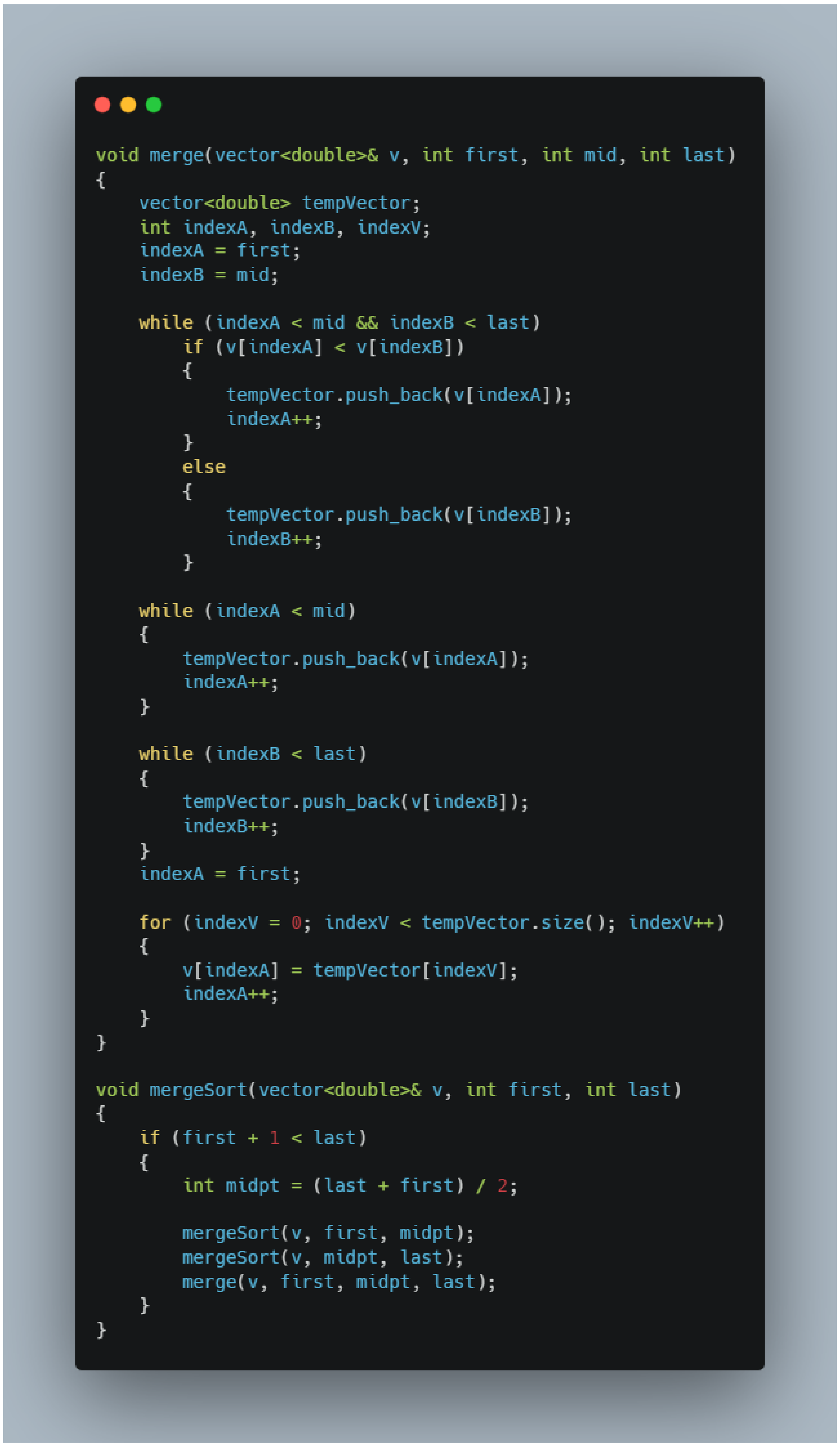
The implementation of the algorithm in **java** is below:



```java
static void merge(Double[] arr, int l, int m, int r)
{

    int n1 = m - l + 1;
    int n2 = r - m;

    double L[] = new double[n1];
    double R[] = new double[n2];

    for (int i = 0; i < n1; ++i)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; ++j)
        R[j] = arr[m + 1 + j];


    int i = 0, j = 0;

    int k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }


    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

static void sort(Double[] arr, int l, int r)
{
    if (l < r) {

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

        sort(arr, l, m);
        sort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
```

Figure 5: Merge Implemented in Java

The implementation of the algorithm in **C++** is below:



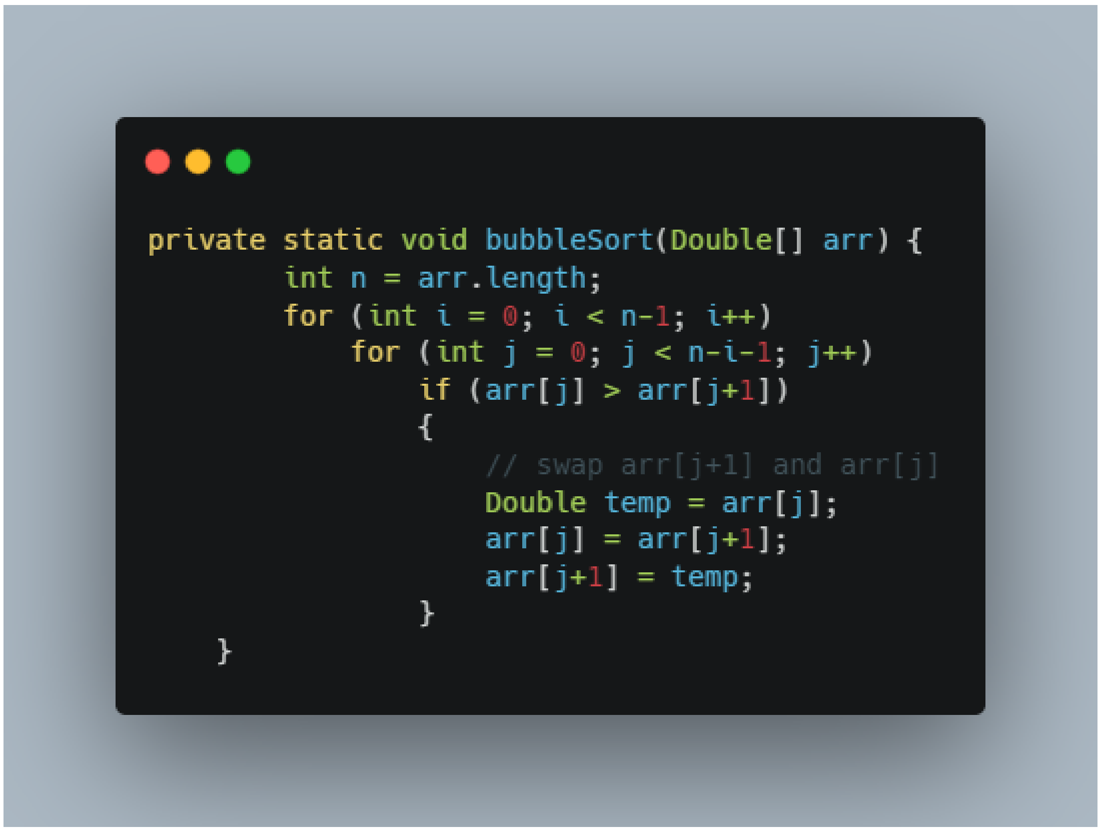
```cpp
void merge(vector<double>& v, int first, int mid, int last)
{
    vector<double> tempVector;
    int indexA, indexB, indexV;
    indexA = first;
    indexB = mid;

    while (indexA < mid && indexB < last)
        if (v[indexA] < v[indexB])
        {
            tempVector.push_back(v[indexA]);
            indexA++;
        }
        else
        {
            tempVector.push_back(v[indexB]);
            indexB++;
        }

    while (indexA < mid)
    {
        tempVector.push_back(v[indexA]);
        indexA++;
    }

    while (indexB < last)
    {
        tempVector.push_back(v[indexB]);
        indexB++;
    }
    indexA = first;

    for (indexV = 0; indexV < tempVector.size(); indexV++)
    {
        v[indexA] = tempVector[indexV];
        indexA++;
    }
}

void mergeSort(vector<double>& v, int first, int last)
{
    if (first + 1 < last)
    {
        int midpt = (last + first) / 2;

        mergeSort(v, first, midpt);
        mergeSort(v, midpt, last);
        merge(v, first, midpt, last);
    }
}
```

Figure 6: Merge Implemented in C++

Bubble Sort

Bubble sort, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list. Time complexity: Best O(n), Worst O(n^2), Average O(n^2). Space Complexity O (1).

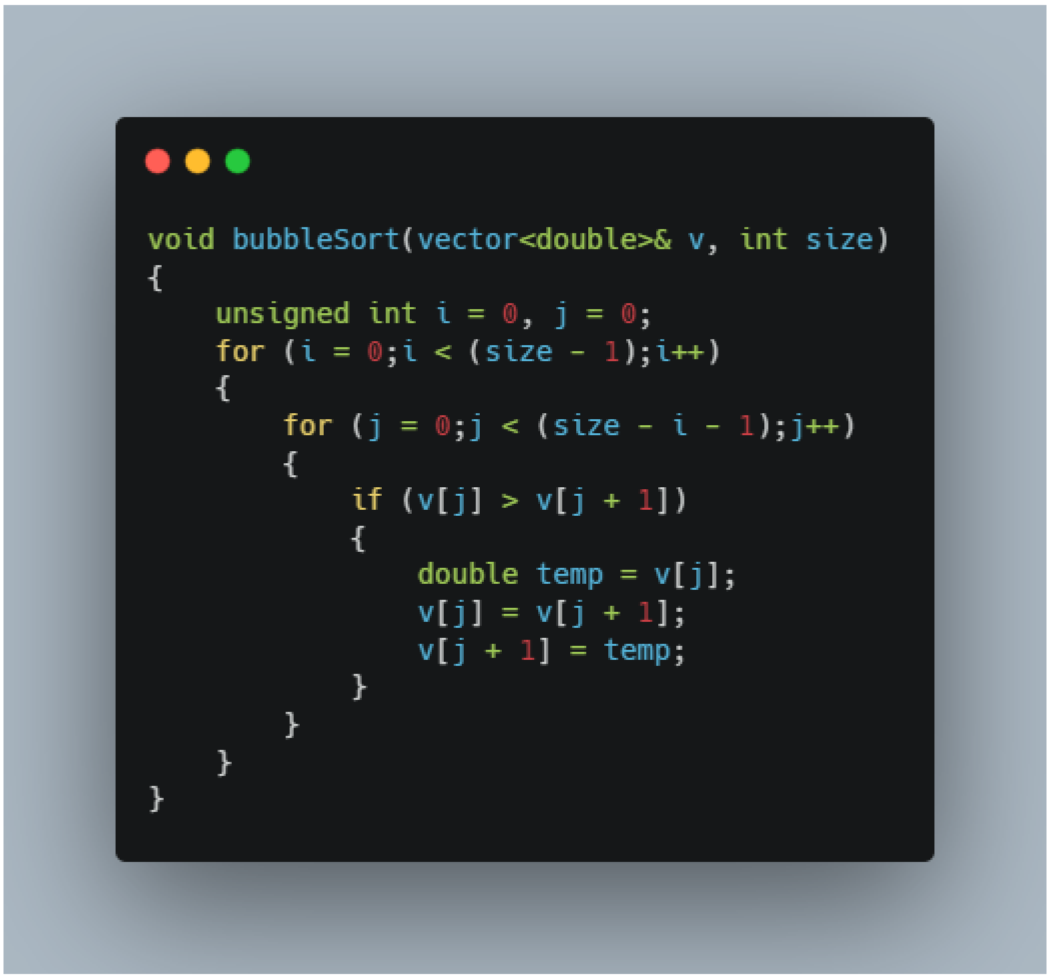
The implementation of the algorithm in **java** is below:



```java
private static void bubbleSort(Double[] arr) {
        int n = arr.length;
        for (int i = 0; i < n-1; i++)
            for (int j = 0; j < n-i-1; j++)
                if (arr[j] > arr[j+1])
                {
                    // swap arr[j+1] and arr[j]
                    Double temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
    }
```

Figure 7: Bubble Sort Implemented in Java

The implementation of the algorithm in **C++** is below:



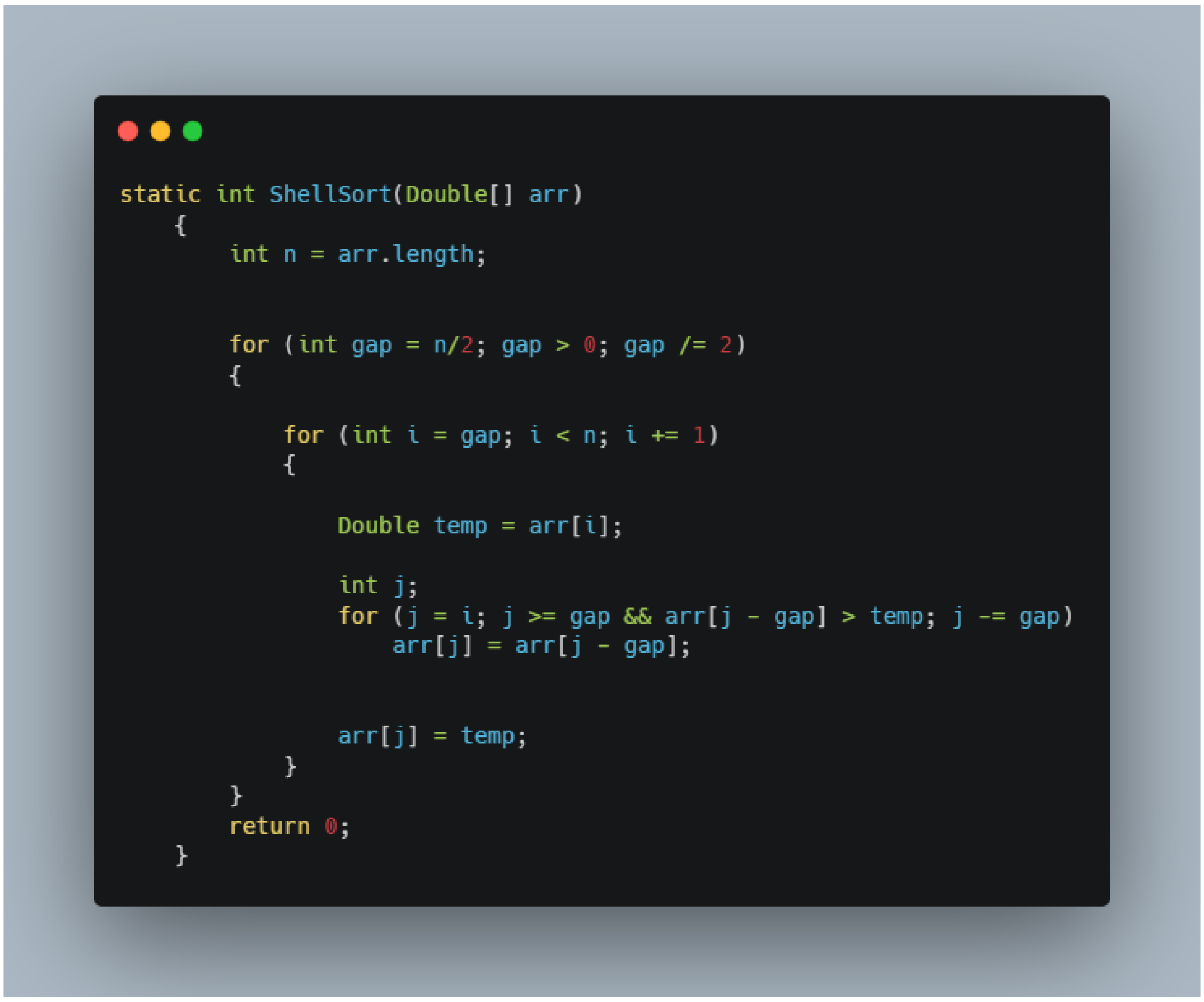
```cpp
void bubbleSort(vector<double>& v, int size)
{
    unsigned int i = 0, j = 0;
    for (i = 0;i < (size - 1);i++)
    {
        for (j = 0;j < (size - i - 1);j++)
        {
            if (v[j] > v[j + 1])
            {
                double temp = v[j];
                v[j] = v[j + 1];
                v[j + 1] = temp;
            }
        }
    }
}
```

Figure 8: Bubble Sort Implemented in C++

Shell Sort

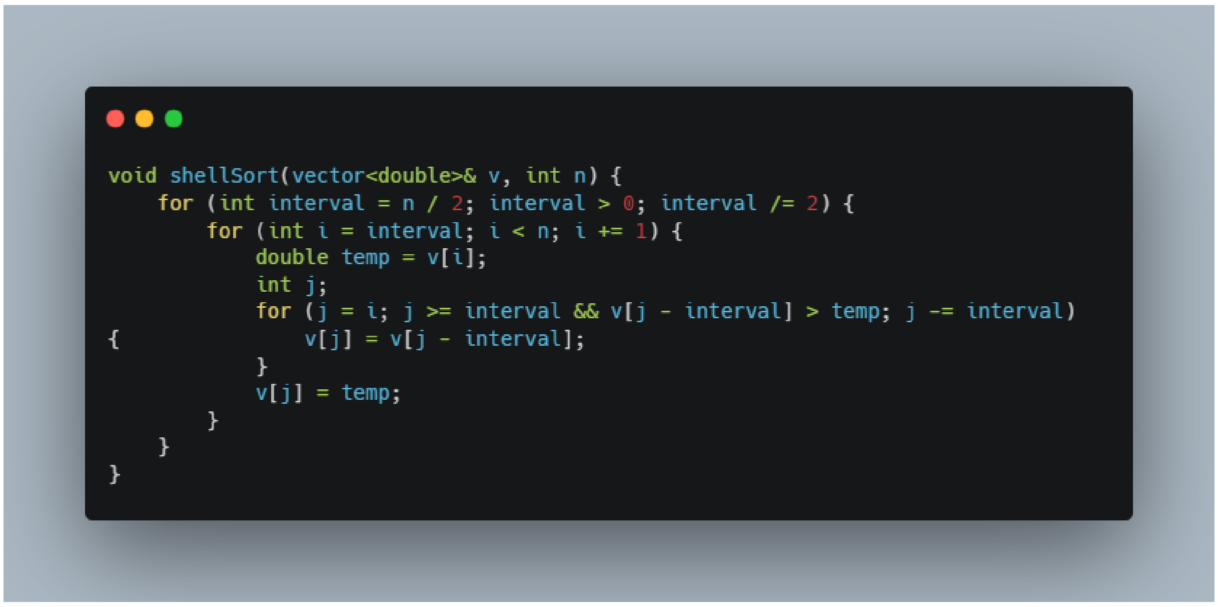
Shell sort starts by sorting pairs of elements far apart from each other, then progressively reducing the gap between elements to be compared. By starting with far apart elements, it can move some out-of-place elements into position faster than a simple nearest neighbour exchange. Time complexity: Best O (nlog n), Worst O(n^2), Average O (nlog n). Space Complexity O (1).

The implementation of the algorithm in **Java** is below:



```java
static int ShellSort(Double[] arr)
    {
        int n = arr.length;


        for (int gap = n/2; gap > 0; gap /= 2)
        {

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

                Double temp = arr[i];

                int j;
                for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
                    arr[j] = arr[j - gap];


                arr[j] = temp;
            }
        }
        return 0;
    }
```

Figure 9: Shell Sort Implemented in Java



```cpp
void shellSort(vector<double>& v, int n) {
    for (int interval = n / 2; interval > 0; interval /= 2) {
        for (int i = interval; i < n; i += 1) {
            double temp = v[i];
            int j;
            for (j = i; j >= interval && v[j - interval] > temp; j -= interval)
{               v[j] = v[j - interval];
            }
            v[j] = temp;
        }
    }
}
```

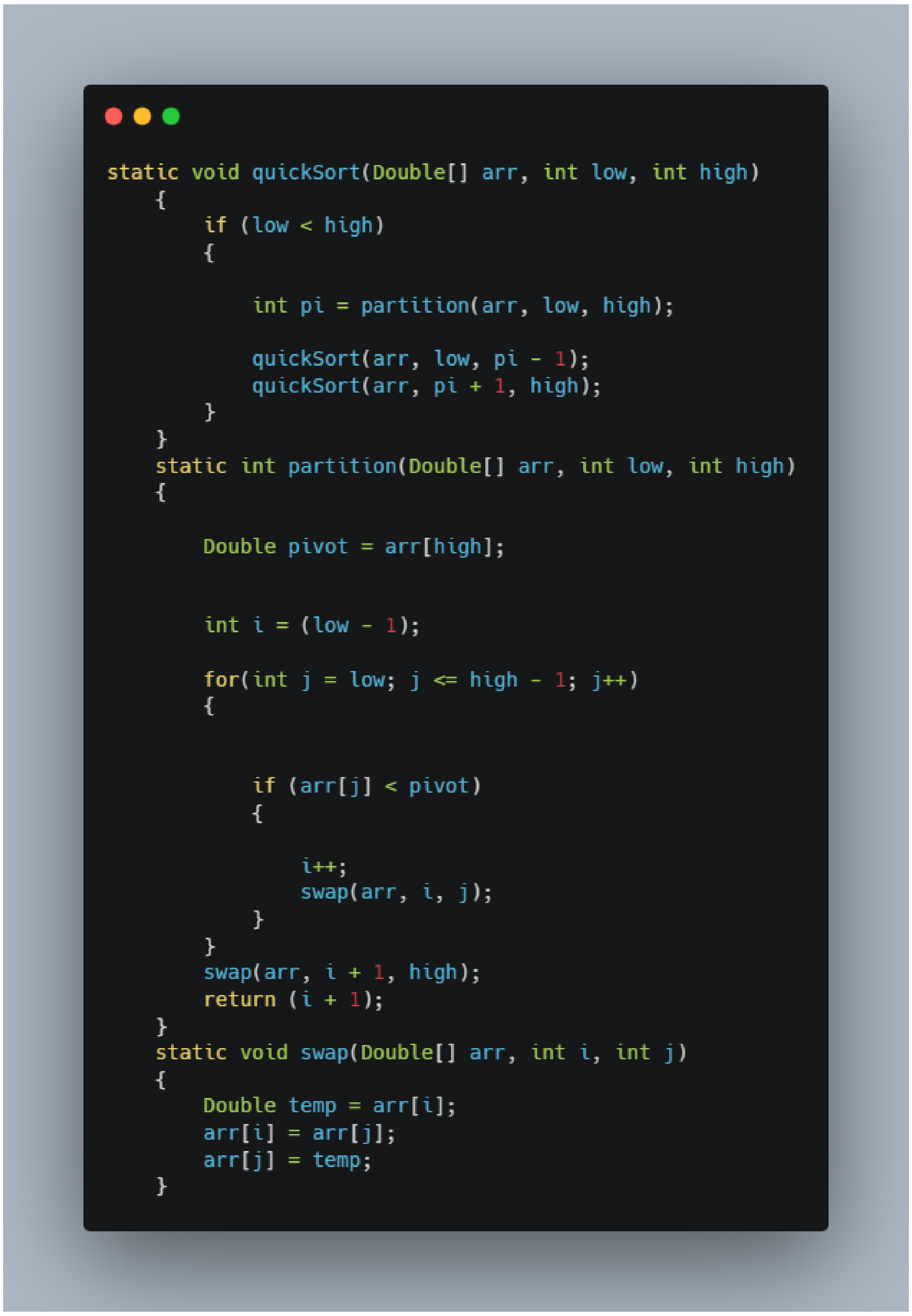
The implementation of the algorithm in **C++** is below:

Figure 10: Shell Sort Implemented in C++

Quick Sort

Quicksort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. For this reason, it is sometimes called partition-exchange sort. The sub-arrays are then sorted recursively. This can be done in-place, requiring small additional amounts of memory to perform the sorting. Time complexity: Best O (n*log n), Worst O(n^2), Average O(n*log n). Space Complexity O (log n)

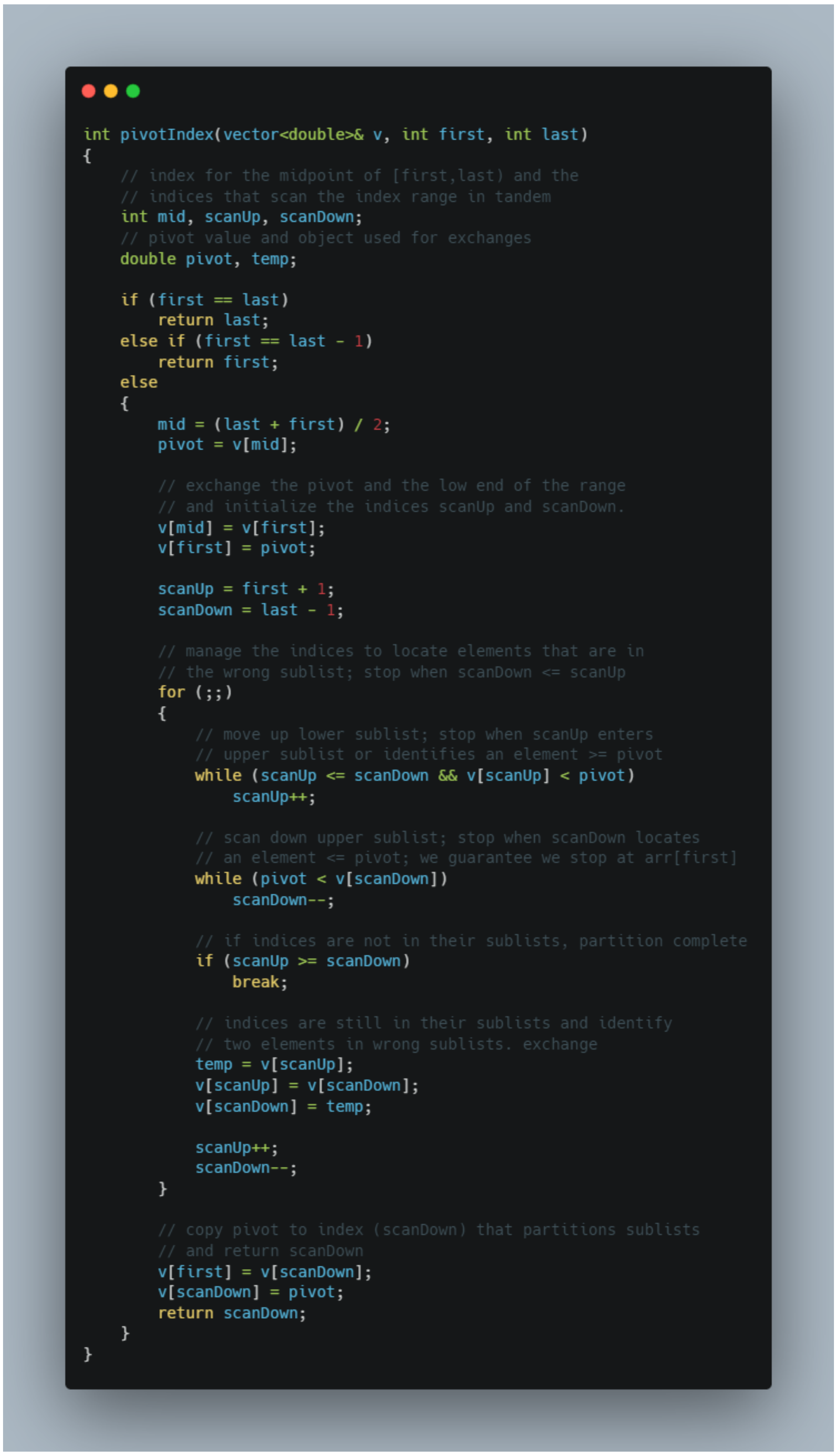
The implementation of the algorithm in **Java** is below:



```java
static void quickSort(Double[] arr, int low, int high)
    {
        if (low < high)
        {

            int pi = partition(arr, low, high);

            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }
    static int partition(Double[] arr, int low, int high)
    {

        Double pivot = arr[high];


        int i = (low - 1);

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


            if (arr[j] < pivot)
            {

                i++;
                swap(arr, i, j);
            }
        }
        swap(arr, i + 1, high);
        return (i + 1);
    }
    static void swap(Double[] arr, int i, int j)
    {
        Double temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
```

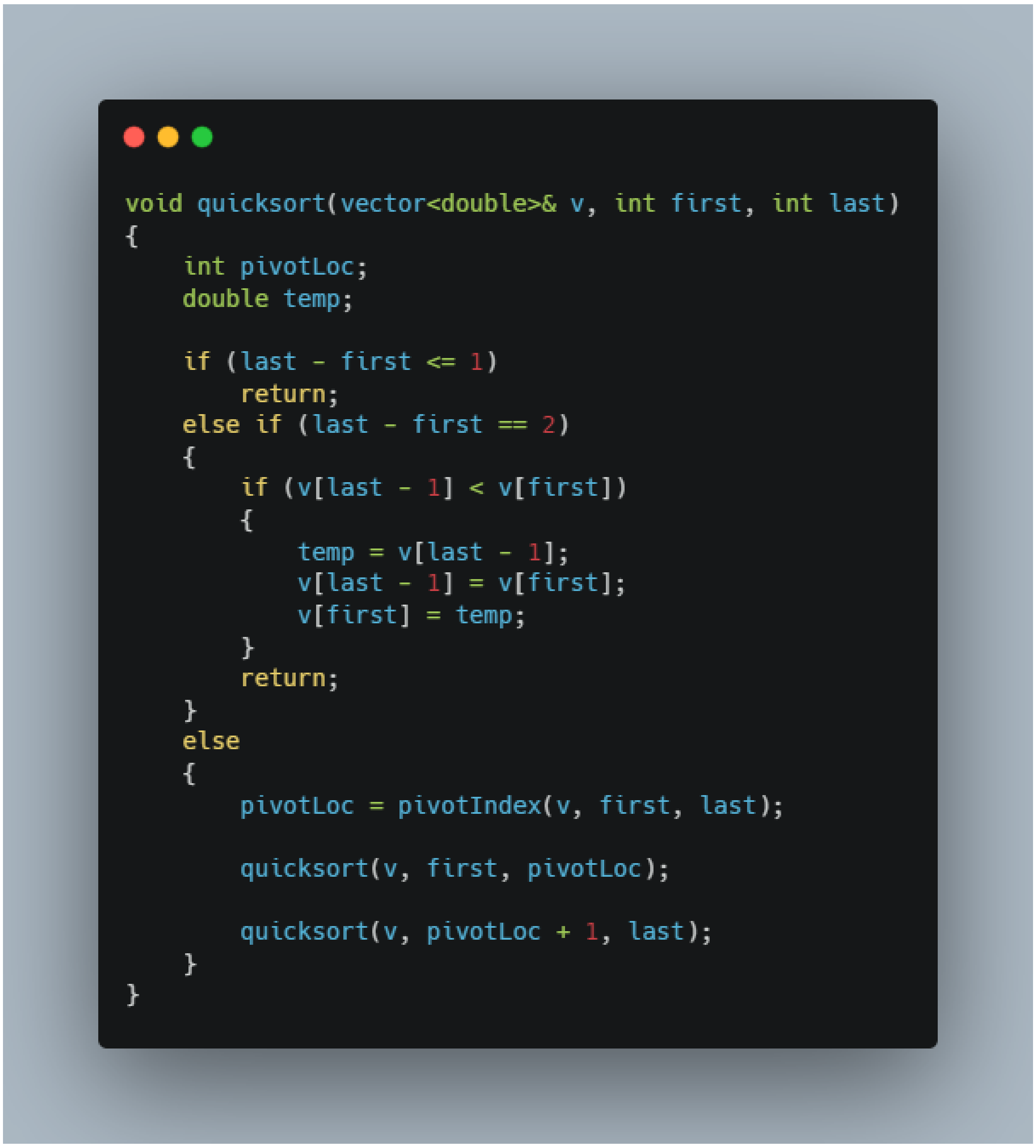
Figure 11: Quick Sort Implemented in Java

The implementation of the algorithm in **C++** is below:



```cpp
int pivotIndex(vector<double>& v, int first, int last)
{
    // index for the midpoint of [first,last) and the
    // indices that scan the index range in tandem
    int mid, scanUp, scanDown;
    // pivot value and object used for exchanges
    double pivot, temp;

    if (first == last)
        return last;
    else if (first == last - 1)
        return first;
    else
    {
        mid = (last + first) / 2;
        pivot = v[mid];

        // exchange the pivot and the low end of the range
        // and initialize the indices scanUp and scanDown.
        v[mid] = v[first];
        v[first] = pivot;

        scanUp = first + 1;
        scanDown = last - 1;

        // manage the indices to locate elements that are in
        // the wrong sublist; stop when scanDown <= scanUp
        for (;;)
        {
            // move up lower sublist; stop when scanUp enters
            // upper sublist or identifies an element >= pivot
            while (scanUp <= scanDown && v[scanUp] < pivot)
                scanUp++;

            // scan down upper sublist; stop when scanDown locates
            // an element <= pivot; we guarantee we stop at arr[first]
            while (pivot < v[scanDown])
                scanDown--;

            // if indices are not in their sublists, partition complete
            if (scanUp >= scanDown)
                break;

            // indices are still in their sublists and identify
            // two elements in wrong sublists. exchange
            temp = v[scanUp];
            v[scanUp] = v[scanDown];
            v[scanDown] = temp;

            scanUp++;
            scanDown--;
        }

        // copy pivot to index (scanDown) that partitions sublists
        // and return scanDown
        v[first] = v[scanDown];
        v[scanDown] = pivot;
        return scanDown;
    }
}
```

Figure 12: Quick Sort Implemented in C++



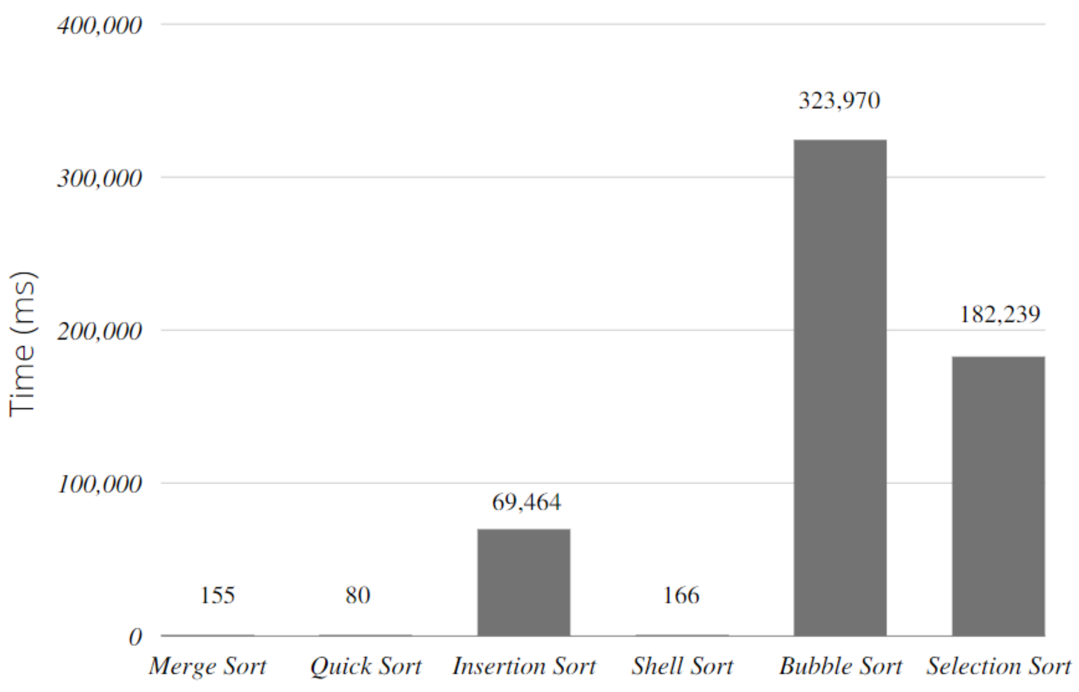
```
void quicksort(vector<double>& v, int first, int last)
{
    int pivotLoc;
    double temp;

    if (last - first <= 1)
        return;
    else if (last - first == 2)
    {
        if (v[last - 1] < v[first])
        {
            temp = v[last - 1];
            v[last - 1] = v[first];
            v[first] = temp;
        }
        return;
    }
    else
    {
        pivotLoc = pivotIndex(v, first, last);

        quicksort(v, first, pivotLoc);

        quicksort(v, pivotLoc + 1, last);
    }
}
```

Figure 13: Quick Sort Implemented in C++

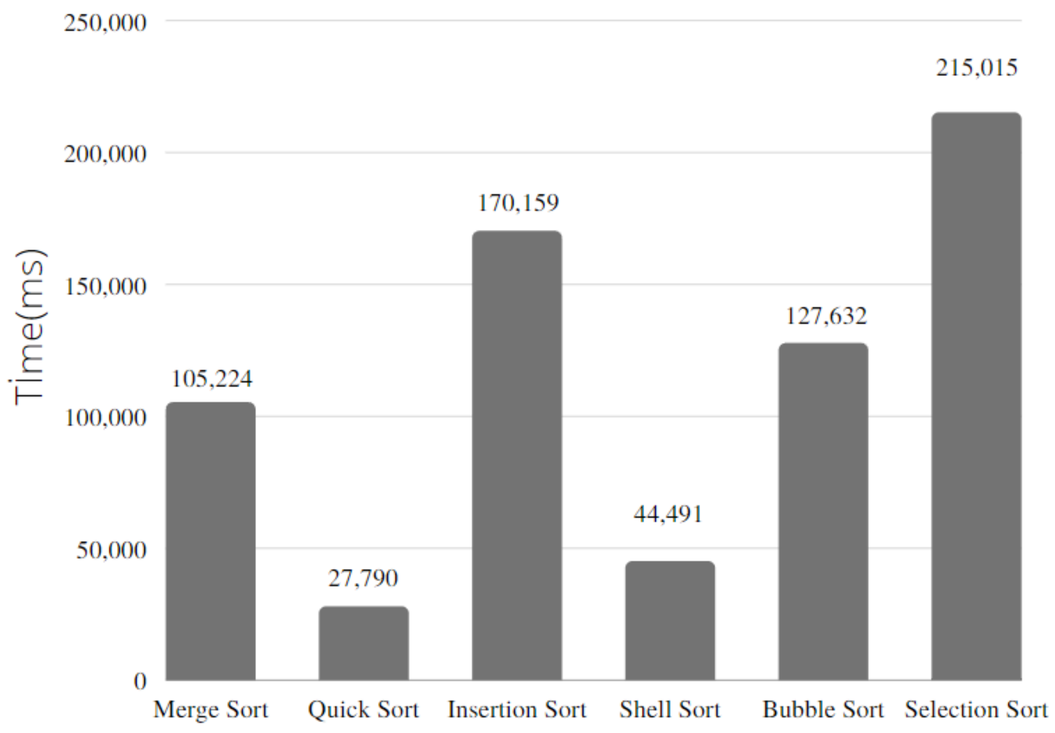
Testing

For the testing phase, algorithms will be tested using data of the same size of various algorithms in both the Java Programming language and the C++ programming language, and their running times will be recorded in milliseconds. The dataset is 259822 double data in the Query.txt file. the results of the tests are below:

The run time of sorting algorithms in Java Programming language.



The run time of sorting algorithms in C++ Programming language.



Conclusion

As a result, sorting algorithms are important algorithms with a wide range of applications. For various needs different algorithms should be used to ensure efficiency. The size, data type and order of the data set are important factors when choosing a sorting algorithm. In addition, the device in which the algorithm is used and the programming language also affect the sorting speed of the algorithms. Many standard sorting algorithms can be improved by analyzing which part of the algorithm is slowing it down and trying to optimize that part. All-in-all sorting algorithms are important algorithms in many ways and they are frequently used in several field.