Sorting Algorithms

Emmanuel Ameh

Sorting Algorithms in Java

The following assignment was completed using the Java Programming Language and Visual Studio Code. It includes Java code for BubbleSort, both integer and string inputs as well as Quicksort, both integer and string inputs. Screenshots of debugging console depicting executed code are included in this assignment. Attached in a zip file are the java codes for implementation. The Big-O Time Complexity Analysis of BubbleSort and QuickSort were also explained.

Java Code for BubbleSort with string and integer input

class BubbleSort {

    public static void bubbleSort(int arr[]) {

        int n = arr.length;

        for (int i = 0; i < n - 1; i++) // iterates O(n) times

            for (int j = 0; j < n - i - 1; j++) // iterates O(n) times

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

                    // swap arr[j+1] and arr[i]

                    int temp = arr[j];

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

                    arr[j + 1] = temp;

                }

        // hence, the total number of operations = O(n) \* O(n) = O(n^2)

    }

    /\* Prints the array \*/

    void printArray(int arr[]) {

        int n = arr.length;

        for (int i = 0; i < n; ++i)

            System.out.print(arr[i] + " ");

        System.out.println();

    }

    // Driver method to test above

    //public static void main(String args[]) {

        //BubbleSort ob = new BubbleSort();

        //int arr[] = {64, 34, 25, 12, 22, 11, 90};

        //ob.bubbleSort(arr);

       // System.out.println("Bubble-Sorted array");

     //   ob.printArray(arr);

   // }

public static void bubbleSort(String[] arr){

int n=arr.length;

for (int i=0;i < n-1;i++) {

for (int j=0;j < n-1;j++) {

if(arr[j].compareTo(arr[j+1]) > 0 ){

//Swap

String temp=arr[j];

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

arr[j+1]=temp;

}

}

}

for (String i : arr) {

System.out.print(i+" ");

}

System.out.println();

}

// Driver method to test above

    public static void main(String args[]) {

        BubbleSort ob = new BubbleSort();

        int arr[] = {64, 34, 25, 12, 22, 11, 90};

        ob.bubbleSort(arr);

        System.out.println("Bubble-Sorted array");

        ob.printArray(arr);

    }

{

String[] sarr={"This","is", "the", "Bubble", "Sorted", "String"};

bubbleSort(sarr);

System.out.println("");

}

}

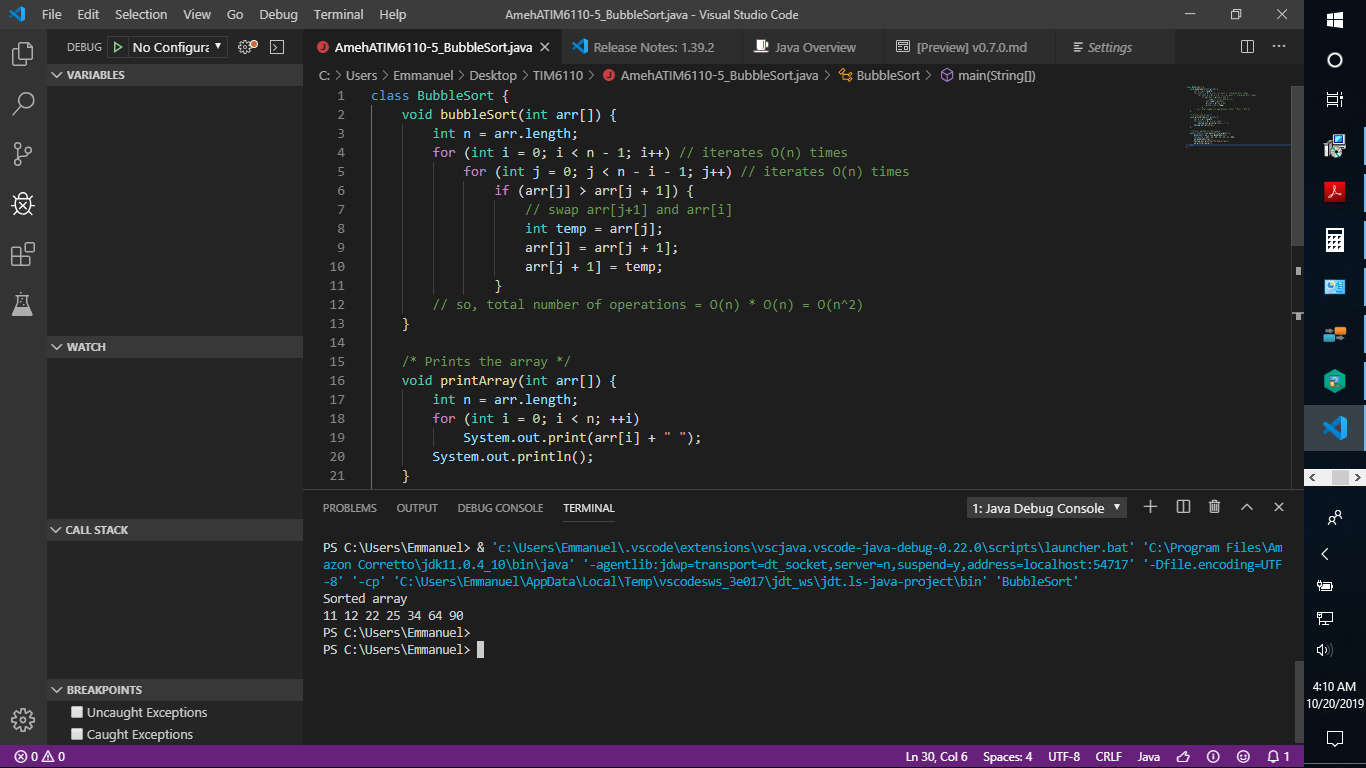
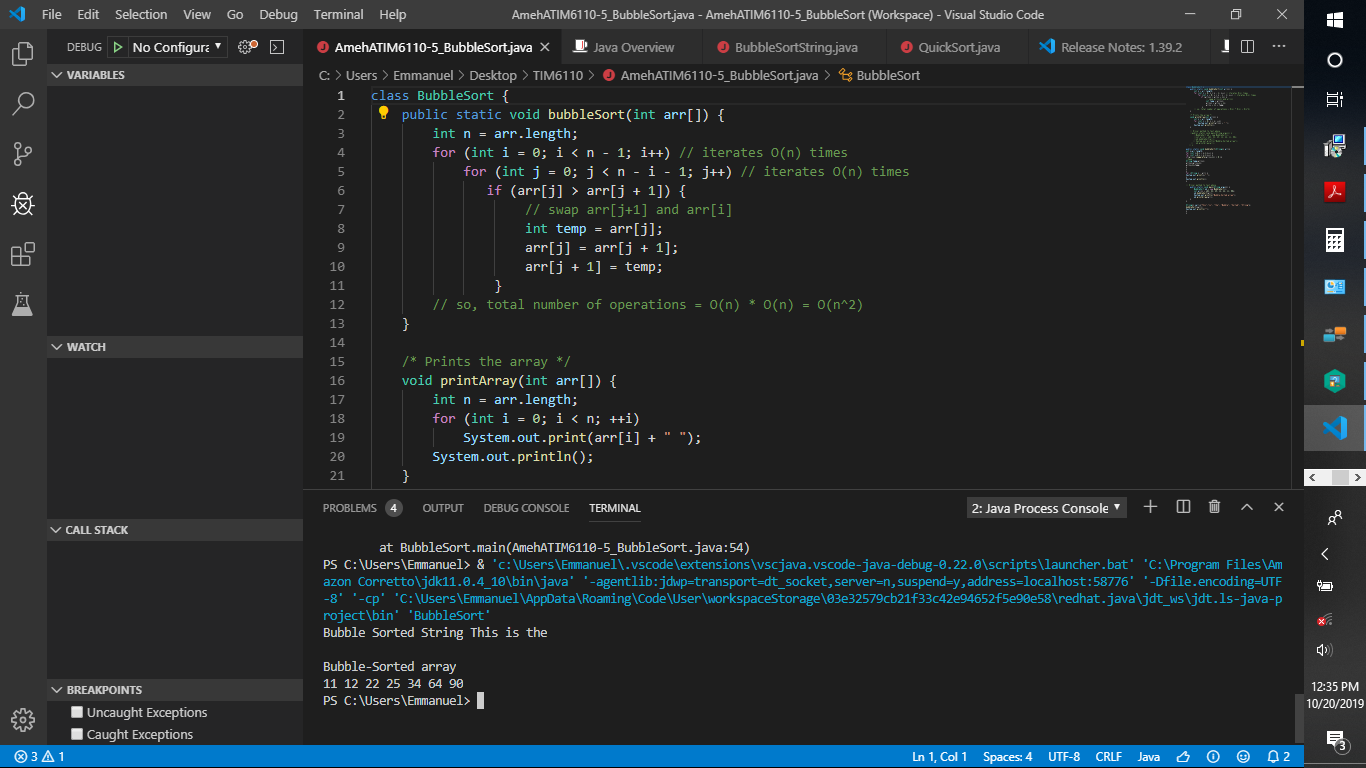
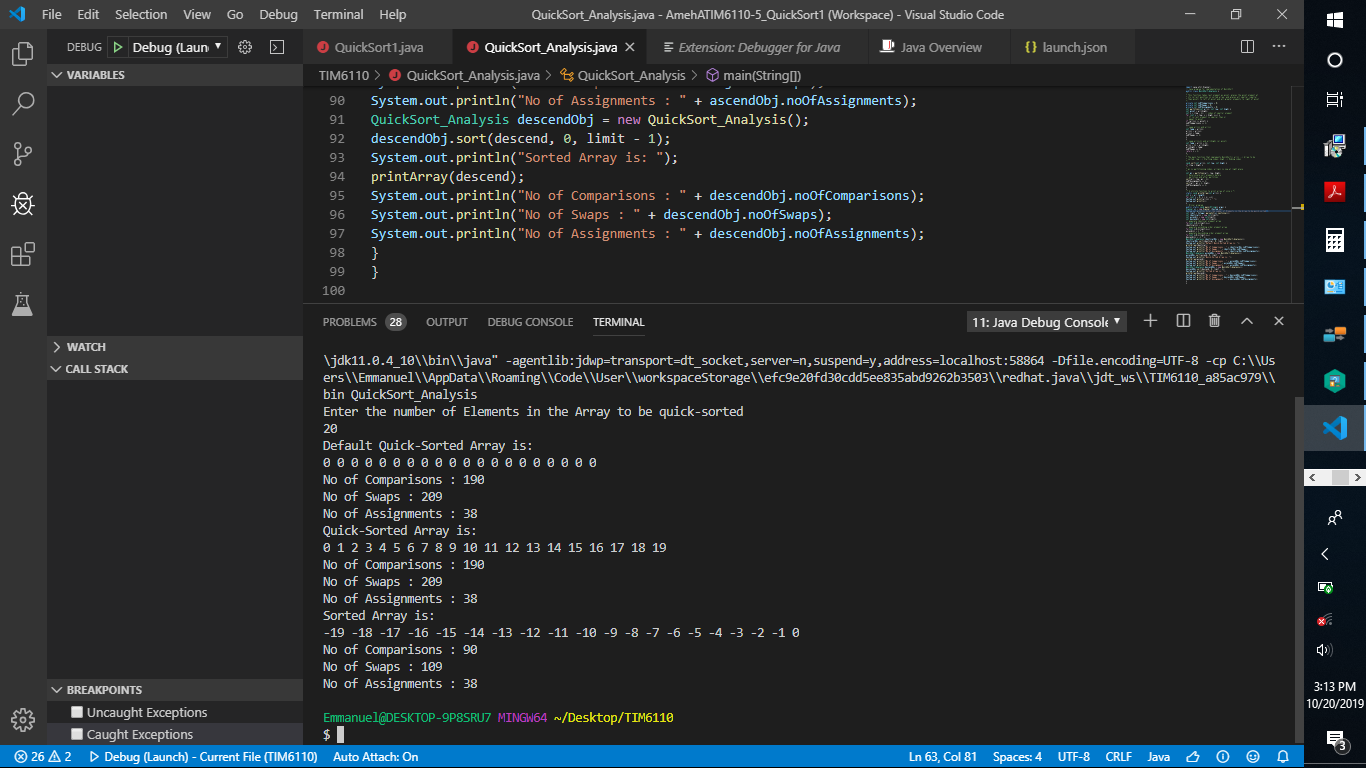


Fig 1. Debug Console showing integer BubbleSorted array.



*Fig 3.* Java Debug Console and Main Editor for Integer and String BubbleSort



*Figure 4.* Java Debug Console for Quick-Sorted Integer Array

**Java Code for Quick Sort Analysis**

import java.util.Scanner;

// Java program for implementation of QuickSort

public class QuickSort\_Analysis {

/\*

\* 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

\*/

private int noOfComparisons = 0;

private int noOfSwaps = 0;

private int noOfAssignments = 0;

int partition(int arr[], int low, int high) {

int pivot = arr[high];

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

for (int j = low; j < high; j++) {

// If current element is smaller than or

// equal to pivot

if (arr[j] <= pivot) {

noOfComparisons += 1;

i++;

// swap arr[i] and arr[j]

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

noOfSwaps += 1;

}

}

// swap arr[i+1] and arr[high] (or pivot)

int temp = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp;

noOfSwaps += 1;

return i + 1;

}

/\*

\* The main function that implements QuickSort() arr[] --> Array to be

\* sorted, low --> Starting index, high --> Ending index

\*/

void sort(int arr[], int low, int high) {

if (low < high) {

/\*

\* pi is partitioning index, arr[pi] is now at right place

\*/

int pi = partition(arr, low, high);

// Recursively sort elements before

// partition and after partition

sort(arr, low, pi - 1);

noOfAssignments += 1;

sort(arr, pi + 1, high);

noOfAssignments += 1;

}

}

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

static void printArray(int arr[]) {

int n = arr.length;

for (int i = 0; i < n; ++i)

System.out.print(arr[i] + " ");

System.out.println();

}

// Driver program

public static void main(String[] args) {

Scanner sc = new Scanner (System.in);

System.out.println("Enter the number of Elements in the Array to be quick-sorted");

int limit = Integer.parseInt(sc.nextLine());

int identical[] = new int[limit];

int ascend[] = new int[limit];

int descend[] = new int[limit];

// populate identical element array

for(int i=0;i<limit;i++)

identical[i] = 0;

// populate ascending order element array

for(int i=0;i<limit;i++)

ascend[i] = 1 \* i;

// populate descending order element array

for(int i=0;i<limit;i++)

descend[i] = -1 \* i;

QuickSort\_Analysis identicalObj = new QuickSort\_Analysis();

identicalObj.sort(identical, 0, limit - 1);

System.out.println("Default Quick-Sorted Array is: ");

printArray(identical);

System.out.println("No of Comparisons : " + identicalObj.noOfComparisons);

System.out.println("No of Swaps : " + identicalObj.noOfSwaps);

System.out.println("No of Assignments : " + identicalObj.noOfAssignments);

QuickSort\_Analysis ascendObj = new QuickSort\_Analysis();

ascendObj.sort(ascend, 0, limit - 1);

System.out.println("Quick-Sorted Array is: ");

printArray(ascend);

System.out.println("No of Comparisons : " + ascendObj.noOfComparisons);

System.out.println("No of Swaps : " + ascendObj.noOfSwaps);

System.out.println("No of Assignments : " + ascendObj.noOfAssignments);

QuickSort\_Analysis descendObj = new QuickSort\_Analysis();

descendObj.sort(descend, 0, limit - 1);

System.out.println("Sorted Array is: ");

printArray(descend);

System.out.println("No of Comparisons : " + descendObj.noOfComparisons);

System.out.println("No of Swaps : " + descendObj.noOfSwaps);

System.out.println("No of Assignments : " + descendObj.noOfAssignments);

}

}

**Java Code for Quick Sort Integer and String**

// Java program for implementation of QuickSort

class QuickSort

{

    /\* 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];

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

        for (int j=low; j<high; j++)

        {

            // If current element is smaller than the pivot

            if (arr[j] < pivot)

            {

                i++;

                // swap arr[i] and arr[j]

                int temp = arr[i];

                arr[i] = arr[j];

                arr[j] = temp;

            }

        }

        // swap arr[i+1] and arr[high] (or pivot)

        int temp = arr[i+1];

        arr[i+1] = arr[high];

        arr[high] = temp;

        return i+1;

    }

    /\* The main function that implements QuickSort()

      arr[] --> Array to be sorted,

      low  --> Starting index,

      high  --> Ending index \*/

    void sort(int arr[], int low, int high)

    {

        if (low < high)

        {

            /\* pi is partitioning index, arr[pi] is

              now at right place \*/

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

            // Recursively sort elements before

            // partition and after partition

            sort(arr, low, pi-1);

            sort(arr, pi+1, high);

        }

    }

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

    static void printArray(int arr[])

    {

        int n = arr.length;

        for (int i=0; i<n; ++i)

            System.out.print(arr[i]+" ");

        System.out.println();

    }

    // Driver program

    public static void main(String args[])

    {

        int arr[] = {1000, -700, +800, 9000, -100, +5000};

        int n = arr.length;

        QuickSort ob = new QuickSort();

        ob.sort(arr, 0, n-1);

        System.out.println("The quick sorted integer and string array is as follows");

        printArray(arr);

    }

}

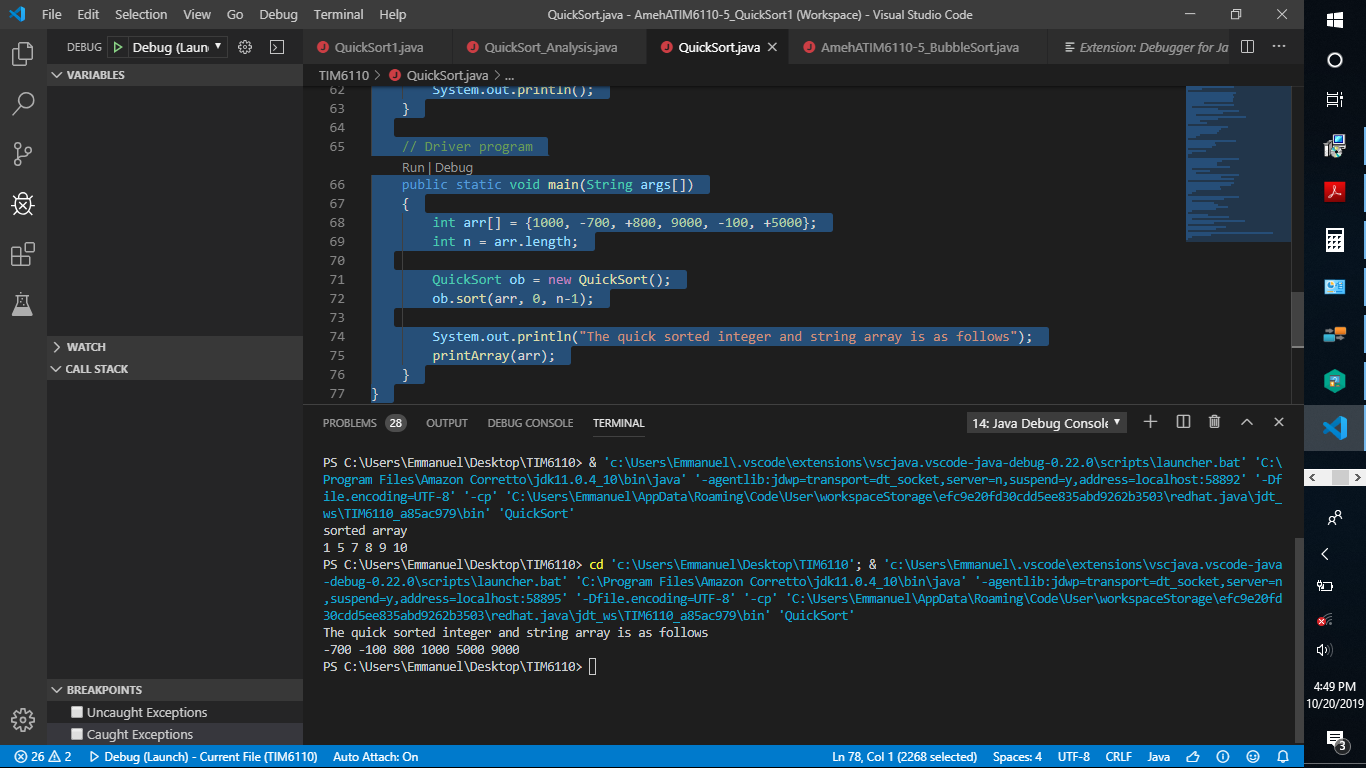


Fig 5. Java Debug console depicting Quick sort of integer and string arrays

**Big-O for Algorithm’s Complexity**

Big-O is the term given to algorithmic efficiency. Efficiency as to do with processing speed and time, and other resources used during code execution. The Big-O of Quicksort’s Complexity was explained using the Java code “Quicksort\_Analysis”. For the BubbleSort, the for loop with n-1 iterations, iterates O(n) times. The subsequent for-loop of array of length ‘j’, the number of times for iteration is also O(n). The total number of operations is therefore O(n) \* O(n) = O(n^2). O(n^2) represents algorithms whose performance is proportional to the arithmetic square of the input data set’s size. Algorithms that involve nested iterations over the data set exhibit O(n^2). In this case there are two for loops executed in the BubbleSort assignment. Deeper nested iterations, such as three or four for-loops will produce O(N3), O(N4) and high power-orders.

QuickSort is a ‘divide and conquer’ type of algorithm just as the Merge Sort algorithm is.

Time elapsed during QuickSort is mathematically given as T(n) = T(k) + T(n-k-1) + (n). This

translates to a best-case scenario of O(NLogN). Quick Sort is generally preferred over Merge

Sort for arrays because it doesn’t require extra storage, whereas merge sort does require O(N)

extra storage, where N is the array size which may translate to more expense. De-allocating and

allocating extra space used by merge sorting correspondingly increases the run-time of the

algorithm. Quick Sort is also considered cache friendly as a sorting algorithm with great

reference locality when applied with arrays.