AQA

Sound Of Sorting

Non-exam assessment

Cameron Dennis

Table of Contents

Analysis

Introduction

The program is to be an algorithm visualiser designed to demonstrate and display the process behind a variety of sorting algorithms. The software is designed with students and classroom use in mind - to be used as a learning aid / demonstration tool for those interested in a more in-depth approach into learning the workings of how data is sorted (each individual comparison and write/read operation being made) beyond the usual whiteboard demonstration. As for the end users of this project, I will be utilising the help of my peers who have an interest in teaching / tutoring computer science. To help keep engagement with the program an audible 'tone' should be played corresponding to the value of the current data being compared. Similar applications have previously existed; however these have been almost entirely accessed through simple website demonstrations or YouTube videos – these have the downside in which they cannot be slowed down or often lack finer details in the visualisations.

The user will be able to select the algorithm to sort by as well as the method used to 'shuffle' the data accessed via drop down menus. The user will be able to control sliders to determine the time between each step in the sort; the size of the array to be sorted; the pitch of the tone played. The user, through the use of buttons, will be able to start and pause the sort; step through the individual steps of the sort; reset the array back to the original state; mute the sound of the tones; select a random sort. The program will have two modes of visualisation – bars and image. At the end of each sort the data will be saved into a csv file for later comparisons by the user.

Research

Overview

To build up a list of objectives for my project, I first had to investigate how sorting algorithms are currently being taught to students. The AQA A Level specification only requires the teaching of 2 different sorting algorithms – bubble sort and merge sort, this only covers a very small basis out of the hundreds of already established sorting algorithms.

In order to teach the algorithms, traditionally a teacher would use a large whiteboard at the front of the classroom to trace out the algorithm on a small, predetermined set of data. This creates an issue of students potentially disengaging with the demonstration as they may find it to be a little 'dull'; coupled with the fact whiteboards are slowly being removed from classrooms in favour of modern projectors and displays, this method is becoming harder and less effective to teach. By hand tracing the algorithm on such a limited set of data, the students fail to see how such an algorithm will perform on larger, real-world data; the differing time complexities between the algorithms and the strengths / weaknesses of each. As a result, the students are unable to determine which algorithms are superior to others and in which circumstances one may be more appropriate over another.

Time Complexity

In computer science, the computational complexity of an algorithm is the amount of a resource that is needed to run it. Time complexity describes the amount of time an algorithm would take to run. Time complexity is measured by:

1. Assume each elementary operation is completed in a fixed amount of time.
2. Counting the number of elementary operations performed by the algorithm.

As the amount of time an algorithm takes to complete can vary greatly between different inputs of the same size, time complexity is often measured and compared using worst-case time complexity which is the greatest length of time needed for a given input size. The other common measurement is average-case time complexity which is average length of time needed for a given input size. Rarely, best-case time complexity is used which is the smallest length of time needed for a given input size.

Big O notation is often the standard notation used to describe how the time complexity of an algorithm scales with its input size. Common notations of Big O include:

- O(1) - Constant time
- O(log n) - Logarithmic time
- O(n) - Linear time
- O(n log n) - Linearithmic time
- $O(n^2)$ - Quadratic time
- $O(2^n)$ - Exponential time
- O(n!) – Factorial time

Where each indicates how runtime scales as n increases.

The time complexity of sorting algorithms is an important measure of how performant they are and is usually the best indicator of how they will perform on a data set of a given size. As such, my program will have to render each step of the various sorting algorithms in a fixed length of time for the end user / students to have a clear visual representation of this data. Coupled with this, a useful feature would be to have a built-in timer so comparisons can be easily drawn between different algorithms and / or the same one on different sets of input data.

Similarly, space complexity describes the amount of memory an algorithm will take to run and is also most often described using Big O notation (where it indicates how memory usage scales as n increases). However, this will prove to be difficult to visualise to an end user and so won't be a focus point of the project.

Algorithms

Initial Shuffling of Data

In order for the data to be correctly sorted, it first must be shuffled. To accomplish this, I have taken the decision to give the user the choice to either reverse the order of the array or to create a 'truly shuffled' array in which every permutation is equally likely.

To reverse the array, I increment through from the start to the midpoint, swapping each element with its equivalent from the end point.

To truly shuffle the array, I decided upon a variation of the 'Fisher-Yates shuffle'. In the original shuffle:

- You have a list of numbers arranged in order from 1 to N (number of elements)
- Select a random number between 1 and the remaining number of elements, X
- Remove the number in the X position of the list and copy it to the end of a separate list
- Repeat the previous 2 steps until all numbers have been moved to the new list

Whereas in a more modern approach devised by Richard Durstenfeld, the algorithm I chose for the shuffle, has been proven more efficient when performed by a computer as instead of counting the amount of remaining numbers in the list every iteration, the number to be moved is instead swapped with the last unaltered number at the end of the list each iteration. In doing so the time complexity of the algorithm has been reduced from O(n^2) to O(n).

The Random class implemented in java is a pseudorandom number generator; meaning that the 'random' output is determined by an algorithm which employs 'deterministic chaos' based of an initial seed value (in this case a value calculated using `System.nanoTime()`). Some of the downsides of this is that the output will eventually repeat itself after enough iterations although this will be far more then could be feasibly achieved by a human or that it is possible for two unique instances of the Random class to have the same initial seed resulting in an identical chain of outputs although this is incredibly unlikely to occur.

The only way around this would be to use a hardware random number generator which exploits a physical source of randomness such as the random atmospheric noise, however this is completely unnecessary for most projects resulting in cryptography being the only practical application for such a device.

Implemented Sorting Algorithms

A sorting algorithm is an algorithm that puts a list of elements into an order. In the scope of this project, a sorting algorithm will take an array of integer values and arrange them into ascending order (smallest to largest). These algorithms are the main feature the program is centred around, and so a lot of care must be taken to make sure they work as intended.

The planned sorting algorithms are: Bogo, Bozo, Bubble, Cocktail Shaker, Comb, Exchange, Gnome, Insertion, Merge, Odd-Even, Pancake, Quick, Selection, Shell, Slow

Bubble Sort & Variations

Bubble sort is a simple in-place algorithm in which iterates over an array of elements comparing each element to the next and swapping accordingly, causing the largest element of each pass to 'bubble' to the top. Bubble sort is often used in an educational setting due to its simple implementation but suffering from being inefficient in real world use cases.

The average time complexity of Bubble sort is O(n^2) and has a space complexity of O(n) as it has to iterate over the whole array for each element in the array. A simple optimisation would be to have it finish each subsequent iteration 1 element sooner due to the final element of each pass being in the correct place.

Here is an example of the Bubble Sort algorithm with the explained optimisation written in Java.

```java
private void bubbleSort(int[] array) {
    int end = array.length - 1;
    boolean swap;
    do {
        swap = false;
        for (int i = 0; i < end; i++) {
            if (array[i] > array[i + 1]) {
                int temp = array[i];
                array[i] = array[i + 1];
                array[i + 1] = temp;
                swap = true;
            }
        }
        end--;
    }
    while (swap);
}
```

Several other sorting algorithms can be categorised as variations/optimisations of bubble sort such as 'Odd-Even Sort' and 'Cocktail Shaker Sort' in which the latter repeatedly performs a bubble sort going from left to right followed by a bubble sort from right to left, causing smaller values in the array to more quickly reach the start of the array.

Another, more effective variation is 'Comb Sort' in which allows turtles (small values at the end of the array) to move more positions per iteration. It does this by performing a bubble sort but instead of comparing and swapping one element to the next (gap of 1), it implements a much larger gap between the elements. This gap is initially the length of the array shrunk by the scale factor and is further shrunk by the scale factor each iteration until a gap of 1 is reached, in which case this pass is equivalent to a standard bubble sort. The scale factor is most optimal at a value of ~1.3 as too small a value would make many unnecessary comparisons and too large a value would mean the turtles are not effectively dealt with.

While the worst-case time complexity of Comb Sort remains as O(n^2), the average time complexity is improved to O($\frac{n^2}{2^p}$) where p is the number of gap increments. The space complexity remains O(n).

Here is an example of the Comb Sort algorithm written in java.

```java
private void combSort(int[] array) {
    boolean sorted = false;
    int gap = array.length;
    do {
        gap = (int) Math.floor(gap / 1.3);
        if (gap <= 1) {
            gap = 1;
            sorted = true;
        }

        for (int i = 0; i + gap < array.length; i++) {
            if (array[i] > array[i + gap]) {
                int temp = array[i];
                array[i] = array[i + gap];
                array[i + gap] = temp;
                sorted = false;
            }
        }
    }
    while (!sorted);
}
```

Impractical Sorting Algorithms

The following three algorithms do not operate in polynomial time and are therefore highly impractical.

Bogo Sort is a highly impractical sorting algorithm in which may never produce a sorted array due to the random nature of it and as such is only useful in an educational setting. However the average time complexity to produce a sorted array is O($n \times n!$). Bogo sort works by generating a permutation of the array (shuffling) and then checking if said permutation is sorted, if not this cycle repeats until the array is sorted.

A slightly optimised, but still highly impractical variation on Bogo Sort is 'Bozo Sort' where rather than shuffling the whole array, two elements are randomly selected and swapped each iteration. The algorithm still may never produce a sorted array but the average time complexity to produce a sorted array is improved to O($n!$).

Here is an example of the Bogo Sort algorithm written in pseudocode.

```
while not sorted(array):
    shuffle(array)
```

The third highly impractical sorting algorithm implemented is Slow sort. Slow sort operates on the principles of 'multiply and surrender' – a parody of the 'divide and conquer' paradigm. The best-case time complexity of Slow sort is O($n^{\log(n)}$)

Here is an outlined version of the Slow Sort algorithm.

1. Recursively sort the first half of the array.
2. Recursively sort the second half of the array.
3. Compare the results of step 1 and 2, to find maximum and place at end of the array.
4. Recursively sort the whole array excluding the maximum at the end.

Other Notable Sorting Algorithms

Insertion

Insertion Sort has an average time complexity of O(n^2), however it is very efficient if the array is already sorted to a large degree or very small in size, and as such an optimised Quick Sort implementation will utilise Insertion Sort for arrays below a predetermined size. Insertion Sort possesses a unique property in that it is considered to be 'online' – the ability to sort data as it is input.

Insertion Sort operates by for each element to be sorted, from start to end of the array, an 'insert' operation is invoked to insert the element into the correct position. The 'insert' operation works by starting at the end of the sorted portion of the array and moving each element one position towards the end until a suitable position is found for the new element and storing it there.

Merge

Merge Sort follows the 'divide and conquer' paradigm and has both an average time complexity and worst-case time complexity of O($nlog(n)$) while having a space complexity of O(n). The implemented merge sort algorithm uses a recursive top-down approach,

Here is an outlined version of the Merge Sort algorithm.

1. Starting with the original array, split the array into 2 sub-arrays and invoke a Merge Sort on each sub-array until a length of 1 is reached (sorted).
2. Involve a merge procedure on the sub-arrays – merging 2 sub-arrays into one larger array with the elements correctly ordered.
3. Repeat step 2 until only one array remains – the sorted output array.

Quick

Quick Sort follows the 'divide and conquer' paradigm and has an average time complexity of O($nlog(n)$) and a worst-case time complexity of O(n^2) while having a space complexity of O($\log(n)$) due to the implementation of Hoare's partition scheme. In the traditional naïve implementation of Quick Sort, only 1 pointer is used whereas in Hoare's implementation 2 pointers are used – one from each end, moving towards each other.

Here is an outlined version of the Quick Sort algorithm.

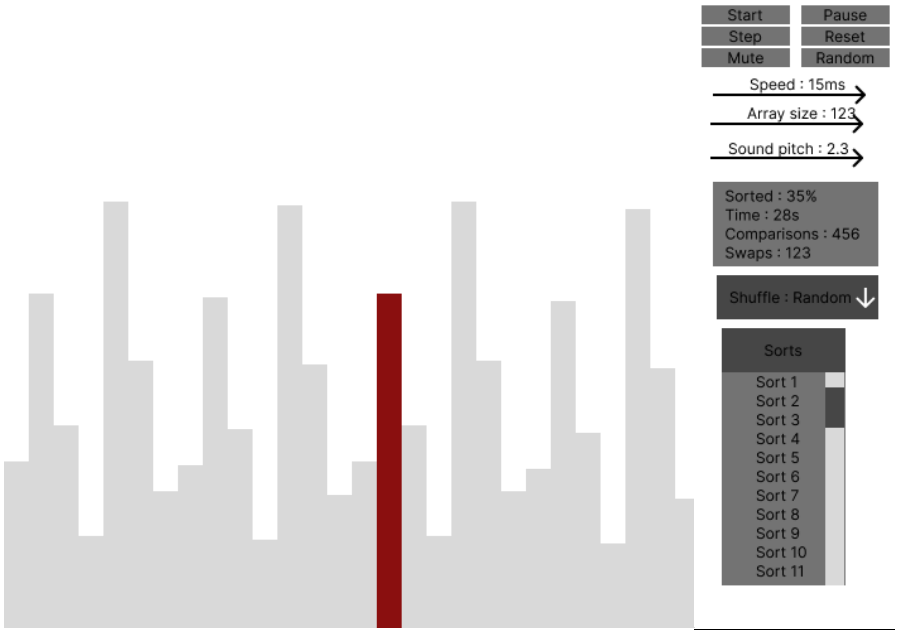
1. Pick a value in the range to use as the pivot point (I used the middle value for simplicity).
2. Initialize pointers at start and end of the array.
3. Increment the left pointer until an element is found greater than or equal to the pivot value.
4. Decrement the right pointer until an element is found less than the pivot value.
5. Swap the values found at the two pointers.

6. Repeat steps 3-5 until the left pointer is greater than or equal to the right pointer.
7. Repeat steps 1-6 recursively for the two sub arrays formed by splitting the array at the pointer value (steps 1-6 are known as partitioning).

User Interface

To construct the mock-up UI, I used the Figma web application as it is a purpose-built design tool for interface designing. The UI is composed of 2 major components – The control / setting panel and the graph visualisation, this allows for the parameters to always be easily visible and changed live without obstructing the visualisation. The UI is designed with the ability to be controlled and navigated using solely a mouse however I do plan on binding certain common actions e.g. start/pause to keyboard inputs as well.

Initial design concept - Figma



I chose the primary method of representing the individual data values in the array as bars (similar to that of a bar graph) as this was the easiest way to convey the value associated with each element in an intuitive way to the user. In order to highlight any extra/important pieces of data within the sorting algorithms e.g. current element and boundary elements the colour of the corresponding bar is changed to either red or green depending on the circumstance. The narrow nature of the bars allows for more elements to be visible at once on the display leading to larger array sizes being visualised without cluttering the visualisation.

The secondary option of visualising the array data is through the pixels of an image uploaded to the program by the user. This will provide a more creative and potentially engaging approach to visualising the data at the detriment to highlighting finer details in the sorting algorithms as opposed to the bars e.g. the change of colour on certain bars to represent key values.

Application Framework

For the rendering I am using the libGDX application framework in which uses the Lightweight Java Game Library (LWJGL) as the backend. I have chosen this platform due to the ease of use of its shape renderer component which renders simple shapes such as rectangles in batches to speed up performance. Due to the vast majority of the application being displayed to the user consisting of vertical rectangles I felt this to be an acceptable decision combing ease of use with reliable performance. I also chose to use libGDX as it provides input handling through the input processor.

An example piece of code using the shape renderer to render a simple rectangle at position (50, 100) with a size of 200 x 200.

```
public class MyGdxGame extends ApplicationAdapter {
    5 usages
    ShapeRenderer sr;

    @Override
    public void create () {
        sr = new ShapeRenderer();
    }

    @Override
    public void render () {
        sr.begin(ShapeRenderer.ShapeType.Filled);
        sr.rect( x: 50,  y: 100,  width: 200,  height: 200);
        sr.end();
    }

    @Override
    public void dispose () {
    sr.dispose();
    }
}
```

Objectives

The overall objective of my project is to create an educational tool for the purposes of demonstrating and explaining sorting algorithms. Below is a list of specific goals required to achieve my objectives.

1) User Interface

a) An area dedicated to controls and settings for the operation of the program.
   i) Containing the following buttons: Start, Pause, Step, Reset, Mute, Random, Render Switch.
      (1) Each main control is also bound to a keyboard hotkey.
   ii) Containing the following sliders for fine value control: Speed (delay between operations), Size (size of the array), Volume (volume of audio), Pitch (pitch of audio).
      (1) Sliders can also be adjusted using the keyboard arrow keys.
   iii) Containing dropdown menus of buttons for shuffle and sort selection.
      (1) Dropdowns can be scrolled using the scroll wheel and keyboard arrow keys.

b) An information panel containing live statistics of the current sorting algorithm.
   i) The following statistics are displayed: Sort name, Time taken, Comparisons, Swaps, Writes, Auxiliary writes.

c) A notification system to alert the user.
   i) The notification should display relevant error messages and general notifications regarding the program.
   ii) The notification should be easily visible utilising a red text colour.
   iii) The notification should be cleared with a hotkey.

d) An area dedicated to displaying the visualisation.
   i) There will be two modes for the visualisation: bars (default) and image.
      (1) The bars option should be the default upon loading the program
         (a) Each bar should represent one element in the array with the height being mapped to the size of the value.
         (b) Each bar should change colours to highlight important values dependent on the current sorting algorithm.
      (2) The image option should let the user map the array elements to an image file of their choosing.
         (a) Each pixel in the image should represent one element in the array with the position being mapped to the size of the value.
         (b) A placeholder should be in place to direct the user on how to add an image to the program if the user has not already done so.

e) The entire program should be accessible using only a mouse.
   i) Keyboard hotkeys should also be available for quick access to main / common button actions.

2) Sorting Algorithms

a) A variety of unique sorting algorithms should be implemented (at least 10).
   i) Each sorting algorithm should take an unsorted array (shuffled) and return it to its original sorted form through a series of algorithms.
   ii) The user should be able to select from all the implemented sorting algorithms.
   iii) The user should have the ability to control the speed and playing / pausing of a sort.

b) The user should be able to select a shuffle option to shuffle the array prior to sorting.

c) The user should be able to adjust the size of the array to be sorted.
d) The array size should be automatically adjusted to accommodate the size of an image.
e) A timer should be automatically started with the start of a sorting algorithm.

3) Audio

a) An audible tone should be generated upon certain events in a sorting algorithm e.g. a comparison.
    i) The note of the tone should correspond to the relevant data values – the greater the value, the higher the note.
b) The audio should have an '8-bit like' quality.
c) The user should have the ability to mute the audio and adjust the volume / pitch.

4) External Files

a) A CSV file should be written at the end of a successful sort.
    i) The file is to contain: Date / time, Sort name, Number of elements, Time taken (formatted and raw), Comparisons, Swaps, Writes, Auxiliary writes.
    ii) If the CSV file already exists, the data should be appended to it on a new line.
    iii) An appropriate error message is to be displayed to the user if the file cannot be saved.
b) An image file should be able to be selected and loaded into the program by the user.
    i) The image file should be input through 'dragging and dropping' an image file onto the program window.
    ii) The loaded image should be reset / if the array size changes.
    iii) An appropriate error message is to be displayed to the user if the image cannot be loaded.

Design

Overview

The project is an educational tool centred around the visualisation of a variety of sorting algorithms. The user has the ability to choose between which sorting algorithm to visualise; the method in which the data is shuffled; the number of elements and the parameters for the visualization – speed, audio volume and pitch. The user will also have the ability to play and pause the visualisation at any time while also stepping through it step by step while being able to toggle between two modes of visualisation – bars and image. The program will display various statistics related to the current visualisation in an easily readable format e.g. number of comparisons and time taken. The final statistics at the end of each sort are to be written and saved to a file for future reference.

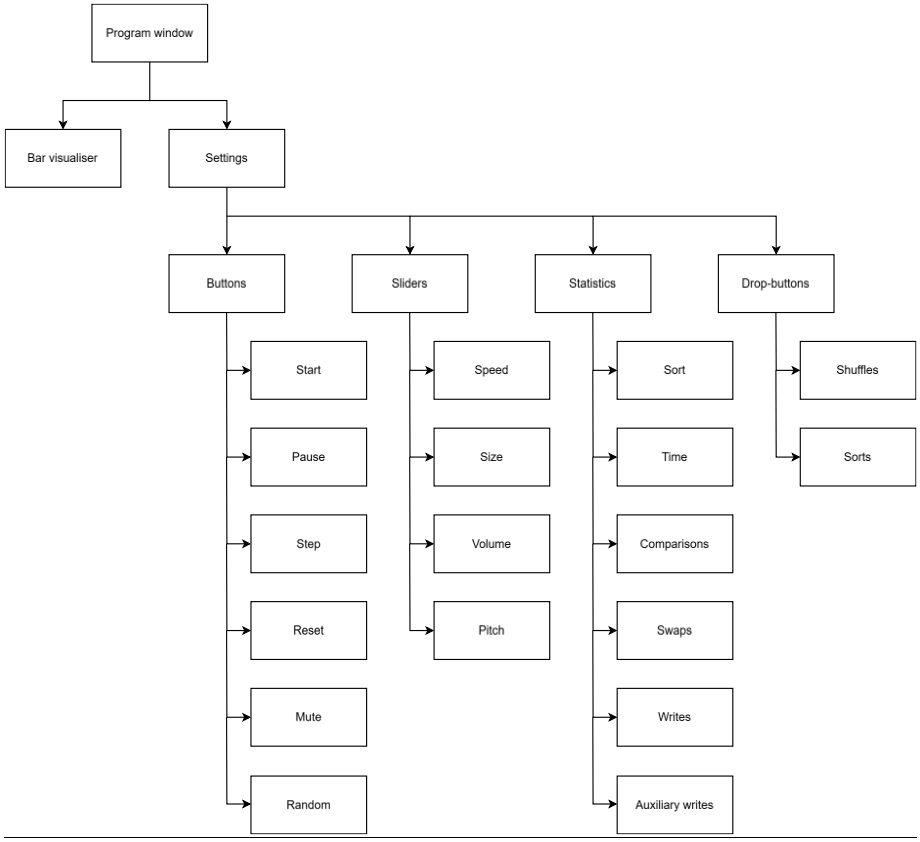
Language, External Programs and Libraries

I chose to write the program using Java as this is the language I am most familiar with and have used for most of my previous programs. By using java, I gained access to the libGDX application framework which is a cross-platform development framework based on OpenGl in which I used for the rendering and input handling for my program. For audio, I used the javax.sound.midi package to create midi data in order to produce sound.

User Interface

Program Window Structure

Hierarchy chart containing the UI components.



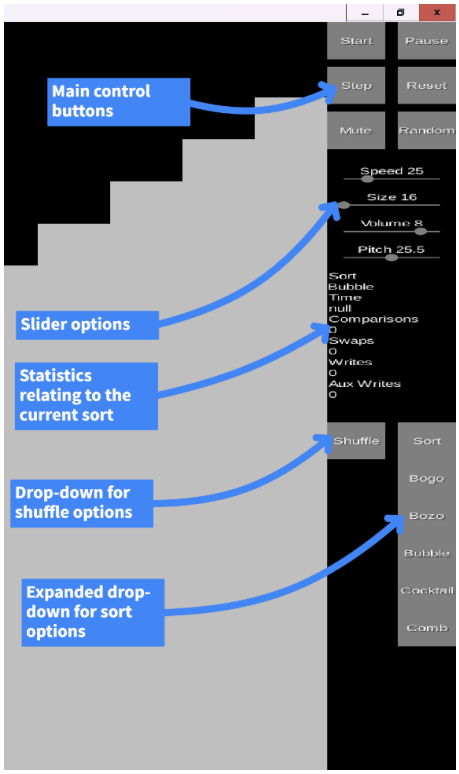
The visualisation bars will take a prominent place on the window with the settings panel situated to the right-hand side. The user will be able to access the settings panel at any point during use without obstructing the visualization of the sorting algorithm. This will also allow the statistics relating to the sort to be permanently visible.

Overall User Interface

The program in the default state with the sort and shuffle drop-downs expanded.

Settings Panel

The buttons can be activated using a left mouse click with each of the main 6 buttons being also bound to a corresponding keyboard input. The drop buttons can be expanded and collapsed by clicking on the corresponding label and are scrolled by either using the scroll wheel or the arrow keys. The sliders can be adjusted by either clicking and dragging the small position indicators or when using the arrow keys the last interacted slider will be adjusted.



Statistics Panel

The statistics panel is a sub-section of the settings panel containing a series of extra information relating to the current sort. The tracked statistics / values are:

- Current sort
- Time elapsed
- Comparisons made
- Swaps made
- Writes made
- Auxiliary writes made

At the end of a complete sort, these statistics are also output to both the console (if applicable) and a csv file for later comparison.

UI Input Implementations

The UI Input classes consist of the Buttons, Drop Buttons, Sliders, Input Manager, Button Methods and Text Method Pair. These classes have been designed to be modular and easily implemented into future programs. Initially the development of the UI Input classed was done in a separate project altogether as this avoided accidently inter-twining the UI code with any existing methods, helping to keep it more modular and I didn't need to worry about the existing features being in the way of testing.

The Input Manager class handles all the input detection for the program including mouse and keyboard events. This class would need to be modified if implemented into a future program to accommodate the program specific button methods and if not all the input modules are to be included.

The Button Methods class stores the enumerators and corresponding method calls for every method that can be activated by the other input classes (buttons and sliders). This class would need to be modified if implemented into a future program for the correct methods and enumerators to be referenced.

The text Method Pair class acts as a utility container class allowing text (Strings) and the Method enumerators (from the Button Methods Class) to be stored inside a single object to be easily pared up and passed through as a single array into a drop-button instance.

The Button class acts as the primary and most basic form of interaction between the user and the program allowing control over the main aspects of the software. The current implementation of buttons allows them to be any colour but in this instance, they consist of a light grey rectangle with white text rendered on top.

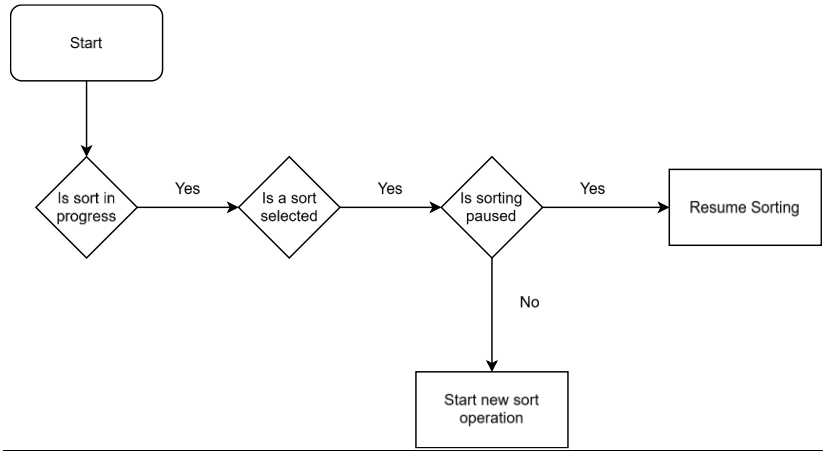
The Drop Button class extends the Button class and toggles a dropdown of sub buttons (Button instances) upon being clicked. The drop downs can be scrolled through using either the scroll wheel or the arrow keys.

The Slider class allows for finer control over certain aspects of the program. Each slider consists of a thin rectangular bar with a small circle acting as the pointer for the current value. Text is rendered above the slider to indicate the current value and its function.

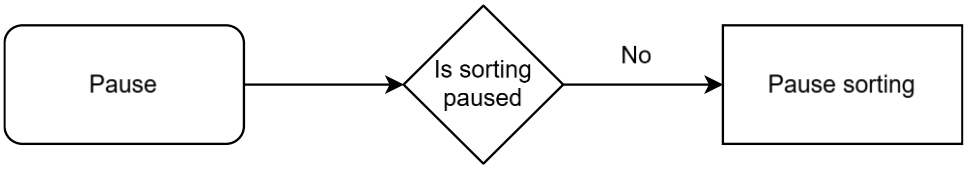
Button Implementations

Various flow diagrams demonstrating the algorithms performed when each main control button is used.

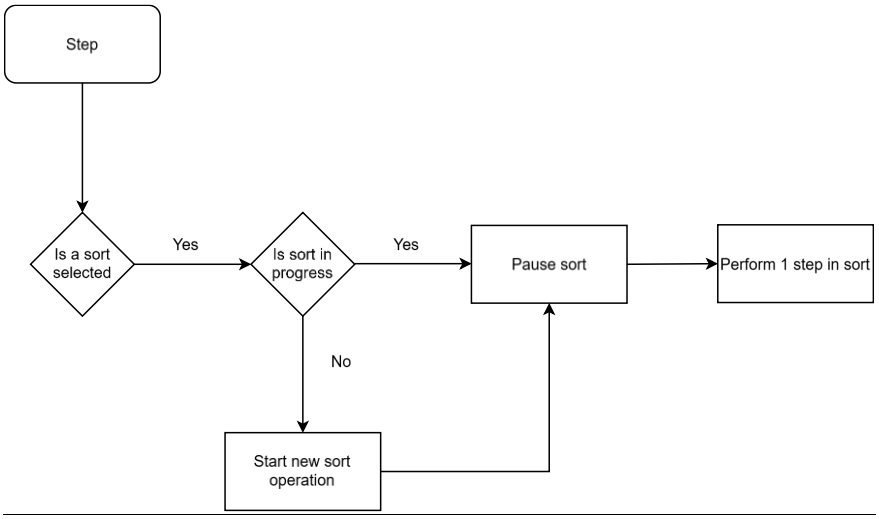
Start button



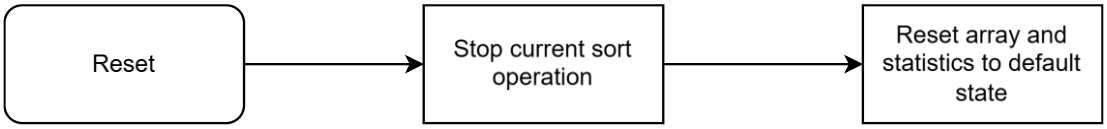
Pause Button



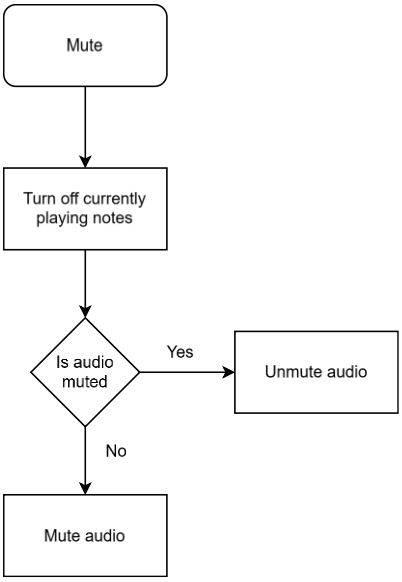
Step Button



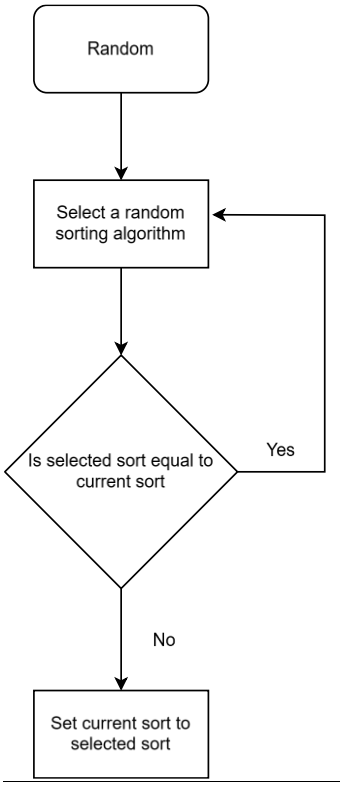
Reset Button



Mute Button



Random Button

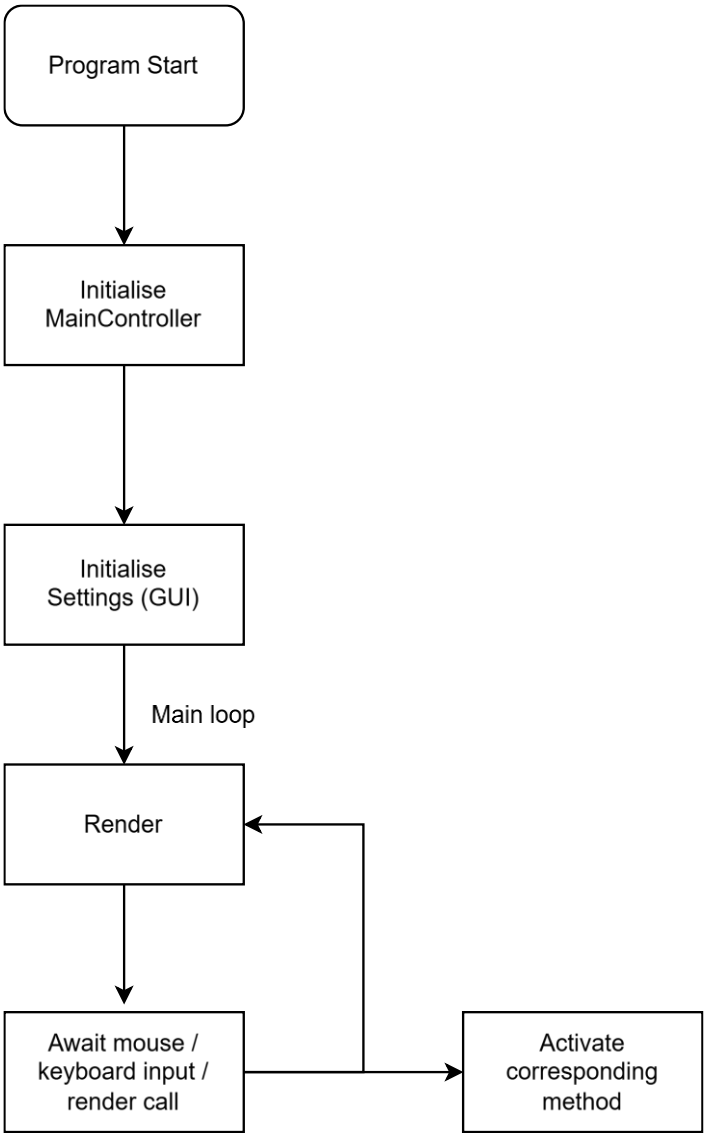


Main Operational Algorithms

Key algorithms for the operation of the program.

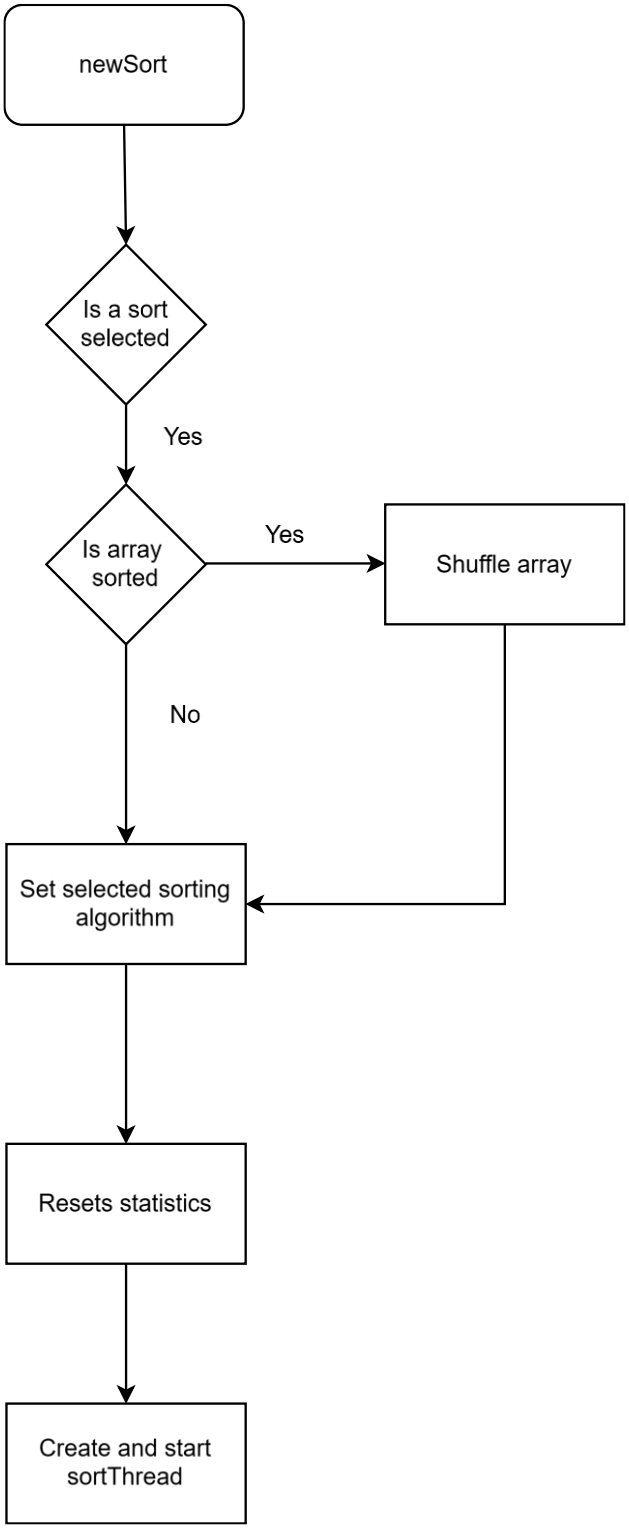
Program Start

Performed on opening the program. The main loop continues to run until the program is exited.



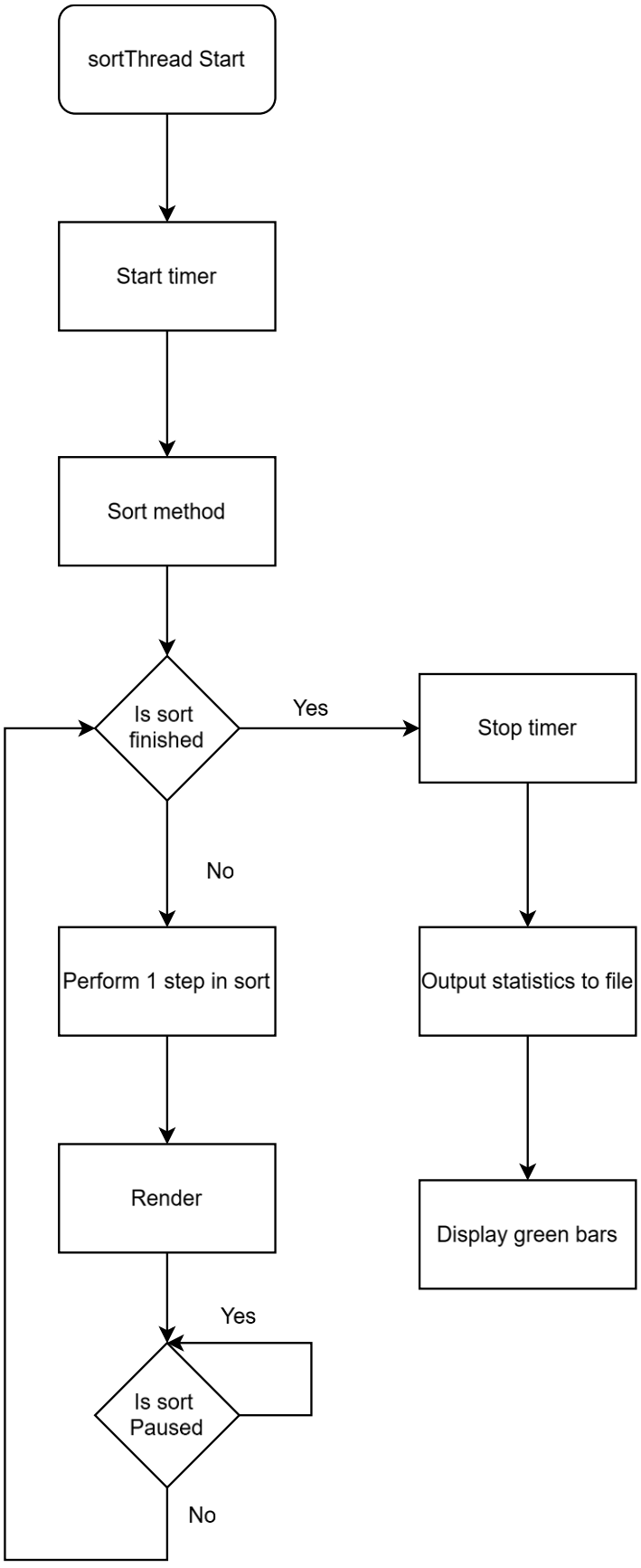
New Sort

Called whenever a new sorting algorithm is to be visualised.



SortThread Start

Called on starting of a new sortThread. Contains the logic for the sorting algorithm and runs until the sort is finished or reset.



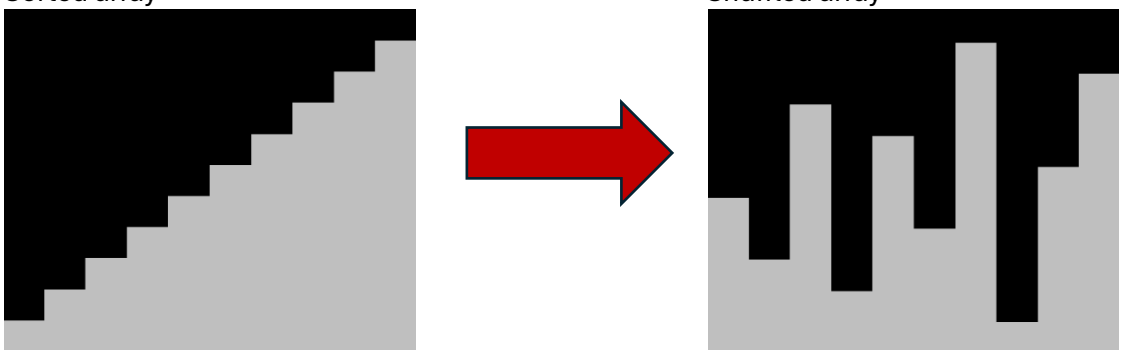
Visualisation

The user has the ability to toggle between two distinct visualisation methods at any point – bars and image with bars being the default option.

Bars

This is the default method used in the program as it provides the clearest visualisation of the sorting algorithm whilst showing the greater level of detail.
The bars visualisation method utilises tall rectangular bars to represent the data values in the array with the height of the bar being determined by each element's value – the greater the value, the taller the bar.

Sorted array



Shuffled array

The following formula calculates the height of each bar:

```
(maxHeight / MainController.arrayController.getLength()) * value
```

- `maxHeight` is the window height * 0.9
- `MainController.arrayController.getLength()` is the number of elements in the array
- `Value` is the value of the data to be represented

The default colour of each bar is light grey, with red and green being used to indicate any special values within the array corresponding to the current sorting algorithm e.g. the boundary positions in a merge sort. The data corresponding to these special values is stored in a separate arrayList – specialElements and is rendered over the top of the original bars rather than replacing the existing ones.

For the sortThread to interact with the specialElements arrayList (an object of the main thread) in a thread-safe manner, specialElements can only be altered through the following methods which each make use of a synchronized block on the specifically designated lock object.

```
public static void specialElementsAdd(IntColourPair pair) {
    synchronized (lock) {
        specialElements.add(pair);
    }
}

public static void specialElementsSet(int i, IntColourPair pair) {
    synchronized (lock) {
        specialElements.set(i, pair);
    }
}

public static void specialElementsRemove(int i) {
    synchronized (lock) {
        specialElements.remove(i);
    }
}

public static void specialElementsClear() {
    synchronized (lock) {
        specialElements.clear();
    }
}
```

Image

This is the secondary method used in the program as whilst it provides a less detailed view of the sorting algorithms it is potentially more intriguing to watch and allows for a more interactive user experience.

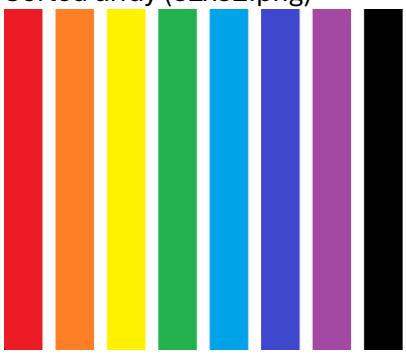
The image visualization method allows for the user to input an image file into the program and use the image's pixel positions / colours to represent the data values in the array with the smallest element being the bottom left pixel (0, 0) and the largest being the top right pixel (w, h) (where w is the width and h is the height of the image).

Each pixel in the image is linked to a value in the array using the following code

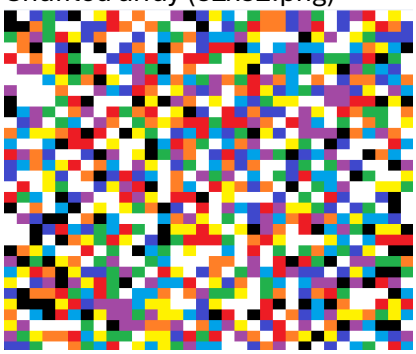
```
imageArray[(arrayController.getElement(i) - 1) %
(width)][(arrayController.getElement(i) - 1) / width]
```

When the array is shuffled this will also result in the pixel positions being randomised in the same manner, creating a jumbled-up version of the image which is slowly returned to its original state as the array is sorted.

Sorted array (32x32.png)



Shuffled array (32x32.png)



In the case no image file is currently loaded, the following message is instead displayed to the user indicating as such.

In order to 'move' the individual pixels of the image I cannot render the image in an ordinary fashion so instead each pixel is individually rendered as its own rectangle with the colour being set using a Color object of the corresponding pixel's ARGB value. Whilst the colour value of a pixel can be accessed directly from the image each time it is needed to be rendered, this is an expensive operation that will drastically slow down the program when repeated for every pixel of the image for every rendered frame. As a result it made more sense to access the colour value of each pixel directly from the image only once and store it in a 2D array of Color objects (RGBA values) to use as a lookup-table for an access time of O(1). This approach may lead to a slight loading time for larger images while the pixel array is generated, however due to the image size constraints set in place as to allow the sorting algorithms to complete in a reasonable length of time and to not cause graphical bugs with the bars visualisation method, this does not occur to a noticeable effect.

The following method handles generating the pixel lookup-table

```
private static Color[][] convertImage(BufferedImage image) {
    Color[][] pixels = new Color[image.getWidth()][image.getHeight()];
    for (int x = 0; x < image.getWidth(); x++) {
        for (int y = 0; y < image.getHeight(); y++) {
            int rgb = image.getRGB(x, y);
            // converts ARGB to RGBA
            rgb = (rgb & 0x00FFFFFF) << 8 | (rgb & 0xFF000000) >>> 24;
            pixels[x][image.getHeight() - y - 1] = new Color(rgb);
        }
    }
    return pixels;
}
```

Audio

An audible tone is to be played upon each comparison made, corresponding to the values of the data being compared. Initially I was going to accomplish this through utilizing a sinewave to generate the values for the tone and use the OpenAL library to play the tone through the system speakers. This approach had numerous drawbacks resulting in highly reduced performance, high system recourse usage and as a result offered subpar audio performance and an undesirable sound due to the large volume of requests being made by the program in quick succession overwhelming the system. The solution was to use the built-in midi library as this offered a solution to the majority of the issues encountered however due to the limited number of notes midi uses (128), it is possible for several elements to produce the same sound. The instrument I eventually selected for the midi audio was ‘Square Wave Lead’ (abbreviated as ‘Square’) as after numerous trials with other sounds, this was found to be easier on the ears than most while providing an ‘8-bit’ like quality.

File Handling

File Output

After a sorting algorithm has successfully finished the statistics relating to the sort are output into a csv file to be further analysed and compared by the user. I chose to use the csv file format (comma-separated values) as it is supported by all major spreadsheet software and is straightforward to implement due to only consisting of plain text formatted with commas.

The following method handles creating and writing to the csv file.

```
private void fileOutput() throws IOException {
        File file = new File("data.csv");
        boolean fileFlag = file.createNewFile();
        FileWriter writer = new FileWriter(file, true);
        if (fileFlag) {
            writer.write("Date,Sort,Elements,Time (Short),Time
(Raw),Comparisons,Swaps,Writes,Aux Writes");
        }
        String currentTime =
LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy/MM/dd
HH:mm:ss"));
        writer.write(("\n" + currentTime + "," + name + "," +
arrayController.export()));
        writer.close();
}
```

The layout of the csv file using data generated by the program.

Date	Sort	Elements	Time (Short)	Time (Raw)	Comparisons	Swaps	Writes	Aux Writes
06/01/2025 09:30:15	Bubble	571	358.4ms	358358300	161994	78537	157074	0
06/01/2025 09:30:57	Cocktail	151	36.11s	36110395200	8910	5767	11534	0
06/01/2025 09:31:27	Cocktail	107	16.51s	16509529900	4131	2582	5164	0
06/01/2025 09:31:33	Merge	2104	49.6ms	49638000	20614	0	23256	23256
06/01/2025 09:31:41	Quick	2104	30.9ms	30948600	32145	5939	11878	0

File Input

In order to input a file into the program, the file has to be dragged and dropped onto the program window. I chose this method of inputting the file as it requires no additional UI as to not clutter the existing interface and is natively supported by creating a new windowListener using the LWJGL3 backend. Rather than directly inputting the files data, this method only obtains the path of the file and passes it to the relevant method to be accessed in an appropriate manner – in this case the selectImage method of the Image class.

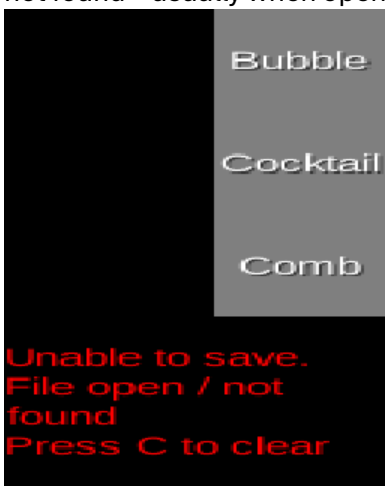
```
config.setWindowListener(new Lwjgl3WindowAdapter() {
    @Override
    public void filesDropped(String[] files) {
       try {
          Image.selectImage(files[0]);
       } catch (IOException e) {
          throw new RuntimeException(e);
       } catch (NullPointerException e) {
MainController.setErrorCode(MainController.Error.UnsupportedImage);
       }
    }
});
```

Exception Handling and Event Notification

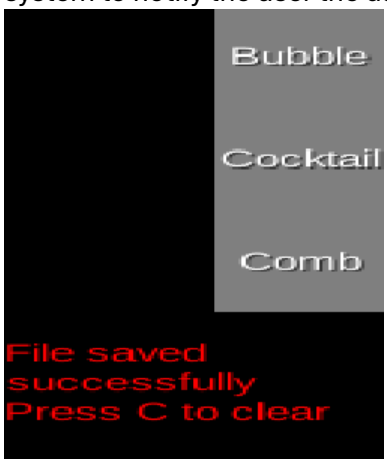
When an error occurs during the running of the program that would be expected to occur during normal operations the user is notified within the program itself, and it will continue to run without unexpected closure. In order to notify the user about non-error related but still key program events the same system is used. There are eight error codes in place:

The first error code is '*SortEnded*'. This occurs when the SortThread is terminated during a reset operation.

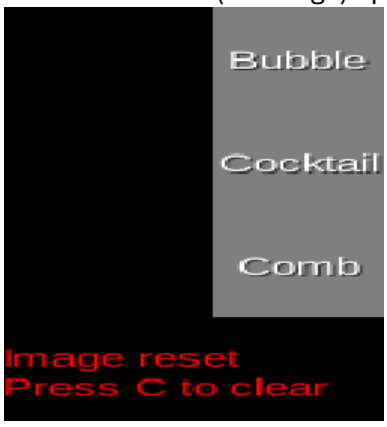
The second error code is '*FileBad*'. This occurs when the data.csv file is unable to be opened / not found – usually when opened by another program.



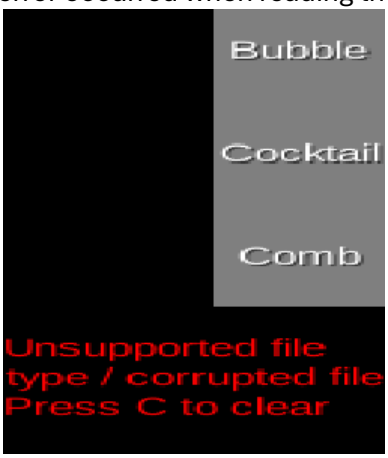
The third error code is '*FileGood*'. This is not caused by an error but instead uses the error system to notify the user the data.csv file has successfully been written to.



The fourth error code is '*ImageReset*'. This is not caused by an error but instead uses the error system to notify the user the image used for the image visualisation method has been reset to the default state (no image) upon the size slider being adjusted.

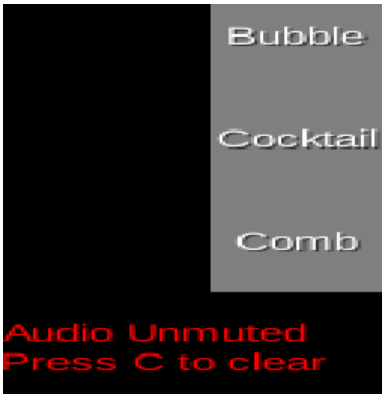


The fifth error code is '*UnsuportedImage*'. This occurs when the file dragged and dropped into the program to use for the image visualisation method is either of an unsupported format or an error occurred when reading the file's data.



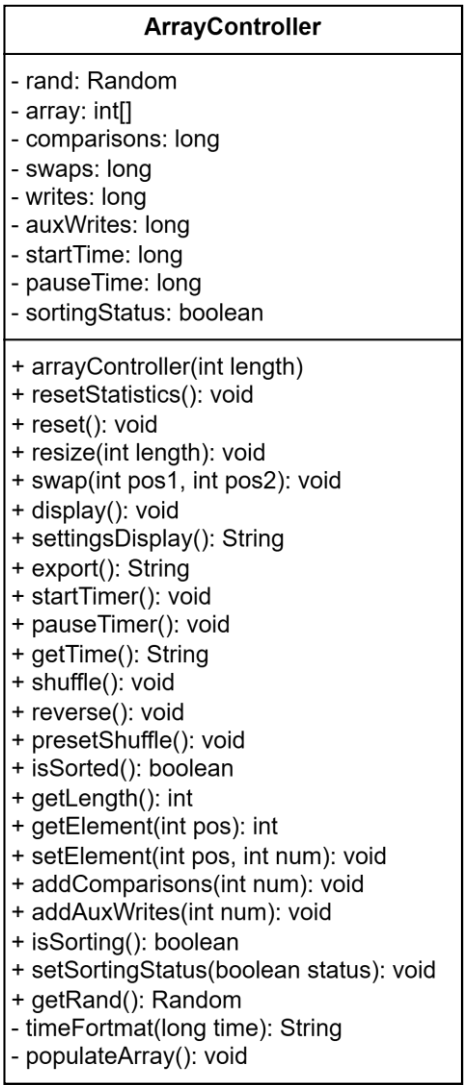
The sixth error code is '*BigImage*'. This is not caused by an error but instead uses the error system to notify the user the resolution of the image used for the image visualisation method is too large (total pixels is greater than the maximum array size).

The seventh and eighth error codes are '*Muted*' and '*UnMuted*'. These are not caused by an error but instead use the error system to notify the user if the audio mute state has changed.



Array Controller

The array controller class contains the array of data to be sorted as well as the methods to perform operations on the array. The class also keeps track of various statistics related to the current sort.



Testing

For the purposes of testing, I have added a preset shuffle option. This method still 'randomly' shuffles the array using the same algorithm as the standard shuffle (Fisher-Yates), however it instead uses a separate instance of the random class with a set seed allowing the outcome to be known in advance. In the case of an array of length 6, the resultant shuffle will be [6, 2, 4, 5, 3, 1].

Iterative Testing

Iterative testing was the main method of testing used whilst developing the program as this allowed me to get each main section / module of code working before moving onto the next. The main individually tested modules are:

- Each individual sorting algorithm
- UI Input
- Bar visualiser
- Image visualiser
- File output
- Audio

Whilst developing each module I would often develop the feature in a separate project to the main program to make sure I wouldn't break any existing features or for them to be in the away. This quickened the testing and implementation as the surrounding program would be better suited to performing quick and arbitrary tests.

Whenever an error / unexpected result would occur I would use a combination of print statements and the IntelliJ debugger to trace through the code and display the contents of the relevant variables. One example of this is when I was testing the timer function of my statistics panel and on certain devices the timer would display a seemingly arbitrary time value for a singular frame at the beginning of a sort.

By utilising the debugger to pause the code execution each time the timer value was updated, I was able to read the relevant variable contents (startTime and pauseTime) which let me deduce the startTime variable was not always reset at the start of each sort. I was able to use this new information to solve the issue by changing where in the code the timer function was started for a new sort.

Final Testing

Link to testing video: https://youtu.be/oi1Aw7ReSEc

Test No.	Relevant Objective(s)	Test Performed	Expected outcome	Test Passed (?)	Timestamp
1	1.a	Click on each button. Adjust each slider to limits. Expand each dropdown, scroll through with scroll wheel.	Name of each button to be output to the console. Displayed slider values to correctly update. Dropdowns should expand and scroll accordingly.	Pass	00:00 - 00:21
2	1.b 1.c 1.e.i	Use On-Screen keyboard to activate button hotkeys E, M, C, Back, Enter, P, Space, R. Arrow keys on slider and drop downs.	Each method to activate appropriately. Slider to increment/decrement and drop down to scroll. Statistics panel should update values to reflect the current sort. Notification should clear.	Pass	00:21 – 01:04
3	1.d.i.1	Shuffle array and run bubble sort on slowest speed.	The visual bars should change positions / heights to reflect the current data value in each array position. Some bars should change to a red colour to reflect currently selected values.	Pass	01:04 – 01:51
4	1.d.i.2.b	Switch to image rendering mode.	A message should fill the visualiser portion of the screen instructing the user to select an image.	Pass	01:51 – 02:03
5	1.d.i.2.a 2.d 4.b.i	Drag a compatible image onto the window and shuffle.	The image should appear in the visualiser portion of the screen and the array size will automatically adjust. The pixels will become jumbled when shuffled to reflect the current	Pass	02:03 – 02:39

			data value in each array position.		
6	1.c 4.b.ii	Change array size using slider.	The array should reset and update to the new size. The image should be removed and leave the placeholder message.	Pass	02:39 – 02:55
7	1.c 4.b.iii	Drag a large compatible image type onto the window.	An error message will appear instructing the image is too large.	Pass	02:55 – 03:14
8	1.c 4.b.iii	Drag an incompatible file type onto the window.	An error message will appear instructing the file type is unsupported or the file is corrupted.	Pass	03:14 – 03:33
9	2.a.i 2.a.ii 2.e	Using an array size of 10 and a speed delay of 10, select and start each sort. (Exclude bogo, bozo, selection and insertion sorts.)	At the start of each sort, the array should become shuffled, and the elements be put into the correct place in the array over time (sorted). The algorithm to do so will be that of the selected sorting algorithm.	Pass	03:33 – 04:30
9b	2.a.i 2.a.ii 2.e	Using an array size of 6, demonstrate selection and insertion sort. Output the array to the console every major step.	Each array output to console should be as expected.	Pass	04:30 – 07:27
10	2.a.iii 2.b 2.c 2.e	Using an array size of 7, shuffle the array using shuffle before starting bogo sort. Pause, step through and resume the sorting and then adjust the speed down to 0 until the sort is complete.	The array will be shuffled randomly before being sorted using the bogo sort algorithm. The sort should pause following the user input, stepped through (one sort step) with another input and finally resumed to full automated sorting. The delay between each sorting step will	Pass	07:27 – 08:22

			decrease to match that of the speed slider.		
11	2.b 2.c 2.e	Using an array size of 5 and speed delay of 1, shuffle the array using reverse before starting bozo sort.	The array will be reverse shuffled (largest element at the start, smallest at the end) before being sorted using bozo sort.	Pass	08:22 – 08:41
12	3	Run gnome sort with speed 100 and array size 20. Alter the pitch and volume sliders throughout. Mute and unmute the audio. Pause and step through several steps.	A tone will be generated whenever a comparison is made – the greater the value, the higher the note. The pitch and volume should change accordingly. The audio will mute and unmute accordingly. Audio will continue to work when stepped through.	Pass	08:41 – 09:39
13	4.a	Show project directory containing data.csv file and the file's contents. Delete the file and run a random sort showing the file being generated and the contents.	The data.csv file will initially contain the data / results of the previous sorts. The file will be shown to generate again once deleted containing one line of results.	Pass	09:39 – 10:52
14	4.a.ii	Close the data.csv file and run another random sort before reopening the file.	The data.csv file will contain the results of the first sort and on the next line the results of last sort.	Pass	10:52 – 11:24
15	4.a.iii	Without closing the data.csv file run another random sort.	An error message will appear alerting the user the file could not be saved.	Pass	11:24 – 11:45

Evaluation

I feel confident in that I have met all my objectives and exceeded my initial vision for the project. However, I feel while having an overall high level of completeness it does not have the level of polish I would have liked e.g. the text being larger than some of the buttons. Due to the incredibly large variety of unique sorting algorithms and various variations on others available, I feel the set I have included in the program is a bit lacklustre however due to time constraints I was unable to include a larger selection.

The image visualisation was a last-minute addition to the program, and I feel this comes across slightly in its implementation - being a bit more cumbersome to use then the basic bars and more restrictive in functionality. However, it is robust and helps act as a potential 'WOW' feature to help the program standout against other similar visualisers.

User Feedback

For user testing, I utilised the help of several of my peers who have an interest in teaching / tutoring computer science. They were given a copy of the program and a README file containing basic information / instructions to use and were asked to provide feedback.

User 1 - Jonathan

Jonathan commented on the overall redundancy of the program and how the interface was easy to navigate, they also felt the visualiser was "satisfying yet informative". However, they felt the 'Speed' slider was misleading and worked in reverse to how they thought. They felt this could be remedied by suggesting it would be better renamed to 'Delay'. The user also mentioned how they would like the addition of a button that would directly open the 'data.csv' file rather than having to manually navigate to it.

User 2 - Jamie

Jamie, a more technically inclined user, also commented on the ease of navigation and the satisfying nature of the visualiser. They felt as a potential future addition, the program could have an option for the user to create / code their own sorting algorithm to visualise. Whilst using the program, Jamie encountered a bug whereby: when sorting a large array with a speed of 0, when changing algorithm mid-sort, the array would sometimes not fully reset and more commonly the audio would continue playing a singular note.

Both users expressed a desire to visualise a greater number of sorting algorithms, especially other well-known algorithms e.g. bucket sort and radix sort. Both users also felt the audio can be slightly jarring at times on certain speeds.

Comments on User Feedback

Some of the feedback from the end users matched up with my own conclusions such as the need for a greater number of sorting algorithms in which I would have included if it were not for the limited time constraints. Overall, I am very happy both users found the UI to be easily navigated and did not find any flaws with the results of the included algorithms. As for Jonathan's first suggestion, I can see how there is ambiguity in the function of the speed slider and as such will include an explanation of what the values mean in the included README file. For Jonathon's other suggestion, the data.csv file should be created in the same directory the program is ran from and if I were to add another button to the settings panel, I feel it would become overly cluttered and messy. However, if I was to add a dedicated button in the future, the following code would work:

```
private static void openDirectory() throws IOException {
    File directory = new File("data.csv");
    if (directory.exists()) {
        Desktop.getDesktop().open(directory);
    }
}
```

Jamie's suggested feature of giving the user the ability to define their own sort to visualise, while a good idea, would prove to be difficult to implement without the user having to manually animate the sorting steps themselves which would go against the idea to begin with; not to mention the struggle to properly sanitise the input which would most likely have to take java code. As such this would go far beyond the objectives set out at the start of the project along with exceeding the scope of an A level project.

While I was able to reliably and fully replicate both issues raised by Jamie, I do not yet know what is causing them or how I would be able to fix them; although I suspect it has something to do with an interaction between the Main and the Sort threads. As this issue does not render the program inoperable and for the most part causes minimal impact to the users experience as it is an edge case, I have decided to not pursue the issue further due to the limited time constraints.

[END OF DOCUMENTATION]