An algorithm visualiser designed to demonstrate and display the process behind a variety of sorting algorithms and to play an audible ‘tone’ corresponding to the value of the current data being compared. 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.

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.

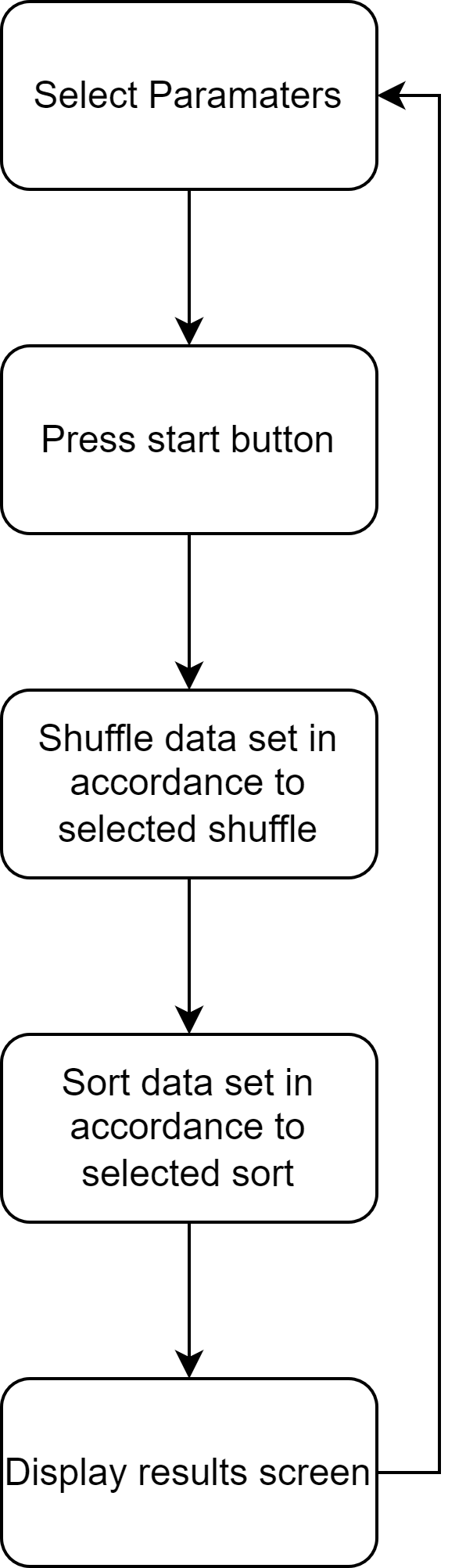
Objectives

1. Display the current state of the array, representing each element as a bar with the height corresponding to the value of the data element
2. Display a results screen containing statistics relating to the sort – sort type, number of elements in array, time taken for sort, number of comparisons made, number of swaps made
3. Create a control/settings panel containing:

* 6 buttons to start, pause, step through and reset the sort as well as mute the tones and select a random sort.
* 3 sliders to change the time between sorting steps; change the array size; change the pitch of the tones played
* Drop down menu to select the method used to shuffle the array
* Drop down menu to select the sorting algorithm used
* Information panel to display percentage of array sorted; time taken so far; comparisons made; swaps made

1. Implementing a variety of sorting algorithms

Overall program loop



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).

Rendering

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.

A screen shot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

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

User Interface

Implemented sorting algorithms

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() and has a space complexity of O 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

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(), the average time complexity is improved to O() where p is the number of gap increments. The space complexity remains O

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

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()

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(), 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() while having a space complexity of O(). 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