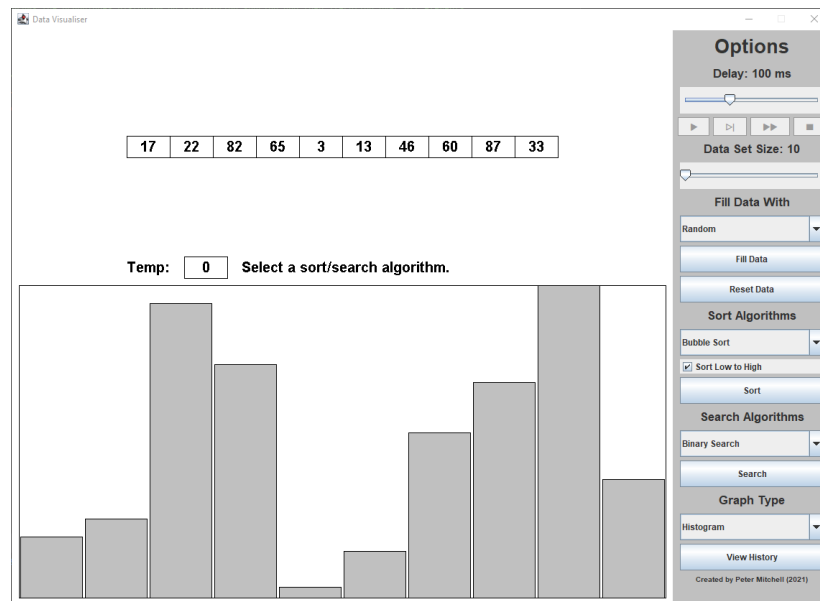# COMP1102/8702 – Practical Class 10

Aims and Objectives

As part of this practical you will be taking a look at writing sorting and searching algorithms. These are to be written using some provided functions as part of the application, allowing you to observe visually the algorithms stepping through each step of the process. You will begin by improving the already provided Bubble Sort to reduce the number of comparisons. Then with a basic understanding of how the integration with the application works you will be implementing both Selection Sort and Insertion Sort. You will be observing the difference in the number of swaps and comparisons used to sort using Bubble Sort (both the slow and your sped up version), Insertion Sort, and Selection Sort. For searching algorithms, you will be implementing Binary Search and comparing it against the provided Linear Search.

Background

The application you will be writing with puts some restrictions on how you implement the algorithms to make it possible for them to show the steps visually. You can find a video introduction to this practical at the link below. It shows a visual example of using all the controls and an explanation of how the initial code works for writing algorithms.

https://youtu.be/Co_E2fda4Nw

You can see a screen capture of the application above. You will see there are many options down the right side that allow you to modify what is shown or initiate algorithms. **See the end of this document for a more extensive text-based explanation of each element of the UI if you are unsure what any part is used for.**

To write the algorithms required for completion of this practical you will need to use some specific methods and data structures. The tables on the following page identify concisely specific constraints on how you need to access and modify the data.

Compare	compare(a, b) returns true if element a is less than element b. This is compared as a greater than operation automatically instead if the sort order is high to low. Where a and b are either both indexes of elements or DataElement objects (this is the type of data stored in the dataElements array).
	compare(1, 5) would perform the boolean operation: dataElements[1].getValue() < dataElements[5].getValue()
	compare(dataElements[1], dataElements[5]) would perform the same thing but referencing the elements directly. You will need to use this for the binary search algorithm. Calling this function is necessary to cause a delay where the elements that are compared will show in cyan.
Swap	swap(a,b) performs a swap operation that will show in yellow the elements that are being swapped in the sequence: 1. Temp = dataElements[a] 2. dataElements[a] = dataElements[b] 3. dataElements[b] = Temp
Length	dataElements.length will give you the number of elements stored in the dataElements array.
Get Element Value	dataElements[i].getValue() will give you the value of the element at position i. target.getValue() as part of the search algorithms only will give you the value of the target you are searching for.
Assign	assign(a, b) will assign the value of the DataElement "b" storing it into the DataElement "a". You will need to use this when writing insertion sort instead of swap because you are not performing a single swap, with instead many smaller assignments. It is essentially equivalent to a = b, but it is only copying the values from b to a and not moving the DataElement container. assign(dataset.getTempValueElement(), dataElements[i]) would assign the value in position i of dataElements into the temporary value. As part of this step, it will highlight the pair of elements yellow. You will also need to consider manually the increase to the number of swaps by using swaps++; and to show the updated number of swaps you can call updateStatusText().
Log Search Steps	There are two methods available for logging during the search algorithms only. highlightAndLog(i) should be used in the linear search to highlight element i right before an equals comparison against the target. highlightAndLog(low, high, mid) should be used for binary search to represent visually the search range right before the equals operation to check for the target.
	Once a search has finished you should store the position where it was found or -1 into the variable "foundIndex" as shown in the LinearSearch code. Then to show the end result based on this foundIndex you can set the label by calling: dataset.setStatusLabel(getSearchSummaryString()); If you are interested in seeing what this function or any other does, you can use Ctrl+B to jump to the details of the function.

Getting Started

Start IntelliJ and open the project "Practical10" (download it from FLO).

If you are running a smaller screen resolution and need to shrink the application, you can do this by opening Main.java and change false in the main method for launchWithAllAlgorithms(false); to true.

Run the application by using Main.java to observe how it currently functions with the initial version of the algorithms. By default, the application will fill with a random set of 10 numbers. You can change this if you want to see a larger dataset by modifying the settings under "Fill Data With", then clicking "Fill Data". The Reset Data button will revert the order of elements to the last time you pressed "Fill Data".

To sort you can select any option from the dropdown under "Sort Algorithms", choose the direction to sort in with the checkbox and click "Sort". You will observe cyan-coloured backgrounds on the number grid at the top and histogram below when a comparison happens showing the two numbers being compared. When a swap occurs yellow will be shown to indicate the transition occurring. Every swap involves swapping one element into the "Temp:" seen in the middle left of the screen, then moving one element to overwrite another, followed by copying the "Temp:" value into the other position.

Task 1 (Checkpoint 47)

1. Observe when running the application that the current implementation of Bubble Sort traverses the entire list during each step of the sort.
   This is not necessary, since after the first traversal the smallest item will be at the end of the list. After the second traversal, the second smallest item will be in its correct position (2nd last) and so on.
2. Open BubbleSort.java (leave BubbleSortSlow.java unchanged) located in the SortingAlgorithms package and modify the provided performAlgorithm() method containing the implementation of Bubble Sort. You should make changes to prevent unnecessary comparisons by making changes to the for loop to make it terminate earlier on each successive step of the algorithm.
3. Evaluate your Bubble Sort comparing it to Bubble Sort Slow by running with the same data using a fixed data set. You can either use a random data set and with pressing "Reset Data" in between observing the number of swaps and comparisons, or with custom data. You should see the same number of swaps for both, but the Bubble Sort should have less comparisons than Bubble Sort Slow. You can verify this using the following steps with custom data.
   a. Set the Data Set Size to "10" and select "Custom" under Fill Data With.
   b. Press "Fill Data" and enter the following data into the dialog:
      17 22 82 65 3 13 46 60 87 33
   c. Select Bubble Sort from the Sort Algorithms dropdown, make sure "Sort Low to High" is ticked, and press "Sort".
   d. You should see the result shown in the middle of the application showing:
      Swaps: 18 Comparisons: 39
   e. Press "Reset Data" and you should see the data revert to the original data order.
   f. Select Bubble Sort Slow from the Sort Algorithms dropdown, and press "Sort".
   g. You should see the result shown in the middle of the application showing:
      Swaps: 18 Comparisons 54

Have the program source code and output marked by a demonstrator.

Task 2 (Checkpoint 48)

1. Create a new Java class file in the SortingAlgorithms package and call it SelectionSort.
2. Add an import statement before the class definition line to import CoreApplication.*; this will import the necessary classes used as part of this algorithm.
3. Modify the class definition line by extending SortAlgorithm.
4. Add a constructor as shown below.

```
public SelectionSort(DataVisualiserApplication main) {
       super("Selection Sort", main);
}
```

This code will inject a reference to the application into your algorithm, and the call to super passes the information along including a String that in this case makes "Selection Sort" appear as the name in the Sort Algorithms dropdown.

5. Declare the required performAlgorithm method. You can also auto fill this by selecting the place where a red underline will appear on the class line and using Alt+Enter (on Windows, or Command + Enter on Mac) and then selecting "Implement Missing Methods".
6. Write code in the performAlgorithm method to complete the Selection Sort by following the below pseudo code steps.

> For every indexA in the dataElements array:
>
> > Initialise a variable with the value indexA to hold the index of the minimum value.
> >
> > For every indexB from after indexA to the end of the array :
> >
> > > If the element at position indexB is less than the value at the current minimum :
> > >
> > > > Set the minimum index to indexB.
> >
> > If the minimum index is not equal to indexA:
> >
> > > Swap the elements at indexA and the minimum index.

Make sure you use the compare and swap functions when doing the less than comparison and swap respectively as described in the Background section at the start of this document.

As you write the code try and understand what each step is accomplishing and why it is important to the completion of the algorithm.

7. Evaluate the number of comparisons and swaps, compare these to the data you observed in the first task. Using the same data (17 22 82 65 3 13 46 60 87 33) you should see the result: **Swaps: 8 Comparisons: 45**

Have the program source code and output marked by a demonstrator.

Task 3 (Checkpoint 49)

1. Repeat step 1 to 5 of Task 2 to create a class InsertionSort that has an appropriate constructor where "Insertion Sort" is the String passed via the call to super as shown below.

```
public InsertionSort(DataVisualiserApplication main) {
      super("Insertion Sort", main);
}
```

2. Write code in the performAlgorithm method to complete Insertion Sort by following the below pseudo code steps.

> For indexA from 1 to the end of the dataElements array:
> > Set the value of Temp to the value at position indexA **(Use assign())**.
> > Set indexB to the position stored in indexA.
> > Loop while indexB is larger than 0 and comparing Temp to the value at [indexB-1] is true:
> > > Set the value of the element at indexB to the element at [indexB-1] **(Use assign())**.
> > > Increase the number of swaps (using swaps++).
> > > Call updateStatusText() to update the number shown for swaps.
> > > Decrement indexB by 1.
> > Set the value of the element at indexB to the value stored in Temp **(Use assign())**.
> > Increase the number of swaps (using swaps++).
> > Call updateStatusText() to update the number shown for swaps.

Make sure to refer to the Background section for information about using the assign function. You will also need to use dataset.getTempValueElement() to get a reference to the temporary value.

3. Evaluate the number of comparisons and swaps, compare these to the data you observed in the first and second task. Using the same data (17 22 82 65 3 13 46 60 87 33) you should see the result:
    Swaps: 27 Comparisons: 26

-------------------- **Checkpoint 49** -------------------

Have the program source code and output marked by a demonstrator.

Task 4 (Checkpoint 50)

1. Start by testing the Linear Search. You can use the data from the previous task. Make sure it is sorted (although for Linear Search this is not necessary it will be when you test Binary Search). Select Linear Search if it is not selected under Search Algorithms and press "Search". A dialog will appear, and you can enter any number to search for. For example, if you enter 60 you should see the following sequence of events happen with cyan colour showing as the search moves from left to right.

   Step 1: Checking [0]=3
   Step 2: Checking [1]=13
   Step 3: Checking [2]=17
   Step 4: Checking [3]=22
   Step 5: Checking [4]=33
   Step 6: Checking [5]=46
   Step 7: Checking [6]=60
   Target 60 was found at [6]=60

2. You will see each of the steps show up as the animation happens. You can view a log of the history showing all the steps performed as part of the search by clicking on "View History". This will display a dialog showing a history of all sorts and searches you have performed.

3. Similar to the steps you completed to create new Sort Algorithms, now you should create a new Java class in the SearchingAlgorithms package folder and call the file BinarySearch.

4. You will still need to import CoreApplication.*, but instead of extending SortAlgorithm you should extend from SearchAlgorithm. Similar to the other constructors you have written, it should be written as the following.

```
public BinarySearch(DataVisualiserApplication main) {
      super("Binary Search", main);
      sleepMultiplier = 15;
}
```

You will notice the addition of the sleepMultiplier variable. Search algorithms require significantly less steps compared to performing sort operations. To slow them down enough for optimal viewing this multiplier changes the impact of each tick on the delay slider. 100ms in this case with 15 would actually become 1500ms.

5. Perform the same step as before to automatically fill in the missing performAlgorithm method. Inside this method you will only need to write two lines that are functionally the same as the Linear Search version of performAlgorithm, with the only change to call a binarySearch() method that you define yourself.

```
foundIndex = binarySearch();
dataset.setStatusLabel(getSearchSummaryString());
```

6. Declare a private method called binarySearch that returns an int type result with no parameters. Inside the method you should follow the steps outlined below in pseudo code.

> Set low to 0.
>
> Set high to the number of elements - 1.
>
> Loop while low is less than or equal to high:
>
> > Set mid to the average of low and high.
> >
> > Call highlightAndLog(low, high, mid); to log the step.
> >
> > If the target is equal to the value of the element at position mid:
> >
> > > Return mid as the position where it was found.
> >
> > If comparing the element at position mid and the target is true:
> >
> > > Set low to the value mid + 1.
> >
> > Else
> >
> > > Set high to the value mid - 1.
>
> Return -1 if the loop terminated.

As with the previous tasks refer back to the Background if you want additional details about what you are implementing. The target you are searching for can be referred to with target.getValue() and you can pass target as a parameter into compare. The highlightAndLog method will highly the low and high indexes with cyan and the mid index with yellow.

7. Evaluate that your search works properly by selecting "Binary Search" from the dropdown and entering 60 again. You should see the steps as seen below:
   Step 1: Low [0]=3 High [9]=87 Mid [4]=33
   Step 2: Low [5]=46 High [9]=87 Mid [7]=65
   Step 3: Low [5]=46 High [6]=60 Mid [5]=46
   Step 4: Low [6]=60 High [6]=60 Mid [6]=60
   Target 60 was found at [6]=60

Breaking down these steps they make sense because in step 1 the low starts at the default index 0 and high starts at index 9. (0+9)/2 = 4 because it is an integer division. 33 is not equal to the target of 60, 33 is however less than the target so low is set to 4 + 1 = 5.

In step 2, the mid is set to (5+9)/2 = 7. Index 7 has the value 65 in it, 65 is not equal to the target 60, and is not less than the target. Therefore, it defaults to the else statement and high is set to 7 − 1 = 6. In step 3, the mid is set to (5 + 6)/2 = 5. The element at position 5 is 46. 46 is not equal to 60, but it is less than the target. Therefore, the low is set to 5 + 1 = 6. Finally in step 4, the mid is set to (6 + 6)/2 = 6. The element at position 6 is equal to 60, therefore 6 is returned from the method.

You should test your search functionality for some other situations, like for example what happens when there is no matching value to be found. And consider when you would use the binary search and linear search as part of different situations.

-------------------- **Checkpoint 50** --------------------

Have the program source code and output marked by a demonstrator.
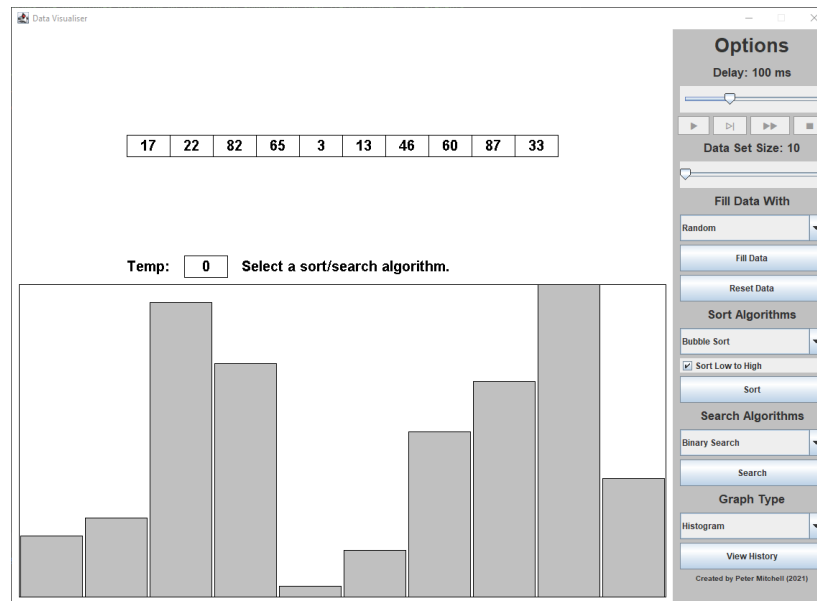
Task 5 (Extension Practice)

Going beyond the tasks there are many things you could choose to expand on. You can choose to explore as many of these extension tasks as you wish for your own interest.

- Try and implement additional Sort Algorithms. The following options are all algorithms that can be implemented within the constraints of this application. You can find implementation details of the algorithms at the provided links. Of these the only sensible one you are likely to use is Quick Sort, the rest are more novelty type sorts.
  - Binary Insertion Sort: https://www.geeksforgeeks.org/binary-insertion-sort/
  - Cocktail Sort: https://www.geeksforgeeks.org/cocktail-sort/
  - Comb Sort: https://www.geeksforgeeks.org/comb-sort/
  - Gnome Sort: https://www.geeksforgeeks.org/gnome-sort-a-stupid-one/
  - Pancake Sort: https://www.geeksforgeeks.org/pancake-sorting/
  - Quick Sort: https://www.geeksforgeeks.org/quick-sort/
- Try and implement additional Search Algorithms. The following are options that you can implement as less used search algorithms.
  - Jump Search: https://www.geeksforgeeks.org/jump-search/
  - Interpolation Search: https://www.geeksforgeeks.org/interpolation-search/
- Collect multiple samples of data for the number of comparisons/swaps for different sizes of data sets or different outliers (e.g., already sorted, or reverse sorted) and observe the impact. You could graph the results and consider when different algorithms would be useful.
- Break down the application that has been provided to you and explore the implementation of individual methods/classes, or how everything interacts. Some of the topics you can investigate include:
  - How the GUI is put together using most significantly the ButtonPanel, DataVisualiserApplication, and VisualiserPanel classes.
  - How the graphs are drawn by investigating the integration with the VisualiserPanel class using the generic Graph class and all the specific types defined in the GraphTypes package.
  - How the algorithms are integrated into the application. This could be looking at many smaller parts, including how the classes are loaded in dynamically using Reflection as part of the ClassCollectorUtility and ButtonPanel. You could also look at the process of what happens from selecting an algorithm from the dropdown in ButtonPanel and how that gets to the VisualiserPanel. Then from the VisualiserPanel how the multithreading is started to handle everything. As part of this you could look at how the AsyncAlgorithm base class has been extended for the SortAlgorithm and SearchAlgorithm classes that are then used for the specific algorithms.

- Look into the ReportGenerator class and investigate how reports are generated. This functionality is triggered by opening View History and then clicking Generate Report.
- Look at expanding or improving some part of the code base. There could be something you find that is a bug, or code that could be rewritten in a better way. Understanding the code and working out why things have been written in a specific way may expand the ideas you have for how to develop your own applications. You could even use parts of this application as inspiration to develop your own utility.

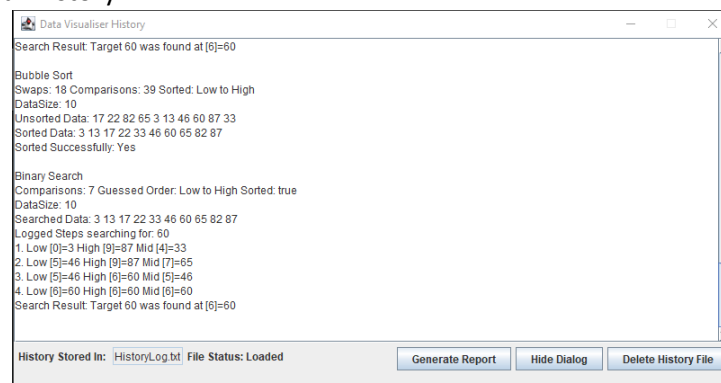# Extended Explanation of Application

Interface Breakdown

On the left side of the screen, you can see there are three sections:

- In the top section is a list of numbers, these represent the actual values of the data that you are performing sorts on.
- In the middle you can see a line with text. The cell marked as Temp is the cell used to represent a temporary variable during swaps, and during searches it represents the target to search for. The text more centrally will change to show status information while algorithms are running. During sorting it will show swaps and comparisons. During searches it will show the steps performed during the sort.
- At the bottom you can see a histogram. This is a graph representing a visual matching the data shown at the top.
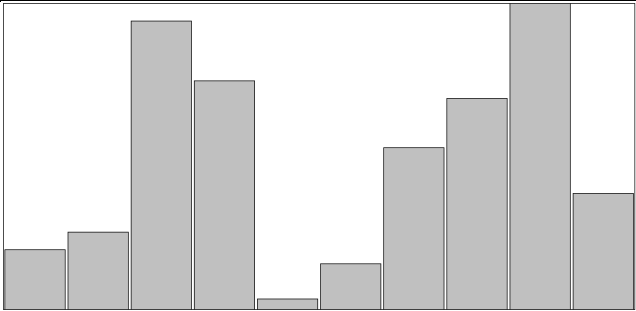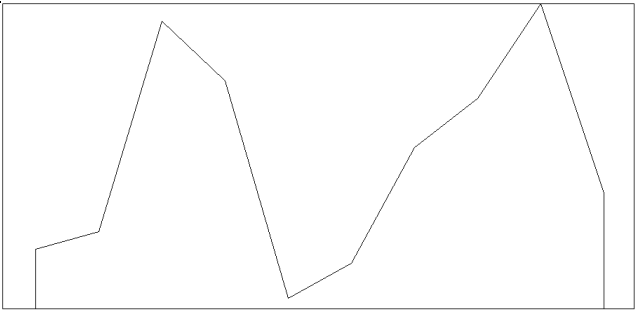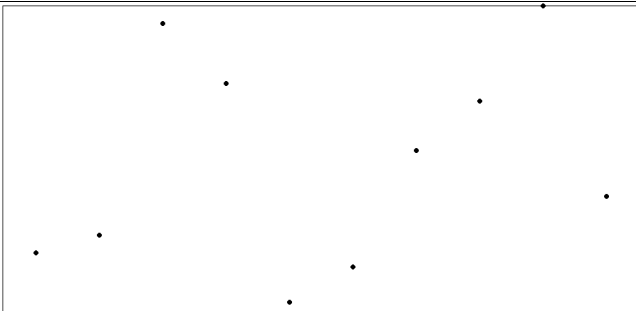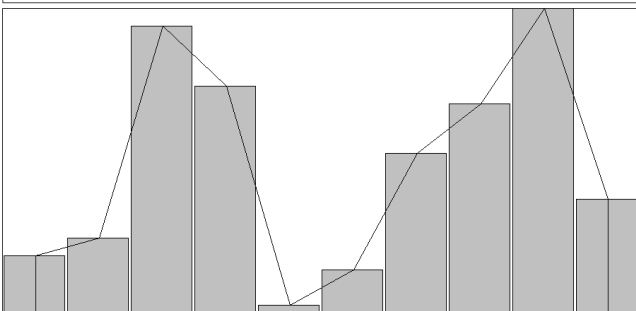
On the right side of the screen, you will see a lot of controls. The following lists in order their function.

- Delay: Defaults to 100ms with a slider going from 1ms to 300ms and changes the time between each animated swap/comparison. A lower number will make the algorithm go faster.
- The four buttons below the Delay control are:
    - Play/Pause: When running you can pause running at any point and click it again to resume.

- o Step Through: Will resume the algorithm and automatically pause at the next change in visual highlighting. You can repeatedly press this button to move through one step at a time.
- o Fast Forward: Removes all delays and completes the algorithm as quickly as possible.
- o Stop: If you have made a mistake causing an infinite loop in your algorithm or there is some other reason you want to terminate the algorithm you can click this to cancel the currently running algorithm and prevent any logging for it.
- Data Set Size: Defaults to 10 and can be between 10 and 100 moving in increments of 10. This will change the number of numbers that appear in the data set on the left that are added when you press "Fill Data".
- Fill Data With Dropdown: This dropdown contains four options Random (default), Sorted, Sorted Reverse, and Custom. You will mostly be using Random, but Custom will allow you to enter specific values into a popup window up to the number in Data Set Size.
- Fill Data: Pressing the button will use the specified data set size and the selection from the fill data with dropdown to populate the data on the left side of the screen.
- Reset Data: This will reset the order of data back to the last time you pressed the Fill Data button.
- Sort Algorithms Dropdown: This will show all algorithms that you have inside the SortingAlgorithms package folder as long as they satisfy the requirements.
- Sort Low to High Checkbox: This will toggle between Low to High and High to Low. This controls the sort order for how the numbers will be sorted left to right.
- Sort: Will use the selected sort algorithm and sort direction to call performAlgorithm() in the specified sort algorithm.
- Search Algorithms Dropdown: This will show all algorithms in the SearchingAlgorithms package folder with the correct requirements.
- Search: When you press search it will prompt you to enter a number to search for. After specifying a valid search target the selected algorithm is run after the program guesses the direction data is sorted in.
- Graph Type: This dropdown has a variety of alternate graphs that can be shown instead of the Histogram (default). You can find examples of the different options shown over the page. The dots and histogram bars will change colour while animating. The line will not change colour at all.
- View History will display a dialog as shown below that shows all the history of searches and sorts that you have performed. You can clear the history by using "Delete History File", hide the dialog with "Hide Dialog", or "Generate Report" will generate a summary of data based on all entries that appear in your history.

Graph Types

Histogram	
Line Graph	
Dot Plot	
Histogram + Line Graph	
Dot Plot + Line Graph	