2017

Visualizing Sorting Algorithms

Brian Faria
Rhode Island College, bfaria1722@gmail.com

VISUALIZING SORTING ALGORITHMS

By

Brian J. Faria

An Honors Project Submitted in Partial Fulfillment

Of the Requirements for Honors in

The Department of Mathematics and Computer Science

Faculty of Arts and Sciences

Rhode Island College

2017

Abstract

This paper discusses a study performed on animating sorting algorithms as a learning aid for classroom instruction. A web-based animation tool was created to visualize four common sorting algorithms: Selection Sort, Bubble Sort, Insertion Sort, and Merge Sort. The animation tool would represent data as a bar-graph and after selecting a data-ordering and algorithm, the user can run an automated animation or step through it at their own pace. Afterwards, a study was conducted with a voluntary student population at Rhode Island College who were in the process of learning algorithms in their Computer Science curriculum. The study consisted of a demonstration and survey that asked the students questions that may show improvement when understanding algorithms. The results and responses are recorded and analyzed in this paper with respect to previous studies.

Contents

List of Figures

1 Introduction

How do you work out a problem? The problem itself doesn't need to be anything overly complex, such as trying to replace a broken headlight in your car (although nowadays, manufacturers are trying the patience of the community with their increasingly abstract, space-age designs). The point is how to *attack* the problem. Do you perform research, such as looking through your car's manual for step-by-step instructions, or is your first instinct to find someone who knows how to do it (whether they are right next to you or in an online video)? My instinct is the latter, as I am a visual learner and am adept to picking up concepts by *seeing* it done, rather than *reading* about it. For example, when I was learning about sorting algorithms while pursuing my Computer Science degree, I found that *seeing* the data move to its correct position under the constraints of an algorithm was much easier to follow than tracing the code by hand.

That led to the inspiration of this paper, which describes a web-based tool I created that animates how sorting algorithms modify and organize a set of data. If you need to organize a list of people by their age in ascending order, for example, there are multiple algorithms that can perform the task. I visualized four of the well-known ones by representing numerical data as a histogram.[1] Each number is illustrated as a bar and has a different height

[1] I also had plans to visualize Quick Sort and Radix Sort. Due to refactoring, I did not have time to implement Quick Sort. I also realized that a histogram would not be the best approach for Radix Sort because it works with the individual digits of a number than the number as an entity.

based on its value. This would make it distinct from the rest of the data as it is being shifted by the algorithm from its original, unordered position to its final ordered position. The four algorithms are: Selection Sort, Bubble Sort, Insertion Sort, and Merge Sort.[2]

Keeping in line with the example of sorting people by age, let's pretend that you have printed the age of each person on a separate index card. One way to go about organizing the cards is to first find the smallest age in the pile and bring it to the front. Then, find the next smallest and place it behind the already ordered first age. Eventually, you will end up with a pile of index cards that list the ages in ascending order. This method is exactly how Selection Sort works, where to sort a set of data, you select the smallest first, and then the next smallest and the next smallest. This algorithm is not very difficult to understand by word of mouth, but more abstract sorting algorithms, such as how Quick Sort requires moving data around a pivot point, may not be intuitive to read through.

I wanted the animation to be web-based to appeal to a wide spectrum of people using different technology media. This way, the user would not need to worry about installing special software or trying to organize configurations to use the tool. The webpage is coded with HTML5 (Hypertext Markup

[2]I modified the implementation of Merge Sort, where it behaves like a hybrid of Merge Sort and Insertion Sort. Merge Sort runs by creating a copy of the data and I needed to find a way to visualize this with only one data set. I got around this by replacing the code responsible for merging the sorted halves with a call to Insertion sort. This clearly visualizes the values moving in-place where they migrate from right to left in each partition, but it does not reflect the true nature of Merge Sort (which would also merge the data faster if not using Insertion Sort).

Language, Version 5), JavaScript, and CSS (Cascading Style Sheets). The physical elements of the webpage (buttons and layout) are coded with very minimal HTML5 code. The next biggest contributor would be the CSS code, which is responsible for the appearance and behavior of the buttons and text. Finally, the rest is devoted to JavaScript, responsible for the histogram generation, movement, algorithm design, and sound. All the buttons refer to designated parts of the JavaScript code to perform the task.

I did have some experience creating web pages from previous internships, but not nearly enough to handle this task effectively. Most of the time, I was coding in a simple text editor and was hunting for syntactical errors by hand. Toward the end of the project was when I found out that browsers have a built-in developer tab that highlights syntactical errors. Before this, when my code broke, the histogram would just disappear and I had no idea why.

Chapter Two of this paper discusses work and conclusions made by other similar studies, and also provides references to types of more recent video animation. Chapter Three is a top-level view of how the animation works and how a user would perform the sorts. Chapter Four is an in-depth view of the underlying structure of the code and how it relates to the final animation. Chapter Five describes my methods and results of user testing and Chapter Six concludes my findings with a tangent of how this would relate to future work. All the code is in the appendix and can be made into a single HTML

file, so if you want to use it, go for it![3]

[3]The files for the sounds and images for the MUTE button are not referenced, so the single HTML file that you could create will not be able play the sounds or show the MUTE button as originally intended. However, the sorting animation should work like the original.

2 Related Work

Over the years, there have been many studies and papers on the use of sorting algorithms as visual aids. Some are comprehensive views on how to create animations and perform statistical analysis, and others focus on different techniques aimed for increased understanding of a similar animation. By similar, I mean that between two studies, the animation used may be similar, but the difference in analysis was geared to how the algorithm was used. I would like to say upfront that John T. Stasko was a prominent figure in researching animations and most of my sources include papers in which he contributed.

The paper "Algorithm Animation," by A. Kerren and J. Stasko [3] is a step-by-step guide to analyzing the environment, means, and available coding methods to use a sorting animation. Many different types of software are listed to be used for animation, one of which was BALSA, which pioneered the *interesting event* approach [3, p. 3]. BALSA was created by Marc Brown and the *interesting event* approach was coined to determine what part of the sorting algorithm was significant to both clearly see and understand how the algorithm performs. Additional software that followed this principle were Zeus, CAT, Tango, and Samba, to name a few [3, p. 2]. These developments prove the continuing interest in creating animation tools. For example, the *interesting event* approach I took focused on animating the movement of data as a visual description of the algorithm.

For a direct analysis of how students respond to sorting animation, the paper "Do Algorithm Animations Assist Learning? An Empirical Study and Analysis" [6] provides an in-depth view. A post-test study was used to gather information on comparing the results of students who only had textbook resources to those who had a textbook as well as an animation for assistance. The post-test was the same for each group of students, which covered a comprehensive view of the topic.

The study found no clear support that an animation would help students with the material significantly. The group of students that had the animation tool in addition to the textbook material averaged correctly answering two more questions than the control group [6]. The paper concluded that visualizing algorithms sounds good, but may not achieve the desired results when implemented.

Kerren and Stasko's results do not apply completely to the project described in this thesis, because their paper focuses on the learnability of data structures using animation (heaps and trees) and not the algorithms themselves. In addition, the paper notes some limitations to its applicability that I have not undertaken. For example, some factors that the study addressed were the availability of appropriate students, a tradeoff of fairness regarding selective academic success, and procedure-specific questions for post-test design to promote quantitative results.

Their study had a total of twenty student volunteers, divided into two groups of ten for the control without the animation and the one that would

have exposure to it. I was surprised to see the small sample size, but realized later that this was a blessing, given the fact that I was only able to produce just over half the sample size with thirteen volunteers. They summarize this difficulty perfectly, stating, "Pragmatically, it is challenging to assemble the appropriate ingredients for an empirical study. An algorithm animation environment must be available, and most importantly, a group of subjects who are at an appropriate point in their educational careers must be available" [6, p. 61].

In order to achieve quantitative results, however, the control versus animation exposed groups were necessary. For example, the study's post-test design included questions to promote general understanding, analytical thinking, and procedural knowledge [6, p. 63]. Therefore, some of the questions could easily be answered with a textbook resource, but the procedural knowledge can directly relate to the animation as it visualizes the process. The results outlined in this paper are not as cut-and-dry, where the animation was introduced after the students already learned the material in class. The goal was to prove if prompting the students to recall what they learned, and applying it by analyzing the effectiveness of the algorithm animation tool, would be better at reinforcing the concepts. This is still a one-sided study, however, as all students were exposed to the tool in a quick introductory lecture that reinforced basic concepts before they could experiment with it. Therefore, even though the results lean in favor of Kerren and Stasko's observation that the students believed the animation tool did help their understanding, the

results are not comparable to this study as there is no control group to give a solid foundation. In the future, I would conduct this study differently where I would create a control versus exposure group analysis, which could then be more comparable to their results and possible provide more ground for support in animation tools as learning aids.

Another article I found is Stasko's "Using Student-Built Algorithm Animations as Learning Aids." [5] It initially stated the same fault found in the previous article, where showing an animation does not promote understanding as much as desired. In an interesting twist, however, students were given assignments to build the animations themselves, rather than use some already made as a means of better understanding. The students were introduced to the visual programming tool Samba. After a few introductory assignments to get the knack for using the software, they were given the option to animate an algorithm, keeping in mind that a person who did not know the material could understand it. The results showed positive feedback and overall better understanding of how the algorithms worked. Also, some students found that they had misconceptions about the material when implementing them in an animation.

Therefore, Stasko's paper pushes learning by actively seeking student involvement in the animation, which I have strived to do in this study to a certain extent. The difference is that I did not encourage the students to make their own animation algorithms, and instead asked them to answer questions geared towards previous learning and how the animation help/disproves it,

if at all.

If there weren't enough twists already, Andrés Moreno took this a step further by introducing conflictive animations [4]. The main idea is that students would learn the material better if the instructor intentionally fed them the wrong answer, allowing the students a chance to detect it. This would demonstrate focus on the students' part to learn the material, along with the ability to apply it in "correcting" the instructor. This concept is believed to not help the students learn the material, per se, but instead allow the students to recognize any misconceptions they may have about the material, which could lead to problems later.

Now, by taking this confliction concept and applying it to sorting animations, you get the thesis statement of this article. If a sorting animation was programed to intentionally fail or execute a part incorrectly, it gives the student a chance to try and detect it. Overall, this would improve the understanding of the material because the process of correcting the animation demonstrates a decent knowledge of how it works. This would also offer corrective criticism, which, theoretically, would motivate the student to learn.

Finally, I bumped into some online videos of algorithm animations. The first I found introduced a new concept where the animations included sound [1]. Timo Bingmann created a blog where he describes his animation experiments using sound to illustrate the difference of each sorting algorithm audibly. A sound would be played after every outer-loop iteration of the animation until it finished. The result would be hearing the animation sequence

gravitate toward playing a glissando from a low note to a high note when the animation completed. Big data sets are used and since the animation executes fast to compensate for this, the effect is continuous jumbled sound for a while that starts to develop a pattern before finally finishing. There is a link in the references to Bingmann's blog that has the videos.

Another set of videos was recommended to me by William Ebeling, a RIC alum. The algorithm visualization in these videos incorporated dance [2]. The videos are on YouTube published by the handle *AlgoRythmics*. Instead of animating the algorithms using graphics, a group of Hungarian dancers represented the data by wearing numbers on their chests. In contrast to Bingmann's approach, the music the dancers perform to is folk music instead of computer generated tones. The folk music played in the background as the group would stand in a line (or multiple lines depending on the sorting algorithm) and then dance to show comparisons and swaps of data. The comparisons are represented as two dancers from the group facing each other temporarily as their data values were being compared. If they needed to swap, they would perform a quick jig to each other's respective position. One feature I found interesting is that they sometimes use a hat to show which index in the data is the pivot or the one being compared to others.

Overall, there has been substantial research in different forms to support the idea of algorithm animations for education/learning aids. Even though some of the results showed minimal improvement with the introduction of an animation, there have still been recent attempts to spin the material in

new ways that may be helpful to someone struggling with algorithms (like videos).

3 Design

3.1 The User-Interface

Even though the underlying back-end code went through a drastic refactor midway through the implementation, the overall design and layout of the user-interface components has remained the same. The interface has twelve components: a canvas area, ten control buttons, and a volume on/off toggle button. Below in Figure 1 is a picture of the web page after it loads.

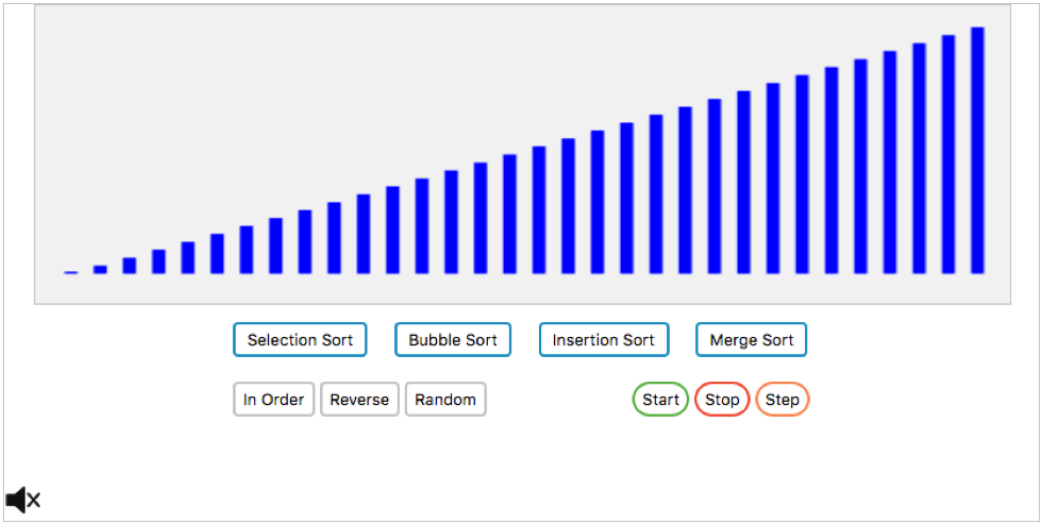

Figure 1: Web site animation tool when loaded.

The top section that shows the blue bars is the canvas area and is what updates to visualize the four sorting algorithms. Below the canvas, the first row of four blue-bordered buttons are the selectable algorithms: Selection Sort, Bubble Sort, Insertion Sort, and Merge/Insertion Sort (the interface does not show the title as "Merge/Insertion Sort" because this was a design

change after the refactor). The user can select any of these algorithms to see the visualization of how that algorithm works. There is no algorithm selected by default, so the user will need to select one before starting the animation.

Before selecting an algorithm, the user must select the type of input data to be sorted. The three gray-bordered buttons on the left of the bottom row allow the user to choose between sorting input data that is already in order (as shown in Figure 1 on the previous page), or in reverse and random orders (shown in Figures 2 and 3 in the following paragraphs). The default is in sorted order.

Once the input and the sorting algorithm have been selected, the user can click the green-bordered "Start" button in the next row of buttons to see the sort run from beginning to end. To see the algorithm execution slowly step-by-step, the user can click the yellow-orange-bordered "Step" button. The "Stop" button simply halts the auto-animating process if in progress.

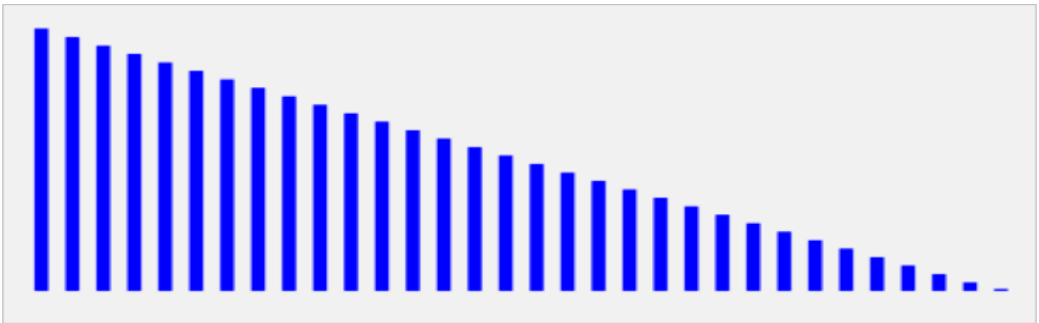

Figure 2: Reverse order.

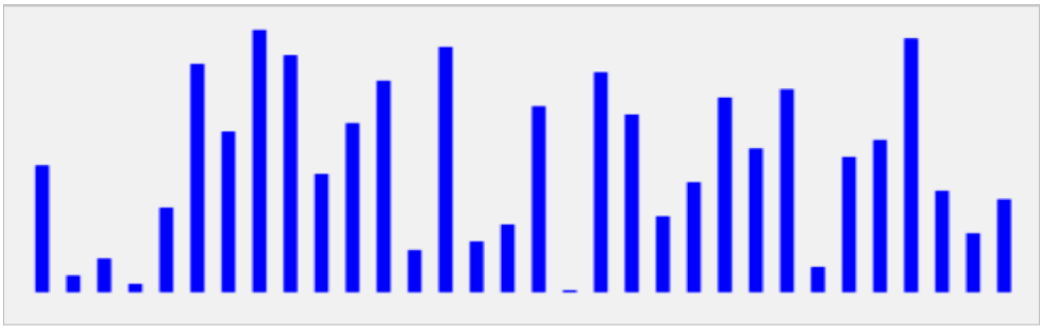

Figure 3: Random order.

I have tried to keep the interface as simple as possible and grouped related buttons together to combat confusion in usability. The algorithm buttons are grouped on their own tier and have a designated blue color. The ordering buttons are also grouped together and have a gray color. The oddballs are the animation controls, where even though they are grouped together, they have separate colors designated to the task they perform. I based the colors on a traffic light where *green* means *go*, *red* means *stop*, and *yellow* means *slow* (or in this case, *yellow-orange* means *pace yourself*).

All the buttons also show feedback by changing color when the cursor hovers over them. An example is shown below where Figure 4 shows an unselected button, and Figure 5 shows the same button with feedback.

Figure 4: Unselected button.

Figure 5: Selected button.

This similar feedback appears on the other buttons according to their

border color.

The final feature available is the volume toggle button, as shown with a speaker image on the lower, left-hand side of the web page. After finding some research involved with adding sounds to animations, I thought it might make the tool more interactive by not only *seeing*, but also *hearing* the animation slowly complete itself. How it works is each bar is assigned a sound based on its height. When the bars are in order, you should hear about four octaves on a piano in order from low to high. When the sound is enabled, a separate sound animation will play the tone of each bar, from left to right as they appear, at key points in the animation. If the bars are out of order, you will hear an out of order sequence of notes and only when sorted will you be able to hear the complete four octaves in order. On the next page in Figure 6, there is a snapshot of the sound animation running when the data is sorted. It is very similar to Figure 1, but the difference is that the twenty-first bar is colored green instead of blue. As the sound animation plays, the bar currently being played will change color to green and then back to blue when finished (which is not well explained by Figure 6, but you get the idea).

It does not, however, work as well as I hoped and since the animation is greedy with memory (which I will discuss in the next chapter), it may cause the animation to halt unexpectedly for a time before continuing. Therefore, the sound animation is disabled by default, but can still be toggled at the user's discretion. TIP: Use the sound animation with Selection Sort for best results.

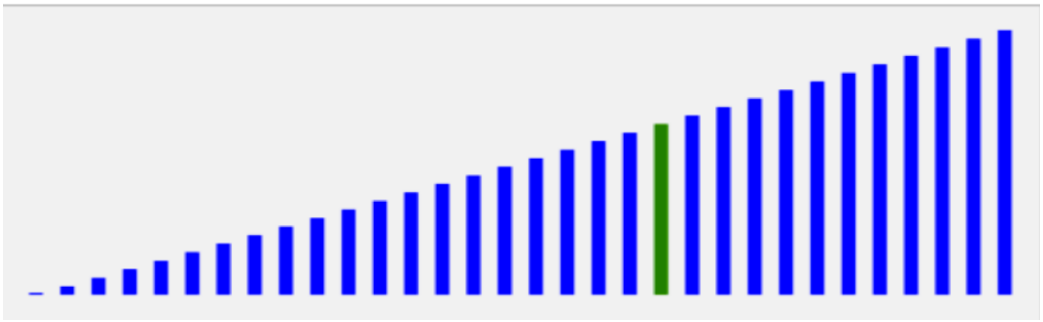

Figure 6: Sound animation in progress from left to right.

A final feature not immediately recognizable is that you can change what sorting algorithm you are using on the fly. For example, if you are stepping through an animation with Selection Sort, you can not only stop the animation, but also select a different algorithm while the animation is stopped, such as Bubble sort. Then by selecting "Start" or "Stop," the animation will continue sorting the bars under the constraints of the new algorithm where the other had left off. This is significant to see how the movement of data can change depending on what algorithms had already affected it. It can also lend a different perspective on how the algorithms perform on semi-sorted data compared to the given ordering options.

3.2 System Architecture

The back-end code is comprised of HTML5, CSS, and JavaScript. All three types of code are contained in one .html file and can be run solely from this file. One of the advantages of HTML 5 is that it is not necessary to include different types of web languages in a single file. Therefore, each type could

have been separated, making a total of three files (plus the miscellaneous sound and image files). This is good practice for readability and keeping related code together. However, I decided not to separate the code for two reasons: 1) to increase the portability of the project by only needing to worry about one project file instead of three, and 2) where in the project file, the change in coding languages is distinctly marked and therefore does not significantly reduce readability. Also, the ability to put more than one web language in a single file is an example of an RIA (Rich Internet Application).

Below in Figure 7 is an illustration of how the three coding languages relate and communicate with each other.

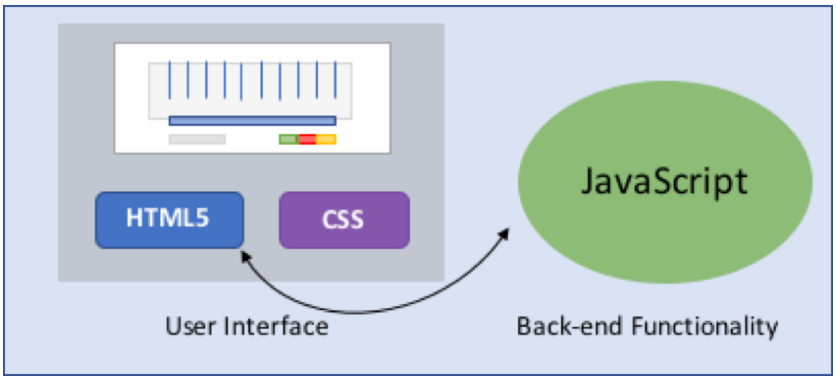

Figure 7: sorting.html file architecture diagram.

As you can see, there are no major components besides the three coding languages. Most websites have tools or scripts that require a server on the back-end (like PHP), but it is not necessary in this case since JavaScript runs right in the user's browser. HTML5 and CSS are used for the interface. The HTML5 communicates with the JavaScript code and vice versa to launch the

appropriate algorithms and update the interface accordingly, as seen with a single, bidirectional arrow.

Throughout the project, the code for the HTML5 and CSS did not change much. As the JavaScript was modified from a functional programming focus to a more object-oriented one, the parts of the HTML5 that did change were the function calls for each button. All of the back-end interaction is abstracted to the various buttons for selecting algorithms and running the animation.

4 Implementation

The implementation of this project is a combination of HTML5 (Hypertext Markup Language 5), JavaScript, and CSS (Cascading Style Sheets). There is only one project file that contains the code and is an HTML file. The only addition to the main HTML file are the individual sound files to support the sound animation feature (saved as ".m4a" audio files). As of now, the preferred browser is Mozilla Firefox as I only performed rigorous testing in this environment. However, quick tests showed possible use in Google Chrome and Safari.

The organization of the code follows both object-oriented and functional programming concepts. Originally, the design was almost completely functional, where only three objects were used: one to control the canvas that displayed the animation, another to represent a piece of data, or "bar" object (blue rectangle with dynamically changing height and position), and a final one to represent the positions that each bar moved to, or "pos" objects. Some instance variables and Boolean values were used to keep track of the algorithm selected and when to animate, but this resulted in a heavily integrated mass of function calls that was difficult to upkeep.

One large refactor later, the code now resembles a Model-View-Controller architecture. Although, due to its functional nature, it has many more individualized functions that update the instance variables and Boolean values, thereby directly updating the View and Controller. A simplified diagram of

the Model-View-Controller relationship is below in Figure 8.

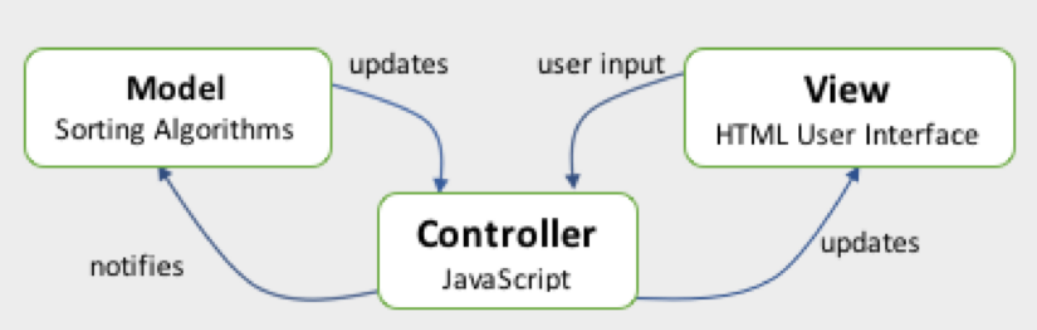

Figure 8: Model-View-Controller diagram of code.

Also, the class/object relationships between components are shown on the next page in Figure 9. The main module represents the global scope in the html file between the `<script></script>` tags. The variables and methods listed in it are accessible by all components.

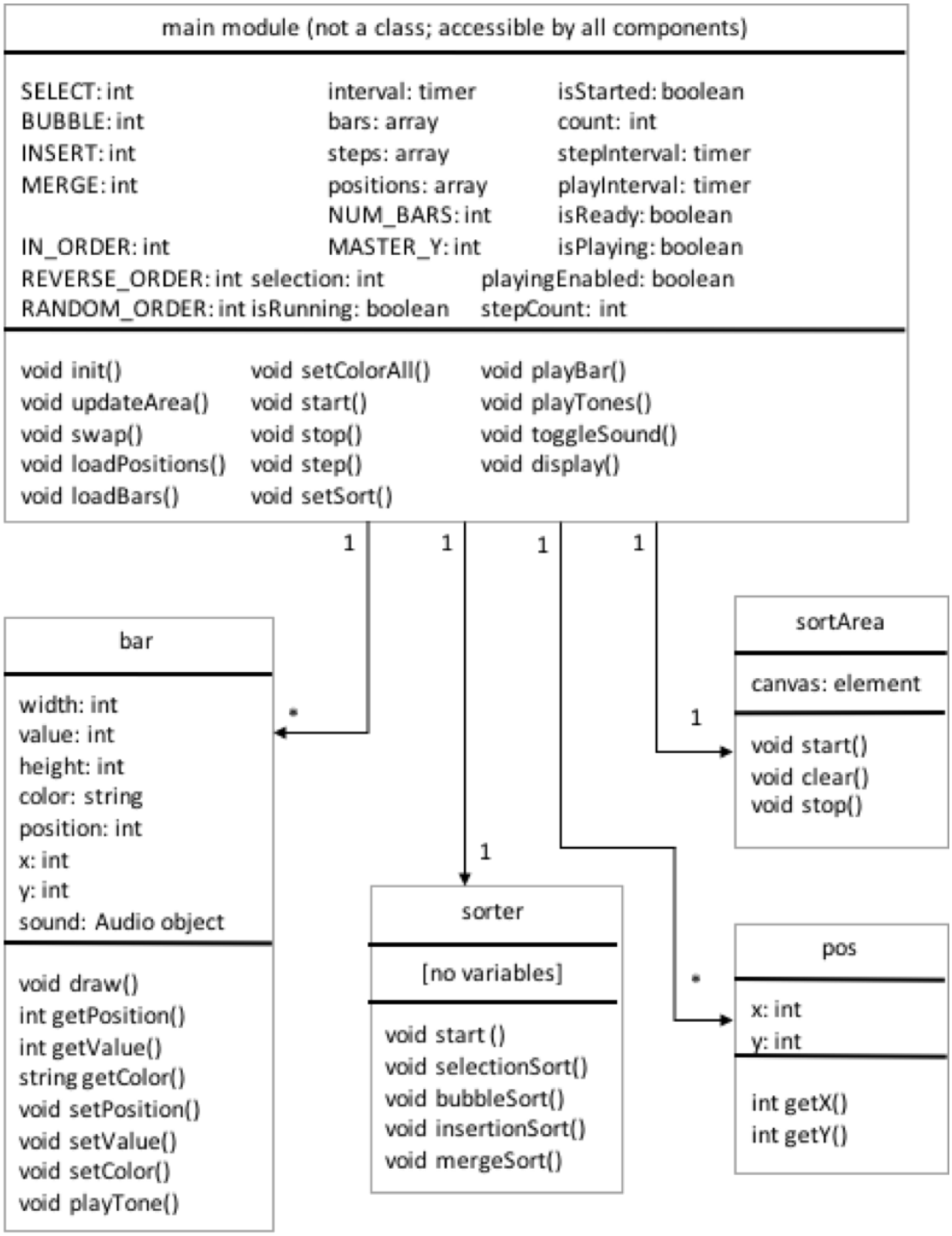

Figure 9: Class/object diagram of instance variables and respective functions in animation tool.

4.1 The View

The View is composed of three objects, called `sortArea`, `bar`, and `pos`. These objects do not belong to any specific object called "View," but instead operate in the space allocated by the `<script></script>` tags in the .html file. You can think of this space as the "main" function, which is the first function called in a program.

`sortArea` is the object that generates the bar graph and runs constantly with a timer to keep the individual bars updated. Therefore, when "Step" is called, it updates the current values of the bars to the new ones based on the steps array (discussed later). Then, the `sortArea` will show the changes by redrawing the rectangles with different heights that reflect the new values at the next iteration of the timer. The bars update sixty times a second, so the change invoked by the "Step" button is seen instantly. I based the timer interval on a game tutorial I studied on the W3Schools website when learning JavaScript.

The **`bar`** object is used to represent each piece of data in the `sortArea`. It includes the attributes color, value, position, height, and sound. By keeping a separate array called `bars` for the current bars in the bar graph, it is then easy to change any or all the attributes by iterating over the array as necessary. The In Order, Reverse, and Random buttons do this by quickly iterating over the array to update the bars to a new data configuration.

Related to the `bar` object is the **`pos`** object (which is short for position). The canvas area that `sortArea` updates is an x-y coordinate grid of pixels. To

make positioning the bars easier, I created this object to assign a coordinate pair to a position number (1-32). Therefore, if I wanted to change the position of a bar, I would only specify the position number. Then, the bar will look up the exact coordinates from that number and move there. For example, position one consists of the coordinate pair (9, 135), which is the bottom left-hand corner of the bar. However, the rectangle object that each bar is drawn from must be given the top left-hand corner coordinates, so the height must also be considered to reposition the bar correctly. Figure 10 shows how the coordinate points relate to the positioning of a bar on the canvas.

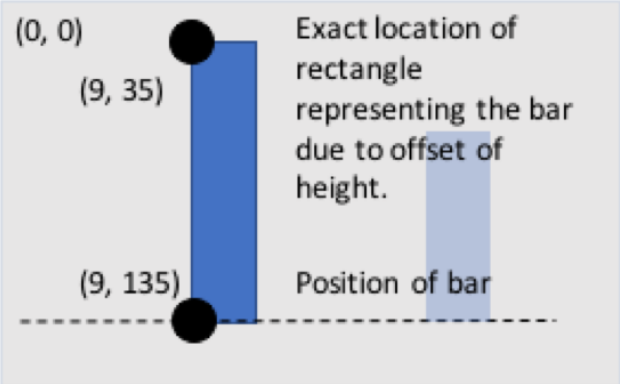

Figure 10: A diagram representing how the position of a bar is calculated.

This feature was used in the earlier version of the program extensively, but after the refactor, it is now only used in the initial setup of the bars when the page loads.

4.2 The Model

The Model is comprised of one object, called `sorter`. This object contains the sorting algorithm code organized as individual methods. A centralized "start" method takes an integer constant as a parameter that the method recognizes as one of the sorting algorithm methods, and then calls the appropriate one. The four buttons labeled as the four sorting algorithms (Selection Sort, Bubble Sort, Insertion Sort, and Merge/Insertion Sort) on the user interface directly control this object.

When the user has selected a data sorting method and clicked on one of the sort-algorithm buttons, the corresponding sorting algorithm method is called, and a trace is generated as the algorithm sorts the data from the current state of the bar graph. This trace is stored as a two-dimensional integer array, called `steps`, that is public and accessible to the methods related to the buttons on the web interface.

The `steps` array is a kind of interface through which the user interface communicates with the back-end code. After the back-end code computes the trace, the "Start," "Stop," and "Step" buttons can directly control which part of the array is visualized on the canvas. Usually, this loads in time before the user selects either "Start" or "Step" to view the animation. The "Step" button is the key player in controlling the animation by loading the next step on the canvas. A timer constantly redraws the bars to update their values, and the result is an animated sequence of a single bar moving through the others to a position determined by the algorithm selected. The "Start"

button simply calls the "Step" button on a timer.

The benefit of using a two-dimensional array is the ability to give the View pieces of the data to visualize the sorting algorithm as steps. If you wanted to add another algorithm, you just need to record the trace of the new algorithm and save it in the same place. Then, the View would iterate over the data and update the bars in the bar graph to reflect the steps that the new algorithm took. One thing to note about how the steps are recorded is that they are only recorded if the algorithm caused a change in the position of a piece of data. For example, Selection Sort finishes quickly because it always moves one piece of data to its correct and final position after each step, where the others can take multiple moves before a piece of data is in its final position. This is deceiving because even though Selection Sort finishes the fastest visually, it does the most work comparing data and is the least favorable choice when sorting data. Therefore, the visualization does not convey data comparisons well, which is the key feature of how sorting algorithms work.

The downside of a two-dimensional array is the memory requirements. Since the trace stores each step needed to sort the data, the size of the array depends on that number of steps. To analyze the space requirements of the algorithm, we can look at the time it requires.

When analyzing time requirements in Computer Science, the common methodology is to use Big-Oh analysis. The notation is a capital letter $\mathcal{O}$ followed by a constraint in parentheses that signifies the worst-case perfor-

mance of the algorithm in question. For example, $\mathcal{O}(n^2)$ — pronounced *Big-Oh of n squared* — is the *time* complexity of Selection Sort and Insertion Sort, as they grow at a rate that is proportionally bounded by n^2, or the number of the items in the set of data squared, to complete. That is not necessarily the same as the space needed for the `steps` array. In all the algorithms, the array is 32 cells wide, because there are 32 pieces of data, but the number of rows needed varies.[4] The size of the `steps` array depends on several things: the sorting algorithm chosen, the points in the algorithm that are displayed, and whether we choose to store the entire array or just compute one row at a time. The points displayed could include, for example, each time any item is moved, or each time an item ends up in its correct location.

The size of the `steps` array directly relates to the algorithm chosen. With Selection Sort, even if we display every time the position of a value changes and store all the rows of the trace in the array at once, we only need n rows. At each step, one of the values is swapped into its correct location, and there are no additional data movements.

Insertion sort, on the other hand, would need $\mathcal{O}(n^2)$ rows. If we display each time a value changes position, the other values must be shifted to put

[4]I picked thirty-two because it is a power of two and divides evenly when Merge/Insertion Sort runs. This way, every time the data divides in half, you get an equal number of values on each side all the way to one. Merge/Insertion Sort will still work on different amounts of data; the difference being that there may be a half with one more value than the other. I also picked thirty-two because it provides a good amount of data to look at visually, making it easier to see a small change propagate through the data.

the value in its correct location. Also, if we only display the state of the array after an entire pass through the data (when the sorted left-hand list of data items contains one more value), we can reduce the number of rows needed to $\mathcal{O}(n)$, but that might not be as useful to someone viewing the animation. Merge/Insertion Sort and Bubble Sort have similar issues. Merge/Insertion Sort must constantly swap the values in small sub arrays to follow its divide-and-conquer technique, creating multiple steps per every merge of a sub array. Bubble Sort almost behaves like Selection Sort, but migrates more than one piece of data in the same direction to their final positions, thereby requiring multiple steps to visualize one outer-loop execution.

These steps are only related, however, to the movement of the data. This is only half the battle, as the comparisons also determine how long an algorithm executes. For example, Selection Sort's poor $\mathcal{O}(n^2)$ runtime is due to how many comparisons it makes, not the memory it requires. The algorithm only performs at most $n-1$ swaps in order to sort all items in the data set, reducing memory need and access. Merge Sort, on the other hand, requires double the memory in order to create a second array as a sorting ground, but by dividing the data into pieces, the number of comparisons made are less when merging the already sorted sub arrays.

Furthermore, computing the complete trace of an algorithm before the animation runs will make the size of the `steps` array vary and create large memory requirements. We already know that to record the movements of the data, a step must be generated for each movement. If the algorithm in ques-

tion is Selection Sort, then the `steps` array will not be very large. However, if Insertion, Bubble, or Merge/Insertion Sort are chosen, the memory need spikes to be able to save the multiple steps needed per every outer-loop execution or recursive call. What can be done instead is to calculate each step as the algorithm runs in parallel in the animation, which in turn uses a one-dimensional array to temporarily hold the current step before being updated to the next one. This will keep a small and constant memory requirement, making the animation tool more stable as it runs (a further discussion after the next few paragraphs).

Overall, in visualizing 32 pieces of data, the worst-case analysis is that the trace will generate $\mathcal{O}(32^2)$ steps, or approximately 1,024 total steps, each of which contain all 32 values in some ordering. Therefore, the two-dimensional array could hold a worse-case total of approximately 32 multiplied by 1,024, or 32,768 pieces of data. The problem this creates is that the browser only has enough memory to support two, maybe three complete animation sequences before being on the verge of crashing due to lack of available memory.

By animation sequence, I mean the process of selecting a data ordering, selecting the algorithm, and then running the animation from start to end. Initializing the two-dimensional array is performed once when an algorithm button is selected. Therefore, the memory constraints only allow about two to three complete sorts. I attempted to combat this by using the JavaScript keywords "`use strict;`" after performing quick research. However, I believe that it only caught some variable initialization errors (such as not fully

declaring variables with a type before assigning a value). The benefit in this case is that it prevented the unwanted creation of additional global variables (ergo, some memory saving), but it did not noticeably improve the performance.

This, in turn, explains the crashing issue mentioned earlier in Chapter 3 with the sound animation running alongside the sorting animation. Since audio files are already large, constantly reloading them to play eats up memory quickly, and the already greedy algorithm animation is enough to start performance issues.

One of my design choices, however, may be well suited to combat this problem. Two of the three control buttons, Start and Step, are directly related to each other. The function of the Step button is to load the next row of data to be displayed in the bar graph, and this change happens immediately with the refresh rate on the canvas. Using this knowledge, I programmed the Start button to call the Step button on a timer, which avoided redundant code and kept the methods small and relatively readable. If I wanted to optimize the generation of the `steps` array, I could do this in the Step function instead, where the function could simply retrieve the next step from the sort algorithm function using a one-dimensional array. As this memory requirement is very small compared to the $\mathcal{O}(n^2)$ analysis earlier, this could ultimately halt the crashing issues. Also, this provides a good place to update the sound animation (discussed later in this chapter), where once a step finishes, the sound animation can directly follow and audibly show the

update.

This appropriate solution would execute the algorithm in steps instead of running it all at once and saving it, and was actually recommended to me by Dr. Robert Ravenscroft, a professor at RIC. He has been building his own sorting animation tool over the past few years, and has implemented this type of stepping solution. He converts his sort methods to yield a step every time they are called successively. This saves space because you only need to store the information for one step at a time, and since the algorithm runs at the same pace as the animation, the old data is quickly replaced with the new data. Therefore, there may never be more than $2 * n$, or in my case $2 * 32 = 64$, pieces of data in memory, instead of 32,768. This also leaves rooms to modify the return type to yield objects instead of raw data, which can then be used to detail comparisons with color changes between the bars. The detailed animation could solve some of the runtime discrepancies that the sorts currently have where Selection Sort runs the fastest even though it has the slowest runtime.

* * *

The second part of the Model is the sound animation feature. This is not represented as an object, but as two methods called with a timer. A method called `playBar` plays the tone assigned to one of the bars in the bar graph. Calling this method on a timer will play the tones of all the bars in order and provides a simple interface for quick use. It will also temporarily change

the color of the bar being played to stress which bar the sound correlates to visually.

Some customization can be added to run through the sound animation at certain points in the algorithm animation sequence, instead of after every step. For example, Selection Sort will call it after every step, but Bubble and Insertion sort call it after every 20 or so steps as those show more movement on average. I found that every 20 steps is a good representation of how the two algorithms sort based on the given ordering options. As the outer loop for each algorithm completes, there are roughly 20 updates to the bar graph where data has moved around. The goal was to play the sound animation directly after each outer loop iteration, but as the data is already sorted and stored in a general array (without using objects), it is impossible to find a correlation after the initial sort.

4.3 The Controller

The Controller consists of all visible buttons on the webpage. These buttons are coded in HTML and directly call a JavaScript method when pressed. The CSS is used for stylizing the appearance, layout, color, and effects of the buttons, along with the canvas. The methods manipulate the state of the Model and perform the necessary updates to prepare and update the View for animation.

The buttons are segregated into four groups. The top, blue-bordered buttons are the sorting algorithm buttons and interact with the `sorter` object

directly. After they are called, the Model builds a `steps` array that is then available for animation. This animation can be controlled by the bottom-right buttons: Start, Stop, and Step. These update the current state of the bar graph by pulling new values from the step array. The View visualizes these updates and the result is the bars changing position based on what algorithm was selected. The Step button is the only button in this group that updates the bars directly. The Start and Stop buttons only control a timer that either calls the Step button at a predetermined speed (every quarter of a second) or stops the timer, respectively.

The bottom-left gray-bordered buttons also directly control the bars. They change the initial position of the bars to be in order, reversed, or random. The positions of the bars do not actually change. Instead, each button simply loads an array of values that reflects the ordering desired. The View instantaneously updates this change on the canvas.

The last group only consists of one button, which is really a picture. The bottom-left hand corner of the webpage has a speaker icon that toggles between "mute" and "audio." This turns on and of the sound animation, so the user can choose whether to run it. By default, the sound animation is muted when the web page loads (for reasons of memory and performance issues).

5 User Testing

In order to test my animation tool, I conducted a demonstration and survey with the assistance of my advisor, Dr. Sanders. The goal was to expose her Computer Science Data Structures class to the tool and have them participate in a survey containing questions that allowed the students to record what they saw and learned, if anything. The questions used in the survey can be found in Appendix C. Even though her class size was of twenty-one students, only thirteen were present, thankfully all of whom completed the survey. This situation was perfect as it met the requirements of my intended audience: a college student population in the field of Computer Science learning algorithms for the first time.

5.1 Designing the Experiment

As this tool is designed to be a learning aid in the classroom, this has presented ethical issues in experiment design. I briefly mentioned this earlier in Chapter 3 as being a part of the paper "Do Algorithm Animations Assist Learning?: An Empirical Study and Analysis" [6]. The students in Stasko's study were randomly separated into two groups. In one, the students were taught the material using conventional methods (i.e. textbook). In the other, the students had both the textbook and an animation tool to assist them when learning the material. This separation is necessary for more reliable results that will hopefully help improve teaching and learning for many

students, but this comes at the cost of giving the participants in the study different experiences. Although Stasko's results proved a non-significant difference in the post-tests of both groups, it still created a noticeable learning gap, giving the students that used the animation tool an advantage over the others.

Given that Stasko had already reported these results and I am fond of fair practices, I decided not to follow his approach and made my post-test into a short survey that asked questions about what they recently learned in class about algorithms and how that changed when exploring my animation tool. An included 5-point rating scale gave the opportunity to judge if they believed the tool helped their current understanding at all. The survey was also voluntary, so it was up to the students' discretion to participate. The animation tool, however, was integrated into their class lecture via my presentation and time dedicated afterward to use the tool. Therefore, all the students were exposed to the tool instead of a select few, which avoided any academic disadvantage.

Some logistics: those who chose to participate in the survey were required to read and sign a consent form so I could use their responses in this paper (as seen in Appendix B). It ensured complete anonymity by using the website SurveyMonkey® to record the responses. The consent forms were also sealed in envelopes and given to me separately to keep discretion between the students. The consent forms and SurveyMonkey® results will be held for three years and secured by means of a locked cabinet and password protec-

tion, respectively. All of this complies with the Institutional Review Board at Rhode Island College, and I have completed training that certified me to conduct this survey (certificate in Appendix E).

5.2 Results

The best way to go about using the tool is to first select the ordering of the data and then select which algorithm to visualize. When any one of the algorithm buttons are selected, it will sort the data as it appears on the interface. The ordering takes precedence, as selecting the ordering after the algorithm updates the interface momentarily, while the code has already run the initialization with the previous data set. After conducting the surveys, this sparked some confusion as the algorithm buttons are listed above the ordering buttons in the interface. One student commented on having difficulty trying to start sorting, thinking that it may be the cause of pressing the buttons in the wrong order, which in turn did not run the animation.

The responses of all the students can be found in Appendix D. Overall, there was not a significant advantage in using my animation tool to help learning about sorting algorithms. By looking at the student responses for question 3, which asked if their understanding of a particular algorithm changed after using the tool, 5 of the 13 students (38%) said yes in some way. The other 7 did not find it very helpful, even though most appreciated the idea of the tool. One student, however, gave a false positive to the tool being helpful (whom I did not include in the 5 that said it was helpful). A

drawback to the animation is that it only shows the movements without the comparisons that lead to the movements of the data. This student saw how Selection Sort completes quickly compared to the other algorithms, as there are $\mathcal{O}(n)$ swaps that take place, which is beneficial in avoiding unnecessary data moves the computer needs to make. In contrast, the process of comparing the data results in a $\mathcal{O}(n^2)$ runtime (the slowest overall). Another student noted this discrepancy in response to question 5 that asked for comments and feedback, noting that Merge Sort is the best of the four sorts. Merge Sort has an average runtime of $\mathcal{O}(n \log_2 n)$, which is the best average runtime out there.

One way to resolve this would be to integrate visualizing the comparisons as well as the movements. This way, the bars would change color when an algorithm is comparing data, taking up more time in the animation. Selection Sort and Bubble sort use the most comparisons, so their time to complete would slow down and be more appropriate compared to the other algorithms.

5.3 Feedback

As I mentioned earlier, the greedy memory requirements of the algorithm create stalemates and crashes during testing. I did let the students know about this issue before hand, and three students politely related to that in response to question 5. I was still surprised to see an average of 3.9 out of 5 stars on question 4 that asked the students to rate the helpfulness of the tool.

Also to my surprise were feature requests that I had previously considered, which I have summarized below:

- Being able to adjust the speed of the animation
- Making the buttons on the user interface stay in a selected state when clicked for visual feedback
- Creating a box-like division for Merge Sort to see the divide steps more clearly
- Change the color of the bars to reflect when they are being compared.

There was a comment that I did not expect from two of the students, who generally stated that they would have preferred to see this animation earlier in the semester. By doing so, they believed the concepts would have been easier to understand. Upon reflection, I realize that the presentation was given right after the students learned about Merge Sort. This sort was fresh in their minds, which explains why 7 of the 13 (54%) chose it to focus on using the animation. Selection Sort and Insertion Sort were taught earlier, so this comment leads me to believe that introducing the animation for each algorithm separately might be a better approach than learning all of the algorithms at once beforehand.

Aside: question 2 asked the students to focus on a particular algorithm, either Selection Sort, Insertion Sort, or Merge/Insertion Sort. Bubble Sort

was omitted from the survey as it is not part of the Data Structures curriculum at RIC. I did demo it along with the other algorithms and was happy to see enlightened faces of understanding for an algorithm they have not seen before. This gives proof to the idea of making animations, and although I experiences some setbacks with my memory organization, I still have successfully introduced a new concept using the animation tool.

6 Conclusions and Future Work

Through much time and effort, I have successfully created a working web-based animation tool for visualizing the following sorting algorithms: Selection Sort, Bubble Sort, Insertion Sort, and Merge/Insertion Sort. Even with its memory overhead, it received overall positive feedback from the students who explored it. I am not surprised that there was not a significant difference in learning the material, which reflects what I found in my previous research. There remains, however, a strong mindset to research and create animations like these to improve learning in the classroom, which I agree with completely.

Learning how to code a web platform was challenging, and I thank the tutorials on W3Schools.com for getting me there. I had a previous internship where I updated the JavaScript on a webpage, but it was much more concise and did not involve objects and HTML5 for visualizations. The good news is that JavaScript is still one of the most popular web languages, so I am not too worried about another big refactor soon for a language update.

For my laundry list of future works, the elephant in the room is to resolve the memory issues. Next would be to modify Merge/Insertion Sort to reflect a true Merge Sort. After which, I would get Quick Sort up and running, as the code is already in a state where it would not be too difficult to integrate. Then, I would add the suggestions listed in the bulleted feedback of Chapter 5 to further promote usability and understandability. Finally, I would make

the web tool public, realizing my most desired feature of making it public. This would also present some new challenges. Even though the animation tool works locally, I have unintentionally avoided the issue of concurrency, where a server can handle multiple requests to the web site by different users. I would need to give more thought on how to optimize the code so that it can work with multiple people using it. This idea was presented to me by Dr. Rafaat Elfouly, a Computer Science professor at RIC. He also commented on how it would be nice to have a feature that would analyze the time it takes for the sorting animations to complete, which would be beneficial for comparative analysis.

Special Thanks

Special thanks to my advisor, Dr. Kathryn Sanders, for helping me organize this paper, the survey, and get approval with the Institutional Review Board. I also thank my previous advisor, Dr. Edward McDowell, for helping me kick start this project by creating a completion plan. Furthermore, I thank Dr. Robert Ravenscroft for his feedback and recommendation for the code refactor when I was experiencing timer integration issues with my previous version of the animation tool. Also, I thank Dr. Rafaat Elfouly for his forward-thinking ideas on how to improve the animation tool. Last, but certainly not least, thanks to all the students who participated in this study and produced valuable feedback.

References

[1] T. Bingmann. “The Sound of Sorting - ‘Audibilization’ and Visualization of Sorting Algorithms.” *Panthemanet Weblog*. Impressum, 22 May 2013. Web. 29 Mar. 2017. <http://panthema.net/2013/sound-of-sorting/>.

[2] *Bubble-sort with Hungarian (“Csángó”) Folk Dance*. Dir. Kátai Zoltán and Tóth László. *YouTube*. Sapientia University, 29 Mar. 2011. Web. 29 Mar. 2017. <https://www.youtube.com/watch?v=lyZQPjUT5B4>.

[3] A. Kerren and J. T. Stasko. (2002) Chapter 1 Algorithm Animation. In: *Diehl S.(eds) Software Visualization*. Lecture Notes in Computer Science, vol 2269. Springer, Berlin, Heidelberg. <http://homepage.lnu.se/staff/akemsi/pubs/22690001.pdf>.

[4] A. Moreno, E. Sutinen, R. Bednarik, and N. Myller. Conflictive animations as engaging learning tools. *Proceedings of the Koli Calling ‘07 Proceedings of the Seventh Baltic Sea Conference on Computing Education Research - Volume 88*, Koli ‘07 (Koli National Park, Finland), pages 203-206. <http://dl.acm.org/citation.cfm?id=2449352>.

[5] J. Stasko. Using Student-built Algorithm Animations As Learning Aids. *Proceedings of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ‘97 (San Jose, California), pages 25-29. <http://doi.acm.org/10.1145/268084.268091>.

[6] J. Stasko, A. Badre, and C. Lewis. Do Algorithm Animations Assist Learning?: An Empirical Study and Analysis. *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, CHI-93 (Amsterdam, the Netherlands), pages 61-66. <http://doi.acm.org/10.1145/169059.169078>.

A Program Code

```
<!DOCTYPE html>
<html>
<head>
<title>Sorting Algorithms</title>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">

<!-- CSS Styling -->
<style>
// outside box border
#main_wrapper {
    border:   1px solid #cccccc;
    width:    52em;
    height:   26em;
    //overflow: hidden;
    margin:   auto;
}

// position sortArea centered in box
#sort_wrapper {
    width:  49em;
    height: 15em;
    margin: auto;
    //border: 5px solid #000000;
}

// canvas area that displays bars
#sortArea {
    width:            49em;
    height:           15em;
    border:           1px solid #c3c3c3;
    background-color: #f1f1f1;
}
```

```
// wrapper for ordering buttons
#adjust_bars {
    //border:   5px solid #c3c3c3;
    position: relative;
    float:    left;
}

// wrapper for Start, Stop, and Step buttons
#start_stop {
    //border:   5px solid #c3c3c3;
    position: relative;
    float:    right;
}

// outside wrapper for bottom buttons
#control_buttons {
    width:  29em;
    height: 3em;
    margin: auto;
    //border: 5px solid #c3c3c3;
}

// wrapper for sorting algorithm buttons
#algorithm_buttons {
    width:  29em;
    height: 3em;
    margin: auto;
    //border: 5px solid #c3c3c3;
}

// style class for button size, curve, and fade effects
.btn {
    border-radius: 4px;
    color: white;
    padding: .25em .25em;
    text-align: center;
    text-decoration: none;
    display: inline-block;
```

```
    font-size: .8em;
    -webkit-transition-duration: 0.4s; /* Safari */
    transition-duration: 0.4s;
    cursor: pointer;
}

// style classes specific to algorithm buttons
.algo {
    background-color: white;
    color: black;
    border: 2px solid #008CBA;
    padding: .25em .5em;
}
.algo:hover {
    background-color: #008CBA;
    color: white;
}

// style classes specific to ordering buttons
.control {
    background-color: white;
    color: black;
    border: 2px solid #c3c3c3;
}
.control:hover {
    background-color: #c3c3c3;
}

// style classes specific to Start/Stop buttons
.act {
    border-radius: 15px;
    background-color: white;
    color: black;
}
.act:hover {
    border-radius: 15px;
    background-color: #f44336;
    color: white;
```

```
}

// Additional styles for Start/Stop/Step colors
#start {
    border: 2px solid #4CAF50;
}
#start:hover {
    background-color: #4CAF50;
}
#stop {
    border: 2px solid #f44336;
}
#stop:hover {
    background-color: #f44336;
}
#step {
    border: 2px solid #ff7f50;
}
#step:hover {
    background-color: #ff7f50;
}

// A text field for showing instructions (depreciated)
#output {
    width:  16em;
    height: 2em;
    margin: auto;
    font-family: Arial;
    //border: 5px solid #c3c3c3;
}

// algorithm button spacing
span ~ span {
    padding-left: 1.1em;
}
</style>
</head>
```

```
<body onload="init();">

  <div id="main_wrapper">
        <div id="sort_wrapper">
      <canvas id="sortArea">Your browser does not support
          the canvas element.</canvas>
    </div> <!-- end sort_wrapper -->

        <div id="algorithm_buttons">
          <p>
        <span>
          <button class="btn algo" id="selection"
          onclick="setSort(SELECT);">Selection Sort</button>
        </span>
        <span>
          <button class="btn algo" id="bubble"
          onclick="setSort(BUBBLE);">Bubble Sort</button>
        </span>
        <span>
          <button class="btn algo" id="insertion"
          onclick="setSort(INSERT);">Insertion Sort</button>
        </span>
        <span>
          <button class="btn algo" id="merge"
          onclick="setSort(MERGE);">Merge Sort</button>
        </span>
      </p>
    </div> <!-- end algorithm_buttons -->
        <div id="control_buttons">
      <div id="actions">
        <div id="adjust_bars">
        <button class="btn control" id="in_order"
        onclick="loadBars(IN_ORDER);">In Order</button>
        <button class="btn control" id="reverse"
        onclick="loadBars(REVERSE_ORDER);">Reverse</button>
        <button class="btn control" id="random"
        onclick="loadBars(RANDOM_ORDER);">Random</button>
        </div> <!-- end adjust_bars -->
```

```
        <div id="start_stop">
            <button class="btn act" id="start"
                onclick="start();">Start</button>
            <button class="btn act" id="stop"
                onclick="stop();">Stop</button>
            <button class="btn act" id="step"
                onclick="step();">Step</button>
         </div> <!-- end start_stop -->
        </div> <!-- end actions -->
    </div> <!-- end control_buttons -->
    <p id="output"></p>
    <div id="sound">
      <img id="audio" src="sound.png" height="30"
          width="30" onclick="toggleSound();"></img>
      <img id="mute" src="mute.png" height="30"
          width="30" onclick="toggleSound();"></img>
    </div>
  </div> <!-- end main_wrapper -->

<script>
"use strict";
// Instance variable for sorts
const SELECT = 1;
const BUBBLE = 2;
const INSERT = 3;
const MERGE = 5;

// Instance variables for data ordering
const IN_ORDER = 1;
const REVERSE_ORDER = 2;
const RANDOM_ORDER = 3;

var interval = null;  // Timer for canvas update
var bars = []; // holds bars that are sorted
var steps = []; // holds steps of algorithm animation
var positions = []; // x-y coordinates for bars
var NUM_BARS = 32; // max number of bars
var MASTER_Y = 135;  // y-coordinate constraint
```

```
var selection = 0; // algorithm user selected
var isRunning = false; // True if canvas is updating
var isStarted = false; // True if sorting animation started
var count = 0;  // counter for playing tones
var stepInterval; // timer for automatic animation
var playInterval; // timer for playing tones of each bar
var isReady = true; // animation is ready to start
var isPlaying = false; // tones are currently being played
var playingEnabled = false; // enable/disable sound feature
var stepCount = 0;  // count for traversing steps

// Launched when web page loads
// Displays bars in order and starts canvas updating/
// redrawing feature
function init() {
    document.getElementById("audio").hidden=true;
    loadPositions();
    loadBars(IN_ORDER);
    sortArea.start(); // draw/update bars
}

// Object that represent canvas that bars are on.
// Has methods to start, stop, and clear canvas
var sortArea = {
    canvas : document.getElementById('sortArea'),
    start : function() {
        this.context = this.canvas.getContext('2d');
        interval = setInterval(updateArea, 60);
        // method below
        isRunning = true;
    },
    clear : function() {
        this.context.clearRect(0, 0,
            this.canvas.width, this.canvas.height);
    },
    stop : function() {
        clearInterval(interval);
        interval = null;
```

```
        isRunning = false;
    }
}

// Object that handles sorting methods
var sorter = {
    // Called to load steps for respective sort
    start : function(SORT) {
        isReady = false;
        switch(SORT) {
            case SELECT:
                this.selectionSort();
                isReady = true;
                break;
            case BUBBLE:
                this.bubbleSort();
                isReady = true;
                break;
            case INSERT:
                this.insertionSort(0, NUM_BARS, true);
                isReady = true;
                break;
            case MERGE:
                steps = [];
                this.mergeSort(0, NUM_BARS);
                isReady = true;
                break;
            default:
                alert("ERROR - Unable to start sorting.");
                break;
        }
    },
    selectionSort : function () {
        // Adapted from: http://codingmiles.com/sorting-
        // algorithms-selection-sort-using-javascript/

        var values = [];
        for (var i = 0; i < NUM_BARS; i++) {
```

```
    values.push(bars[i].getValue());
}

steps = [];
steps.push(values.slice(0)); // store copy of values
                             // (a single step)

for (var i = 0; i < NUM_BARS - 1; i++) {
    //Number of passes
    var min = i;
    // set current min as the initial value found
    for (var j = i + 1; j < NUM_BARS; j++) {
        //Note that j = i + 1 as we
        / only need to go through unsorted array
        //this.clone(bars);

        if (values[j] < values[min]) {
            //Compare the numbers
            //this.clone();
            min = j;
            //Change the current min number
            // position if a smaller num is found
        }
    }
    if (min != i) {
        //After each pass, if the current min num
        // != initial min num, exchange the position.
        //Swap the numbers
        var tmp = values[i];
        values[i] = values[min];
        values[min] = tmp;
        steps.push(values.slice(0));
    }
}
//var text = ""
//for (var i = 0; i < steps.length; i++) {
//    for (var j = 0; j < NUM_BARS; j++) {
//        text += steps[i][j] + " ";
```

```
    //     }
    //     text += "\n";
    //}
    //console.log(text);
},
bubbleSort : function() {
    // Adapted from: http://codingmiles.com/sorting
    //-algorithms-bubble-sort-using-javascript/
    var values = [];
    for (var i = 0; i < NUM_BARS; i++) {
        values.push(bars[i].getValue());
    }

    steps = [];
    steps.push(values.slice(0));

    for (var i = NUM_BARS - 1; i >= 0; i--) {
        for (var j = 0; j <= i; j++) {
            if (values[j] > values[j+1]) {
                var tmp = values[j];
                values[j] = values[j+1];
                values[j+1] = tmp;
                steps.push(values.slice(0));
            }
        }
    }
    //var text = ""
    //for (var i = 0; i < steps.length; i++) {
    //    for (var j = 0; j < NUM_BARS; j++) {
    //        text += steps[i][j] + " ";
    //    }
    //    text += "\n";
    //}
    //console.log(text);
},
insertionSort : function(start, end, isSingleRun) {
    var values = [];
```

```
if (isSingleRun) {
    for (var i = 0; i < NUM_BARS; i++) {
        values.push(bars[i].getValue());
    }

    steps = [];
    steps.push(values.slice(0));
}
else {
    if (steps.length < 1) {
        for (var i = 0; i < NUM_BARS; i++) {
            values.push(bars[i].getValue());
        }
    }
    else {
        values = steps[steps.length - 1].slice(0);
    }
}

for (var i = start; i < end; i++) {
    var tmp = values[i]; //Copy of the current element.
    /*Check through the sorted part and compare with
    the number in tmp. If large, shift the number*/
    for (var j = i - 1; j >= start
            && (values[j] > tmp); j--) {
        var swapping = values[j];
        values[j] = values[j+1];
        values[j+1] = swapping;
        steps.push(values.slice(0));
    }
}

//var text = "";
//for (var i = 0; i < steps.length; i++) {
//    for (var j = 0; j < NUM_BARS; j++) {
//        text += steps[i][j] + " ";
//    }
//    text += "\n";
```

```
        //}
        //console.log(text);
    },
    mergeSort : function(start, end) {
        // END IS NOT INCLUSIVE
        if (end - start < 2) {
            return; // Do nothing
        }

        var middle = parseInt((start + end) / 2);
        this.mergeSort(start, middle);
        this.mergeSort(middle, end);
        this.insertionSort(start, end, false);
    }
}

// Class bar
// Creates a sortable rectangle object with a
//  value, position, and color.
// The value determines the height of the bar
//  (height = value * 4px).
function bar(value, posn, color) {
    this.width = 4;
    this.value = value;
    this.height = (value == 0) ? 1 : (this.width * value);
        // Allows you to see a value of zero
    this.color = color;
    this.position = posn;
    this.x = positions[posn].getX();
    this.y = positions[posn].getY() - this.height;
    this.sound = new Audio((this.value + 1) + ".m4a");

    this.draw = function() {
        sortArea.context.fillStyle = this.color;
        sortArea.context.fillRect(this.x, this.y,
                                  this.width, this.height);
    };
    this.getPosition = function() {
```

```
        return this.position;
    };
    this.getValue = function() {
        return this.value;
    };
    this.getColor = function() {
        return this.color;
    };
    this.setPosition = function(pos) {
        this.position = pos;
        this.x = positions[pos].getX();
        this.y = positions[pos].getY() - this.height;
    };
    this.setValue = function(val) {
        this.value = val;
        this.height = (val == 0) ? 1 : (this.width * val);
        this.y = positions[this.position].getY()
            - this.height;
        this.sound = new Audio((val + 1) + ".m4a");
    };
    this.setColor = function(aColor) {
        this.color = aColor;
    };
    this.playTone = function() {
        //this.sound.pause();
        this.sound.currentTime = 0;
        this.sound.play();
    };
}

// Swap bar objects in array
// NOT CURRENTLY USED
function swap(pos1, pos2) {
    temp = bars[pos1];
    // Move bar2 to bar1's position in array
    bars[pos1] = bars[pos2];
    // Enforce position in bar2 to be position 1
    bars[pos1].setPosition(pos1);
```

```
    // Move bar1 to bar2's position in array
    bars[pos2] = temp;
    // Enforce position in bar1 to be position 2
    bars[pos2].setPosition(pos2);
}

// Class pos
// Represents a position for a bar on the canvas.
function pos(x, y) {
    this.x = x;
    this.y = y;
    this.getX = function () {return this.x;};
    this.getY = function () {return this.y;};
}

// Called by the sortArea object to redraw the bars
function updateArea() {
    sortArea.clear();
    for (var i = 0; i < NUM_BARS; i++) {
        bars[i].draw();
    }
}

// Generates pos objects for positioning bars on canvas
function loadPositions() {
    // 14 positions total: 0-13
    var offset = 9;
    // Create space between first bar and left
    //  side of sort area.
    for (var i = 0; i <= (NUM_BARS * offset); i += offset) {
        positions.push(new pos(i + offset, MASTER_Y));
    }
}

// Redraw bars in a different ordering
function loadBars(ORDER) {
    bars = []; // clear array
```

```
    switch (ORDER) {
        case IN_ORDER:
            for (var i = 0; i < NUM_BARS; i++) {
                bars.push(new bar(i, i, "blue"));
            }
            break;
        case REVERSE_ORDER:
            var count = 0;
            for (var i = NUM_BARS - 1; i >= 0; i--) {
                bars.push(new bar(i, count, "blue"));
                count++;
            }
            break;
        case RANDOM_ORDER:
            var randomValues =
                [15, 2, 4, 1, 10, 27, 19, 31, 28,
                14, 20, 25, 5, 29, 6, 8, 22, 0,
                26, 21, 9, 13, 23, 17, 24, 3, 16,
                18, 30, 12, 7, 11];

            for (var i = 0; i < NUM_BARS; i++) {
                bars.push(new bar(randomValues[i], i, "blue"));
            }
            break;
        default:
            alert("ERROR - Incorrect argument for loadbars.");
            break;
    }
}

// Change color of all bars
// NOT CURRENTLY USED
function setColorAll(aColor) {
    for (i = 0; i < NUM_BARS; i++) {
        bars[i].setColor(aColor);
    }
}
```

```
// Runs animation sequence
function start() {
    if (! isStarted) {
        isStarted = true;
        stepInterval = setInterval(step, 250);
    }
}

// Stops animation sequence
function stop() {
    //alert("Done!");
    isStarted = false;
    clearInterval(stepInterval);
    stepInterval = null;
}

// Loads a step of the algorithm at a time
function step() {
    if (isReady && ! isPlaying) {
        if (stepCount < steps.length) {
            //bars = [];
            for (var i = 0; i < NUM_BARS; i++) {
                bars[i].setValue(steps[stepCount][i]);
            }
            stepCount++;
            playTones();
        }
        else {
            stop();
            isReady = false;
            stepCount = 0;
        }
    }
}

// Record selection of user and prep algorithm
function setSort(num) {
    selection = num;
```

```
    sorter.start(num);
}

// Plays tones for respective sorting algorithm
function playTones() {
    if (playingEnabled) {
        count = 0;
        isPlaying = true;
        switch(selection) {
            case SELECT:
                playInterval = setInterval(playBar, 75);
                break;
            case BUBBLE: case INSERT:
                if (stepCount % (NUM_BARS - 8) == 0) {
                    playInterval = setInterval(playBar, 75);
                }
                else {
                    isPlaying = false;
                }
                break;
            case MERGE:
                if (stepCount <= 1 || stepCount
                        == steps.length) {
                    playInterval = setInterval(playBar, 75);
                }
                else {
                    isPlaying = false;
                }
                break;
            default:
                break;
        }
        //playInterval = setInterval(playBar, 75);
    }
}

// Called when sound is enabled
function playBar() {
```

```
    if (count < NUM_BARS) {
        bars[count].playTone();
        bars[count].setColor("green");
        if (count != 0) {
            bars[count - 1].setColor("blue");
            // Set back to normal color
        }
        count++;
    }
    else {
        isPlaying = false;
        clearInterval(playInterval);
        playInterval = null;
        bars[count - 1].setColor("blue");
        // Set last bar back to normal color
    }
}

// Enables/disables sound and changes sound icon
function toggleSound() {
    if (playingEnabled) {
        playingEnabled = false;
        document.getElementById("audio").hidden=true;
        document.getElementById("mute").hidden=false;
    }
    else {
        playingEnabled = true;
        document.getElementById("audio").hidden=false;
        document.getElementById("mute").hidden=true;
    }
}

// Display text to certain part of screen
// NOT CURRENTLY USED
function display(text) {
    document.getElementById('output').innerHTML = text;
}
</script></body></html>
```

B Consent Form

The next couple pages are the documents that represent the consent form for taking the survey.

CONSENT DOCUMENT
Rhode Island College

VISUALIZING SORTING ALGORITHMS

You are being asked to be in a research study about the significance of visual aids as learning tools in an academic environment. You are being asked because you are in the Computer Science curriculum at Rhode Island College and are/will be/have been exposed to sorting algorithms, which is what will be visualized in the form of an interactive web page. Please read this form and ask any questions that you have before choosing whether to be in the study.

Brian Faria, a student at Rhode Island College, is doing this study with Dr. Kathryn Sanders, a professor at Rhode Island College, as his advisor.

Why this Study is Being Done (Purpose)
We are doing this study to learn about the possible benefit of learning abstract course material in a visual form, rather than conventional textbook practices. Significant research has been done in visualizing sorting algorithms because of their already abstract nature, which can lead to difficulty in understanding the concepts fully.

What You Will Have to Do (Procedures)
All students in the class will watch a brief presentation on what the project is about and the visual tool developed as part of the project (including a demo of the tool) and try out the tool.

If you choose to be in the study, we will ask you to also do the following:
- fill out a short, anonymous survey on whether the tool helps you to learn.

You Will NOT Be Paid (Compensation)
There is no compensation.

Risks or Discomforts
There are no risks involved in the study other than those involved in completing the usual coursework for a class. Nevertheless, if you feel uncomfortable at any time, you do not have to continue with the survey and may leave at any point. In addition, on the survey, you can skip any questions you don't want to answer.

Benefits of Being in the Study
There are no direct benefits. All students are being exposed to a different form of the material they are learning, which may clarify concepts or ideas. There are no additional benefits tied to filling out the survey.

Deciding Whether to Be in the Study

Being in the study is your choice to make. Nobody can force you to be in the study. You can choose not to be in the study, and nobody will hold it against you. You can change your mind and quit the study at any time, and you do not have to give a reason. If you decide to quit later, nobody will hold it against you.

How Your Information will be Protected

Because this is a research study, results will be summarized across all participants and shared in reports that we publish and presentations that we give. Your name will not be used in any reports. We will take several steps to protect the information you give us so that you cannot be identified. Instead of using your name, your information will be given a code number. The information will be kept in a locked office file, and seen only by myself and other researchers who work with me. The only time I would have to share information from the study is if it is subpoenaed by a court, or if you are suspected of harming yourself or others, then I would have to report it to the appropriate authorities. Also, if there are problems with the study, the records may be viewed by the Rhode Island College review board responsible for protecting the rights and safety of people who participate in research. The information will be kept for a minimum of three years after the study is over, after which it will be destroyed.

Who to Contact

You can ask any questions you have now. If you have any questions later, you can contact me at my student email: bfaria_1722@email.ric.edu or my phone: (401) 829-5971 or Dr. Sanders at ksanders@ric.edu or her phone: (401) 456-8038.

If you think you were treated badly in this study, have complaints, or would like to talk to someone other than the researcher about your rights or safety as a research participant, please contact Cindy Padula at IRB@ric.edu, by phone at 401-456-9720.

You will be given a copy of this form to keep.

Statement of Consent

I have read and understand the information above. I am choosing to be in the study Visualizing Sorting Algorithms". I can change my mind and quit at any time, and I don't have to give a reason. I have been given answers to the questions I asked, or I will contact the researcher with any questions that come up later. I am at least 18 years of age.

Print Name of Participant: _____

Signature of Participant: _____ Date: _____

Name of Researcher Obtaining Consent: Brian Faria

C Survey Questions

Below is the list of questions that the survey contained, as cited in Chapter 5. These numbers can be used as a reference for the responses in Appendix D. Note: For answers to question four in Appendix D, the answers taken were represented as a star rating out of five. I modified the quotation by saying the number of stars given explicitly. Also, any omitted answers are shown as "[skipped]."

1. Please choose one of the following algorithms to examine using this tool:
   (a) Selection Sort
   (b) Insertion Sort
   (c) Merge Sort
2. Before experimenting with the tool, briefly explain how the algorithm you chose works (at most 1 paragraph):
3. Go ahead and experiment with the tool! You can find it `here`. Make sure to try the algorithm you selected. Do you understand the algorithm differently after using the tool? If so, please explain in what way your understanding changed:
4. On a scale of 1-5, did this tool help you understand the algorithm you selected?
5. Please feel free to leave any other observations or comments below!

D Survey Responses

Respondent 1

1. “Merge Sort”
2. “divides array in halves”
3. “no”
4. “[5 out of 5 Stars]”
5. “very good presentation, wish I saw this at the beginning of the semester. or in class while learning it”

Respondent 2

1. “Merge Sort”
2. “separates the items in half until just 2 are left then sorts over and over”
3. “No”
4. “[1 out of 5 Stars]”
5. “Tough to see how it works without some sort of visualization of the bars being separated in half. Also takes a long time for the merge to complete and from my understanding it is supposed to be one of the most efficient sorts.”

Respondent 3

1. “Merge Sort”
2. “Merge sort works by dividing an array into two halves, and from there, dividing one of the two halves into two more halves. Those are then divided until they can be divided no more. They are then sorted in their respective groups, and merged together, after being sorted. The same is done with the second half, and then they are finally merged all together.”

3. “With how the program seems to be running, it appears to be sorting as anticipated. The two halves are divided into groups, sorted, and then merged with one another, respectively.”

4. “[5 out of 5 Stars]”

5. “The presentation and program was very helpful on the concepts of the different sorting methods. I had a good understanding of the sorting methods presented beforehand, but I believe this one have helped me greatly, if presented to me in the beginning.”

Respondent 4

1. “Selection Sort”

2. “selection sort uses the ‘swap’ method to pull the smallest pillar to move it to the far left position, and continues to find the next smallest one until everything is arranged by height from smallest to largest.”

3. “Helps to have a visual, easier to understand exactly what is happening.”

4. “[4 out of 5 Stars]”

5. “[skipped]”

Respondent 5

1. “Merge Sort”

2. “splits list into two equal halves, then take the first halves split that in half until you are left with two terms. compare terms then sort. repeat this with other halves until the whole list is sorted.”

3. “very nice visual representation of the different types of sort”

4. “[4 out of 5 Stars]”

5. “[skipped]”

Respondent 6

1. “Selection Sort”
2. “swaps from the original index and swaps any number in the array that is smaller.”
3. “yes”
4. “[5 out of 5 stars]”
5. “great, its more than enough to have the general idea of the sorting algorithms.”

Respondent 7

1. “Merge Sort”
2. “split in half and sort”
3. “Yes.”
4. “[5 out of 5 stars]”
5. “[skipped]”

Respondent 8

1. “Merge Sort”
2. “Merge sort works by dividing the list into smaller and smaller sections of the list, and then sorting them in the smallest form of the list before merging them back together.”
3. “Watching the merge sort a reverse ordered list was satisfying to watch because you could see it divide the list into smaller and equal parts before merging them.”
4. “[4 out of 5 Stars]”
5. “Having a box that showed the steps involved with each sorting method would help me understand what is actually happening in the visual example.”

Respondent 9

1. "Selection Sort"

2. "The smallest is switched with the left most that is not smaller then the smallest that is being switched. So, n switches at index 0 unless index 0 is smaller than n so it will move onto compare with the next index. Then it just keeps going. The switch is faster than the other 2 sorts."

3. "I understand the algorithm differently. Where n is and the index it is comparing to, they switch right away. It is a quick compare and contrast and switches right away."

4. "[5 out of 5 Stars]"

5. "The change in color would give a better view on where it is changing. Could change the color to help the user identify what is changing then switch it back to the original color. It would of helped me a lot more. The noise would stimulate what was being done, as you explained. Any sense that is tapped while doing the sorting would help an user identify what is happening and helps them remember more. It would be very helpful but due to the issues with it, it is still done very well. I would use this to help later classes in this Data Structure course."

Respondent 10

1. "Merge Sort"

2. "Elements of the list are grouped into smaller lists and then compared to one another until all of the smaller lists are compiled into the larger list. At this point the larger list will be completely sorted."

3. "I understood the algorithms before I used the tool so it had no modification of my knowledge."

4. "[1 out of 5 Stars]"

5. "It's fine I guess."

Respondent 11

1. "Selection Sort"

2. "Moves through a list of values looking for the smallest value remaining and inserts that value in proper order."

3. "It was nice to be able to visualize the process. Insertion sort is pretty straight forward, but seeing merge sort really helped my understanding of how the process works, and I feel this program is a very useful visual aid."

4. "[4 out of 5 Stars]"

5. "The ability for the user to control the speed of the start animation would be helpful. To see it a little slower without having to step through it would be nice. Overall I think its very helpful."

Respondent 12

1. "Selection Sort"

2. "Selection sort goes through the array and finds the largest or smallest value, depending on how you are sorting your data, and switches it with a previous index of the array. Visually, the array is sorted from left to right."

3. "I don't think I understand the algorithm differently after using the tool, unfortunately. I think when using this tool I was a little confused with which items were being switched. It would be helpful to see where the indexes are."

4. "[3 out of 5 Stars]"

5. "Graphically, it would be helpful to know when a button has been pressed - like if it stayed highlighted after clicking on it. It might have been because of the browser but sometimes I would click on things (maybe in the wrong order so this might not matter) and was confused that nothing happened afterwards."

Respondent 13

1. “Insertion Sort”

2. “insertion sort goes through the array and places the current element relative to the other data around it in an ordered manner”

3. “understanding remains the same, this program enhances the ideas of each of the sorting methods. Good work!”

4. “[5 out of 5 Stars]”

5. “Overall this is an interesting program that would likely be beneficial to other people pursuing computer sciences. Something I think would be beneficial to people trying to understand how the sorting algorithm would work would be to have some kind of movement log, but just a thought. There is bugs that I noticed you are probably aware of but stopping partway through a sort and selecting a different sort, the first part of the graph will look like its somewhat sorted and would continue from this point on, just an observation. Good luck and good work!”

E IRB Survey Certification

The next couple pages are the documents that certified me for conducting a study with human subjects.

COLLABORATIVE INSTITUTIONAL TRAINING INITIATIVE (CITI PROGRAM)

COURSEWORK REQUIREMENTS REPORT*

* NOTE: Scores on this Requirements Report reflect quiz completions at the time all requirements for the course were met. See list below for details. See separate Transcript Report for more recent quiz scores, including those on optional (supplemental) course elements.

- **Name:** Brian Faria (ID: 5474097)
- **Email:** bfaria_1722@email.ric.edu
- **Institution Affiliation:** Rhode Island College (ID: 1746)
- **Institution Unit:** Computer Science

- **Curriculum Group:** Students conducting no more than minimal risk research
- **Course Learner Group:** Students - Class projects
- **Stage:** Stage 1 - Basic Course
- **Description:** This course is appropriate for students doing class projects that qualify as "No More Than Minimal Risk" human subjects research.

- **Report ID:** 19124183
- **Completion Date:** 04/13/2016
- **Expiration Date:** 04/12/2021
- **Minimum Passing:** 80
- **Reported Score*:** 100

REQUIRED AND ELECTIVE MODULES ONLY	DATE COMPLETED	SCORE
Rhode Island College (ID: 13859)	04/12/16	No Quiz
Belmont Report and CITI Course Introduction (ID: 1127)	04/12/16	3/3 (100%)
Students in Research (ID: 1321)	04/13/16	5/5 (100%)
Unanticipated Problems and Reporting Requirements in Social and Behavioral Research (ID: 14928)	04/13/16	5/5 (100%)

For this Report to be valid, the learner identified above must have had a valid affiliation with the CITI Program subscribing institution identified above or have been a paid Independent Learner.

CITI Program
Email: citisupport@miami.edu
Phone: 305-243-7970
Web: https://www.citiprogram.org

COLLABORATIVE INSTITUTIONAL TRAINING INITIATIVE (CITI PROGRAM)

COURSEWORK TRANSCRIPT REPORT**

** NOTE: Scores on this Transcript Report reflect the most current quiz completions, including quizzes on optional (supplemental) elements of the course. See list below for details. See separate Requirements Report for the reported scores at the time all requirements for the course were met.

- **Name:** Brian Faria (ID: 5474097)
- **Email:** bfaria_1722@email.ric.edu
- **Institution Affiliation:** Rhode Island College (ID: 1746)
- **Institution Unit:** Computer Science

- **Curriculum Group:** Students conducting no more than minimal risk research
- **Course Learner Group:** Students - Class projects
- **Stage:** Stage 1 - Basic Course
- **Description:** This course is appropriate for students doing class projects that qualify as "No More Than Minimal Risk" human subjects research.

- **Report ID:** 19124183
- **Report Date:** 04/13/2016
- **Current Score**:** 100

REQUIRED, ELECTIVE, AND SUPPLEMENTAL MODULES	MOST RECENT	SCORE
Students in Research (ID: 1321)	04/13/16	5/5 (100%)
Rhode Island College (ID: 13859)	04/12/16	No Quiz
Belmont Report and CITI Course Introduction (ID: 1127)	04/12/16	3/3 (100%)
Unanticipated Problems and Reporting Requirements in Social and Behavioral Research (ID: 14928)	04/13/16	5/5 (100%)

For this Report to be valid, the learner identified above must have had a valid affiliation with the CITI Program subscribing institution identified above or have been a paid Independent Learner.

CITI Program
Email: citisupport@miami.edu
Phone: 305-243-7970
Web: https://www.citiprogram.org