CS142 Coursework 2

Merge sort animation

Joe Moore, u1917702

May 2020

1 Introduction

The merge sort is a divide and conquer sorting algorithm, invented by John von Neumann in 1945. Conceptually it works by dividing an unsorted unordered list (containing n elements) into n sublists, such that the each contains only one element. From here since a list of one element is considered sorted the algorithm has given us n ordered lists, such sublists can be repeatedly merged to produce new sorted lists until there is only one sublist remaining. This sublist is a sorted version (of length n) of the original list. In sorting n objects, merge sort has an average and worst-case performance of $O(n \log n)$. It is one of the most efficient sorts with more simple approaches to sorting often having $O(n^2)$ efficiency.

For a large value of n and a randomly ordered list, merge sorts worst case is equal to or slightly smaller than $n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1$. However, its expected or average number of comparisons is $\cdot n$ smaller than this. Where $\alpha = -1 + \sum_{k=0}^{\infty} \frac{1}{2^k+1} \approx 0.2645$ [3]

This practically means that in its worst case the merge sort performs roughly 39% fewer comparisons [5] that a quick sort would in its average case. Naturally, the efficiency of the algorithm comes from it being of much higher complexity that some more orthodox sorting methods. As such I feel it will be a good algorithm to visualise as it can take some time to understand.

I settled finally on displaying the array as 7 boxes and when they were next to one another (with only a minuscule space between them) they were considered one array and when separate the opposite. I aim to animate the splitting up and putting back together of the array. I have also determined the visualisation must work for all possible values in the array. With the custom size of 7 I have chosen this means of 80,000 possible different visualisations that all need to work. Although my original design plans were different.

Hopefully, the visualisation should be able to inform someone unfamiliar with sorting methods altogether how a merge sort takes and unordered array and returns an ordered one. Allowing the user to change the values and see a different

animation unfold should help this visualisation to give a better understanding of the algorithm.

2 Design

Initially I had looked into using bar charts to show the size of the objects I had been sorting and showing them as seen in popular videos such as the one in figure 1. It is strong at showing the process occurring however only to someone with previous knowledge of a similar sort. Having spoken to 5 individuals throughout this process who have no prior knowledge of sorting their feedback for this idea was that they did not really understand what this meant

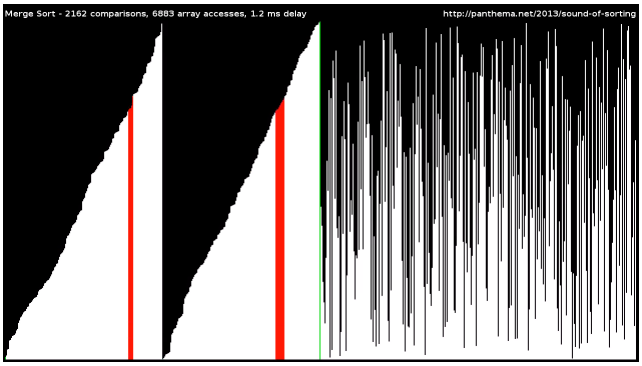

Figure 1: An example of a visualisation of a merge sort which denotes the size pf the values with bars of different heights [1]

.

All 5 in fact said they had no real idea what was occurring despite the fact they could see the data being sorted. I felt that size was something key to keep but position was also important too. It is very hard to see the actual comparisons and swaps occurring despite it being very easy to see which that the array was becoming more sorted.

I then decided to copy the more common style of having an array of roughly 7 size, which I would show being sorted with animations showing the swapping of the array positions. However, I thought, in order to present the data in a slightly unique way I would show data in boxes according to their sizes. My intention being that this would allow the user to see that the array was becoming more sorted. Such as in figure 2.

In his book "Visualising data"[2], Ben fry talks about the importance of not falling into the trap of what he calls the, "All-You-Can-Eat Buffet" [page 19]. Suggesting that data presented in less detail will almost certainly convey

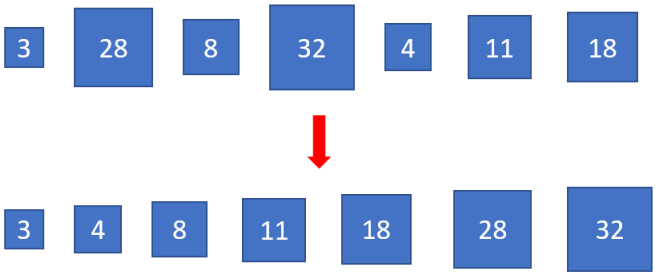

Figure 2: An example of the type of design work I did in trying to work out whether to pursue a design based off of size

more information. This certainly seemed true, since as 3 of the 5 people I asked questions to throughout the design process said they thought the larger sizes meant a larger memory size rather than just numbers. Hence, I concluded this would certainly be confusing, and so strayed away from this idea. Additionally, this was a weak idea as Mazza in his book "Visualising data"[7] he explains that area whilst good at conveying information it is not nearly as strong as position.

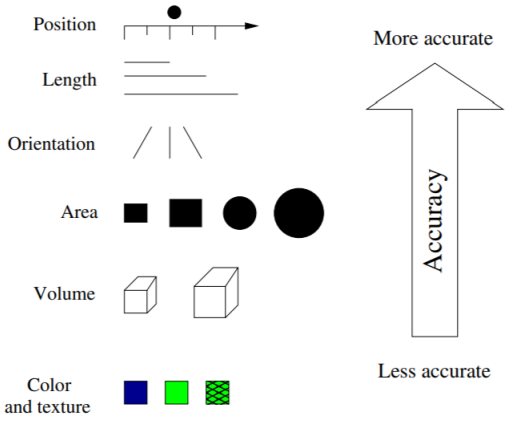

Figure 3: Accuracy in the perception of quantitative values for some graphical and spatial elements from "Introduction to Information Visualization" [7]

I decided then that I would allow the user to use their own values. This would allow the users to see the same algorithm under many different examples. It is also true that Hai-Ning Liang, Paul C. Parsons, Hsien-Chi Wu and Kamran Sedig, showed in their paper that "An Exploratory Study of Interactivity in Visualization Tools: 'Flow' of Interaction"[6] that interactivity in visualisations allowed for much better in take of knowledge. And although not as relevant,

this was also backed up by Christopher S. Gourley, in his research into interactivity used in the presentation and visualisation of multiresolution models [4]. Consequently, I have chosen to show the data in boxes of the same size but allowing the user to change the data used and for the animation to work for those data.

3 Implementation

3.1 Buttons

For the buttons I created a custom class, with the parameters of an integer array, two integers and a string. The array was the coordinates and size of the array. The first two values were its x and y position respectively and the second two were its width and length. The integers denoted the fill colour and font size with the text being what needed to be printed inside the button. These are set within the class constructor.

A Boolean procedure `hoverOverButton` was created which checked if the mouse was within the area the button occupied. This could therefore be called within `mousePressed` in order to detect whether a button had been pressed.

The class had one final procedure, `drawButton`, which drew the button to the user's screen. Within it, `hoverOverButton` was called, such that If the mouse were over the button then it would draw the button as grey rather than white. Each iteration of `draw()`, the `drawButton` procedure for each button was called in order to print all the buttons onto the screen.

3.2 User input

To take the user input when they click the "use own numbers" button I use two string values. One which is the current value the user is typing and the second being the saved values. Using a `keyPressed` procedure, every time a key is pressed, if the state means such that the user is currently on the screen asking them to input numbers, then the keys pressed by the user are added to the string. Finally, when the user hits enter the string is saved.

```
void keyPressed() {
  if(state == 2 && inputNumbers==true){
    // If the return key is pressed save the string and clear it
    if (key == '\n' ) {
      confirmedUserInput = userInput;
      // it is also cleared by setting it equal to ""
      userInput = "";
    } else {
      // every character is added to the end of the variable.
      userInput = userInput + key;
    }
```

```
  }
}
```

The program then checks it contains no letters and if it does not then that value is converted to an integer and saved. The process repeats till the program has 7 integers.

3.3 Animation

The animation was controlled by an integer variable, `stageOfVisualisation`. As this incremented so did the amount of the animation on the screen. The value of this variable was controlled by 2 buttons next and previous in the bottom of the screen during the animation. The drawing of the objects involved in the animation were split between 2 procedures. Everything stationary, (such as the original array of the individual sublists) were drawn within the merge procedure. This procedure simply went through a list of if clauses asking what the `stageOfVisualisation` was equal to or greater than in order to determine how much of the static part of the visualisation to draw. Naturally, merge is called within draw.

In instances whereby part of an array needs to be shown such as towards the end of the animation. Where the array is being sorted into its final state. On each even value of the stage of visualisation another stationary part of the array needs to be drawn. This is done by subtracting one from the stage if it is odd and then halving the result. Then for every value this is over half of the starting stage for this part draw an additional box.

```
int temp; //therefore we define a temp variable
if(stageOfVisualisation % 2 ==0) temp = (stageOfVisualisation-1)/2;
//minus one off of odd stages of visualisation
else temp = stageOfVisualisation/2; //and divide it by 2
if(stageOfVisualisation > 27) temp = 13;
//then otherwise set temp tp 13 so it doesn't draw more than 4 boxes at
    higher values
for(int i =0; i < temp-9; i++){
  fill(255);
  stroke(0);
  rect(171+(i*(boxwidth + 5)), 480, boxwidth, boxwidth);
  fill(0);
  text(firstHalf[i], 171+(i*(boxwidth + 5))+boxwidth/2, 480+boxwidth/2);
}
```

The moving parts of the animation are drawn within the draw function rather than a separate submethod. This is achieved with the use of two integer arrays, `movementStart` and `movementEndHeight`. At the beginning of the stage of each animation, 2 values of `movementStart` are set at indexes `x` and `x+1`, whereby `movementStart[x]` is the starting position, this is determined by call-

ing `largerArray()` which takes the arrays to determine which index is the next to move. From there it is easy to calculate which moves by increasing the first value by that `index * boxwdith`. `movementEndHeight` is also set to the final height the ox needs to end that. Then by drawing a box at `movementStart[x]` and its height and continually incrementing both until both get to `movementStart[x+1]` and `movementEndHeight[x]` we create the animation of the boxes moving in the correct way. After this the stage is incremented when the box reaches its destination and then the box is drawn permanently as the increase in the stage means that when merge is called that box is then drawn in the new `movementStart[x+1]` position.

4 Resulting Visualisation

Please see fig 4.

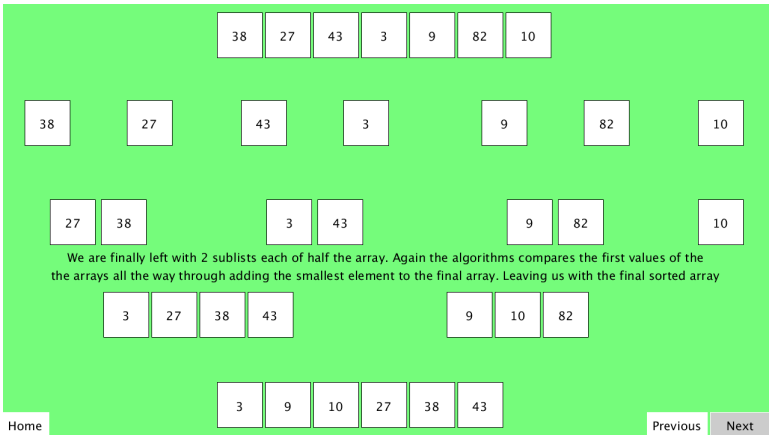

Figure 4: My visualisation at its largest extent

5 Conclusion

Overall, I am pleased with my visualisation. The fact it can visualise the merge sort for any set of 7 integers correctly is a great asset to it and definitely makes it more effective at conveying its information and so by extension does also make it a more effective visualisation. I am please with the simplicity of the design and the fact that it is clean, and the animation looks professional at all stages. All 5 of the people I spoke to said they felt it did successfully explain to them the process behind a merge sort and allowed for better understanding of the algorithm.

I do, however, have several criticisms of my visualisation that I would, given more time, likely change. Firstly, in order to save space on the window the first few stages of the algorithm disappear such that by the end it displays only the starting array and then next the 7 components. This means there is no point at which the whole process is displayed on the screen and this is something the people I surveyed said they felt could have improved it. Secondly due to the slightly disorderly nature of my code I found it increasing hard to highlight which of the elements were being compared. This was a sacrifice I chose to make in order to allow the visualisation to display all possible sorts and allow for user inputs. These flaws could have been overcome with more time and feel would really take the visualisation from effective to good.

Despite these flaws I am still incredibly pleased with the visualisation I have created, particularly due to its interactivity nature that allows for the user to enter their own numbers since this dramatically aids its ability to visualise the algorithm and impart knowledge.

References

[1] Timo Bingmann. 15 sorting algorithms in 6 minutes, 2013.

[2] Ben. Fry. *Visualising data*. O'Reilly Media, Inc, USA, 2008. ISBN 978-0-59651-455-6.

[3] Viliam Geffert and Jozef Gajdoš. Multiway in-place merging. In Mirosław Kutyłowski, Witold Charatonik, and Maciej Gebala, editors, *Fundamentals of Computation Theory*, page 133 to 144, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[4] Christopher S Gourley. *Pattern vector based reduction of large multimodal data sets for fixed rate interactivity during visualization of multiresolution models*. PhD thesis, Citeseer, 1998.

[5] Jyrki Katajainen and Jesper Larsson Träff. A meticulous analysis of mergesort programs. In Giancarlo Bongiovanni, Daniel Pierre Bovet, and Giuseppe Di Battista, editors, *Algorithms and Complexity*, page 217 to 228, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[6] Hai-Ning Liang, Paul C. Parsons, Hsien-Chi Wu, and Kamran Sedig. An exploratory study of interactivity in visualization tools: 'flow' of interaction. *Journal of Interactive Learning Research*, 21(1):5–45, January 2010.

[7] R. Mazza. *Introduction to Information Visualization*. Springer-Verlag London Limited, 2009. ISBN 978-1-84800-218-0.