Analyzing Complexity of Sorting Algorithms Using Empirical Modelling

Student ID: 1053046

Abstract

This paper demonstrates various methods to understand the concept of sorting, to apprehend the complexity of sorting algorithms and to identify the most efficient sorting method based on experimentation and learning from observation. The focus is primarily on the disparity between what is expected and what is observed when given a certain input, thereby formulating a process of trial and error which encourages incremental learning for a user and advocates further development of the model. The principles of empirical modelling are used to simulate sorting of an array using insertion sort and selection sort, and using a set of methods a comparison is done of the computational complexities which varies depending on the input array.

1 Introduction

Sorting is the process of arranging elements of a set in numerical or lexicographical order such that the relation '<=' is true for ever pair of consecutive elements in the set. A sorting algorithm is one that reorders the elements of a set in a sorted order. Sorting algorithms find use in a variety of practical applications such as search engine results ordering, shortest route algorithms, binary search algorithms, computational biology, data compression, etc.

Sorting is a fundamental problem in Computer Science and several approaches have been studied for optimization. The computational complexity of sorting algorithms is evaluated based on number of comparisons and swaps required to achieve the result, and is generally expressed in terms of set-size 'n' using the Big-O notation. Depending on the order of the elements in the set, sorting algorithms display good, average or worst case performance, i.e. the complexity varies for different sorting algorithms depending on whether the array is fully sorted, partially sorted or unsorted. In terms of Empirical Modelling, the array of numbers is an observable and the complexity performance, number of comparisons and number of swaps will be dependencies.

Empirical Modelling has been previously used as beneficial tool in educational systems to teach various concepts, and promote learning by allowing students to experiment with the dependencies in the model, and observe the state changes. Previous attempts to teach students about sorting focus on the algorithms and do not pay much attention to the calculation of computational complexity and its dependence on the input. This is a critical issue and students can easily misconstrue the evaluation, if unable to estimate the number of comparisons and swaps. The model proposes several methods to envision the sorting of an array and the complexity of a sorting algorithm and examines how an empirical approach is more effective while teaching the concept compared to traditional programming languages. The major advantage of incorporating EM principles to illustrate this concept is that the user is given the freedom to input an array, observe its sorting and its complexity analysis using different algorithms and then redefine the array (observable) to get a better a better grasp of the evaluation mechanism.

1.1 Sorting and EM concepts

In terms of EM, learning can be defined as a continuous scheme of three tasks - interacting with the model, observing model behaviour and redefining observables, as depicted in Figure 1. Knowledge gained from each stage can be used to understand the other two stages better. The overall goal of experimenting with the model is to gain significant knowledge, as theory is better understood only when applied. For instance, computer science students learn programming faster if they implement it on the computer, rather than being taught theoretically in class.

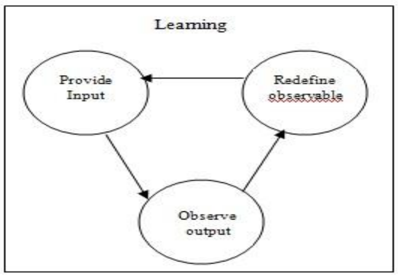


Figure 1: Learning in terms of EM

In [1] Beynon states that EM is concerned with building an interactive artefact whose current state at any time reflects that of a current situation and context. The computer teaching approach has advantages over the chalk-and-blackboard approach because the "current state" in progress can be viewed and this enables the learner to see the effects of the sorting process in operation and helps the learner to imagine the mechanisms by which this process is guided.

The proposed sorting model can be used as a tool for comparison of computational complexity of sorting algorithms, thereby suggesting the most efficient algorithm for a particular set of elements and utilising Empirical Modelling concepts for a contemporary purpose. This is termed as virtual experimentation [2], where the user is encouraged to interact with the model, provide any input, and observe the consequences. This facilitates better understanding of model behaviour.

The proposed project aims to model sorting of an array of 7 numbers, using two popular sorting algorithms – selection sort and insertion sort, and demonstrates the state change over every pass by keeping a count of the number of swaps. Also, it outlines a unique graphical representation of the sorting algorithms and calculates number of swaps and comparisons based on co-ordinate values in the graph, as described later in section 2.3. This is a theory which hasn't been discussed before and may prove useful for better understanding of insertion sort. It permits the student to change the array of values and observe the changing consequences in the complexity graph as well as in the computation of the number of swaps and comparisons over every pass of the data structure.

1.1.1 Sorting Algorithms

The model demonstrates two sorting algorithms – insertion sort and selection sort, and a brief overview is given this section.

Insertion Sort: This is a comparison sort in which an element is chosen and inserted in the right place in a sorted list. Initially, the sorted list is empty and all elements are in the unsorted list. Then, over every pass, one element is removed from the unsorted list and placed in the right position in the sorted list. The best case scenario is when the array is already sorted and hence there is only 1 comparison [Complexity: O(*n*)]. For an array of size *n*, the worst case scenario is an array sorted in reverse order where number of comparisons = *n*(n-1)/2*, [Complexity: O(*n*n*)].

Selection Sort: This is a simple comparison algorithm in which the array is scanned for the smallest element and it is swapped with the first element, and so on. For an array of size *n*, in the first pass, *n* elements are scanned to find the smallest and it is swapped with the first element. In the second pass, *(n-1)* elements are scanned and swapped with the second element, and so on. In total: *[(n-1)+(n-2)...+2+1]* comparisons, i.e. *n(n-1)/2* => Complexity O(*n*n*). One element is swapped over every pass and hence number of swaps = *(n-1).*

2 Overview of the Model

In order to understand the principles of EM effectively, the model is built in incremental stages following the convention of "learning by making" [3], where the author has cumulatively developed the model in stages based on input from the user, output from the model and observations.

2.1 Running the Model

The model was created using tkeden 1.73 for Windows[1]. It was tested on the Windows environment. To run the model, execute the file

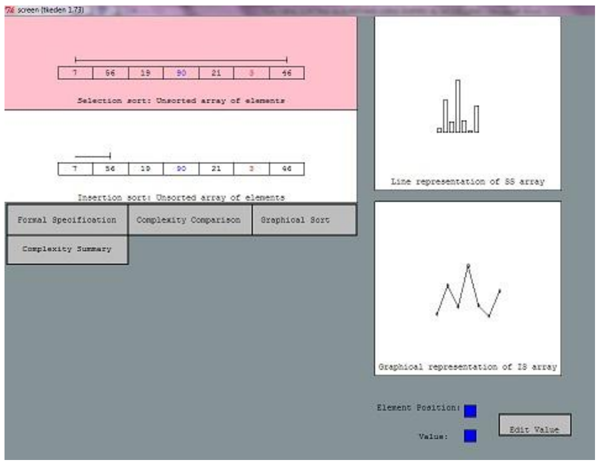
1. http://www2.warwick.ac.uk/fac/sci/dcs/research/em/software/eden/

‘Run.e’ which in turns loads the necessary files for complete model functionality.

2.2 Model Functionality

The model is developed in Eden, Scout and DoNaLD, which are primary definitive notations used in Empirical Modelling. Figure 2 depicts the initial state of the model. As observed, the sorting model is divided into 3 distinct viewports.

The left-hand side contains the two arrays to be sorted – the upper one using Selection Sort algorithm and the lower one using Insertion Sort algorithm, inspired from the existing model [4]. The content of the array is an observable and can be modified in two ways. First, via the Eden interpreter (by entering *val = [new array of 7 elements]* for the top array and *num = [new array of 7 elements]* for the lower array). Or by using the input window in the right bottom corner of the interface through which one can edit the element value in an array position ranging from 1-7.


screen (tkeden 1.73)
7
56
19
90
21
3
46
Selection sort: Unsorted array of elements
7
56
19
90
21
3
46
Insertion sort: Unsorted array of elements
Formal Specification
Complexity Comparison
Graphical Sort
Complexity Summary
Line representation of SS array
Graphical representation of IS array
Element Position:
Value:
Edit Value
Figure 2: Initial screenshot of the model

The top window on the right-hand side depicts the array to be sorted with selection sort, in a histogram graphical representation. Over every pass of sorting, the histogram changes to demonstrate the exchange of numbers. The lower window on the right-hand side is a graphical representation of the array of numbers to be sorted with insertion sort, having the element’s position in the array on the X-axis and value of the element on the Y-axis. The graph changes over every pass of insertion sort, and the final result is always an upward curve.

There are 4 buttons under the array window which serve different functionalities. The first button ‘Comments’ logs the flow of events including number of passes, elements to be swapped, etc and can be clicked anytime during the sorting flow to view history. The second button ‘Complexity Graph’ displays a graph plotted between the number of passes over the array and number of comparisons done, which in turn represents the computational complexity. The third button ’Graphical Sort’ demonstrates sorting in a pictorial format for better understanding. Insertion sort is represented as a graph plotted between array position and value of the element, whereas selection sort is represented as a histogram. Finally, the last button ‘Complexity Summary’ displays a table summary comprising of number of passes, comparisons and exchanges to evaluate computational complexity.

2.3 Dependencies

Empirical Modelling concepts are used to model the dependencies between the input array and computational complexity which facilitates understanding of the behaviour of sorting algorithms. The dependencies that the proposed model focuses on are:

1. For Insertion Sort: A graph is plotted between number of passes and number of comparisons (shown in Figure 3). Represent elements as a coordinate *{x,y}* where *x* is the array position of the element and *y* is its value. The number of swaps and exchanges required for element *{x,y}* is calculated by counting the number of other plotted points *{a,b}* such that *x>a* and *y<b*. So, by observing the co-ordinate values, number of comparisons = number of exchanges, which is dependent on the values of elements of the input array.

2. For selection sort: Number of comparisons is calculated as *(n-1)+(n-2).. +2+1*, where *n* is size of the array. Number of swaps = *n-1*, i.e. one exchange in each pass. So, number of comparisons and exchanges are dependent on size of array (shown in Figure 3).

3. The pictorial sorting graphs are dependent on the input array and rearrange themselves

according to changes over every pass of the array.

4. Sorting consists of an initial empty sorted list and an unsorted list (i.e. initial input array). Over each pass an element is removed from the unsorted array and placed in the sorted array. Hence, the range and set of elements vary after every pass over the unsorted array. Colour blue is used to identify the smallest element and red for the largest element in the unsorted array, which changes over each pass.

5. Complexity analysis table describing number of passes, number of swaps and number of comparisons is dependent on input array. Number of swaps decreases over every pass and is zero after the last pass when the array is sorted, because the table is dependent on the input array. Number of comparisons and swaps is computed by plotting the array elements and observing the co-ordinate values as described earlier.

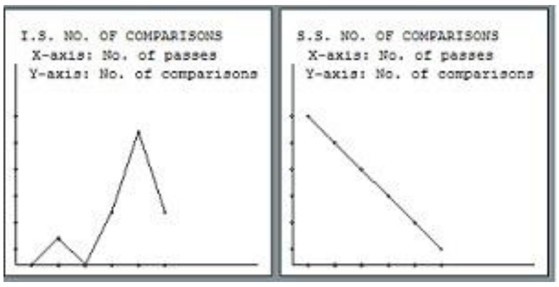


Figure 3: Complexity Efficiency graph

2.4 Model Construction and Challenges

The model was constructed in stages and was modified to incorporate improvements at every stage. Initially, the array window was created to include the two sorting arrays and the eden script *'selsort.e'* contains logic for implementation of insertion and selection sort. There is a range line over the array to indicate span of each pass. Attached to the array window, is the description window, which can be displayed by clicking on the 'Description' button and logs information of every pass.

In stage 2, the concept of sorting was simplified by depicting it pictorially. The DoNaLD script '*graph.d*' plots a graph between an array position *i* and value of *i* th element *num[i]*, and over every exchange the graph depicts the shift of *num[i]*. It has been observed for every element *num[i]* in the graph with co-ordinates *{i,num[i]}*, the number of insertion sort exchanges can be calculated by counting number of other elements with co-ordinates *{j,num[j]}* such that *i>j* and *num[i]<num[j]*. Another script '*linesort.d'* depicts sorting using histograms. The height of the bars is dependent on the value of the array elements and over every pass of selection sort the elements in the histogram rearrange themselves according to the array.

The aim of the model was to compare computational complexity based on number of comparisons and swaps. Hence, in the next stage a table was created for each sorting method by reusing the eden script '*grid.e'* from [5]. The table for insertion sort contains the number of comparisons and exchanges computed in the previous stage. For selection sort, the number of comparisons is computed in '*ss_tablevals.e'* by modelling the algorithm and incrementing a counter when a comparison takes place. The number of exchanges is fixed since array size is fixed.

Finally, an efficiency graph was included to compare the computational complexities. The initial plan was to create a generic efficiency graph, which was constant complexity *n*n* for selection sort and *n* for insertion sort for a sorted array, else *n*n*. However, this did not model the dependencies of the system. The graph was changed to plot the number of passes against the number of comparisons which functioned as a better dependency to judge complexity behaviour.

2.4.1 Challenges and Known Problems

1. The complexity graph for insertion sort depends on the input array *num* and hence at the end of 6 passes when *num* is sorted, the plot between the number of passes vs. number of comparisons falls on the X-axis since number of comparisons for a sorted array = 0 and so the graph looks empty.
2. Array size is fixed to 7 elements and this limits the complexity analysis of selection sort.

3. The complexity analysis table is dependent on the input array and displays the number of comparisons and swaps required for the current value of *num*. As the array gets sorted over every pass, the number of swaps and comparisons decreases in the table. And finally when the array is sorted, the table displays number of comparisons = number of swaps = 0.
4. The sorting does not work if a number that has been parsed and placed in the correct position is then changed in the middle of the sorting process.

3 Evaluation

3.1 Evaluating the Model

The aim of the Eden model was to sort an array using insertion sort and selection sort, compute the computational complexities and compare the same. Additions were made at each stage to the model to include the complexity analysis table, the pictorial representation of sorting and the complexity graph between number of passes and number of comparisons. Principles of EM were included to effectively model the complexity as an observable and its dependency on the input array, and to promote incremental learning for a user. Overall, I am happy with the way the model has turned out and I believe it can be used as a resourceful tool to teach sorting and compute the most efficient sorting algorithm for a particular input array.

3.2 Future Work

The current scope of the model is limited because sorting has been demonstrated only for an array of size seven and hence limits the scope of complexity analysis. The model can be developed to take in a dynamic input as array size which would enable a better understanding of the dependency of the number of swaps on array size for selection sort. The model can also be extended to include various sorting algorithms such as bubble sort, quick sort, merge sort, etc and computing the computational complexity and comparing it for each algorithm, would provide the use a single interface to decide the best mechanism to sort a certain input.

4 Conclusion

This paper proposes a mechanism for utilising empirical modelling techniques to promote experiential learning of sorting and complexity analysis of sorting algorithms. It can be concluded that using EM over procedural languages is more advantageous for learning. This is demonstrated by implementation of insertion sort and selection sort where the array of numbers is plotted on a graph and the number of comparisons can be computed by observing the co-ordinate values on the graph. The model is successful within its scope but should be extended to include the future work so as to function as a complete learning package of sorting and complexity analysis.

Acknowledgements

Special thanks to Meurig Beynon for his unconditional support, guidance and invaluable inputs during design and implementation of the model.

References

[1] Meurig Beynon, “Constructivist Computer Science Education Reconstructed”, *Available*: http://www.ics.heacademy.ac.uk/italics/vol8iss2/pdf/ItalicsVol8Iss2Jun2009Paper8.pdf

[2] An Educational Model for Teaching the Gas Laws, WEB-EM-4, *Available*: http://www2.warwick.ac.uk/fac/sci/dcs/research/em/publications/web-em/04/thegaslaws.pdf

[3] An Empirically Modelled Student Post Room, WEB-EM-2, *Available:* http://www2.warwick.ac.uk/fac/sci/dcs/research/em/publications/web-em/02/postroom.pdf

[4] Meurig Beynon, “Bubblesort”, *Available*: http://empublic.dcs.warwick.ac.uk/projects/bubblesortBeynon1998/

[5] Antony Harfield., “Agent-oriented Parser”, *Available:* http://empublic.dcs.warwick.ac.uk/projects/agentparserHarfield2003/