The Mathematics Enthusiast

Volume 16
Number 1 *Numbers 1, 2, & 3*

Article 15

2-2019

A Mathematical Analysis of Student-Generated Sorting Algorithms

Audrey Nasar

Recommended Citation
Nasar, Audrey (2019) "A Mathematical Analysis of Student-Generated Sorting Algorithms," *The Mathematics Enthusiast*: Vol. 16 : No. 1 , Article 15.
DOI: https://doi.org/10.54870/1551-3440.1460
Available at: https://scholarworks.umt.edu/tme/vol16/iss1/15

A Mathematical Analysis of student-generated sorting algorithms

Audrey A. Nasar[1]
Borough of Manhattan Community College at the City University of New York

Abstract: Sorting is a process we encounter very often in everyday life. Additionally it is a fundamental operation in computer science. Having been one of the first intensely studied problems in computer science, many different sorting algorithms have been developed and analyzed. Although algorithms are often taught as part of the computer science curriculum in the context of a programming language, the study of algorithms and algorithmic thinking, including the design, construction and analysis of algorithms, has pedagogical value in mathematics education. This paper will provide an introduction to computational complexity and efficiency, without the use of a programming language. It will also describe how these concepts can be incorporated into the existing high school or undergraduate mathematics curriculum through a mathematical analysis of student-generated sorting algorithms.
Keywords**:** Sorting; Algorithm; Complexity; Computational Thinking; Mathematics Education; Pre-calculus

I. Introduction

You've seen the headlines. It's pretty clear that we live in the era of Big Data. Beyond the collection of all this data, as a society we have become reliant on algorithms to sort the data and come up with something meaningful. Sorting, is a fundamental operation in computer science as well as a process we perform naturally in everyday life. Children are exposed to basic sorting activities from an early age, with toys such as the "Russian nesting doll", where dolls of decreasing size are placed one inside the other, and the "ring pyramid", where different sized rings are placed on a center pole from largest to smallest to build a pyramid. The general value of sorting is based on the fact that sorted

[1] anasar2@gmail.com

The Mathematics Enthusiast, **ISSN 1551-3440, vol. 16, nos.1,2&3**, pp.315-330

data are much easier to maneuver. If a phone book listed names at random rather than alphabetically, the prospect of finding a desired name would be a rather daunting task.

Given the impact of computers and computing on almost every aspect of society, the ability to develop, analyze, and implement algorithms is gaining more focus. A primary goal of mathematics education is to prepare students to be flexible problem solvers. The study of algorithms and algorithmic thinking contributes to the understanding of problem solving techniques and therefore has pedagogical value. According to Knuth (1974), a person does not really understand something until he teaches it to someone else. He goes on to clarify that a person does not really understand something until he can teach it to a computer, that is, express it as an algorithm. The mathematical ideas behind the design, construction and analysis of algorithms, are important for students' mathematical education. Furthermore, discovering and exploring algorithms can help students see mathematics as a meaningful and creative subject.

In secondary school, algorithms are usually restricted to the computer science curriculum and as a result, the important relationship between mathematics and computer science is often overlooked (Henderson, 1992). An algorithm is a mathematical object. The program, on the other hand, depends on the computer and/or the programming language used. Gal-Ezer and Zur (2004) found that the study of algorithms gives the learner insight into the problems involved by providing techniques for solutions that are independent of programming languages. Hence, if we can describe algorithms without having to rely on a programming language this gives us the opportunity to focus on their mathematical characteristics which could be a valuable addition to a student's secondary mathematics education. Teaching algorithms in high school would afford teachers and

students the opportunity to "apply the mathematics they know to solve problems arising in everyday life, society and the workplace" (CCSSI, 2010). Furthermore, since students are already comfortable with the process of sorting, sorting algorithms can serve as an entry point for teaching algorithmic thinking.

There have been several studies in computer science education that highlight methods used to teach sorting algorithms at the secondary level, including using a hands-on approach, flow-charts, and computer animations (Bernat, 2014; Végh, & Stoffová, 2017), mobile device apps (Boticki, Barisic, Martin & Drljevic, 2013), and a carefully designed web-based environment (Kordaki, Miatidis & Kapsampelis, 2008). While informative, these studies do not emphasize the mathematical characteristics of the sorting algorithms. Natov (2009) compares the complexities and running times of two sorting algorithms in a paper geared towards instructors of discrete mathematics and algorithms. Although he describes a mathematical analysis of complexity, he assumes the reader to be familiar with more advanced computer science concepts such as Big O notation and running time.

Lovász (2013) writes: "an important task for mathematics educators of the near future (both in college and high school) is to develop a smooth and unified style of describing and analyzing algorithms. A style that shows the mathematical ideas behind the design; that facilitates analysis; that is concise and elegant would also be of great help in overcoming the contempt against algorithms that is still often felt both on the side of the teacher and of the student" (pg. 7). This paper will address this need by providing a framework with which to perform a mathematical analysis of the complexity and efficiency of sorting algorithms without the use of a programming language, thus

allowing for a more seamless integration into the existing high school or undergraduate mathematics curriculum.

The observations made in this paper are based on a two-week mini-course on algorithms that was taught by the author to a class of ten high school students. All ten students had previously taken calculus, however some experience with precalculus would have been sufficient. The mini-course introduced students to the concept of complexity and then exposed them to several novel algorithmic problems, including sorting. They were asked to generate and analyze their own algorithms and then determine which was more efficient. This paper will follow the same format as the mini-course, starting with an introduction to the concept of complexity, as it was explained to the high school students, followed by a mathematical analysis of two of the algorithms that the students generated for the sorting problem.

III. Complexity

The complexity, or computational difficulty, of an algorithm estimates how many computations are needed to solve the algorithmic problem. Proulx (1997) found that the ability to measure and interpret complexity in addition to a good sense of scale makes students aware of the fact that some problems are indeed difficult, while many other seemingly complex problems can be solved rather easily. To see both the power and the limitations of computers, an understanding of how the complexities of different algorithms compare is necessary. In particular, students need to understand that some problems cannot be solved efficiently. Although technology may change the relative importance of individual algorithms, the mathematical ideas behind the design,

construction, and analysis of algorithms and the experience of applying some of these ideas to devise and improve existing algorithms is of importance for students' long-term mathematical education and will never become obsolete. Lovász (1996) writes "complexity, I believe, should play a central role in the study of a large variety of phenomena, from computers to genetics to brain research to statistical mechanics. In fact, these mathematical ideas and tools may prove as important in the life sciences as the tools of classical mathematics (calculus and algebra) have proved in physics and chemistry" (pg. 1).

Complexity can be measured by isolating a particular operation fundamental to the problem and then counting the number of times the algorithm performs this operation for an input of a given size. We will refer to this operation as an 'elementary operation'. This method provides criteria for comparing several algorithms for the same problem to determine which is the most efficient with respect to the chosen operation. This analysis would give students the opportunity to "construct and compare linear, quadratic, and exponential models and solve problems" (CCSSI, 2010).

The choice of elementary operation will vary depending on the nature of the problem that the algorithm is designed to solve (and in some cases may involve more than one operation). For sorting algorithms, we will count the number of comparisons needed to sort the elements. The number of elementary operations performed by an algorithm tends to grow with the size of the input. As such, it is traditional to describe the complexity of an algorithm as a function of n, its input size. The best notion for input size depends on the nature of the problem being studied. For sorting problems, the most natural measure of n is the number of items in the input sequence. When n is sufficiently

small, different algorithms may require the same number of elementary operations to solve a given problem. For example, two sorting algorithms for alphabetizing a list of names may only require one comparison when the input sequence has length n=2. However, as n increases, one of the algorithms may perform significantly fewer comparisons than the other, and would therefore be considered more efficient. Maurer and Ralston (2004) note that as computers get faster, people use them on larger and larger problems, so if there are a number of competing algorithms to solve the same problem, it is important to know which algorithm is most efficient for large n. As such, the choice of an algorithm for small inputs is not critical.

In addition to input size, the measure of complexity should also reflect the structure of the input. Even for inputs of the same size, the number of elementary operations performed by an algorithm can vary depending on the specific input. For example, an algorithm for alphabetizing a list of names may require very little work if only a few of the names are out of order, but it may involve substantially more work if many of the names are out of order. It is worthwhile to look at algorithms whose complexity depends on the structure of the input as well as algorithms whose complexity is the same for all inputs of a given size. An algorithm is said to be 'oblivious' if its complexity is independent of the structure of the input and 'non-oblivious' if the complexity depends on the structure of the input (Libeskind-Hadas, 1998). For 'non-oblivious' algorithms, we differentiate between the worst-case, average-case, and best-case scenarios by defining a separate complexity function for each. Whereas, for 'oblivious' algorithms, it suffices to describe their complexity by a single function (as their worst-case, average-case, and best-case scenarios are all the same).

Definition 1: The worst-case complexity of an algorithm is the greatest number of operations needed to solve the problem over all inputs of size *n*.

Definition 2: The best-case complexity of an algorithm is the least number of operations needed to solve the given problem for all inputs of size *n*.

Definition 3: The average-case complexity which is the most typical of the three but usually much more difficult to compute, quantifies the algorithm's average performance over all possible inputs of the same size assuming all inputs are equally likely.

Given two algorithms with worst-case complexity functions $f(n)$ and $g(n)$ respectively, the algorithm with worst-case complexity function $f(n)$ is considered to be more efficient than the algorithm with worst-case complexity function $g(n)$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. This can also be explained without using a limit by comparing the degree of the leading terms (in the case of polynomial functions) or graphing both functions and looking at the end behavior. Note that in the case of 'oblivious' algorithms, we need not specify worst-case.

III. Student-generated sorting algorithms

After the concept of complexity was introduced, the author presented the sorting problem and encouraged the students to develop their own algorithms. The 'General Sorting Problem' can be described as follows:

The General Sorting Problem: For a given sequence of *n* distinct elements $\langle a_1, a_2, ..., a_n \rangle$ where each pair of elements can be ordered, the output is a reordering $\langle a'_1, a'_2, ..., a'_n \rangle$ of the given sequence such that $a'_1 \le a'_2 \le ... \le a'_n$.

In addition to developing their own algorithms, students were encouraged to answer the following questions:

(1) What "elementary operation" is used to solve the problem?

(2) Describe a function that gives the number of operations performed for a given input for your algorithm.

The first algorithm proposed by a student, commonly known as 'selection sort,' finds the smallest element in the sequence, followed by the next smallest element, and so on, until the entire sequence is sorted. It can be described as follows: for a given sequence $\langle a_1, a_2, ..., a_n \rangle$, to find the smallest element, a'_1, start by comparing the first and second elements in the sequence. Once the smaller of the two is established go on to compare it with the third element in the sequence. Once the smaller of the two is established go on to compare it with the fourth element in the sequence, and so on. After the last element in the sequence has been compared, this process will result in the identification of a'_1. Next, move a'_1 to the first position in the sequence shifting the remaining unsorted elements to the right. In order to find the second smallest element, a'_2, compare the second and third elements in the sequence. Once the smaller of the two is established go on to compare it with the fourth element in the sequence. Once the smaller of the two is established go on to compare it with the fifth element in the sequence, and so on. After the last element in the sequence has been compared, this process will result in the identification of a'_2. Next, move a'_2 to the second position in the sequence shifting the remaining unsorted elements

to the right. Continue this process until only one unsorted element remains. By default this element is a'_n and what results is the sorted sequence $\langle a'_1, a'_2, ..., a'_n \rangle$.

Example: Use 'selection sort' to sort the sequence of numbers $\langle 5,9,2,7,1 \rangle$. To find the smallest element in the sequence, a'_1, compare 5 and 9. Since 5 is less than 9 go on to compare 5 and to 2. Since 2 is less than 5 go on to compare 2 and 7. Since 2 is less than 7 go on to compare 2 and 1. Since 1 is less than 2 it must be the smallest element in the sequence, hence set a'_1=1. Next 1 is moved to the first position in the sequence, and the remaining elements are shifted to the right. Now consider finding the second smallest element in the sequence. Since 5 is less than 9 go on to compare 5 and 2. Since 2 is less than 5 go on to compare 2 and 7. Since 2 is less than 7 it must be the second smallest element in the sequence, hence a'_2=2. Next 2 is moved to the second position in the sequence, shifting the remaining elements to the right. Now to find the third smallest element in the sequence compare 5 and 9. 5 is less than 9 so go on to compare 5 and 7. 5 is also less than 7 and so it must be the third smallest element in the sequence, hence a'_3=5. Incidentally it is already in the third position so does not need to be moved. To find the fourth smallest element in the sequence, compare 9 and 7. 7 is less than 9 and so it must be the fourth smallest element in the sequence, hence a'_4=7. 7 is moved to the fourth position, shifting the remaining element to the right. This results in the sorted sequence $\langle 1,2,5,7,9, \rangle$.

After students successfully applied the algorithm to a sequence of numbers, they were asked to derive the complexity function. To find the complexity function they identified the elementary operation for this algorithm as making a comparison. Although the algorithm also required shifting elements, for simplicity only the comparisons were

counted. They observed that the number of comparisons used to order the sequence did not depend on the extent to which the elements were already ordered. Therefore, this algorithm could be classified as 'oblivious.' Next, the complexity function was defined as follows:

Let $f(n)$ represent the complexity function for 'selection sort.' Then $f(n)$ gives the number of comparisons necessary to sort a sequence of length n. Finding the smallest element a'_1 uses n-1 comparisons, finding the second smallest element a'_2 uses n-2 comparisons, and in general, finding the k^{th} smallest element a'_k uses n-k comparisons. Applying this general formula for k=n-1, finding a'_{n-1} would use n-(n-1) or 1 comparison, namely the comparison of the last two unsorted elements in the sequence. Once the smaller of the two is identified and placed in the second to last position, the entire sequence will be sorted. Therefore, sorting a sequence of length n uses a total of (n-1)+(n-2)+(n-3)+…+3+2+1= (n)(n-1)/2 comparisons. Now (n)(n-1)/2 is equivalent to $n^2/2 - n/2$, so we can write $f(n) = n^2/2 - n/2$. Note that in order to sort the sequence $\langle 5,9,2,7,1 \rangle$, a total of 10 comparisons were made. This result can be confirmed by plugging in n=5 to the complexity function $f(n) = n^2/2 - n/2$. 'Selection sort' can be described as having 'quadratic complexity' as its complexity function is a quadratic.

The second algorithm proposed by another student is commonly known as 'insertion sort.' It orders the first two elements, then incorporates the third element, then the fourth element, and so on until the entire sequence is ordered. It can be described as follows: for a given sequence, compare the first and second element in the sequence placing the smaller of the two in the first position and the larger in the second position. Then compare the third element with the element in the second position. If it is larger,

then place it in the third position but if it is smaller then go on to compare it to the first element in the sequence. If it is larger than the first element then place it in the in second position but if it is smaller then place it in the first position. Next compare the fourth element with the third element. If it is larger then place it in the fourth position but if it is smaller then go on to compare it to the third element, second element, and first element as necessary. Continue this process until all the elements in the sequence have been ordered.

*Example***:** Use 'insertion sort' to sort the sequence of numbers $\langle 5,9,2,7,1 \rangle$. First compare 5 and 9. Since 5 is less than 9 keep 5 in the first position and 9 in the second position. Next compare 2 and 9. Since 2 is less than 9 compare it to 5. Since 2 is less than 5 place it in the first position shifting 5 and 9 to the second and third position respectively. Next compare 7 to 9. Since 7 is less than 9 go on to compare 7 and 5. Since 7 is greater than 5 place it in the third position shifting 9 to the fourth position. Next compare 1 and 9. Since 1 is less than 9, compare it to 7. Since it is less than 7, compare it to 5. Since it is less than 5 compare it to 2. Since it is less than 2 place it in the first position shifting the remaining elements to the right. This results in the sequence $\langle 1,2,5,7,9 \rangle$.

After the students successfully applied the algorithm to a sequence of numbers, they moved on to analyzing its complexity. Although the algorithm also requires shifting elements, the students were advised only to count the comparisons. They observed that the number of comparisons used to order the sequence depended on the extent to which the elements were already ordered. In other words, unlike the previous algorithm, this algorithm was not 'oblivious.' After further analysis it became clear that the best-case

scenario would result if the sequence was already in order and the worst-case scenario would result if the sequence were in reverse order.

The complexity functions were defined as follow: Let $b(n)$ represent the best-case complexity function. The algorithm starts by comparing the first two elements, and then compares each of the remaining n-2 elements once each for a total of 1+(n-2) comparisons. Hence $b(n)=n-1$. Given the sequence $\langle 1,2,5,7,9 \rangle$ the algorithm would use 4 comparisons. Let $w(n)$ represent the worst-case complexity function. The algorithm would start by comparing the first two elements, and then comparing the third element with each of the first two elements and the fourth element with each of the first three elements and in general, the k^{th} element with each of the k-1 elements that precede it, for a total of (n-1)+(n-2)+(n-3)+…+3+2+1= (n)(n-1)/2 comparisons. Hence $w(n) = n^2/2 - n/2$ which is the same as the complexity function for 'selection sort.' Given the sequence $\langle 9,7,5,2,1 \rangle$ the algorithm would use 10 comparisons. The average case complexity function $a(n)$ is a lot more complicated and was not covered in class, but can serve as a deeper exploration. Note that in order to sort the sequence $\langle 5,9,2,7,1 \rangle$ a total of 9 comparisons, which is between the best case and worst case. 'Insertion sort' can be described as having 'quadratic complexity' in its worst case and 'linear complexity' in its best case. The students observed that while the worst-case complexity function for 'insertion sort' is the same as the complexity function for 'selection sort,' 'insertion sort' is more efficient because it will often use fewer operations than 'selection sort.'

While we did not have time during the mini-course to delve deeper into the mathematical analysis of sorting algorithms, some possible extensions are:

(1) To consider how the complexity would change if the elements in the sequence were not distinct.

(2) To describe the average-case complexity function for 'insertion sort.'

(3) To introduce 'merge-sort' and demonstrate why it is the most efficient of any sorting algorithm.

At the end of the mini-course, many students reflected that they enjoyed learning about sorting because it was very relevant to their lives. They also liked that there were so many possible solutions. In addition, because students generated their own algorithms, they were able to really take ownership over them and were more invested in the complexity analysis and determining which is more efficient. I don't think they would have been as interested had the algorithms been given. Furthermore, students were pleasantly surprised at how mathematical the analysis of sorting algorithms actually was. In conclusion, the sorting problem proved to be a rich introduction to the mathematical analysis of algorithms and was very accessible to students at the high school level.

References

Bernát, P. (2014). The Methods and Goals of Teaching Sorting Algorithms in Public Education. *Acta Didactica Napocensia*, 7(2): 1-9.

Boticki, I., Barisic, A., Martin, S., & Drljevic, N. (2013). Teaching and learning computer science sorting algorithms with mobile devices: A case study. *Computer Applications in Engineering Education*, 21(S1): E41-E50.

Common Core State Standards Initiative. (2010). Common core state standards for mathematics. Retrieved from http://www.corestandards.org/assets/CCSSI_Math%20Standards.pdf.

Gal-Ezer , J. and Zur, E. (2004). The Efficiency of Algorithms- Misconceptions. *Computers and Education*, 42(3): 215-226.

Henderson, P. B. (1992). Computer Science, Problem Solving, and Discrete Mathematics. In *Discrete Mathematics in the Schools*.

Knuth, D. E. (1974). Computer Science and its Relation to Mathematics. *The American Mathematical Monthly*, 81: 323-343.

Kordaki, M., Miatidis, M., & Kapsampelis, G. (2005). Multi Representation Systems in the Design of a Microworld for the Learning of Sorting Algorithms. In *CELDA* (pp. 363-366).

Kordaki, M., Miatidis, M., & Kapsampelis, G. (2008). A computer environment for beginners' learning of sorting algorithms: Design and pilot evaluation. *Computers & Education*, 51(2): 708-723.

Libeskin-Hadas, R. (1998) Sorting in Parallel. *The American Mathematical Monthly*, 105(3): 238-245.

Lovász, L. (1996). Information and complexity (how to measure them?). *The Emergence of Complexity in Mathematics, Physics, Chemistry and Biology, Pontifical Academy of Sciences*, 65-80.

Lovász, L. (2013). Trends in mathematics: How they could change education. In *Notices of the International Congress of Chinese Mathematicians* (Vol. 1, No. 2, pp. 79-84). International Press of Boston.

Maurer, S. B. and Ralston, A. (2004). *Discrete Algorithmic Mathematics* (3rd Edition). Massachusetts: A K Peters, Ltd.

Natov, J. (2009). Running Times versus the Master Theorem. *Mathematics and Computer Education*, 43(3): 231.

Proulx, V. K. (1992). The Role of Computer Science and Discrete Mathematics in the High School Curriculum. In *Discrete Mathematics in the Schools*.

Végh, L., & Stoffová, V. (2017). Algorithm Animations for Teaching and Learning the Main Ideas of Basic Sortings. *Informatics in Education*, 16(1): 121-140.