The Mathematics Enthusiast

Volume 13
Number 3 *Number 3*

Article 4

8-2016

The history of Algorithmic complexity

Audrey A. Nasar

The history of Algorithmic complexity

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

ABSTRACT: This paper provides a historical account of the development of algorithmic complexity in a form that is suitable to instructors of mathematics at the high school or undergraduate level. The study of algorithmic complexity, despite being deeply rooted in mathematics, is usually restricted to the computer science curriculum. By providing a historical account of algorithmic complexity through a mathematical lens, this paper aims to equip mathematics educators with the necessary background and framework for incorporating the analysis of algorithmic complexity into mathematics courses as early on as algebra or pre-calculus.

Keywords: Algorithm, Complexity, Discrete Mathematics, Mathematics Education

[1] anasar2@gmail.com

The Mathematics Enthusiast, **ISSN 1551-3440, vol. 13, no.3**, pp. 217 – 242

I. Introduction

Computers have changed our world. The exploding development of computational technology has brought algorithms into the spotlight and as a result, the analysis of algorithms has been gaining a lot of focus. An algorithm is a precise, systematic method for solving a class of problems. Algorithmic thinking, which is a form of mathematical thinking, refers to the thought processes associated with creating and analyzing algorithms. Both algorithms and algorithmic thinking are very powerful tools for problem solving and are considered to be key competences of students from primary to higher education (Cápay and Magdin, 2013). An integral component of algorithmic thinking is the study of algorithmic complexity, which addresses the amount of resources necessary to execute an algorithm. Through analyzing the complexity of different algorithms, one can compare their efficiencies, their mathematical characteristics, and the speed at which they can be performed.

The analysis of algorithmic complexity emerged as a scientific subject during the 1960's and has been quickly established as one of the most active fields of study. Today, algorithmic complexity theory addresses issues of contemporary concern including cryptography and data security, parallel computing, quantum computing, biological computing, circuit design, and the development of efficient algorithms (Homer and Selman, 2011). 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).

High School mathematics courses tend to be focused on preparing students for the study of calculus. Sylvia da Rosa (2004) claims that this has led students to the common misconception that mathematics is always continuous. In addition, few students are excited by pure mathematics. For most students to begin to appreciate mathematics, they have to see that it is useful. Incorporating the topic of 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). Bernard Chazelle, a professor of computer science at Princeton University, in an interview in 2006 said on the future of computing: "The quantitative sciences of the 21st century such as proteomics and neurobiology, I predict, will place algorithms rather than formulas at their core. In a few decades we will have algorithms that will be considered as fundamental as, say, calculus is today."

Although there have been papers and books written on the history of algorithms, the history of algorithmic complexity has not been given nearly as much attention (Chabert, 1999; Cormen, et al., 2001). This paper will address this need by providing a history of algorithmic complexity, beginning with the text of Ibn al-majdi, a fourteenth century Egyptian astronomer, through the 21st century. In addition, this paper highlights the confusion surrounding big-O notation as well as the contributions of a group of mathematicians whose work in computability theory and complexity measures was critical to the development of the field of algorithmic complexity and the development of the theory of NP- Completeness (Knuth, 1976; Garey and Johnson, 1979). In effort to provide educators with context for which these topics can be presented, the problem of finding the maximum and minimum element in a sequence is introduced, along with an

analysis of the complexity of two student-generated algorithms.

II. History of Algorithmic Complexity

Historically, an interest in optimizing arithmetic algorithms can be traced back to the Middle Ages (Chabert, 1999). Methods for reducing the number of separate elementary steps needed for calculation are described in an Arabic text by Ibn al-Majdi, a fourteenth century Egyptian astronomer. He compared the method of translation, which was used to find the product of two numbers, as well as the method of semi-translation, which was used only for calculating the squares of numbers. Based on Ibn al-Majdi's writings, if one were to use the method of translation to find the square of 348, for example, it would require nine multiplications. However, if one were to use the method of semi-translation to calculate the same product, it would require only six multiplications. In general, when squaring a number of n digits, the translation method takes n^2 elementary multiplications while the semi-translation method takes $n(n\text{-}1)/2$ elementary multiplications. In considering the number of steps, it is important to note that Ibn al-Majdi counted neither the number of elementary additions nor the doublings (Chabert, 1999).

Attempts to analyze the Euclidean algorithm date back to the early nineteenth century. In his well-known 1844 paper, Gabriel Lamé proved that if $u > v > 0$, then the number of division steps performed by the Euclidean algorithm $E(u,v)$ is always less than five times the number of decimal digits in v (Shallit, 1994). Although Lamé is generally recognized as the first to analyze the Euclidean algorithm, several other mathematicians had previously studied it. Shallit (1994) notes that sometime in between 1804 and 1811, Antoine-Andre-Louis Reynaud proved that the number of division steps performed by the

Euclidean algorithm, E(*u*,*v*), is less than or equal to *v*. Several years later he refined his upper limit to *v*/2 (which Lamé proved to be false).

Schreiber (1994) notes that algorithmic thinking has been a part of the study of geometric constructions since Antiquity. He explains that "the first studies on the unsolvability of problems by special instruments (i.e. by special classes of algorithms) and the first attempts to measure, compare, and optimize the complexity of different algorithms for solving the same problem" were in the field of geometric constructions (Schreiber, 1994, p. 691). These attempts can be seen in Émil Lemoine's 1902 'Géométrographie' (Schreiber, 1994).

In 1864, Charles Babbage predicted the significance of the study of algorithms. He wrote, "As soon as an Analytical Engine [i.e. general purpose computer] exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise - By what course of calculation can these results be arrived at by the machine in the shortest time (Knuth, 1974, p. 329)?" The time taken to execute an algorithm, as Babbage predicted, is an important characteristic in quantifying an algorithm's efficiency.

In 1937, Arnold Scholtz studied the problem of optimizing the number of operations required to compute x^n (Kronsjö, 1987). In order to compute x^{31}, for example, it can be done in seven multiplications: $x^2, x^3, x^5, x^{10}, x^{20}, x^{30}, x^{31}$. However, if division is allowed, it can be done in six arithmetic operations. Kronsjö (1987) notes that the problem of computing x^n with the fewest number of multiplications is far from being solved.

In the 1950's and 1960's, several mathematicians worked on optimization problems similar to that of Scholtz and Ibn al-Majdi. In order to evaluate a polynomial function $f(n) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$ at any point, it requires at most *n* additions and 2*n*-1 multiplications. Horner's method, which involves rewriting the polynomial in a different form, only requires *n* multiplications and *n* additions, which is optimal in the number of multiplications under the assumption that no preprocessing of the coefficients is made (Kronsjö, 1987). In order to multiply two 2x2 matrices using the standard method, it requires eight multiplications and four additions[2]. In 1968, V. Strassen introduced a 'divide-and-conquer' method that reduced the number of multiplications to seven at the cost of fourteen more additions (Wilf, 2002). Generalizing his method to the multiplication of two *n*x*n* matrices (where *n* is an even number) reduces the number of multiplications from n^3 multiplications to $7n^3/8$ multiplications (Kronsjö, 1987). It is important to note that if additions are taken into consideration, Strassen's method is less efficient than the standard method for small inputs (Sedgewick, 1983). In order to compare different methods for solving a given problem, one needs to decide which operations should be counted and how they should be weighted. This decision, which is highly contingent on the method of implementation, is extremely important when comparing the efficiency of different algorithms. For example, computers typically consider multiplication to be more complex than addition. As a result, reducing the number of multiplications at the expense of some extra additions was preferable.

The introduction of the Turing machine in 1937 led to the development of the theory explaining which problems can and cannot be solved by a computer. This brought

[2] See Appendix for a more detailed comparison of the standard algorithm and Strassen's algorithm for matrix multiplication.

about questions regarding the relative computational difficulty of computable functions, which is the subject matter of computational complexity (Cook, 1987). Michael Rabin was one of the first to address what it means to say that a function *f* is more difficult to compute than a function *g* in his 1960 paper *Degree of Difficulty of Computing a Function and Hierarchy of Recursive Sets*. Juris Hartmanis and Richard Stearns, in their 1965 paper *On the Computational Complexity of Algorithms*, introduced the notion of complexity measure defined by the computation time on multitape Turing machines. Around the same time, Alan Cobham published *The Intrinsic Computational Difficulty of Functions*, which discussed machine-independent theory for measuring computational difficulty of algorithms. He considered questions such as whether multiplication is harder than addition and what constitutes a "step" in computation (Cook, 1987). These fundamental questions helped shape the concept of computational complexity. The field of computational complexity, which may be credited to the pioneering work of Stephen Cook, Richard Karp, Donald Knuth, and Michael Rabin, categorizes problems into classes based on the type of mathematical function that describes the best algorithm for each problem.

Several of the early authors on computational complexity struggled with the question of finding the most appropriate method to measure complexity. Although most agreed on computational time and space, the methods posed for these measurements varied. Kronsjö (1987) and Rosen (1999) define the time complexity of an algorithm as the amount of time used by a computer to solve a problem of a particular size using the algorithm. They define the space complexity of an algorithm as the amount of computer memory required to implement the algorithm. Thus, space and time complexity are tied

to the particular data structures used to implement the algorithm. Although the concept of complexity is often addressed within the context of computer programs, given the variability in the space and speed of computers, a measurement that is independent of the method of implementation is often preferred.

A useful alternative, Kronsjö (1987) and Rosen (1999) note, is a mathematical analysis of the intrinsic difficulty of solving a problem computationally. They describe the computational complexity of an algorithm as the computational power required to solve the problem. This is measured by counting the number of elementary operations performed by the algorithm. The choice of elementary operation or operations will vary depending on the nature of the problem that the algorithm is designed to solve. It should however, be fundamental to the algorithm; for example, the number of real number multiplications and/or additions needed to evaluate a polynomial, the number of comparisons needed to sort a sequence, the number of multiplications needed to multiply two matrices. As long as the elementary operations are chosen well and are proportional to the total number of operations performed by the algorithm, this method will yield a consistent measurement of the computational difficulty of an algorithm which can be used to compare several algorithms for the same problem. Generally, by analyzing the complexity of several candidate algorithms for a problem, the most efficient one can be identified.

The number of elementary operations performed by an algorithm typically grows with the size of the input. Therefore, it is customary to describe the computational complexity or simply, complexity of an algorithm as a function of n, the size of its input (Cormen, et al., 2001). Additionally, the choice of n depends on the context of the

problem for which the algorithm is being used. For sorting and searching problems, the most natural measure of *n* is the number of items in the input sequence. If the problem were to evaluate a polynomial, *n* would be better suited as the degree of the polynomial. If it were to multiply square matrices, on the other hand, *n* would represent the degree of the matrices. When *n* is sufficiently small, two algorithms may solve a problem using the same number of elementary operations. As *n* increases, one of the algorithms may perform significantly fewer elementary operations than the other, which would identify it as being more efficient. As such, it is important to consider the behavior of algorithms for large values of *n*. This can be done by comparing the growth rates of the complexity functions for each algorithm (to be discussed shortly).

For certain algorithms, even for inputs of the same size, the number of elementary operations performed by the algorithm can vary depending on the structure of the input. For example, an algorithm for alphabetizing a list of names may require a small number of comparisons if only a few of the names are out of order, and more comparisons if many of the names are out of order. We can classify algorithms according to whether or not their complexity depends on the structure of the input. An algorithm is said to be 'oblivious' if its complexity is independent of the structure of the input (Libeskind-Hadas, 1998). For 'non-oblivious' algorithms (whose complexity depends on the structure of the input) one must differentiate between the worst-case, average-case, and best-case scenarios by defining a separate complexity function for each case. 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).

The worst-case complexity of an algorithm is the greatest number of operations needed to solve the problem over all inputs of size n. The best-case complexity of an algorithm is the least number of operations needed to solve the problem over all inputs of size n. The average-case complexity of an algorithm, is the average number of operations needed to solve the problem over all possible inputs of size n assuming all inputs are equally likely (Maurer and Ralston, 2004).

III. Big-O Notation

In order to compare the efficiencies of competing algorithms for a given problem, it is necessary to consider the number of operations performed by each algorithm for large inputs. This is done by classifying and comparing the growth rates of each algorithm's complexity function. Big-O notation, which was introduced by the German mathematician Paul Bachmann in 1894, is used extensively in the analysis of algorithms to describe the order of growth of a complexity function (Rosen, 1999). In particular, big-O gives an upper bound on the order of growth of a function.

Definition 1: "big-O": Let $f(n)$ and $g(n)$ be two positive valued functions, we say that $f(n) = O(g(n))$, if there is a constant c such that $f(n) \leq cg(n))$ for all but finitely many n.

If $f(n) = O(g(n))$ we say "$f(n)$ is $O(g(n))$." A geometric representation of $f(n) = O(g(n))$ can be seen in the figure below.

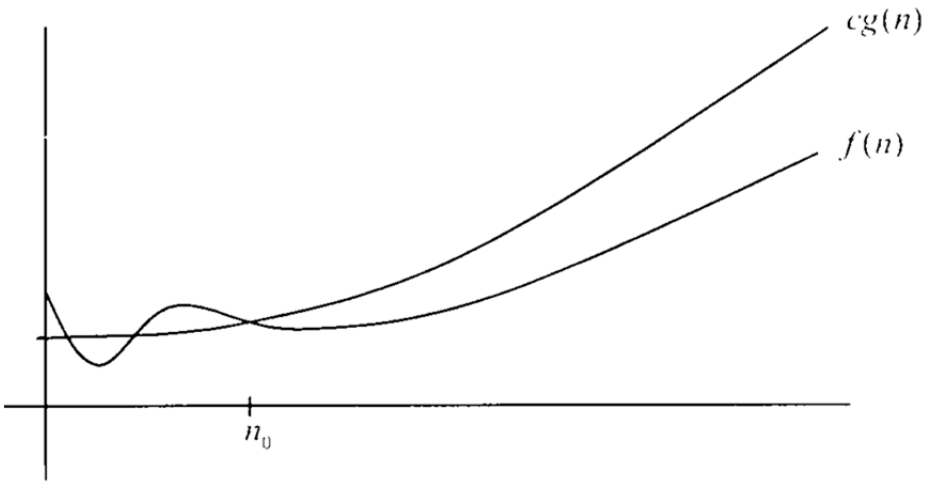

Figure 1

Knuth (1976) mentions that people often misuse big-O notation by assuming that it gives an exact order of growth; using it as if it specifies both a lower bound as well as an upper bound. Motivated by the misuse of big-O notation, Knuth introduced the big-Omega and big-Theta notations in the 1970s (Knuth, 1976). Big-Omega (denoted by the symbol Ω) provides a lower bound on the order of growth of a function, while big-Theta (denoted by the symbol Θ) provides both an upper and lower bound on the order of growth of a function. One algorithm is considered to be more efficient than another algorithm if its worst-case-complexity function has a lower order of growth. Unlike big-O and big-Omega, big-Theta gives an exact order without being precise about constant factors and is often the most appropriate of the three for comparing the efficiencies of different algorithms.

Definition 2 "big-Theta": Let $f(n)$ and $g(n)$ be two positive valued functions. We say that $f(n)= \Theta(g(n))$ if there is a constant $c \neq 0$ such that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \qquad (1)$$

or if there are positive constants c_1 and c_2 such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \qquad (2)$$

for all but finitely many *n*.

Note that condition (1) is a special case of condition (2) in that for condition (1) to be satisfied the function must have a limit, whereas for condition (2) the function need only be bounded above and below. Condition (1) is included because it is simpler and is sufficient to analyze almost all complexity functions that arise when studying algorithms. A geometric representation of $f(n) = \Theta(g(n))$ can be seen in the figure below.

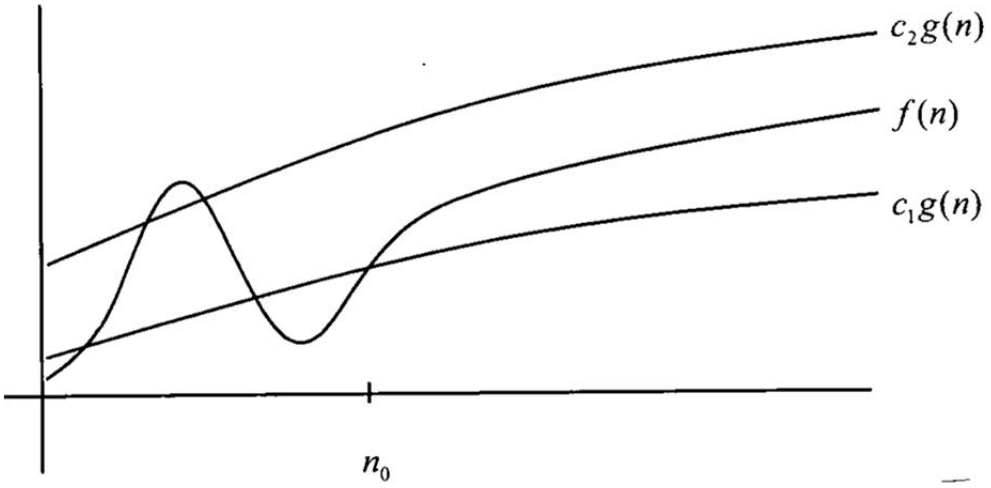

Figure 2

Definition 3 "big-Omega": Let $f(n)$ and $g(n)$ be two positive valued functions, we say that $f(n) = \Omega(g(n))$, if there is a constant *c* such that

$$f(n) \geq cg(n))$$

for all but finitely many *n.*

If $f(n) = \Omega\ (g(n))$ we say "$f(n)$ is $\Omega(g(n))$." A geometric representation of $f(n) = \Omega(g(n))$ can be seen in the figure below.

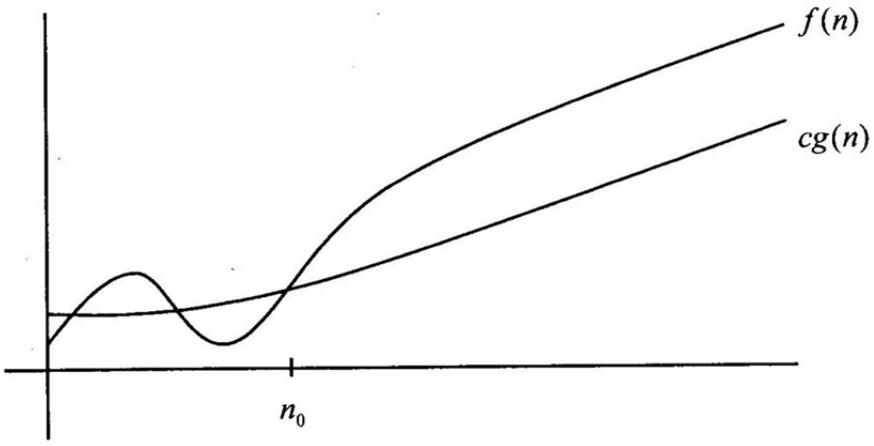

Figure 3

Of the three notations, big-O is the most widely used to classify the complexity of algorithms. In particular, an algorithm with an input size of n is said to have constant time complexity if its complexity is O(1), in other words, if its complexity is bounded above by a constant. Furthermore, an algorithm with an input size of n is said to have logarithmic time complexity if its complexity is O(log$n$), linear time complexity if its complexity is O(n), $n$log$n$ time complexity if its complexity is O($n$log$n$), polynomial time complexity if its complexity is O(n^k), exponential time complexity if its complexity is O(b^n) where b>1 and factorial time complexity if its complexity is O(n!).

An advantage of using big-O instead of big-Theta or big-Omega, is that when big-O is used to describe the worst-case complexity function of an algorithm, it also gives an upper bound on the complexity of the algorithm for *every* input. For example, if the worst-case complexity function of an algorithm is O($g(n)$), we can conclude that the complexity of the algorithm is O($g(n)$). On the other hand, if the worst-case complexity function of an algorithm is Θ($g(n)$), this does not does not imply an Θ($g(n)$) bound on the complexity of the algorithm for *every* input of size n. If however, the algorithm is

'oblivious' (in that its complexity is the same for all inputs of a given size), big-Theta can be used to describe the complexity for *every* input. When Ω-notation is used to describe the best-case complexity function of an algorithm, it also gives a lower bound on the complexity of the algorithm. For example, if the best-case complexity function of an algorithm is $\Omega(g(n))$ this lower bound also applies to its complexity on *every* input. With big-Theta, however, if we find that the best-case complexity function of an algorithm is $\Theta(g(n))$, this does not does not imply a $\Theta(g(n))$ bound on the complexity of the algorithm for *every* input.

Maurer and Ralston (2004) and Knuth (1976) note that some textbooks mistakenly treat big-O and big-Theta as though they are the same. This has led to the misconception that algorithms can be compared by looking at what they are big-O of, which is not always the case. Suppose that one algorithm for a given problem has worst-case complexity function $f(n)=n$ and another algorithm for the same problem has worst-case complexity function $g(n)=n^2$. We can say that both $f(n)$ and $g(n)$ are $O(n^2)$. If we compared what the functions are big-O of, there is no indication as to which algorithm is more efficient. On the other hand if we considered using big-Theta to describe the growth of the complexity function we would find that $f(n)= \Theta(n)$ and $g(n)= \Theta(n^2)$. By using big-Theta we can observe that one algorithm (namely the one with worst-case complexity function $f(n)$) is more efficient than the other. As this example illustrates, the issue that arises in using big-O for comparing worst-case complexity functions lies in the fact that big-O notation only provides an upper bound which is not uniquely defined. Big-Theta, on the other hand, provides both an upper and lower bound. When we show that $f(n)= \Theta(g(n))$ it follows that $f(n)=O(g(n))$ but the converse may be false. In other words, if

$f(n)=O(g(n))$ it does not necessarily follow that $f(n)= \Theta(g(n))$. Hence if one wants to compare the worst-case complexity function of different algorithms to see which is more efficient, big-Theta would be the most appropriate. If, however, there is only an interest in providing an upper bound on the complexity of an algorithm, it suffices to describe its worst-case complexity function using big-O notation. Note that in the case of 'oblivious' algorithms where the complexity is the same for all inputs of size n, worst-case does not need to be specified.

In order to describe the complexity of an algorithm one can show that the worst-case complexity function of the algorithm is $O(g(n))$ or that worst-case complexity function of the algorithm is $\Theta(g(n))$. In either case it can be said that the complexity of the algorithm is $O(g(n))$). Note that the second case follows from the fact that if a function $f(n)$ is $\Theta(g(n))$ then $f(n)$ is $O(g(n))$.

IV. NP-Completeness

An important concept that was developed by 1965 was the identification of the class of problems solvable by algorithms with polynomial time complexity. The distinction between algorithms with polynomial and exponential time complexities was made as early as 1953 by Von Neumann, but the class of problems was not defined formally until Cobham in 1964 (Cook, 1987). Exponential time algorithms often perform exhaustive searches, whereas polynomial time algorithms rely on deeper insight into the structure of a problem. There is wide agreement that a problem has not been "well-solved" until a polynomial time algorithm is known for it (Garey and Johnson, 1979). A problem that is solvable using an algorithm with polynomial time complexity is called 'tractable' or 'easy', whereas problems that cannot be solved using an algorithm with

polynomial time complexity are called ‘intractable’ or ‘hard’ (Rosen, 1999). The first examples of ‘intractable’ problems were obtained in the early 1960’s, as part of work on complexity “hierarchies” by Hartmanis and Stearns (Garey and Johnson, 1979). By the late 1960’s, a sizable class of practical problems that had not so far been solved with polynomial time algorithms was developed. These problems, which were known as NP-Complete, are believed to have the property that no algorithm with polynomial time complexity solves them, but that once a solution is known, it can be checked in polynomial time complexity. Additionally, these problems share the property that if any of them can be solved by an algorithm with polynomial time complexity, then they can all be solved by algorithms with polynomial time complexity (Rosen, 1999). To understand this difference, consider the problem of finding a solution to a Diophantine equation. There is no general method for finding a solution, however, it is relatively easy to check a proposed solution (Karp, 1987).

The foundations for the theory of NP-Completeness were laid in Cook’s 1971 paper, entitled *The Complexity Theorem Proving Procedures.* Subsequently, Karp presented a collection of results in his influential 1972 paper, which showed that twenty one important problems are NP-Complete. This generated tremendous interest in the notion of NP-Completeness. The question of whether or not the NP-Complete problems are ‘intractable’ is considered to be one of the foremost open questions of contemporary mathematics and computer science (Garey and Johnson, 1979). While no efficient algorithm for an NP-Complete problem has been found, it has yet to be proven that an efficient algorithm for one cannot exist.

Two well-known NP-Complete problems are the ‘Number Partitioning Problem’

and 'The Traveling Salesman Problem.' Suppose a group of athletes want to split up into two teams that are evenly matched. If the skill of each player is measured by an integer, can the athletes be split into two groups such that the sum of the skills in each group is the same? This is an example of the 'Number Partitioning Problem', a classic and surprisingly difficult problem in computer science, often called the 'easiest hard problem' (Hayes, 2002). The problem can be described as follows: given a set of n positive integers, separate them into two subsets such that the sum of the subsets is as close as possible to each other. Ideally, the two sums would be equal, but this is possible only if the sum of the entire set is even; in the event of an odd total, the best you can possibly do is to choose two subsets that differ by one. Try the problem on an arbitrary set of numbers such as {62, 24, 59, 71, 32, 25, 21, 39, 36, 63}. How many different possibilities are there? As you see, for large values of n this can become very time consuming! In order to find the subset pair whose sum is the closest, consider all possible subset pairs, calculate their sums, and return the pair whose sum is the closest. Since the number of subsets for an n element set is given by 2^n, as n increases the number of possibilities grows exponentially.

The 'Traveling Salesman Problem' can be described as follows: given n cities where n is a positive integer, and the distances between every pair of n cities, find a tour of minimal length, where a tour is a closed path that visits every city exactly once. Consider for example, the problem of finding the best tour of the state capitals of the United States. Provided the cost of traveling between cities is symmetric, the number of tours of n cities is $(n\text{-}1)!/2$. In order to find the best tour of the state capitals, this would require calculating 49!/2 distances and then finding the shortest one. This would take

even the fastest computers billions of years to solve (Papadimitriou and Steiglitz, 1982).

Although there are no efficient algorithms for finding the best possible solution to NP-Complete problems like the 'Number Partitioning Problem' and the 'Traveling Salesman Problem,' many approximation algorithms which provide good but not necessarily the best solution have been developed. Knowledge of NP-Complete problems is important because they arise surprisingly often in real-life applications and once a problem has been identified as NP-Complete, a lot of time can be saved by finding a good solution instead of trying to find the best possible solution (Cormen, et al., 2001).

V. A Teaching Example

This section will provide some insight on how the concept of algorithmic complexity can be incorporated into the existing high school or undergraduate mathematics curriculum. Teachers can start by presenting students with novel problems and encouraging them to develop their own algorithmic solutions. This can be more empowering to students than simply giving them an algorithm and asking them to analyze it. In particular, I have found students to be successful in developing their own algorithms for the 'Minimum/Maximum Problem.' In the pages that follow I will introduce two student-generated algorithms for this problem as well as an analysis of their respective complexities.

The 'Minimum/Maximum Problem' can be presented as follows:

Given a sequence of n distinct elements where each pair of elements can be ordered, find the minimum and maximum elements in the sequence. Students generally prefer using numbers to represent the elements in the sequence, though letters work just as well. In order to solve the problem of finding the minimum and maximum it may be helpful to

work through the simpler problem of just finding the minimum. The only operation that can be used to gain information about the sequence is the comparison of two elements. Hence, we will consider the comparison of two elements to be an elementary operation. Let $f(n)$ represent the number of comparisons necessary to find the minimum element of a sequence of n elements (also known as a sequence of length n). For n=2, to find the minimum, we compare the two elements in the sequence, hence, $f(2)=1$. If we were to add an element to this sequence, we would compare the additional element to the minimum of the existing two elements; hence, for n=3, $f(3)=2$. In general, if we know the minimum element of a sequence of length k-1, then we could compare this minimum to the k^{th} element to find the minimum of a sequence of length k. So for a sequence of length n, the number of comparisons is given by $f(n)=n-1$.

Next, let's consider the 'Minimum/Maximum Problem.' Students can be encouraged to pick a sequence of numbers and then consider the steps they would go through in order to find the minimum and maximum values. Students can formulate their algorithm by detailing the steps for a general sequence. They can then use algebra to analyze and describe the complexity of their algorithm. Some questions to pose to students to aid in their analysis are as follows:

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

(2) What properties of the input does the number of operations performed by the algorithm depend upon?

(3) Can you determine if your algorithm is 'oblivious' or 'non-oblivious'?

(4) Can you construct a complexity function to describe the number of operations performed by the algorithm for an input of a given size?

(5) If your algorithm is 'non-oblivious' can you construct a complexity function for the best and worst case scenarios?

In my experience with the 'Minimum/Maximum Problem,' high school mathematics students were able to come up with the following two algorithms rather quickly. The first algorithm uses the same approach as the method described earlier to find the minimum. Start by comparing the first two elements in the sequence to determine their minimum. Then compare the minimum to the third element to determine the minimum for the first three elements in the sequence. Continue this process until you've compared the last element in the sequence to the minimum of the others, for a total of n-1 comparisons. Once the minimum element of the entire sequence has been found, apply the same method to find the maximum, comparing elements to see which is greater. This would use an additional n-2 comparisons (as the minimum element need not be compared). Notice that this algorithm is 'oblivious,' in that the number of comparisons is the same for all inputs of size n. Let $g(n)$ represent the complexity function for this algorithm. Thus $g(n)=(n-1)+(n-2)=2n-3$ gives the total number of comparisons necessary to find the minimum and maximum of a sequence of length n. Using big-O notation we could classify this algorithm as having linear time complexity, or $O(n)$. However, since the algorithm is 'oblivious,' it can also be described as having complexity $\Theta(n)$.

Now let's consider a second, more complicated, student-generated algorithm. Start by comparing the first two elements in the sequence. Create an ordered pair with the two elements so that the smaller is in the first position and the larger is in the second position. Then compare the third element to the first term of the ordered pair. If it is smaller, let it take the place of the first term. Otherwise, compare it to the second term. If

it is smaller than the second term, move on to the next element in the sequence. If it is larger than the second term, let it take the place of second term. Continue this process for the remaining elements in the sequence comparing them to the first and last term of the ordered pair until there are no more elements in the sequence to compare. The first term of the ordered pair will be the minimum of the sequence and the last term will be the maximum of the sequence.

As with the first algorithm, the only operation that can be used to gain information about the sequence is the comparison of two elements. Hence, we will consider the comparison of two elements to be an elementary operation. Can you see why this algorithm is more complicated than the other algorithm? This algorithm is 'non-oblivious.' The number of comparisons that it performs will depend on the size as well as the structure of the sequence. The structure is relevant to this algorithm because when each element is compared to the first term in the ordered pair, a decision is made whether or not to compare it to the second term of the ordered pair. As a result, the algorithm may perform one or two comparisons for each element beyond the first two elements. In the best-case scenario, the algorithm would use one comparison for the first two elements and then one comparison for each of the remaining n -2 elements. This gives a best-case complexity function of $b(n)= 1+ (n-2)= n - 1$. Can you think of what sequence structure would result in the best-case scenario? One example is when the sequence is in decreasing order. Now let's consider the worst-case scenario. In this case, the algorithm would use one comparison for the first two elements and then two comparisons for each of the remaining n-2 elements. This gives a worst-case complexity function of $w(n)= 1+ 2(n-2)=2n - 3$. Can you think of what sequence structure would result in the worst-case

scenario? One example is when the sequence is in increasing order. Since the worst-case complexity functions is linear we can describe this algorithm as having linear time complexity, or O(n). However since the best-case complexity function is linear as well, we could also describe this algorithm as being Ω(n). Furthermore, since the algorithm is both O(n) and Ω(n), it follows that complexity is also Θ(n)!

Although big-O, Ω, and Θ are widely used to compare complexity functions, for the two algorithms presented, the notation is not helpful in determining which is more efficient. This is due to the fact that they both have a complexity of Θ(n). We can however, perform a comparison of the actual complexity functions *g*, *b,* and *w*. Graph the three functions on the same axes. Notice that for a sequence of length n=2, both algorithms would perform only one comparison. As n increases, for some inputs the second algorithm will use fewer comparisons than the first algorithm. Therefore we can conclude that the second algorithm is more efficient. Now you might be wondering why we haven't looked at the average-case complexity of the second algorithm. Oftentimes, the average-case complexity function is much harder to describe. Students may assume that the average-case complexity function is $a(n) = 1.5n-2$ but further investigation will show that this is not the case.

VI. Conclusion

Algorithms are used to solve many of the problems that govern our everyday lives. In reviewing the history of algorithmic complexity we can see how early thinkers beyond developing the algorithms to solve problems, were interested in evaluating their techniques and working towards more efficient solutions. Knowledge of whether a problem can be solved efficiently is fundamental to solving the problem. Although

computing power has increased in the last few decades, the amount of data that we are processing as a society has also grown tremendously. As a result, the ability to identify efficient algorithms for solving certain types of problems is gaining in importance (Homer and Selman, 2011). When reading texts on the analysis of algorithms it is helpful to understand the difference between the notations used to measure complexity and to be aware of the confusion that is prevalent in the literature. To better appreciate efficient algorithms we consider the alternative, algorithms whose complexity functions grow exponentially which the field of NP-Completeness encompasses. Furthermore, the 'Minimum/Maximum Problem' is included in order to provide a rich example of how algorithmic complexity can be analyzed at a level that is appropriate for high school or undergraduate mathematics students.

References

Capáy, M., and Magdin, M. (2013) Alternative Methods of Teaching Algorithms. *Social and Behavioral Sciences*, 83, 431 – 436.

Chabert, J. L. (Ed.) (1999). *A History of Algorithms: From the Pebble to the Microchip*. Berlin: Springer.

Common Core State Standards Initiative (CCSSI). (2010d) *Common Core State Standards for Mathematics*. Retrieved from http://www.corestandards.org/Math/.

Cook, S. A. (1987). An Overview of Computational Complexity. *ACM Turing Award Lectures: The First Twenty Years 1966-1985*. ACM Press Anthology Series.

Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2001). *Introduction to Algorithms* (2nd Edition). Massachusetts: The MIT Press.

da Rosa, S. (2004). Designing Algorithms in High School Mathematics. *Lecture Notes in Computer Science*, 3294, 17-31.

Garey, M. R. and Johnson, D. C. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: W.H. Freeman and Company.

Hayes, B. (2002). The Easiest Hard Problem. *American Scientist*, 90(2), 113-117.

Homer, S. and Selman, A. (2011). *Computability and Complexity Theory* (2nd Edition). New York: Springer.

Karp, R. M. (1987). Combinatorics, Complexity, and Randomness, In *ACM Turing Award Lectures: The First Twenty Years 1966-1985* (pp. 433-453). New York: ACM Press Anthology Series.

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

Knuth, D. E. (1976). Big Omicron and Big Omega and Big Theta. *ACM SIGACT News*, 8(2), 18-24.

Kronsjö, L. (1987). *Algorithms: Their Complexity and Efficiency* (2nd Edition). Great Britain: John Wiley and Sons.

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?). In B. Pullman (Ed.), *The Emergence of Complexity in Mathematics, Physics, Chemistry and Biology, Pontifical Academy of Sciences* (pp. 65-80), Vatican City: Princeton University Press.

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

Papadimitriou, C. H. and Steiglitz, K. (1982). *Combinatorial Optimization: Algorithms and Complexity*. New Jersey: Prentice-Hall, Inc.

Rosen, K. H. (1999). *Discrete Mathematics and its Applications*. United States: WCB McGraw-Hill.

Schreiber, P. (1994). *Algorithms and Algorithmic Thinking Through the Ages*. In I. Grattan-Guinness (Ed.), Companion Encyclopedia of the History and Philosophy of the Mathematical Sciences. New York: Routledge.

Sedgewick, R. (1983). *Algorithms*. Massachusetts: Addison-Wesley Publishing Company.

Shallit, J. (1994). Origins of the Analysis of the Euclidean Algorithm. *Historia Mathematica*, 21, 401-419.

Wilf, H. S. (2002). *Algorithms and Complexity*. Massachusetts: A K Peters, Ltd.

Appendix

Standard Algorithm:

Suppose you have two 2x2 matrices A=$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ and B=$\begin{bmatrix} e & f \\ g & h \end{bmatrix}$. Then the product matrix C=AB is given by C=$\begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$. An inspection of this example shows that at most 8 multiplications and 4 additions are necessary to multiply two 2x2 matrices. More generally to multiply two *n*x*n* matrices, computing each entry of the product matrix uses *n* multiplications and *n*-1 additions. Hence to compute the n^2 entries in the product matrix needs at most $n(n^2)=n^3$ multiplications and most $(n\text{-}1)(n^2)=n^3\text{-}n^2$ additions.

Strassen's Divide and Conquer Algorithm:

To multiply two 2x2 matrices A=$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ and B=$\begin{bmatrix} e & f \\ g & h \end{bmatrix}$, Strassen's algorithm first computes the following seven quantities, each of which requires exactly one multiplication:

$X_1=(a + b)(e + h)$

$X_2=(b - d)(g + h)$

$X_3=(a - c)(e + f)$

$X_4=(a + b)h$

$X_5=(c + d)e$

$X_6=a(f - h)$

$X_7=d(-e + g)$

Then the entries of the product matrix C=AB are computed as follows:

$ae + bg = X_1 + X_2 – X_4 + X_7$

$af + bh = X_4 + X_6$

$ce + dg = X_5 + X_7$

$cf + dh = X_1 – X_3 – X_5 + X_6$

To multiply two 2x2 matrices, Strassen's algorithm uses 7 multiplications and 18 additions, which reduces the number of multiplications used by the standard algorithm by 1 at the cost of 14 more additions.

Strassen's algorithm can be used to multiply larger matrices as well. Suppose we have two *n*x*n* matrices A and B where *n* is a power of 2. We partition A and B into four *n*/2x*n*/2 matrices and then multiply the parts recursively by computing the seven quantities defined above. Using Strassen's method uses $7n^3/8$ multiplications (Kronsjö, 1987). Manber (1988) notes that empirical studies indicate that *n* needs to be at least 100 to make Strassen's algorithm faster than the standard algorithm.