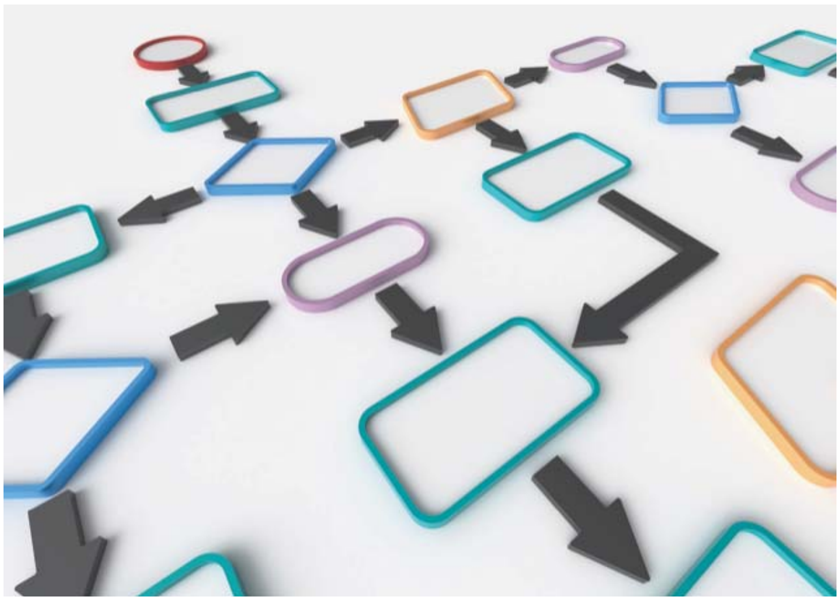
How Fast Do Algorithms Improve?

By YASH SHERRY
MIT Computer Science & Artificial Intelligence Laboratory, Cambridge, MA 02139 USA

NEIL C. THOMPSON
MIT Computer Science & Artificial Intelligence Laboratory, Cambridge, MA 02139 USA
MIT Initiative on the Digital Economy, Cambridge, MA 02142 USA



Algorithms determine which calculations computers use to solve problems and are one of the central pillars of computer science. As algorithms improve, they enable scientists to tackle larger problems and explore new domains and new scientific techniques [1], [2]. Bold claims have been made about the pace of algorithmic progress. For example, the President's Council of Advisors on Science and Technology (PCAST), a body of senior scientists that advise the U.S. President, wrote in 2010 that "in many areas, performance gains due to improvements in algorithms have vastly exceeded even the dramatic performance gains due to increased processor speed" [3]. However, this conclusion was supported based on data from progress in linear solvers [4], which is just a single example. With no guarantee that linear solvers are representative of algorithms in general, it is unclear how broadly conclusions, such as PCAST's, should be interpreted. Is progress faster in most algorithms? Just some? How much on average?

A variety of research has quantified progress for particular algorithms, including for maximum flow [5], Boolean satisfiability and factoring [6], and (many times) for linear solvers [4], [6], [7]. Others in academia [6], [8]–[10] and the private sector [11], [12] have looked at progress on benchmarks, such as computer chess ratings or weather prediction, that is not strictly comparable to algorithms since they lack either mathematically defined problem statements or verifiably optimal answers. Thus, despite substantial interest in the question, existing research provides only a limited, fragmentary view of algorithm progress.

In this article, we provide the first comprehensive analysis of algorithm progress ever assembled. This allows us to look systematically at when algorithms were discovered, how they have improved, and how the scale of these improvements compares to other sources of innovation. Analyzing data from 57 textbooks and more than 1137 research papers reveals enormous variation. Around half

This article has supplementary downloadable material available at https://doi.org/10.1109/JPROC.2021.3107219, provided by the authors.
Digital Object Identifier 10.1109/JPROC.2021.3107219

of all algorithm families experience little or no improvement. At the other extreme, 14% experience transformative improvements, radically changing how and where they can be used. Overall, we find that, for moderate-sized problems, 30%-43% of algorithmic families had improvements comparable or greater than those that users experienced from Moore's Law and other hardware advances. Thus, this article presents the first systematic, quantitative evidence that algorithms are one of the most important sources of improvement in computing.

I. RESULTS

In the following analysis, we focus on exact algorithms with exact solutions. That is, cases where a problem statement can be met exactly (e.g., find the shortest path between two nodes on a graph) and where there is a guarantee that the optimal solution will be found (e.g., that the shortest path has been identified).

We categorize algorithms into **algorithm families**, by which we mean that they solve the same underlying problem. For example, Merge Sort and Bubble sort are two of the 18 algorithms in the "Comparison Sorting" family. In theory, an infinite number of such families could be created, for example, by subdividing existing domains so that special cases can be addressed separately (e.g., matrix multiplication with a sparseness guarantee). In general, we exclude such special cases from our analysis since they do not represent an asymptotic improvement for the full problem.[1]

To focus on consequential algorithms, we limit our consideration to those families where the authors of a textbook, one of 57 that we examined, considered that family important enough to discuss. Based on these inclusion criteria, there are 113 algorithm families. On average, there are eight algorithms per family. We consider an algorithm as an improvement if it reduces the worst case asymptotic time complexity of its algorithm family. Based on this criterion, there are 276 initial algorithms and subsequent improvements, an average of 1.44 improvements after the initial algorithm in each algorithm family.

A. Creating New Algorithms

Fig. 1 summarizes algorithm discovery and improvement over time. Fig. 1(a) shows the timing for when the first algorithm in each family appeared, often as a brute-force implementation (straightforward, but computationally inefficient), and Fig. 1(b) shows the share of algorithms in each decade where asymptotic time complexity improved. For example, in the 1970s, 23 new algorithm families were discovered, and 34% of all the previously discovered algorithm families were improved upon. In later decades, these rates of discovery and improvement fell, indicating a slowdown in progress on these types of algorithms. It is unclear exactly what caused this. One possibility is that some algorithms were already theoretically optimal, so further progress was impossible. Another possibility is that there are decreasing marginal returns to algorithmic innovation [5] because the easy-to-catch innovations have already been "fished-out" [13] and what remains is more difficult to find or provides smaller gains. The increased importance of approximate algorithms may also be an explanation if approximate algorithms have drawn away researcher attention (although the causality could also run in the opposite direction, with slower algorithmic progress pushing researchers into approximation) [14].

Fig. 1(c) and (d), respectively, show the distribution of "time complexity classes" for algorithms when they were first discovered, and the probabilities that algorithms in one class transition into another because of an algorithmic improvement. **Time complexity classes**, as defined in algorithm theory, categorize algorithms by the number of operations that they require (typically expressed as a function of input size) [15]. For example, a time complexity of $O(n^2)$ indicates that, as the size of the input n grows, there exists a function Cn^2 (for some value of C) that upper bounds the number of operations required.[2] Asymptotic time is a useful shorthand for discussing algorithms because, for a sufficiently large value of n, an algorithm with a higher asymptotic complexity will always require more steps to run. Later, in this article, we show that, in general, little information is lost by our simplification to using asymptotic complexity.

Fig. 1(c) shows that, at discovery, 31% of algorithm families belong to the **exponential complexity** category (denoted $n!|c^n$)—meaning that they take at least exponentially more operations as input size grows. For these algorithms, including the famous "Traveling Salesman" problem, the amount of computation grows so fast that it is often infeasible (even on a modern computer) to compute problems of size $n = 100$. Another 50% of algorithm families begin with polynomial time that is quadratic or higher, while 19% have asymptotic complexities of $n \log n$ or better.

Fig. 1(d) shows that there is considerable movement of algorithms between complexity classes as algorithm designers find more efficient ways of implementing them. For example, on average from 1940 to 2019, algorithms with complexity $O(n^2)$ transitioned to complexity $O(n)$ with a probability of 0.5% per year, as calculated using (3). Of particular note in Fig. 1(d) are the transitions

[1]The only exception that we make to this rule is if the authors of our source textbooks deemed these special cases important enough to discuss separately, e.g., the distinction between comparison and noncomparison sorting. In these cases, we consider each as its own algorithm family.

[2]For example, the number of operations needed to alphabetically sort a list of 1000 filenames in a computer directory might be $0.5(n^2 + n)$, where n is the number of filenames. For simplicity, algorithm designers typically drop the leading constant and any smaller terms to write this as $O(n^2)$.

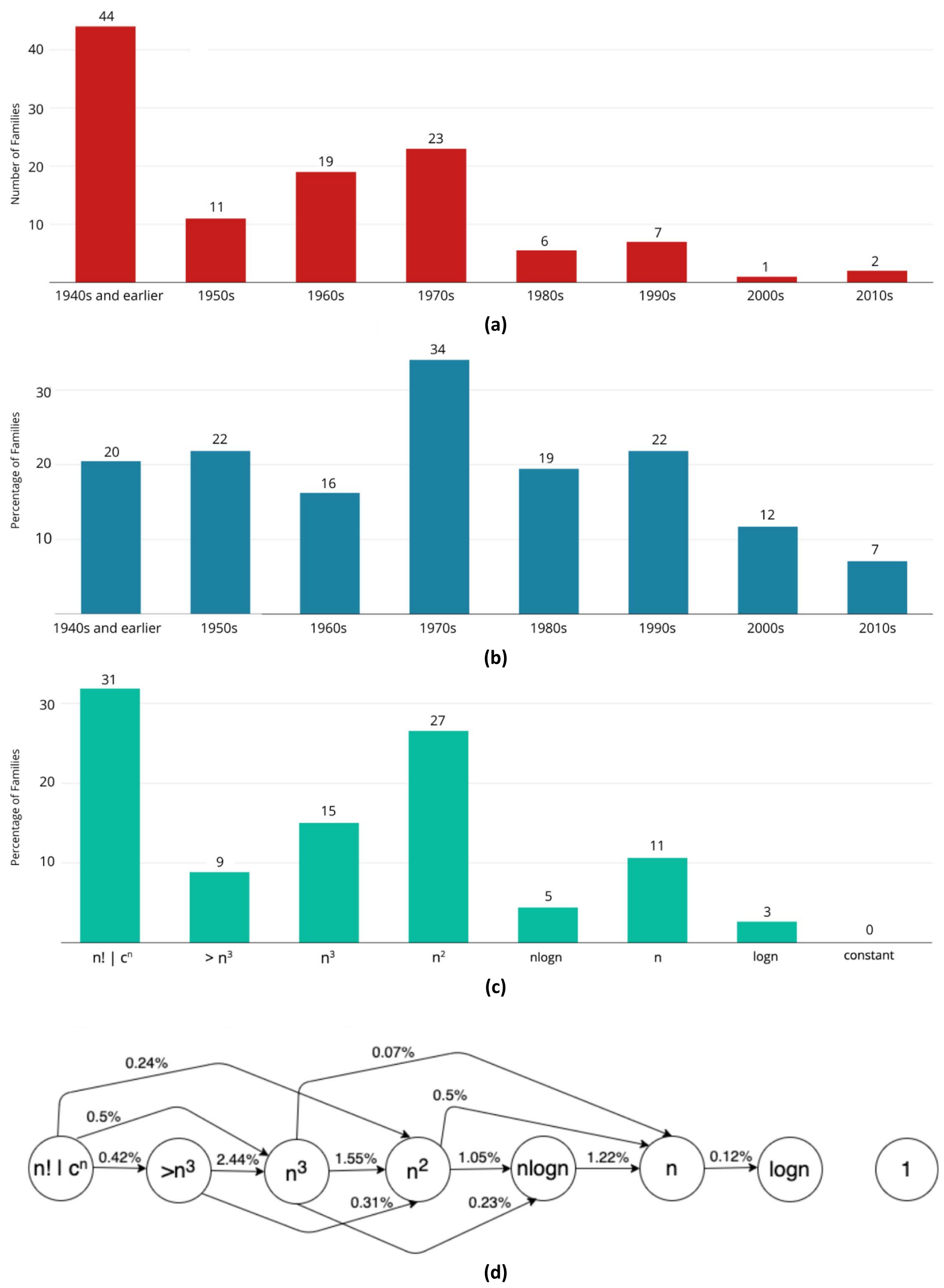


Fig. 1. *Algorithm discovery and improvement. (a) Number of new algorithm families discovered each decade. (b) Share of known algorithm families improved each decade. (c) Asymptotic time complexity class of algorithm families at first discovery. (d) Average yearly probability that an algorithm in one time complexity class transitions to another (average family complexity improvement). In (c) and (d) "$>n^3$" includes time complexities that are superpolynomial but subexponential.*

from factorial or exponential time ($n!\,|\,c^n$) to polynomial times. These improvements can have profound effects, making algorithms that were previously infeasible for any significant-sized problem possible for large datasets.

One algorithm family that has undergone transformative improvement is generating optimal binary search trees. Naively, this problem takes exponential time, but, in 1971, Knuth [16] introduced a dynamic programming solution using the properties of weighted edges to bring the time complexity to cubic. Hu and Tucker [17], in the same year, improved on this performance with a quasi-linear [$O(n \log n)$] time solution using minimum weighted path length, which remains the best asymptotic time complexity achieved for this family.[3]

B. Measuring Algorithm Improvement

Over time, the performance of an algorithm family improves as new algorithms are discovered that solve the same problem with fewer operations. To measure progress, we focus on discoveries that improve asymptotic complexity—for example, moving from $O(n^2)$ to $O(n \log n)$, or from $O(n^{2.9})$ to $O(n^{2.8})$.

Fig. 2(a) shows the progress over time for four different algorithm families, each shown in one color. In each case, performance is normalized to 1 for the first algorithm in that family. Whenever an algorithm is discovered with better asymptotic complexity, it is represented by a vertical step up. Inspired by Leiserson *et al.* [5], the height of each step is calculated using (1), representing the number of problems that the new algorithm could solve in the same amount of time as the first algorithm took to solve a single problem (in this case, for a problem of size $n = 1$ million).[4] For example, Grenander's algorithm for the maximum subarray problem, which is used in genetics (and elsewhere), is an improvement of one million $\times$ over the brute force algorithm.

To provide a reference point for the magnitude of these rates, the figure also shows the SPECInt benchmark progress time series compiled in [20], which encapsulates the effects that Moore's law, Dennard scaling, and other hardware improvements had on chip performance. Throughout this article, we use this measure as the hardware progress baseline. Fig. 2(a) shows that, for problem sizes of $n =$ 1 million, some algorithms, such as maximum subarray, have improved much more rapidly than hardware/Moore's law, while others, such as self-balancing tree creation, have not. The orders of magnitude of variation shown in just these four of our 113 families make it clear why overall algorithm improvement estimates based on small numbers of case studies are unlikely to be representative of the field as a whole.

An important contrast between algorithm and hardware improvement comes in the predictability of improvements. While Moore's law led to hardware improvements happening smoothly over time, Fig. 2 shows that algorithms experience large, but infrequent improvements (as discussed in more detail in [5]).

The asymptotic performance of an algorithm is a function of input size for the problem. As the input grows, so does the scale of improvement from moving from one complexity class to the next. For example, for a problem with $n = 4$, an algorithmic change from $O(n)$ to $O(\log n)$ only represents an improvement of 2 (=4/2), whereas, for $n = 16$, it is an improvement of 4 (=16/4). That is, algorithmic improvement is more valuable for larger data. Fig. 2(b) demonstrates this effect for the "nearest-neighbor search" family, showing that improvement size varies from 15$\times$ to $\approx$4 million$\times$ when the input size grows from 10^2 to 10^8.

While Fig. 2 shows the impact of algorithmic improvement for four algorithm families, Fig. 3 extends this analysis to 110 families.[5] Instead of showing the historical plot of improvement for each family, Fig. 3 presents the average annualized improvement rate for problem sizes of one thousand, one million, and one billion and contrasts them with the average improvement rate in hardware as measured by the SPECInt benchmark [20].

As these graphs show, there are two large clusters of algorithm families and then some intermediate values. The first cluster, representing just under half the families, shows little to no improvement even for large problem sizes. These algorithm families may be ones that have received little attention, ones that have already achieved the mathematically optimal implementations (and, thus, are unable to further improve), those that remain intractable for problems of this size, or something else. In any case, these problems have experienced little algorithmic speedup, and thus, improvements, perhaps from hardware or approximate/heuristic approaches, would be the most important sources of progress for these algorithms.

The second cluster of algorithms, consisting of 14% of the families, has yearly improvement rates greater than 1000% per year. These are algorithms that are benefited from an exponential speed-up, for example, when the initial algorithm had exponential time complexity, but later improvements made the problem solvable in polynomial time.[6] As this high improvement rate makes clear, early implementations of these algorithms would have been impossibly slow for even moderate size problems, but the algorithmic improvement has made larger data feasible. For these families, algorithm improvement has far outstripped improvements in computer hardware.

Fig. 3 also shows how large an effect problem size has on the improvement rate, calculated using (2). In particular, for $n = 1$ thousand, only 18% of families had improvement rates faster than hardware, whereas 82% had slower rates. However, for $n = 1$ million and $n = 1$ billion, 30% and 43% improved faster than hardware. Correspondingly, the median algorithm family improved 6% per year for $n = 1$ thousand but 15% per year for $n = 1$

[3]There are faster algorithms for solving this problem, for example, Levcopoulos's linear solution [18] in 1989 and another linear solution by Klawe and Mumey [19], but these are approximate and do not guarantee that they will deliver the exact right answer and, thus, are not included in this analysis.

[4]For this analysis, we assume that the leading constants are not changing from one algorithm to another. We test this hypothesis later in this article.

[5]Three of the 113 families are excluded from this analysis because the functional forms of improvements are not comparable.

[6]One example of this is the matrix chain multiplication algorithm family.

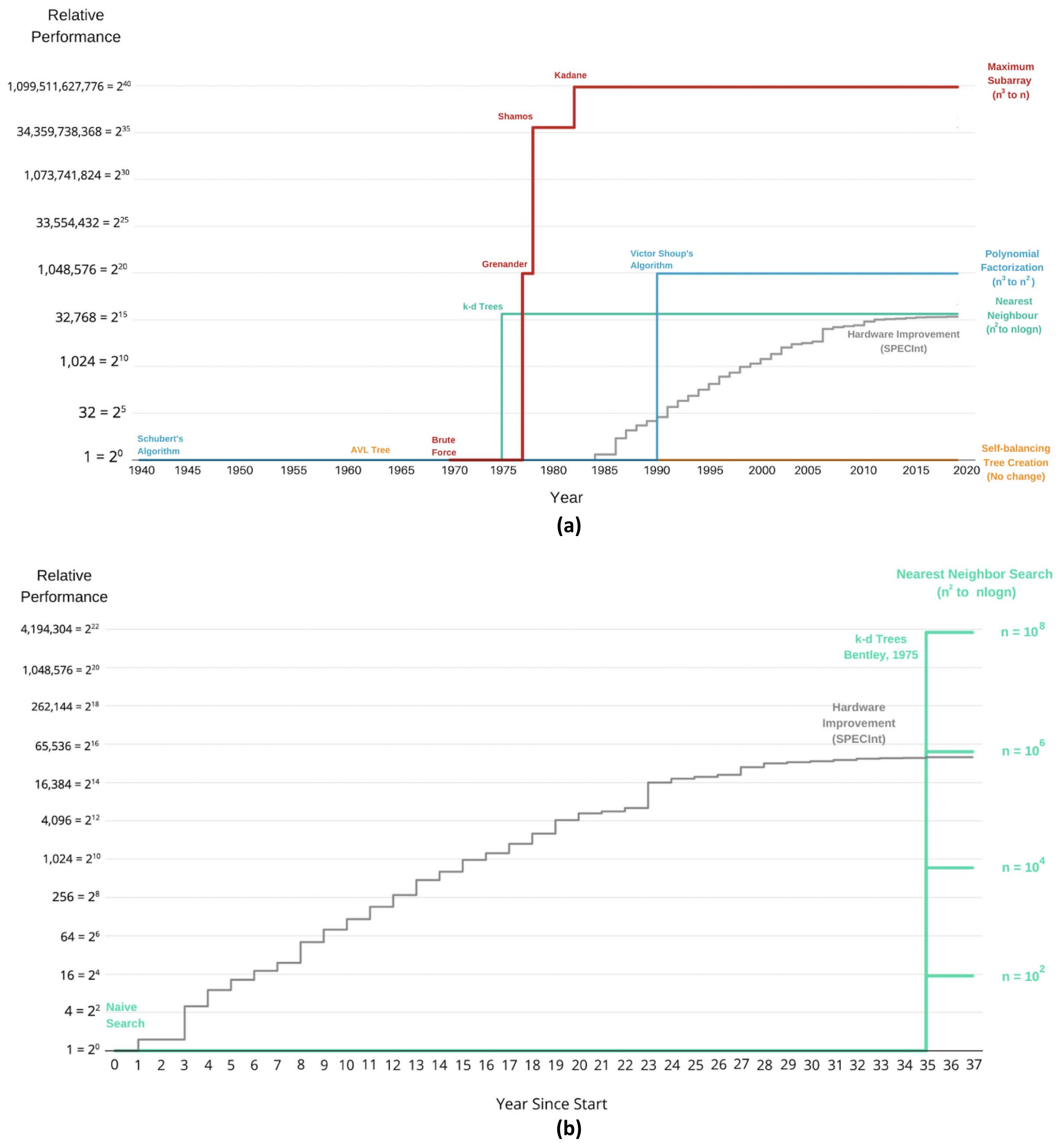


Fig. 2. ***Relative performance improvement for algorithm families, as calculated using changes in asymptotic time complexity. The comparison line is the SPECInt benchmark performance [20]. (a) Historical improvements for four algorithm families compared with the first algorithm in that family ($n = 1$ million). (b) Sensitivity of algorithm improvement measures to input size (n) for the "nearest-neighbor search" algorithm family. To ease comparison of improvement rates over time, in (b) we align the starting periods for the algorithm family and the hardware benchmark.***

million and 28% per year for $n = 1$ billion. At a problem size of $n = 1.06$ trillion, the median algorithm improved faster than hardware performance.

Our results quantify two important lessons about how algorithm improvement affects computer science. First, when an algorithm family transitions from exponential to polynomial complexity, it transforms the tractability of that problem in a way that no amount of hardware improvement can. Second, as problems increase to billions or trillions of data points, algorithmic improvement becomes substantially more important than hardware improvement/Moore's law in terms of average yearly

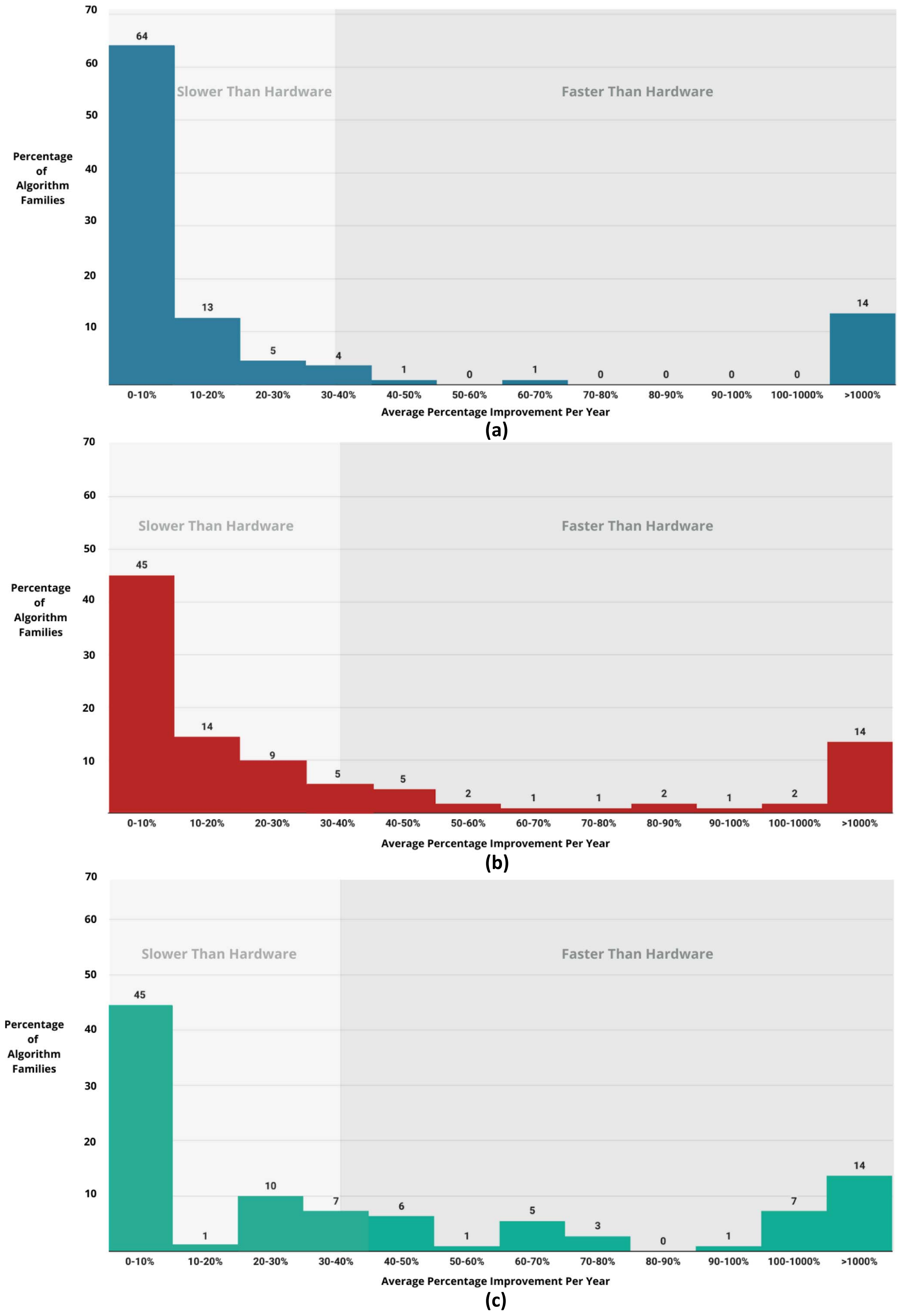


Fig. 3. ***Distribution of average yearly improvement rates for 110 algorithm families, as calculated based on asymptotic time complexity, for problems of size: (a) $n = 1$ thousand, (b) $n = 1$ million, and (c) $n = 1$ billion. The hardware improvement line shows the average yearly growth rate in SPECInt benchmark performance from 1978 to 2017, as assembled by Hennessy and Patterson [20].***

improvement rate. These findings suggest that algorithmic improvement has been particularly important in areas, such as data analytics and machine learning, which have large datasets.

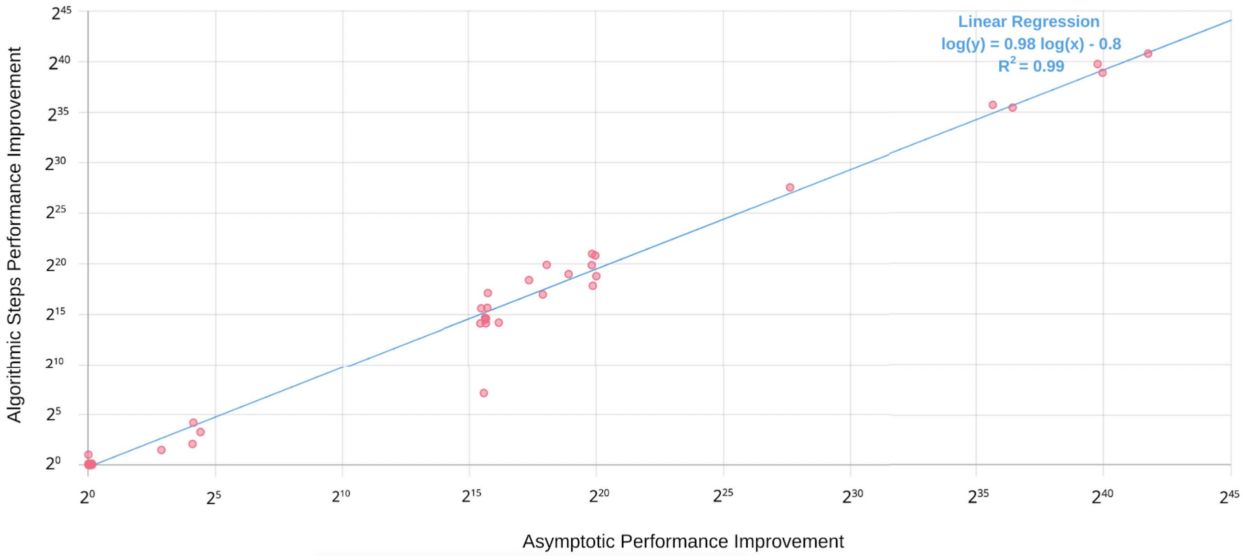


Fig. 4. *Evaluation of the importance of leading constants in algorithm performance improvement. Two measures of the performance improvement for algorithm families (first versus last algorithm in each family) for n = 1 million. Algorithmic steps include leading constants in the analysis, whereas asymptotic performance drops them.*

C. Algorithmic Step Analysis

Throughout this article, we have approximated the number of steps that an algorithm needs to perform by looking at its asymptotic complexity, which drops any leading constants or smaller-order terms, for example, simplifying $0.5(n^2 + n)$ to n^2. For any reasonable problem sizes, simplifying to the highest order term is likely to be a good approximation. However, dropping the leading constant may be worrisome if complexity class improvements come with inflation in the size of the leading constant. One particularly important example of this is the 1990 Coppersmith–Winograd algorithm and its successors, which, to the best of our knowledge, have no actual implementations because "the huge constants involved in the complexity of fast matrix multiplication usually make these algorithms impractical" [21]. If inflation of leading constants is typical, it would mean that our results overestimate the scale of algorithm improvement. On the other hand, if leading constants neither increase nor decrease, on average, then it is safe to analyze algorithms without them since they will, on average, cancel out when ratios of algorithms are taken.

To estimate the fidelity of our asymptotic complexity approximation, we reanalyze algorithmic improvement, including the leading constants (and call this latter to construct the **algorithmic steps** of that algorithm). Since only 11% of the papers in our database directly report the number of algorithmic steps that their algorithms require, whenever possible, we manually reconstruct the number of steps based on the pseudocode descriptions in the original papers. For example, Counting Sort [14] has an asymptotic time complexity of $O(n)$, but the pseudocode has four linear for-loops, yielding $4n$ algorithmic steps in total. Using this method, we are able to reconstruct the number of algorithmic steps needed for the first and last algorithms in 65% of our algorithm families. Fig. 4 shows the comparison between algorithm step improvement and asymptotic complexity improvement. In each case, we show the net effect across improvements in the family by taking the ratio of the performances of the first and final algorithms (kth) in the family $(\text{steps}_1)/(\text{steps}_k)$.

Fig. 4 shows that, *for the cases where the data are available*, the size of improvements to the number of algorithmic steps and asymptotic performance is nearly identical.[7] Thus, for the majority of algorithms, there is virtually no systematic inflation of leading constants. We cannot assume that this necessarily extrapolates to unmeasured algorithms since higher complexity may lead to both higher leading constants and a lower likelihood of quantifying them (e.g., matrix multiplication). However, this analysis reveals that these are the exception, rather than the rule. Thus, asymptotic complexity is an excellent approximation for understanding how most algorithms improve.

II. CONCLUSION

Our results provide the first systematic review of progress in algorithms—one of the pillars underpinning computing in science and society more broadly.

[7]Cases where algorithms transitioned from exponential to polynomial time (7%) cannot be shown on this graph because the change is too large. However, an analysis of the change in their leading constants shows that, on average, there is a 28% leading constant *deflation*. That said, this effect is so small compared to the asymptotic gains that it has no effect on our other estimates.

We find enormous heterogeneity in algorithmic progress, with nearly half of algorithm families experiencing virtually no progress, while 14% experienced improvements orders of magnitude larger than hardware improvement (including Moore's law). Overall, we find that algorithmic progress for the median algorithm family increased substantially but by less than Moore's law for moderate-sized problems and by more than Moore's law for big data problems. Collectively, our results highlight the importance of algorithms as an important, and previously largely undocumented, source of computing improvement.

III. METHODS

A full list of the algorithm textbooks, course syllabi, and reference papers used in our analysis can be found in the Extended Data and Supplementary Material, as a list of the algorithms in each family.

A. Algorithms and Algorithm Families

To generate a list of algorithms and their groupings into algorithmic families, we use course syllabi, textbooks, and research papers. We gather a list of major subdomains of computer science by analyzing the coursework from 20 top computer science university programs, as measured by the QS World Rankings in 2018 [22]. We then shortlist those with maximum overlap amongst the syllabi, yielding the following 11 algorithm subdomains: combinatorics, statistics/machine learning, cryptography, numerical analysis, databases, operating systems, computer networks, robotics, signal processing, computer graphics/image processing, and bioinformatics.

From each of these subdomains, we analyze algorithm textbooks—one textbook from each subdomain for each decade since the 1960s. This totals 57 textbooks because not all fields have textbooks in early decades, e.g., bioinformatics. Textbooks were chosen based on being cited frequently in algorithm research papers, on Wikipedia pages, or in other textbooks. For textbooks from recent years, where such citations' breadcrumbs are too scarce, we also use reviews on Amazon, Google, and others to source the most-used textbooks.

From each of the 57 textbooks, we used those authors' categorization of problems into chapters, subheadings, and book index divisions to determine which algorithms families were important to the field (e.g., "comparison sorting") and which algorithms corresponded to each family (e.g., "Quicksort" in comparison sorting). We also searched academic journals, online course material, and Wikipedia, and published theses to find other algorithm improvements for the families identified by the textbooks. To distinguish between serious algorithm problems and pedagogical exercises intended to help students think algorithmically (e.g., Tower of Hanoi problem), we limit our analysis to families where at least one research paper was written that directly addresses the family's problem statement.

In our analysis, we focus on exact algorithms with exact solutions. That is, cases where a problem statement can be met exactly (e.g., find the shortest path between two nodes on a graph), and there is a guarantee that an optimal solution will be found (e.g., that the shortest path has been identified). This "exact algorithm, exact solution" criterion also excludes, amongst others, algorithms where solutions, and even in theory, are imprecise (e.g., detect parts of an image that might be edges) and algorithms with precise definitions but where proposed answers are approximate. We also exclude quantum algorithms from our analysis since such hardware is not yet available.

We assess that an algorithm has improved if the work that needs to be done to complete it is reduced, asymptotically. This, for example, means that a parallel implementation of an algorithm that spreads the same amount of work across multiple processors or allows it to run on a GPU would not count toward our definition. Similarly, an algorithm that reduced the amount of memory required, without changing the total amount of work, would similarly not be included in this analysis. Finally, we focus on worst case time complexity because it does not require assumptions about the distribution of inputs and it is the most widely reported outcome in algorithm improvement papers.

B. Historical Improvements

We calculate historical improvement rates by examining the initial algorithm in each family and all subsequent algorithms that improve time complexity. For example, as discussed in [23], the "Maximum Subarray in 1D" problem was first proposed in 1977 by Ulf Grenander with a brute force solution of $O(n^3)$ but was improved twice in the next two years—first to $O(n^2)$ by Grenander and then to $O(n \log n)$ by Shamos using a Divide and Conquer strategy. In 1982, Kadane came up with an $O(n)$ algorithm, and later that year, Gries [24] devised another linear algorithm using Dijkstra's standard strategy. There were no further complexity improvements after 1982. Of all these algorithms, only Gries' is excluded from our improvement list since it did not have a better time complexity than Kadane's.

When computing the number of operations needed asymptotically, we drop leading constants and smaller order terms. Hence, an algorithm with time complexity $0.5(n^2 + n)$ is approximated as n^2. As we show in Fig. 4, this is an excellent approximation to the improvement in the actual number of algorithmic steps for the vast majority of algorithms.

To be consistent in our notation, we convert the time complexity for algorithms with matrices, which are typically parameterized in terms of the dimensions of the matrix, to being parameterized as a function of the input size. For example, this means that the standard form for writing the time complexity of naive matrix multiplication goes from N^3, where $N \times N$ is the dimension of the matrix, to an $n_{\text{input}}^{1.5}$ algorithm when n is the size of the input and $n = N \times N$.

In the presentations of our results in Fig. 1(d), we "round-up"—meaning that results between complexity classes round up to the next highest category. For example, an algorithm that scales as $n^{2.7}$ would be between n^3 and n^2 and so would get rounded up to n^3.[8] Algorithms with subexponential but superpolynomial time complexity are included in the $>n^3$ category.

C. Calculating Improvement Rates and Transition Values

In general, we calculate the improvement from one algorithm, i, to another j as

$$\text{Improvement}_{i\to j} = \frac{\text{Operations}_i(n)}{\text{Operations}_j(n)} \quad (1)$$

where n is the problem size and the number of operations is calculated either using the asymptotic complexity or algorithmic step techniques. One challenge of this calculation is that there are some improvements, which would be mathematically impressive but not realizable, for example, an improvement from 2^{2n} to 2^n, when $n = 1$ billion is an improvement ratio of $2^{1\,000\,000\,000}$. However, the improved algorithm, even after such an astronomical improvement, remains completely beyond the ability of any computer to actually calculate. In such cases, we deem that the "effective" performance improvement is zero since it went from "too large to do" to "too large to do." In practice, this means that we deem all algorithm families that transition from one factorial/exponential implementation to another has no effective improvement for Figs. 3 and 4.

[8]For ambiguous cases, we round based on the values for an input size of $n = 1\,000\,000$.

We calculate the average per-year percentage improvement rate over t years as

$$\text{YearlyImprovement}_{i\to j} = \left(\frac{\text{Operations}_i(n)}{\text{Operations}_j(n)}\right)^{1/t} - 1. \quad (2)$$

We only consider years since 1940 to be those where an algorithm was eligible for improvement. This avoids biasing very early algorithms, e.g., those discovered in Greek times, toward an average improvement rate of zero.

Both of these measures are intensive, rather than extensive, in which they measure how many more problems (of the same size) could be solved with a given number of operations. Another potential measure would be to look at the increase in the problem size that could be achieved, i.e., $(n_{\text{new}}/n_{\text{old}})$, but this requires assumptions about the computing power being used, which would introduce significant extra complexity into our analysis without compensatory benefits.

Algorithms With Multiple Parameters: While many algorithms only have a single input parameter, others have multiple. For example, graph algorithms can depend on the number of vertices, V and the number of edges, E, and, thus, have input size $V + E$. For these algorithms, increasing input size could have various effects on the number of operations needed, depending on how much of the increase in the input size was assigned to each variable. To avoid this ambiguity, we look to research papers that have analyzed problems of this type as an indication of the ratios of these parameters that are of interest to the community. For example, if an average paper considers graphs with $E = 5$ V, then we will assume this for both our base case and any scaling of input sizes. In general, we source such ratios as the geometric mean of at least three studies. In a few cases, such as Convex Hull, we have to make assumptions to calculate the improvement because newer algorithms scale differently because of output sensitivity and, thus, cannot be computed directly with only the inputs parameters. In three instances, the functional forms of the early and later algorithms are so incomparable that we do not attempt to calculate rates of improvement and instead omit them.

D. Transition Probabilities

The transition probability from one complexity class to another is calculated by counting the number of transitions that did occur and dividing by the number of transitions that could have occurred. Specifically, the probability of an algorithm transitioning from class a to b is given as follows:

$$\text{prob}(a \to b) = \frac{1}{T}\sum_{t\in T} \frac{||a \to b||_t}{||a||_{t-1} + \sum_{c\in C} ||c \to a||_t} \quad (3)$$

where t is a year from the set of possible years T and c is a time complexity class from C, which includes the null set (i.e., a new algorithm family).

For example, say we are interested in looking at the number of transitions from cubic to quadratic, i.e., $a = n^3$ and $b = n^2$. For each year, we calculate the fraction of transitions that did occur, which is just the number of algorithm families that did improve from n^3 to n^2 (i.e., $||a \to b||_t$) divided by all the transitions that could have occurred. The latter includes to three terms: all the algorithm families that were in n^3 in the previous year (i.e., $||a||_{t-1}$), all the algorithm families that move into n^3 from another class (c) in that year (i.e., $\sum_{c\in C} ||c \to a||_t$), and any new algorithm families that are created and begin in n^3 (i.e., $||\emptyset \to a||_t$). These latter two are included because a family can make multiple transitions within a single year, and thus, "newly arrived" algorithms are also eligible for asymptotic improvement in that year. Averaging across all the years in the sample provides the average transition probability from n^3 to n^2, which is 1.55%.

E. Deriving the Number of Algorithmic Steps

In general, we use pseudocode from the original papers to derive the number of algorithmic steps needed for an algorithm when the authors have not done it. When that is not available, we also use pseudocode from textbooks or other sources. Analogously to asymptotic complexity calculations, we drop smaller order terms and their constants because they have a diminishing impact as the size of the problem increases. ■

Acknowledgment

The authors would like to acknowledge generous funding from the Tides Foundation and the MIT Initiative on the Digital Economy. They would also like to thank Charles Leiserson, the MIT Supertech Group, and Julian Shun for valuable input.

Author Contributions Statement

Neil C. Thompson conceived the project and directed the data gathering and analysis. Yash Sherry gathered the algorithm's data and performed the data analysis. Both authors wrote this article.

Data Availability and Code Availability

Data are being made public through the online resource for algorithms community at algorithm-wiki.org and will launch at the time of this article's publication. Code will be available at https://github.com/canuckneil/IEEE_algorithm_paper.

Additional Information

The requests for materials should be addressed to Neil C. Thompson (neil_t@mit.edu).

References

[1] *Division of Computing and Communication Foundations CCF: Algorithmic Foundations (AF)—National Science Foundation*, Nat. Sci. Found., Alexandria, VA, USA. [Online]. Available: https://www.nsf.gov/funding/pgm_summ.jsp?pims_id=503299

[2] *Division of Physics Computational and Data-Enabled Science and Engineering (CDS&E)—National Science Foundation*, Nat. Sci. Found., Alexandria, VA, USA. [Online]. Available: https://www.nsf.gov/funding/pgm_summ.jsp?pims_id=504813

[3] J. P. Holdren, E. Lander, and H. Varmus, "President's council of advisors on science and technology," Dec. 2010. [Online]. Available: https://obamawhitehouse.archives.gov/sites/default/files/microsites/ostp/pcast-nitrd-report-2010.pdf

[4] R. Bixby, "Solving real-world linear programs: A decade and more of progress," *Oper. Res.*, vol. 50, no. 1, pp. 3–15, 2002, doi: 10.1287/opre.50.1.3.17780.

[5] C. E. Leiserson *et al.*, "There's plenty of room at the top: What will drive computer performance after Moore's law?" *Science*, vol. 368, no. 6495, Jun. 2020, Art. no. eaam9744.

[6] K. Grace, "Algorithm progress in six domains," Mach. Intell. Res. Inst., Berkeley, CA, USA, Tech. Rep., 2013.

[7] D. E. Womble, "Is there a Moore's law for algorithms," Sandia Nat. Laboratories, Albuquerque, NM, USA, Tech. Rep., 2004. [Online]. Available: https://www.lanl.gov/conferences/salishan/salishan2004/womble.pdf

[8] N. Thompson, S. Ge, and G. Filipe, "The importance of (exponentially more) computing," Tech. Rep., 2020.

[9] D. Hernandez and T. Brown, "A.I. and efficiency," OpenAI, San Francisco, CA, USA, Tech. Rep. abs/2005.04305 Arxiv, 2020.

[10] N. Thompson, K. Greenewald, and K. Lee, "The computation limits of deep learning," 2020, *arXiv:2007.05558*. [Online]. Available: https://arxiv.org/abs/2007.05558

[11] M. Manohara, A. Moorthy, J. D. Cock, and A. Aaron, *Netflix Optimized Encodes*. Los Gatos, CA, USA: Netflix Tech Blog, 2018. [Online]. Available: https://netflixtechblog.com/optimized-shot-based-encodes-now-streaming-4b9464204830

[12] S. Ismail, *Why Algorithms Are the Future of Business Success*. Austin, TX, USA: Growth Inst. [Online]. Available: https://blog.growthinstitute.com/exo/algorithms

[13] S. S. Kortum, "Research, patenting, and technological change," *Econometrica*, vol. 65, no. 6, p. 1389, Nov. 1997.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.

[15] J. Bentley, *Program. Pearls*, 2nd ed. New York, NY, USA: Association for Computing Machinery, 2006.

[16] D. E. Knuth, "Optimum binary search trees," *Acta Inform.*, vol. 1, no. 1, pp. 14–25, 1971.

[17] T. C. Hu and A. C. Tucker, "Optimal computer search trees and variable-length alphabetical codes," *SIAM J. Appl. Math.*, vol. 21, no. 4, pp. 514–532, 1971.

[18] C. Levcopoulos, A. Lingas, and J.-R. Sack, "Heuristics for optimum binary search trees and minimum weight triangulation problems," *Theor. Comput. Sci.*, vol. 66, no. 2, pp. 181–203, Aug. 1989.

[19] M. Klawe and B. Mumey, "Upper and lower bounds on constructing alphabetic binary trees," *SIAM J. Discrete Math.*, vol. 8, no. 4, pp. 638–651, Nov. 1995.

[20] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. San Mateo, CA, USA: Morgan Kaufmann, 2019.

[21] F. Le Gall, "Faster algorithms for rectangular matrix multiplication," *Commun. ACM*, vol. 62, no. 2, pp. 48–60, 2012, doi: 10.1145/3282307.

[22] *TopUniversities QS World Rankings*, Quacquarelli Symonds Ltd., London, U.K., 2018.

[23] J. Bentley, "Programming pearls: Algorithm design techniques," *Commun. ACM*, vol. 27, no. 9, pp. 865–873, 1984, doi: 10.1145/358234.381162.

[24] D. Gries, "A note on a standard strategy for developing loop invariants and loops," *Sci. Comput. Program.*, vol. 2, no. 3, pp. 207–214, 1982, doi: 10.1016/0167-6423(83)90015-1.