Measuring the Pace of Innovation: Evidence from Algorithms

Yash Mohan Sherry[1] **and Neil C. Thompson**[1,2,*]

[1]MIT Computer Science & Artificial Intelligence Lab, Cambridge, 02139, USA
[2]MIT Initiative on the Digital Economy, Cambridge, 02142, USA
[*]Corresponding author: neil_t@mit.edu

ABSTRACT

In many settings, quantifying the pace of innovation is nearly impossible. But with algorithms – the routines that computers use to solve problems - the pace of innovation can be studied precisely, offering a quantitative measure of how human ingenuity creates progress over time. Using data from more than 57 textbooks and 1,137 research papers, we present the first comprehensive look at algorithm progress ever assembled. We find that, for some algorithms, progress has easily outpaced those from other important sources (e.g. Moore's Law for computer hardware), while in other cases progress has been minimal. We also find evidence that algorithm improvement may have been much more important than computer hardware improvement in creating the Big Data revolution. Collectively, our results shed light on an important, but previously overlooked source of innovation.

Introduction

There can be little doubt of the immense progress that has been made in computing. But what is the source of this progress? One widely touted answer is Moore's Law, which describes a doubling of the computing power of computer hardware every two years.[1,2] By one measure, hardware has improved $50,000\times$ since 1978[3] and has had profound impacts on firm performance[4]. But hardware determines the number of operations a computer can do in a period of time, which is only one aspect of how quickly it operates. If methods can be discovered to solve problems with fewer operations, that will also increase performance (analogously to how, in economics, productivity improvements can be more important that accumulating additional inputs of capital or labor). Measuring progress in **algorithms** — the procedures that a computer uses to take a list of inputs, process it, and provide a correct solution — allows us to quantify how fast innovation is happening in this area.

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 to explore new domains and new scientific techniques.[5,6] 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 "Performance gains due to improvements in algorithms have vastly exceeded even the dramatic performance gains due to increased processor speed."[7] The report cites commentary by Grötschel about work in Linear Solvers[8], "a benchmark production planning model solved using linear programming would have taken 82 years to solve in 1988, using the computers and the linear programming algorithms of the day. Fifteen years later – in 2003 – this same model could be solved in roughly 1 minute, an improvement by a factor of roughly 43 million. Of this, a factor of roughly 1,000 was due to increased processor speed, whereas a factor of roughly 43,000 was due to improvements in algorithms!" Despite the impressiveness of algorithm improvement in linear solvers, it is hard to generalize about all algorithms based on it, since it is just a case study. Thus, because there is no guarantee that linear solvers are representative of algorithms more broadly, it is unclear how much confidence should be placed in sweeping conclusions, such as PCAST's.

A variety of research has quantified progress for particular algorithms, including for Maximum Flow[2], Boolean Satisfiability and Factoring[9], and (many times) for Linear Solvers[8–10]. Others[9, 11–13], have looked at progress on benchmarks, such as computer chess ratings or weather prediction that are not strictly comparable to algorithms since they lack either mathematically-defined problem statements or verifiably-optimal answers. The private sector has also seen substantial success in improving algorithms. For example, Netflix, the streaming service, has worked on improving the data compression algorithms that they use to send videos to customers over the internet. In 2018, Netflix announced an improvement that allowed them to deliver the same video quality using just one-third the bandwidth that they had been using 2 years earlier[14]. Another example comes from routing algorithms in the logistics industry. UPS estimates that, by applying telematics and algorithms, their 55,000 trucks avoid driving 85 million miles a year — yielding $2.55 billion in savings for the company[15]. But, again, while these results are impressive, they are not strictly defined algorithms. 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.

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 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 2 of the 25 algorithms in the "Sorting" family. In theory, an infinite number of such families could be created, for example by sub-dividing existing domains so that special cases can be addressed separately. To focus on consequential algorithms, we limit our consideration to those families where the authors of a textbook, one of the 57 we examined, considered that family important enough to discuss. Based on these inclusion criteria, there are 128 algorithm families. On average, there are 6.2 algorithms per family.

We categorize an algorithm as an improvement if it reduces the worst-case asymptotic time complexity of its algorithm family. Based on this criterion, there are 310 improvements, an average of 1.4 improvements over the initial algorithm in each algorithm family.

Creating new algorithms

Figure 1 summarizes algorithm discovery and improvement over time. Panel (a) shows the timing for when the first algorithm in each family appeared, often as a brute-force implementation (straightforward, but computationally inefficient) and (b) shows the share of algorithms in each decade where asymptotic time complexity improved. For example, together figures (a) and (b) reveal that, in the 1970s, 25 new algorithm families were discovered and 30% 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 there are decreasing marginal returns to algorithmic innovation[2] because the easy-to-catch innovations have already been "fished-out"[16] and what remains is more difficult or provides smaller gains. Another explanation could be the emergence of approximate algorithms (although this might also be an effect rather than a cause)[17] .

Panels (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 they require (typically expressed as a function of input size)[18]. 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.[1] Asymptotic time complexity is 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 the paper we show that, in general, little information is lost by our simplification to using asymptotic complexity.

Panel (c) shows that, at discovery, 27% of algorithm families belong to the **exponential complexity** category — meaning that they take exponentially, c^n, or 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 53% of algorithm families begin with polynomial time that is quadratic or higher, while 20% have asymptotic complexities of $n \log n$ or better.

Panel (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 0.41% per year. Of particular note in (d) are the transitions from $n!/c^n$ (factorial or exponential) time to polynomial times. These improvements can have profound effects, making algorithms that were previously infeasible for any significant-sized problem possible for large data sets. As we will show, these are the most important contributions to algorithmic improvement.

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 $O(n^{2.8})$.

Figure 2 (a) shows the progress over time for five 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[2], the height of each step represents 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).[2] For example, Grenander's algorithm for the maximum subarray problem, used in genetics (and elsewhere), is an improvement of 1 million $\times$ over the brute force algorithm.

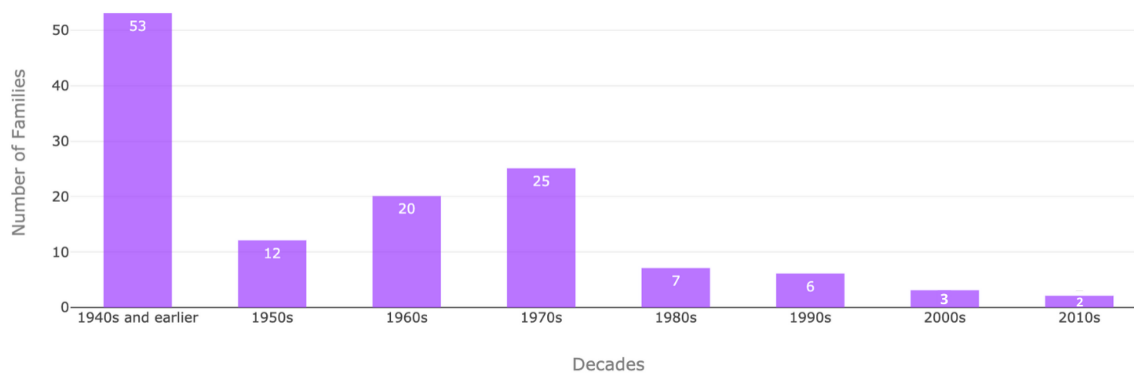
To provide a reference point for the magnitude of these rates, the figure also shows two measures of hardware improvement: an idealized Moore's Law rate of $2\times$ every two years, and the SPECInt benchmark progress time series compiled in[3] (the latter we take as the hardware progress baseline throughout the article). Figure 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 like Expectation Maximization have not. The orders of magnitude of variation shown in just these 5 of our 128 families makes 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.

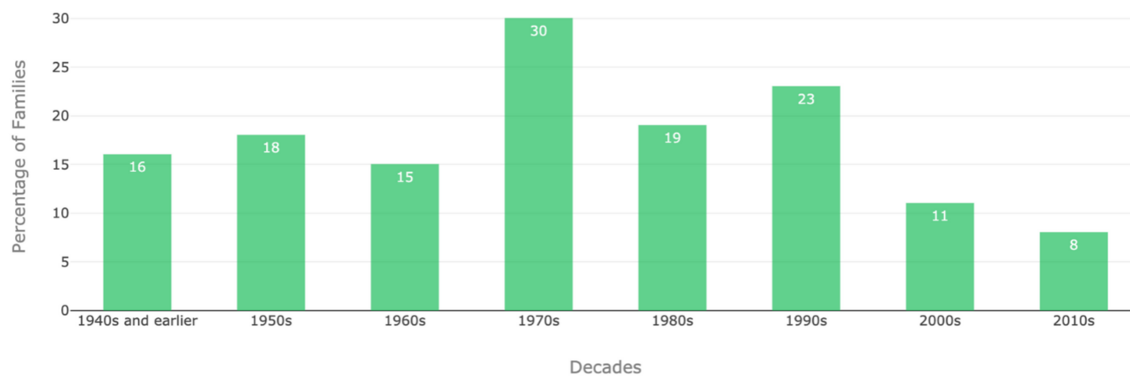
An important contrast between algorithm and hardware improvement comes in the evenness of improvements. Whereas Moore's Law led to hardware improvements happening smoothly over time, figure 2 shows that algorithms experience large, but infrequent improvements (as discussed in more detail in[2]).

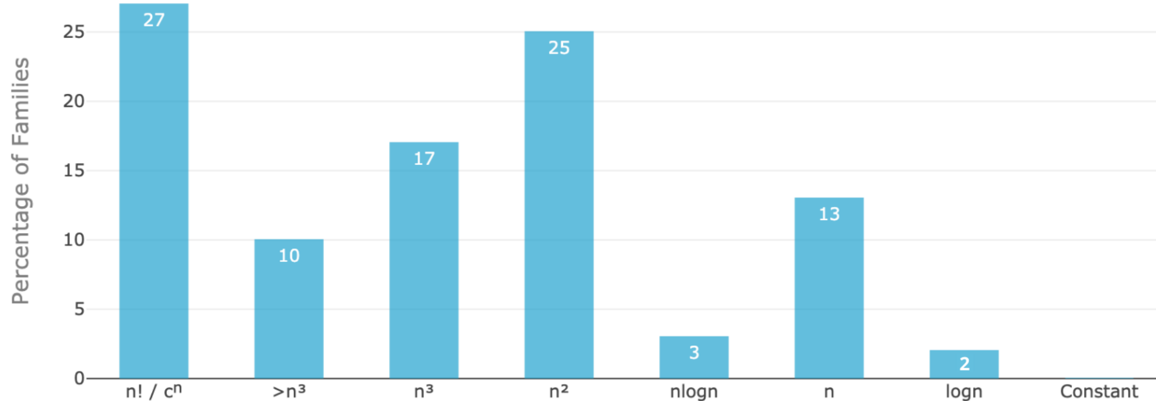
The asymptotic performance of an algorithm is a function of input size for the problem. As the input grows, so does the scale of improvement from moving from one complexity class to the next. For example, for a problem with $n = 4$ an algorithmic change from $O(n)$ to $o(\log n)$ only represents an improvement of 2 ($= \frac{4}{2}$), whereas for $n = 16$ it is an improvement of 4 ($= \frac{16}{4}$). That

[1] For example, the number of operations needed to alphabetically sort a list of 1,000 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 $O(n^2)$.

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







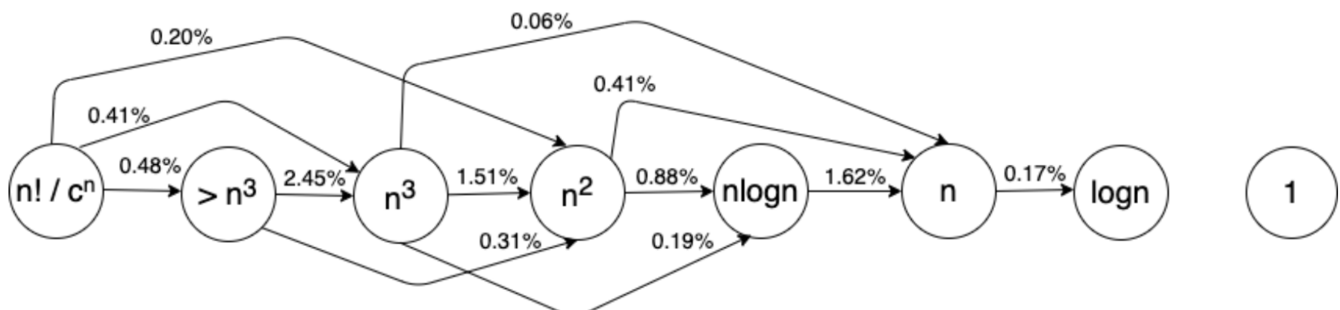
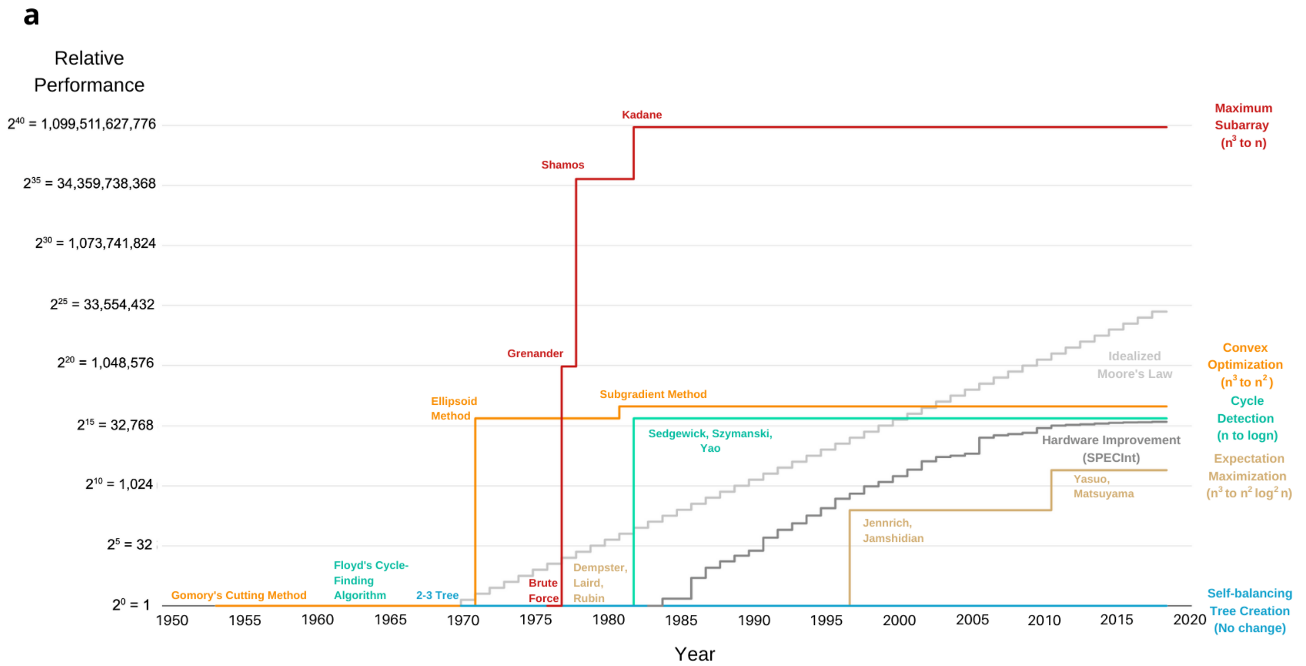


Figure 1. Algorithm discovery and improvement. **a** Number of new algorithm families discovered each decade. **b** Share of known algorithm families improved each decade. **c** Asymptotic time complexity class of algorithm families at first discovery. **d** Average yearly probability that an algorithm in one time complexity class transitions to another



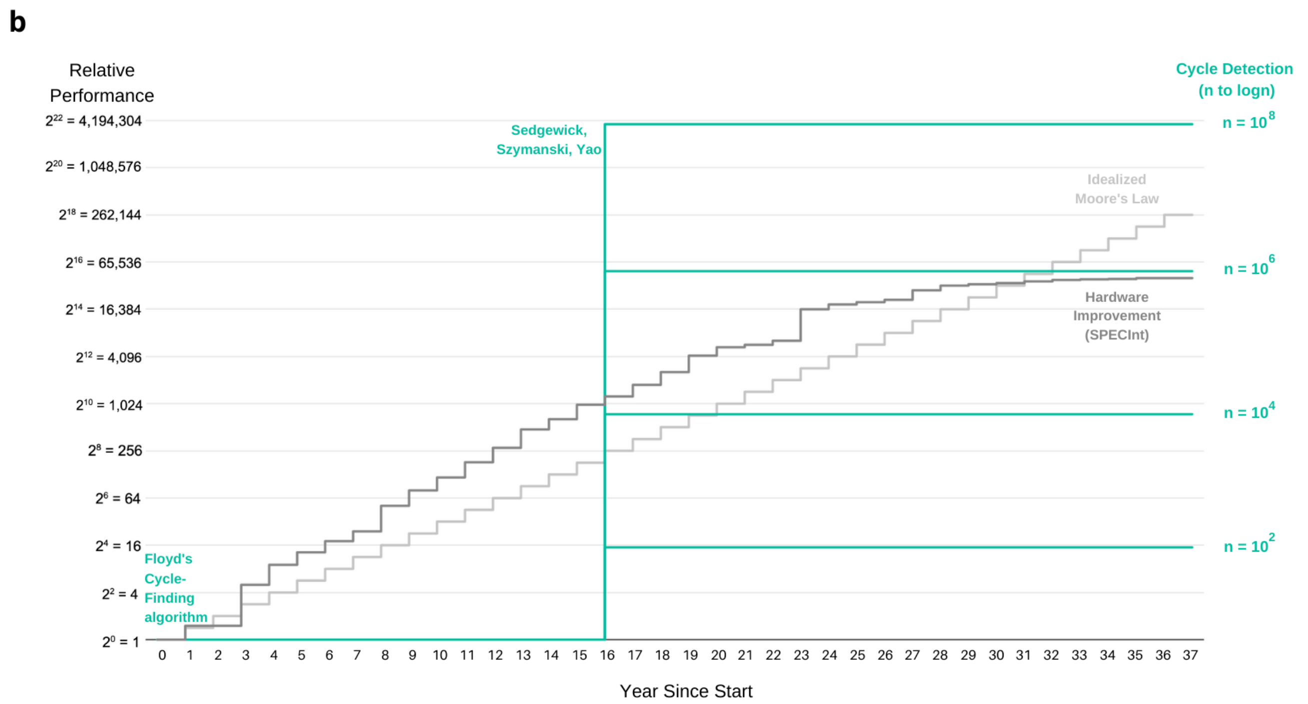


Figure 2. Relative performance improvement for algorithm families, as calculated using changes in asymptotic time complexity. Comparison lines are SPECInt benchmark performance[3] and an idealized Moore's Law ($2\times$ every 2 years). **a** Historical improvements for five algorithm families, as compared with the first algorithm in that family ($n = 1$ million). **b** Sensitivity of algorithm improvement measures to input size (n) for the "Cycle Detection" algorithm family.

is, algorithmic improvement is more valuable for larger data. Figure 2 (b) demonstrates this effect for the "Cycle Detection" family, showing that improvement size varies from $16\times$ to ≈ 4 million$\times$ when input size grows from 10^2 to 10^8.

Whereas figure 2 shows the impact of algorithmic improvement for 5 algorithm families, figure 3 extends this analysis to 125 families[3]. Instead of showing the historical plot of improvement for each family, Figure 3 presents the average annualized improvement rate for problem sizes of 1 thousand, 1 million, and 1 billion.

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 13% of the families, has yearly improvement rates greater than 1,000% per year. These are algorithms that 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[4]. As this high improvement rate makes clear, early implementations of these algorithms would have been impossibly slow for even moderate size problems, but algorithmic improvement has made larger data feasible. For these families, algorithm improvement has far outstripped improvements in computer hardware.

Figure 3 also shows how large an effect problem size has on the improvement rate. In particular, for $n = 1$ thousand, only 17% of families had improvement rates faster than hardware, whereas 83% had slower rates. But, for $n = 1$ million and $n = 1$ billion, 30% and 45% improved faster than hardware. Correspondingly, the median algorithm family improved 6% per year for $n = 1$ thousand, but 15% per year for $n = 1$ million, and 28% per year for $n = 1$ billion. At a problem size of $n = 1.23$ trillion, the median algorithm improved faster than computer hardware.

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. Secondly, 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 improvement rate. These findings suggests that algorithmic improvement has been particularly important in areas, like data analytics and machine learning, that have large datasets.

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. But dropping the leading constant may be worrisome if complexity class improvements come with inflation in the size of leading constant. One particularly important example of this is the 1990 Coppersmith-Winograd algorithm and its successors, which to our knowledge have no actual implementations because "the huge constants involved in the complexity of fast matrix multiplication usually make these algorithms impractical"[19]. 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 or 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 re-analyze algorithmic improvement including the leading constants (and call this latter construct the **algorithmic steps** of that algorithm). Since only 11% of the papers in

[3] 3 of the 128 families are excluded from this analysis because the functional forms of improvements are not comparable.

[4] One example of this is the Matrix Chain Multiplication algorithm family.

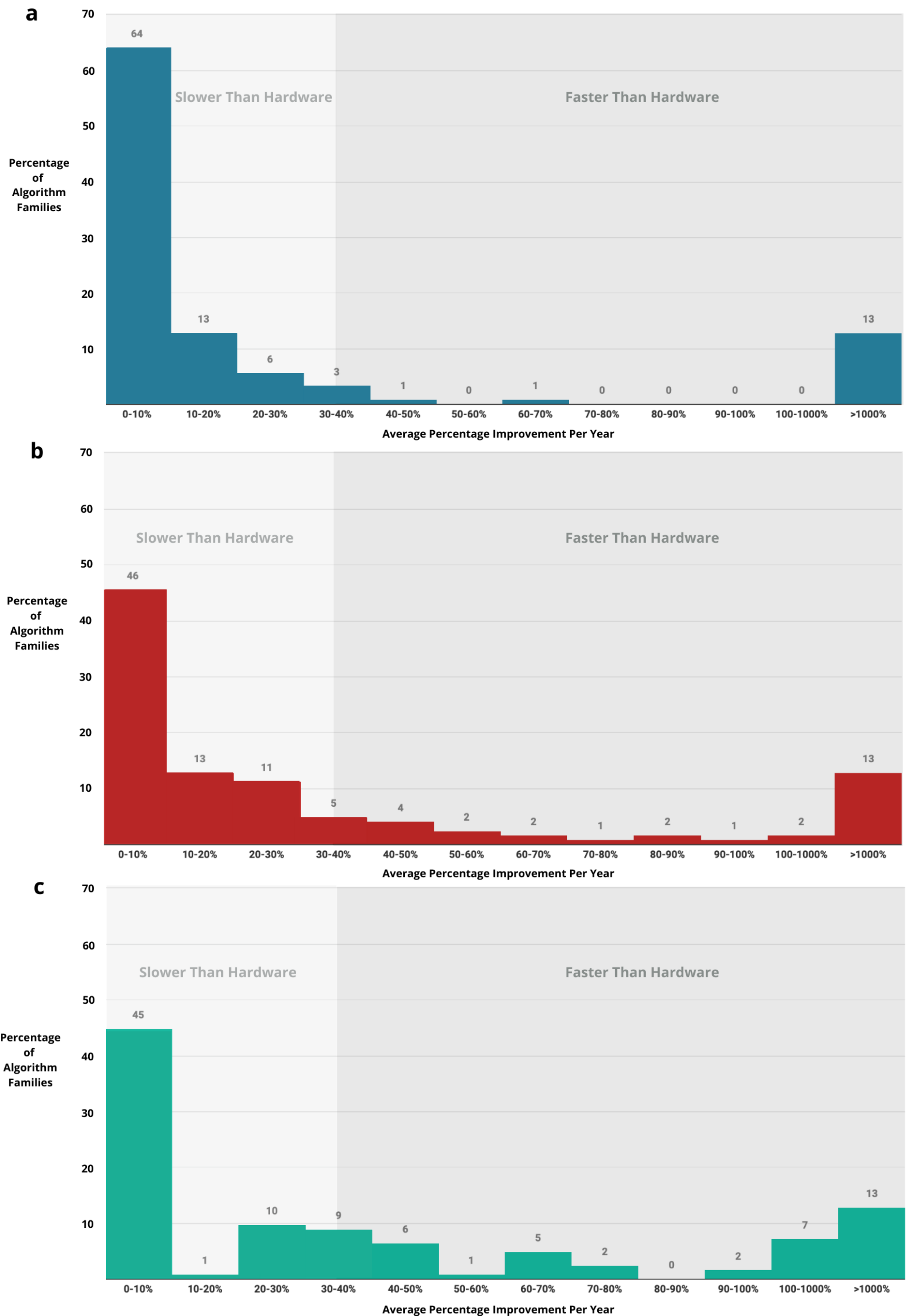


Figure 3. Distribution of average yearly improvement rates for 125 algorithm families, as calculated based on asymptotic time complexity, for problems of size: **a** $n = 1$ thousand, **b** $n = 1$ million, and **c** $n = 1$ billion. The hardware improvement line shows the average yearly growth rate in SPECInt benchmark performance from 1978 to 2017, as assembled by[3].

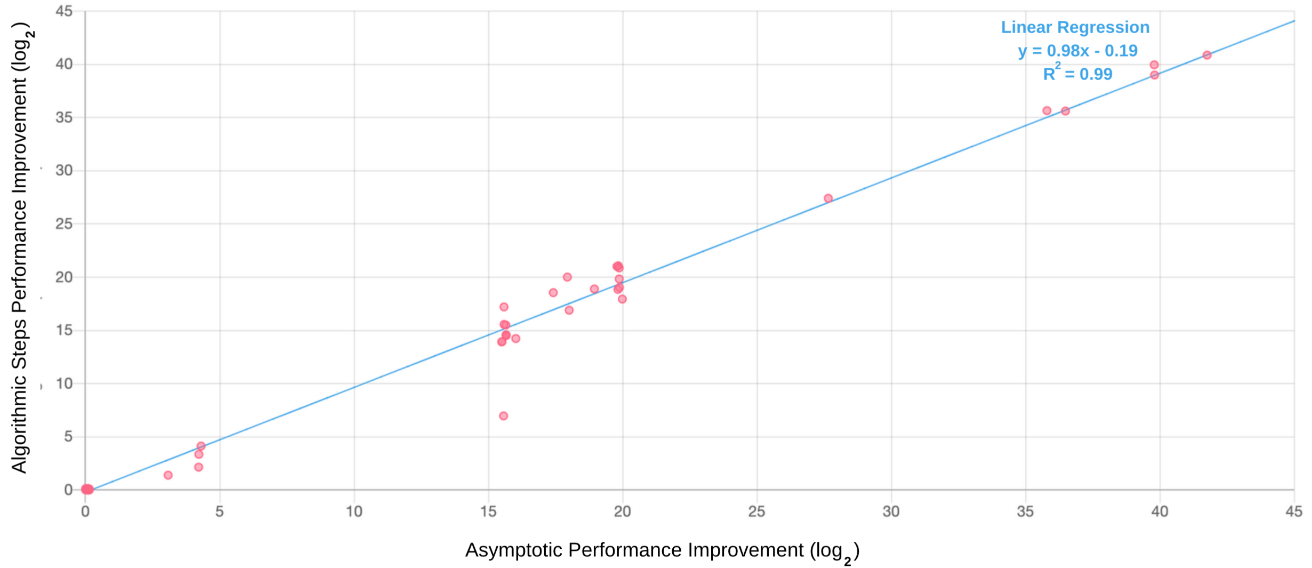


Figure 4. Evaluation of the importance of leading constants in algorithm performance improvement. Two measures of the performance improvement for algorithm families (first vs. last algorithm in each family) for $n = 1$ million. Algorithmic steps includes leading constants in the analysis, whereas asymptotic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

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 pseudo-code descriptions in the original papers. For example, Counting Sort[17] has an asymptotic time complexity of $O(n)$ but the pseudo-code has 4 linear for-loops, yielding $4n$ algorithmic steps in total. Using this method, we are able to able to reconstruct the number of algorithmic steps needed for the first and last algorithm in 62% of our algorithm families. Figure 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 (k^{th}) in the family ($\frac{steps_1}{steps_k}$).

Figure 4 shows that, *for the cases where the data is available*, the size of improvements to the number of algorithmic steps and asymptotic performance are nearly identical[5]. 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). But this analysis reveals that these are the exception, rather than the rule. Thus, asymptotic complexity is an excellent approximation for understanding how algorithms progress for most algorithms.

Discussion

Our results provide the first systematic review of progress in algorithms – one of the pillars underpinning computing in science and in society more broadly. We find enormous heterogeneity in algorithmic progress, with nearly half of algorithm families experiencing virtually no progress, while 13% 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 undocumented, source of

[5] 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 29% leading constant *deflation*. That said, this effect is so small compared to the asymptotic gains that it has no effect on our other estimates

computing improvement. Unusually, they also provide a way of measuring innovation directly and quantitatively. We find that productivity improvements in this area of computing (as with computer hardware) has significantly outpaced overall innovation in the economy, making it an important omission from other estimates of the impact of I.T. on the economy.

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 Information, as can a list of the algorithms in each family.

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 sub-domains of computer science by analyzing the coursework from the top 20 computer science university programs, as measured by the QS World Rankings in 2018[20]. We then shortlist those with maximum overlap amongst the syllabi, yielding the following 11 algorithm sub-domains: combinatorics, statistics, cryptography, numerical analysis, databases, operating systems, computer networks, robotics, signal processing, computer graphics/image processing, and bioinformatics.

From each of these sub-domains, we analyze algorithm textbooks — one from each for each decade since the 1960s for a total of 57 (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, sub-headings, and book index divisions to determine which algorithms families were important to the field (e.g. "sorting") and which algorithms corresponded to each family (e.g. "Quicksort" in sorting). We also searched academic journals, online course material, Wikipedia, and published theses to find other algorithm improvements for the families identified by the textbooks.

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, 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 towards 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 doesn't require assumptions about the distribution of inputs and because it is the most-widely reported outcome in algorithm improvement papers.

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[21], 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[22] 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.

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 Figure 4, this is an excellent approximation to the improvement in the actual number algorithmic steps for the vast majority of algorithms.

For algorithms with matrices, we follow algorithm theory convention in parameterizing the input variables as a function of matrix dimension (rather than as a total input size as is typical elsewhere). For example, the standard form for writing the time complexity of the naive implementation of matrix multiplication is n^3 where $n \times n$ is the dimension of the matrix. This results in it being classified as in the cubic (n^3) time complexity class. Were this instead parameterized by input size, then the $n \times n$ matrix size would make this an $n_{input}^{1.5}$ algorithm.

In the presentations of our results in figure 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 $n^{1.9}$ would be between $n \log n$ and n^2, and so would get rounded-up to n^2.

Calculating Improvement Rates and Transition Values

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

$$Improvement_{i \to j} = \frac{Operations_i(n)}{Operations_j(n)}$$

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 have no effective improvement for figures 3 and 4.

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

$$YearlyImprovement_{i \to j} = \left(\frac{Operations_i(n)}{Operations_j(n)}\right)^{1/t} - 1$$

We only consider years since 1940 to be those where an algorithm was eligible for improvement, which avoids biasing very-early algorithms, e.g. those discovered in Greek times, towards zero.

Both of these measures are intensive, rather than extensive, in that 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. $\frac{n_{new}}{n_{old}}$, 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 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 $E = 5V$ 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 3 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 3 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.

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 as follows:

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

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).

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 diminishing impact as the size of the problem increases.

References

1. Moore, G. E. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Soc. Newsl.* **11**, 33–35, DOI: 10.1109/N-SSC.2006.4785860 (2006).
2. Leiserson, C. E. *et al. There's Plenty of Room at the Top: What will drive computer performance after Moore's Law*.
3. Hennessy, J. L. & Patterson, D. A. *Computer Architecture: A Quantitative Approach* (Morgan Kaufmann Publications, 2019), 5th edn.
4. Thompson, N. C. The economic impact of moore's law: Evidence from when it faltered (2017).
5. *Division of Computing and Communication Foundations CCF: Algorithmic Foundations (AF) - National Science Foundation*.
6. *Division of Physics Computational and Data-Enabled Science and Engineering (CDSE) - National Science Foundation*.
7. President's council of advisors on science and technology report. Tech. Rep., Networking and Information Technology Research and Development (2010)).
8. Bixby, R. Solving real-world linear programs: A decade and more of progress. DOI: 10.1287/opre.50.1.3.17780 (2002).
9. Grace, K. Algorithm progress in six domains. (2013).

10. Womble, D. E. Is there a moore's law for algorithms? Tech. Rep., Sandia National Laboratories (2004).

11. Thompson, N., Ge, S. & Filipe, G. The importance of (exponentially more) computing. (2020).

12. Hernandez, D. & Brown, T. A.i. and efficiency. (2020).

13. Thompson, N., Greenewald, K. & Lee, K. The computation limits of deep learning. (2020).

14. Manohara, M., Moorthy, A., Cock, J. D. & Aaron, A. *Netflix Optimized Encodes* (2018).

15. Ismail, S. *Why Algorithms Are The Future Of Business Success*.

16. Kortum, S. S. Research, patenting, and technological change. *Econometrica* (1997).

17. Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. *Introduction to Algorithms* (MIT Press, 2009), 3rd edn.

18. Bentley, J. *Programming Pearls* (Association for Computing Machinery, New York, NY, United States, 2006), 2nd edn.

19. Le Gall, F. Faster algorithms for rectangular matrix multiplication. DOI: 10.1145/3282307 (2012).

20. *TopUniversities QS World Rankings 2018* (2018).

21. Bentley, J. Programming pearls: algorithm design techniques. DOI: 10.1145/358234.381162 (1984).

22. Gries, D. A note on a standard strategy for developing loop invariants and loops. DOI: 10.1016/0167-6423(83)90015-1 (1982).

Additional Information

Acknowledgements

The authors would like to acknowledge generous funding from the Tides foundation and from the MIT Initiative on the Digital Economy. We would also like to thank Charles Leiserson, the MIT Supertech group, and Julian Shun for invaluable input.

Author contributions statement

NT conceived the project and directed the data gathering and analysis. YS gathered the algorithms data and performed the data analysis. Both NT & YS wrote the paper.

Data Availability and Code Availability

Data is 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.

Additional Information

Supplementary Information is available for this paper.

Correspondence and requests for materials should be addressed to Neil Thompson (`neil_t@mit.edu`).