Twenty Questions for Donald Knuth, to celebrate the ePublication of *TAOCP*

To celebrate the publication of the eBooks of *The Art of Computer Programming* (*TAOCP*), Pearson asked several computer scientists, contemporaries, colleagues, and well-wishers to pose one question each to author Donald E. Knuth. Here are his answers. (Reprinted in *TUGboat* by kind permission of Pearson, from `www.informit.com/promotions/impact-of-the-art-of-computer-programming-139881`.)

1. Jon Bentley, researcher: What a treat! The last time I had an opportunity like this was at the end of your data structures class at Stanford in June, 1974. On the final day, you opened the floor so that we could ask any question on any topic, barring only politics and religion. I still vividly remember one question that was asked on that day: "Among all the programs you've written, of which one are you most proud?"

Your answer (as I approximately recall it, four decades later) described a compiler that you wrote for a minicomputer with 1024 available bytes of memory. Your first draft was 1029 bytes long, but you eventually had it up and running and debugged at 1023 bytes. You said that you were particularly proud of cramming so much functionality into so little memory.

My query today is a slight variant on that venerable question. Of all the programs that you've written, what are some of which you are most proud, and why?

Don Knuth: I'd like to ask you the same! But that's something like asking parents to name their favorite children.

Of course I'm proud of TEX and METAFONT, because they seem to have helped to change the world, and because they led to many friendships. Furthermore they've made these eBooks possible: I'm enormously happy that the work I did more than 30 years ago has miraculously survived many changes of technology, and that the 3,000 pages of *TAOCP* now look so great on a little tablet — even after zooming.

While I was preparing for Volume 4 of *TAOCP* in the 90s, I wrote several dozen short routines using what you and I know as "literate programming." Those little essays have been packaged into *The Stanford GraphBase* (1994), and I still enjoy using and modifying them. My favorite is the implementation of Tarjan's beautiful algorithm for strong components, which appears on pages 512–519 of that book.

I have to admit some pride also in the implementation of IEEE floating-point arithmetic that appears in my book *MMIXware* (1999), as well as that book's metasimulator for MMIX, in which I explain many principles of advanced pipelined computers from the ground up.

Literate programming continues to be one of the greatest joys of my life. In fact, I find myself writing roughly two programs per week, on average, both large and small, as I draft new material for the next volumes of *TAOCP*.

2. Dave Walden, TEX Users Group: Might you publish the original 3,000-page version of *TAOCP* (before the decision to change it into seven volumes), as a historical artifact of your view of the state of the art of algorithms and their analysis circa 1965? I think lots of people would like to see this.

Don Knuth: Scholars can look at the handwritten pages that led to Volumes 1–3 by going to the Stanford Archives, and all of the remaining pages will be deposited there eventually. I see little value in making those drafts more generally available — although some of the material about baseball that I decided not to use is pretty cool. Archives from the real pioneers of computer science, who wrote in the 40s and 50s, should be published first.

I do try to retain the youthful style of the original, in the pages that I write today, except where my first draft was embarrassingly naive or corny. I've also learned when to say "that" instead of "which," thanks in part to Guy Steele's tutelage.

3. Charles Leiserson, MIT: *TAOCP* shows a great love for computer science, and in particular, for algorithms and discrete mathematics. But love is not always easy. When writing this series, when did you find yourself reaching deepest into your emotional reservoir to overcome a difficult challenge to your vision?

Don Knuth: Again, Charles, I'd like to ask you exactly the same question!

For me, I guess, the hardest thing has always been to figure out what to cut. And I obviously haven't been very successful at that, in spite of much rewriting.

The most difficult technical challenge was to write the metasimulator for MMIX. I needed to do that behind the scenes, in order to shape what actually appears in the books, and it was surely the toughest programming task that I've ever faced. Without the methodology of literate programming, I don't think I could have finished that job successfully.

Many of the "starred" mathematical sections also stretched me pretty far. Overall, however, after working on *TAOCP* for more than fifty years, I can't think of any aspect of the activity where the effort

of writing wasn't amply repaid by what I learned while doing it.

4. Dennis Shasha, NYU: How does a beautiful algorithm compare to a beautiful theorem? In other words, what would be your criteria of beauty for each?

Don Knuth: Beauty has many aspects, of course, and is in the eye of the beholder. Some theorems and algorithms are beautiful to me because they have many different applications; some because they do powerful things with severely limited resources; some because they involve aesthetically pleasing patterns; some because they have a poetic purity of concept.

For example, I mentioned Tarjan's algorithm for strong components. The data structures that he devised for this problem fit together in an amazingly beautiful way, so that the quantities you need to look at while exploring a directed graph are always magically at your fingertips. And his algorithm also does topological sorting as a byproduct.

It's even possible sometimes to prove a beautiful theorem by exhibiting a beautiful algorithm. Look, for instance, at Theorem 5.1.4D and/or Corollary 7H in *TAOCP*.

5. Mark Taub, Pearson: Does the emergence of "apps" (small, single-function, networked programs) as the dominant programming paradigm today impact your plans in any way for future material in *TAOCP*?

Don Knuth: People who write apps use the ideas and paradigms that are already present in the first volumes. And apps make use of ever-growing program libraries, which are intimately related to *TAOCP*. Users of those libraries ought to know something about what goes on inside.

Future volumes will probably be even more "app-likable," because I've been collecting tons of fascinating games and puzzles that illustrate programming techniques in especially instructive and appealing ways.

6. Radia Perlman, Intel: (1) What is not in the books that you wish you'd included? (2) If you'd been born 200 years ago, what kind of career might you imagine you'd have had?

Don Knuth: (1) Essentially everything that I want to include is either already in the existing volumes or planned for the future ones. Volume 4B will begin with a few dozen pages that introduce certain newfangled mathematical techniques, which I didn't know about when I wrote the corresponding parts of Volume 1. (Those pages are now viewable from my website in beta-test form, under the name "mathematical preliminaries redux.") I plan to issue similar gap-filling "fascicles" when future volumes need to refer to recently invented material that ultimately belongs in Volume 3, say.

(2) Hey, what a fascinating question — I don't think anybody else has ever asked me that before!

If I'd been born in 1814, the truth is that I would almost certainly have had a very limited education, coupled with hardly any access to knowledge. My own male ancestors from that era were all employed as laborers, on farms that they didn't own, in what is now called northern Germany.

But I suppose you have a different question in mind. What if I had been one of the few people with a chance to get an advanced education, and who also had some flexibility to choose a career?

All my life I've wanted to be a teacher. In fact, when I was in first grade, I wanted to teach first grade; in second grade, I wanted to teach second; and so on. I ended up as a college teacher. Thus I suppose that I'd have been a teacher, if possible.

To continue this speculation, I have to explain about being a geek. Fred Gruenberger told me long ago that about 2% of all college students, in his experience, really resonated with computers in the way that he and I did. That number stuck in my mind, and over the years I was repeatedly able to confirm his empirical observations. For instance, I learned in 1977 that the University of Illinois had 11,000 grad students, of whom 220 were CS majors!

Thus I came to believe that a small percentage of the world's population has somehow acquired a peculiar way of thinking, which I happen to share, and that such people happened to discover each other's existence after computer science had acquired its name.

For simplicity, let me say that people like me are "geeks," and that geeks comprise about 2% of the world's population. I know of no explanation for the rapid rise of academic computer science departments — which went from zero to one at virtually every college and university between 1965 and 1975 — except that they provided a long-needed home where geeks could work together. Similarly, I know of no good explanation for the failure of many unsuccessful software projects that I've witnessed over the years, except for the hypothesis that they were not entrusted to geeks.

So who were the geeks of the early 19th century? Beginning a little earlier than 1814, I'd maybe like to start with Abel (1802); but he's been pretty much claimed by the mathematicians. Jacobi (1804), Hamilton (1805), Kirkman (1806), De Morgan (1806), Liouville (1809), Kummer (1810), and China's Li Shanlan (1811) are next; I'm listing "mathematicians" whose writings speak rather directly to the

geek in me. Then we get precisely to your time period, with Catalan (1814) and Sylvester (1814), Boole (1815), Weierstrass (1815), and Borchardt (1817). I would have enjoyed the company of all these people, and with luck I might have done similar things.

By the way, the first person in history whom I'd classify as "100% geek" was Alan Turing. Many of his predecessors had strong symptoms of our disease, but he was totally infected.

7. Tony Gaddis, author: Do you remember a specific moment when you discovered the joy of programming, and decided to make it your life's work?

Don Knuth: During the summer of 1957, between my freshman and sophomore years at Case Tech in Cleveland, I was allowed to spend all night with an IBM 650, and I was totally hooked.

But there was no question of viewing that as a "life's work," because I knew of nobody with such a career. Indeed, as mentioned above, my life's work was to be a teacher. I did write a compiler manual in 1958, which by chance was actually used as the textbook for one of my classes in 1959(!). Still, programming was for me primarily a hobby at first, after which it became a way to support myself while in grad school.

I saw no connection between computer programming and my intended career as a math professor until I met Bob Floyd late in 1962. I didn't foresee that computer science would ever be an academic discipline until I met George Forsythe in 1964.

8. Robert Sedgewick, Princeton: Don, I remember some years ago that you took the position that you weren't trying to reach everyone with your books — knowing that they would be particularly beneficial to people with a certain interest and aptitude who enjoy programming and exploring its relationship to mathematics. But lately I've been wondering about your current thoughts on this issue. It took a long time for society to realize the benefits of teaching everyone to read; now the question before us is whether everyone should learn to *program*. What do you think?

Don Knuth: I suppose all college professors think that their subject ought to be taught to everybody in the world. In this regard I can't help quoting from a wonderful paper that John Hammersley wrote in 1968:

> Just for the fun of getting his reactions, I asked an eminent scholar of English Literature what educational benefits might lie in the study of goliardic verse, Erse curses, and runic erotica. 'A working background of goliardic verse would be more than helpful to anyone hoping to have some modest facility in his own mother tongue', he declared; and with that he warmed to his subject and to the poverties of unlettered science, so that it was some minutes before I could steer him back to the Erse curses, about which he seemed a good deal less enthusiastic. 'Really', he said, 'that sort of thing isn't my subject at all. Of course, I applaud breadth of vocabulary; and you never know when some seemingly useless piece of knowledge may not turn out to be of cardinal practical importance. I could certainly *envisage* a situation in which they might come in very handy indeed'. 'And runic erotica?' 'Not extant'. (Was it only my fancy that heard a note of faint regret in his reply?) Certainly the higher flights of scholarship can add savour; but does the man-in-the-street have the time and the pertinacity and the intellectual digestion for them?

Programming, of course, is not just an ordinary subject. It is intrinsically empowering, and applicable to many different kinds of knowledge. And I also know that you've been having enormous successes, at Princeton and online, teaching advanced concepts of programming to students from every discipline.

But your question asks about *everybody*. I still think many years will have to go by before I would recommend that my own highly intelligent wife, son, and daughter should learn to program, much less that everybody else I know should do so.

Nick Trefethen told me a few years back that he had just visited his son's high school in Oxford, which is one of the best anywhere, and learned that not a single student knew how to program! Britain is now beginning to change that, indeed at a more rapid pace than in America. Yet such a revolution almost surely needs to take place over a generation or more. Where are the teachers going to come from?

My own experience is with the subset of college students who are sufficiently interested in programming that they expect it to become an integral part of their life. *TAOCP* is essentially for specialists. I've primarily been writing it for geeks, not for a general audience, because somebody has to write books that aren't for dummies. (By a "dummy" I mean a smart non-geek. That's a much larger market, and very important; but it's not my target audience, and general education is not my forte.)

On the other hand, believe it or not, I try to explain everything in my books by imagining a non-specialist reader. My goal is to be jargon-free whenever possible; I especially try to avoid terms from higher mathematics that tend to frighten the programmer-on-the-street. Whenever possible I try to translate results from the theoretical literature into a language that high-school students could understand.

I know that my books still aren't terribly easy to fathom, even for geeks. But I could have made them much, much harder.

9. Barbara Steele: What was the conversion process, and what tools did you use, to convert your print books to eBooks?

Don Knuth: I knew that these volumes would not work especially well as eBooks unless they were converted by experts. Fortunately I received some prize money in 2011, which could be used to pay for professional help. Therefore I was able to achieve the kind of quality that I envisioned, without delaying my work on future volumes, by letting the staff at Mathematical Sciences Publishers in Berkeley (MSP) handle all of the difficult stuff.

My principal goal was to make the books easily searchable — and that's a much more challenging problem than it seems, if you want to do it right. Secondarily, I wanted to let readers easily click on the number of any exercise or equation or illustration or table or algorithm, etc., and to jump to that exercise; also to jump readily between an exercise and its answer.

The people at MSP wrote special software that converts my source text into suitable input to other software that creates PDF files. I don't know the details, except that they use "change files" analogous to those used in WEB and CWEB. I've checked the results pretty carefully, and I couldn't be more pleased. Moreover, they've designed things so that it won't be hard for me to make changes next year, as readers discover bugs in the present editions.

(My style of writing tends to maximize the number of opportunities to make mistakes, hence I would be fooling myself if I thought that the books were now perfect. Therefore it has always been important to keep future errata in mind. The production staff at Addison-Wesley has been consistently wonderful in the way they allow me to correct about fifty pages every year in each volume.)

10. Silvio Levy, MSP: Could you comment on the differences between the print, PDF, EPUB, etc., editions of *TAOCP*? What would you say is gained or lost with each?

Don Knuth: The printed versions weigh a *lot* more, but they don't need battery power or a tether to electricity. They are always there; I don't have to turn them on, and I can have them all open at once.

I can scribble in the margins (and elsewhere) of the print versions, and I can highlight text in different colors. Ten years from now I expect analogous features will be commonly available for eBooks.

I'm used to flipping pages and finding my way around a regular book, much more so than in an eBook; but my grandchildren might have the opposite reaction.

The great advantage of an eBook is the reader's ability to search exhaustively. What fun it is to look for all occurrences of a random word like 'game', or for a random word fragment like 'gam' or 'ame', and find lots of cool material that I don't recall having written. The search feature on these books works even better than I had a right to hope for.

The index in a printed book has the advantage of being more focused. But that index also appears in the eBook, and in the eBook you can even click in the index to get to the cited pages.

Today's eBook readers are often inconvenient for setting bookmarks and going back to where you were a couple of minutes ago, especially after you click on an Internet link and then want to go back to reading. But that software will surely improve, and so will today's electronic devices.

In the future I look forward to curated eBooks that have additional notes by experts — and possibly even graffiti in the style of *Concrete Mathematics* — somewhat analogous to the "director's comments" and other extras found on the DVDs for films. One could select different subsets of these comments when reading.

11. Peter Gordon, Addison-Wesley (retired): If the full range of today's eBook features and functionalities had been available when *TAOCP* was first published, would you have written those volumes very differently?

Don Knuth: Well, I don't think I would have gotten very far at all. I would have had to think about doing everything in color, and with interactive figures, tables, equations, and exercises. A single person cannot use the "full range" of features that eBooks potentially have.

But by limiting myself to what can be presented well in black-and-white type, on printed pages of a fixed size, I was fortunately able to complete 3,000 pages over a period of 50 years.

12. Udi Manber, Google: The early volumes of *TAOCP* established computer programming as computer science. They introduced the necessary rigor. This was at the time when computers were used mostly for numerical applications. Today, most applications are related to people — social interaction, search, entertainment, and so on. Rigor is rarely used in the development of these applications. Speed is not always the most important factor, and "correctness" is rarely even defined. Do you have any advice on how to develop a new computer science that can introduce rigor to these new applications?

Don Knuth: The numerical computations that were somewhat central when computer science was

born are by no means gone; they continue to grow, year by year. Of course, they now represent a much smaller piece of the pie, but I don't believe in concentrating too much on the big pieces.

My work on METAFONT introduced me to applications where "correctness" cannot be defined. How do I know, for example, that my program for the letter *A* produces a correct image? I never will; and I've learned to live with that uncertainty. On the other hand, when I implemented the routines that interpret specifications and draw the associated bitmaps, there was plenty of room for rigor. The algorithms that go into font rendering are among the most interesting I've ever seen.

As a user of products from Google and Adobe and other corporations, I know that a tremendous amount of rigor goes into the manipulation of map data, transportation data, pixel data, linguistic data, metadata, and so on. Furthermore, much of that processing is done with distributed and decentralized algorithms that require more rigor than anybody ever thought of in the 60s.

So I can't say that rigor has disappeared from the computer science scene. I do wish, however, that Google's and Adobe's and Apple's programmers would learn rigorously how to keep their systems from crashing my home computers, when I'm not using Linux.

In general I agree with you that there's no decrease in the need for rigor, rather an increase in the number of kinds of rigor that are important. The fact that correctness can't be defined on the "bottom line" should not lull people into thinking that there aren't intermediate levels within every nontrivial system where correctness is crucial. Robustness and quality are compromised by every weak link.

On the other hand, I certainly don't think that everything should be mathematized, nor that everything that involves computers is properly a subdiscipline of computer science. Many parts of important software systems do not require the special talents of geeks; quite the contrary. Ideally, many disciplines collaborate, because a wide variety of orthogonal skill sets is a principal reason why life is such a joy. *Vive la difference.*

Indeed, I myself follow the path of rigor only partway: Rarely do I ever give a formal proof that any of my programs are correct, once I've constructed an informal proof that convinces me. I have no real interest, for example, in defining exactly what it would mean for TEX to be correct, or for verifying formally that my implementation of that 550-page program is free of bugs. I know that anomalous results are possible when users try to specify pages that are a mile wide, or constants that involve a trillion zeros, etc. I've taken care to avoid catastrophic crashes, but I don't check every addition operation for possible overflow.

There's even a fundamental gap in the foundations of my main mathematical specialty, the analysis of algorithms. Consider, for example, a computer program that sorts a list of numbers into order. Thanks to the work of Floyd, Hoare, and others, we have formal definitions of semantics, and tools by which we can verify that sorting is indeed always achieved. My job is to go beyond correctness, to an analysis of such things as the program's running time: I write down a recurrence, say, which is supposed to represent the average number of comparisons made by that program on random input data. I'm 100% sure that my recurrence correctly describes the program's performance, and all of my colleagues agree with me that the recurrence is "obviously" valid. *Yet I have no formal tools by which I can prove that my recurrence is right.* I don't really understand my reasoning processes at all! My student Lyle Ramshaw began to create suitable foundations in his thesis (1979), but the problem seems inherently difficult. Nevertheless, I don't lose any sleep over this situation.

13. Al Aho, Columbia: We all know that the Turing Machine is a universal model for sequential computation.

But let's consider reactive distributed systems that maintain an ongoing interaction with their environment — systems like the Internet, cloud computing, or even the human brain. Is there a universal model of computation for these kinds of systems?

Don Knuth: I'm not strong on logic, so *TAOCP* treads lightly on this sort of thing. The *TAOCP* model of computation, discussed on pages 4–8 of Volume 1, considers "reactive processes," a.k.a. "computational methods," which correspond to single processors. I've long planned to discuss recursive coroutines and other cooperative processes in Chapter 8, after I finish Chapter 7. The beautiful model of context-free parsing via semiautonomous agents, in Floyd's great survey paper of 1964, has strongly influenced my thinking in this regard.

I'd like to see extensions of the set-theoretic model of computation at the beginning of Volume 1 to the things you mention. They might well shed light on the subject.

But fully distributed processes are well beyond the scope of my books and my own ability to comprehend them. For a long time I've thought that an understanding of the way ant colonies are able to perform incredibly organized tasks might well be the

key to an understanding of human cognition. Yet the ants that invade my house continually baffle me.

14. Guy Steele, Oracle Labs: Don, you and I are both interested in program analysis: What can one know about an algorithm without actually executing it? Type theory and Hoare logic are two formalisms for that sort of reasoning, and you have made great contributions to using mathematical tools to analyze the execution time of algorithms. What do you think are interesting currently open problems in program analysis?

Don Knuth: Guy, I'm sure you aren't really *against* the idea of program execution. You and I both like to know things about programs *and* to execute them. Often the execution contradicts our supposed knowledge.

The quest for better ways to verify programs is one of the famous grand challenges of computer science. And as I said to Udi, I'm particularly rooting for better techniques that will avoid crashes.

Just now I'm writing the part of Volume 4B that discusses algorithms for satisfiability, a problem of great industrial importance. Almost nothing is known about why the heuristics in modern solvers work as well as they do, or why they fail when they do. Most of the techniques that have turned out to be important were originally introduced for the wrong reasons!

If I had my druthers, I wish people like you would put a lot of effort into a problem of which I've only recently become aware: The programmers of today's multithreaded machines need new kinds of tools that will make linked data structures much more cache-friendly. One can in many cases start up auxiliary parallel threads whose sole purpose is to anticipate the memory accesses that the main computational threads will soon be needing, and to preload such data into the cache. However, the task of setting this up is much too daunting, at present, for an ordinary programmer like me.

15. Robert Tarjan, Princeton: What do you see as the most promising directions for future work in algorithm design and analysis? What interesting and important open problems do you see?

Don Knuth: My current draft about satisfiability already mentions 25 research problems, most of which are not yet well known to the theory community. Hence many of them might well be answered before Volume 4B is ready. Open problems pop up everywhere and often. But your question is, of course, really intended to be much more general.

In general I'm looking for more focus on algorithms that work fast with respect to problems whose size, n, is *feasible*. Most of today's literature is devoted to algorithms that are asymptotically great, but they are helpful only when n exceeds the size of the universe.

In one sense such literature makes my life easier, because I don't have to discuss those methods in *TAOCP*. I'm emphatically *not* against pure research, which significantly sharpens our abilities to deal with practical problems and which is interesting in its own right. So I sometimes play asymptotic games. But I sure wouldn't mind seeing a lot more algorithms that I could also use.

For instance, I've been reading about algorithms that decide whether or not a given graph G belongs to a certain class. Is G, say, chordal? You and others discovered some great algorithms for the chordality and minimum fillin problems, early on, and an enormous number of extremely ingenious procedures have subsequently been developed for characterizing the graphs of other classes. But I've been surprised to discover that very few of these newer algorithms have actually been implemented. They exist only on paper, and often with details only sketched.

Two years ago I needed an algorithm to decide whether G is a so-called comparability graph, and was disappointed by what had been published. I believe that all of the supposedly "most efficient" algorithms for that problem are too complicated to be trustworthy, even if I had a year to implement one of them.

Thus I think the present state of research in algorithm design misunderstands the true nature of efficiency. The literature exhibits a dangerous trend in contemporary views of what deserves to be published.

Another issue, when we come down to earth, is the efficiency of algorithms on real computers. As part of the Stanford GraphBase project I implemented four algorithms to compute minimum spanning trees of graphs, one of which was the very pretty method that you developed with Cheriton and Karp. Although I was expecting your method to be the winner, because it examines much of the data only half as often as the others, it actually came out two to three times *worse* than Kruskal's venerable method. Part of the reason was poor cache interaction, but the main cause was a large constant factor hidden by O notation.

16. Frank Ruskey, University of Victoria: Could you comment on the importance of working on unimportant problems? My sense is that computer science research, funding, and academic hiring is becoming more and more focused on short-term problems that have at their heart an economic motivation. Do you agree with this assessment, is it a bad

trend, and do you see a way to mitigate it?

Similarly, could you comment on the demise of the individual researcher? So many papers that I see published these days have multiple authors. Five-author papers are routine. But when I dig into the details it seems that often only one or two have contributed the fresh ideas; the others are there because they are supervisors, or financial contributors, or whatever. I'm pretty sure that Euler didn't publish any papers with five co-authors. What is the reason for this trend, how does it interfere with trying to establish a history of ideas, and what can be done to reverse it?

Don Knuth: I was afraid somebody was going to ask a question related to economics. I've never understood anything about that subject. I don't know why people spend money to buy things. I'm willing to believe that some economists have enough wisdom to keep the world running some of the time, but their reasons are beyond me.

I just write books. I try to tell stories that seem to be important, at least for geeks. I've never bothered to think about marketing, or about what might sell, except when my publishers ask me to answer questions as I'm doing now!

Three years ago I published *Selected Papers on Fun and Games*, a 750-page book that is entirely devoted to unimportant problems. In many ways the fact that I was able to live during a time in the history of the world when such a book could be written has given me even more satisfaction than I get when seeing the currently healthy state of *TAOCP*.

I've reached an age where I can fairly be described as a "grumpy old man," and perhaps that is why I strongly share your concern for the alarming trends that you bring up. I'm profoundly upset when people rate the quality of my work by measuring the extent to which it affects Wall Street.

Everybody seems to understand that astronomers do astronomy because astronomy is interesting. Why don't they understand that I do computer science because computer science is interesting? And that I'd do it regardless of whether or not it made money for anybody? The reason is probably that not everybody is a geek.

Regarding joint authorship, you are surely right about Euler in the 18th century. In fact I can't think of *any* two-author papers in mathematics, until Hardy and Littlewood began working together at the beginning of the 20th century.

In my own case, two of my earliest papers were joint because the other authors did the theory and I wrote computer programs to validate it. Two other papers were related to the ALGOL language, and done together with ACM committees. In a number of others, written while I was at Caltech, I did the theory and my student co-authors wrote computer programs to validate it. There was one paper with Mike Garey, Ron Graham, and David Johnson, in which they did the theory and my role was to explain what they did. You and I wrote a joint paper in 2004, related to recursive coroutines, in which we shared equally.

The phenomenon of hyperauthorship still hasn't infected computer science as much as it has hit physics and biology, where I've read that Thomson-Reuters indexed more than 200 papers having 1,000 authors or more, in a single recent year! When I cite a paper in *TAOCP*, I like to mention all of the authors, and to give their full names in the index. That policy will become impossible if CS publication practices follow in the footsteps of those fields.

Collaborative work is exhilarating, and it's wonderful when new results are obtained that wouldn't have been discovered by individuals working alone. But as you say, authors should be authors, not hangers-on.

You mention the history of ideas. To me the method of discovery tends to be more important than the identification of the discoverers. Still, credit should be given where credit is due; conversely, credit shouldn't be given where credit isn't due.

I suppose the multiple-author anomalies are largely due to poor policies related to financial rewards. Unenlightened administrators seem to base salaries and promotions on publication counts.

What can we do? As I say, I'm incompetent to deal with economics. I've gone through life refusing to go along with a crowd, and bucking trends with which I disagree. I've often declined to have my name added to a paper. But I suppose I've had a sheltered existence; young people may be forced to bow to peer pressure.

17. Andrew Binstock, Dr. Dobb's: At the ACM Turing Centennial in 2012, you stated that you were becoming convinced that $P = NP$. Would you be kind enough to explain your current thinking on this question, how you came to it, and whether this growing conviction came as a surprise to you?

Don Knuth: As you say, I've come to believe that $P = NP$, namely that there does exist an integer M and an algorithm that will solve every n-bit problem belonging to the class NP in n^M elementary steps.

Some of my reasoning is admittedly naive: It's hard to believe that $P \neq NP$ and that so many brilliant people have failed to discover why. On the other hand if you imagine a number M that's finite

but incredibly large — like say the number $10\uparrow\uparrow\uparrow\uparrow 3$ discussed in my paper on "coping with finiteness" — then there's a humongous number of possible algorithms that do n^M bitwise or addition or shift operations on n given bits, and it's really hard to believe that all of those algorithms fail.

My main point, however, is that I don't believe that the equality $P = NP$ will turn out to be helpful even if it is proved, because such a proof will almost surely be nonconstructive. Although I think M probably exists, I also think human beings will never know such a value. I even suspect that nobody will even know an upper bound on M.

Mathematics is full of examples where something is proved to exist, yet the proof tells us nothing about how to find it. Knowledge of the mere *existence* of an algorithm is completely different from the knowledge of an actual algorithm.

For example, RSA cryptography relies on the fact that one party knows the factors of a number, but the other party knows only that factors exist. Another example is that the game of $N\times N$ Hex has a winning strategy for the first player, for all N. John Nash found a beautiful and extremely simple proof of this theorem in 1952. But Wikipedia tells me that such a strategy is still unknown when $N = 9$, despite many attempts. I can't believe anyone will ever know it when N is 100.

More to the point, Robertson and Seymour have proved a famous theorem in graph theory: Any class $\mathcal{C}$ of graphs that is closed under taking minors has a finite number of minor-minimal graphs. (A minor of a graph is any graph obtainable by deleting vertices, deleting edges, or shrinking edges to a point. A minor-minimal graph H for $\mathcal{C}$ is a graph whose smaller minors all belong to $\mathcal{C}$ although H itself doesn't.) Therefore there exists a polynomial-time algorithm to decide whether or not a given graph belongs to $\mathcal{C}$: The algorithm checks that G doesn't contain any of $\mathcal{C}$'s minor-minimal graphs as a minor.

But we don't know what that algorithm is, except for a few special classes $\mathcal{C}$, because the set of minor-minimal graphs is often unknown. The algorithm exists, but it's not known to be discoverable in finite time.

This consequence of Robertson and Seymour's theorem definitely surprised me, when I learned about it while reading a paper by Lovasz. And it tipped the balance, in my mind, toward the hypothesis that $P = NP$.

The moral is that people should distinguish between known (or knowable) polynomial-time algorithms and arbitrary polynomial-time algorithms. People might never be able to implement a polynomial-time-worst-case algorithm for satisfiability, even though P happens to equal NP.

18. Jeffrey O. Shallit, University of Waterloo: Decision methods, automated theorem-proving, and proof assistants have been successful in a number of different areas: the Wilf-Zeilberger method for combinatorial identities and the Robbins conjecture, to name two. What do you think theorem discovery and proof will look like in 100 years? Rather like today, or much more automated?

Don Knuth: Besides economics, I was also afraid that somebody would ask me about the future, because I'm a notoriously bad prophet. I'll take a shot at your question anyway.

Assuming 100 years of sustainable civilization, I'm fairly sure that a large percentage of theorems (maybe even 38.1966%) will be discovered with computer aid, and that a nontrivial percentage (maybe 0.7297%) will have computer-verified proofs that cannot be understood by mortals.

In my Ph.D. thesis (1963), I looked at computer-generated examples of small finite projective planes, and used that data to construct infinitely many planes of a kind never before known. Ten years later, I discovered the so-called Knuth-Morris-Pratt algorithm by studying the way one of Steve Cook's automata was able to recognize concatenated palindromes in linear time. Such investigations are fun.

A few months ago, however, I tried unsuccessfully to do a similar thing. I had a 5,000-step mechanically discovered proof that the edges of a smallish flower snark graph cannot be 3-colored, and I wanted to psych out how the machine had come up with it. Although I gave up after a couple of days, I do think it would be possible to devise new tools for the study of computer proofs in order to identify the "aha moments" therein.

In February of this year I noticed that the calculation of an Erdős-discrepancy constant — made famous by Tim Gowers' Polymath project, in which many mathematicians collaborated via the Internet — makes an instructive benchmark for satisfiability-testing algorithms. My first attempt to compute it needed 49 hours of computer time. Two weeks later I'd cut that down to less than 2 hours, but there still were 20 million steps in the proof. I see no way at present for human beings to understand more than the first few thousand of those steps.

19. Scott Aaronson, MIT: Would you recommend to other scientists to abandon the use of email, as you have done?

Don Knuth: My own situation is unusual, because I do my best work when I'm not interrupted. I eat,

sleep, and write content, more-or-less as a recluse who spends considerable time reading archives and other people's code. As I say on my home page (`http://www-cs-faculty.stanford.edu/~uno`), most people need to keep on top of things, but my role is to get to the bottom of things.

So I don't recommend a no-email policy to people who thrive on communication. And I actually take advantage of others in this respect (either shamelessly or shamefully, I'm not sure which), by pestering them with random questions, even though I don't want anybody to pester me — except about the one topic that I happen to be zooming in on at any particular time.

I do welcome email that reports bugs in *TAOCP*, because I always try to correct them as soon as possible.

Other unsolicited messages go to the bit bucket in the sky, otherwise known as `/dev/null`.

20. J. H. Quick, blogger: Why is this multi-interview called "twenty questions," when only 19 questions were asked?

Don Knuth: I'm stumped. No, wait — Radia asked two.

Incidentally, the eVolumes of *TAOCP* contain some 4,500 questions, and almost as many answers.

– – * – –

The panel

1. Jon Bentley, author of "Programming Pearls" in *Communications of the ACM*.
2. Dave Walden, TUG board member and coordinator of the TUG Interview Corner.
3. Charles Leiserson, MIT; theory of parallel computing and distributed computing, and the practical applications thereof.
4. Dennis Shasha, NYU; biological computing, pattern recognition, and machine learning.
5. Mark Taub, Pearson.
6. Radia Perlman, Intel; software designer and network engineer.
7. Tony Gaddis, author of computer science books.
8. Robert Sedgwick, Princeton; analysis of algorithms; one of Don Knuth's Ph.D. students.
9. Barbara Steele, contributor to *Common Lisp: The Language*.
10. Silvio Levy, co-author with Don Knuth of *The CWEB System of Literate Programming*, and with Raymond Seroul of *A Beginner's Book of TEX*; professional goal: to further the communication of mathematics.
11. Peter Gordon, Don Knuth's editor at Addison-Wesley from the early 1980s until his retirement in 2014; see his TUG interview at `http://tug.org/interviews/gordon.html`.
12. Udi Manber, a vice president of engineering at Google, responsible for search products.
13. Al Aho, Columbia University; programming languages, compilers and related algorithms, and prolific author of textbooks on the art and science of computer programming; co-author of the AWK programming language.
14. Guy Steele, designer and writer of numerous programming language specifications, including the original command set of Emacs; the first person to port TEX (from WAITS to ITS).
15. Robert Tarjan, Princeton; known for his pioneering work on graph theory algorithms and data structures; his dissertation was supervised by Don Knuth.
16. Frank Ruskey, University of Victoria; research includes algorithms for exhaustively listing discrete structures, and various combinatorial topics.
17. Andrew Binstock, Editor-in-Chief, *Dr. Dobb's Journal*.
18. Jeffrey Outlaw Shallit, University of Waterloo; combinatorics on words, formal languages, automata theory, and algorithmic number theory; also Vice President of Electronic Frontier Canada.
19. Scott Aaronson, MIT; theory of computational complexity and quantum computing.
20. J. H. Quick, blogger.

– – * – –

We conclude with another quote from **Radia Perlman**, regarding how *TAOCP* affected her, which also nicely expresses how many of us TEX users feel about *Computers & Typesetting*:

Having the books on my bookshelf gave me a sense of security ... that pretty much anything I'd wonder about would be explained there. Today Wikipedia serves some of that purpose. It would have been nice 20 years ago to have had a (more) portable version of Knuth so that I could know, wherever I was, that I could quickly look something up. But 20 years ago there was nothing else, so I'd have to wait until I was back at home to consult the copy in my bedroom bookshelves, or wander the halls at work to find someone who had a copy in their office. I did actually have a 2nd copy that was supposed to be at work, but it was always being "borrowed," so I could never find my own copy at work when I needed it. ■