WHAT'S SPECIAL ABOUT COMPUTER SCIENCE?

1. It’s a young field. The life of computer science (50 years) is a blink of an eye in the history of mathematics.

2. Youth implies fruitful exploration: scientifically, it has been possible to invent entire subfields from very basic principles; commercially, of course, as well.


JOHN BACKUS (FORTRAN, Backus-Naur form, FP)
Necessity, says the adage, is the mother of invention. What motivated John Backus?

Father – chemist. Tried to eliminate problem of gunpowder plants blowing up at Dupont. After World War I, became a stock broker. Very wealthy. Sends John to fancy private school.

"I flunked out every year. I never studied. I hated studying. I was just goofing around. It had the delightful consequence that every year I went to summer school in New Hampshire where I spent the summer sailing and having a nice time."

After World War II: "I really didn't know what the hell I wanted to do with my life. I decided that what I wanted was a good hi fi set because I liked music. In those days, they didn't really exist so I went to a radio technicians' school. I had a very nice teacher --- the first good teacher I ever had --- and he asked me to cooperate with him and compute the characteristics of some circuits for a magazine.

"I remember doing relatively simple calculations to get a few points on a curve for an amplifier. It was laborious and tedious and horrible, but it got me interested in math. The fact that it had an application --- that interested me."

Backus enrolled at Columbia University's School of General Studies to take some math courses. He disliked calculus but enjoyed algebra. By the spring of 1949, the 25 year-old Backus was a few months from graduating with a B.S. in mathematics, still without any idea what to do with his life.

One day that spring, Backus visited the IBM Computer Center on Madison Avenue. He was taken on a tour of the Selective Sequence Electronic Calculator (SSEC), one of IBM's early electronic (vacuum tube) machines. He was interested, took a test, and got a job. Now he was a programmer.

"You just read the manual and got the list of instructions and that was all you knew about programming. Everybody had to figure out how to accomplish something and there were of course a zillion different ways of doing it and people would do it in a zillion different ways."

Working with Harlan Herrick at IBM, Backus created a program called Speedcoding to support computation with "floating point numbers." Floating point numbers carry their scaling factor around with them, thus relieving the programmer from the drudgery of that responsibility. Backus' experience with Speedcoding set the stage for a much greater challenge.
In December of 1953, Backus wrote a memo to his boss at IBM, Cuthbert Hurd, suggesting the design of a programming language for the IBM 704. That project became known as Formula Translation --- FORTRAN. Its goals were straightforward.

“It would just mean getting programming done a lot faster. I had no idea that it would be used on other machines. There were hardly any other machines.”

But first, Backus had to overcome the persuasive arguments of von Neumann who was then a consultant to IBM.

“He didn’t see programming as a big problem. I think one of his major objections was you wouldn’t know what you were getting with floating point calculations. You at least knew where trouble was with fixed point if there was trouble. But he wasn’t sensitive to the issue of the cost of programming. He really felt that FORTRAN was a wasted effort.”

Hurd approved the project anyway and von Neumann didn’t fight it any further. Backus hired an eclectic team of experienced programmers and young mathematicians straight out of school. By the fall of 1954, the Programming Research Group had a clear vision: to create a language that would make programming easier for the IBM 704.

As they saw it, designing the language was the easy part. Translating it to the language that the machine understood directly was the real challenge – the compiler. Specifically, they had no good way to design the heart of the compiler, the part known as the "parser" or grammatical analyzer. It had to be very efficient or programmers would stick with assembly.

"It seems very unfair to me that I get so much credit for these guys [Robert Nelson, Harlan Herrick, Lois Haibt, Roy Nutt, Irving Ziller, Sheldon Best, David Sayre, Richard Goldberg, Peter Sheridan] who invented a tremendous amount of stuff. It didn't take much to manage the team. Everybody was having such a good time that my main function was to break up chess games that began at lunch and would go on to 2 P.M."

After FORTRAN, Backus was invited to participate in the international language ALGOL. He liked the ideas embodied by ALGOL, but felt frustrated by the difficulty of expressing them clearly.

"They would just describe stuff in English. Here’s this statement --- and here’s an example. You were hassled in these committees [with unproductive debates over terminology] enough to realize that something needed to be done. You needed to learn how to be precise.”

To address this problem, Backus applied a formalism called context-free languages that had just been invented by linguist Noam Chomsky. With Peter Naur, the Danish computer scientist, this became a standard description of computer languages.
Backus had invented one of the world's first and most popular programming languages and had developed a notation that would permit the definition of over a thousand more. Many people, even many great scientists, after such achievements might have coasted. Not Backus. He wasn't sure he liked what he had done.

“Once you've written a FORTRAN program, you can't tell what's going on really. It takes these two numbers and multiplies them and stores them here and does some other junk and then makes this test. Trying to figure out what is actually being calculated is not easy. Trying to do that calculation in a different way [is very difficult] because you basically don't understand what the program is doing.”

Backus's ideal was to enable the programmer to state “what you want to be done without having to get involved with the how”

Backus didn't solve that one. He posed it in his Turing award lecture. His proposal was to use a functional language related to APL.

Some inventors are motivated less by necessity than by sheer irritation at the messiness or inefficiency of the way things are. John Backus was such an inventor. Do you know others?

EDSGER W. DIJKSTRA
Science, like dress design, has its fashions. The few who ignore fashions in order to grapple with the fundamental questions of their discipline take a big gamble. Those who succeed earn the right to criticize.

Throughout a career dating back to the 1950's, Edsger W. Dijkstra has both gambled successfully and criticized severely. What drove him to do this?

Born in Rotterdam in 1930, Dijkstra is the son of two scientists --- his father was a chemist; his mother, a mathematician. Early on, Dijkstra demonstrated an aptitude and a liking for science.

"I asked my mother whether mathematics was a difficult topic. She said to be sure to learn all the formulas and be sure you know them. The second thing to remember is if you need more than five lines to prove something, then you're on the wrong track."

Dijkstra thought he might study law and serve his country in the United Nations, but his father dissuaded him.

"I was talked out of law --- the grades I had on my final examinations for mathematics, chemistry, and physics were so glorious."

Having elected to study theoretical physics, Dijkstra observed that many problems in the field required extensive calculation, so he decided to learn to program. Thanks to their wartime code-breaking work, the British led the development of European computing in the 1940's and 50's. In 1951, Dijkstra attended summer school in programming at Cambridge University. In March,
1952, based on his newly acquired knowledge, he got a part-time job at the Mathematical Centre in Amsterdam and became one of the first programmers in the Netherlands.

In the early 1950s, before the advent of FORTRAN or LISP, programmers wrote to suit the idiosyncratic design of each computer. A typical programmer would receive a list of the instructions that the machine could perform. If the hardware designers could simplify their design at the expense of programming complexity, they would not hesitate to do so. The hardware designers working with Dijkstra, however, did just the opposite.

"They would never include something in the machine unless I thought it was okay. I was to write down the functional specification that was the machine's reference manual. They referred to it as 'The Appalling Prose' --- it was as rigorous as a legal document.

"I finished my studies at Leiden as quickly as possible. Physics was a very respectable intellectual discipline. I explained to [Adriaan] van Wijngaarden [his advisor and an early Dutch computer pioneer] my hesitation about being a programmer. I told him I missed the underlying intellectual discipline of physics. He agreed that until that moment there was not much of a programming discipline, but then he went on to explain that automatic computers were here to stay, that we were just at the beginning and could not I be one of the persons called to make programming a respectable discipline in the years to come?"

Still at the Mathematical Centre, Dijkstra was asked to demonstrate the powers of the ARMAC for the forthcoming International Mathematical Conference of 1956. He started to think about the problem of determining the shortest route between two points on a railroad map. A short time later, Dijkstra and his wife were sitting on the terrace of a cafe sipping coffee on a sunny Saturday morning. Suddenly he fell silent.

"I was engaged in thought. My wife knew such periods.... The problem was so simple that you could find the solution without pencil and paper.

[You want to go from a departure city d to an arrival city a.]

The cost to get from d to d is 0 and the route is empty. Call the known cities C. Initially, C contains d alone. Loop until C contains a.

Add a city x to C if the total cost to go from d to x is less than or equal to the cost to go to any other city outside C. End loop.
"This was the first graph problem I ever posed myself and solved. The amazing thing was that I didn't publish it. It was not amazing at the time. At the time, algorithms were hardly considered a scientific topic.

"The mathematical culture of the day was very much identified with the continuum and infinity. Could a finite discrete problem be of any interest? Obviously the number of paths from here to there on a finite graph is finite. Each path is of finite length. You must search for the minimum of a finite set. Any finite set has a minimum --- next problem, please.

"For many years, I had felt guilty about my lack of mathematical education. But eventually, I think I was glad I was spared the mathematical prejudices of the day."

The shortest path algorithm, now known simply as Dijkstra's algorithm, has since been used in road building, routing through communications networks, airline flight planning, and gene comparison -- any application in which one must find the best way to travel to a destination. This is a quality of great algorithms -- so few assumptions, so many applications.
Dijkstra's work on mutual exclusion and cooperating sequential processes in the early 1960's were spurred by his involvement with the designs of the ARMAC's successors, the X1 and later the X8 in the Netherlands.

Dijkstra arranged for each device attached to the computer to perform its tasks one step at a time, while exchanging messages with the computer. In the jargon, these are called communicating sequential processes. Since Dijkstra abhors confusion, he started to think about ways to coordinate or synchronize these processes "so that I could reason about them."

Programmers face this challenge very often. Suppose the two processes must access the same data. One process might modify the data and therefore cause the other process to behave incorrectly. Avoiding bad behavior requires that while one of the processes accesses the shared data, the other one does not. This is what Dijkstra meant by synchronization.

Dijkstra thought again about trains --- this time a train signaling system known as a semaphore. Suppose that there are two separate tracks between cities X and Y, one from X to Y and the other from Y to X. If the two tracks narrow to one during a portion of the route, then trains in both directions will use the same piece of track. To avoid collisions, engineers use semaphores to ensure that only one train is on the shared track at any one time. The semaphores ensure that there is a green light in only one direction at a time and that the lights don't change color while there is a train on that critical piece of track. The result is that the semaphore ensures that there is only one train on the critical track at a given time. This is called "mutual exclusion."

Dijkstra applied the notion of mutual exclusion to the communication between the computer and its attached keyboard. These two devices exchange
information through a communication area in memory known as a "buffer." The basic rule is that only one of these two should be reading or writing the buffer at a time.

"I realized that the coupling between the typewriter [the keyboard with its circuitry] and the machine was totally symmetric. Just as the machine would be forced to wait while the buffer was still full, so the typewriter would be forced to wait while the buffer was still empty. I knew we had a logical symmetry. I remember it was very liberating, very refreshing.

"The cooperation of a number of units each with its own speed and clock --- that was the given of the technology. What I wanted to do was arrange the cooperation in such a way that it would be independent of the relative speed ratios. I wanted to do this for safety's sake."

Dijkstra's last work:

"I am working on streamlining the mathematical argument: making the argument simpler, cleaner. It's really trying to transfer experience from programming into the wider area of mathematics.

"We all know that if you want to make something big, you have to compose it out of components --- modules of some sort. You must be able to isolate parts.... It's well known from programming that this is not just a matter of division of labor because if you choose the wrong interface or an inappropriate one, the work explodes by a factor of ten --- its not just a sum.

"As an example, I have four composers living in different towns and they decide to compose a string quartet. You do the first movement, you do the adagio, you do the finale... Another way of dividing it is you do the first violin, you do the cello, you do the viola... In that latter distribution, an enormous amount of communication would be necessary between composers. That's a nice example of a practical and an impractical division of labor. Programmers have to think about this. A well-engineered mathematical theory has all the characteristics of the practical division of labor. The standard reaction of the inexperienced theorist who has demonstrated a complicated argument is to fall in love with that argument."

To Dijkstra, good definitions and a well-crafted argument are as essential as the idea itself.

"Whenever you are developing something new, you have tasks. You have to create a new subject matter. You have to create a language which is appropriate to discuss the subject matter. Many people are insufficiently aware of that second obligation."

He offers three "golden rules."

"1. Never compete with colleagues.

"2. Try the most difficult thing you can do."
"3. Choose what is scientifically healthy and relevant. Don't compromise on scientific integrity."

MICHAEL RABIN

Digital computers do exactly what they are told --- no more, no less. Programming languages often underscore this point by providing imperatives like "do," "assign," and "begin" without so much as a please or a thank you. The subtext is: the machine is your slave; tell it your bidding. Given such encouragement, few people would imagine allowing the computer to make guesses or, even worse, to behave in a manner determined by chance. How does one come up with such an idea and show it is very useful.

Rabin's father became a high school principal in Haifa soon after the family arrived in Israel from Germany in the 1930s. Michael began a religious elementary school.
"My sister, who is five years older than I, brought home The Microbe Hunters by Paul de Kruif. Reading that and other books on the pioneers of microbiology sparked my imagination, so that from the time I was eight until about twelve, I thought I might become a microbiologist.

"Then one day serendipity played a role and I was kicked out of class. There were two ninth-grade students sitting in the corridor solving Euclid style geometry problems. I looked at what they were doing. There was a problem they couldn't solve. So they challenged me and I solved it. The beauty of that, the fact that by pure thought you can establish a truth about lines and circles by the process of proof, struck me and captivated me completely."

Rabin tried to persuade his father to send him to the Reali School, which was modeled after the German Gymnasium but specialized in the sciences. His father wanted him to go to a religious high school, but the son prevailed.

"He correctly predicted that fascination with the exact sciences was going to drive me away from religion. I promised him it wouldn't be the case, but actually that was what happened."

Rabin went on to do his PhD in Princeton.

In 1957, while Rabin was writing up his thesis results, IBM Research offered summer jobs to him and another young logician named Dana Scott. The company left them free to do whatever struck their fancy and the two collaborated on what was to become a fundamental theorem in computer science. As part of their work, they introduced the notion that a computer might "guess" and arrive at simpler solutions.

Rabin and Scott began by considering a limited form of Turing's theoretical computer: one that was forbidden from writing on the tape. Such a computer, also known as a finite state machine, records what it learns in a memory whose number of states is fixed once and for all.
Rabin and Scott were intrigued by an implicit limitation of Turing's model: a machine with a given set of instructions and a particular input will always behave in the same way. Its behavior is "deterministic."

In Turing's language, a deterministic human computer will pass through the same "states of mind" every time it is presented with the same sequence of inputs. Such determinism was a basic tenet of Alan Turing's work. To explain nondeterministic behavior, Rabin offers a dinnertime analogy.

"We postulated that when the machine is in a particular state and is reading a particular input, it will have a menu of possible new states into which it can enter. Now there is not a unique computation on a given input, because there are many paths."
By the terms of Rabin's analogy, if your weather automaton provides you with at least one way to stay dry, then it is performing correctly. Otherwise it isn't. So this one is still good:

```
Wake Up
   /   \
  /     \       
Rain   Sunny
     /     \   /     \      
  /       \
Umbrella Raincoat
            /     \     
           /       \
              /        \ 
              /          \ 
           Tee-Shirt
```

But this one is not:

```
Wake Up
   /   \
  /     \       
Rain   Sunny
     /     \   /     \
  /       \
Umbrella Tee-Shirt
          /  \       
         /    \
        /     \
      Tee-Shirt
```

But this one is not:
A guessing or nondeterministic machine might seem to be a mere intellectual curiosity, except for two things.

First, Rabin and Scott showed that for finite state machines, any problem solved with a nondeterministic machine can also be solved with a deterministic one, though many more states may be needed. They showed how to convert a machine from one type to the other.

Second, nondeterministic finite state machines turned out to be an excellent way to express pattern searches in language translation, library document searching, and word processing programs. In fact, every time you do a pattern search with your computer (e.g. comp* lit*ure), it probably uses some variant of Rabin and Scott's procedure to find the matching text in your file. Rabin and Scott worked all this out at olympic speed.

"That was a fabulous collaboration... One of us would formulate a question and then we would go to our respective corners and the other one would come up with a solution. Maybe overnight. Within maybe three weeks we had all the answers, including many results that did not go into the paper because we didn't want to make it too long. These were later rediscovered by other people.

"I must say we didn't see all the implications of this. Our notion of nondeterminism was for us a mathematical creation. I've done a lot of consulting in industry since then. I may sit listening to practical systems engineers, say operating system experts, discussing the design of an operating system and somebody says 'I want you to specify whether this is deterministic or nondeterministic.' I then smile to myself because this harks back to these mathematical abstractions of nondeterministic machines."
In the summer of 1958, Rabin returned to IBM's Think Tank-on-the-Hudson, the Lamb Estate. John McCarthy was there struggling with list processing in FORTRAN and posed a puzzle to Rabin:

"There are two countries in a state of war. One country is sending spies into the other country. The spies do their spying and then they come back. They are in danger of being shot by their own guards as they try to cross the border.

"So you want to have a password mechanism. The assumption is that the spies are high caliber people and can keep a secret. But the border guards go to the local bars and chat---so whatever you tell them will be known to the enemy.

"Can you devise an arrangement where the spy will be able to come safely through, but the enemy will not be able to introduce its own spies by using information entrusted to the guards?"

Hint 1:
Rabin came up with a solution using what is now known as a "one-way" function. Rabin used a function developed by the mathematician John von Neumann.

"Suppose you take a 100-digit number, X, and square it. This is easy to do by computer. You get a 200 digit number. Out of these 200 digits, you take the middle 100. Call the resulting number Y. Now if I give you X, you can calculate Y. But if I give you Y and ask you to calculate X, you have presumably to try all the possible Xs of which there are many. So "middle squaring" is a function which is easy to compute and difficult to invert. The solution to the spy password problem was by means of a one-way function."
In 1975, I came on sabbatical to MIT. I came across a result by Gary Miller, who showed that using an unproven assumption, the so-called Riemann hypothesis, one can test very large numbers for primality by ordinary deterministic algorithms.

Underlying the primality test is the concept of a "witness" to the compositeness of a number. Consider the number 143. If I give you the number 13, you can divide 143 by 13, concluding that 143 is 11 times 13, so that 143 is composite. Thus 13 is a witness providing proof of the compositeness of 143; similarly for the other factor, 11. The trouble is that many numbers such as 143 have just a few witnesses. One is unlikely to find them by chance.

I have devised another type of witness to compositeness: numbers which constitute proof that a number n is composite if they stand in a certain relationship to n. For an n which is composite, we can prove that at least 3/4 of the numbers between 1 and n are witnesses to the compositeness of n.

The randomized test for primality of n now proceeds as follows. Randomly choose, say, 150 numbers between 1 and n. Check for each of these numbers whether it is a witness in the new sense, to the compositeness of n. If any one of the chosen numbers is a witness, then we known that n is not a prime. The novel aspect is that if none of the chosen numbers is a witness, then we declare that n is a prime.

Can it happen that even though n is composite, we erroneously declare it to be prime? Yes! But for this to occur it must happen that, even though at most 1/4 of the numbers between 1 and n are non-witnesses, in randomly drawing numbers in this range 50 times, we come up with a non-witness every time. This event has a totally negligible probability [smaller than one in a trillion trillion trillion trillion trillion trillion].

In the culture of computer science, an idea that works in one situation is called a hack, an idea that works twice is called a trick, and an idea that works often and pervasively is called a technique. Where would randomization fit? Shortly after his discovery, Rabin gave a lecture about his findings at Carnegie-Mellon University.

After the lecture, I stood in the hall and there were people standing around me in a semicircle and saying that it was very nice, but the consensus of opinion, with one exception, was that this was very specialized, that I was using specialized properties of prime numbers, namely the theorems about witnesses and specialized properties of the geometric problems that I was studying to yield the solution of these two particular problems. This was not, people said to me, of wide utility.

The only dissenting opinion came from Joe Traub [then chairman of Carnegie's Computer Science department and a computational theorist]. He said that using randomization and allowing the possibility of errors is a new departure.
1. It is math (e.g. primality test, nature of proof).
2. It is design (e.g. programming languages and the linguistics of mathematical expression)
3. It is engineering (e.g. concurrent systems use semaphores because they enable systems to work)
4. It is a human science (e.g. Fortran took off because people found they made fewer errors with a higher level language; the use of one way functions was inspired by an emotive story about spies and is used in modern day cryptography)