



*Selected  
Papers on  
Computer  
Science*

Donald E. Knuth

CSLI LECTURE NOTES NUMBER 59

*Selected  
Papers on  
Computer  
Science*



Donald E. Knuth

**CSLI** Publications

 **CAMBRIDGE**  
UNIVERSITY PRESS

T<sub>E</sub>X is a trademark of the American Mathematical Society.  
METAFONT is a trademark of Addison-Wesley Publishing Company, Inc.

Copyright ©1996  
Center for the Study of Language and Information  
Leland Stanford Junior University  
Co-published by Cambridge University Press  
Printed in the United States  
03 02 01 00 99 6 5 4 3 2

Library of Congress Cataloging-in-Publication Data

Knuth, Donald Ervin, 1938-

Selected papers on computer science / Donald E. Knuth  
xii,274 p. 23 cm. -- (CSLI lecture notes ; no. 59)

Includes bibliographical references and index.

ISBN 1-881526-91-7 (paperback : alk. paper) --

ISBN 1-881526-92-5 (hardback : alk. paper)

1. Computer science. I. Title. II. Series.

QA76.6.K537 1996

004.1'1--dc20

96-11382

CIP

## Chapter 1

---

# Computer Science and its Relation to Mathematics

*[Originally published in American Scientist, vol. 61, no. 6, November-December 1973; and in The American Mathematical Monthly, vol. 81, no. 4, April 1974, with additional material that is reproduced here.]*

A new discipline called Computer Science has recently arrived on the scene at most of the world's universities. The present article gives a personal view of how this subject interacts with Mathematics, by discussing the similarities and differences between the two fields, and by examining some of the ways in which they help each other. A typical nontrivial problem is worked out in order to illustrate these interactions.

### What is Computer Science?

Since Computer Science is relatively new, I must begin by explaining what it is all about. At least, my wife tells me that she has to explain it whenever anyone asks her what I do, and I suppose most people today have a somewhat different perception of the field than mine. In fact, no two computer scientists will probably give the same definition; this is not surprising, since it is just as hard to find two mathematicians who give the same definition of Mathematics. Fortunately it has been fashionable in recent years to have an "identity crisis," so computer scientists have been right in style.

My favorite way to describe computer science is to say that it is the study of algorithms. An algorithm is a precisely-defined sequence of rules telling how to produce specified output information from given input information in a finite number of steps. A particular representation of an algorithm is called a program, just as we use the word "data" to stand for a particular representation of "information" [16]. Perhaps the most significant discovery generated by the advent of computers will

## 6 Selected Papers on Computer Science

turn out to be that algorithms, as objects of study, are extraordinarily rich in interesting properties; and furthermore, that an algorithmic point of view is a useful way to organize knowledge in general. G. E. Forsythe has observed that "the question 'What can be automated?' is one of the most inspiring philosophical and practical questions of contemporary civilization" [8].

From these remarks we might conclude that Computer Science should have existed long before the advent of computers. In a sense, it did; the subject is deeply rooted in history. For example, I recently found it interesting to study ancient documents, learning to what extent the Babylonians of 3500 years ago were computer scientists [18]. But computers are really necessary before we can learn much about the general properties of algorithms; human beings are not precise enough nor fast enough to carry out any but the simplest procedures. Therefore the potential richness of algorithmic studies was not fully realized until general-purpose computing machines became available.

I should point out that computing machines (and algorithms) do not only compute with *numbers*. They can deal with information of any kind, once it is represented in a precise way. We used to say that a sequence of symbols, such as a name, is represented inside a computer as if it were a number; but it is really more correct to say that a number is represented inside a computer as a sequence of symbols.

The French word for computer science is *Informatique*; the German is *Informatik*; in Danish, the word is *Datalogi* [23]. All of these terms wisely imply that computer science deals with many things besides the solution to numerical equations. However, these names emphasize the "stuff" that algorithms manipulate (the information or data), instead of the algorithms themselves. The Norwegians at the University of Oslo have chosen a somewhat more appropriate designation for computer science, namely *Databehandling*; its English equivalent, "Data Processing" has unfortunately been used in America only in connection with business applications, while "Information Processing" tends to connote library applications. Several people have suggested the term "Computing Science" as superior to "Computer Science."

The search for a perfect name is somewhat pointless, of course, since the underlying concepts are much more important than the name. Yet we cannot help noticing that these other names for computer science all de-emphasize the role of computing machines themselves, apparently in order to make the field more legitimate and respectable. Many people's opinion of a computing machine is, at best, that it is a necessary evil: a difficult tool to be used if other methods fail. Why should we give

## Chapter 11

---

### Ancient Babylonian Algorithms

[Written for the 25th birthday of the Association for Computing Machinery; originally published in *Communications of the ACM*, Volume 15, Number 7, July 1972, with errata in *Communications of the ACM*, Volume 19, Number 2, February 1976.]

One of the ways to help make computer science respectable is to show that it is deeply rooted in history, not just a short-lived phenomenon. Therefore it is natural to turn to the earliest surviving documents that deal with computation, and to study how people approached the subject nearly 4000 years ago. Archaeological expeditions in the Middle East have unearthed a large number of clay tablets that contain mathematical calculations, and we shall see that these tablets give many interesting clues about the life of early “computer scientists.”

#### Introduction to Babylonian Mathematics

The tablets in question come from the general area of Mesopotamia (present day Iraq), between the Tigris and Euphrates rivers, centered more or less about the ancient city of Babylon near present-day Baghdad. Each tablet is covered with cuneiform (i.e., “wedge-shaped”) script, a form of writing that goes back to about 3000 B.C. The tablets of greatest mathematical interest were written about the time of the Hammurabi dynasty, about 1800–1600 B.C., and the following comments are based primarily on texts that date from this so-called Old-Babylonian period.

It is well known that Babylonians worked in a *sexagesimal* (radix 60) number system, and that our present sexagesimal units of hours, minutes, and seconds are vestiges of their system. But it is less widely known that the Babylonians actually worked with *floating-point* sexagesimal numbers, using a rather peculiar notation that did not include

## Chapter 13

---

# The IBM 650: An Appreciation from the Field

*[Originally published in Annals of the History of Computing, Volume 8, Number 1, January 1986, a special issue devoted to the IBM 650.]*

I suppose it was natural for a person like me to fall in love with his first computer. But there was something special about the IBM 650, something that has provided the inspiration for much of my life's work. Somehow this machine was powerful in spite of its severe limitations. Somehow it was friendly in spite of its primitive man-machine interface.

I had just turned 19 when I was offered a part-time job helping the statisticians at Case Institute of Technology. My first task was to draw graphs; but soon I was given some keypunching duties, and I was taught how to use the wondrous card sorter. Meanwhile a strange new machine had been installed across the hall—it was what our student newspaper called “an IBM 650 Univac,” or a “giant brain.” I was fascinated to look through the window and see the lights flashing on its console.

One afternoon George Haynam explained some of the machine's internal code to a bunch of us freshmen who happened to be in the lab. It all sounded mysterious to me, but it seemed to make a bit of sense, so I got ahold of some manuals. My first chance to try the machine came a few weeks later, when one of the upperclassmen at the fraternity I was pledging needed to know the five roots of a particular fifth degree equation. I decided that it would be fun to compute the roots by using the 650. More precisely, I had been reading the manual of the Bell Interpretive System [14], and I decided that polynomial root finding would be a good test case.

A program for the Bell System (as we called it) consisted of 10-digit numbers like “1 271 314 577,” which meant, “Add the (floating-point) number in location 271 to the (floating-point) number in location 314

and put the result in location 577." I found a book that gave formulas for the roots of a general fourth-degree equation; so it was easy to factor a general real polynomial of degree 5 by first doing a simple-minded search for a real root  $r$ , then dividing by  $x - r$  and plugging the result into the formulas for quartics.

I realize now how lucky I was to have had such a good first encounter with computers. The polynomial problem was well matched to my mathematical knowledge and interests; and I had a chance for hands-on experience, pushing buttons on the machine and seeing it punch the cards containing the answers. Furthermore the Bell language was an easy way to learn the notion of a program that a machine could carry out. I've forgotten the name of the fraternity brother who asked me to solve this particular problem, but I bet he's kicking himself now for not having done it himself.

I often wonder whether it might not still be best to teach programming to novices by starting with a numeric language like that of the Bell interpreter, instead of an algebraic language like BASIC or LOGO. I think a small child can understand machine-like language better than an algebraic language. But I know that such ideas are now considered out of date, and I suppose I'm being an old fogey.

I learned a few years ago that the Bell interpreter had been inspired by John Backus's Speedcoding System for the IBM 701 [1]. During my student days I had never heard of the 701; and this, I think, leads to an important point: The IBM 650 was the first computer to be manufactured in really large quantities. Therefore the number of people in the world who knew about programming increased by an order of magnitude. Most of the world's programmers at that particular time knew only about the 650, and were unaware of the already extensive history of computer developments in other countries and on other machines. We can still see this phenomenon occurring today, as the number of programmers continues to grow rapidly.

When I did finally learn about the existence of the IBM 701, it had been improved to the 709, and it was shortly to become the 7090; but I must confess that I still liked my good old 650 a whole lot better. The 650 had only 44 operation codes [2], while the 709 had more than 200; yet I never enjoyed coding for the 709, because I never seemed to be able to write short and elegant programs for it—the opcodes didn't blend together especially well. By contrast, it was somehow quite easy and pleasant to do complex things on the 650 with very few instructions. Most of the commands in the 650's repertoire accomplished several things at once, and it was frequently possible to make good use

of the side effects. For example, the instruction

60 1234 1009

meant, "Load the contents of location 1234 into the upper accumulator and the distributor; set the lower accumulator to zero; and then go to location 1009 for the next instruction." All four of these actions were often useful in the subsequent program steps.

In fact, I usually got by with only 34 of the 44 opcodes, because I seldom had a good application for the ten "branch on distributor digit equal to 8" commands. After 25 years I still can remember the numeric codes for most of the remaining 34 ops; and I'll never forget the fact that addresses 8001, 8002, and 8003 referred to the distributor, lower, and upper accumulator registers.

The 650's "one-plus-one address" code, in which each instruction designated the location of its successor (and branch instructions designated both successors) has been rejected by modern machine designers, for good reasons. But it was in fact extremely effective, because it allowed convenient subroutine linkage and because it became easy to execute instructions from registers. A one-plus-one scheme was important, of course, on a machine without random access memory, because it meant that instructions could be located in "optimum" places on the magnetic drum.

The incredible thing about the 650 was that we could do so many things with it, although it was three orders of magnitude slower than today's computers, and it had three orders of magnitude less memory. The memory consisted of 2000 words, where each word had ten decimal digits plus a sign bit; thus, the total storage capacity was less than 10K bytes. These 2000 words were stored on a magnetic drum that rotated at 12,500 rpm; that's slightly less than 5 milliseconds per revolution. One drum revolution was 50 word times. You could add two numbers in 7 word times (672 microseconds) provided that the addition instruction and its operand and the next instruction were located at just the right places on the drum. But addition would take 106 word times (10.2 milliseconds) if the data was pessimally placed. A division instruction took at least 63 word times plus twice the sum of the digits in the quotient.

In practice, the memory space limitation was more important than anything else during my first year of programming. I had to learn how to pack data and how to use subroutines in order to save space. For example, my first large program was a tic-tac-toe routine that "learned" to play by remembering the relative desirability or undesirability of each

position that it had ever encountered. The hardest part was figuring out how to keep one digit of memory for each possible configuration of the board; board positions that were equivalent under rotation or reflection were considered to be identical.

The first program that I ever wrote in machine language still stands out in my mind. It was June, 1957, and my freshman year at Case had just ended. I decided to hang around Cleveland instead of going home, and I was allowed to stay up all night playing with the computer by myself. So I attempted to write a program that would find prime factors. The idea was that a person could set up any 10-digit number in the console switches and start my routine, which would punch the corresponding prime factors on a card and stop; then another number could be set up and factored in the same way, etc. I think my first draft program was about 80 instructions long; but I didn't save it, so I can't be sure. Anyway, I wrote it as a sequence of about 80 decimal numbers, and punched it onto cards—much as I had done with my previous (Bell System) program for root-finding. Then I sat down at the console of the machine and began to learn how to debug, using the "half cycle" switch to step through the instructions slowly, or using the "address stop" switch to discover when the program used particular locations for data or instructions. The 650 console was excellent for on-line debugging, and nobody else was using the machine at that time of night.

Well, my program was riddled with errors, and I removed them one by one during the next two weeks. Besides the "obvious" mistakes, I hadn't realized at first that a 10-digit number can have as many as 33 prime factors. Only eight numbers could be punched on a card, so the program would sometimes have to punch up to five cards. (My original program was only able to punch a single card.) Then I had to clear the memory between runs so that spurious data from a previous factorization wouldn't appear on the next one, and so on. You know the story; we all make the same mistakes. I was lucky enough to have the opportunity to make lots of mistakes right from the beginning, and to diagnose them all by myself, sitting at the machine. All the facts I needed were available to me, because I was working in machine language and no operating system or other software was interposing itself between me and what I needed to know. Debugging took a long time at first, but I think I had the machine to myself about six hours every night.

Finally I arrived at a program that was satisfactory. I vaguely recall that it took about 11 minutes to determine that the number 9999999967 was prime, although at one point this particular test case had taken 17 minutes.

By this time my program had grown to 140 words long, and I think I had changed each of the instructions at least twice. I had also learned about the SOAP assembly language [12], so my final program was expressed in symbolic form; I had been weaned away from numeric machine language during those two weeks. The success of this program gave me the confidence to try another, which converted a given number on the console switches to a specified radix. Then I was ready for tic-tac-toe.

The SOAP language allowed symbols to be up to five letters long, and I recall spending a lot of time trying to come up with suitable names. It was a great moment when I hit on the right term for the program step to be executed when the computer had won at tic-tac-toe by finding three  $\times$ 's in a row: I called that step BINGO.

I regret to report that I've just recently looked again at my programs for prime factors and tic-tac-toe, and they are entirely free of any sort of comments or documentation.

Shortly afterwards I encountered the SOAP II manual [11], which impressed me greatly and had an enormous influence on my subsequent career. This manual included the entire listing of SOAP II in its own language, and the program was absolutely beautiful. Reading Poley's code was like listening to a symphony. I wanted to be able to compose programs like that. The SOAP II program also taught me several new techniques, such as hashing. My next project was to write a modification of SOAP II that would have worked on a 650 with only 1000 words of memory. (I had heard that such machines existed, but I never actually saw one.) Then I spent the rest of the summer writing SOAP III [4], which went the other way by adding additional features for enhanced 650s with index registers and/or floating-point hardware.

SOAP III was my introduction to software writing. In particular, I learned about what is now called "creeping featurism," where each of my friends would suggest different things they wanted in an assembler. I probably tried to accommodate them all, since SOAP III had 24 pseudo-operations that were not in SOAP II. I also left 150 memory locations available for user-defined pseudo-operations. And I put liberal comments into the code, having learned that lesson at last.

Our lab received an amazing 650 program from Carnegie Tech during the summer of '57, namely the famous IT compiler by Perlis, Smith, and Van Zoeren [10]. IT took algebraic statements as input, then computed awhile, and punched SOAP programs as output. I had no idea how such a feat would be possible, but I got a copy of the program listing at the end of the summer and read it while vacationing with my parents at a beach resort on Lake Erie. This program was not beautifully written

like Poley's, but it accomplished remarkable things. So naturally I had an urge to rewrite everything, in the Poley-like style of 650 coding that I had just learned. Bill Lynch and I began this project late in 1957, under the direction of Fred Way III and George Haynam. We first called our program Compiler III, but it eventually became known as RUNCIBLE ([6] and [13]). The language was a superset of Perlis's IT, and we worked very hard to squeeze in as many new features as we could.

Somehow it was possible to cram a rather complex compiler into the 2000 words of the 650. Yet when we were done, I don't think we could have gotten by with only 1999 words, because we had spent considerable time finding every last bit of space—by using terrible tricks. Small changes to one part of our code would usually cause some apparently unrelated part to blow up. I guess Parkinson's Law applies to programs as well as to organizations; we kept adding features until the space was filled.

RUNCIBLE had four versions called AX, AY, BX, and BY, where X stood for object code that invoked subroutines for floating-point arithmetic while Y stood for object code that used the 650's optional floating-point hardware; A stood for SOAP output, while B stood for directly loadable machine language programs punched five per card (bypassing the need for assembly). It turned out that the X version became a Y version by replacing exactly 95 instructions by 95 others; similarly, the A version became a B by replacing exactly 406 instructions by 406 others. If we discovered a way to save one line of code in, say, the A version, we looked closely at the B version until we had saved a line there too.

We called the A version "two-pass operation" while the B version was called "one-pass." At the end of the summer I hacked together a "zero-pass" version that took one less pass than B, since it loaded machine instructions directly into memory locations instead of punching anything on cards. For this I had to eliminate the matrix feature of IT; that is, doubly subscripted arrays were not permitted in "RUNCIBLE zero." My main goal was to prove that 2000 locations were not too few for a compile-load-and-go system, because somebody (Perlis?) had reportedly said that it would be impossible.

By 1959 our lab had acquired the ultimate in 650 upgrades: We had a full 653 system [3] including index registers, floating-point hardware, and 60 whole new words of core memory! It was heavenly. Besides this, we put our printer on-line (so that listings didn't have to be made via cards); we acquired a RAMAC disk storage and several tape units.

At this point it was desirable to have a new assembly program so that we could make proper use of the new equipment. I therefore wrote

SuperSoap [5], a major improvement over SOAP III. I'm still pretty proud of SuperSoap, because it introduced some good ways of dealing with programs that would be loaded into the drum but executed from core, and because I had the courage to remove some features of SOAP III that didn't work as well as planned. Furthermore SuperSoap introduced what I think was the best approach to the problem of "optimizing" the drum locations of data and instructions for the 650; it was a combination of machine and hand methods [7].

The name SOAP, by the way, stood for "Symbolic Optimal Assembly Program"; and "optimal" meant that the machine would choose drum locations so that at least one reference to that location would involve no delay. Such optimization was much better than random placement of instructions. I had (for fun) experimented with what I called SHOAP, a "Symbolic Horribly Optimizing Assembly Program" that used the algorithm of SOAP in reverse: At least one SHOAP'd reference to each location would lead to a 49-word-time delay. By adding seven cards to the normal SOAP program deck, you had SHOAP, which produced extremely slow programs. Conversely, it was possible to improve significantly on SOAP's performance by choosing locations carefully by hand and rewriting the program when necessary, as I discuss in [7]. The Bell Interpretive System had been hand-optimized in a particularly beautiful way, which was quite an inspiration to me. In 1958 I wrote HAND SOAP, which permitted me to hand optimize the locations without giving up the advantage of symbolic assembly. We used HAND SOAP to prepare the run-time system for RUNCIBLE. SuperSoap was designed later to incorporate similar ideas into a full-fledged assembler.

All of this software was given away free, of course. I don't believe my cohorts and I ever thought about making a penny from it. We were motivated by the fact that our programs made it easier for people like ourselves to use marvelous machines like the 650 more effectively. Our ultimate thrill was to find a user who appreciated what we did.

Somebody in 1958 or so circulated a joke about a program called RINSO, a "Real Ingenious New Symbolic Optimizer"; we were carried away by acronyms in those days. For some reason there has been an intimate relation between cleaning agents and the software that I've written through the years, even though my programs haven't always been very clean. For example, John McNeley and I devised a system called SOL in 1963 [9]; when I visited Norway a few years later I learned that SOL is the name of a Norwegian laundry detergent. Even more amazing was that my MIXAL assembler language [8] turned out to have the same name as a popular detergent in Yugoslavia — although I had