Note to readers:
Please ignore these sidenotes; they're just hints to myself for preparing the index, and they're often flaky!

KNUTH

THE ART OF COMPUTER PROGRAMMING

VOLUME 4 PRE-FASCICLE 16A

INTRODUCTION TO RECURSION

DONALD E. KNUTH *Stanford University*

ADDISON–WESLEY

January 1, 2020

Internet
Stanford GraphBase
MMIX

Internet page http://www-cs-faculty.stanford.edu/~knuth/taocp.html contains current information about this book and related books.

See also http://www-cs-faculty.stanford.edu/~knuth/sgb.html for information about *The Stanford GraphBase*, including downloadable software for dealing with the graphs used in many of the examples in Chapter 7.

See also http://www-cs-faculty.stanford.edu/~knuth/mmixware.html for downloadable software to simulate the MMIX computer.

Zeroth printing (revision -98), 1 January 2020

PREFACE

But that is not my point.
I have totally forgotten my point.
— DAVE BARRY (2012)

THIS BOOKLET contains draft material that I'm circulating to experts in the field, in hopes that they can help remove its most egregious errors before too many other people see it. I am also, however, posting it on the Internet for courageous and/or random readers who don't mind the risk of reading a few pages that have not yet reached a very mature state. *Beware:* This material has not yet been proofread as thoroughly as the manuscripts of Volumes 1, 2, 3, and 4A were at the time of their first printings. And alas, those carefully-checked volumes were subsequently found to contain thousands of mistakes.

Given this caveat, I hope that my errors this time will not be so numerous and/or obtrusive that you will be discouraged from reading the material carefully. I did try to make the text both interesting and authoritative, as far as it goes. But the field is vast; I cannot hope to have surrounded it enough to corral it completely. So I beg you to let me know about any deficiencies that you discover.

To put the material in context, this portion of fascicle 16 previews the opening pages of Chapter 8 of *The Art of Computer Programming*, entitled "Recursion." It introduces the topics of that chapter.

At present I've only got a few small scraps of copy that I've occasionally put into my computer, on days when possibly relevant material occurred to me. Thus almost everything you see here is more or less a place-holder for better things that hopefully will come later. Some day, however, I hope that I'll no longer have to apologize for what is now just a bunch of crumbs.

* * *

The explosion of research in computer science since the 1970s has meant that I cannot hope to be aware of all the important ideas in this field. I've tried my best to get the story right, yet I fear that in many respects I'm woefully ignorant. So I beg expert readers to steer me in appropriate directions.

Please look, for example, at the exercises that I've classed as research problems (rated with difficulty level 46 or higher), namely exercises ...; I've also implicitly mentioned or posed additional unsolved questions in the answers to exercises .... Are those problems still open? Please inform me if you know

of a solution to any of these intriguing questions. And of course if no solution is known today but you do make progress on any of them in the future, I hope you'll let me know.

Knuth
HAUPTMAN

I urgently need your help also with respect to some exercises that I made up as I was preparing this material. I certainly don't like to receive credit for things that have already been published by others, and most of these results are quite natural "fruits" that were just waiting to be "plucked." Therefore please tell me if you know who deserves to be credited, with respect to the ideas found in exercises .... Furthermore I've credited exercises ... to unpublished work of .... Have any of those results ever appeared in print, to your knowledge?

* * *

Special thanks are due to ... for their detailed comments on my early attempts at exposition, as well as to numerous other correspondents who have contributed crucial corrections.

* * *

I happily offer a "finder's fee" of \$2.56 for each error in this draft when it is first reported to me, whether that error be typographical, technical, or historical. The same reward holds for items that I forgot to put in the index. And valuable suggestions for improvements to the text are worth 32¢ each. (Furthermore, if you find a better solution to an exercise, I'll actually do my best to give you immortal glory, by publishing your name in the eventual book:–)

Cross references to yet-unwritten material sometimes appear as '00'; this impossible value is a placeholder for the actual numbers to be supplied later.

Happy reading!

Stanford, California D. E. K.
99 Umbruary 2016

For all such items, my procedure is the same:
I write them down — and then write them up.
— DON HAUPTMAN (2016)

CHAPTER EIGHT

COWLEY
SWIFT
ACTON
EL-AHRAIRAH
Adams
recurrence relations
harmonic numbers
Fibonacci numbers
divide and conquer

RECURSION

But then my father, mother, and my brother
Recurse unto my thoughts.

— ABRAHAM COWLEY, *Loves Riddle, A Pastorall Comædie* (1638)

So, Nat'ralists observe, a Flea
Hath smaller Fleas that on him prey,
And these have smaller Fleas to bite 'em,
And so proceed ad infinitum.

— JONATHAN SWIFT, *On Poetry: A Rapsody* (1733)

It is probably time to commit a public heresy by denouncing recursive calculations.
I have never *seen a numerical problem arising from the physical world*
that was best calculated by a recursive subroutine—
that is, by a subroutine that called itself.
I admit the idea is cute and, once mastered, it tends to impel its owner
to apply it wherever possible—all questions of appropriateness aside.

— FORMAN S. ACTON, *Numerical Methods That Work* (1970)

```
curse(int n) = if n>0: say('!'),curse(n-1); if n>1: say('@'),curse(n-2).
```

— EL-AHRAIRAH (1972)

SOME PEOPLE think that "recurse" means "to curse again." Others are fascinated by the idea that things can actually contain themselves.

We've already seen hundreds of examples of recursion in previous chapters. Early on, we encountered sequences of numbers that were defined by relating each member to earlier members after getting started. These "recurrence relations" were sometimes simple formulas like $H_n = H_{n-1} + 1/n$ for the harmonic numbers, sometimes more involved like $F_n = F_{n-1} + F_{n-2}$ for the Fibonacci numbers, and sometimes considerably more elaborate and complex. We also found that basic data structures for trees are inherently recursive, because all but the simplest trees are composed of subtrees. We discussed numerous instances where problems of sorting or searching or optimization could be solved effectively by a "divide and conquer" approach, with which a large problem is reduced to smaller problems of the same kind.

Yet we didn't dwell on the recursive nature of those concepts. We transformed them into equivalent notions that could be dealt with directly and nonrecursively. Because "at bottom" a computer operates by means of the physical properties of electrons, which are *not* recursive.

recursion versus iteration
iteration versus recursion

For example, in Algorithm 2.3.1T we traversed the nodes of a binary tree by using an auxiliary stack `A`, not by using the recursive definition of symmetric order to define that algorithm in terms of itself.

High-level languages for computing — which themselves are defined recursively, because algebraic formulas are built up from algebraic formulas — allow programmers to pretend that recursion is real, that a computer is somehow manufactured from recursive components. But programs that invoke themselves are actually implemented by means of a stack structure behind the scenes. A good programmer is able to understand such algorithms by viewing them simultaneously at a high level and at the machine level, and at many levels in between, thereby perceiving what a real computer can actually do well.

On the other hand, recursion is quite fundamental. It is *not* inherently a high-level concept, suitable only for grownups but not for children. Most people actually grew up with an implicit understanding of rudimentary recursion long *before* they understood iteration — that is, before they understood how to carry out a loop of instructions. Given a list of tasks to do, the simplest mode of operation is a recursive approach that charges blindly ahead:

$$\text{Do Jobs} = \begin{cases} \text{If the list of jobs is empty, stop;} \\ \text{otherwise remove one job from the list,} \\ \text{and do it; then Do Jobs.} \end{cases} \tag{1}$$

It's only *after* we've gained experience with such an almost mindless method that we acquire a more global view of the process:

$$\text{Do Jobs} = \text{Do each job on the list.} \tag{2}$$

Little children don't do algebra; but if we philosophically enumerate the list items abstractly as, say, $\{a, b, c\}$, we can say that the essence of (2) is to elaborate (1) so that it becomes

$$\text{Do } a\text{, then do } b\text{, then do } c. \tag{3}$$

This is a significantly more advanced concept than (1). Even more advanced is to realize that we can accomplish a list of tasks $\{a_1, a_2, \ldots, a_n\}$ via

$$\text{Do } a_1\text{, then do } a_2\text{, } \ldots\text{, then do } a_n \tag{4}$$

or even

$$\text{Do } a_k \text{ for } 1 \le k \le n. \tag{5}$$

All computer programmers have in fact progressed from stage (1) to these later stages rather early in our lives, because of an inner desire to "see through" the entire chain of consequences that result from primitive actions.

Furthermore, once we reached these later stages, we entirely forgot that they were actually elaborations of the recursive procedure in (1). There was no point in narrowing our perspective to the simple-minded form. The vast majority of recursive situations that faced us as children fell into very simple patterns that could readily be "seen through" and understood as a unit. Thus we came to understand iteration as an elementary idea.

author
interpretive routines
compilers
tail recursion

It was only later, when faced with difficult problems whose recursive structure *cannot* be visualized in straightforward terms such as operations on arrays of elements, that we re-learned the importance of recursion. Therefore we now tend to regard recursion as a high-level concept, although it is really fundamental.

Why then is such a basic notion being explored for the first time in Chapter 8, not in Chapters 1 and 2? The short answer is that the author tried a different order of exposition, and didn't like it. He found that much more of importance could be taught by first building up a large collection of worked examples, and by keeping a machine's capabilities firmly in view instead of jumping into the clouds. Each of the case studies in previous chapters has given us insights about how a recursive process can satisfactorily be implemented without recursion. One of the advantages is that we're able to debug programs more easily, as we look "under the covers" at what happens at each level of detail.

Now we're ready to build on those examples and to understand the processes in their proper generality. As a result we'll also know how to implement interpretive routines and compilers that deal with highly recursive constructs.

We'll also learn good ways to save time, when recursions have particularly nice forms. For example, (1) is a typical example of *tail recursion*, when the final operation of a recursive procedure is to invoke itself (or another procedure), without changing any parameters. In such cases there is no need to add a new "return address" to the stack, because the previous return address is already correct. Tail recursion accounts for the fact that (1) reduces easily to the nonrecursive formulations in (2), (3), (4), and (5).

SECTION 8

INDEX AND GLOSSARY

Hippocrates
D'ISRAELI

I, for my part, venerate the inventor of indexes;
and I know not to whom to yield the preference,
either to Hippocrates, who was the first great anatomiser of the human body,
or to that unknown labourer in literature,
who first laid open the nerves and arteries of a book.

— ISAAC D'ISRAELI, *Miscellanies* (1796)

When an index entry refers to a page containing a relevant exercise, see also the *answer* to that exercise for further information. An answer page is not indexed here unless it refers to a topic not included in the statement of the exercise.