Discussion and correspondence
A note on the Towers of Hanoi problem

P. J. Hayes

Department of Computer Science, University of Essex, Wivenhoe Park, Colchester CO4 3SQ

The Towers of Hanoi problem is often used in introductory programming texts to illustrate the power of recursion. In this note, however, I shall investigate the problem a little further and show how a rather different approach yields a very different nonrecursive solution. I shall use this example to illustrate some vague general points about techniques of programming and the meaning of programs.

(Received January 1977)

1. Disclaimer

Everything contained herein is elementary and has almost certainly been discovered many times before. I am writing this note not to claim any originality, but to disseminate these interesting facts more widely, and as an excuse to pontificate.

2. The problem and its recursive solution

There are three pegs, on one of which rests a pile of N discs of decreasing size, the largest at the bottom. The problem is to move all the discs from this peg to one of the other pegs, by moving them one at a time: but subject to the constraint that at no time shall any disc rest upon a smaller disc. No doubt the reader is aware of the (unique) solution strategy for moving discs. I shall, however, pretend that we have to discover the solution anew, as I will wish later to discuss the reasoning which leads to this discovery.

We shall begin by looking at a very simple case. It is not hard to solve the problem for *two* discs. Move the smallest disc to the third (spare) peg, move the largest disc to the target peg, replace the smallest disc on top of the largest disc: and we are done in three moves.

Now look at three pegs. We can move the top two (using the above method) to the spare peg, (using the target peg as temporary spare), transfer the largest across to the target, and move the top two onto the top of it, again using the above method. This takes $3 + 1 + 3 = 7$ moves. How about four discs? Well, we now know how to move three, so we can get them out of the way onto the spare peg, move the bottom one, then transfer the three on top of it, then we've done four discs in $7 + 1 + 7 = 15$ moves.

And so on. The pattern is obvious. To move N discs onto the target peg, we move $N - 1$ discs onto the spare peg, transfer the biggest disc to the target, then transfer the $N - 1$ discs back on top of it. (In retrospect, that even applies to the first case $N = 2$, and the *really* trivial first case is $N = 1$, where we just move the disc directly to where it wants to go. (Well, in fact, one might say that the most trivial case of all is $N = 0$, where we don't do *anything*. We'll discuss this a little more below.))

It's almost as obvious how many moves it takes. Clearly, we're more or less doubling the number of moves by adding a disc, because we have to do *two* transfers of the original pile. So one might guess that it takes about 2^N moves. Look at the low numbers. Two discs is three moves. Three discs is seven moves. Clearly $2^N - 1$ is right, and the proof is immediate:

$$2 . (2^N - 1) + 1 = 2^{(N+1)} - 1 .$$

The algorithm is (I use BCPL notation, but have stolen the algorithm from Dijkstra (1971):

```
let MOVETOWER(M, A, B, C) be
  test M = 1 then MOVEDISC(A, C)
  or $( MOVETOWER(M - 1, A, C, B)
        MOVEDISC(A, C)
        MOVETOWER(M - 1, B, A, C)
     $)
```

where A is the source peg, C the target peg, and B the spare; and MOVEDISC is a suitable routine which actually moves a 'disc' in some data structure representing a state of the problem. (Or perhaps merely prints out the move. If we wanted to be able to print out at each stage a picture of the whole state, we would, of course, have to have a suitable data structure.)

And this is where the matter usually rests.

3. The recursive program is not an intuitive process description

This elegant program illustrates very well the power of recursion and the problem reduction approach to problem solving. Whenever a problem can be reduced to a collection of similar but simpler problems, then recursion can be used to solve it. The recursive algorithm embodying the solution calls itself to solve the simpler subproblems. Such a situation often arises when the problem is to apply some function to a data structure which itself has a heirarchical structure, and the value of the function can be defined in some sense piecewise on the values of the function applied to parts of the data structure, these themselves being data structures of the same kind.

Looked at in this light, the process of discovering the solution to the towers of Hanoi problem, outlined above, is really a process of discovering the hierarchical structure of the problem. Once that is clear, the algorithm is *obvious*.

Obvious, that is, to one familiar with the use of recursion. Algorithms like this are often very surprising to students beginning to program. They understand what it means, but they have trouble believing that it works. And it is hard to explain to them *why* it will work. One has first to give the student faith in recursion, rather than a coherent account of how it is implemented (although the latter might be one way to achieve the former). It is noticeable in many introductory texts that a recursive algorithm is introduced rather in the manner of a conjuror's rabbit from a hat (see for example Dijkstra, 1971, chapter 9).

The beginner's feeling of mystification when faced with a recursive program like MOVETOWER is, I wish to suggest, rooted in something deeper than mere unfamiliarity with the subject. There *is* something odd about elegant programs like this.

The main oddity is that it is very difficult to see what solution process it describes. The solution process is a sequence of moves of discs, and while this is of course determined by the program (by running it), it's not easy to mentally generate the sequence. For example, what is the first move? To run the program requires a stack, and rather a lot of procedure calls and parameter passing before the first move gets made.

Again suppose some call of MOVEDISC generated an error and halted: where could we locate the error in the main program? There's hardly anywhere *to* locate it (only three lines).

Obviously, we would have to refer to the stack trace of calls of MOVETOWER to identify the 'place' where the error occurred. And this entity—the stack—is not mentioned in the program.

Both of these properties contrast markedly with iterative programs, which wear, as it were, their control structure on their sleeve.

Again, there are an awful *lot* of calls of MOVETOWER. To move a tower of N discs takes $2^N - 1$ calls of MOVEDISC, and each such call is textually enclosed in a unique call of MOVETOWER, so there must be that many calls of MOVETOWER. Is all this stack machinery really necessary? If we had a physical model of the towers of Hanoi, would we really need to keep track of all this information (calls of MOVETOWER, with associated parameters) in order to solve the problem?

In spite of its elegance, then, MOVETOWER might reasonably be thought a fairly mystifying and perhaps inefficient program. As though to rub it in, one can make it even more elegant, probably more mystifying, and certainly more inefficient, by taking advantage of the earlier remark that the case $N = 0$ is really the most trivial 'bottom' case, giving the lovely program:

```
let MOVESTEEPLE (M, A, B, C) be
 unless M = 0 do
  $( MOVESTEEPLE (M - 1, A, C, B)
     MOVEDISC(A, C)
     MOVESTEEPLE(M - 1, B, A, C)
  $)
```

I leave it as an exercise for the reader to verify that MOVESTEEPLE does about 50% more work to achieve the same result as MOVETOWER.

The MOVETOWER program specifies how to solve the problem in terms of how to solve its subproblems. Running it, one would know from the text of the program exactly which subproblem one was tackling at each stage. One would be able to answer the question—'*why* are you moving that disc?', at every stage of the solution process. It is exactly this problem-subproblem structure which one needs a stack to keep track of.

But our original brief was to solve the problem, not to explain the solution. Perhaps there is another way. Instead of concentrating on the structure of the *problem*, let us look at the structure of the *solution*.

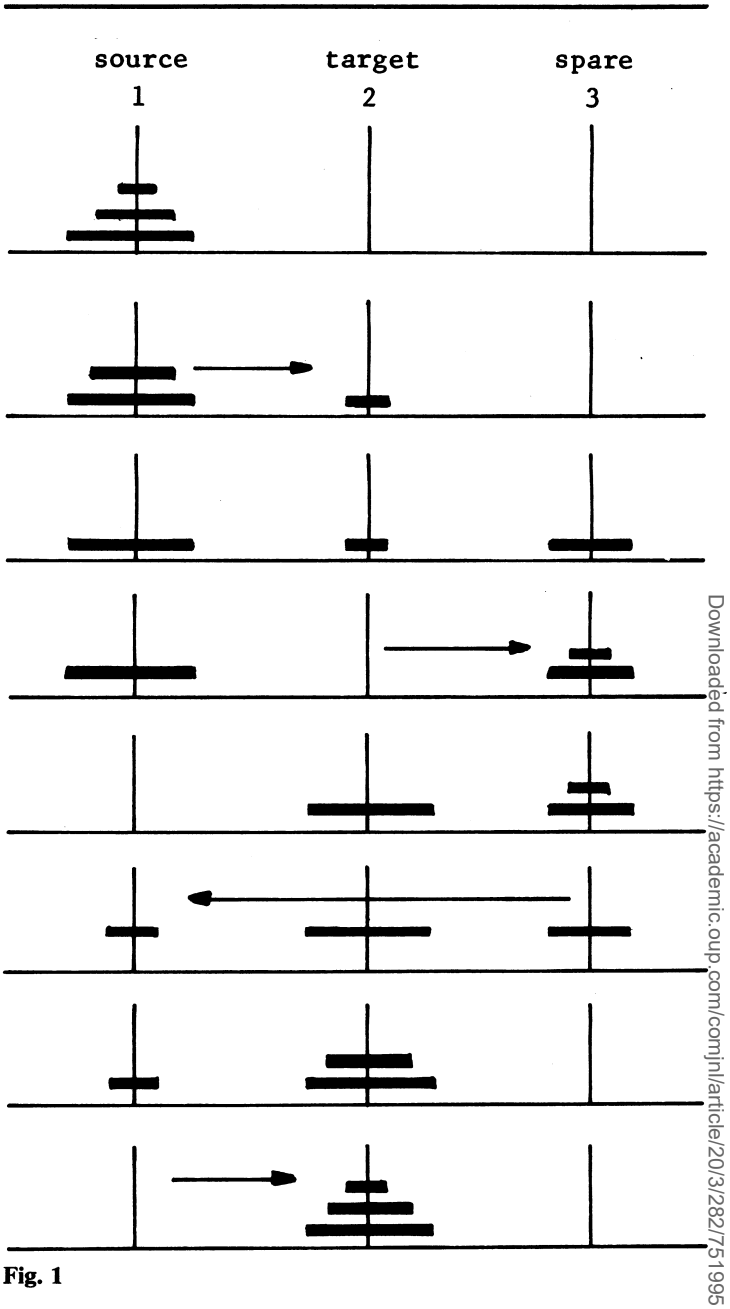

Fig. 1

4. Another approach leads to a finite state solution

Any solution is a sequence of moves of discs, and at some typical intermediate point we can expect that all three pegs will have discs on them. Let us examine the moves which are possible.

First, the smallest disc must be on top of one peg or other. Call this peg A. Of the other two, one must have a smaller disc on the top of it than the other has—call them respectively B and C. Now there aren't many moves which are possible. We can MOVEDISC(B, C), or we can move the smallest disc: MOVEDISC(A, B) or MOVEDISC(A, C). Any other move would result in an illegal position, with a larger disc on top of a smaller disc.

Suppose we moved the small disc. Then at the next move we are apparently faced with a similar choice of moves. But in fact only one makes sense. For if we move the small disc again this time, then either we put it back to its former position, or else we move it to a peg where we could have put it directly in the first move. Either way, a move is wasted. It doesn't make sense to move the small disc twice in succession.

Contrariwise, suppose we MOVEDISC(B, C). On the next move, we can either move the smallest disc, or we can move the disc we just moved back to its former position (it must be smaller than the one which was formerly underneath it, so *that* one can't be moved). But again the latter doesn't make sense, so we have to move the smallest disc. So, small-disc moves must alternate with non-small-disc moves. And, as we have seen, moves of the latter kind are uniquely determined by the situation. This means that if we can decide, at every small disc move, which of the other two pegs to move it to, then the entire process is determined. All we need for a solution is a small-disc moving policy.

The reader might like at this point to work out the correct policy herself. I found it by looking at the small number cases. Take $N = 3$ for example (see **Fig. 1**).

The pattern seems clear: the smallest disc moves cyclically around the pegs, if one imagines them arranged in a circle. To move a 3-tower one step clockwise, the smallest peg must move consistently clockwise. Obviously, if we had chosen the anti-clockwise policy, the tower would have moved anticlockwise.

Now consider four discs. We know that to move a 4-tower from 1 to 2, we have to move a 3-tower from 1 to 3, then from 3 to 2. These are both anticlockwise moves. So to move a 4-tower clockwise, we have to move the smallest disc (and the 3 towers, therefore) *anti*clockwise (**Fig. 2**). Again the pattern is obvious. For odd N, the smallest disc rotates around the pegs in the

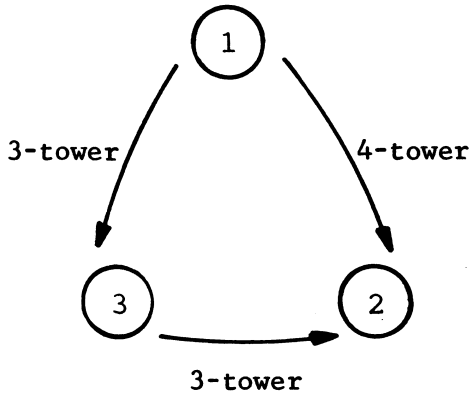

Fig. 2

same sense as the tower. For even N, in the opposite sense. Notice that this is *not* an inductive definition.

Now we can almost write a program. First we must choose a concrete data structure to represent the pegs and discs. This is necessary now because *this* program, unlike MOVETOWER, has to look at the data structure from time to time to decide what to do. A vector of three lists of integers seems suitable, so that for example the size of the top disc on peg 3 is head(peg[3]). The routine MOVEDISC can now be defined:

```
let MOVEDISC(A, B) be
  $( peg[B] := cons(head(peg[A]), peg[B])
     peg[A] := tail(peg[A])
  $)
```

and the program for moving a tower from peg 1 to peg 2 as follows:

```
let HANOI(N) be
  $( let PARITY = (EVEN(N) → DECR, INCR)
     //anticlockwise or clockwise
     let SMALLPEG = 1
     $( let DESTINATION = PARITY(SMALLPEG)
        let A, B = SMALLPEG,
        6-SMALLPEG-DESTINATION
          //destination of the small disc, and the other two
          //pegs
        MOVEDISC(SMALLPEG, DESTINATION)
        SMALLPEG := DESTINATION
        if peg[A] = peg[B] then return // they must both be
                                       //NIL
        test head(peg[A]) > head(peg[B])
         then MOVEDISC(B, A)
         or MOVEDISC(A, B)
     $) repeat
  $)
and DECR(X) = ((X + 1)REM 3 + 1)
and INCR(X) = (X REM 3 + 1)
```

This program is less elegant than MOVETOWER: but also less mystifying and rather more efficient, both in time and memory requirements. It isn't recursive, doesn't need a stack.

5. How different can equivalent programs be?

MOVETOWER and HANOI are about as different as two programs can get. One is irreducibly recursive, the other needs no extra memory. From the former it is obvious (for example) that the number of moves is $2^N - 1$, but not from the latter: while from the latter, but not the former, it is obvious (for example) that the smallest disc is moved first and that every alternate move is a small disc move. One requires some rather peculiar-looking base-3 arithmetic functions, the other doesn't. One needs to inspect the pegs to decide what to do, the other doesn't.

The processes by which they were discovered were also quite different. MOVETOWER is suggested by an inductive description of the problem in terms of subproblems of the same kind. HANOI is suggested by an analysis of the structure of possible move sequences, that is of possible solutions to the problem.

And yet these two programs are in a strong sense equivalent. They certainly define the same function from data structures to data structures (they both move the tower): but more, they actually generate *exactly* the same sequence of moves: tracing MOVEDISC would yield identical traces from the two programs.

Should one therefore say that they have the same *meaning*? This question, of what counts as the meaning of a program, has received a lot of attention. One view is that a program is a specification of how to solve a problem; of a solution *method.* Another view is that a program is a description of a *process.* It is the latter view which is probably the basis of the traditional programmer's intuition, and which underlies almost all the formal semantic theories of programming languages which have been proposed. (Including 'mathematical semantics' when the domains used contain such entities as states of an underlying machine, or continuations, etc. Contrast this with a semantics for a purely applicative language such as pure LISP, in which function definitions denote functions on data objects such as integers and lists.) It is the former view which is emphasised by many proponents of 'structured programming'.

Now these views really are different, I wish to suggest. It really is difficult to think of MOVETOWER as a process description, and if one were faced with HANOI without previous explanation, it would be difficult to say what problem it was solving. And, equally clearly, both views are partly correct. Sometimes the first view leads to an algorithm, sometimes the second does. Neither should be insisted upon as *the* meaning of a program, or *the* best way to approach programming. A good programmer needs to have both ways of thinking (and probably many others) in his mental armoury.

Proponents of the 'process' view might argue at this point that of course MOVETOWER describes a process, viz. the sequence of stack states which the interpreter would go through while running the program. I want to argue against this view. I do not believe that, in composing programs such as MOVETOWER, one thinks at all about the stack evaluation mechanism. Languages such as POP-2 (Burstall, Collins and Popplestone, 1971) which have an explicit stack available for the programmer's use, require quite a different style of thinking. One finds onseself writing such things as

$$x; y; \rightarrow x; \rightarrow y;$$

to swap x and y, for example. In the normal use of recursion, one is thinking rather about problems and subproblems and how to solve them. As further evidence, note the obvious fact that languages which allow recursion rarely have syntactic constructs which allow one to refer to stack frames or any other part of the recursive evaluation machinery. Languages which *do* enable one to refer to such entities, such as CONNIVER (Sussman and McDermott, 1974), have again quite a different flavour and require quite a different style of thinking.

As an interesting exercise in such thinking, the reader might try adapting HANOI to work *without* looking at the data structure of pegs and discs. She will find that it can indeed be done (you have to know the number of discs), but that it is necessary to simulate the stack, probably by using a vector and a pointer.* But then to realise that one has, in effect, painfully

*My colleague I. R. McCallum has implemented such an algorithm in 99 steps on a Texas Instruments SR-56, using a count of the move number to represent the stack.

reconstituted an implementation of the recursive algorithm, is to make a leap of insight. It's not at all *obvious*; which it would be, one supposes, if the recursive algorithm had *meant* such a process all the time.

I feel that an overemphasis on the 'problem reduction' approach to program design—encouraged in recent years by the structured programming crusade—may have a deleterious effect on our students' abilities to think in process structure terms, rather than in problem reduction terms. The most pernicious effect of such an overemphasis is a belief that an elegant and economical *program* structure is to be equated with an elegant and economical *process* structure. The contrast between MOVESTEEPLE and HANOI would be a good counterexample to such a belief. I am not arguing here against structured programming, but against certain aspects of Structured Programming.

MOVETOWER is doubly recursive and cannot be made iterative by a trivial syntactic transformation. The nonrecursive HANOI is possible because the data structure being manipulated contains enough information to decide what to do at each stage, making the control structure—the stack—superfluous. One might draw a (rather weak) analogy with the use of threaded trees to eliminate recursion in tree traversing programs, where the data structure is modified to include just the information needed by the recursive algorithm. We might say that HANOI is an optimisation of MOVETOWER, in the same sense that an iterative *factorial* program is an optimisation of the recursive algorithm, or the iterative 'backstitching' list reversing algorithm is an optimisation of the obvious recursive list reverser. But, I suggest, it would be a very nontrivial exercise to convert MOVETOWER to HANOI, let alone convert it mechanically. In fact, to close, I hereby offer it as a challenge to optimistic optimisers, and to those who make it their business to prove that equivalent programs are equivalent.

References

DIJKSTRA, (1971). *A short Introduction to the Art of Programming.*

BURSTALL, COLLINS, and POPPLESTONE. (1971). *Programming in POP-2*, Edinburgh.

SUSSMAN and MCDERMOTT. (1974). CONNIVER reference manual. *Memo 259A*, A. I. Laboratory, MIT.

To the Editor
The Computer Journal

Comments on 'Fitting data to nonlinear functions with uncertainties in all measurement variables'

Recently in this *Journal*, W. H. Southwell (1976) presented an expansion of an earlier paper (Southwell, 1969) dealing with the important topic of curve fitting and parameter estimation when the function is nonlinear and when errors are assumed to be present in both the dependent and independent variables. His expansion is, fortunately, largely a clarification of many of the misleading statements and a correction of several errors in his 1969 article (See Powell and Macdonald, 1972). Since much of the 1976 paper appears to be an attack on the least squares method published earlier by the present authors (Powell and Macdonald, 1972), it is desirable that a reply be made so that further confusion may be avoided.

1. We *still* claim that, contrary to Southwell's assertion (p. 69), his 1969 method will not in general converge, for *nonlinear* models, to the least squares solution. We invite details of the 'highly successful' (p. 70) use of the 1969 algorithm for such models. None has yet been given by Southwell. It is important here to appreciate the distinction, glossed over by Southwell, between the method he described in 1969 and that described in his 1976 paper.
2. The proof that the expressions for 'exact' parameter variances given in the 1969 article were themselves approximations for all but linear functions was supplied to Southwell by one of us (Macdonald, private communication, 1973). Southwell's fundamental error in statistical analysis is fortunately cleared up in the recent (1976) paper.
3. Our error of $\sqrt{2}$ in parameter standard deviations was corrected in print in 1974 and in 1975, apparently too late for Southwell to acknowledge in his 1976 paper.
4. Southwell is incorrect (p. 71) in asserting that our method depends on convergence of only the a's. In fact, our method depends on convergence of both the x's and a's, as shown by our Eqs. 2 and 3.
5. Southwell's wording is occasionally misleading. For example, on p. 71, it would have been more appropriate to have said: 'The method here described is, with one difference, equivalent to the earlier Powell and Macdonald method.'.
6. Since Southwell has still not provided any illustrations showing convergence to a least squares solution of his 1969 method for functions non-linear in their parameters, it seems odd for him to imply (pp. 70–71) that our 1972 method, which does yield such convergence, is essentially equivalent to the 1969 method. It is not. Even his present (1976) method is not equivalent to ours since his requires evaluation of third-order mixed partial derivatives while ours required only second-order partials, which we actually evaluate numerically.
7. It is true, as Southwell asserts (p. 71), that we did not use the chain rule when applying our analytical method. We were, in fact, simply indicating that such methods *as described* (O'Neill, Sinclair, and Smith, 1969, and Powell, Macdonald, 1972), would not always converge without implicit mixed partials.
8. We note that the one additional iteration required for our method (Southwell's Table 2, 1976) is a small price to pay for not having to supply analytical derivatives.
9. We did not say (p. 72) that the problem $f(x) = a_1 + a_2x$ with uncertainties in both x and y 'remains linear in the parameters'. We *did* say that the least squares condition $\dfrac{\partial S}{\partial x_i} = 0$ can be solved exactly, thus explicitly eliminating the x's, when one has a model which is linear in the independent variable. Southwell correctly points out that there are other models for which $\dfrac{\partial S}{\partial x_i} = 0$ can be solved analytically.
10. We re-did Southwell's example 3 using the Powell–Macdonald method on an IBM 370 computer using partial double precision. Our results are:

$$\begin{aligned} a_1 &= 5{\cdot}914859 \\ a_2 &= -0{\cdot}6035114 \\ a_3 &= -0{\cdot}07996518 \\ a_4 &= 0{\cdot}02619385 \\ a_5 &= -8{\cdot}086482 \times 10^{-4} \\ a_6 &= -1{\cdot}685054 \times 10^{-4} \end{aligned}$$

with $2S = 0{\cdot}4503256$. We used the increment $\Delta = 10^{-4}$ to calculate our numerical derivatives, and convergence was achieved in seven iterations. It is not necessary, as Southwell implies (p. 72), to employ full extended precision on a CDC 6600-type machine to use our estimated partial derivatives program. The solution set obtained from our program is indeed somewhat different from Southwell's results, probably owing to the relatively large parameter standard deviations found in the quintic model. We believe that ours is as good a least squares solution as Southwell's.

Finally, we would like to point out that a technique's value depends