Computational Recursion and Mathematical Induction

URI LERON, RINA ZAZKIS

Introduction

Intuition tells us that the notions of (mathematical) induction and (computational) recursion are intimately related. A recursive *process* invokes a version of itself as a subprocess during execution; a recursive *procedure* contains a version of itself as a subprocedure in its definition; in a proof by mathematical induction, one version of the theorem is used to derive another version. On the other hand, there are differences. For instance, compare the definition of the Fibonacci numbers in mathematics and in Logo. While the definitions look pretty much the same (except for notation), the processes implicitly defined by them show interesting differences. Ask a mathematician in what way the definition specifies, say, the 10th Fibonacci number. Will the answer indicate an "upward" or a "downward" process, or perhaps neither? Ask a programmer (or the computer itself) the same question. Will the answer indicate an "upward" or a "downward" process, or perhaps both?

Our purpose in this article is to elaborate on the relation between the mathematical and computational aspects of recursion, analyse some examples, and speculate on possible education benefits. This treatment, we hope, will shed some light on both aspects, and serve as a basis for further discussion. We consider three major uses of induction in mathematics—definition, proof and construction—and discuss their interconnections as well as their counterparts in computer languages. We illustrate these ideas with Logo procedures that give a computational view of some famous mathematical objects and processes. The word "induction" will always mean in this article mathematical (or "complete") induction.

A first example: recursive definition

We consider the example of the factorial function in order to compare a recursive procedure with one use of induction in mathematics—definition by induction. Further examples along the same line are the definitions of the exponential function and the Fibonacci numbers.

For a natural n, "n-factorial" (written $n!$) is defined informally to be the product of all the natural numbers from from 1 to n, inclusive:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n .$$

Thus, $1! = 1$, $2! = 2$, $3! = 6$, $4! = 24$, etc.

From this informal definition it follows that

$$n! = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n = (n-1)! \cdot n ,$$

which can now be used as the basis for a new, more rigorous definition by induction: The factorial function (denoted by !) is the function satisfying the following conditions:

$1! = 1$, and
$n! = n \cdot (n-1)!$ for all $n > 1$

(Recall that this definition is logically legitimate since it can be proved that there is a unique function satisfying these conditions.)

The computational approach is demonstrated by the following recursive procedure in Logo*:

```
TO FACTORIAL :N
  IF :N = 1 [OUTPUT 1]
  OUTPUT :N * FACTORIAL :N-1
END
```

The similarities between the two definitions are clear. Both have two parts. In the first, the value of the function is defined explicitly for $n = 1$. In the second, the value of the factorial function for any n (greater than 1) is defined in terms of the value of the same function for $n - 1$. This is typical of all recursive definitions, where (to borrow Hofstadter's phrase) a procedure is defined in terms of simpler versions of itself. In this example, "simpler" means the same function with smaller inputs.

Formally, we can use induction to *prove* that the two definitions indeed specify the same function.

Defining a function vs. computing a value

Interestingly, the language that people use to talk about the above definitions (e g. in textbooks) reveals that mathematicians think of the first part of the definition as a "start rule", whereas computer scientists refer to it as a "stop rule". This seems to indicate an intuitive "upward" interpretation of the computational process in the first case and a "downward" interpretation in the second. That is, mathematicians, if pressed to "justify" inductive definition (e g., "why does this definition of $n!$ make sense?") are likely to explain that this definition enables us to compute 1!, then 2!, then 3! and so on to any desired n (each time using the value computed in the previous case). A computer scientist

*Our procedures are written in LCSI Logo. They may need minor modifications to run in other dialects.

is more likely to answer the same question by saying that we can compute $n!$ as soon as we know $(n-1)!$, which in turn can be computed if we know $(n-2)!$, and so on until we reach 1! which is given explicitly. One reason for this difference in the "dynamic" interpretation of the definition is that for mathematicians, the object of interest is *the entire function,* not the computation of a particular value of it. Thus they are thinking of the generation of the entire sequence of the natural numbers, an upward process (it has a beginning but no end). This is also related to the two different ways of viewing the symbol $n!$ or, more generally, $f(x)$: $f(x)$ as standing for *the function* (better denoted by f alone), or $f(x)$ as *the value* of the function f at the particular value x. It also has to do with the "scope" or the "genericity" of x. If we think of x generically, that is as representing "any number", then $f(x)$ can be viewed as a "generic value" of the function, thus representing the entire function. This view brings to mind different images than, say, x and $f(x)$ as representing independent and dependent variables.

Other uses of induction: proof and construction

Perhaps the most prevalent use of recursion in mathematics is in *proof* by induction. This aspect of recursion seems to have no clear counterpart in programming. However, an important link between proof and definition (thus, a link with recursive procedures) is established by mathematical *construction*. On the one hand, consider the relation of construction problems—"construct a certain mathematical object satisfying given requirements"—to definition and proof: To solve the problem, one first *defines* the object, then *proves* that it indeed satisfies the given requirements. (This is not to say that the actual human activity involved in solving the problem actually proceeds in this fashion. The description here applies to the form in which the solution to the problem is eventually formalized.) On the other hand, most nontrivial proofs pivot around an act of construction. It can be demonstrated (see [4, 5]) that such proofs have the following structure: First the construction of a new object is announced whose existence and properties immediately yield the conclusion of the theorem. From here on, the proof is reduced to a construction problem, which is then solved as above (i.e., "define and prove").

Examples

1. THE GRAM-SCHMIDT ORTHOGONALIZATION PROCESS

This process can be used to demonstrate both directions of the above relation between proof and construction. On the one hand, suppose we start with the construction problem: Given a finite-dimensional inner-product space V and a basis B, construct an orthonormal basis E so that every "initial segment" of E spans the same subspace as the corresponding initial segment of B. Then the construction involves definition (of the basis) and proof (that it is orthonormal and satisfies the additional "initial segment" condition). On the other hand, we can view the construction as being part of the proof of the theorem that for every finite-dimensional inner-product space there *exists* an orthonormal basis (with some additional properties). The recursive procedure in Logo for outputting this basis with an arbitrary basis as input (see [6]) corresponds to the definition part of the complete process.

2. THE TOWER-OF-HANOI PUZZLE

This well-known recursive procedure for transferring n rings from one pole to another can be viewed as either a definition (of an algorithm), or as part of proof-of-existence (of a solution or algorithm), or even as part of the proof that the transfer is possible in 2^n-1 moves. (See e.g. [1] and [2] for, respectively, a mathematical and computational treatment of this lovely problem.) The two recursive calls in the Logo procedure correspond to the two uses of the induction hypothesis in the inductive proof.

3. THE PRIME-DECOMPOSITION THEOREM

Once more, we may think of the prime-decomposition procedure as part of a theorem to prove ("Every natural number other than 1 can be expressed as a product of prime numbers") or as a construction problem ("construct an algorithm to carry out the prime decomposition of any given natural number" or, more simply, "given a natural number, construct its prime decomposition"). Both the mathematical and the computational processes are based on the same recursive idea: If the given number is prime, output that number; else, decompose the given number into a product of two smaller numbers and output the product of their prime decompositions. Here is the top level of a Logo implementation of this idea

```
TO DECOMPOSE :N
  IF :N=1 [PRINT [1 has no prime divisors.] STOP]
  IF PRIME? :N [OUTPUT SENTENCE :N []]
  OUTPUT SENTENCE (DECOMPOSE  FIRST
                              FACTOR :N)
                  (DECOMPOSE  LAST
                              FACTOR :N)
END
```

This procedure outputs a list containing the prime divisors of its input, each appearing as often as its multiplicity. (It needs a bit more list processing if you want it to collect equal primes.) It uses two non-primitive subprocedures: PRIME? is a predicate that tests whether its input is prime. FACTOR outputs a two-element list representing a nontrivial factorization of its input. For example, we could have FACTOR 10 output [2 5], FACTOR 105 → [3 35], etc.

We note that while there is nothing fundamentally new in this procedure from the computational point of view, mathematically it represents a different kind of induction, the so-called *strong induction*. Here, to prove the conclusion of the theorem for n, we assume its validity not only for $n-1$, but for all natural numbers smaller than n. (Actually it can be proved that these two forms of mathematical induction are logically equivalent.)

A soft entry

We consider the question of what is a good way to introduce novices to mathematical induction. One way is to confront them with an interesting problem that admits naturally of an inductive solution. Take for example the

problem of how many regions are formed in the plane by n straight lines (Assume no two lines are parallel and no three are concurrent) Since the answer is difficult to see directly from the problem, there is need for investigating. What is the answer for one line? Two lines? And so on. From here on the investigation may branch into two main routes. The more common one is to look for a pattern in the cases investigated (e g., by arranging the results in a table) and generalize. This way one comes to perceive the answer but not the reasons for it. The second route is to look for a pattern not merely in the sequence of numbers, but in the way they are generated. Rather than the original question—how many regions for n lines?—we switch our attention slightly but significantly to the question: How many *more* regions are generated by the addition of one more line? This is where inductive thinking begins This approach is very powerful and potentially very satisfying, for it leads to perceiving the answer as well as the reasons for it, so that it can lead to the feeling that one has *really* understood the solution. Didactically, we would tackle this revised question starting with small numbers, and watch for the emerging pattern as we gradually increase n. Thus suppose we already have 3 lines. How many new regions are formed by adding a fourth line? Can you do that without counting the regions? We would then repeat for 4, 5, etc , until the class is bored. This boredom is welcome and can be put to good use, for what it expresses—"why do we keep playing *the same* game over and over?"—is what we are after From here, the final step of playing the same game *once and for all* is not too big to take; that is, we can give the general argument for going from n lines to $n+1$ lines *even though we do not know the answer for n*. Thus, to summarize, mathematical induction captures *the pattern of the argument* as we go up from one number to its successor. In this respect it is a condensed form of an *upward process*

A similar introduction can be given to recursion in Logo, though this has not been the usual practice Here, too, recursion appears as a condensed form of an upward process. Note, however, that this is not the "down-then-up" process generated by the computer as it interprets the procedure's definition, but the process of the human learner's mind as he or she is working on the problem. The reader is referred to Harvey's book [3], where this approach to teaching recursion is actually taken. To demonstrate, we give a brief account of his approach in introducing a TREE procedure.

One starts by defining procedures TREE1, TREE2, TREE3, etc., each specializing in drawing the tree up to the corresponding level, and each invoking its predecessor as a subprocedure. Thus e.g. TREE3 invokes TREE2 to draw the first two levels, then proceeds to add the next level. After going on long enough (say, up to TREE7), it becomes evident that these procedures (excluding perhaps the first one) are all the same except for the indices. Thus it is not too unnatural to try to collapse all of them into one procedure by simply omitting the indices Trying this out, the need for a stop rule arises and, eventually, a recursive procedure results that draws the tree to any desired level.

In programming, this step-by-step upward approach is mainly limited to novices and is seldom taken by expert programmers. (And even with novices, we do not yet have enough experience to tell when it would be beneficial to use this approach and when some of the other, perhaps more standard, methods.) In mathematics the same observation is true concerning inductive proofs, but mature mathematicians still find this approach useful for *heuristic* purposes, that is, for discovering theorems or proofs. A programmer will occasionally also need to spread out (or "trace") the recursive process for debugging purposes, but this again is a different process—the one taken by the computer in executing the procedure.

Some educational speculations

We consider the question: Which of the two topics is "simpler"—computational recursion (as represented, say, in Logo) or mathematical induction? Which should come first in the learning sequence? For a mathematician or a math teacher, it may seem that induction is simpler since there is less to explain. However, while it is quite usual to see 13-year olds engaged in recursive programming in Logo, one hardly ever tries to teach the same children mathematical induction. By this we do not mean to imply that learning recursion is easy. In fact, classroom experience shows that most of these children are far from having a good grasp of the subtleties involved in recursion. Still, the fact remains that in spite of their partial understanding, many of them manage to successfully use some form of recursion in their programming projects.

It seems, then, that for many people, recursion is initially more accessible than induction. Looking for explanations, we speculate that it is not recursion *per se* that is easier than induction. Rather, it is *recursive activities with the computer* that are easier than *inductive proofs with pencil and paper*. Specifically:

— The *context* for induction is proof (hard) whereas for recursion it is *action* (easy). This affects both the motivation and the ease of learning. For most people, it is easier and more attractive to draw pictures with the computer than to prove propositions about numbers. Also, students have difficulties writing correct proofs in general, and it is possible that inductive proofs are just a special case.

— The *criterion for success* in recursion is the result of your action—an easily observable phenomenon In induction, it is the validity of your proof, a very mysterious entity that even adult students find hard to evaluate. Thus there is more use of corrective *feedback* in learning recursion than in learning induction.

— Recursion may be learned gradually, by bits, starting from graphics-based tail-recursion and progressing through many steps to list-processing-based recursive operations.

— The possibility of playing through dynamic models of

execution (the little people metaphor) greatly enhances the learnability of recursion

Perhaps the whole issue is concisely summarized by thinking of recursion as an "executable version of induction"?

In the future, many students will arrive at the age in which induction is usually taught, with fluency in some recursive computer language. It seems plausible that the teacher could somehow use this fluency as a stepping-stone towards a better understanding of induction.

References

[1] Auerbach, B & Chein, O., *Mathematics: problem solving through recreational mathematics* Freeman, 1980, pp 274, 276-9

[2] Harvey, B , Why Logo?. *Byte*, Vol 7, No. 8, 1982, p. 170

[3] Harvey, B., *Computer science Logo style* MIT Press, 1985, Chapter 5 1

[4] Leron, U., Structuring Mathematical Proofs. *American Mathematical Monthly*, Vol. 90, 1983

[5] Leron, U , Heuristic Presentations: The Role of Structuring *For the Learning of Mathematics*, Vol. 5, No. 3, 1985

[6] Zazkis, R , Recursion—a Powerful Idea in Mathematics, Computer Science and Education. Unpublished M A. Thesis., Haifa University, 1985

Continued from page 24

up the above study by developing a revised treatment of functions which is being trial-tested in the present school year. Although the trials are not complete, it seems that already some of the difficulties (e g., student addiction to linear functions, the positioning of images and preimages correctly on the axes) are being lessened.

School curricula in general, and maths curricula in particular, have been buffeted by all sorts of "winds of change" in the last quarter of a century. Neither the good nor the bad have had sufficient time to justify their retention or removal. Making even a small part of a maths curriculum work is not likely to be achieved by jumping from band wagon to band wagon (even if the latter is computerised rathern than horse drawn) Maybe maths educators will get a better return for their effort by applying themselves seriously to understanding the difficulties in present curricula and overcoming them gradually

References

Boyer, C.B. Proportion, equation, function. Three steps in the development of a concept *Scripta Mathematica* (1946) Vol. 12., 5-13

Buck, R C. Functions In E. Begle (ed) *Mathematics education.* Yearbook of the National Society for the Study of Education (1970) Vol. 69, 236-239

Godfrey, C. The algebra syllabus in the secondary school In *Special reports on educational subjects 26. the teaching of mathematics in the United Kingdom,* Part I (1912) HMSO, London, 280-311

Hight, D W. Functions: dependent variables to fickle pickers *The Mathematics Teacher* (1968) Vol. 61, 575-579

Karplus, R. Continuous functions: students' viewpoint *European Journal of Science Education* (1979) Vol. 1, 379-415

Khinchin, A Y. *The teaching of mathematics.* English Univ. Press, 1968

Malik, M A. Historical and pedagogical aspects of the definition of function, *International Journal of Mathematics Education in Science and Technology* (1980) Vol. 11, 489-492

Markovits, Z., Eylon, B., Bruckheimer, M Functions: linearity unconstrained, in *Proceedings of the 7th Conference of the International Group for the Psychology of Mathematics Education* (1983) 271-277

Marnyanskii, J A. Psychological characteristics of pupils' assimilation of the concept of a function, in *Problems of instruction Soviet Studies in the Psychology of Learning and Teaching Mathematics* (1969) Vol. 12 Stanford: School Mathematics Study Group 163-172

Read, C.B. The treatment of the concept of function *School Science and Mathematics* (1969) Vol. 69, 695-696

SMP (School Mathematics Project) *Book 2* Cambridge University Press (1969) 158

SMSG (School Mathematics Study Group), *Intermediate Mathematics* Yale University Press (1960) 166

Vinner, S. Concept definition, concept image and the notion of function, unpublished manuscript, 1979