STUDYING MATHEMATICAL INDUCTION AND RECURSIVE PROGRAMMING TOGETHER*

Keith Brandt
Mathematics, Computer Science, and Physics
Rockhurst University
Kansas City MO, 64110
keith.brandt@rockhurst.edu

Margaret Richey
Department of Computer Science
University of Wisconsin
Madison WI, 53706
richeym@cs.wisc.edu

ABSTRACT

Mathematical induction is a proof technique used throughout math-ematics, and recursion is a programming concept frequently used in computer science. This note will explore the parallel between induction proofs and recursive programs by providing several example problems that lead to an induction proof and a corresponding recursive program. We feel that students who are exposed to this parallel will gain a deeper understanding of both topics.

Key Words: mathematical induction, recursion, algorithms.

INTRODUCTION

Mathematical induction is a proof technique used throughout mathematics, and recursion is a programming concept frequently used in computer science. Both topics are essential and are used in courses such as discrete mathematics, data structures, algorithms, and theory of computation. We suggest developing this relationship into an effective teaching tool. Students of both computer science and mathematics would benefit from this interdisciplinary view of two closely related topics.

We provide several examples of results whose induction proofs correspond to a recursive program. In particular, we will focus on proofs that show a certain task can be performed. The corresponding program will then carry out the task. This parallel is implicitly included in some textbooks (see Examples 4 and 6 below) and is explicitly mentioned in others (see [2, p. 89]). In fact, all but one of our examples come from

---

standard texts. We have not, however, found many books whose examples discuss the proofs and the programs together. Nor have we seen many exercises that ask the student to complete both the proof and the program. We feel that the examples in this note are best suited for students in discrete mathematics or data structures courses.

EXAMPLE PROOFS AND PROGRAMS

We have selected several examples of varying difficulty that illustrate the similarity between induction proofs and recursive programs. They can be presented in class or assigned as projects. Also, instructors can experiment with the order of events. For example, instructors could present a proof and then ask their students to write the corresponding program or vice versa. Or instructors could assign both the proof and the program. In this case, it is interesting to see whether students first work on the proof or the program. Clearly, the answer will depend on the student and the particular problem assigned. In either case, the writing of one part-the proof or the program-facilitates the writing of the other.

Example 1

The fundamental theorem of arithmetic is an excellent introductory example.

Theorem: Any integer n greater than 1 can be written as a product of primes.

Proof: The proof uses complete (strong) induction. If n is not prime, write it as a product of two smaller integers, say r and s, and then apply the induction assumption to r and s. See, for example, [5, p. 198].

Program: The program follows the same train of thought. Given an input n, return n if n is prime. Otherwise, write n as a product of smaller integers r and s (find them anyway you like) and call the program on them. For our purposes, we are not concerned with the complexity of the program. Instructors may want to discuss with their students different ways to find r and s and describe how these choices affect the efficiency of the program.

Example 2

Theorem: Every natural number can be written as the sum of distinct powers of two [6, p. 63].

Proof: Use complete induction. Clearly $1 = 2^0$. Now assume the statement is true for all natural numbers less than or equal to k. Show the statement is true for k+1.

If k+1 is even, then k+1 = 2m, where 1 # m # k. Thus, by the induction assumption, $m = 2^{(a_1)} + 2^{(a_2)} + ... + 2^{(a_p)}$, where $0 = a_1 < a_2 < ... < a_p$. Then,

$k+1 = 2(2^{(a_1)} + 2^{(a_2)} + ... + 2^{(a_p)}) = 2^{(a_1+1)} + 2(a_2+1) + ... + 2(a_p+1)$.

If k+1 is odd, then we can write $k = 2^{(b_1)} + 2^{(b_2)}+ ... + 2^{(b_q)}$, where $0 < b_1 < b_2 < ... < b_q$. Then,

$$k+1 = 2^{(0)} + 2^{(b_1)} + 2^{(b_2)}+ ... + 2^{(b_q)}.$$

This proof is an outstanding example of complete induction. It complements the standard examples where complete induction is used in factorization theorems for integers and polynomials. Students could also be asked to discuss (and prove) the uniqueness of the decomposition. Furthermore, the proof establishes a basic fact that is used throughout computer science. The proof also very clearly suggests a recursive algorithm to write any natural number as the sum of distinct powers of two.

First version of program: This program follows the logic of the proof very closely. The output is not simplified.

```
procedure SumPower2 (n: integer)
if n > 1 then
        if n even then
             write ( "2*(" )
             SumPower2 (n/2)
             write ( ")" )
        else
             write ( "20 +")
             SumPower2 (n-1)
    else write ( "20" )
end procedure SumPower2
```

For example, calling `SumPower2(11)` produces the output

```
20 + 2*(20 + 2*(2*(20))).
```

Second version of program: Students could also simplify the output. This program uses an array a of zeros and ones that keeps track of the powers of two being used. Given an input, the procedure UpdateArray assigns the appropriate combination of zeros and ones to the array a. The procedure SimpSumPower2 calls UpdateArray and then reads through a and prints the corresponding powers of 2.

```
procedure UpdateArray (n, j: integer)
    if n > 1 then
        if n even then
             UpdateArray (n/2, j)
             (*  Now shift contents of array:  *)
             for i:=1 to j do a(j+1-i):= a(j-i)
             a(0):=0
        else
             UpdateArray (n-1, j)
             a(0):=1
    else a(0):=1
end procedure UpdateArray

procedure SimpSumPower2 (n: integer)
    j:= Floor(log2(n))
    for i:=0 to j do a(i):=0
    UpdateArray(n, j)
    for i:=0 to j do
        if a(i) <> 0 then write ( "+2^", i)
end procedure SimpSumPower2
```

For example calling `SimpSumPower2(2220)` produces the output

```
+2^2 + 2^3 + 2^6 + 2^7 + 2^11.
```

Note the obvious similarities between the proof and the program(s). Both separate the cases n even and n odd. The induction step in the proof corresponds to the recursive call in the program.

Example 3

Theorem: For every positive integer n, the expression $7^n - 2^n$ is a multiple of 5 [3, Exercise 38, p. 108].

Proof: Clearly, $7^1 - 2^1 = 5$. The induction step relies on the following algebra:

$7^k - 2^k = (5+2)*7^{k-1} - 2*2^{k-1} = 5*7^{k-1} + 2*(7^{k-1} - 2^{k-1})$.

Program: With the proof at hand, it is now easy to write a program that writes $7^n - 2^n$ as a multiple of 5. Given an input n, the procedure SevenTwoFive finds an x such that $7^n - 2^n = 5x$.

```
procedure SevenTwoFive (n: integer,  var x: integer)
    if n=1 then x:= 1
    else
        SevenTwoFive (n-1, x)
        x:= 7n-1 + 2x
end procedure SevenTwoFive
```

For example, executing the lines

```
n:=5
SevenTwoFive (n, x)
write("7^", n," - 2^", n," = ",5*x," = 5*",x)
```

produces the output `7^5 - 2^5 = 16775 = 5*3355.`

Example 4

Theorem: Any postage of 8 cents or more can be obtained using 3 and 5 cent stamps [3, Example 24, p. 105].

Proof: Clearly, 8 = 5 + 3. Now assume the statement is true for some k = 8, and show it is true for k+1. Write k = 3a + 5b where a and b are non-negative integers. If b > 0, we can remove one 5 cent stamp and replace it with two 3 cent stamps. In other words, we can write

$$k+1 = 3a + 5(b-1) + 6 = 3(a + 2) + 5(b-1).$$

If b = 0, then we must have a = 3. In this case, we can remove three 3 cent stamps and replace them with two 5 cent stamps. Then,

$$k+1 = 3(a-3) + 5b + 10 = 3(a-3) + 5(b+2).$$

Program: Once again the flow of the program is nearly identical to that of the proof. The procedure Stamps returns integers a and b corresponding to the number of 3 and 5 cent stamps respectively. See [3, Exercise 4, p. 158].

```
procedure Stamps(n: integer, var a, b: integer)
    if n=8 then a:=1; b:=1
    else
        Stamps(n-1, a, b)
        if b > 0 then a:= a+2; b:= b-1
```

```
        else a:= a-3; b:= b+2
end procedure Stamps
```

For example, executing the lines

```
n:=13
Stamps(n, a, b)
write(n, "=3*", a,"+5*", b)
```

gives the output `13 = 3*1 + 5*2.`

Example 5

Our next example (see [4]) is also the most difficult. We will outline a proof and a program and let the reader fill in the details or, perhaps, devise another approach (see, for example, [1]). Recall that the Fibonacci numbers are defined (recursively) by f1 = f2 = 1 and fj+2 = fj+1 + fj.

Theorem: For each positive integer n, there is a positive integer k and integers $a_1, a_2, ..., a_k$ 0 {1,2} such that $n = a_1f_1 + a_2f_2 + ... + a_kf_k$.

Proof: We first introduce some notation and terminology. Denote the positive integers by N. For a positive integer k, let [k] = {1, 2, ..., k}. Let W be the set of all finite strings of 1's and 2's. Make one minor (and artificial) modification to W. If the last digit of a word is 2, add the digit 0 to the end of the word. In what follows, if the last digit is 0, it will be ignored or eventually changed to a 1. Thus,

$W = \{ a_1a_2...a_k \mid a_j$ 0 {1,2} (1 # j # k-1), a_k ({0,1}}.

For each $w = a_1a_2...a_k$ 0 W, let $s(w) = a_1f_1 + a_2f_2 + ... + a_kf_k$. Let S = {s(w) | w 0 W}. Clearly, S is a subset of N. We wish to show that S = N.

Let $w = a_1a_2...a_k$ 0 W. We make some additional definitions. Say an index j0[k] is relevant if j > 1 and $a_{j-1} = a_j = 2$. Let Rel(w) = {j([k] | j is relevant}.

Let count(w) = 3(k-j), where the sum is over all j0Rel(w). Say w is reduced if Rel(w) is empty. Thus if w is reduced, count(w) = 0 and w has no consecutive 2's. The function count is a measure of how far a word is from being reduced.

For example, let u = 1221220 and v = 22212211. Then, Rel(u) = {3, 6},

count(u) = (7-3) + (7-6) = 5, and s(u) = 1(1) + 2(1) + 2(2) + 1(3) + 2(5) + 2(8) = 36.

Also, Rel(v) = {2, 3, 6}, count(v) = (8-2) + (8-3) + (8-6) = 13, and

s(v) = 2(1) + 2(1) + 2(2) + 1(3) + 2(5) + 2(8) + 1(13) + 1(21) = 71.

Now use the following:

Lemma: Each word in W can be reduced. That is, for each w0W, there is a reduced $w_1$0W such that $s(w_1) = s(w)$.

To prove the lemma, use complete induction on the count of w. The count can be reduced by finding the rightmost pair of consecutive 2's (i.e. the largest relevant index). Replace these 2's with 1's and increment the digit immediately following them.

Finally, we must show that for each n0N, there exists a w0W such that s(w) = n. This proof follows by regular induction on n and use of the above lemma.

Program: Store words in an array a consisting of 1's and 2's. We can write a recursive procedure Reduce that reduces words by following the approach described above. The main body of the program will also use recursion. Note that once a word $a_1a_2...a_k$ is reduced, we know that at least one of a_1 and a_2 is equal to 1. Given an input n, the procedure Expand returns the array a containing the corresponding sequence of 1's and 2's.

```
procedure Expand(n: integer)
    if n=1 then a(1):=1
    else
        Expand(n-1)
        Reduce (a)
        if a(1)=1 then a(1):=2
        else a(2):=2
end procedure Expand
```

For example, calling Expand(7) returns a = 212, which is equivalent to saying

$7 = 2(1) + 1(1) + 2(2)$.

Example 6

Our last example is treated in several discrete mathematics texts. We will let the reader supply the details.

Theorem: For any positive integer n, if any one square is removed from a $2^n \times 2^n$ checkerboard, then the remaining squares can be covered with L-shaped pieces consisting of 3 squares each.

Proof: See [5, Example 12, p. 196].

Program: Students could be asked to write an interactive program that lets the user select the square to be removed and then covers the remaining squares with L-shaped pieces [5, Computer Project 3, p. 229].

CONCLUSION

We have presented several examples that demonstrate a parallel between induction proofs and recursive programs. Although not every induction proof has a corresponding recursive program and vice versa (instructors may want to include examples that make this point), we believe studying the two topics together will give students a better understanding of both. Some students may prefer to work out the details of the proof before starting on the program; others may prefer to begin with the program.

This paper grew out of a student research project completed by the second author under the supervision of the first. The second author, who intends to pursue a career in computer science education, was eager to study an issue relating to pedagogy. She wrote proofs and programs for most of the examples presented here. For Examples 1 and 2, she first wrote the proofs and then used the ideas in the proofs to develop the programs. For Example 4, she first wrote the program and immediately saw how to "translate" it into the proof. For the more difficult Example 5, she worked for some time to first devise general "plan of attack." Armed with a plan, she chose to work on the program first, believing

that task to be more concrete and manageable. Once the program was running, she was confident that her approach worked and was motivated to tackle the technical notation and definitions needed for the proof. For each example, she relied heavily on the ideas used in the first task to compete the second. Writing the code-and seeing it work-not only boosted her confidence, but also helped solidify her understanding of the details of a proof. By enumerating a process and coding it herself, she got hands-on experience with the logic behind the mathematical principles used in the solution of each problem. Not only do these principles include the induction process, but also the implicit recursive solution as well as taking a different look at the math used in the induction step, which may not be intuitive (Example 5). Furthermore, she deduced that different versions of the program could lead to different versions of the proof. For instance, in Example 4, one could check more base cases, call Stamps(n-3, a, b), and then increment a. There is, clearly, a corresponding version of the proof. The code for Examples 1, 2, 4, and 5 is posted on her website: http://www.cs.wisc.edu/~richeym/pub.html. We ask readers to share with us other examples that demonstrate this parallel.

One final thought: We hope this note will call further attention to the more general connection between writing proofs (especially constructive proofs) and writing programs. Both require logical sequencing of ideas, proper syntax, and attention to detail. For this reason, we feel students should study both topics extensively.

ACKNOWLEGMENT

We thank Kevin Burger for many helpful comments.

REFERENCES

[1] Abad, J.A.P. Solution to Problem Proposal #1657. *Mathematics Magazine*. Vol 76. no. 5 (December, 2003), p. 400.

[2] Bailey, D.A. *Java Structures* (second edition). McGraw Hill (2002).

[3] Gersting, J.L. *Mathematical Structures for Computer Science* (fourth edition). W.H. Freeman and Co. (1999).

[4] Lampakis, E. Problem Proposal #1657. *Mathematics Magazine*. Vol. 75, no. 5 (December, 2002), p. 399.

[5] Rosen, K.H. *Discrete Mathematics and Its Applications* (fourth edition). McGraw Hill (1999).

[6] Schumacher, C. *Chapter Zero* (second edition). Addison-Wesley (2001).