Journal of Humanistic Mathematics

Volume 11 | Issue 1 January 2021

Computational Thinking in Mathematics and Computer Science: What Programming Does to Your Head

Al Cuoco
Education Development Center

E. Paul Goldenberg
Education Development Center

Cover Page Footnote
Supported by NSF Grant #1441075, The Beauty and Joy of Computing.

Computational Thinking in Mathematics and Computer Science: What Programming Does to Your Head

Al Cuoco

Education Development Center, Waltham, Massachusetts, USA
acuoco@edc.org

E. Paul Goldenberg

Education Development Center, Waltham, Massachusetts, USA
EPGoldenberg@edc.org

Synopsis

How you think about a phenomenon certainly influences how you create a program to model it. The main point of this essay is that the influence goes both ways: creating programs influences how you think. The programs we are talking about are not just the ones we write for a computer. Programs can be implemented on a computer or with physical devices or in your mind. The implementation can bring your ideas to life. Often, though, the implementation and the ideas develop in tandem, each acting as a mirror on the other. We describe an example of how programming and mathematics come together to inform and shape our interpretation of a classical result in mathematics: Euclid's algorithm that finds the greatest common divisor of two integers.

How you think about a phenomenon certainly influences how you create a program to model it. The main point of this essay is that the influence goes both ways: *creating programs influences how you think*.

The programs we are talking about are not just the ones we write for a computer. Programs can be implemented on a computer or with physical devices or in your mind. The implementation can bring your ideas to life.

Often, though, *the implementation and the ideas develop in tandem*, each acting as a mirror on the other.

This is not a new thesis. Researchers have studied how the use of "artifacts" is shaped by and shapes the user (see [6, 7, 5, 1, 3], for example). In what follows we'll give some examples that show how our own thinking is co-formed by the habit of working in a functional programming language.

First, a caveat: Most real programing languages, including the ones we've used, support all kinds of thinking. But we are using a small subset of the features of these modern languages, one where there's no data mutation, no iteration, and no variable assignment. Programs are models of mathematical functions that pass values around to each other, and the values can be any kind of data, including functions themselves. These languages are all in the Lisp tradition; here we use Snap*!*, the language used in our *Beauty and Joy of Computing* curriculum (an AP *CS Principles* course available at `https://bjc.edc.org`).

Example: Unstacking a recurrence

Recursive thinking has been used for a long time. One of the earliest uses of recursion in arithmetic surely has to be Euclid's formulation of an algorithm (now named after him) to compute the greatest common divisor for two non-negative integers. If a and b are such numbers, their greatest common divisor, $\gcd(a, b)$ is, as the name suggests, the largest integer that is a factor of both a and b. So, for example, the $\gcd(124, 1028)$ is 4.

Greek mathematicians used a process called *antanairesis*, a free translation of which is "back and forth subtraction" when they realized that one consequence of the arithmetic structure of the integers is that

$$\text{if } a < b, \text{ then } \gcd(a, b) = \gcd(b - a, a)$$

If, as in the case of $\gcd(124, 1028)$, one such subtraction leaves the first argument to gcd greater than the second (that is, $b - a > a$), no matter, because $\gcd(b-a, a) = \gcd(a, b-a)$ so we can just subtract that smaller value again, gradually whittling the larger of the two original numbers down until the result is less than the smaller of the two original values. That repeated subtraction is division, and the result we're keeping is the remainder.

A Story From AC

I (AC) first came to understand this process for executing Euclid's algorithm through the way it's typically developed in algebra and number theory: write out all the divisions and use the fact that the remainders decrease, and hence eventually vanish. Written out in general, the details look onerous. For gcd(a, b), it looks like this:

> Set $b = r_0$ and $a = r_1$, so that the equation $b = qa + r$ reads $r_0 = q_1 a + r_2$. There are integers q_i and positive integers r_i such that
>
> $$\begin{aligned} b = r_0 &= q_1 a + r_2, & r_2 &< a \\ a = r_1 &= q_2 r_2 + r_3, & r_3 &< r_2 \\ r_2 &= q_3 r_3 + r_4, & r_4 &< r_3 \\ &\vdots & &\vdots \\ r_{n-3} &= q_{n-2} r_{n-2} + r_{n-1}, & r_{n-1} &< r_{n-2} \\ r_{n-2} &= q_{n-1} r_{n-1} + r_n, & r_n &< r_{n-1} \\ r_{n-1} &= q_n r_n \end{aligned}$$

This is one of the more tame versions, taken from [4]. It's not as bad as it looks, especially if you try it with numbers, say, 124 and 1028. It is presented as an iterative sequence of steps, but if you do this, you get the feeling that it's "always the same." Sure, the numbers change, but the *steps* are the same—the division, the swapping of quotients and remainders—and the rhythm builds as you work it through.

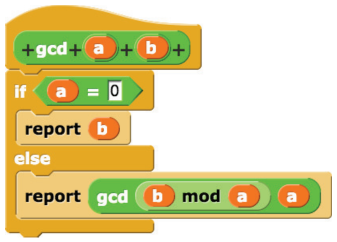
For me, it stayed that way for years—a general feeling that the equations above described a machine, one that could be run on any pair of integers to produce their gcd.

That machine became real when one of my high school students introduced me to Logo. I was immediately drawn to the feature of the language that allows you to build computational models of recursively defined mathematical functions. This seemed miraculous to me, and it gave me the inspiration to express the essence of the "general feeling" I had about processes like the one above as *recursively defined computer programs.* That is, the essence of

Euclid's algorithm is that

$$\gcd(a, b) = \gcd(b \bmod a, a).$$

This one concise statement captures what the algorithm does and expresses what it does as a statement about a *function*. It captures the essence of the iterative process expressed in the entire seven-line, two-column set of symbols above. By showing $\gcd(a, b)$ defined in terms of the gcd of two smaller numbers, it even clarifies how the process can end when one of the numbers vanishes. Getting a computer to execute it requires telling the computer when that "final step" has been reached, and that's all:



```
+gcd+ a + b +
if a = 0
    report b
else
    report gcd (b mod a) a
```

Figure 1: Euclid's Algorithm, expressed in Snap*!*

Expressed in conventional notation, this becomes

$$\gcd(a, b) = \begin{cases} b & \text{if } a = 0 \\ \gcd(b \bmod a, a) & \text{if } a > 0 \end{cases}$$

Our colleague Brian Harvey recently talked about "the most beautiful piece of code." Code is an articulation of the thinking that created it; if the thinking is elegant, and the computer language supports that thinking, then the elegance of the thinking is what makes the code beautiful.

I can't remember which idea came to me first—the computer program or the realization that Euclid's algorithm can be viewed as an inductively defined *function of two variables*. Most likely, the two ideas developed in tandem, with refinements of one helping to refine the other. But I'm certain of one thing:

> *The insight that Euclid's algorithm can be expressed as a recursively defined function co-evolved from my attempts to model it as a recursively defined computer program.*

The elegant one-line recursive function idea that summarizes the many-step iterative process shown above was *not* in the books at that time. It's not an exaggeration to say that my early book on algebra and number theory [2] was built from the thinking that went into this.

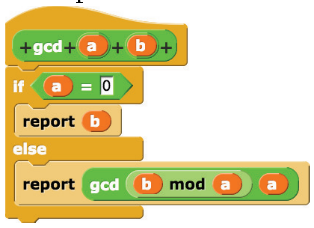
———o———0———o———

So, we have two models of Euclid's algorithm, each expressed as a function of two variables:

- The mathematical function defined by

$$\gcd(a,b) = \begin{cases} b & \text{if } a = 0 \\ \gcd(b \bmod a, a) & \text{if } a > 0 \end{cases}$$

- The computer program expressed as



```
+gcd+ a + b +
if ⟨ a = 0 ⟩
    report b
else
    report gcd (b mod a) a
```

Carefully playing with the program and its structure can lead to results that seem to drop out of the sky in many number theory books. For example, the teachers with whom we work like to "play computer" and arrange the execution of $\gcd(124, 1028)$ like this:

```
          8
    124) 1028
         -992     3
           36) 124
              -108   2
                16) 36
                   -32   4
                      4) 16
                        -16
                          0
```

Another feature of computational thinking is to look at the *form* of algorithms, rather than their output, seeking some kind of regularity in the execution of the operations. This habit develops with experience, learning to see the value in "stepping back" to see if you are performimg the same operations, over and over, independent from the inputs to those operations. It's all about rhythm, and, as in music, the rhythm is an ingredient in the elegance of a program. For example,

Imagine that one of the authors spilled his wine (a 2010 Barbera) over the other author's calculation:

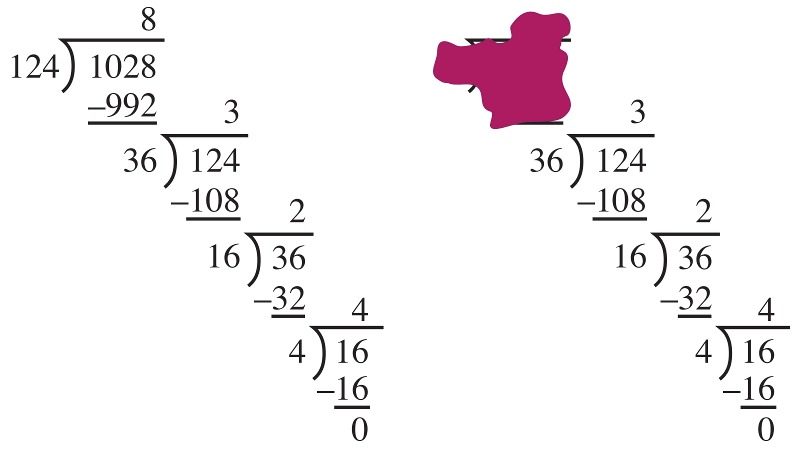


Figure 2: A clumsy author

What's left is the calculation of gcd(36, 124). So gcd(36, 124) is also 4. And more spills show that each pair in the sequence produces a smaller version of the same calculation. The fact that you can pick up the calculation at any step makes the recursive call in Euclid's algorithm come alive.

Another example: The last non-zero remainder is 4, so $4 = \gcd(124, 1028)$. It seems as if we should be able to unstack all this, starting with 4, winding our way all the way up to the top. The habit of looking for such things is yet another aspect of computational thinking: the habit of analyzing computations to see what stories they have to tell. Let's look at this idea in more detail.

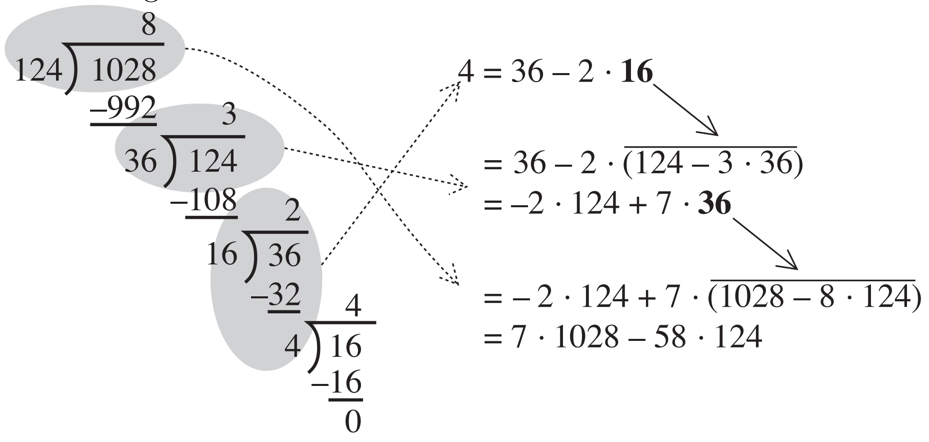
Well, start at the bottom—look at that next-to-last division that produces 4. It says that 4 is the remainder when 36 is divided by 16. More precisely, it says that

$$\mathbf{4} = 36 - 2 \cdot \mathbf{16}$$

But now 16 plays the same role in the next-up division that 4 plays here:

$$\mathbf{16} = 124 - 3 \cdot \mathbf{36}$$

As Van Morrison said so well [8], it's too late to stop now. We can unstack the whole thing like this:



And look: we've written 4 as a *linear combination* of 124 and 1028. That is, we can express 4 as *something times 124* plus *something else times 1028.* And there's nothing special about 124 and 1028—it will always be possible to write $\gcd(a, b)$ as a linear combination of a and b.

But "it will always be possible" is too wimpy. Computational thinking demands we ask *how* one can write $\gcd(a, b)$ as a linear combination of a and b. That is, there must be a way to find the numbers to multiply a and b by. These numbers will depend on a and b, so there must be two new functions of a and b—let's call them s and t—so that

$$s(a, b)\, a + t(a, b)\, b = \gcd(a, b)$$

For example,

$$s(124, 1028) = -58 \quad \text{and} \quad t(124, 1028) = 7$$

It would be worth it here to stop reading and work out a few more examples. Then try to build computational models of s and t. We do that next, but don't look until you try it for yourself . . . .

Here we go. . .

We're on a hunt for the functions s and t. A function is determined by its values and its domain. The domain for both is the set of integers. The values? It would be great to have an explicit algorithm that describes how s and t produce their outputs. A recursive definition seems like the most likely candidate because of how Euclid's algorithm works[1]. And the search for a recurrence usually starts with a careful analysis of numericals.

Let's start with another numerical example, just for variety. The very pretty calculation of $\gcd(216, 3162)$ is depicted in Figure 3.

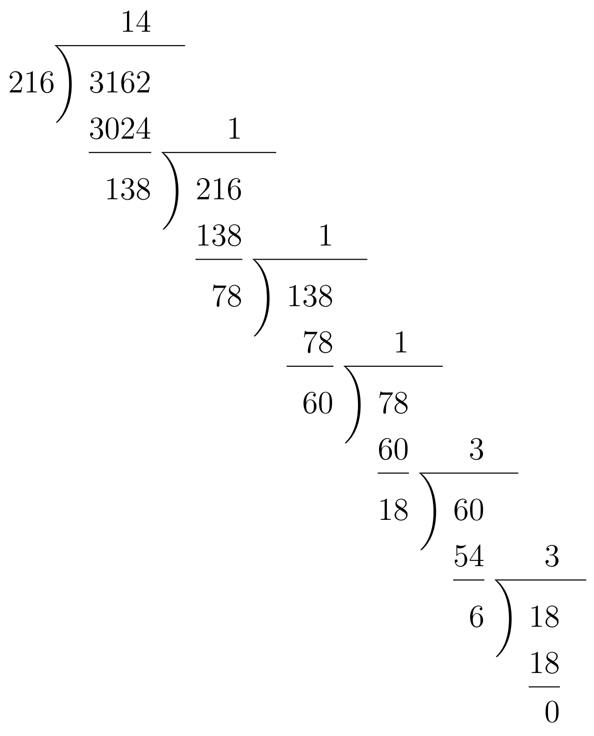


Figure 3: Another Example

A mental image like the one in Figure 2 on page 251 shows that each pair in

[1]It turns out that there's more than one definition that works. Stay tuned.

the sequence

$$(216, 3162) \to (138, 216) \to (78, 138) \to (60, 78)$$
$$\to (18, 60) \to (6, 18) \to (0, 6)$$

has the same gcd, a fact that was already implicit in Euclid and that is made explicit in the Snap*!* program in Figure 1. We can exploit this insight to build computational models for the functions s and t:

1. Express each of the remainders in terms of the division that produced it. For example, $3162 \div 216$ yields a quotient of 14 and remainder of 138, so
$$138 = 3162 - 14 \cdot 216$$
And in general:
$$\begin{aligned} 138 &= 3162 - 14 \cdot 216 \\ 78 &= 216 - 1 \cdot 138 \\ 60 &= 138 - 1 \cdot 78 \\ 18 &= 78 - 1 \cdot 60 \\ 6 &= 60 - 3 \cdot 18 \end{aligned}$$
2. Now, start with the last equation, and inductively back-substitute, simplifying at each step:
$$\begin{aligned} 6 &= -3 \cdot 18 + 60 \\ &= -3\overline{(78 - 1 \cdot 60)} + 60 = 4 \cdot 60 - 3 \cdot 78 \\ &= 4\overline{(138 - 1 \cdot 78)} - 3 \cdot 78 = -7 \cdot 78 + 4 \cdot 138 \\ &= -7(216 - 1 \cdot 138) + 4 \cdot 138 = 11 \cdot 138 - 7 \cdot 216 \\ &= 11(3162 - 14 \cdot 216) - 7 \cdot 216 = -161 \cdot 216 + 11 \cdot 3162 \end{aligned}$$
3. So, the calculation unstacks (adding two more steps for completeness) as
$$\begin{aligned} 6 &= 0 \cdot 0 + 1 \cdot 6 \\ &= 1 \cdot 6 + 0 \cdot 18 \\ &= -3 \cdot 18 + 1 \cdot 60 \\ &= 4 \cdot 60 - 3 \cdot 78 \\ &= -7 \cdot 78 + 4 \cdot 138 \\ &= 11 \cdot 138 - 7 \cdot 216 \\ &= -161 \cdot 216 + 11 \cdot 3162 \end{aligned}$$
These two steps don't appear explicitly in the calculation above but follow the pattern back to its logical root.

These five steps appear explicitly in the calculation above.

We find that $6 = -161 \cdot 216 + 11 \cdot 3162$. Notice the rhythm. And there's more: Notice that 6 is not only a combination of 216 and 3126, it is also a combination of 18 and 60, of 60 and 78, of 78 and 138, and of 138 and 216. Notice also that the pairs

$$(18, 60)\ (60, 78)\ (78, 138)\ (138, 216)\ (216, 3162)$$

are just the quotients and remainders in the calculation pictured in Figure 3.

Shoe-horning a gut instinct into a precise formulation is another useful habit in computational (and mathematical) thinking. *Saying*, precisely, what you mean helps you understand what you mean. This is certainly possible in mathematical notation alone. But—especially for those who are less constantly immersed in and fluent with mathematical notation—this is one of the main benefits of programming. So, let's look more carefully at this whole process. Suppose that, using long division, we find an integer q (the quotient when b is divided by a) so that

$$b = qa + \text{mod}(b, a)$$

To streamline the notation, let $r = \text{mod}(b, a)$, so that

$$b = qa + r \quad \text{and} \quad 0 \le r < a$$

This is just algebraic notation for what fourth graders learn about checking a division problem. If you've divided b by a, and gotten q as the quotient and r as the remainder, then, to check it, multiply a by q, add r, and you should get b; oh, and by the way, the remainder must be less than the number you divide by.

Then the calculation for finding $\gcd(a, b)$ starts out like this:

$$(a, b) \longrightarrow (r, a) \longrightarrow \cdots$$

Suppose that $d = \gcd(a, b)$, so that $d = \gcd(r, a)$ (Euclid, again). As in the calculation with 216 and 3162, take the coefficients for r and a—these are just the values for s and t at r and a—and "lift" them to coefficients for a and b. That is, if we suppose we've already found $s(r, a)$ and $t(r, a)$, we can use that result to derive $s(a, b)$ and $t(a, b)$. This "make believe" habit is another useful piece of computational thinking, the analogue of how lemmas are used in mathematics. Here's how that would work. We know that

$$d = s(r, a)r + t(r, a)a$$

and we have supposed that we know these values of s and t. So, since $r = b - aq$, we know that

$$\begin{aligned} d &= s(r,a)(b-aq) + t(r,a)a \\ &= (t(r,a) - q\,s(r,a))\,a + s(r,a)\,b \\ &= (\text{something}) \times a + (\text{something else}) \times b \end{aligned}$$

Bingo. We can take

$$s(a,b) = t(r,a) - q\,s(r,a)$$

and

$$t(a,b) = s(r,a)$$

This pair of functional equations is very unusual: s is defined in terms of t, and t is defined in terms of s. Yes, the arguments decrease, but still, this is not a structure that is commonly seen. . . .

The pair can be expressed like this:

$$s(a,b) = \begin{cases} \text{some base case} & \\ t(r,a) - q\,s(r,a) & \text{otherwise} \end{cases}$$

$$t(a,b) = \begin{cases} \text{some other base case} & \\ s(r,a) & \text{otherwise} \end{cases}$$

where q is the quotient when b is divided by a—that is the floor of $\frac{b}{a}$.

Ah. . . a can't be 0. So, maybe something like this will work:

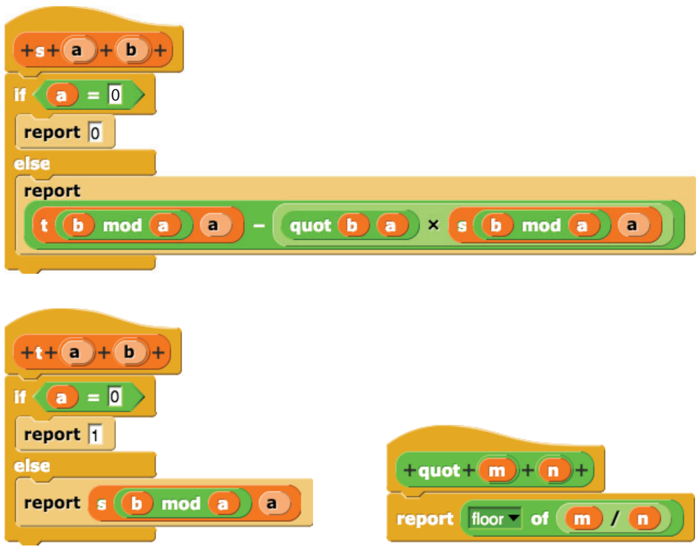
$$s(a,b) = \begin{cases} \text{some base case} & \text{if } a = 0 \\ t(r,a) - q\,s(r,a) & \text{otherwise} \end{cases}$$

$$t(a,b) = \begin{cases} \text{some other base case} & \text{if } a = 0 \\ s(r,a) & \text{otherwise} \end{cases}$$

Oh, but if $a = 0$, there's an easy choice for $s(0,b)$ and $t(0,b)$ because $\gcd(0,b) = b$ and $b = 0 \cdot 0 + 1 \cdot b$. Maybe, then, this:

$$s(a,b) = \begin{cases} 0 & \text{if } a = 0 \\ t(r,a) - q\,s(r,a) & \text{otherwise} \end{cases} \quad \text{and} \quad t(a,b) = \begin{cases} 1 & \text{if } a = 0 \\ s(r,a) & \text{otherwise} \end{cases}$$

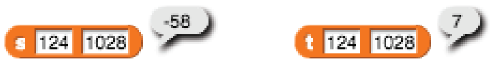
Snap*!* is a perfect tool for this kind of investigation; it provides a medium for an almost direct translation of the mathematical definitions, as in Figure 4.



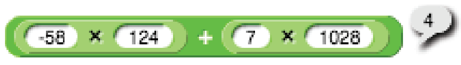
```
+s+ a + b +
if a = 0
  report 0
else
  report ( t (b mod a) a ) − ( quot b a ) × ( s (b mod a) a )

+t+ a + b +
if a = 0
  report 1
else
  report s (b mod a) a

+quot+ m + n +
report floor of ( m / n )
```

Figure 4: Computational models for s and t

This should make your head hurt—a double recursion. But if we take the models out for a spin. . .



```
s 124 1028    -58
t 124 1028    7
```



```
(-58 × 124) + (7 × 1028)    4
```

As one of our colleagues likes to say, *Holy Moly!* It works, just as it's supposed to work. It took some time and paper to convince ourselves that this was destined to work—it's all in the careful mathematics, careful coding, and powerful medium.

Sit back in delight and watch this work—over and over with many different examples, some designed to trick it. It gives back the ideas used to create it. We came up with this model through an analysis of the recursive form of Euclid's algorithm. And, the fact that we had been using Logo, Scheme, and Snap*!* in certain ways was an essential ingredient into the inspiration for and creation of these functions.

The programs give back

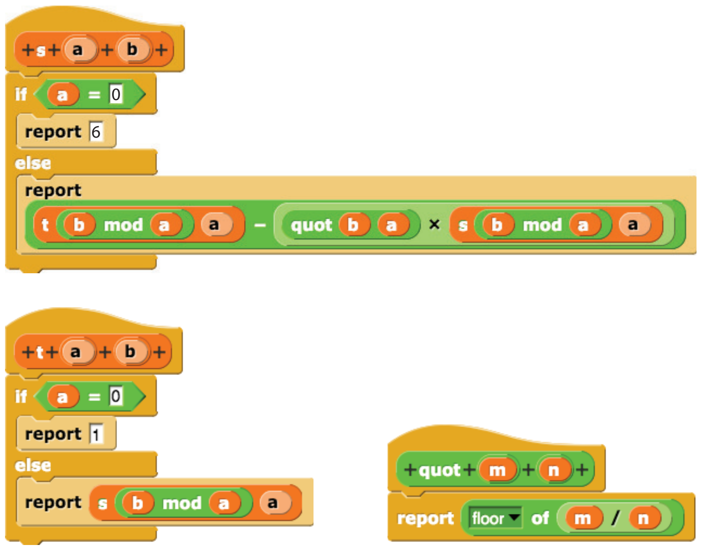
An old friend and longtime mathematics teacher, Angelo DiDomenico, used to paraphrase a quote by D'Alembert saying: "Mathematics is generous. It gives back to you more than you ask." The same is true about programs. Analyzing what we have just created gives new insights into results, results that may have been known before but that take on new impact because they emerge from computation.

There was some arbitrariness in our definition of the functions s and t. On page 256, we said, "Oh, but if $a = 0$, there's an easy choice for $s(0, b)$ and $t(0, b)$ because $\gcd(0, b) = b$ and $b = 0 \cdot 0 + 1 \cdot b$." And that led to two programs, pictured in Figure 4.

But, if you think about it, if $a = 0$, it doesn't matter what the coefficient of a is

$$\gcd(0, b) = \text{anything} \cdot 0 + 1 \cdot b,$$

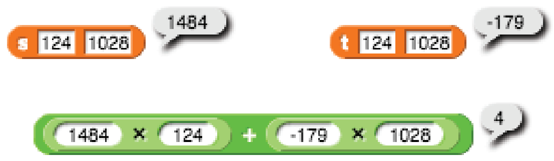
(another application of looking at the structure of an algorithm). So, we can change the base case for s to anything we want, and it should still work. Returning to $\gcd(124, 1028)$, suppose, for example, that you edit the code for s to make $s(0, b) = 6$ (instead of 0) and keep the rest the same:



```
+s+ a + b +
if  a = 0
    report 6
else
    report  t (b mod a) a  −  (quot b a) × s (b mod a) a

+t+ a + b +
if  a = 0
    report 1
else
    report  s (b mod a) a

+quot+ m + n +
report  floor of (m / n)
```

Figure 5: Change the base case to output 6

Let's see:



```
s 124 1028    1484
t 124 1028    -179
(1484 × 124) + (-179 × 1028)    4
```

Holy Moly. It works again. We tried several other choices for $s(0, a)$, and, of course, they all worked. We wondered if there's a closed form—an explicit formula—for $s(a, b)$ and $t(a, b)$ in terms of the base cases. That is, if $s(0, a)$ is, say, x, what would $s(a, b)$ and $t(a, b)$ in terms of x?

One computational habit is to start from examples—try different values for $s(0, a)$, organizing the data in a regular way. In that tradition, here's table for several values of $s(0, a)$, looking to see how it depends on a:

a	$s(0, a)$
0	-58
1	199
2	456
3	713
4	970
5	1227

The usual habit (after you've done this kind of thing for some years) is to look at first differences:

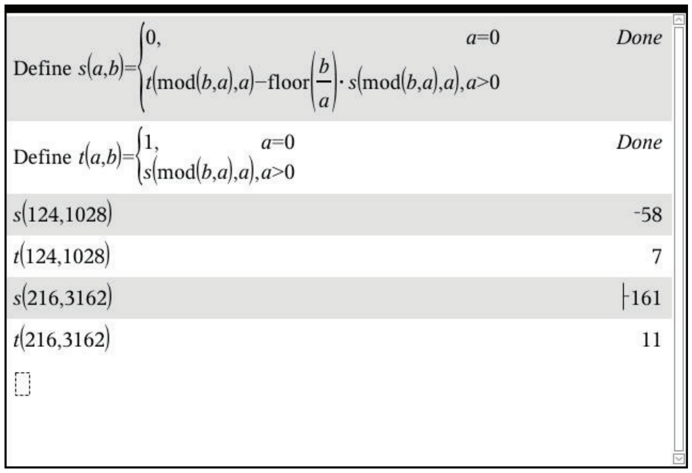
a	$s(0, a)$	Δ
0	-58	257
1	199	257
2	456	257
3	713	257
4	970	257
5	1227	

So, it seems (with overwhelming evidence) that a linear function fits the table:

$$s(0, a) = -58 + 257a$$

Something's going on.

Someday, Snap*!* will have a built-in CAS. Until then, we fired up a trusty CAS (the TI nspire), seeing what it has to say:

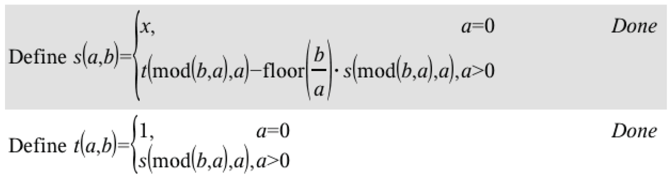


$$\text{Define } s(a,b)=\begin{cases}0, & a=0\\ t(\text{mod}(b,a),a)-\text{floor}\left(\frac{b}{a}\right)\cdot s(\text{mod}(b,a),a), & a>0\end{cases} \qquad \textit{Done}$$

$$\text{Define } t(a,b)=\begin{cases}1, & a=0\\ s(\text{mod}(b,a),a), & a>0\end{cases} \qquad \textit{Done}$$

```
s(124,1028)          -58
t(124,1028)            7
s(216,3162)         -161
t(216,3162)           11
```

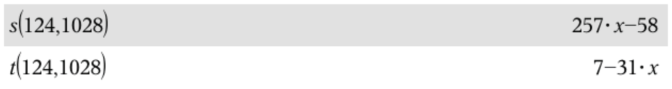
Yup—it works. Now for the fun part. Replace $s(0, a)$ with a "placeholder" x:



$$\text{Define } s(a,b)=\begin{cases}x, & a=0\\ t(\text{mod}(b,a),a)-\text{floor}\left(\frac{b}{a}\right)\cdot s(\text{mod}(b,a),a), & a>0\end{cases} \qquad \textit{Done}$$

$$\text{Define } t(a,b)=\begin{cases}1, & a=0\\ s(\text{mod}(b,a),a), & a>0\end{cases} \qquad \textit{Done}$$

And now. . .



```
s(124,1028)     257·x-58
t(124,1028)      7-31·x
```

Oh, this deserves another *Holy Moly!*. Look:

$$(257x - 58)124 + (7 - 31x)1028 = 4$$

And the "reason" behind this is that

$$257 \cdot 124 = 31 \cdot 1028$$

So, the coefficients are adjusted (by our trusty CAS) so that the adjustments cancel out.

This whole exercise is so very pleasing. Seeing mathematical phenomena come alive like this reinforces our longstanding feeling that mathematics is real and open to experiment.

What is to be learned from this?

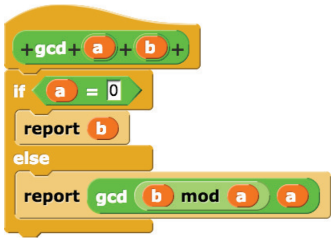
These results are in all the number theory books. Sometimes they are motivated (as in [4], we hope). But here, they emerge from what Hoyles and colleagues call "instrumental genesis"—this back and forth between your head and your tools.

Much of the literature about computational thinking focuses one direction: the implementation of ideas. This, by itself, is a reason enough to infuse computational media into mathematics education at all levels. In addition to modeling, simulations, and data mining, there are benefits to the "bread and butter" areas of algebra and geometry.

- Conventional mathematical notation on paper sits there, correct or incorrect, until it is reenacted in the writer's mind. And it *must* be reenacted, not only to catch possible errors, but also to continue and extend a correct line of thought. It takes work to read and understand—even regular prose takes work to read and understand—and depending on all sorts of other factors including time, distraction, and experience, that can become a filter that limits who does the reading and who does not. Writing

  $$\gcd(a,b) = \begin{cases} b & \text{if } a = 0 \\ \gcd(b \bmod a, a) & \text{if } a > 0 \end{cases}$$

  as

  ```
  +gcd+ a + b +
  if a = 0
      report b
  else
      report gcd (b mod a) a
  ```

  also takes effort, but is now possible, and the effort to do so is repaid. For one thing, the computer notation is runnable. We can detect errors in it by running it on cases we know easily, whereas detecting an error in the conventional notation requires either knowing what it should say, or carefully working through multi-step calculations.

- For another thing, the building blocks of the computer notation can be played with independently, starting with a very inelegant formulation and working toward the algorithm over time.

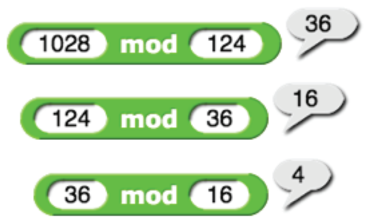


Figure 6: Building up from regularity in calculations

- We also get a surprise. The initial claim that if $a < b$, then $\gcd(a, b) = \gcd(b - a, a)$ specifies that a must be less than b (so that the repeated subtraction doesn't blow up).

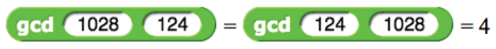


Figure 7: It doesn't care

By looking at that last line and figuring out what it does, we can explain why the program (and therefore, the algorithm) doesn't actually care which of a and b is greater.

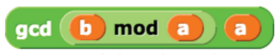


Figure 8: The last line

- And, finally, though the kind of experimentation that leads to this formulation can—and historically did—get "run" in one's head, many people who don't start out with that agility and focus can learn to do that, even in their heads, when the first steps require less patience and writing and rereading and reenactment. The act of building a runnable algorithm rather than describing one that one can only imagine running is a plausible help—and, in our experience, a practical and observable help—allowing students to internalize the way of thinking and, later, run those imagined algorithms internally.

Runnable notation, modularity, analysis of outputs, and experimentation are all crucial parts of the story.

But there's another equally important part, one that doesn't get enough play. It's the argument that we make in this paper: the claim on page 247 that the "arrows go both ways"—*creating programs influences and is influenced by how you think*.

Acknowledgments. This work is partially supported by NSF Grant #1441075, The Beauty and Joy of Computing.

References

[1] Michèle Artigue, "Learning Mathematics in a CAS Environment: The Genesis of a Reflection about Instrumentation and the Dialectics between Technical and Conceptual Work", *International Journal of Computers for Mathematical Learning*, Volume **7** (2002), pages 245-274.

[2] Al Cuoco, *Investigations in Algebra*, MIT Press, Cambridge MA, 1990.

[3] Al Cuoco, "Thoughts on Reading Artigue's "Learning Mathematics in a CAS Environment", *International Journal of Computers for Mathematical Learning*, Volume **7** (2002), pages 293-299.

[4] Al Cuoco and Joseph J. Rotman, *Learning Modern Algebra*, MAA Press, Washington DC, 2013.

[5] Celia Hoyles and Richard Noss and Phillip Kent, "On the Integration of Digital Technologies into Mathematics Classrooms", *International Journal of Computers in Mathematics Learning*. Volume **9** (2004), pages 309-326.

[6] Luc Trouche, "Managing the Complexity of Human/Machine Interaction in a Computer Based Learning Environment: Guiding Student's Process Command through Instrumental Orchestrations", Plenary presentation at the Third Computer Algebra in Mathematics Education Symposium, Reims, France, June 2003.

[7] Pierre Vérillon and Pierre Rabardel, "Cognition and Artefacts: a contribution to the study of thought in relation to instrumented activity", *European Journal of Psychology of Education*, Volume **10**, pages 77-101.

[8] Wikipedia contributors, "Into the Mystic", *Wikipedia, The Free Encyclopedia*; available at `https://en.wikipedia.org/wiki/Into_the_Mystic`, last accessed on January 28, 2021.