Chapter 10

Patient Coding, Faster Code

Chapter 10

How Working Quickly Can Bring Execution to a Crawl

My grandfather does *The New York Times* crossword puzzle every Sunday. In ink. With nary a blemish.

The relevance of which will become apparent in a trice.

What my grandfather is, is a pattern matcher *par excellence*. You're a pattern matcher, too. So am I. We can't help it; it comes with the territory. Try focusing on text and not reading it. Can't do it. Can you hear the voice of someone you know and not recognize it? I can't. And how in the Nine Billion Names of God is it that we're capable of instantly recognizing one face out of the thousands we've seen in our lifetimes—even years later, from a different angle and in different light? Although we take them for granted, our pattern-matching capabilities are surely a miracle on the order of loaves and fishes.

By "pattern matching," I mean more than just recognition, though. I mean that we are generally able to take complex and often seemingly woefully inadequate data, instantaneously match it in an incredibly flexible way to our past experience, extrapolate, and reach amazing conclusions, something that computers can scarcely do at all. Crossword puzzles are an excellent example; given a couple of letters and a cryptic clue, we're somehow able to come up with one out of several hundred thousand words that we know. Try writing a program to do that! What's more, we don't process data in the serial brute-force way that computers do. Solutions tend to be virtually instantaneous or not at all; none of those "N log N" or "N^2" execution times for us.

It goes without saying that pattern matching is good; more than that, it's a large part of what we are, and, generally, the faster we are at it, the better. Not always, though. Sometimes insufficient information really is insufficient, and, in our haste to get the heady rush of coming up with a solution, incorrect or less-than-optimal conclusions are reached, as anyone who has ever done the *Times* Sunday crossword will attest. Still, my grandfather does that puzzle every Sunday *in ink.* What's his secret? Patience and discipline. He never fills a word in until he's confirmed it in his head via intersecting words, no matter how strong the urge may be to put something down where he can see it and feel like he's getting somewhere.

There's a surprisingly close parallel to programming here. Programming is certainly a sort of pattern matching in the sense I've described above, and, as with crossword puzzles, following your programming instincts too quickly can be a liability. For many programmers, myself included, there's a strong urge to find a workable approach to a particular problem and start coding it *right now,* what some people call "hacking" a program. Going with the first thing your programming pattern matcher comes up with can be a lot of fun; there's instant gratification and a feeling of unbounded creativity. Personally, I've always hungered to get results from my work as soon as possible; I gravitated toward graphics for its instant and very visible gratification. Over time, however, I've learned patience.

I've come to spend an increasingly large portion of my time choosing algorithms, designing, and simply giving my mind quiet time in which to work on problems and come up with non-obvious approaches before coding; and I've found that the extra time up front more than pays for itself in both decreased coding time and superior programs.

In this chapter, I'm going to walk you through a simple but illustrative case history that nicely points up the wisdom of delaying gratification when faced with programming problems, so that your mind has time to chew on the problems from other angles. The alternative solutions you find by doing this may seem obvious, once you've come up with them. They may not even differ greatly from your initial solutions. Often, however, they will be much better—and you'll never even have the chance to decide whether they're better or not if you take the first thing that comes into your head and run with it.

The Case for Delayed Gratification

Once upon a time, I set out to read *Algorithms,* by Robert Sedgewick (Addison-Wesley), which turned out to be a wonderful, stimulating, and most useful book, one that I recommend highly. My story, however, involves only what happened in the first 12 pages, for it was in those pages that Sedgewick discussed Euclid's algorithm.

Euclid's algorithm (discovered by Euclid, of Euclidean geometry fame, a very long time ago, way back when computers still used core memory) is a straightforward algorithm that solves one of the simplest problems imaginable: finding the greatest common integer divisor (GCD) of two positive integers. Sedgewick points out that this is useful for reducing a fraction to its lowest terms. I'm sure it's useful for other things, as well, although none spring to mind. (A long time ago, I wrote an article about optimizing a bit of code that wasn't even vaguely time-critical, and got swamped with letters telling me so. I knew it wasn't time-critical; it was just a good example. So for now, close your eyes and *imagine* that finding the GCD is not only necessary but must also be done as quickly as possible, because it's perfect for the point I want to make here and now. Okay?)

The problem at hand, then, is simply this: Find the largest integer value that evenly divides two arbitrary positive integers. That's all there is to it. So warm up your pattern matchers...and go!

The Brute-Force Syndrome

I have a funny feeling that you'd already figured out how to find the GCD before I even said "go." That's what I did when reading *Algorithms;* before I read another word, I had to figure it out for myself. Programmers are like that; give them a problem and their eyes immediately glaze over as they try to solve it before you've even shut your mouth. That sort of instant response can certainly be impressive, but it can backfire, too, as it did in my case.

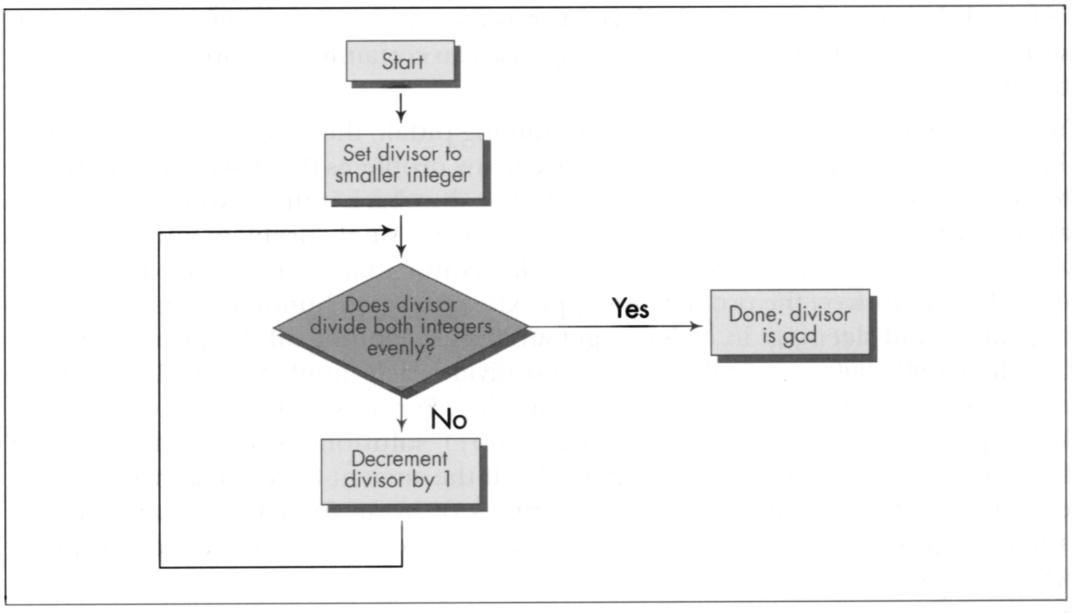
You see, I fell victim to a common programming pitfall, the "brute-force" syndrome. The basis of this syndrome is that there are many problems that have obvious, brute-force solutions—with one small drawback. The drawback is that if you were to try to apply a brute-force solution by hand—that is, work a single problem out with pencil and paper or a calculator—it would generally require that you have the patience and discipline to work on the problem for approximately seven hundred years, not counting eating and sleeping, in order to get an answer. Finding all the prime numbers less than 1,000,000 is a good example; just divide each number up to 1,000,000 by every lesser number, and see what's left standing. For most of the history of humankind, people were forced to think of cleverer solutions, such as the Sieve of Eratosthenes (we'd have been in big trouble if the ancient Greeks had had computers), mainly because after about five minutes of brute force-type work, people's attention gets diverted to other important matters, such as how far a paper airplane will fly from a second-story window.

Not so nowadays, though. Computers love boring work; they're very patient and disciplined, and, besides, one human year = seven dog years = two zillion computer years. So when we're faced with a problem that has an obvious but exceedingly lengthy

solution, we're apt to say, "Ah, let the computer do that, it's fast," and go back to making paper airplanes. Unfortunately, brute-force solutions tend to be slow even when performed by modern-day microcomputers, which are capable of several MIPS except when I'm late for an appointment and want to finish a compile and run just one more test before I leave, in which case the crystal in my computer is apparently designed to automatically revert to 1 Hz.)

The solution that I instantly came up with to finding the GCD is about as brute- force as you can get: Divide both the larger integer (iL) and the smaller integer (iS) by every integer equal to or less than the smaller integer, until a number is found that divides both evenly, as shown in Figure 10.1. This works, but it's a lousy solution, requiring as many as iS*2 divisions; *very* expensive, especially for large values of iS. For example, finding the GCD of 30,001 and 30,002 would require 60,002 divisions, which alone, disregarding tests and branches, would take about 2 seconds on an 8088, and more than 50 milliseconds even on a 25 MHz 486—a *very* long time in computer years, and not insignificant in human years either.

Listing 10.1 is an implementation of the brute-force approach to GCD calculation. Table 10.1 shows how long it takes this approach to find the GCD for several integer pairs. As expected, performance is extremely poor when iS is large.



Using a brute-force algorithm to find a GCD.
Figure 10.1

	Integer pairs for which to find GCD				
	90 & 27	**42 & 998**	**453 & 121**	**27432 & 165**	**27432 & 17550**
Listing 10.1 (Brute force)	60µs (100%)	110µs (100%)	311ms (100%)	426µs (100%)	43580µs (100%)
Listing 10.2 (Subtraction)	25 (42%)	72 (65%)	67 (22%)	280 (66%)	72 (0.16%)
Listing 10.3 (Division: code recursive Euclid's algorithm)	20 (33%)	33 (30%)	48 (15%)	32 (8%)	53 (0.12%)
Listing 10.4 (C version of data recursive Euclid's algorithm; normal optimization)	12 (20%)	17 (15%)	25 (8%)	16 (4%)	26 (0.06%)
Listing 10.4 (/Ox = maximumoptimization)	12 (20%)	16 (15)	20 (6%)	15 (4%)	23 (0.05%)
Listing 10.5 (Assembly version of data recursive Euclid's algorithm)	10 (17%)	10 (9%)	15 (5%)	10 (2%)	17 (0.04%)

Note: Performance of Listings 10.1 through 10.5 in finding the greatest common divisors of various pairs of integers. Times are in microseconds. Percentages represent execution time as a percentage of the execution time of Listing 10.1 for the same integer pair. Listings 10.1-10.4 were compiled with Microsoft C /C++ except as noted, the default optimization was used. All times measured with the Zen timer (from Chapter 3) on a 20 MHz cached 386.

Table 10.1 Performance of GCD algorithm implementations.

LISTING 10.1 L10-1.C

```
/* Finds and returns the greatest common divisor of two positive
   integers. Works by trying every integral divisor between the
   smaller of the two integers and 1, until a divisor that divides
   both integers evenly is found. All C code tested with Microsoft
   and Borland compilers.*/

unsigned int gcd(unsigned int int1, unsigned int int2) {
   unsigned int temp, trial_divisor;
   /* Swap if necessary to make sure that int1 >= int2 */
   if (int1 < int2) {
      temp = int1;
      int1 = int2;
      int2 = temp;
   }
```

```
   /* Now just try every divisor from int2 on down, until a common
      divisor is found. This can never be an infinite loop because
      1 divides everything evenly */
   for (trial_divisor = int2; ((int1 % trial_divisor) != 0) ||
         ((int2 % trial_divisor) != 0); trial_divisor--)
      ;
   return(trial_divisor);
}
```

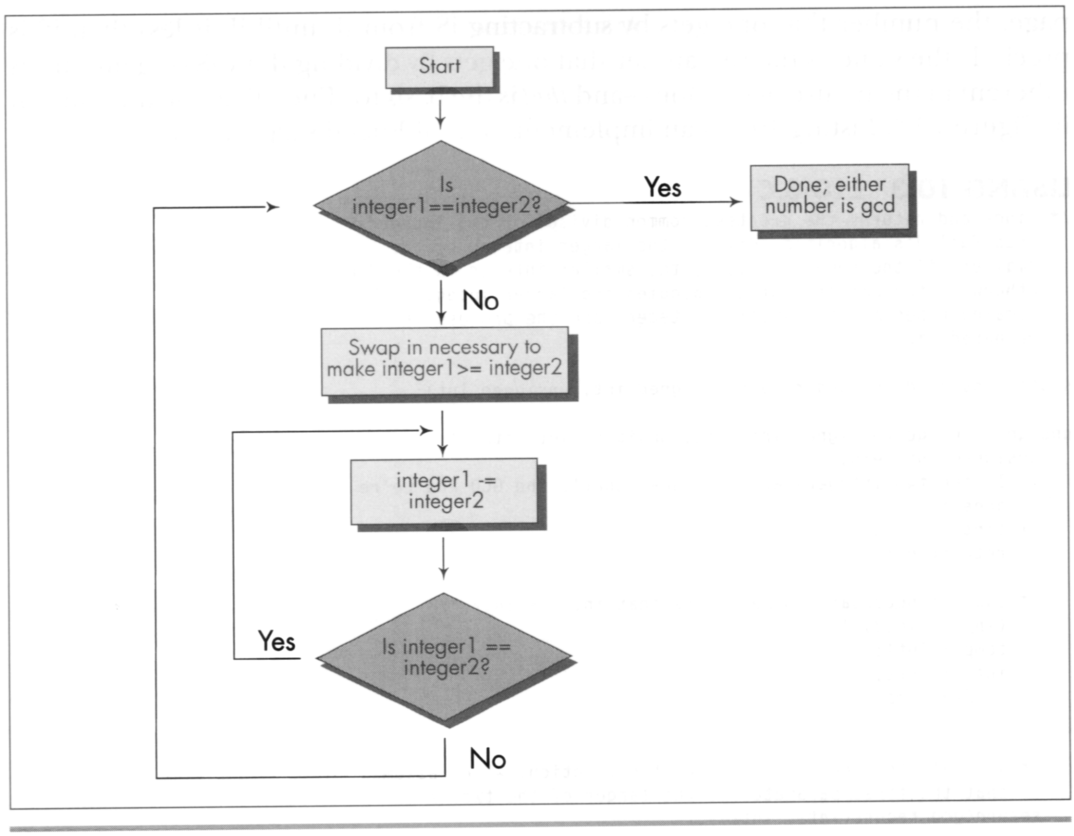
Wasted Breakthroughs

Sedgewick's first solution to the GCD problem was pretty much the one I came up with. He then pointed out that the GCD of iL and iS is the same as the GCD of iL-iS and iS. This was obvious (once Sedgewick pointed it out); by the very nature of division, any number that divides iL evenly nL times and iS evenly nS times must divide iL-iS evenly nL-nS times. Given that insight, I immediately designed a new, faster approach, shown in Listing 10.2.

LISTING 10.2 L10-2.C

```
/* Finds and returns the greatest common divisor of two positive
   integers. Works by subtracting the smaller integer from the
   larger integer until either the values match (in which case
   that's the gcd), or the larger integer becomes the smaller of
   the two, in which case the two integers swap roles and the
   subtraction process continues. */

unsigned int gcd(unsigned int int1, unsigned int int2) {
   unsigned int temp;
   /* If the two integers are the same, that's the gcd and we're
      done */
   if (int1 == int2) {
      return(int1);
   }
   /* Swap if necessary to make sure that int1 >= int2 */
   if (int1 < int2) {
      temp = int1;
      int1 = int2;
      int2 = temp;
   }

   /* Subtract int2 from int1 until int1 is no longer the larger of
      the two */
   do {
      int1 -= int2;
   } while (int1 > int2);
   /* Now recursively call this function to continue the process */
   return(gcd(int1, int2));
}
```

Listing 10.2 repeatedly subtracts iS from iL until iL becomes less than or equal to iS. If iL becomes equal to iS, then that's the GCD; alternatively, if iL becomes *less* than iS, iL and iS switch values, and the process is repeated, as shown in Figure 10.2. The number of iterations this approach requires relative to Listing 10.1 depends heavily on the values of iL and iS, so it's not always faster, but, as Table 10.1 indicates, Listing 10.2 is generally much better code.



Using repeated subtraction algorithm to find a GCD.
Figure 10.2

Listing 10.2 is a far graver misstep than Listing 10.1, for all that it's faster. Listing 10.1 is obviously a hacked-up, brute-force approach; no one could mistake it for anything else. It could be speeded up in any of a number of ways with a little thought. (Simply skipping testing all the divisors between iS and iS/2, not inclusive, would cut the worst-case time in half, for example; that's not a particularly *good* optimization, but it illustrates how easily Listing 10.1 can be improved.) Listing 10.1 is a hack job, crying out for inspiration.

Listing 10.2, on the other hand, has gotten the inspiration—and largely wasted it through haste. Had Sedgewick not told me otherwise, I might well have assumed that Listing 10.2 was optimized, a mistake I would never have made with Listing 10.1. I experienced a conceptual breakthrough when I understood Sedgewick's point: A smaller number can be subtracted from a larger number without affecting their GCD, thereby inexpensively reducing the scale of the problem. And, in my hurry to make this breakthrough reality, I missed its full scope. As Sedgewick says on the very next

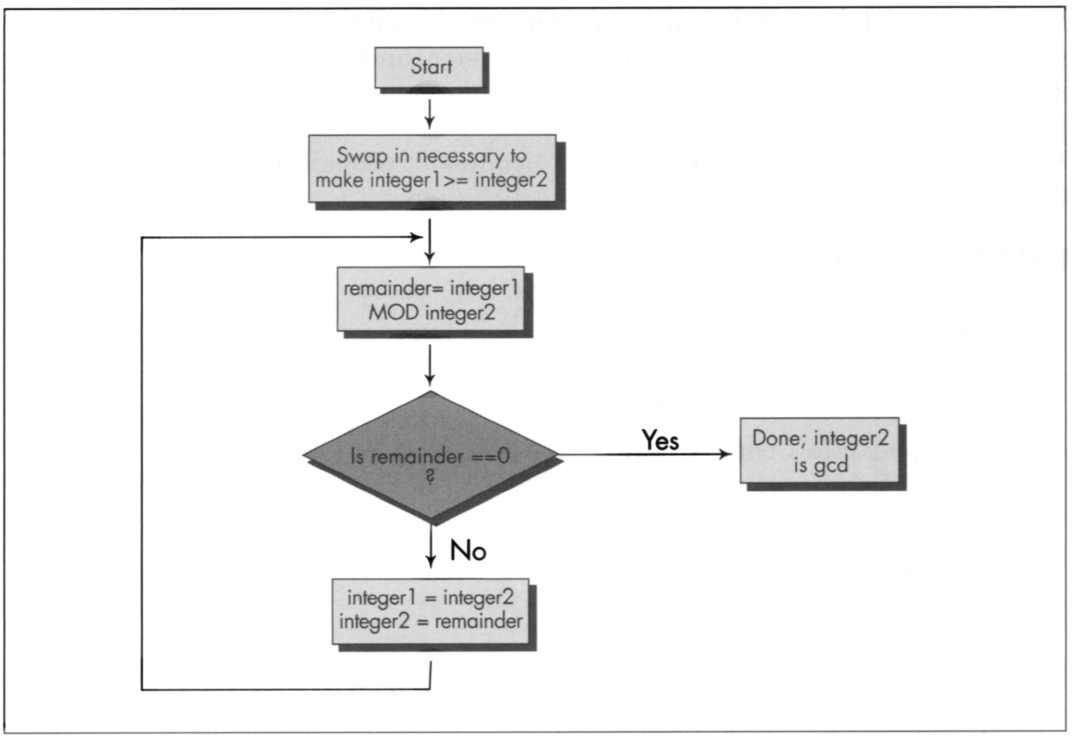
page, the number that one gets by subtracting iS from iL until iL is less than iS is precisely the same as the remainder that one gets by dividing iL by iS—again, this is inherent in the nature of division—and *that* is the basis for Euclid's algorithm, shown in Figure 10.3. Listing 10.3 is an implementation of Euclid's algorithm.

LISTING 10.3 L10-3.C

```
/* Finds and returns the greatest common divisor of two integers.
   Uses Euclid's algorithm: divides the larger integer by the
   smaller; if the remainder is 0, the smaller integer is the GCD,
   otherwise the smaller integer becomes the larger integer, the
   remainder becomes the smaller integer, and the process is
   repeated. */

static unsigned int gcd_recurs(unsigned int, unsigned int);

unsigned int gcd(unsigned int int1, unsigned int int2) {
   unsigned int temp;
   /* If the two integers are the same, that's the GCD and we're
      done */
   if (int1 == int2) {
      return(int1);
   }
   /* Swap if necessary to make sure that int1 >= int2 */
   if (int1 < int2) {
      temp = int1;
      int1 = int2;
      int2 = temp;
   }

   /* Now call the recursive form of the function, which assumes
      that the first parameter is the larger of the two */
   return(gcd_recurs(int1, int2));
}

static unsigned int gcd_recurs(unsigned int larger_int,
      unsigned int smaller_int)
{
   int temp;

   /* If the remainder of larger_int divided by smaller_int is 0,
      then smaller_int is the gcd */
   if ((temp = larger_int % smaller_int) == 0) {
      return(smaller_int);
   }
   /* Make smaller_int the larger integer and the remainder the
      smaller integer, and call this function recursively to
      continue the process */
   return(gcd_recurs(smaller_int, temp));
}
```

As you can see from Table 10.1, Euclid's algorithm is superior, especially for large numbers (and imagine if we were working with large *longs!*).

Had I been implementing GCD determination without Sedgewick's help, I would surely not have settled for Listing 10.1—but I might well have ended up with Listing 10.2 in my enthusiasm over the "brilliant" discovery of subtracting the lesser



Using Euclid's algorithm to find a GCD.
Figure 10.3

number from the greater. In a commercial product, my lack of patience and discipline could have been costly indeed.

Give your mind time and space to wander around the edges of important programming problems before you settle on any one approach. I titled this book's first chapter "The Best Optimizer Is between Your Ears," and that's still true; what's even more true is that the optimizer between your ears does its best work not at the implementation stage, but at the very beginning, when you try to imagine how what you want to do and what a computer is capable of doing can best be brought together.

Recursion

Euclid's algorithm lends itself to recursion beautifully, so much so that an implementation like Listing 10.3 comes almost without thought. Again, though, take a moment to stop and consider what's really going on, at the assembly language level, in Listing 10.3. There's recursion and then there's recursion; code recursion and data recursion, to be exact. Listing 10.3 is code recursion—recursion through calls—

the sort most often used because it is conceptually simplest. However, code recursion tends to be slow because it pushes parameters and calls a subroutine for every iteration. Listing 10.4, which uses data recursion, is much faster and no more complicated than Listing 10.3. Actually, you could just say that Listing 10.4 uses a loop and ignore any mention of recursion; conceptually, though, Listing 10.4 performs the same recursive operations that Listing 10.3 does.

LISTING 10.4 L10-4.C

```
/* Finds and returns the greatest common divisor of two integers.
   Uses Euclid's algorithm: divides the larger integer by the
   smaller; if the remainder is 0, the smaller integer is the GCD,
   otherwise the smaller integer becomes the larger integer, the
   remainder becomes the smaller integer, and the process is
   repeated. Avoids code recursion. */

unsigned int gcd(unsigned int int1, unsigned int int2) {
   unsigned int temp;

   /* Swap if necessary to make sure that int1 >= int2 */
   if (int1 < int2) {
      temp = int1;
      int1 = int2;
      int2 = temp;
   }
   /* Now loop, dividing int1 by int2 and checking the remainder,
      until the remainder is 0. At each step, if the remainder isn't
      0, assign int2 to int1, and the remainder to int2, then
      repeat */
   for (;;) {
      /* If the remainder of int1 divided by int2 is 0, then int2 is
         the gcd */
      if ((temp = int1 % int2) == 0) {
         return(int2);
      }
      /* Make int2 the larger integer and the remainder the
         smaller integer, and repeat the process */
      int1 = int2;
      int2 = temp;
   }
}
```

Patient Optimization

At long last, we're ready to optimize GCD determination in the classic sense. Table 10.1 shows the performance of Listing 10.4 with and without Microsoft C/C++'s maximum optimization, and also shows the performance of Listing 10.5, an assembly language version of Listing 10.4. Sure, the optimized versions are faster than the unoptimized version of Listing 10.4—but the gains are small compared to those realized from the higher-level optimizations in Listings 10.2 through 10.4.

LISTING 10.5 L10-5.ASM

```
; Finds and returns the greatest common divisor of two integers.
; Uses Euclid's algorithm: divides the larger integer by the
; smaller; if the remainder is 0, the smaller integer is the GCD,
```

```
; otherwise the smaller integer becomes the larger integer, the
; remainder becomes the smaller integer, and the process is
; repeated. Avoids code recursion.
;
;
;
; C near-callable as:
; unsigned int gcd(unsigned int int1, unsigned int int2);

; Parameter structure:
parms struc
      dw    ?           ;pushed BP
      dw    ?           ;pushed return address
int1  dw    ?           ;integers for which to find
int2  dw    ?           ; the GCD
parms ends

      .model      small
      .code
      public      _gcd
      align 2
_gcd  proc  near
      push  bp          ;preserve caller's stack frame
      mov   bp,sp       ;set up our stack frame
      push  si          ;preserve caller's register variables
      push  di

;Swap if necessary to make sure that int1 >= int2
      mov   ax,int1[bp]
      mov   bx,int2[bp]
      cmp   ax,bx       ;is int1 >= int2?
      jnb   IntsSet     ;yes, so we're all set
      xchg  ax,bx       ;no, so swap int1 and int2
IntsSet:

; Now loop, dividing int1 by int2 and checking the remainder, until
; the remainder is 0. At each step, if the remainder isn't 0, assign
; int2 to int1, and the remainder to int2, then repeat.
GCDLoop:
                        ;if the remainder of int1 divided by
                        ; int2 is 0, then int2 is the gcd
      sub   dx,dx       ;prepare int1 in DX:AX for division
      div   bx          ;int1/int2; remainder is in DX
      and   dx,dx       ;is the remainder zero?
      jz    Done        ;yes, so int2 (BX) is the gcd
                        ;no, so move int2 to int1 and the
                        ; remainder to int2, and repeat the
                        ; process
      mov   ax,bx       ;int1 = int2;
      mov   bx,dx       ;int2 = remainder from DIV

;-start of loop unrolling; the above is repeated three times-
      sub   dx,dx       ;prepare int1 in DX:AX for division
      div   bx          ;int1/int2; remainder is in DX
      and   dx,dx       ;is the remainder zero?
      jz    Done        ;yes, so int2 (BX) is the gcd
      mov   ax,bx       ;int1 = int2;
      mov   bx,dx       ;int2 = remainder from DIV
;-
      sub   dx,dx       ;prepare int1 in DX:AX for division
      div   bx          ;int1/int2; remainder is in DX
```

```
        and     dx,dx       ;is the remainder zero?
        jz      Done        ;yes, so int2 (BX) is the gcd
        mov     ax,bx       ;int1 = int2;
        mov     bx,dx       ;int2 = remainder from DIV
;-
        sub     dx,dx       ;prepare int1 in DX:AX for division
        div     bx          ;int1/int2; remainder is in DX
        and     dx,dx       ;is the remainder zero?
        jz      Done        ;yes, so int2 (BX) is the gcd
        mov     ax,bx       ;int1 = int2;
        mov     bx,dx       ;int2 = remainder from DIV
;-end of loop unrolling-
        jmp     GCDLoop

        align 2
Done:
        mov     ax,bx       ;return the GCD
        pop     di          ;restore caller's register variables
        pop     si
        pop     bp          ;restore caller's stack frame
        ret
_gcd    endp
        end
```

Assembly language optimization is pattern matching on a local scale. Frankly, it's also the sort of boring, brute-force work that people are lousy at; compilers could out-optimize you at this level with one pass tied behind their back *if* they knew as much about the code you're writing as you do, which they don't.

Design optimization—conceptual breakthroughs in understanding the relationships between the needs of an application, the nature of the data the application works with, and what the computer can do—is global pattern matching.

Computers are *much* worse at that sort of pattern matching than humans; computers have no way to integrate vast amounts of disparate information, much of it only vaguely defined or subject to change. People, oddly enough, are *better* at global optimization than at local optimization. For one thing, it's more interesting. For another, it's complex and imprecise enough to allow intuition and inspiration, two vastly underrated programming tools, to come to the fore. And, as I pointed out earlier, people tend to perform instantaneous solutions to even the most complex problems, while computers bog down in geometrically or exponentially increasing execution times. Oh, it may take days or weeks for a person to absorb enough information to be able to reach a solution, and the solution may only be near-optimal—but the solution itself (or, at least, each of the pieces of the solution) arrives in a flash.

Those flashes are your programming pattern matcher doing its job. *Your* job is to give your pattern matcher the opportunity to get to know each problem and run through it two or three times, from different angles, to see what unexpected solutions it can come up with.

Pull back the reins a little. Don't measure progress by lines of code written today; measure it instead by overall progress and by quality. Relax and listen to that quiet inner voice that provides the real breakthroughs. Stop, look, listen—and think. Not only will you find that it's a more productive and creative way to program—but you'll also find that it's more fun.

And think what you could do with all those extra computer years!