Call Me

Recursion, see Recursion

by Julian Bucknall

It occurred to me, as I read the proof of June's article on Ternary Trees, that I was tossing around willy-nilly sentences like 'The algorithm described is recursive, although the recursion can be fairly easily removed by use of an explicit stack.' And I would return to the same theme over and over. I suddenly thought, maybe some of my readers don't know how to do this, or don't know about when recursion is good, when it is bad, when to use it, when not to. Enough, this instalment of *Algorithms Alfresco* will concentrate on recursion.

In The Flesh

So what is recursion? Simply put, it is a routine that calls itself. Now, it must be said, this definition leaves a lot to be desired; after all, it smacks of the infinite loop. Indeed, recursion, if we are not very careful, will certainly turn into a series of never-ending calls that will eventually crash the application (be warned: I forgot to add a stop condition for one of the recursive routines I was writing for this article and, boom!, a nasty Windows stack overflow and a frozen machine). So, a recursive routine must also have a terminating condition to ensure that it doesn't continue calling itself, *ad infinitum*.

For those of you who did some mathematics (I can't be the only one!), recursion is related to the induction in 'proof by induction'. Proof by induction is when you prove a particular theorem by (1) proving it for a simple case (generally the theorem in question relates to numbers, so the simple case would be for the number 0 or 1) and (2) proving it for a general case (ie you assume that it's true for all numbers up to N-1, and then proving it for the number N, where N > 1). The induction bit goes like this: if you proved it for 1, then you can prove it for 2, by applying the second part of the proof, and similarly for 3, 4, and as far as you want to go. The theorem is said to be proved true by induction. An example is the proof that 1+2+3+...+N equals ½N(N+1). Hint: it's true for 1 (duh!) and if you assume that it's true for N-1, it's fairly easy algebra to show that it's true for N.

So what? The important thing to realize is that there's a built-in stop: the number 1 (or whatever limit you use) is a special case. It must be the same for recursive routines as well: there must be a built-in stop condition.

Picture This

Let's take a simple algorithm and write a recursive routine for it. Instead of using the hoary chestnut of factorial numbers, we'll use Fibonacci numbers to illustrate recursion. A Fibonacci number is defined as follows:

```
F(0) = 1
F(1) = 1
F(N) = F(N-1) + F(N-2)
```

So the Fibonacci numbers form a series: 1, 1, 2, 3, 5, 8, 13, 21, 34, ..., where each number is the sum of the previous two. Fibonacci numbers appear all over the place: in nature the seeds in a sunflower head form a Fibonacci sequence. In mathematics we find the sequence F(N-1)/F(N) approaches 0.618 or $\frac{1}{2}\left(\sqrt{5}-1\right)$ as N tends to infinity, the so-called 'golden ratio' much favored by the ancient Greeks. Also, the sequence F(N)/F(N-1) approaches 1.618, or the golden ratio plus 1. In computer data structures there is a structure called a Fibonacci heap which has certain desirable properties.

Taking the definition, it's easy to cast it into the routine in Listing 1. We can see intuitively that the routine will terminate and is not an infinite loop, since each recursive call to Fibonacci uses a smaller parameter than it started with, and we're making an explicit check for 0 and 1.

So how does it work? It's not immediately obvious why it should. Take N=3 for example. The sequence of calls that gets done can be viewed as a tree (or visually, an outline) of calls:

```
Fibonacci(3)
  Fibonacci(2)
    Fibonacci(1)
    Fibonacci(0)
  Fibonacci(1)
```

In other words, to calculate `Fibonacci(3)` we have to calculate `Fibonacci(2)`, which requires calculating `Fibonacci(1)` and then `Fibonacci(0)`. And then back at the '3' level, we need to calculate `Fibonacci(1)` to finish it off. As we go down the tree and return we have to save and restore some partial results for the recursion to work properly. Where do these results get saved? After all, the routine doesn't save anything explicitly. The answer is: on the program

➤ *Listing 1: A recursive Fibonacci number routine.*

```
function Fibonacci(N : cardinal) : cardinal;
begin
  if N <= 1 then
    Result := 1
  else
    Result := Fibonacci(N-1) + Fibonacci(N-2);
end;
```

stack. When we call the Fibonacci routine recursively, the compiler inserts some machine code to store the partial results on the stack before calling the routine, and so they're there and available when we return (actually, in 32-bits it turns out that it's the called routine which pushes registers on the stack, but either way the effect is the same). Bear this automatic process in mind when we get to the subject of removing recursion.

Some of you may already have wondered why I've chosen such a routine to illustrate recursion (two recursive calls per call to the routine?). The reason is that calculating Fibonacci numbers by this process is very slow and it would be better not to use a recursive routine at all. Why is it slow? To answer this I wrote a small program that counted how many times the Fibonacci function is called for various values. It is instructive to peruse the results in Figure 1.

We see that to calculate the 6th number we require 15 calls to the Fibonacci routine, for the 10th number we need 109 calls. In fact, to calculate the 36th Fibonacci number (which is 14,930,352) requires nearly 30 million calls and takes over 4 seconds on my Pentium 120. Pretty bad. The reason? Well, look at the call tree above and notice that we have to calculate `Fibonacci(1)` twice, since we've thrown away the result of the first call. The larger the number we start with, the more intermediary results we discard and have to recalculate.

As an aside: whenever you start playing around with Fibonacci numbers it is amazing how often they start to crop up in whatever you are doing. The number of calls required to calculate `Fibonacci(N)` is... `(2 * Fibonacci(N)) - 1`. The proof is by induction.

The recursive Fibonacci routine is an excellent example of when we really need to remove the recursion, or, in other words, to rewrite it as some kind of loop so that recursion doesn't occur. In the case of calculating the Fibonacci numbers we can start from

N	0	1	2	3	4	5	6	7	8	9
Fibonacci(N)	1	1	2	3	5	8	13	21	34	55
Calls required	1	1	3	5	9	15	25	41	67	109

➤ *Figure 1: Number of calls required to the recursive routine to calculate Fibonacci numbers.*

```
function FastFib(N : cardinal) : cardinal;
var
  i : integer;
  FibN : cardinal;
  FibNminus1 : cardinal;
  FibNminus2 : cardinal;
begin
  if (N <= 1) then
    Result := 1
  else begin
    FibNminus1 := 1;
    FibNminus2 := 1;
    for i := 2 to N do begin
      FibN := FibNminus1 + FibNminus2;
      FibNminus2 := FibNminus1;
      FibNminus1 := FibN;
    end;
    Result := FibN;
  end;
end;
```

➤ *Listing 2: Calculating Fibonacci numbers with a loop.*

first principles to remove the recursion. The routine in Listing 2 will do fine.

Notice that the loop just maintains two variables `FibNminus1` and `FibNminus2` to hold the previous two Fibonacci numbers. We never have to recalculate them like in the recursive version. As it happens, this routine calculates the 36th Fibonacci number too fast for the clock to tick on my machine (if you do compile this routine, notice that it fools Delphi 3's warning analyzer: it complains that the variable `FibN` might not have been initialized, and yet we've ensured that the loop will be executed at least once).

Heart Of Glass

From the above discussion, you'll probably be thinking that recursion is always bad. Actually, no it isn't. Thinking of an algorithm in terms of recursion often produces an elegant design and elegant code. Sometimes, we'll find that removing recursion either doesn't produce any benefit, or, if it does, the code becomes a maintenance nightmare and is probably not worth it.

Let's use another algorithm: calculating an integer power of a number. For example, to calculate 1.2^{50} involves multiplying 1.2 by itself 49 times:

```
Result := 1.2;
for i := 1 to 49 do
  Result := Result * 1.2
```

Right, so where's the problem? Actually, we can do better than this. Think about it like this: to calculate x^4, we can either multiply x by itself 3 times, or we can calculate x^2 (which is one multiplication) and multiply the result by itself (another multiplication). We have reduced 3 multiplications to 2. Larger powers would reduce the number of multiply operations even more. This algorithm is an example of a divide and conquer technique: we recast in terms of smaller numbers and thereby make it more efficient. We can posit our rules for the algorithm to calculate x^n as follows:

$x^0 = 1$
$x^1 = x$
if n is even then $x^n = (x^2)^{n/2}$
if n is odd then $x^n = x^{n-1}.x = (x^2)^{(n-1)/2}.x$

In other words, if n is even we calculate the square of x and then raise that to the power $n/2$, and there's a similar process if n is odd. Looking at these rules, we see that

```
function Power(X : double; N : cardinal) : double;
begin
  if (N = 0) then
    Result := 1.0
  else if (N = 1) then
    Result := X
  else begin
    if Odd(N) then
      Result := Power(X*X, (N-1) div 2) * X
    else {N is even}
      Result := Power(X*X, N div 2);
  end;
end;
```

➤ *Listing 3: Raising a number to an integer power.*

```
routine TailEndRecursion(Parameters)
  if Parameters have reached Limit
    do final stuff
  else
    do stuff
    call TailEndRecursion(smaller Parameters)
end
```

➤ *Listing 4*

```
routine TailEndWithLoop(Parameters)
  Finished = false
  while not Finished do
    if Parameters have reached Limit
      do final stuff
      Finished = true
    else
      do stuff
      make Parameters smaller
end
```

➤ *Listing 5*

we either produce an answer straight away (*n* = 0 or 1) or we can recast it in terms of (*n*-1)/2 or *n*/2, in other words, reducing the scale of the problem by a half until we reach the limiting cases, 0 and 1. Incidentally, we manage to avoid an infinite loop again.

Producing a recursive routine out of these rules is simplicity itself, please refer to Listing 3. For our example of 1.2^{50} we have to call the `Power` routine 6 times (for N = 50, 25, 12, 6, 3, 1) and there will be 7 multiplications done (when N is odd, two multiplications are done, when even, only one). Compare that with the original 49 multiplications.

Notice how easily the rules we had were converted into actual recursive code; the reason is, of course, that we originally stated the rules recursively. Anyway, the algorithm is very efficient: every time we recursively call the routine we do so with a number half the size it was when the routine was called. We've converted an algorithm that required a number of multiplications proportional to N, to one that requires a number proportional to logN.

Atomic

The `Power` routine we have created is an example of 'tail-end recursion'. This means that the recursive call to itself occurs at the end of the routine: after we call it recursively there's no other code to execute and we return immediately to our caller. In pseudo-code, a tail-end recursive routine looks like Listing 4.

The reason for making a fuss about tail-end recursion is that it is generally easy to remove. Look at the pseudo-code for the tail-end recursion. At the recursive call to `TailEndRecursion`, there is no longer any need for the original `Parameters`, after all, immediately we regain control after the recursive call we shall return to our caller. So we can remove the recursion quite simply by use of a dreaded `goto`: change the parameters for the smaller case and then jump to the top of the routine. But of course, since we are all followers of Edsger Dijkstra, we get rid of the `goto` by means of a flag (see Listing 5).

Bingo! No more recursion. No more needless calling ourselves again and again; instead: a nice fast loop.

So, let us apply this method to our recursive `Power` function. The first thing to note is that the recursive `Power` routine has two tails, so we need to remove them both. The tail when N is even is the easiest to remove: we multiply X by itself and divide N by 2, and then go round the loop again. The tail when N is odd is a little more involved. Think about how the recursive version works: we need to multiply X by X squared to the power of (N-1)/2. If we accumulate the powers of X so far in `Result`, then all we need to do is to multiply X by `Result` and store the answer back in `Result`. When N is zero, we're done and `Result` holds the correct value. When N is 1, we need to multiply X by `Result` and return that answer. I came up with Listing 6, which when cleaned up made Listing 7 (note that, in this latter listing, I'm making use of the fact that if N is odd, `N div 2` is the same as `(N-1) div 2`).

Having seen how to remove tail-end recursion, we'll now consider the harder case when recursion occurs in the middle of the routine.

Dreaming

Time for a story. In Hanoi, there is a secretive sect of Buddhist monks who are engaged on a mission. They have a tower of 64 disks, each of differing widths, stacked in one pile so that no disk is on a smaller disk. The disks are in fact threaded on a pole (each disk has a hole in the middle like a Polo or a Lifesaver). There are two other poles. Their job is to move all the disks from the original pole to another, one at a time. Each disk, when moved, must be threaded on a pole such that it doesn't cover a disk that is smaller. Once they complete the job (and they've been doing it, man and boy, for a very long time), Buddha will bring about the end of the universe.

Let's ignore the possibility that the end of the world is nigh, and concentrate on seeing how to do the job. Let's assume we have devised an algorithm (with many, many moves) with which we can move a stack of N disks from one pole to another. Calling the poles A, B, and C, with A as the original and B as the target, all the monks have to do is to move the 63 smallest disks from pole A to pole C, move the largest 64th disk from

```
function FastPower(X : double;
  N : cardinal) : double;
var Finished : boolean;
begin
  Result := 1.0;
  Finished := false;
  while not Finished do begin
    if (N = 0) then begin
      Finished := true;
    end else if (N = 1) then begin
      Result := X * Result;
      Finished := true;
    end else begin
      if Odd(N) then begin
        Result := X * Result;
        X := X * X;
        N := (N-1) div 2;
      end else {N is even} begin
        X := X * X;
        N := N div 2;
      end;
    end;
  end;
end;
```

➤ *Listing 6: Power routine with recursion removed, but messy.*

```
function CleanFastPower(X : double; N : cardinal) : double;
begin
  Result := 1.0;
  if (N = 0) then Exit;
  while True do begin
    if Odd(N) then
      Result  := X * Result;
    if (N = 1) then Exit;
    X := X * X;
    N := N div 2;
  end;
end;
```

➤ *Listing 7: Power routine with recursion removed and cleaned up.*

```
procedure Hanoi(N : byte; FromPole, ToPole, SparePole : char);
{move N disks from FromPole to ToPole using SparePole as a spare}
begin
  {if there's just one disk, move it}
  if (N = 1) then begin
    writeln('Move disk 1 from ', FromPole, ' to ', ToPole);
  end else {move more than one disk} begin
    {First: move N-1 disks to SparePole, using ToPole as the spare}
    Hanoi(N-1, FromPole, SparePole, ToPole);
    {Second: move the Nth disk}
    writeln('Move disk ', N, ' from ', FromPole, ' to ', ToPole);
    {Last: move N-1 disks from SparePole to ToPole, using FromPole as
     the spare}
    Hanoi(N-1, SparePole, ToPole, FromPole);
  end;
end;
```

➤ *Listing 8: The Tower of Hanoi recursive solution.*

pole A to pole B, and then move the 63 smallest disks from pole C to pole B. To move the 63 smallest disks the first time, they first have to move the 62 smallest disks from A to B, then move the 63rd disk to C and then move the 62 smallest disks to C. And so on, so forth, down to 1 disk. With one disk the solution is easy, we move it from the pole it's on to the pole it needs to be on. Poof, the end of the world!

Well, this algorithm calls out for recursion, so let's do it. What we'll do for a 'move' in our routine is to write out the disk number and how it's to be moved. Listing 8 was my result. If you look at it, you'll see that I pass the number of disks to move and I define each pole according to its function. There's the `FromPole` where the disks currently reside, the `ToPole` where I want them and the other pole is the `SparePole` which can be used to temporarily hold some disks. The routine follows the algorithm I described, exactly. Listing 9 shows the result of calling `Hanoi(3, 'A', 'B', 'C')`.

If you follow this with three coins of different sizes, you'll see that it is correct. For fun, I timed how long it would take, on average, to move a disk on my machine (in other words, I'm really timing the time it takes to output the string to the console, since that will swamp the time for the recursive calls, etc). It was about 2.6 milliseconds. Given that it takes $2^n - 1$ moves to move *n* disks (proof by induction), it would take well over 1.5 billion years to bring about the end of the world on my PC. We're safe.

Rip Her To Shreds

All right. Now I've saved the universe, back to work. The first thing to notice about the Tower of Hanoi recursive routine is that it is partly a tail-end recursion. We could remove that part quite simply (I won't show the result here though: see if you can do it). The problem though is the first recursive call to Hanoi. How do we get rid of that?

Recall at the beginning when I told you how recursive routines work at the machine level. The compiler inserts code to push the current data on the program stack before recursively calling the routine. Well, that's how we remove recursion for a recursive call in the middle of the routine: we simulate a stack.

We'll need to push and pop 4 values from the stack at a time: the disk number (or disk count), and the from, to and spare pole names. We rearrange the routine slightly to use our stack in a loop: each time round the loop we pop something off the stack and that tells us what to do for that iteration. In pseudo-code, it looks like Listing 10. Notice

```
Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C
Move disk 3 from A to B
Move disk 1 from C to A
Move disk 2 from C to B
Move disk 1 from A to B
```

➤ *Listing 9*

that the two recursive calls are pushed onto the stack in reverse order: a stack is a LIFO structure and as we want to process the first recursive call first, we must push its data last. The only other hack is that we must push something onto our stack to prompt us to make a move. To make a move we only need the disk number and the from and to poles, so we'll use a special value in the spare pole data item. If you look at the pseudo-code, it's as if we've turned the original Hanoi routine upside down. In fact, if we removed the recursion using a queue (a FIFO container) we would have the routine doing things in the same order as the original. However, it's easier and quicker to simulate a stack as we'll see.

There's one burning question left. How big should the stack be? Think about the original problem: before we can move N disks, we have to move N-1 disks, and before that, N-2 disks, and so on. Translating this into our stack-based

```
Hanoi routine(Parameters)
  push Parameters onto stack
  while stack is not empty do
    pop Parameters
    if the disk number is 1 or it's special
      write out the move
    else
      push Parameters for 2nd recursive call
      push special Parameters for move
      push Parameters for 1st recursive call
  endwhile
```

➤ *Listing 10: A pseudo-code stack-based Tower of Hanoi solution.*

routine, we push 3 items onto the stack and pop 1 every time round the loop until we get down to disk number 1. That's two pushes per disk. So, for 64 disks the stack should be 128 items in size (and do note that we'll never use them all because it just takes so long to do that many).

In Listing 11, notice how we declare a type to hold the parameters for a single invocation of the original routine, and the stack is a simple array of 128 elements of this type. The stack pointer is an integer, `SP`, that denotes the top of the stack. Pushing onto the stack involves setting the fields of the element at `SP` and then incrementing `SP`. Popping the stack involves decrementing `SP` and then accessing the element at `SP`. If `SP` is zero the stack is empty. The rest is easy to follow, using the pseudo-code in Listing 10 as a guide.

So was it worth it to remove the recursion in the Tower of Hanoi puzzle? The first, and most obvious, point to make is that the stack-based routine is a nightmare to look at. You have to have a pretty good idea what it's doing before you can begin to understand what's going on. Compare it with the original recursive routine: the latter is concise and elegant. Visualizing the process that's being done by looking at the first routine is easy; it's harder to see that the stack-based routine works.

The second, more telling point, is that the stack-based routine is *slower*. There's just so much extra work going on. I commented out the code that writes the moves to the console, and then timed both routines on a tower of 25 disks. On my machine, the recursive routine took 7.6 seconds and the stack-based routine 14.2 seconds, taking 85% longer.

Rapture

What conclusions can we come to? Firstly, if the algorithm you're considering calls out for a recursive routine, write one. Secondly, if the routine has a tail-end recursion, remove it and replace it with a loop. Test to see whether you've managed to speed it up. Thirdly, if the recursive call is in the middle of the routine, leave well enough alone. The recursive routine will probably be faster than one using an explicit stack and it certainly will be easier to read and maintain.

Union City Blues

I have a confession to make. When I started this article I thought I knew all there was to know about recursion. After all I'd been using the technique since the old Turbo Pascal days, and therein lay the problem. In the old days, you had a limited amount of program stack. If you had a process or routine that used a lot of stack, alarm bells were supposed to ring in your mind and you had to rethink something. One of the classic examples was a standard binary search tree, one without any balancing algorithms. Traversing such a tree which had degenerated into a linked list using a recursive algorithm was likely to blow your stack, and you had to rethink the problem (either using balancing algorithms during insert and delete, or traversing with the use of an external stack on the heap). In 32-bit Delphi programming, your stack is a default of 1Mb in size (compare *that* with BP7 real mode's default 16Kb stack) and so these problems don't surface all that much any more.

➤ *Listing 11: The stack-based Tower of Hanoi routine.*

```
procedure StackHanoi(N : byte; FromPole, ToPole,
  SparePole : char);
{move N disks from FromPole to ToPole using SparePole
 as a spare}
type
  TParam = record
    pN : byte;
    pFrom, pTo, pSpare : char;
  end;
var
  Stack : array [0..127] of TParam;
  SP    : integer;
  P     : TParam;
begin
  SP := 0;  {clear the stack}
  {push our parameters onto the stack}
  with Stack[SP] do begin
    pn := N;
    pFrom := FromPole;
    pTo := ToPole;
    pSpare := SparePole;
  end;
  inc(SP);
  {while the stack is not empty...}
  while (SP <> 0) do begin
    {pop the stack}
    dec(SP);
    longint(P) := longint(Stack[SP]);
    {if the disk number is 1, or it's a move, move it}
    if (P.pN = 1) or (P.pSpare = #0) then begin
      inc(MoveCount);
      writeln('Move disk ', P.pN, ' from ',
        P.pFrom, ' to ', P.pTo);
    end else begin
      {push the parameters to move N-1 disks from the
       spare to the to pole}
      with Stack[SP] do begin
        pN := P.pN - 1;
        pFrom := P.pSpare; pTo := P.pTo; pSpare := P.pFrom;
      end;
      inc(SP);
      {push the parameters for the move}
      with Stack[SP] do begin
        pn := P.pN;
        pFrom := P.pFrom; pTo := P.pTo; pSpare := #0;
      end;
      inc(SP);
      {push the parameters to move N-1 disks from the
       from to the spare pole}
      with Stack[SP] do begin
        pn := P.pN - 1;
        pFrom := P.pFrom; pTo := P.pSpare; pSpare := P.pTo;
      end;
      inc(SP);
    end;
  end;
end;
```

So, I've changed my position on recursive routines, especially those containing recursion that cannot be categorized as tail-end recursion. I would now leave such routines as recursive.

The code that accompanies this article is freeware and can be used as-is in your own applications, several times.

Julian Bucknall often gets a sense of *déjà vu*, sometimes when he's already having a feeling of *déjà vu*. One of his cats has a nickname of Ari, but none of them are called Debbie. He can be contacted at julianb@turbopower.com

Errata

A user of my EZDSL data structures library pointed out an error in the implementation of its hash table class. Since I derived it from the hash table I wrote for my February and March 1998 articles in *The Delphi Magazine*, I thought I should pass it on. In the hash table, I was using an external hash function to return an unsigned 32-bit integer. Dear old Delphi still doesn't have one (we'll see about Delphi 4), so the hash routine was in fact returning a longint. The htlHash method in the class took this value and performed a MOD operation with the table size to obtain an index into the hash array. Normally, well in mathematics anyway, the modulus operation will produce a value between 0 and one less than the divisor. Well, I'd completely forgotten that Delphi's MOD operator, when presented with a negative number, will produce a *negative* result. And, as far as the compiler was concerned, the result of the hash function was a signed integer, no matter what I might have thought it was. The result was an attempt to access an element of the table at a negative index. The solution is to normalize the result of the MOD operator (if it was negative) by adding the table size. If you download EZDSL 3.01 from my website (www.home.turbopower.com/~julianb) then you'll have the corrected version.

Talking of hash tables, TurboPower has just released version 2 of SysTools. Whilst preparing for it, TurboPower's Gary Frerking discovered a peculiarity about the hash function being used by SysTools' string dictionary class, and I confirmed it with a test program. If the hash function was presented with a set of independent strings (in other words, they tended not to look like each other) then the hash function produced a nice spread of values with approximately the right number of collisions, as predicted by theory. However, if the hash function was presented with strings with a high degree of similarity instead (in essence differing in just a few characters) then the number of collisions in the returned values went way up. The hash function broke down. I replaced it with the ELF hash function which I used in my articles. This hash function is much better behaved in the latter situation.

Finally, the author of *Algorithm K* (see my May article on random numbers), got in touch with me. He was most gracious and thanked me for showing the problem with the algorithm. I too can be gracious if you see a problem with my articles, and I'll write them up in a sidebar like this. But, you've got to let me know first!