What's The Difference

This month we look at the longest common subsequence

by Julian Bucknall

Back at TurboPower HQ, we have a set of bookcases holding programming and computer magazines. Every now and then we have to cull them a little, otherwise we'd be swamped in glossy paper. This has been made easier to do ever since magazines put all their back issues on CD: the amount of room taken by a CD is a couple of magnitudes smaller than that taken by five years' worth of magazines.

The last time this happened, a couple of years back, I had a fun half-hour flipping through back issues of *Byte* and *Dr Dobbs*, and so on. There were some even older magazines, long dead, whose titles have been forgotten, and a box of clippings and photocopies of old articles. This box certainly was fascinating, illuminating the history of TurboPower through the choice that someone had made in selecting them. There were articles on protected mode DOS, writing assembly language TSRs (*Terminate and Stay Resident* programs for you young 'uns), and writing directly to the CGA video memory buffer without causing snow on the screen. All interesting stuff for a programmer like me who remembers using an original PC when a 10Mb hard drive was a luxury.

Anyway, also in this box were a series of algorithm articles. Being an inveterate hoarder of these things I kept some of them, thinking that they might trigger an article some day. One of them has been niggling at me ever since I read it: *What's the Diff? A File Comparator for CP/M Plus*, by D E Cortesi, published in *Dr Dobbs Journal* in August 1984. It seemed very arcane to me when I read it. Also, it didn't help that the article said things like: 'Pascal is not the best language for this [Diff] program. It really needs more freedom to allocate storage.' Well, ptouii to that! The source code supplied by the author was Standard Pascal to the core, and I put a project onto the back burner to research it more one day.

What the article was trying to show, and what I wanted to do, was write a routine that would take two copies of the same text file, one being a later edition of the other, presumably containing several differences, and work out what changes were made in going from the older file to the newer. Unix has a program to do this called DIFF, which is the granddaddy of all file difference programs. I use one from the Windows SDK called WinDiff. This displays the two versions of the file as vertical bars side by side with lines going from the left bar to the right bar recording the changes. The red lines are deletions, the yellow ones insertions. You can also see the text lines as well in the same manner (text lines in red are deleted, text lines in yellow are inserted). This tool is invaluable for me; for example, I use it to determine changes in the VCL source code between one version of Delphi and the next. Visual SourceSafe (the version control system we use) also has a tool like this for showing the differences between one version of a source file and another: this has saved my bacon many a time.

In the meantime, I did do a little research on file comparison algorithms, but didn't turn up anything interesting. It didn't help that I didn't know what I was looking for: I had no idea what the algorithm was called that did what I wanted. Also, I felt a little foolish searching for 'file comparator', something that sounded like it had been used in a 1950s science fiction B-movie. So, I put it aside: sometimes I've found that if I'm not looking for something it turns up eventually.

And so it has. A couple of weeks ago I was idly flipping through an algorithms book, looking for something else, when a diagram in the text caught my eye: it looked a little like the WinDiff display but shown horizontally rather than vertically. The topic was how to work out the *longest common subsequence*. Bingo! I had the name. And then of course I found it in lots of books and, after a bit more research, thought it would make a good algorithm to discuss in this column.

Initial Thoughts

So how's it done? Before I reveal all, let's think about the problem. We have two text files, let's call them the *source* file and the *destination* file. We assume that they represent two different versions of the same text, with the destination file being a later version of the source file. We want to be able to show the lines that were changed going from one to the other. By *changed*, what do I mean? Well, boiled down to its essence, all changes to a text file can be viewed as *deletions* and *insertions*. In other words, if someone changed a line by adding a word to it, we won't track that as an alteration of a single line, but as a deletion of the original line, followed by an insertion of the new one with the same text and the extra word. That way, we don't get drawn into semantic difficulties trying to decide when a line is merely changed a little versus one that was completely rewritten.

So, there we are with two files. Let's make it easier for ourselves and assume that we've read them into their own string lists. Talking about array accesses is easier than talking about how to seek to a line and read it in a text file. Without the benefit of hindsight, I suppose I'd be trying out something like this to mark the changes. Read the two files in parallel, comparing one line from the source with one from the destination, until we reach the first difference. This difference may be due to two things: the source line was deleted or the destination line was inserted. Let's assume the second.

I'd make a note of where we were in the destination file, and then continue reading through the destination file until I ran out of lines or I found a line equal to the source line. If I ran out of lines I'd assume that the source line had been deleted, and seek through the source file until I found a line equal to the destination line. And so on, and so forth.

I hope you can see the complexity of this first-stab-at-it algorithm. For every line in the source or destination file, you could find yourself reading through the rest of the opposite file. And there's no guarantee you'd get anything sensible out at the end. Consider, for example, the number of lines in a Delphi source file that just consist of the text 'begin' or 'end;' starting at the first character. You could find yourself jumping over whole swathes of lines to match up with the wrong 'end;'. No, all in all, our 'let's take a stab at this' algorithm isn't worth bothering with.

Strings Are Easier

Now we've exhausted our initial take on inventing an algorithm, we'll take a look at the longest common subsequence algorithm. For our initial exposition, we shall assume that we are trying to find the differences between two strings. Then, having seen how it works for strings, we'll extend it to files.

I'm sure we've all played those children's word puzzles where you change one word into another by altering a single letter at a time. All the intermediary steps should be words as well. So, to take a simple example, to change CAT into DOG, we might take the following steps: CAT, COT, COG, DOG. Changing WORK into PLAY is much more difficult; after about 10 minutes I came up with WORK, WORD, WOAD, ROAD, READ, REAR, PEAR, PEAN, PLAN, and PLAY.

Anyway, these word games were merely deleting a letter and inserting a new one at each step. If we didn't have the limitations imposed by the rules of the puzzle, we could certainly transform any word into another by deleting all the old characters and inserting all the new ones. That's the sledgehammer approach, but we'd like to be a little more subtle.

Suppose our goal were to find the smallest number of edits needed to convert one word to another. Let's take, as example, changing BEGIN to FINISH. Looking at this you can see that we should delete B, E, G, and then insert F before what's left, and I, S, H afterwards. So how do we implement this as an algorithm, without resorting to the 'it's easy' answer?

One way is to look at the subsequences of each word and see if we can't get two subsequences to match up. A *subsequence* of a string is the string less one or more characters. The remaining characters should not be rearranged. For example, the four-letter subsequences of BEGIN are EGIN, BGIN, BEIN, BEGN, and BEGI. As you can see, you form them by dropping each character in turn. The three-letter subsequences are BEG, BEI, BEN, BGI, BGN, BIN, EGI, EGN, EIN, and GIN. There are 10 two-letter subsequences and 5 single letter ones. So for a five letter word there are a total of 30 possible subsequences and, in fact it can be shown that for an *n* letter sequence the number of subsequences is about 2^n. Hold that thought.

The brute-force algorithm, if I may call it that, is to look at the two words BEGIN and FINISH and enumerate their 5-letter subsequences to see if any match. No, so do the same for the 4-letter subsequences of each word. Again, no, proceed to the 3-letter subsequences. Yet another no, and we move on to the 2-letter subsequences. Bingo! There's IN. From that we can work out what to delete and what to insert.

Now for small words, like our example, this process isn't too bad. But imagine that we're looking at a 100-letter 'word'. This is where the thought comes in that I asked you to hold. The brute-force algorithm is *exponential*. For even medium-sized data sets, the algorithm's search space grows alarmingly fast. And with the growth in the search space comes a dramatic increase in the time taken to find the solution. To drive home the point: suppose we could generate one billion subsequences per second (that is, 2^{30}, or one subsequence per cycle on a one gigahertz PC). A year is about 2^{25} seconds, so, to generate the entire set of subsequences for a 100-letter word would take 2^{45} years; a number with 14 digits. And remember here that the 100-letter word is merely a simplification of what we want to do: find differences for a 600-line source file, for example.

Thanks, but no thanks; I've got better things to do.

The subsequence idea does have its merits, though, we just need to approach it from a different angle. Instead of enumerating all of the subsequences in the two words and comparing, let's see if we can't do it in a stepwise progression.

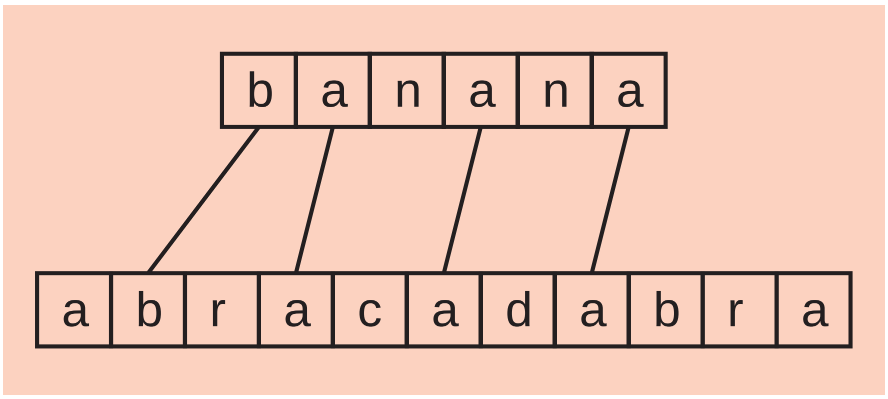
Longest Common Subsequence

Let's suppose we have worked out a longest common subsequence for two words (we'll abbreviate *longest common subsequence* to LCS from now on). We could then draw lines between the letters in the LCS from the first word to the corresponding ones in the second word. These lines would not cross. (Why? Because a subsequence is defined so that no rearrangement of the letters is allowed, therefore the letters in the LCS would appear in the same order in both words.) Figure 1 shows the LCS for the words banana and abracadabra (that is, b, a, a, a) with lines drawn to show the equivalent subsequence letters. Notice that there are several possible longest common subsequences for this word pair: the figure just shows the first (the one that appears closest to the left).

Let's assume that we have worked out, one way or another, an LCS between the two words. The length of this subsequence is *x*, say. Take a look at the final letters for the two words. If they are the same letter then this *must* appear as the final letter in the LCS, and there would be a linking line between them. (If it doesn't appear as the final letter of the subsequence, then we could add it, making the LCS one letter longer, contradicting our assumption about having the longest one in the first place.) Remove this final letter from the two words and also from the subsequence. This shortened subsequence of length *x*-1 is an LCS of the two abbreviated words. (If it were not, then there would be a common subsequence of *x* or larger for the two abbreviated words. Adding in the final letters would increase the length of this new common subsequence by one, so that there would be a common subsequence between the complete words of *x*+1 letters or longer. This now contradicts our assumption that we had an LCS.)

Suppose now that the final letter in the LCS were not the same as the final letter of the first word. This would mean that the LCS between the two complete words was also the LCS between the first word less its final letter and the second word (if it were not, we could add back the final letter of the first word and find a longer LCS for the two words). The same argument applies to the case where the final letter of the second word was not the same as the final letter in the LCS.

All very well, but what does this show? A longest common subsequence contains within it a longest common subsequence of the truncated parts of the two words. To find an LCS of *X* and *Y*, we break the problem down into smaller problems. If the final character of *X* and *Y* were the same, we would have to find the LCS of *X* and *Y* minus their final letters, and then add in this common letter. If not, we would have to find out the LCS of *X* minus its final letter and *Y*, and that of *X* and *Y* minus its final letter, and choose the longer of the two. This almost seems too good to be true: a simple recursive algorithm. Yippee!



➤ *Figure 1*

Aside On Fibonacci

Before we put fingers to keyboard, let me add a word of caution by bringing up a calculation and what seems to be a simple recursive solution to it. To calculate the *n*th Fibonacci number, we could use its definition: it is equal to the sum of the two previous Fibonacci numbers. Given that the first and second Fibonacci numbers are both equal to 1, we could write the recursive routine in Listing 1 to calculate the *n*th Fibonacci number. We would get the usual series: 1, 1, 2, 3, 5, 8, 13, 21, etc.

But consider what is happening under the hood. Let's calculate the 5th Fibonacci number. To do that, we need to calculate the 3rd and 4th ones. To calculate the 3rd Fibonacci number we would have to calculate the 1st and the 2nd. The 1st, by definition is 1, and the 2nd is also one. So the 3rd is 2. To calculate the 4th, we have to calculate the 2nd and the 3rd. The 2nd is 1 by definition. The 3rd is the sum of the 1st and the 2nd. The 1st is 1 and so is the 2nd, and so the 3rd Fibonacci number is 2. That, in turn, makes the 4th equal to 3. And finally we can calculate the 5th to be 5. In doing this recursive calculation we worked out the value of F1 twice, F2 three times, F3 twice, and F4 once. Nasty. Of course, this simple example shows that sometimes recursive solutions are not always the best, despite the fact that they seem to fit rather well.

The problem with the recursive solution to the Fibonacci calculation is that there are two recursive calls inside the routine. These recursive calls will make their own recursive calls and, lo and behold, you end up calculating the same values over and over again. The same problem occurs with our recursive solution to calculating the LCS: we can (and will) end up calculating the same LCS over and over again: a sheer waste of effort and time.

What can we do? The recursive solution certainly has its merits. Go back to the Fibonacci case. If we wanted to use the recursive method, what could we do to speed things up? One answer would be to create an array to contain the Fibonacci numbers we've calculated so far. Set all elements

➤ *Listing 1: The slow recursive Fibonacci routine.*

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

of the array to -1 (to signify 'uncalculated') except for the first two, which are both set to 1. Now write the Fibonacci routine to look in the array for the answer. If the answer is not there, it makes the recursive calls to calculate the answer. Once a call has calculated the answer, it fills the relevant element of the array. As you can imagine, you would be reducing the number of recursive calls quite dramatically by using this simple cache. Listing 2 shows this more efficient recursive routine. (As an aside, this is an example of making calculations faster at the expense of extra memory.) To show you the effect of this cache, I ran both versions of the program (the recursive version without the cache and the one with it) to calculate the 40th Fibonacci number. The version with caching was virtually instantaneous; the version without took over 25 seconds.

```
procedure PrepareFibCache;
var
  i : integer;
begin
  FibCache := TList.Create;
  FibCache.Count := 1001;
  FibCache[1] := pointer(1);
  FibCache[2] := pointer(1);
  for i := 3 to 1000 do
    FibCache[i] := pointer(-1);
end;
function FastFibonacci(N : integer) : integer;
begin
  Result := integer(FibCache[N]);
  if (Result = -1) then begin
    Result := FastFibonacci(N-2) + FastFibonacci(N-1);
    FibCache[N] := pointer(Result);
  end;
end;
```

➤ *Listing 2: The faster recursive Fibonacci with a cache.*

Calculating The LCS

Back to the algorithm for calculating the LCS. The description I gave earlier was accurate but verbose. Let's reduce it to its basics. First, we assume that the *X* string has *n* characters and the *Y* string has *m*. We shall write X_i to mean the string formed from the first *i* characters of X. *i* can also take the value zero to mean the empty string (this convention will make things easier to understand in a moment). X_n is then the whole string. Using this nomenclature, the algorithm reduces to this: if the last two characters of X_n and Y_m are the same, the longest common subsequence is equal to the LCS of X_{n-1} and Y_{m-1} plus this last character. If they are not the same, the LCS is equal to the longer of the LCS of X_{n-1} and Y_m and the LCS of X_n and Y_{m-1}. To calculate these 'smaller' LCSs we of course recursively call the same routine.

You saw that it is possible to speed up a recursive algorithm by using a cache. Unlike the Fibonacci example with its linear array, the LCS algorithm will require us to create and fill a matrix. As a first thought, each element of the matrix would be a string showing the LCS for that particular case. However, if you think about it, that's not too helpful. We would like to know the *index* of the matching characters, not the matching characters themselves, for then we could work out the sequence of deletions and insertions to get from the first string to the second. Hence a better idea would be to store a string of character indexes at each element, each index defining a matching character in the LCS. The problem with that idea, though, is that there would have to be two strings of indexes, one for *X* and one for *Y*.

Even that is way overkill: a better approach would be to store enough information at each element of the matrix to enable us to rebuild the LCS at will. When we build the matrix we would have to store the length of the LCS so far (for without at least that piece of information, it would be hard indeed to work out the *longest* common subsequence!). The other information we would need to store would be a pointer to the previous element that was used to build the LCS for this element. That way we could work our way back from the final cell to the beginning (and, if we used a recursive routine to walk the matrix, we could easily work out the LCS).

We're getting ahead of ourselves here. Before we can discuss walking the LCS matrix, we have to build one. For now, each element of the matrix will store two pieces of information: the length of the LCS at that point and the position of the previous matrix element that forms the prequel for this LCS. There are only three possible cells for this latter value: the one just above (north), the one to the left (west), and the one on the upper left diagonal (northwest), so we might as well use an enumerated type for this.

Let's calculate the LCS by hand for the BEGIN/FINISH case. We'll have a 6x7 matrix (we take into account empty substrings, so we should start indexing at 0). Rather than fill in the matrix recursively (it's hard for us to keep all those recursive calls straight), we'll calculate all the cells iteratively from the top left all the way down to the bottom right, going from left to right along each row for every row. The first row and column are easy: all zeroes. Why? Because the longest common subsequence between an empty string and any other string is zero, that's why. From this we can start working out the LCS for cell (1,1), or the two strings B and F. The two final characters of these one-character strings are not equal, therefore the length of the LCS is the maximum of the previous cells to the north and west. This is zero, so the value of the cell is zero. Cell (1,2) is for the strings B and FI. Again, zero. Cell (2,1) is for BE and F: the LCS length is zero again. Continuing like this we can fill in all the 42 cells in the matrix. Notice the cells for the matching characters: this is where the LCS length gets greater. Table 1 shows the answer.

Writing this manual process in code is not too bad. For a start, I decided early on to make the matrix a class. Internal to this class, the matrix is held as a TList of TLists, with the major TList being rows in the matrix and the minor TLists, cells across the columns for a particular row. To make it even easier still (I didn't want to spend precious time writing the code to grow the matrix by increasing the number of rows or columns, for example) I made the matrix class specific to the LCS problem in hand: all right, I admit I didn't have a nice matrix class already written! So simplicity was the order of the day. I won't bother showing the matrix class here: the code is fairly trivial and it's on this month's disk anyway.

Iterative Versus Recursive

The LCS code is also fairly easy to write. I was torn when I first coded it: should I follow the recursive method already outlined, or should I follow the manual process I just described? For fun, and for experimentation's sake, I wrote both. Listing 3 shows the iterative solution. We start off by filling the top row and the left column of the matrix with 'null' cells. These cells all have an LCS length of zero (remember that they describe an LCS between an empty string and another), and I just set the direction flag to point to the previous cell that's closer to (0,0). Next comes the loop within a loop. For every row, we calculate the LCS for each of the cells from left to right. We do this for all rows from top to bottom. First test is the test to see whether the two characters referenced by the cell are equal. (A cell in the matrix is at the junction of a character in the from string and one in the to string.) If they are, then we know that the LCS length at this cell is equal to the LCS length from the cell adjacent at the northwest, plus one. Notice that the way we're calculating the cells means that this cell being referenced has already been calculated (that's one reason why we 'pre-calculated' the cells along the top and left sides). If the two characters are *not* equal, then we have to look at the cell to the north and the one to the west. We select the one that has the longest LCS length, and use that length for this cell. If the two lengths are equal, we could select either one. We shall, however, make a rule that we would preferentially choose the one to the left. The reason for this is that, once we have calculated a path through the matrix to produce the LCS of both strings, the deletions from the first string will occur before the insertions into the second string.

Notice that the method shown in Listing 3 takes a constant time for two strings, no matter how many similarities there are. If the two strings have length *n* and *m*, the time taken in the main loop will be proportional to *n*m*, since that's the number of cells you will have to calculate (the cell for which you *really* want the answer is the last one to be calculated).

As I said, I wrote this method in preference to the recursive one and played around with it for a while. One of the examples I chose was converting *illiteracy* to *innumeracy*. This pair of words has an LCS of 6: i, e, r, a, c, y. When you plot the matrix (see Table 2) you can see that the end of the LCS path is diagonal. (Table 2 shows characters that show the direction to follow for each cell: remember that you should take a pen, start at the bottom right hand cell and move to the top left one, following the |, \, or - directions. Once you've traced the path, you can follow it back to describe the changes that need to be made to convert the first string into the second. If you go down, you delete a character. If you move right, you

		F	I	N	I	S	H
	0	0	0	0	0	0	0
B	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0
I	0	0	1	1	1	1	1
N	0	0	1	2	2	2	2

➤ *Table 1: Calculating the LCS matrix for BEGIN/FINISH.*

➤ *Listing 3: Calculating the LCD matrix for a pair of strings via iteration.*

```
procedure TaaStringLCS.slFillMatrix;
var
  FromInx : integer;
  ToInx   : integer;
  FromCh  : PAnsiChar;
  ToCh    : PAnsiChar;
  NorthLen: integer;
  WestLen : integer;
  LCSData : PaaLCSData;
begin
  {Create the empty items along the top and left sides}
  for ToInx := 0 to length(FToStr) do begin
    New(LCSData);
    LCSData.ldLen := 0;
    LCSData.ldPrev := ldWest;
    FMatrix[0, ToInx] := LCSData;
  end;
  for FromInx := 1 to length(FFromStr) do begin
    New(LCSData);
    LCSData.ldLen := 0;
    LCSData.ldPrev := ldNorth;
    FMatrix[FromInx, 0] := LCSData;
  end;
  {fill in the matrix, row by row, from left to right}
  FromCh := PAnsiChar(FFromStr);
  for FromInx := 1 to length(FFromStr) do begin
    ToCh := PAnsiChar(FToStr);
    for ToInx := 1 to length(FToStr) do begin
      {create the new item}
      New(LCSData);
      {if the two current chars are equal, increment the
        count from the northwest, that's our previous item}
      if (FromCh^ = ToCh^) then begin
        LCSData^.ldPrev := ldNorthWest;
        LCSData^.ldLen :=
          succ(FMatrix[FromInx-1, ToInx-1]^.ldLen);
      end
      { otherwise the current characters are different:
        use the maximum of the north or west (west is
        preferred)}
      else {current chars are different} begin
        NorthLen := FMatrix[FromInx-1, ToInx]^.ldLen;
        WestLen := FMatrix[FromInx, ToInx-1]^.ldLen;
        if (NorthLen > WestLen) then begin
          LCSData^.ldPrev := ldNorth;
          LCSData^.ldLen := NorthLen;
        end
        else begin
          LCSData^.ldPrev := ldWest;
          LCSData^.ldLen := WestLen;
        end;
      end;
      {set the item in the matrix}
      FMatrix[FromInx, ToInx] := LCSData;
      {move one char on in the to string}
      inc(ToCh);
    end;
    {move one char on in the from string}
    inc(FromCh);
  end;
  {at this point the item in the bottom right hand corner
    has the length of the LCS and the calculation is
    complete}
end;
```

insert one. A diagonal indicates no change.) The parts of the matrix above that diagonal and to its left do not need to be calculated: they can play no part in the final LCS. That got me thinking about the recursive method again, and so I wrote it.

Listing 4 shows this recursive method. It's coded as a function that returns the LCS length for a particular cell, given by its row and column index (which are after all indexes into the from string and the to string). First big difference: we don't have to generate the 'null' cells along the top and down the left side, that's now taken care of with a simple `if` statement. (To be fair, we could get away without calculating them in the iterative case, but the inner code in the loop would become more complicated to understand and maintain, and so, in the interest of simplicity, we pre-calculated those cells.) If the cell has already been calculated, we simply return its LCS length. If not, we do the same checking as before: are the two characters equal? Yes, add one to the LCS length from the cell at the northwest. No, use the larger LCS length value from the cells at the north or at the west. These LCS values are of course calculated from recursive calls to this routine.

Using this recursive version, I generated the matrix for *illiteracy* to *innumeracy*. My thoughts were confirmed: Table 3 shows that many of the cells are simply not calculated; these are the ones with a question mark. (Please note that it's not just a question of calculation, it's also the fact that we have to allocate a cell off the heap. Saving both calculation and allocation time is not to be sneezed at, but admittedly in these simple examples, it's not that important. It'll be more significant later on when we deal with text files again.)

Generating The Edit Sequence

Great, so now we have a matrix that defines the longest common subsequence. How can we use it? I decided to write a routine that created a text file that described the changes. This would make it easier for us to write the equivalent for the file case: the ultimate aim of this article. Also, by doing it this way, we won't get embroiled in bizarre formats or different needs. Instead we shall see a simple traversal technique, shorn of complexity, which we can bend to our own desires. Listing 5 shows this code. It comprises two methods: the first that gets called by the user with a file name, and the second a recursive routine that writes the data to the file. All the action is in this second routine. Since the matrix encodes the LCS path backwards (in other words you have to start at the finish, work your way back to the start to discover the path that you can then follow forwards) we write the method to call itself recursively first and then write out the data for the current position. For a recursive routine we have to make sure it terminates (my first coding attempt at this didn't!), and this is taken to be the case where the

➤ *Table 2: The full LCS matrix for illiteracy/innumeracy.*

```
         i    n    n    u    m    e    r    a    c    y
  - 0  - 0  - 0  - 0  - 0  - 0  - 0  - 0  - 0  - 0  - 0
i | 0  \ 1  - 1  - 1  - 1  - 1  - 1  - 1  - 1  - 1  - 1
l | 0  | 1  - 1  - 1  - 1  - 1  - 1  - 1  - 1  - 1  - 1
l | 0  | 1  - 1  - 1  - 1  - 1  - 1  - 1  - 1  - 1  - 1
i | 0  \ 1  - 1  - 1  - 1  - 1  - 1  - 1  - 1  - 1  - 1
t | 0  | 1  - 1  - 1  - 1  - 1  - 1  - 1  - 1  - 1  - 1
e | 0  | 1  - 1  - 1  - 1  - 1  \ 2  - 2  - 2  - 2  - 2
r | 0  | 1  - 1  - 1  - 1  - 1  | 2  \ 3  - 3  - 3  - 3
a | 0  | 1  - 1  - 1  - 1  - 1  | 2  | 3  \ 4  - 4  - 4
c | 0  | 1  - 1  - 1  - 1  - 1  | 2  | 3  | 4  \ 5  - 5
y | 0  | 1  - 1  - 1  - 1  - 1  | 2  | 3  | 4  | 5  \ 6
```

➤ *Table 3: The partial LCS matrix for illiteracy/innumeracy.*

```
         i    n    n    u    m    e    r    a    c    y
  ? 0  ? 0  ? 0  ? 0  ? 0  ? 0  ? 0  ? 0  ? 0  ? 0  ? 0
i ? 0  \ 1  - 1  - 1  - 1  - 1  ? 0  ? 0  ? 0  ? 0  ? 0
l ? 0  | 1  - 1  - 1  - 1  - 1  ? 0  ? 0  ? 0  ? 0  ? 0
l ? 0  | 1  - 1  - 1  - 1  - 1  ? 0  ? 0  ? 0  ? 0  ? 0
i ? 0  \ 1  - 1  - 1  - 1  - 1  ? 0  ? 0  ? 0  ? 0  ? 0
t ? 0  | 1  - 1  - 1  - 1  - 1  ? 0  ? 0  ? 0  ? 0  ? 0
e ? 0  ? 0  ? 0  ? 0  ? 0  ? 0  \ 2  ? 0  ? 0  ? 0  ? 0
r ? 0  ? 0  ? 0  ? 0  ? 0  ? 0  ? 0  \ 3  ? 0  ? 0  ? 0
a ? 0  ? 0  ? 0  ? 0  ? 0  ? 0  ? 0  ? 0  \ 4  ? 0  ? 0
c ? 0  ? 0  ? 0  ? 0  ? 0  ? 0  ? 0  ? 0  ? 0  \ 5  ? 0
y ? 0  ? 0  ? 0  ? 0  ? 0  ? 0  ? 0  ? 0  ? 0  ? 0  \ 6
```

➤ *Listing 4: Calculating the LCD matrix for a pair of strings via recursion.*

```
function TaaStringLCS.slGetCell(aFromInx, aToInx :
  integer) : integer;
var
  LCSData : PaaLCSData;
  NorthLen: integer;
  WestLen : integer;
begin
  if (aFromInx = 0) or (aToInx = 0) then
    Result := 0
  else begin
    LCSData := FMatrix[aFromInx, aToInx];
    if (LCSData <> nil) then
      Result := LCSData^.ldLen
    else begin
      {create the new item}
      New(LCSData);
      {if the two current chars are equal, increment the
        count from the northwest, that's our previous item}
      if (FFromStr[aFromInx] = FToStr[aToInx]) then begin
        LCSData^.ldPrev := ldNorthWest;
        LCSData^.ldLen :=
          slGetCell(aFromInx-1, aToInx-1) + 1;
      end
      {otherwise the current characters are different: use
        the maximum of the north or west (west preferred)}
      else begin
        NorthLen := slGetCell(aFromInx-1, aToInx);
        WestLen := slGetCell(aFromInx, aToInx-1);
        if (NorthLen > WestLen) then begin
          LCSData^.ldPrev := ldNorth;
          LCSData^.ldLen := NorthLen;
        end
        else begin
          LCSData^.ldPrev := ldWest;
          LCSData^.ldLen := WestLen;
        end;
      end;
      {set the item in the matrix}
      FMatrix[aFromInx, aToInx] := LCSData;
      {return the length of this LCS}
      Result := LCSData^.ldLen;
    end;
  end;
end;
```

```
procedure TaaStringLCS.slWriteChange(var F : System.Text;
  aFromInx, aToInx : integer);
var
  Cell : PaaLCSData;
begin
  {if both indexes are zero, this is the first
   cell of the LCS matrix, so just exit}
  if (aFromInx = 0) and (aToInx = 0) then
    Exit;
  {if the from index is zero, we're flush against the left
   hand side of the matrix, so go up; this'll be a deletion}
  if (aFromInx = 0) then begin
    slWriteChange(F, aFromInx, aToInx-1);
    writeln(F, '-> ', FToStr[aToInx]);
  end
  {if the to index is zero, we're flush against the top side
   of the matrix, so go left; this'll be an insertion}
  else if (aToInx = 0) then begin
    slWriteChange(F, aFromInx-1, aToInx);
    writeln(F, '<- ', FFromStr[aFromInx]);
  end
  {otherwise see what the cell says to do}
  else begin
    Cell := FMatrix[aFromInx, aToInx];
    case Cell^.ldPrev of
      ldNorth :
        begin
          slWriteChange(F, aFromInx-1, aToInx);
          writeln(F, '<- ', FFromStr[aFromInx]);
        end;
      ldNorthWest :
        begin
          slWriteChange(F, aFromInx-1, aToInx-1);
          writeln(F, '   ', FFromStr[aFromInx]);
        end;
      ldWest :
        begin
          slWriteChange(F, aFromInx, aToInx-1);
          writeln(F, '-> ', FToStr[aToInx]);
        end;
    end;
  end;
end;
procedure TaaStringLCS.WriteChanges(const aFileName :
  string);
var
  F : System.Text;
begin
  System.Assign(F, aFileName);
  System.Rewrite(F);
  try
    slWriteChange(F, length(FFromStr), length(FToStr));
  finally
    System.Close(F);
  end;
end;
```

Above, Listing 5: Writing out the changes to convert one string to another.

Below, Listing 6: The class for calculating the LCS for two files.

```
constructor TaaFileLCS.Create(const aFromFile, aToFile :
  string);
begin
  {create the ancestor}
  inherited Create;
  {read the files}
  FFromFile := TStringList.Create;
  FFromFile.LoadFromFile(aFromFile);
  FToFile := TStringList.Create;
  FToFile.LoadFromFile(aToFile);
  {create the matrix}
  FMatrix := TaaLCSMatrix.Create(FFromFile.Count,
    FToFile.Count);
  {now fill in the matrix}
  slGetCell(pred(FFromFile.Count), pred(FToFile.Count));
end;
destructor TaaFileLCS.Destroy;
begin
  {destroy the matrix}
  FMatrix.Free;
  {free the string lists}
  FFromFile.Free;
  FToFile.Free;
  {destroy the ancestor}
  inherited Destroy;
end;
function TaaFileLCS.slGetCell(aFromInx, aToInx : integer) :
  integer;
var
  LCSData : PaaLCSData;
  NorthLen: integer;
  WestLen : integer;
begin
  if (aFromInx = -1) or (aToInx = -1) then
    Result := 0
  else begin
    LCSData := FMatrix[aFromInx, aToInx];
    if (LCSData <> nil) then
      Result := LCSData^.ldLen
    else begin
      {create the new item}
      New(LCSData);
      {if the two current lines are equal, increment the
        count from the northwest, that's our previous item}
      if (FFromFile[aFromInx] = FToFile[aToInx]) then begin
        LCSData^.ldPrev := ldNorthWest;
        LCSData^.ldLen :=
          slGetCell(aFromInx-1, aToInx-1) + 1;
      end
      {otherwise the current lines are different: use the
        maximum of the north or west (west preferred)}
      else begin
        NorthLen := slGetCell(aFromInx-1, aToInx);
        WestLen := slGetCell(aFromInx, aToInx-1);
        if (NorthLen > WestLen) then begin
          LCSData^.ldPrev := ldNorth;
          LCSData^.ldLen := NorthLen;
        end
        else begin
          LCSData^.ldPrev := ldWest;
          LCSData^.ldLen := WestLen;
        end;
      end;
      {set the item in the matrix}
      FMatrix[aFromInx, aToInx] := LCSData;
      {return the length of this LCS}
      Result := LCSData^.ldLen;
    end;
  end;
end;
procedure TaaFileLCS.slWriteChange(var F : System.Text;
  aFromInx, aToInx : integer);
var
  Cell : PaaLCSData;
begin
  {if both indexes are less than zero, this is the first
    cell of the LCS matrix, so just exit}
  if (aFromInx = -1) and (aToInx = -1) then
    Exit;
  {if the from index is less than zero, we're flush against
    the left hand side of the matrix, so go up; this'll be a
    deletion}
  if (aFromInx = -1) then begin
    slWriteChange(F, aFromInx, aToInx-1);
    writeln(F, '-> ', FToFile[aToInx]);
  end
  {if the to index is less than zero, we're flush against
    the top side of the matrix, so go left; this'll be an
    insertion}
  else if (aToInx = -1) then begin
    slWriteChange(F, aFromInx-1, aToInx);
    writeln(F, '<- ', FFromFile[aFromInx]);
  end
  {otherwise see what the cell says to do}
  else begin
    Cell := FMatrix[aFromInx, aToInx];
    case Cell^.ldPrev of
      ldNorth :
        begin
          slWriteChange(F, aFromInx-1, aToInx);
          writeln(F, '<- ', FFromFile[aFromInx]);
        end;
      ldNorthWest :
        begin
          slWriteChange(F, aFromInx-1, aToInx-1);
          writeln(F, '   ', FFromFile[aFromInx]);
        end;
      ldWest :
        begin
          slWriteChange(F, aFromInx, aToInx-1);
          writeln(F, '-> ', FToFile[aToInx]);
        end;
    end;
  end;
end;
procedure TaaFileLCS.WriteChanges(const aFileName : string);
var
  F : System.Text;
begin
  System.Assign(F, aFileName);
  System.Rewrite(F);
  try
    slWriteChange(F, pred(FFromFile.Count),
      pred(FToFile.Count));
  finally
    System.Close(F);
  end;
end;
```

routine is called for cell (0,0); don't write anything to the file for this case. If the index into the to string is zero, we make the recursive call moving up the matrix (the index into the from string is decremented) and the action is taken to be the deletion of the current character in the from string. If the index into the from string is zero, we make the recursive call moving left through the matrix, and the action is inserting the current character into the to string. Finally, if both indexes are non-zero, we find the cell in the matrix and make the requisite recursive call, and write the action to the file. For a down move, it's a deletion; for a right move, it's an insertion; for a diagonal move, it's neither (the character is 'carried over'). For a deletion, we use a right facing arrow (<-); for an insertion, a left facing arrow (->); and for a carry over, nothing.

Here is the text file that was generated for converting *algorithms* into *alfresco*.

```
   a
   l
<- g
<- o
-> f
   r
<- i
<- t
<- h
<- m
-> e
   s
-> c
-> o
```

Pretty easy to understand at a glance! You can see the longest common subsequence (a, l, r, s), and you can easily identify the deletions and insertions.

And Text Files?

With all that under our belts, we can now attack the text file problem. If we assume that we can use `TStringLists` for both files, I'm sure that you can see that much of the code is really similar. Obviously we're now comparing whole text lines (strings) at a time, instead of characters, but the main algorithm remains the same and so it becomes a matter of cut and paste. There is one major gotcha, though: with strings we start counting the characters at 1, with a stringlist we start counting the strings (the lines in the original file) at zero. We must therefore make *some* changes.

The first change is that I didn't code the iterative method for calculating the LCS. If you recall, the iterative method required the 'zero' cells to be pre-calculated. That's fine for a couple of strings where characters are counted from one, but makes a whole mess when we're talking about stringlists when the 'zero' cells presumably become the 'minus one' cells. That means changes to the matrix class as well as everything else. Brrr, no thanks. So I only coded the recursive method.

The next change is that the top and left sides of the matrix are now 'virtual', since they use index -1. That's no problem for the recursive code; they're always assumed to not exist.

Listing 6 shows the new class that generates the LCS for a pair of files.

This is the point where we stop with this month's article. We've shown how to generate the longest common subsequence for a pair of files, and, even better, shown how to edit one file to produce the other. There are lots of possibilities and questions left for the adventurous reader, however. What if the files are very large? Is there a better structure for analyzing them than just loading them both into string lists? I've used a recursive method here, what are the ramifications regarding the stack? For large files, would there be too much recursion, blowing the stack? I've used a simple comparison test between lines in the files, what other more complex tests could we do? Stripping trailing spaces is one example. How could we create a 'patch' to go from one file to another?

Anyway, I hope you've enjoyed this article as much as I have. I've learnt a lot myself this time, I hope it was the same for you.

Julian Bucknall can be reached at julianb@turbopower.com The code that accompanies this article is freeware and can be used as-is in your own applications.