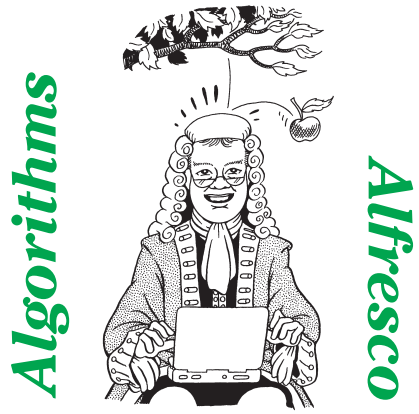
Push The Limits

This month: a collection of tips and tricks for performance and optimization



by Julian Bucknall

This morning my wife, Donna, and I went to a local café for breakfast. I was a bit uncommunicative and she asked me what was wrong. I explained the problem: I had to write this column, I was late, and I was fresh out of ideas for a good topic. After telling me off for being late (*Our Esteemed Editor* seems to have acquired Donna's email address, methinks), she started firing off a few ideas. Did I write Part 1 of a pair of articles a couple of months back and therefore should I write Part 2? The only thing I could come up with was Part 2 of my series on encryption, but I want to write that in England on my next trip, for obvious reasons. How about a set of algorithmic tips and tricks? I replied that I leave that to my Father Christmas articles.

However, that idea got me thinking. A couple of months back I wrote Chapter 1 of my Algorithms book. It was on performance issues: talking about the big-Oh notation, memory paging, using the cache, memory versus speed, and so on. Also, a couple of weeks or so ago, I spent a few days doing an intensive code review on one of TurboPower's new products (*Internet Professional* should be out by the time you read this: end of gratuitous plug!) and had culled a set of performance-related tips from that process.

So, bear with me, gentle reader, as I dedicate this article to exploring various performance tips and optimization tricks. You should find one or two nuggets here that will help you in your own development. I warn you that, this month, the code will only work with a 32-bit version of Delphi, but some results will also apply to Delphi 1.

Endless Quest

The first thing I should immediately state is that performing any kind of optimization process is impossible to do without measurement. You need to measure how your program runs, the speed with which it performs a particular operation. You need to take these measurement statistics and, applying your knowledge of Delphi, algorithms and data structures, alter the routine(s) to be faster (hopefully), and then measure the operation's speed again. If there was no improvement, or it was worse, you might as well ignore those changes. If there was an improvement, well, award yourself a pastry from the local bakery.

Notice the emphasis on measurement: you *cannot* know how good your optimization efforts are without measuring the before and after runtime characteristics. Simply plugging in another algorithm, just because someone (even Donald Knuth) says it's better, is simple delusion. It may well be that the attributes of your process mean that the supposed more efficient algorithm is anything but.

So, how do you measure the speed of a program, an operation, or a routine? The easiest answer is to use a profiler. There are two non-intrusive profilers for Delphi that I know of: Sleuth QA Suite from TurboPower and QTime from Automated QA, Inc. Don't rely on me to recommend one, I'm biased! See what other people say *[Both have been reviewed in Developers Review recently. Ed]*. What I do recommend is that you buy one and use it. You can get away with timing using Windows' `GetTickCount`, but it's intrusive (you have to alter your source code to perform timing operations) and it's not very accurate (both of the aforementioned products use special programming tricks to get accurate timing values). You could, if you were perverse enough, use a profiler that alters your source code to add profiling calls, but my experience with these has not been good.

I realize that all of my readers may not have access to a profiler, so I've written the code this month to use the results from `GetTickCount` calls to time operations. This is easy for me, since the routines I'll be presenting are all isolated and independent and hence lend themselves to being incorporated in a driver program for timing purposes. You'll see what I mean as I go into more depth. If you do not have access to a profiler, using this technique (that is, isolating the routine and writing a driver program to do the timing) is far better than altering your magnum opus by inserting calls to `GetTickCount`.

TNT For The Brain

What I would like to do now is to introduce the big-Oh notation. Instead of jumping right in and

➤ *Listing 1: Sequential search to find an item in a list.*

```
function SeqSearch(aStrs : PStringArray; aCount : integer;
  const aName : string5) : integer;
var
  i : integer;
begin
  for i := 0 to pred(aCount) do
    if CompareText(aStrs^[i], aName) = 0 then begin
      Result := i;
      Exit;
    end;
  Result := -1;
end;
```

Number Of Items	Time Taken
100	1.49
1,000	15.28
10,000	145.60
100,000	1449.34

➤ *Table 1: Timing a sequential search.*

throwing around big-Oh of this, big-Oh of that, let's start off with a simple example. You have a `TList` of customer records, sorted by last name, and you want to find Mr Smith in the list. A simple database-like query, in other words. As a first stab, you might try Listing 1.

This routine shows a simple sequential search through the list. You start at the beginning of the list, visit every item in it, and, for each item, compare the last name to `Smith`. Simple enough.

For this article, we'll do a little more than just testing that the routine works. We'll time it. What I did was to write a small driver program that creates a list with a varying number of items, sort it, obviously, and time how long it takes to find the required item. Table 1 shows the results for 100, 1,000, 10,000 and 100,000 items as run on my work machine. Even without any formal mathematical training, I'm sure you can see the relationship between the time taken for the routine to find an item and the number of items in the list. For ten times the number of items, it takes about ten times the time to find a particular item. The relationship is linear. In algebraic terms:

Time taken = k * number of items

Where k is some constant value that depends on the machine, the operating system, or the environment in which the program is running.

Using this relationship we can easily predict the time it would take for 500 or 5,000 items. If you have a scientific calculator with linear regression capabilities, it'll tell you what the value of k is for your machine, and therefore the time taken for a given number of items.

After checking back issues of *The Delphi Magazine*, you may have come across the article where I showed binary search. You decide to apply it and come up with Listing 2.

Binary search makes use of the fact that the list is sorted. You look at the middle item in the list. There are three possibilities: the item is the one you want (hooray, you found the item so you can stop straightaway), the item is less than the one you want, or the item is greater than the one you want. In the second case, you can immediately deduce that the item must be found in the latter half of the list. In the last case, you can deduce that the item must be in the initial half of the list. Either way, you can ignore half of the list and concentrate on the other half. Again you look at the middle item, but of the half of the list you've isolated. Again you either found the item or you can ignore half of this half of the list. You continue in this fashion, slicing and dicing the list, until you've got a sub-sub-sub-list with only one item that either is the one you want or it isn't. In the latter case, this shows that the item you were looking for isn't in the list.

Once more, after testing that the routine works and finds a given item, we time it (this is a simple rule: there's no point in timing something that doesn't work!). I profiled the binary search with the same 100, 1,000, 10,000 and 100,000 items. The timing results are shown in Table 2. The relationship between number of items and speed doesn't look as obvious as before. For ten times the number of items, the time taken for the routine to execute increases by a constant amount. What kind of relationship is this? Think back to your secondary school days when you learned about logarithms. To multiply a number by a factor, you would take the logarithm of the number and add the logarithm of the value, and then take the anti-logarithm of the result. (Am I showing my age by revealing that this is how I learned how to do long-winded calculations?) Anyway, this process gives us a hint that the relationship is a logarithmic one:

Time taken =
k * log(number of items)

Where, again, k is some constant or other determined from our runtime statistics. I deliberately didn't say which logarithmic base I'm using: it doesn't matter. It could be base e, or base 10, or base 2; the relationship is the same, the only difference being the value of the constant k.

With some minor speed testing we've determined some important results. Sequential search is a

➤ *Table 2: Timing binary search.*

Number Of Items	Time Taken
100	0.48
1,000	0.99
10,000	1.56
100,000	2.02

➤ *Listing Listing 2: Binary search to find an item in a list.*

```
function BinarySearch(aStrs : PStringArray; aCount : integer;
  const aName : string5) : integer;
var
  L, R, M : integer;
  CompareResult : integer;
begin
  L := 0;
  R := pred(aCount);
  while (L <= R) do begin
    M := (L + R) div 2;
    CompareResult := CompareText(aStrs^[M], aName);
    if (CompareResult = 0) then begin
      Result := M;
      Exit;
    end else if (CompareResult < 0) then
      L := M + 1
    else
      R := M - 1;
  end;
  Result := -1;
end;
```

linear process: the time taken is proportional to the number of items. Binary search, on the other hand, is a logarithmic process: the time taken is proportional to the log of the number of items.

This is getting entirely too long-winded, we need a succinct notation to describe these conclusions. Enter the big-Oh notation. We say that sequential search is O(n) and binary search is O(log(n)). We read the notation as 'for a sufficiently large value of n, the number of items, sequential search takes time proportional to n,' and, 'for a sufficiently large value of n, the number of items, sequential search takes time proportional to log(n).'

A quick note to all you computer science purists out there who are jumping up and down saying that what I've just said is a gross over-simplification. Yes, I know that I'm simplifying the actual definition of the big-Oh notation. My purpose in this article is not to overwhelm the reader with too much mathematical jargon, but to lay a foundation for understanding any big-Oh numbers they may come across in my articles, or anywhere else for that matter. In the same vein, I won't be describing Theta or Omega functions, or introducing the term *asymptotic*. And now we return you to our normal programming.

As you have seen, the big-Oh notation is a succinct way to describe the runtime characteristics of an algorithm or operation. Having introduced the notation, you should be able to read and understand the terms O(n^2), or O($n$log(n)), or any other big-Oh term that anyone throws at you.

Morphing Thru Time

Now, looking at the big-Oh values for binary and sequential search and knowing that the log of a positive integer is always less than the number itself (the proof being left as an exercise for the reader!), we could make the deduction that binary search is always faster than sequential search. Is this deduction valid? Is binary search always faster than sequential search?

This is one area where the big-Oh notation lets us down. The notation tells us what happens for *sufficiently large values of n*. For small values of n, the notation doesn't help; in fact, it doesn't tell us anything. For our example of sequential versus binary search, consider which one might be faster for one item, or three items. Binary search has more set up or initialization code than sequential search, which uses a fast `for` loop instead. For items where we are comparing strings for equality (a slow process) I would estimate that three items is the point at which using binary search takes over from sequential search. For a faster comparison operation, say comparing two integers, the break point might be larger. Only experiment will show us.

The reason for this uncertainty lies in the proportionality constants we're using. For example, by experimenting with two algorithms X and Y, we may deduce the following timing formulae:

$$\text{Time for X} = 100 * n$$

$$\text{Time for Y} = 1000 * \ln(n) + 100$$

In other words, we can immediately say that X is O(n) and Y is O(log(n)).

So, which is faster? We may initially just go with Y since log(n)<n, but that is only half the story. If n=1, then X and Y both take 100 units of time. For n=2, X takes 200 units, Y 793 units. For n=10, X takes 1000 units, Y 2402 units. Eventually, with n=38, X takes 3800 units and Y 3787 units, the first time that X takes longer than Y. Thereafter, for increasing n, Y is always faster. So, you see the big-Oh notation does not obviate speed testing. In this admittedly contrived example, we may find that we never have more than 30 items to run through algorithm X or Y, so we should go with X, even though Y has a 'better' big-Oh value.

So far, I've just been talking about *time efficiency*; that is, how fast a routine is, how quickly it runs for different amounts of input. We've seen that the big-Oh notation tells us succinctly how efficiently a routine runs. There is another set of measurements we can make: *space efficiency*, or how much memory a routine consumes. The big-Oh notation can be used for this as well. If we say that a routine consumes O(n) of heap space, we mean that, for sufficiently large n, the routine requires an amount of space on the heap that is proportional to n. Lots of routines we use only require O(1) of space, that is, a constant amount of memory. A simple example should make this clear, that doesn't require me to write any code to explain it (lazy, *moi*?).

Suppose you were writing a file copy routine. There are two main algorithms you could use. The first one requires O(n) extra memory: allocate a buffer big enough to read the entire source file into memory, do so, and then write it out to the destination file. All I can say about this one is it gives me the shivers. Brrr. The second algorithm is the one people normally implement: allocate a buffer of some predetermined size (say, 4Kb), and enter a loop where you read 4Kb from the source file and write this buffer-full to the destination file. Continue cycling round this loop until the source file is exhausted. This is a routine with O(1) space requirements, and, I'm sure you will agree, *much* more manageable.

Out From The Deep

Many times, algorithm writers introduce the notion that different algorithms to perform the same job have *space versus time* trade-offs. A faster algorithm may require more memory than a slower one. This is by no means a universal law, but it does crop up often enough to merit discussion.

Suppose you wanted a routine to calculate the number of days in a particular month of a given year. You remember the nursery rhyme from your childhood: '30 days hath September, April, June and November, ...' and you know the rule for determining whether a year is a leap year or not, so you code Listing 3. It's a simple enough routine (the only difficult bit really is the determination of leap year).

It takes 6.3 seconds on my machine to do 168 million iterations. Pretty fast, I'm sure you'll agree.

Those of my readers who've been reading these articles for a while will know a better way of doing this calculation. You define an array of integers to hold the numbers of days in each month, and then index into that. The only 'difficult' case is again that of February. Listing 4 shows this routine. It takes 4.6 seconds on my machine, a saving of 27%. Not to be sneezed at, especially for such a self-contained routine. And the cost for this saving is a small array of integers.

This is a small example of what is meant by a space versus time trade-off. The faster algorithm requires an array of pre-calculated values in order to speed things up. The slower algorithm does not require this array. Now it might be argued that the price to pay for the faster routine and its measly extra 48 bytes of constant values is well worth it. But this is just one routine out of many thousands in a typical application. If many of these routines *also* had constant arrays in order to speed them up, the application as a whole would be much larger. It would take longer to load the program and present the initial window to the user. This problem is another aspect of the time versus space trade-off: for the sake of faster routines during runtime, the application takes longer to load and start up.

Beyond The Invisible

Up to now, we've been very theoretical about optimization and performance issues. Let's be a little more concrete and discuss a few optimization tricks specifically for Delphi. We'll use my favorite type: the long string. Favorite in a cynical sense, of course: although the type was invented for making our lives easier, it does come with a *lot* of performance disabling baggage.

Long strings are a boon to the Delphi developer: they can be very long indeed, they're reference counted so string assignments take virtually no time, the compiler makes sure that when we alter a string we have our own copy, and it also makes sure to free local strings from the heap when necessary. However, you should be aware of what goes on underneath.

Let's take the first one: when writing a routine, should you use `const` with long string parameters? In the old days, with Delphi 1 and Borland Pascal 7 and earlier, you learnt that using `const` for a string parameter was A Good Thing. It avoided a compiler-generated copy of the string being passed onto the stack. (Since you had not declared the string parameter as `const`, the compiler would assume that you were going to change the string inside the routine and therefore would copy the string to avoid any changes to the caller's string.) In 32-bit Delphi, the long string is a pointer to a specially formatted block of memory on the heap; therefore, since a pointer is the size of a register in the CPU, so the reasoning goes, you don't need to use `const`.

In effect, the compiler makes the same assumption about the string parameter as in the old days. However, this time it doesn't make a copy of the string (excellent!) but instead it increments the string's reference count (even better!). At the end of the routine, it decrements the string's count as we leave the routine. All in all it seems like the ideal situation: no copying, and a mere couple of twiddles with the reference count. In reality, though, the situation is different.

Let's use a simple routine that counts the alphabetic characters in a string (no, I don't know what you'd use this for, but it makes a great illustration of what we're discussing). Listing 5 shows the routine declared with a non-`const` string parameter and also a version with a `const` parameter. The only difference, as you can see, is the keyword `const`. We'll create a test bed driver program and time both routines. On my machine, the `const` version takes 4.4 seconds whereas the other 5.0 seconds for 10 million iterations. Why such a big difference of 12%?

The reason is that the compiler implements a `try..finally` block for the first routine, the non-`const` one. It declares a hidden local string variable, sets it equal to the passed in string, and increments the reference count for the pointed-to string. It then sets up a `try` block before going into the code we actually wrote. At the end of the routine, a `finally` block is created to clear the automatically-created local string variable, which decrements the reference count for the string on the heap. An awful lot of new code is inserted for us, just because we could not

➤ *Listing 3: Calculating the number of days in a month.*

```
function CalcDaysInMonth1(Month, Year : integer) : integer;
begin
  if (Month = 4) or (Month = 6) or (Month = 9) or (Month = 11) then
    Result := 30
  else if (Month = 2) then begin
    if (((Year div 4) = 0) and
        (((Year div 100) <> 0) or ((Year div 400) = 0))) then
      Result := 29
    else
      Result := 28
  end else
    Result := 31;
end;
```

➤ *Listing 4: Calculating the number of days in a month faster.*

```
function CalcDaysInMonth2(Month, Year : integer) : integer;
const
  DaysInMonth : array [1..12] of integer =
    (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
begin
  Result := DaysInMonth[Month];
  if (Month = 2) then begin
    if (((Year div 4) = 0) and
        (((Year div 100) <> 0) or ((Year div 400) = 0))) then
      Result := 29;
  end;
end;
```

be bothered to declare the string parameter as `const`. So, the rule is, if you pass a string into a routine, and nothing in that routine alters it, declare the string as `const`.

Next up for discussion is a trick where again the compiler is oh-so-obliging, but which results in a less-than-efficient bit of code. Many times we may try to find a character in a long string by coding something like this:

```
PosOfCh := Pos(Ch, St);
```

In other words, `Ch` is the character for which we're looking in the string `St`. Looks innocent enough?

Again the compiler works overtime behind our back. The declaration for `Pos` in the `System` unit has the first parameter as a string. The compiler notices that we're passing a single character instead, so what does it do? That's right, it declares another hidden local string variable, calls a special routine in the `System` unit that allocates a single character string on the heap (`_LStrFromChar`), and then calls `Pos`. Of course, this string must be deallocated at some time, and so another `try..finally` block comes into play. Yuk.

```
function CountAlpha1(S : string) : integer;
var
  i : integer;
begin
  Result := 0;
  for i := 1 to length(S) do
    if (S[i] in ['A'..'Z','a'..'z']) then
      inc(Result);
end;
function CountAlpha2(const S : string) : integer;
var
  i : integer;
begin
  Result := 0;
  for i := 1 to length(S) do
    if (S[i] in ['A'..'Z','a'..'z']) then
      inc(Result);
end;
```

➤ *Listing 5: Two routines to count alphabetic characters in a string.*

```
function PosCh(aCh : char; const S : string; aStart : integer) : integer;
var
  i : integer;
begin
  if (aStart < 1) then
    aStart := 1;
  for i := aStart to length(S) do
    if (S[i] = aCh) then begin
      Result := i;
      Exit;
    end;
  Result := 0;
end;
```

➤ *Listing 6: Fast routine to find a character in a string.*

I timed the difference between the normal way of calculating the position of a character in a string with `Pos` with a simple Pascal routine that searches for the character in a loop and it was over five times slower. Listing 6 shows the simple Pascal routine that outdoes `Pos` in this situation. Not only is it faster, but this `PosCh` routine is even more functional than `Pos`: you can decide where in the string to start the search. I recommend that you use this `PosCh` function instead of `Pos` for the times when you are looking for a character within a string.

Another, similar, case is using mixed string types with `Pos`. Searching for a short string in a long string will require a temporary long string being allocated. Indeed, this becomes another steadfast rule: don't mix string types. If you are going to use long strings, don't use short strings, and vice-versa. Although your code will work, it will be less than efficient. (When I was writing FlashFiler, TurboPower's database engine, all those years ago, I wrote a whole series of short string routines, trimming, path manipulation and the like, so that I was sure that I wouldn't inadvertently call a routine that would convert a short string to a long string and vice versa.)

➤ *Listing 7: Replacing escape sequences in a string.*

```
function ConvertEscapes1(const aSt : string) : string;
var
  i : integer;
  PosEscape: integer;
  ASCIIStr : string;
  ASCIIVal : integer;
  ec       : integer;
begin
  Result := aSt;
  i := 1;
  while i <= length(Result) do begin
    {find the next escape character in the remaining string}
    PosEscape := PosCh('\', Result, i);
    {if there is no escape, exit}
    if (PosEscape = 0) then
      Exit;
    {if this position is right at the end of the string, exit, we're done}
    if PosEscape = length(Result) then
      Exit;
    {if the next character is an backslash, then replace the
      double backslash by just one of them}
    if (Result[PosEscape+1] = '\') then
      Delete(Result, PosEscape, 1)
    else if (Result[PosEscape+1] in ['0'..'9']) and
      (PosEscape <= length(Result) - 3) then begin
      {if the next character is a digit, there should be three of them, convert
       the four characters \nnn to an ASCII character, ignore all errors (ie,
       don't convert the backslash)}
      ASCIIStr := Copy(Result, PosEscape+1, 3);
      Val(ASCIIStr, ASCIIVal, ec);
      if (ec = 0) and (ASCIIVal <= 255) then begin
        Delete(Result, PosEscape, 4);
        Insert(char(ASCIIVal), Result, PosEscape);
      end;
    end;
    i := PosEscape + 1;
  end;
end;
```

Second Chapter

What's next? Exposing some evil `System` string routines, that's what. Listing 7 shows a small routine that performs a specific type of search and replace operation on a string.

```
function ConvertEscapes(const aSt : string) : string;
var
  DestInx   : integer;
  SourceInx : integer;
  ASCIIStr  : string;
  ASCIIVal  : integer;
  ec        : integer;
begin
  {assume that we won't convert any escapes: the result
    string will be the same length as the source}
  SetLength(Result, length(aSt));
  {go through the source, character by character}
  DestInx := 0;
  SourceInx := 1;
  while SourceInx <= length(aSt) do begin
    {non-escape characters pass straight through}
    if (aSt[SourceInx] <> '\') then begin
      inc(DestInx);
      Result[DestInx] := aSt[SourceInx];
      inc(SourceInx);
    end else begin
      {otherwise it's an escape character}
      {if the escape is at the end of source string, pass
       it through: there cannot be any other characters
       to convert}
      if (SourceInx = length(aSt)) then begin
        inc(DestInx);
        Result[DestInx] := '\';
        inc(SourceInx);
      end else if (aSt[SourceInx+1] = '\') then begin
        {if it's the first of a double escape, pass a single
         one through to the result string}
        inc(DestInx);
        Result[DestInx] := '\';
        inc(SourceInx, 2);
      end else if (aSt[SourceInx+1] in ['0'..'9']) and
        (SourceInx <= length(aSt) - 3) then begin
        ASCIIStr := Copy(aSt, SourceInx+1, 3);
        Val(ASCIIStr, ASCIIVal, ec);
        if (ec = 0) and (ASCIIVal <= 255) then begin
          inc(DestInx);
          Result[DestInx] := char(ASCIIVal);
          inc(SourceInx, 4);
        end;
      end else begin
        {otherwise it *is* an escape character, but part
         of a badly formed sequence: just pass it through}
        inc(DestInx);
        Result[DestInx] := '\';
        inc(SourceInx);
      end;
    end;
  end;
  {finally set the correct length of the result: DestInx is
    the index of the last character written}
  SetLength(Result, DestInx);
end;
```

➤ *Listing 8. A better way to replace escape sequences in a string.*

We are searching in the source string for all occurrences of character sequences of the form *nnn* and replacing them with the ASCII character defined by *nnn*. A \\ sequence is replaced by a single \. This is a fairly typical routine for some types of application where you want the user to be able to enter characters that are not on his or her keyboard. The sequences starting with a \ are known as *escape sequences*.

The routine in Listing 7 works by copying the input string to the result and then finding and replacing the escape sequences. The replacements are done by deleting the sequence with `Delete` and inserting the replacement character with `Insert`. The routine works, but, boy, is it inefficient. We seem to have taken to heart the previous recommendations: the input string is a `const` parameter, we're using the proper search-for-a-character function. For a million repetitions on my machine it takes 3.33 seconds.

```
function UniqueChars1(const aSt : string) : string;
var i : integer;
begin
  Result := '';
  for i := 1 to length(aSt) do begin
    if (PosCh(aSt[i], Result, 1) = 0) then
      Result := Result + aSt[i];
  end;
end;
function UniqueChars(const aSt : string) : string;
var
  i  : integer;
  Ch : char;
  DestInx : integer;
  CharSet : set of char;
begin
  {clear the set of found characters}
  FillChar(CharSet, sizeof(CharSet), 0);
  {find the unique characters in the input string}
  for i := 1 to length(aSt) do
    Include(CharSet, aSt[i]);
  {store the unique characters in the Result string}
  SetLength(Result, length(aSt));
  DestInx := 0;
  for Ch := #0 to #255 do
    if (Ch in CharSet) then begin
      inc(DestInx);
      Result[DestInx] := Ch;
    end;
  {readjust the length of the result string}
  SetLength(Result, DestInx);
end;
```

➤ *Listing 9: Two ways to find the unique characters in a string.*

The problem, though, is the calls to `Delete` and `Insert`. Each of them causes the result string to be reallocated on the heap. For a lengthy input string with lots of escape sequences, these two routines would be called an awful lot. There would be string reallocations galore. Indeed, the number of reallocations would be proportional to the number of escape sequences in the original string: the function is O(*n*) with respect to memory allocations, in other words.

Better would be to use Listing 8. This routine specifically makes sure that there are only two allocations for the result string. First, we assume that there would be no escape sequence replacements at all. We assume that the result string will be equal in length to the output string. Then, we transfer characters one by one from the source string to the result string. If we encounter an escape character, we convert the sequence it introduces and transfer the resulting character to the result string. Finally, we adjust the length of the result string to comply with the number of characters we transferred. Two memory allocations, no matter how many escape sequences in the original string, is an O(1) algorithm. Consequently, it should come as no surprise that Listing 8 takes just under 1 second for a million repetitions: three times as fast.

Note that in Listing 8 I am cheating a little: I'm using a long string to hold the *nnn* part of an ASCII escape sequence, and then converting from that. This is of course a memory allocation. If I were writing this function for real, I would of course use a different methodology to convert the *nnn* sequences. My reason for this example is to warn against the `Delete` and `Insert` procedures from the `System` unit, especially in loops like this.

In general, for functions of this type where we are generating a result long string from some kind of input, it is better to overestimate the length of the result string and then re-adjust at the end, than to incur a series of memory reallocations throughout the routine. Listing 9 drives this point home with a routine that, given an input string, returns a string containing just the unique characters in the input string. The first implementation uses the string + concatenation operator to build up the string. The second estimates that the result string is going to be at least the length of the input string, and then readjusts at the end. Both routines have to read through the entire string, an O(*n*) process. The first routine is highly dependent on the number of unique characters: the search with `PosCh` is an O(*n*) routine, where *n* is the number of unique characters. The second routine, on the other hand, is an O(1) routine as far as searching for unique characters: we're using a set of characters to hold the unique ones found. The generation of the result string is O(*n*) in the first case (we're concatenating characters one by one on the end of the result string, which results in some memory reallocations). In the second case, we only have two memory allocations, as discussed before. Experimental evidence backs this up. If we are processing an input string that just consists of repetitions of *a* and *b*, then the first routine is faster than the second. If, however, the input string consists of mainly different characters, with only a few repetitions, the second routine is faster. If you had need of a function that performed this operation, you would have to analyse whether your input was strings with lots of duplicates or not many duplicates and use the appropriate function.

With this routine we have reinforced what I was saying earlier: although big-Oh values tell us something about a routine, they don't tell us the whole story. We must experiment and measure the effect different routines have in *our* applications, not just take as gospel what some algorithm hack columnist says. If you come away with anything from this article, I'll have done my job if you understand the big-Oh notation and its drawbacks, and the importance of profiling your code.

Mea Culpa

Unfortunately, I didn't get round to evaluating your responses to my simulated annealing contest as promised last month. Sorry about that. I'll announce the results next month instead.

Julian Bucknall is enigmatic. Even worse, he seems to be late with everything, this article, his book, his work, *sigh*. Julian can be reached eventually at julianb@turbopower.com. The code that accompanies this article is freeware and can be used as-is in your own applications.