# The Holly And The Ivy

## *A stocking full of algorithms*

Algorithms Alfresco

by Julian Bucknall

I'd dropped my wife Donna off at the north entrance of the Air Force Academy at the trailhead for the Santa Fe Trail and had driven down south to the Woodmen trailhead to pick her up. We have this scheme for her marathon training: she runs down the trail from north to south (about 12 miles), and I drive to the next trailhead, get on my mountain bike, and cycle to meet her. That way she gets to run something continuously without having to stop and double back, and I get to really ride at speed up the trail until we meet. After that, I coast beside her back to the car *[So, Julian, when you complained in those emails about how Donna always beat you back to the car you were just kidding? Ed]*.

And riding at speed in the brisk air on that trail was glorious. The leaves were turning, the aspens were now their world-famous golden color, and I was filled with a sense of being alive. The trail winds its way through some wonderful unspoilt countryside on the Academy land and the hustle and bustle of Colorado Springs seemed miles away.

That late afternoon there were practically no people on the trail and I'd picked up a nice rhythm, speeding down hills, switching gears on the upslope to get the power to reach the next summit. I rounded a bend, leaning into it at just the right angle, and had to slide suddenly to a halt: a rather corpulent man in red stood with his back to me blocking the path. He turned at the noise.

"How are you, dear chap?" Father Christmas asked.

I watched some pebbles dislodged by my sudden stop fall over the edge of the path and tumble thirty feet down into Monument Creek.

"Oh, fine. A little breathless, perhaps. It's not often I go round that corner to find someone blocking the way." I got up and dusted myself off. The bike was fine; I propped it up against a spruce and sat down on the grass. With a wheeze, Father Christmas sat next to me.

"I don't know how you can ride that thing," he said. "Much more comfortable in a sleigh." He nodded over to where a sleigh was parked incongruously in the trees, with the reindeer introducing themselves to some local deer. We exchanged a few pleasantries, catching up after another year of not seeing one another.

"I dare say since it is autumn, you're here to collect some small algorithms for Christmas again?" I asked, having regained my breath and enjoying the sit-down.

I should explain to my new readers. A while back, the original Father Christmas decided there was too much work for one giver to do, so he cloned himself and set up a variety of donor markets with a different Father Christmas looking after each. The Father Christmas I'd nearly crashed into was the Father Christmas of Programmers. He's more interested in developing software for the new e-Christmas venture than in wandering around leaving festive algorithms on programmers' hard drives as presents, and so he comes to me. I write a few algorithms up in an article for the December issue and he gets to stay in on Christmas Eve and puzzle over some JavaScript.

I kept the thought to myself that maybe he should get out and do more and have some exercise instead of programming in JavaScript (after all, I like presents at Christmas too) and asked him whether he'd got any ideas to start us off.

"Actually, there was something I wanted to discuss that stemmed from one of your articles," he started off. "Not really an algorithm, but I think your readers would be interested." It turned out that he was talking about the article I'd written on heap debugging (this appeared in Issue 62). "I found the information interesting and I was able to clean up some memory problems in an old Delphi app, but there was something else happening in this code that required me to find out the creation order of various objects."

"So, put some debugging code in the object's constructor to write to a log file," I said.

"But the problem was that I didn't have the source code to some of the classes."

I pondered. It was a tricky problem indeed. Suppose you have a black box library that uses objects all over the place that were interacting with your objects: how can you work out what was getting created and when? You can't access the constructor to *every* class whose creation you want to track: it's too unwieldy, especially when you don't have the code.

"It seems like an ideal application of the replacement heap manager stuff you were talking about in the article," he added, "but I cannot for the life of me work out from where the `GetMem` routine is being called."

I agreed with him. It seemed like an ideal use of a heap manager replacement, but all `GetMem` gets called with is a size parameter. We don't get a flag parameter to say, "this is for an object," or "this is for a long string." I needed to experiment so I asked for his notebook. He fiddled in his computer bag and brought out this curvy, ice blue, pop art notebook like nothing I'd seen before except in Apple designers' dreams. It was an

instant-on machine as well and without waiting I had Delphi 5 ready to rock.

Some time later, after some poking around with the CPU view, I had my answer. In the initialization section of my heap manager unit, I stored the addresses of `TObject.NewInstance`, `TObject.FreeInstance` and `TObject.Create`. Since these are written in that order in the `System` unit, with no routines in between them, if I have a return address between `TObject.NewInstance` and `TObject.FreeInstance` then it must be within the code for `NewInstance`. Similarly, I can detect a return address within `TObject.FreeInstance`. (Luckily, Father Christmas had a complete collection of Delphi versions and I can confirm that this is true for Delphi 3, 4, and 5.)

Now the clever stuff. My heap manager's `GetMem` gets called, and the first thing I do is to capture the base pointer value (EBP). Using the CPU view of the debugger I made a note of the values on the stack when a constructor called my routine. There were four 4-byte values of interest pointed to by EBP. The first was a `longint`: the value of the caller's EBP. The second was a pointer: the return address of the `System` unit's `_GetMem` routine. The third was another pointer: the return address within the `InitInstance` routine. The fourth and last value was another pointer: the class's VMT.

Of course, if `GetMem` were called by some other code, there would still be at least four 4-byte values there on the stack. What I needed to do was to work out whether the third 4-byte value was within `TObject.NewInstance`. If so, then it was an object constructor doing the memory allocation. At that point I could take the class pointer, typecast it to a `TObject` and call the `ClassName` method to get the class name and write it to a log. This would obviously not work for those classes that override their `NewInstance` method, but those tend to be few and far between. (In Delphi 5, there is only one class that overrides `NewInstance` and that's `TInterfacedObject`. Luckily, this overridden method calls the inherited `NewInstance`, the one we're looking for, and so this trick will still work since the stack looks the same.)

I also worked out the same kind of trick for an object's destructor. Here EBP points to three 4-byte values: the caller's EBP, the return address of `System`'s `_FreeMem` routine, and the return address of the `FreeInstance` method. I could easily check to see whether the `FreeInstance` return address was valid, in which case I could typecast the pointer I was freeing as a `TObject` and call its `ClassName` method again.

I typed away for a while, crashed my test app a few times until I got the interface to the call stack right, and produced Listing 1. Father Christmas nodded happily. "Brilliant, dear chap. It's amazing how you can work your way around the CPU view; it scares the living

➤ *Listing 1: Logging object creation and destruction.*

```
var
  Log : System.Text;
  OrigHeap : TMemoryManager;
  OurHeap  : TMemoryManager;
  NewInstAddr  : longint;
  FreeInstAddr : longint;
  CreateAddr   : longint;
type
  PNewInstCallStack = ^TNewInstCallStack;
  TNewInstCallStack = record
    csOldEBP        : longint;
    csGetMemRetAddr : longint; // actually a pointer
    csNewInstRetAddr: longint; // actually a pointer
    csClassInstance : TClass;
  end;
  PFreeInstCallStack = ^TFreeInstCallStack;
  TFreeInstCallStack = record
    csOldEBP         : longint;
    csFreeMemRetAddr : longint; //pointer;
    csFreeInstRetAddr: longint; //pointer;
  end;
function OurGetMem(Size: Integer): Pointer;
var
  CallStack : PNewInstCallStack;
  PtrString : array [0..8] of char;
begin
  {get the call stack}
  asm
    mov CallStack, ebp
  end;
  {allocate the memory}
  Result := OrigHeap.GetMem(Size);
  {if this was called from TObject.NewInstance, output a
    line to the log showing the object details}
  if (NewInstAddr <= CallStack^.csNewInstRetAddr) and
     (CallStack^.csNewInstRetAddr < FreeInstAddr) then begin
    PointerAsHex(PtrString, Result);
    writeln(Log, 'New:  ', PtrString, ' ', Size:10, ' ',
      CallStack^.csClassInstance.ClassName);
  end;
end;
function OurFreeMem(P : Pointer) : integer;
type
  PClass = ^TClass;
var
  CallStack : PFreeInstCallStack;
  ClassPtr  : PClass;
  PtrString : array [0..8] of char;
begin
  {get the call stack}
  asm
    mov CallStack, ebp
  end;
  {if this was called from TObject.FreeInstance, output a
    line to the log showing the object details. Note this
    only works because the first field of the object being
    freed is the class pointer}
  if (FreeInstAddr <= CallStack^.csFreeInstRetAddr) and
     (CallStack^.csFreeInstRetAddr < CreateAddr) then begin
    PointerAsHex(PtrString, P);
    ClassPtr := P;
    writeln(Log, 'Free: ', PtrString, ' ', ' ':10, ' ',
      ClassPtr^.ClassName);
  end;
  {free the memory}
  Result := OrigHeap.FreeMem(P);
end;
procedure InitializeUnit;
begin
  {get the addresses of NewInstance,
   FreeInstance and Create as integers}
  NewInstAddr := longint(@TObject.NewInstance);
  FreeInstAddr := longint(@TObject.FreeInstance);
  CreateAddr := longint(@TObject.Create);
  {open up the log file}
  System.Assign(Log, 'C:\AAClass.LOG');
  System.Rewrite(Log);
  writeln(Log,
    'Algorithms Alfresco Object Creation/Destruction Log');
  writeln(Log);
  writeln(Log, 'Type   Address       Size Class');
  {get the original manager}
  GetMemoryManager(OrigHeap);
  {set up our heap manager}
  OurHeap.GetMem := OurGetMem;
  OurHeap.FreeMem := OurFreeMem;
  OurHeap.ReallocMem := OrigHeap.ReallocMem;
  {replace heap manager with ours}
  SetMemoryManager(OurHeap);
end;
procedure FinalizeUnit;
begin
  {restore the original manager}
  SetMemoryManager(OrigHeap);
  {close the log}
  writeln(Log, '..finished..');
  System.Close(Log);
end;
```

daylights out of me." I responded that sometimes the CPU view was the only thing that would save your bacon and it was worth getting to know.

"Whilst playing around with this code, I remembered an email I'd received about the heap article.

"Someone who'd read the piece had a problem with memory fragmentation and he wanted to know whether I had any ideas about solving it. I thought about it for a while, and suggested a couple of avenues to explore.

"The first idea was to look at the heap manager that calculated distributions that I provided as part of the heap manager article. This would give you an idea of what size of allocations you were making and how many of them. For a typical application, the number of small allocations far exceeds that for the larger allocations. All those form objects, button objects, edit control objects and so on are typically held in fairly small memory blocks. Borland have obviously done their homework with the standard memory manager: it's biased towards small allocations. However, the standard manager does try and coalesce freed blocks together.

"So, the first plan would be to create free lists for small allocations. How many? Well, I don't see why we should ignore Borland's research: let's use a free list for each allocation up to 4Kb; that's what they do in theirs. The difference is that we won't be coalescing freed blocks.

"Now that, in its turn, will produce problems. We may (for example) allocate 1,000 blocks of 122 bytes, free them all and never allocate another: we're left with this long free list of unusable memory. So, ideally, we should look at the granularity of our allocations so that we have less free lists. Delphi's heap uses granularity of 4 bytes. Let's change that to a granularity of 32 bytes. Before you roll your eyes at me, consider

➤ *Listing 2: Trying to alleviate heap fragmentation.*

```
type
  PFreeNode = ^TFreeNode;
  TFreeNode = packed record
    fnNext : PFreeNode;
  end;
const
  MinFreeInx = 0;
  MaxFreeInx = (4096 div 32) - 1;
var
  OrigHeap : TMemoryManager;
  OurHeap  : TMemoryManager;
  {the free lists for blocks of size 32 to 4096}
  FreeList : array [MinFreeInx..MaxFreeInx] of pointer;
function OurGetMem(Size : integer) : pointer;
var
  Inx : integer;
begin
  {make a decision based on the size: if it's less than
or
    equal to 4096 we can get the allocation from our free
    lists...}
  if (Size <= 4096) then begin
    {round up to the nearest 32 bytes}
    Size := (Size + 31) and not integer(31);
    {if there is a node free on the relevant free list,
     use it}
    Inx := pred(Size div 32);
    if (FreeList[Inx] <> nil) then begin
      Result := FreeList[Inx];
      FreeList[Inx] := PFreeNode(Result)^.fnNext;
    end
    {otherwise allocate from Delphi's heap manager}
    else
      Result := OrigHeap.GetMem(Size);
  end
  {otherwise the size is too great for our linked lists,
    round up to nearest 1KB and then allocate it from
    Delphi's heap manager}
  else begin
    Size := (Size + 1023) and not integer(1023);
    Result := OrigHeap.GetMem(Size);
  end;
end;
function OurFreeMem(P : pointer) : integer;
type
  PInteger = ^integer;
var
  Size : integer;
  Inx  : integer;
begin
  {make a decision based on the block's size: if it's
less
    than or equal to 4096 we can store it on our free
    lists...}
  Size :=
    PInteger(PChar(P) - sizeof(integer))^ -
sizeof(integer);
  if (Size <= 4096) then begin
    Inx := pred(Size div 32);
    PFreeNode(P)^.fnNext := FreeList[Inx];
    FreeList[Inx] := PFreeNode(P);
    Result := 0; {no error}
  end
  {otherwise just free it with the original heap manager}
  else
    Result := OrigHeap.FreeMem(P);
end;
function OurReallocMem(P : pointer; Size : integer) :
  pointer;
var
  OldSize : integer;
begin
  {Realloc is complicated: we need to trap reallocations
    using our free lists. Realloc can be called with 4
    possibilities:
     P = nil, Size = 0: return nil
     P = nil, Size > 0: equivalent to GetMem(Size),
       return new block
     P <> nil, Size = 0: equivalent to FreeMem(Size),
       return nil
     P <> nil, Size > 0: equivalent to GetMem(Size),
       copy old data to new block, FreeMem(P), return new
       block }
  if (P = nil) then begin
    if (Size <> 0) then
      Result := OurGetMem(Size)
    else
      Result := nil;
  end else begin
    if (Size = 0) then begin
      OurFreeMem(P);
      Result := nil;
    end else begin
      Result := OurGetMem(Size);
      OldSize := PInteger(PChar(P) - sizeof(integer))^ -
        sizeof(integer);
      if (OldSize <= Size) then
        Move(P^, Result^, OldSize)
      else
        Move(P^, Result^, Size);
      OurFreeMem(P);
    end;
  end;
end;
procedure InitializeUnit;
begin
  {initialize the freelists}
  FillChar(FreeList, sizeof(FreeList), 0);
  {get the original manager}
  GetMemoryManager(OrigHeap);
  {set up our heap manager}
  OurHeap.GetMem := OurGetMem;
  OurHeap.FreeMem := OurFreeMem;
  OurHeap.ReallocMem := OurReallocMem;
  {replace heap manager with ours}
  SetMemoryManager(OurHeap);
end;
procedure FinalizeUnit;
var
  i    : integer;
  P    : PFreeNode;
  Temp : PFreeNode;
begin
  {free all blocks on the free lists}
  for i := MinFreeInx to MaxFreeInx do begin
    P := FreeList[i];
    while (P <> nil) do begin
      Temp := P;
      P := P^.fnNext;
      OrigHeap.FreeMem(Temp);
    end;
  end;
```

string manipulations. With strings, we may have lots of reallocations going on as we build up string text. Having a larger granularity will help in this and make that type of code much faster.

"So now we have 32-byte granularity and we have 128 free lists for the freed blocks of size 32, 64, 96 and so on bytes. For the allocations that are greater than 4Kb we still have a problem. The answer is to make the granularity much larger for those big blocks. We'll be brave here and allocate memory with a 1Kb granularity." He raised his eyebrows at this.

"Think of it like this. For those large allocations, typically, we're reading bitmaps into memory and then freeing them. Or we are allocating a big buffer to help read a file in big chunks. We could call these one-shot allocations, if you like. Having certain specific memory block sizes would result in more memory reuse than when we had a 4-byte granularity: it's more likely we could reuse a memory block.

"Another use of big allocations is to create a big `TList` or some other expandable monolithic data structure. We could call these machine-gun allocations: bang it's this size, bang it's another, bang a third, rat-tat-a-tat. Having this kind of large granularity will help reduce fragmentation due to continually growing `TLists`. It won't remove it completely, but it'll do for a first stab. Now, with this scheme we would then use up more memory more quickly, but to quantify the effect would need some experimentation. Maybe a smaller granularity would be better for certain applications."

I was coding as I spoke. The 128 free lists would be simple stacks using a linked list of freed blocks. If a particular free list was empty, I coded it such that the original memory manager was called. The memory blocks over 4Kb were even easier to code: all allocations went through the original memory manager and hence I used the original free list code in that. My first attempt leaked memory like crazy, but between us we nailed that one down and TurboPower's Sleuth Codewatch gave us a clean bill of health. Listing 2 was the result.

We looked out over the Front Range mountains for a while and thought of other algorithms. I came up with one first.

"I was writing a test program the other day for which I had to produce a series of random numbers. These numbers weren't in a uniform distribution; instead it was as if I was trying to simulate a loaded die. For an example, suppose that I had to produce a 1 with probability 0.1, a 2 with probability 0.2, a 3 with probability 0.3 and a 4 with probability 0.2, and a 5 or a 6 with probability 0.1. If you add all those probabilities up you'll get a total of 1.0, indicating that there are no other possibilities. The virtual die I'm then simulating is weighted towards 2s, 3s, or 4s and in fact, funnily enough, the probabilities are known as *weights*. How do I go about generating a stream of numbers that satisfy this distribution?"

Father Christmas thought about it for a long moment. "OK, how about this? Generate a number from 1 to 10. If the number is 1, we output a 1. If the number is 2 or 3, we output a 2. If the number is 4, 5 or 6, we output a 3; for a 7 or 8, a 4 for a 9 a 5; for a 10 a 6. That gives the right probabilities." I nodded: it was a good answer and writing the code to implement it would be trivial.

"All right," I said. "Now suppose you want to write a random number generator that generates numbers from 1 to *n* according to an array of *n* weights that you supply to the generator? How would you do that?"

This time he took the notebook from me and started typing. After a couple of false starts, he suddenly smacked his head and said "Doh, it's easy!" He explained what to do. "We need to take the weights array and calculate a cumulative distribution for them in another array. The first data point is the probability for 1. The second data point is the sum of the probabilities for 1 and 2, the third the sum for 1, 2, and 3, and so on. The last data point, although it may not calculate to that exactly, should be 1.0. These cumulative

➤ *Listing 3: Generating random numbers according to their weights.*

```
type
  TaaRandomWeightedGenerator = class
    private
      FCount   : integer;
      FWeights : array of double;
    protected
    public
      constructor Create(const aWeights : array of double);
      destructor Destroy; override;
      function Get : integer;
  end;
constructor TaaRandomWeightedGenerator.Create(
  const aWeights : array of double);
var
  i : integer;
  TotalWeight : double;
begin
  inherited Create;
  Assert(length(aWeights) <> 0,
    'An array of weights must be provided');
  SetLength(FWeights, length(aWeights));
  TotalWeight := 0.0;
  FCount := succ(High(aWeights));
  for i := 0 to pred(FCount) do begin
    TotalWeight := TotalWeight + aWeights[i];
    FWeights[i] := TotalWeight;
  end;
  if (abs(TotalWeight - 1.0) > Epsilon) then
    raise Exception.Create('The weights to not total 1.0');
  FWeights[pred(FCount)] := 1.0;
end;
destructor TaaRandomWeightedGenerator.Destroy;
begin
  SetLength(FWeights, 0);
  inherited Destroy;
end;
function TaaRandomWeightedGenerator.Get : integer;
var
  Value : double;
  i     : integer;
begin
  Value := Random;
  for i := 0 to pred(FCount) do begin
    if (Value < FWeights[i]) then begin
      Result := succ(i);
      Exit;
    end;
  end;
  {we shouldn't ever get here: this staement is merely to
    fool the compiler into not giving us a warning about the
    result value}
  Result := FCount;
end;
```

data points, obviously, are in increasing order.

"Generate a floating point random number between 0.0 and 1.0. This random number will fall somewhere within our array, generally in between two of the data points. We can do a normal search within the array to find out where this is. The place where the random number falls indicates the number we are supposed to output. For example, if it fell between the first and second data points, the number we output is 2.

"It's as if we are throwing darts at a long strip of paper, divided into sections according to the original weights." He grinned, typed away and produced Listing 3. He'd decided to make it a class since the calculation of the cumulative weights for every random number generated was too inefficient.

"Now here's one for you," he said. "I've been taking a course in statistics for this e-Christmas venture. My fellow Fathers want a bunch of trend data, correlations and that kind of thing to see how they're doing and to spot opportunities in their various markets. In general, calculating the statistical values is all fairly easy: you just follow the formulae in the course book and convert to code. One set of measures has me stumped though: the median and the various required percentiles. As far as I can see, the only way to calculate these values is to sort the data and then choose the middle one for the median, the one $1/20^{th}$ along for the $5^{th}$ percentile, the one $19/20^{th}$ along for the $95^{th}$ percentile and so on. Is there a better way?"

I stared at him. "You expect me to remember this stuff off the top of my head? We're out here in the middle of nowhere and you want me to just explain it?" He looked abashed. "Sorry, old chap, I forgot." He rummaged around again and produced a DVD. "It's got all the important algorithm books on it," he explained. "I got one of the elves to scan 'em all in."

I browsed a bit. "Ah ha, I remember now, there is a faster method using something similar to quicksort. However we don't go all the way and actually sort the data, all we do is just organize the data enough that the median falls in the right place.

"Let's go back and review quicksort for an array. The way quicksort works is that we *partition* the array into two parts about a *pivot* element. All elements that are less than the pivot are placed to the left of the pivot value, and all those greater than the pivot value to the right. The two parts are not sorted, by any means, but at least we can say that the pivot element is in the correct place in the array if the array was really sorted.

"We now perform the quicksort algorithm on the left hand side and then on the right hand side, that is quicksort can be written as, partition, quicksort the left, quicksort the right. This is of course a recursive algorithm. Eventually, we recurse to a point where we can no longer partition (there's only one element in the part we're looking at) and of course this means the sub-sub-sub array we have is

```
function CalcPercentile(var aItemArray : array of
  TDataElement; aLeft, aRight : integer; aLessThan :
  TLessFunction; aPosn : integer) : TDataElement;
  function Partition(L, R : integer): integer;
  var
    i, j : integer;
    Last : TDataElement;
    Temp : TDataElement;
  begin
    {set up the indexes}
    i := L;
    j := pred(R);
    {get the partition element}
    Last := aItemArray[R];
    {do forever (we'll break out of the loop when needed)}
    while true do begin
      {find the first element greater than or equal to the
        partition element from the left; note that our
        partition element will stop this loop}
      while aLessThan(aItemArray[i], Last) do
        inc(i);
      {find the first element less than the partition
        element from the right; check to break out of the
        loop if we hit the left element - we have no
        sentinel there}
      while aLessThan(Last, aItemArray[j]) do begin
        if (j = L) then
          Break;
        dec(j);
      end;
      {if we crossed get out of this infinite loop to swap
        the partition element into place}
      if (i >= j) then
        Break;
      {otherwise swap the two out-of-place elements}
      Temp := aItemArray[i];
      aItemArray[i] := aItemArray[j];
      aItemArray[j] := Temp;
      {and continue}
      inc(i);
      dec(j);
    end;
    {swap the partition element into place, return the
      dividing index}
    aItemArray[R] := aItemArray[i];
    aItemArray[i] := Last;
    Result := i;
  end;
var
  DividingItem : integer;
begin
  while (aLeft < aRight) do begin
    {partition about the final element in the set}
    DividingItem := Partition(aLeft, aRight);
    {select which part to further partition}
    if (DividingItem = aPosn) then begin
      Result := aItemArray[DividingItem];
      Exit;
    end;
    if (DividingItem < aPosn) then
      aLeft := succ(DividingItem)
    else
      aRight := pred(DividingItem);
  end;
  Result := aItemArray[aLeft];
end;
```

➤ *Listing 4: Calculating the value of an element in a sorted array without sorting it.*

sorted and we start completing all the recursive calls we made to get to where we were.

"Well, with percentile calculations, we don't quite go all the way. Suppose we perform the first partition call. At this point we have a single element in the correct position, and furthermore to the left all the elements are less than this and to the right they're all greater. If the position of the element we've just placed is exactly halfway through the array then we've found the median and can stop. Generally, of course, we haven't. However, we can easily work out where the median element lies (either to the left or to the right) and so we can partition the relevant part of the array. This puts another element into its correct place, and again we can either stop because that was the median, or select the correct part of the partition and do it again.

"This process is almost like binary search, in a way. On average, we shall ignore roughly half of the array every time we partition the array, and we can zero in on the median fairly quickly.

"The only thing left is to recall the partition algorithm. Going back to the *Algorithms Alfresco* article for Issue 37, we can steal the partition code from that.

"Percentile calculations work in exactly the same way, except that we're no longer trying to find the value of the element that's in the middle, we're trying to find the value of the element 5% or 10% along (or whatever the percentile value is we're looking for). Listing 4 shows the completed code for calculating a given percentile of a given array. (It actually calculates the value of a given element in an array, if that array were sorted. The median would be the value of the element in the middle of the array, that is at the position given by the count divided by two.)

"Once we do that of course we can do things like extract out all the elements between the 5% and 95% percentile, ignoring the 'way out' data out in the tails, and then calculate the mean, standard deviation, and so on of the best 90% of the data."

I leaned back against the trunk as Father Christmas looked over the code. "I don't suppose you'd convert it to Java," he started to ask and then fell quiet as he saw my face. "No, I suppose not. I'll get one of the elves to do it."

Father Christmas delved into his bag again and pulled out two ice-cold bottles of Evian and offered one to me. "In last month's article on external mergesort you were kind enough to show us how to mergesort a file," he said, after drinking his fill. "But how can you modify the algorithm to sort an arbitrary set of data? Maybe the data doesn't come in a handy file."

"Oh, that's easy," I said. "But we do need to lay down some ground rules. Suppose we want to write a sorter object to do this. It has a property defining the length of the records we want to sort. We then proceed to give it a set of records through many calls to a method (call this one `Add`). After we've given it the whole set of records, we get it to sort them, and then we proceed to extract the records from the object in sorted order through many calls to another method (call this one `Get`). What we do with the records in sorted order is entirely up to us: the sorter object doesn't need to know.

"So we have three phases in our hypothetical object: adding, sorting and retrieving. Actually, as it happens, we can get away with only two: adding and retrieving, because the very first call to the `Get` method causes the records we've added to be sorted." Father Christmas smiled at this and nodded.

"What we need to do is really simple. We allocate a big block of memory as an array; sufficient to hold many of the records we may

receive. During calls to the Add method we append the records we're given onto this array. Eventually, one of two things will happen: either the user calls the first Get, or we have no more room to add a record. In the first case, we quicksort the array of records we've accumulated and then start doling them out to the user. In the second case, we write the records to a merge file, called F1, to use the language of last month's article."

Father Christmas suddenly took over. "Ah, I see. What you are doing is using the Add method to build up blocks of records that you subsequently write to files F1 and F2. At the first call to Get, you then proceed to merge F1 and F2 into files G1 and G2, doubling the run length, and then merge G1 and G2 into F1 and F2, until you are left with a single file containing all the records, in sorted order. In other words, the Add method replaces the SplitFile routine you presented last month."

"Exactly," I said. He took his machine, made a copy of the mergesort code, and of my quicksort code from way back when, and then started to cast the whole lot into a class. "How much memory do I use for the array of records?" he asked half way through. "Make it a property," I answered and watched the deer. Finally he looked up triumphantly and presented the code to me (Listing 5 shows his Add and Get methods for this class).

"What next?" he asked, rubbing his hands. "A final algorithm, something to set us on our way?"

"OK, how about this one? At TurboPower, we have a weekly task. We each write a tech tip on one of our products in rotation. The last one I wrote was on how to read a file or a stream created with zlib with our Abbrevia compression library. If you remember, zlib is an open source compression library and you get a unit for using it with Delphi. It turns out that the compression part isn't a problem, the difficulty is that zlib uses a different checksum to zip files. Zip files use an ordinary 32-bit CRC (see my article from August 1999 for details), but zlib uses a special one called an Adler checksum, named after one of the authors of zlib.

"The algorithm itself is pretty simple and can be explained in a few sentences. Optimizing it is an interesting exercise, though.

"We start off with a longword variable, the checksum, with the value 1. For each byte in the stream, we do this: split the current checksum into two 16-bit halves, the top and the bottom. Add the byte to the bottom half, mod 65521. Now add the bottom half to the top half, also mod 65521. Join the two halves together again to form a 32-bit checksum. Repeat this process with every byte in the stream." I coded Listing 6 as I spoke: it calculates the Adler checksum for a block of data. I made a couple of

➤ *Listing 5: A sorter class using mergesort: Add and Get.*

```
procedure TaaSorter.Add(var aRecord);
begin
  {if we've no buffer, allocate one}
  if (FBuffer = nil) then
    srGetBuffer;
  {check to see whether we've filled the buffer}
  if (FRecCount = FMaxRecCount) then begin
    {if this was the first time that we filled the buffer,
      create the merge files}
    if (FState = AddingState) then
      srCreateMergeFiles;
    {sort then copy this bufferful of records to the correct
      Fx file}
    srQuicksortBuffer;
    TFileStream(FDestFile).WriteBuffer(FBuffer^, FRecLen *
      FRecCount);
    {change the destination file for the next one}
    if (FDestFile = FF1) then
      FDestFile := FF2
    else
      FDestFile := FF1;
    {reset the buffer}
    FRecCount := 0;
    {make sure the state is correct}
    FState := AddWithMergeState;
  end;
  {add this record to the buffer}
  Move(aRecord, FBuffer[FRecLen * FRecCount], FRecLen);
  inc(FRecCount);
  {make sure the state is correct}
  if ((FState=WaitingState) or (FState=FinishedState)) then
    FState := AddingState;
end;
function TaaSorter.Get(var aRecord) : boolean;
var
  BytesRead : integer;
begin
  {get rid of the simple case}
  if (FState = FinishedState) then begin
    Result := false;
    Exit;
  end;
  {if the state is "adding records" then we need to
    quicksort the buffer and change the state to "getting
    records"}
  if (FState = AddingState) then begin
    srQuicksortBuffer;
    FCurRec := 0;
    FState := GettingState;
  end;
  {if the state is "adding records using mergefile" then we
    need to write out the final buffer to the correct
    destination file, and merge the files. The state gets
    changed to "getting records with merge"}
  if (FState = AddWithMergeState) then begin
    srQuicksortBuffer;
    TFileStream(FDestFile).WriteBuffer(FBuffer^, FRecLen *
      FRecCount);
    srMergeFiles;
    FCurRec := 0;
    FRecCount := 0;
    FState := GetWithMergeState;
  end;
  {if the state is "getting records" return the next one in
    the buffer; if there is none, return false}
  if (FState = GettingState) then begin
    if (FCurRec = FRecCount) then begin
      Result := false;
      FState := FinishedState;
    end else begin
      Move(FBuffer[FCurRec * FRecLen], aRecord, FRecLen);
      inc(FCurRec);
      Result := true;
    end;
  end
  {if the state is "getting records with merge" return the
    next one in the buffer; if there is none, try and read
    another buffer full from the final merge file; if
    there's still none, we're finished}
  else begin
    if (FCurRec = FRecCount) then begin
      BytesRead := TFileStream(FSrcFile).Read(FBuffer^,
        FMaxRecCount * FRecLen);
      {if there's nothing left in the final merge file,
        we're done}
      if (BytesRead = 0) then begin
        Result := false;
        FState := FinishedState;
        Exit;
      end;
      {calculate the number of records in this final buffer}
      FRecCount := BytesRead div FRecLen;
      FCurRec := 0;
    end;
    {copy the current record over}
    Move(FBuffer[FCurRec * FRecLen], aRecord, FRecLen);
    inc(FCurRec);
    Result := true;
  end;
end;
```

enhancements in that I only split the checksum into two at the beginning and rejoined them at the end.

"The problem with this algorithm is not in the checksum it produces (in fact it can be shown that the properties of an Adler checksum are commensurate with those of the ordinary CRC) but in the nasty calculation requirements. If you recall CRCs are calculated with a big table and lots of bit operations like XORs and shifts. The Adler checksum needs two additions and two divisions per byte.

"So how can we improve on this? The first answer we can give, it seems, is not a lot. The algorithm is pretty clear about what goes on. As in the code I've just given, we need the checksum of a block of bytes, or we can arrange things so that we calculate an incremental checksum based on blocks of bytes, rather than a single byte at a time. So our second answer is to analyse the algorithm based on several bytes at once.

"Suppose we have a stream of bytes: `b1`, `b2`, `b3`, etc. Call the two halves of the checksum `S1` and `S2`. After the first byte is processed we have:

```
S1=(1+b1) mod p
S2=(1+b1) mod p
```

Where `p` is 65,521 and we remember that we start the whole thing off with an initial checksum value of 1. With the next byte it all seems to get hairy, really quickly. For instance, `S1` is now:

```
S1=(((1+b1) mod p)+b2) mod p
```

It looks pretty bad until we remember some equalities that exist with modulus arithmetic. Namely, adding two values and then calculating their modulus is the same as adding the two values after taking their modulus, and then taking the modulus of the result. In other words:

```
S1=(1+b1+b2) mod p
```

And similarly:

```
S2=(2+2*b1+b2) mod p
```

Writing out the next few values, we see that after n bytes we have:

```
S1=(1+b1+b2+..+bn) mod p
S2=(n+n*b1+(n-1)*b2+..+bn)
  mod p
```

So now the problem boils down to something different: how many bytes can we process before one of the calculations overflows a `longword` (we'll assume that we hold `S1` and `S2` in `longwords`). When that point is reached we'll need to take the modulus of both values (producing a value between 0 and 65,520, a 16-bit value) and then continue. The answer for `S1` is fairly easy to calculate: assuming we start at 1, we can add a little less than $2^{24}$ bytes of value 255 (the maximum) before we overflow. (This is the minimum of course, we may be lucky and add more, but at least this gives us a flavor of the value). Once we've gone through the cycle once, our starting point will be a number of maximum value 65,521, but this isn't going to affect things too much.

"The answer for `S2` is going to be smaller, way smaller. We're essentially calculating a triangular number. Make all the `b`s equal to 255 and `S2` then evaluates to

```
S2=(n+255*tri(n)) mod p
```

Where tri($n$) is the $n$th triangular number. A triangular number is like a factorial, except you add the terms instead of multiplying them:

```
tri(1)=1
tri(2)=2+1=2
tri(3)=3+2+1=6
```

and so on. The formula for Tri($n$) is thus $n(n+1)/2$ (the proof is by induction for all you math whizzes out there). This amounts to solving a good old quadratic equation, and if you do so you get $n$=5,552. Hence we can add 5,552 bytes into our partial sums `S1` and `S2` before we need to calculate their modulus with respect to 65,521. We have

➤ *Listing 6: Simple implementation of the Adler checksum algorithm.*

```
function UpdateAdlerSimple(aAdler : longword;
  var aBuffer; aCount : integer) : longword;
var
  S1 : longword;
  S2 : longword;
  i  : integer;
  Buffer : PChar;
begin
  S1 := aAdler and $FFFF;
  S2 := aAdler shr 16;
  Buffer := @aBuffer;
  for i := 0 to pred(aCount) do begin
    S1 := (S1 + ord(Buffer^)) mod 65521;
    S2 := (S2 + S1) mod 65521;
    inc(Buffer);
  end;
  Result := (S2 shl 16) or S1;
end;
```

➤ *Listing 7: Optimized Adler checksum.*

```
function UpdateAdler(aAdler : longword;
  var aBuffer; aCount : integer) : longword;
var
  S1 : longword;
  S2 : longword;
  i  : integer;
  Buffer : PChar;
begin
  Assert(aCount <= 4096,
    'the UpdateAdler routine has been optimized for buffers up to 4KB');
  S1 := aAdler and $FFFF;
  S2 := aAdler shr 16;
  Buffer := @aBuffer;
  for i := 0 to pred(aCount) do begin
    inc(S1, ord(Buffer^));
    inc(S2, S1);
    inc(Buffer);
  end;
  S1 := S1 mod 65521;
  S2 := S2 mod 65521;
  Result := (S2 shl 16) or S1;
end;
```

two additions per byte, with two divisions per 5,552 bytes. Much easier, and more efficient to boot."

Father Christmas nodded, and typed Listing 7. He assumed we were calculating the Adler checksum for a stream which we were reading in blocks of 4,096 bytes, a favourite stream buffer size of mine. We tested both routines: the simple one took 4,982 milliseconds for a multi-megabyte file, the optimized one 206 milliseconds. Not bad for a simple application of elementary mathematics.

Suddenly, up ahead on the trail, I saw the figure of my wife, running towards us. I jumped up and waved at her. She soon arrived. I looked around to introduce her, but we were alone. The deer looked at us, startled, as if they'd never noticed us before, and bounded away. My wife wanted to know what was I doing just sitting around on the grass instead of biking and getting some healthy exercise, and took off down the trail toward the car. I sighed, picked up my bike and followed, carrying the two empty bottles of Evian.

---

Julian Bucknall wishes all his readers a Merry Christmas and a Happy New Year (or whatever they may celebrate). He'll be back in 2001, after enjoying the *real* start of the Millennium. He 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.