Silent Night

A stocking full of algorithms

by Julian Bucknall

The house was quiet. My wife had given up on law school homework and fallen asleep on the bed, the cats with her. I was in the office, algorithm books spread across the desk, researching graph structures and algorithms for my next *Delphi Magazine* column. From the Windows 95 CD player came the soft repetitive tune from a Philip Glass piece helping me in my contemplation of Dijkstra's algorithm. Outside it was a silent, still night, leading up to a sharp frost.

I was about ready to call it a night, when I heard a rustling behind me. I turned, expecting to find that one of the cats had crept in to keep me company, but was surprised to see a rather portly gentleman with a large white beard, dressed in red, standing behind my chair.

"What on Earth!" I started saying, as a prelude to something intelligent. I didn't get there because he placed his finger to his lips, turned and shut the door.

"I'm the Father Christmas of Programmers," he said. I must have looked dubious, because he went on. "Look, you'll admit that it's impossible for a single Father Christmas to deliver all those presents to all those people at all those houses all on one night." I nodded. "So, there's actually a whole bunch of us, each one attending to one particular segment of the market. And I'm the one for programmers." He positively beamed at this. I suddenly noticed that his red jacket had a pen protector in the top pocket, filled with old pencils, ballpoints and highlighters.

"So?" I asked.

"Well, normally I go around on Christmas Eve, fire up programmers' computers and leave them a little source file containing an algorithm in their \PROJECTS directory, but to be honest, this year, I feel like doing it another way. And you're going to help me."

"Uh huh." My brain still hadn't quite caught up yet and the clip-clopping sound from the roof above my head was distracting me a little.

"Yes, you can write an article containing several algorithms, get it published as the December edition of your column, and I fulfill my obligation to all the programmers who've been good this year. And I can put my feet up with a glass of sherry on Christmas Eve and watch the telly like everyone else." His eyes twinkled at the thought and fell silent for a moment.

"Like what?" I said. "Normally I write an article on a large topic, one that will engage my readers, and one from which I hope they'll have learnt something by the end. Besides which, my Esteemed Editor kind of expects it from me."

"Oh come on, he'll accept it. Otherwise he won't get any presents in his stocking." The twinkling had gone and my blood ran cold at the thought.

He suddenly swept all the books off my desk before I had a chance to protest. Somehow his hand flashed though the air and plucked one out of the air before it hit the floor. "Mustn't damage Knuth Volume III," he muttered as he set it on the shelf. On my desk he placed a notebook he'd retrieved from his jacket. I just had a chance to spot the model, Dellwaypaq 9000, on the lid before he opened it. One click of the mouse and Delphi 4 opened up in a heartbeat. He either didn't have any components on his palette or this was one mean machine.

"Right, what do you know about credit card numbers?" he asked.

"Well, they're a number with a checksum. You apply some arithmetic operation on the digits of the credit card number including the check digit and the result is supposed to be zero. It's one of a whole class of similar self-validating numbers: ISBN numbers use a different checksum algorithm, some barcodes have check digits, and so on. It's supposed to catch a whole class of operator input errors." I rummaged though my algorithms file box for a moment or two and pulled out a paper.

"Let's see. Credit card numbers are validated as follows. Take each digit in turn, starting from the right. Form a running sum. For the digits in the odd positions (the last, third-from-last, fifth-from-last, etc), just add the digit to the sum. For the digits in the even positions (second-from-last, etc) multiply each by 2. If the result of this multiplication is less than 10, add the result to the sum. If the result is greater than 10, add each of the individual digits of the result to the sum. The sum, after this is over, must be a multiple of 10. In other words, the sum mod 10 equals zero. Apparently this algorithm is known as the IBM mod 10 check." I scribbled on a pad. "Let's try it on 123456789. Set my sum to zero. Add in the 9. Double the 8, to make 16, and then add the 1 and the 6 separately to my sum. Add in the 7. Double the 6, to make 12, and then add the 1 and the 2 to my sum. Add in the 5. Double the 4 to make 8 and add it to my sum. Add the 3. Double the 2 to make 4 and add it to my sum. Add the 1. Grand total: 47. Hence it fails the validation. However if the final digit were 2 instead of 9 it'd pass."

Father Christmas looked blank, so I pulled his notebook over and started coding. "Let's assume that we have a validation routine that accepts a credit card number as a string and returns whether the

number is a valid number or not. Furthermore the routine will just ignore all characters that aren't numeric digits, so that the operator can enter numbers with spaces or without, or with dashes or something." I typed away for 5 minutes and produced Listing 1.

"You said ISBN numbers follow a similar scheme," Father Christmas said.

"Yes, but this time they use a mod 11 check. We start from the last digit again. Form a running sum. Add the last digit, twice the second-from-last digit, three times the third-from-last, and so on. If the number is long enough, we'll add 10 times the tenth-from-last, the eleventh-from-last, twice the twelfth-from-last, and so on. The final total mod 11 must be zero. If you think about it, it may mean that the last digit, the check digit, be ten to pass the validation. Since ten is not a digit, they use X instead, presumably from Roman numerals." Father Christmas' eyes lit up at this mention, but I went on. "Let's use Knuth Volume III's ISBN number as an example." I picked up the book from the shelf and opened it. "It's 0-201-89685-0. Starting from the end, we calculate (0*1) + (5*2) + (8*3) + (6*4) + (9*5) + (8*6) + (1*7) + (0*8) + (2*9) + (0*10). Which produces a total of 176. And 176 mod 11 equals 0." I typed some more and produced Listing 2.

"Very good," said Father Christmas and offered me a can of cola. We popped the rings and drank. He burped delicately and continued. "In your previous job writing software for Swaps traders, you often had to calculate the dates of the third Wednesday of March, June, September or December, because they were the futures settlement dates or something. How did you do that?"

I had to switch tracks and think a minute. "All right, let's see. In those days all this was done using my own date unit, but let's use `TDateTime` variables and the routines in `SysUtils` instead. Firstly you calculate the day of the week that the first of the month falls on. That's easy. We use `EncodeDate` to calculate the date of the first of the month in question. If we use December 1998 as our example month, then we code:

```
Dec_1_1998 :=
  EncodeDate(1998, 12, 1);
```

"And then we use the `DayOfWeek` function to return the day of the week, a number from 1 to 7 with 1 representing Sunday and 4 for Wednesday. Now comes the fun stuff. The first Wednesday of a month must be between the first to the seventh day of the month, the second must be between the 8th to the 14th of the month, the third Wednesday must appear between 15th and the 21st inclusive. If the first of the month was a Sunday (a `DayOfWeek` number of 1) then the first Wednesday would be on the 4th, and hence the third would be on the 18th, fourteen days later. If the first of the month was a Monday then the first Wednesday would be on the 3rd and the third Wednesday would be on the 17th. And so on."

Father Christmas took control of his notebook and started coding. "A first attempt at this might look like this then," he said, showing me Listing 3.

```
function ValidateCreditCardNumber(const aValue : string) : boolean;
var
  i, Total, Dbl : integer;
  Ch : char;
  IsOddPosn : boolean;
begin
  if (aValue = '') then begin
    Result := false;
    Exit;
  end;
  Total := 0;
  IsOddPosn := true;
  for i := length(aValue) downto 1 do begin
    Ch := aValue[i];
    if IsDigit(Ch) then begin
      if IsOddPosn then
        inc(Total, ord(Ch) - ord('0'))
      else begin
        Dbl := (ord(Ch) - ord('0')) * 2;
        inc(Total, Dbl);
        if Dbl >= 10 then dec(Total, 9);
      end;
      IsOddPosn := not IsOddPosn;
    end;
  end;
  Result := (Total mod 10) = 0;
end;
```

➤ *Listing 1: Verify credit card numbers.*

```
function ValidateISBN(const aValue : string) : boolean;
var
  i     : integer;
  Total : integer;
  Multiplier : integer;
  FirstDigit : boolean;
  Ch    : char;
begin
  if (aValue = '') then begin
    Result := false;
    Exit;
  end;
  FirstDigit := true;
  for i := length(aValue) downto 1 do begin
    Ch := aValue[i];
    if FirstDigit then begin
      if IsDigitOrX(Ch) then begin
        if (Ch = 'X') then
          Total := 10
        else
          Total := (ord(Ch) - ord('0'));
        FirstDigit := false;
        Multiplier := 2;
      end
    end else begin
      {not the first digit}
      if IsDigit(Ch) then begin
        inc(Total, (ord(Ch) - ord('0')) * Multiplier);
        inc(Multiplier);
        if (Multiplier = 11) then
          Multiplier := 1;
      end;
    end;
  end;
  Result := (Total mod 11) = 0;
end;
```

➤ *Listing 2: Verify ISBN.*

"Pretty good," I said. "For maintenance' sake I'd probably leave it like that as well, maybe replacing the 4s with some named constant to make it clear. Essentially, that's what my old code did. Of course, in my line of work at the time that calculation was all I needed, but it would be better to make the routine more generic so that you could calculate the first Sunday, the fourth Friday, or whatever. You'd pass the year and the month, the day of the week and which one you wanted: the first one, the second, or whatever."

He typed away again and produced Listing 4. Just to needle him a little, I said, "Of course, now you can calculate the ISO week of the year for a given date, providing you know that week 1 always contains the first Thursday of the year and the first day of a week is Monday." He looked askance at me while he thought, and then typed away. Listing 5 was the result, after a bit of testing and realizing that a date in a given year could appear in the final few days of the final week of the previous year.

I must have needled his programming prowess, for then he came back with a rather barbed comment. "One of my elves tells me that you were having a problem passing data around with Winsock the other day and it took you a couple of days to resolve the bug. Care to elucidate?" I held my tongue from a rather barbed comment of my own, after all I like presents on Christmas Day as well, and started explaining.

"TCP/IP communications using Winsock has guaranteed delivery. I knew that. However, it is not a block-oriented communications protocol, but a streaming protocol. In other words, if I send two blocks of data from one machine to another, then the receiver may get the same two blocks of data, or, depending on the loading of the machines or the network, just one block containing both my original blocks, or three or more blocks, each with some part of my original blocks. I would be responsible for parsing out my original blocks at the receiver. In my rush to code this bit of code, I'd forgotten that TCP/IP was such a protocol, and had assumed that each block would be received entire. In fact, during my simplistic testing, this was the case and it was only under a stressed network and stressed machines that this assumption broke down.

"So once I'd worked out the why and wherefore of the bug, I had to squash it with some new data structure. I wanted some explicit functionality. There must be an `Add` operation to add a bunch of data (a block of bytes) to the structure.

"There must be a `Remove` operation to remove a set of bytes from the structure.

"I must be able to know how many data bytes there were in the structure, a `Count` property if you will.

➤ *Listing 3*

```
FirstOfMonth := EncodeDate(1998, 12, 1);
FirstDay := DayOfWeek(FirstOfMonth);
if FirstDay <= 4 then
  ThirdWed := 4 - FirstDay + 14 + FirstOfMonth
else
  ThirdWed := 4 - FirstDay + 21 + FirstOfMonth
```

➤ *Listing 4: Calculating the nth Monday, etc, of a month.*

```
function GetDateForDayOfMonth(aWhichOne : integer; aDay : integer; aMonth :
  integer; aYear : integer) : TDateTime;
var
  Month1st : TDateTime;
  Day1st   : integer;
begin
  ... {validate}
  {calculate}
  Month1st := EncodeDate(aYear, aMonth, 1);
  Day1st := DayOfWeek(Month1st);
  if (Day1st <= aDay) then
    Result := aDay - Day1st + ((aWhichOne-1) * 7) + Month1st
  else
    Result := aDay - Day1st + (aWhichOne * 7) + Month1st;
  if (Result - Month1st + 1) > MonthDays[IsLeapYear(aYear), aMonth] then
    Result := 0.0;
end;
```

➤ *Listing 5: Calculating the ISO date.*

```
function CalcFirstWeek(aYear : integer) : TDateTime;
{returns date of the Monday of week 1 of the given year}
const
  DOWThu = 5;
var
  Month1stJan : TDateTime;
  Day1stJan   : integer;
begin
  Month1stJan := EncodeDate(aYear, 1, 1);
  Day1stJan := DayOfWeek(Month1stJan);
  if (Day1stJan <= DOWThu) then
    Result := DOWThu - Day1stJan + Month1stJan - 3
  else
    Result := DOWThu - Day1stJan + Month1stJan + 4;
end;
procedure GetISODate(aDate : TDateTime; var aYear :
  integer; var aWeek : integer; var aDay  : integer);
var
  WeekOneStart: TDateTime;
  Year     : word;
  Month    : word;
  Day      : word;
begin
  {calculate date of Monday for first week for date's year}
  DecodeDate(aDate, Year, Month, Day);
  WeekOneStart := CalcFirstWeek(Year);
  {if the given date is greater than/equal to the 1st week
   start date calculate the week number and day number}
  if (aDate >= WeekOneStart) then begin
    aYear := Year;
    aWeek := (Trunc(aDate - WeekOneStart) div 7) + 1;
    aDay := (Trunc(aDate - WeekOneStart) mod 7) + 1;
    {check to see if the given date could appear in the
     first week of the following year, if so so do the same
     calculation again, but for the next year}
    if ((aDate - WeekOneStart) > 364) then begin
      WeekOneStart := CalcFirstWeek(Year+1);
      if (aDate >= WeekOneStart) then begin
        aYear := Year+1;
        aWeek := (Trunc(aDate - WeekOneStart) div 7) + 1;
        aDay := (Trunc(aDate - WeekOneStart) mod 7) + 1;
      end;
    end;
  end
  {if the given date is less than the 1st week start date,
   it'll be in the last week of the previous year, so do
   the same calculation again, but for the prior year}
  else begin
    dec(Year);
    WeekOneStart := CalcFirstWeek(Year);
    aYear := Year;
    aWeek := (Trunc(aDate - WeekOneStart) div 7) + 1;
    aDay := (Trunc(aDate - WeekOneStart) mod 7) + 1;
  end;
end;
```

"The structure must be able to grow itself if required.

"And finally, I needed a Peek operation. The blocks of data I was passing from machine to machine were messages. Each message had the same header, and then there might be some extra data tacked onto the end of the message. There was a field in the message header that detailed the size of the entire message in bytes. If I had an operation that allowed me to quickly look at the first few bytes of data, I could peek at the length field for the next message and determine from the Count property whether the entire message was present in the structure or not."

I took a sip of cola; Father Christmas finally pulled up the other chair in the office and sat down.

"My first thought was a circular queue. In other words, have a large array of bytes and have two pointers into that array: the head pointer, where I remove data and the tail pointer, where I add data. Of course, there comes a point when the tail pointer wraps around from the end of the array to the beginning again and the available data consists of the bit from the head pointer to the end of the array, and the bit from the beginning to the tail pointer.

"But I just have this abhorrence of circular queues and try and avoid them whenever possible. I don't know why, but I just get this series of mental blocks when I try and code around the data wrap-around I just described. Luckily for me, my required Peek operation precluded me using a circular queue. There's no way I'd countenance having a Peek operation that required me to stitch together the two pieces of data after a wrap-around had happened. It was likely I would be using Peek a lot, so it had to be fast.

"So, what else could I use then? I thought about it for a while and decided on a linear queue instead, one with an overflow area. Let me explain what I mean. Let's suppose that the user of the queue states that he wants enough room for 1,024 bytes of data. In fact, the queue allocates an array of bytes of twice that size so that it has an overflow area. As you add and remove data from the queue the head and tail pointers move along the array into the overflow area. Think of the Add operation: the first thing that happens is that we must check to see if there's enough room in the array to hold the new block of data. If so fine, we add it and advance the tail pointer to the end of the new data. If we *don't*, we create a new array of a larger size (say twice as large), copy over the data in the existing array, and then add the data as before. But we don't copy the data into the same place in the new array: we copy it so that it appears at the front of the array and reset our head and tail pointers accordingly. Since we have to copy the data when growing the array, let's make the most of it and copy the data to the front of the array and put off any other moves as long as we can.

"Now think of the Remove operation. We copy the data from the head pointer onwards into the user's buffer, and advance the head pointer. If the head pointer now equals the tail pointer, there's no more data in the array, so the queue resets both pointers to the front of the queue. If this isn't the case, there is still data in the array, so we check to see whether the head pointer has passed into the overflow area; in other words whether we've reached and passed the half way mark. If we have, all of the data appears in the overflow area and so we take the opportunity to copy it all back into the first half of the array.

"As you can imagine, I'm always trying to avoid the copying of data when the head pointer reaches the half-way mark in the array, by having the queue reset the head and tail pointers towards the front of the array whenever I can.

"And the big plus of this queue is that the Peek operation just returns the head pointer. Mind you, a big minus is that the Peek operation makes the queue completely non-thread-safe: since the Peek operation returns a pointer into an internal structure, you'll have a problem if several threads are adding or removing data. The internal structure may be replaced by another array, the data may be removed by another thread, and so on. The only way round this is to alter the Peek operation so that it copies the data into a user buffer in a thread-safe manner, but of course this still means that there can only be one consumer of data from the queue: it's all very well peeking at data only to find that it's gone when you go to remove it." Father Christmas nodded sagely at this.

I showed him the code I'd written (available on this month's disk) and he pored over it for a while. "Neat," was his only comment.

"While we're knocking around ideas for stocking algorithms," I said, "I've got a real good one. This is something that caught me out a few years ago: shuffling cards. Imagine you have a zero-based array with 52 elements containing the numbers from 1 to 52 in order, representing a sorted deck of cards. Implement an algorithm that efficiently shuffles the elements in an unbiased way."

He thought for a while, smiled and said, "All right, for each element in turn, generate a random number from 1 to 52 and swap it with the element at that random position." He typed:

```
for i := 0 to 51 do begin
  RandInx := Random(52);
  Temp := CardsArray[RandInx];
  CardsArray[RandInx] :=
    CardsArray[i];
  CardsArray[i] := Temp;
end;
```

"Pretty good," I said. "That's exactly the algorithm I came up with. But, and it's a big but, it gives way too many permutations of the cards, and worse, some permutations appear more often than others." He looked thoughtful, and I went on. "After one move, the permutation we get is just one of 52 possible ones. After two moves, the permutation we get is again one out of a possible 52, so over the two moves we now have one out of a possible 52^2 permutations.

As you can see, each move multiplies the number of permutations by 52, so after 52 moves we have a single permutation out of a whopping 52^{52} of them, which according to my calculator is..." Tap, tap, tap. "$1.7 * 10^{89}$."

"So?" Father Christmas asked. "It sounds pretty thorough to me."

"Let's go back to first principles. In a shuffled deck, which is a single permutation of the cards, the probability that the first card in our permutation appears at the first position is 1/52. Given that, the probability that the second card appears at the second position is 1/51, since we already know where one card is, remember. The probability that the third card appears at the third position is 1/50, and so on. The probability therefore of getting our particular permutation is 1/52!, where 52! means 52 factorial, or 52 * 51 * 50..., all the way down to 1. So there are 52! distinct different permutations of a deck of cards, which is $8.0 * 10^{67}$, another whopping huge number. But the problem is that 52! does not divide 52^{52} exactly (pretty obvious since 52! has some prime divisors that don't divide 52) and so some permutations must occur more often than others, making the algorithm biased. No, what we must do is shuffle the cards the way we calculated the number of permutations for a deck: select one out of the 52, then select one out of the remaining 51, and then select one out of the remaining 50 and so on. Of course, you can apply this shuffling algorithm to randomly select a subset of elements from an array." I typed up Listing 6.

```
procedure Shuffle(var aArray; aItemCount : integer; aItemSize : integer);
var
  Inx, RandInx  : integer;
  SwapItem : PByteArray;
  A        : PByteArray absolute aArray;
begin
  if (aItemCount > 1) then begin
    GetMem(SwapItem, aItemSize);
    try
      for Inx := 0 to (aItemCount - 2) do begin
        RandInx := Random(aItemCount - Inx);
        Move(A^[Inx*aItemSize], SwapItem^, aItemSize);
        Move(A^[RandInx*aItemSize], A^[Inx*aItemSize], aItemSize);
        Move(SwapItem^, A^[RandInx*aItemSize], aItemSize);
      end;
    finally
      FreeMem(SwapItem, aItemSize);
    end;
  end;
end;
```

➤ *Listing 6: Shuffling an array.*

"Excellent!" Father Christmas beamed. "This is exactly the kind of thing I wanted. A couple more of these stocking algorithms and I think we're done." I nodded, suppressing a yawn. It was getting rather late and I had to go to work in the morning.

"Here's an interesting one," he said. "Given a floating point value between 0 and 1, in a double variable say, how do you work out the nearest fraction that equals it? For example, if I gave you 0.625, how

would you calculate that it equals 5/8?"

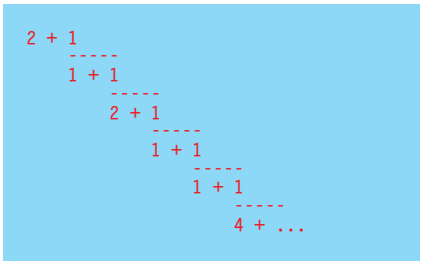
I stared at him. "You're nuts. Anyway, what about rounding errors? Unfortunately, it's not a very well known fact that binary floating point values can only store powers of ½ exactly, everything else is an approximation. It's a bit like saying 1/7 equals 0.142857: of course it doesn't exactly because it's just an approximation to 6 decimal places."

"Well, devise an algorithm that approximates the fraction then." Smug wasn't the word.

"I suppose multiplying by 10 and stripping off the integer part until the remainder was less than some small value won't count. For my example of 0.142857, we'd get a fraction of 142857/1000000." If I'd had a spittoon, Father Christmas would have availed himself of it.

I pondered for a while and scribbled on a piece of paper. I seemed to vaguely recollect something I'd written at university about continued fractions, but when you get as old as me, you have to gently coax your neurons towards the right answer. Some scribbling time later, I came up with a solution.

"Here goes. I'll use continued fractions. A continued fraction looks a little bizarre (see Figure 1) and can make your head hurt a little, especially if they're infinite in extent, but in our algorithm we won't worry about that. You can reduce the amount of space needed to write a continued fraction which has all the numerators



➤ *Figure 1: A continued fraction.*

as 1 by using a simple array representation: the one in the figure is [2, 1, 2, 1, 1, 4, ...] for example. To calculate the decimal representation of a continued fraction, you work your way up the fraction, or if we use the array representation, from the right to the left. There is also a matrix algorithm for calculating the value of a continued fraction from the left, but we won't go into that now.

"What we do is this: given a number, subtract the integer part, store it in the first array element. Then find the reciprocal of the fractional part. Subtract the integer part, store it in the second array element. Then find the reciprocal of the fractional part. Continue like this until the fractional part is less than some agreed constant, usually called epsilon, say 0.0001. What we've done is to find the nearest continued fraction to our original number. All that's required then is to convert the continued fraction into a normal, or vulgar, fraction. Let's try it with 3.14159292, almost π. Subtract 3, store it. Find the reciprocal of the fractional part: 7.062500017656. Subtract 7, store it, find the reciprocal: 15.99999548. Subtract 15, store it, find the reciprocal: 1.00000452. Subtract 1, store it. The result of the subtraction is smaller than our assumed epsilon and so we stop. We then calculate the vulgar fraction from the continued fraction and get 355/113."

"Amazing," said Father Christmas. "Let's see the code!" This time it took a little longer and the testing was a little more intense and involved much use of my calculator. You can see the code in Listing 7. With some testing we noticed that the algorithm wasn't perfect: it seemed that the number of inversions the routine performed increased the error margin quite quickly. But it was pretty good, nevertheless.

When I finished I was tired and slumped back on my chair. "I've really got to go to bed now, TurboPower awaits me in the morning."

"Just one more algorithm, is all I ask," he wheedled. "Something you said earlier triggered something in my mind; and this algorithm will wrap things up nicely. When we were talking about checksums, you mentioned that the ISBN check digit can sometimes be 10, or X in Roman numerals." I nodded, light slowly dawning in my head, and he continued, confirming my suspicions. "So for the last algorithm write a routine or pair of routines that converts from a binary integer to a Roman numeral string and *vice-versa*." I could hear the relish with which he said the final Latin words.

➤ *Listing 7: Converting decimal to vulgar fraction.*

```
procedure ConvertFraction(aValue : double;
  var aNumerator : longint; var aDenominator : longint);
var
  Sign, Iter, i : integer;
  Num, Denom, Temp : double;
  ContFrac : array [0..pred(MaxContFracDepth)] of integer;
begin
  if aValue < 0.0 then begin  {get sign of decimal fraction}
    Sign := -1;
    aValue := abs(aValue);
  end else
    Sign := 1;
  {create the continued fraction}
  FillChar(ContFrac, sizeof(ContFrac), 0);
  ContFrac[0] := Trunc(aValue);
  Iter := 1;
  aValue := Frac(aValue);
  while (aValue >= CvtFracEpsilon) and
    (Iter < MaxContFracDepth) do begin
    aValue := 1.0 / aValue;
    ContFrac[Iter] := Trunc(aValue);
    inc(Iter);
    aValue := Frac(aValue);
  end;
  dec(Iter);
  {convert continued fraction into normal vulgar fraction}
  if (Iter = 0) then begin
    aNumerator := ContFrac[Iter];
    aDenominator := 1;
  end else begin
    Num := 1;
    Denom := ContFrac[Iter];
    for i := pred(Iter) downto 0 do begin
      Temp := Denom * ContFrac[i] + Num;
      Num := Denom;
      Denom := Temp;
    end;
    if (Denom > MaxLongint) or (Num > MaxLongint) then begin
      aNumerator := -1;
      aDenominator := -1;
    end else begin
      aNumerator := Sign * Trunc(Denom);
      aDenominator := Trunc(Num);
    end;
```

"Believe it or not, the conversion from a Roman number to an ordinary number was the first bit of code I copied from a magazine article, way back when. I wrote it all out longhand, maybe the photocopier wasn't working. Anyway..." I rummaged though my filing system again, and produced a piece of paper with a flourish. "It's even written in GWBASIC with line numbers and everything." We looked at it, perusing the unfamiliar `DATA` statements, the `DIM`s, the `LET`s, the myriad `GOTO`s to line numbers, the general spaghettiness of it all.

"Good grief," he muttered. "This'll take ages."

"Rubbish," I said matter-of-factly. "Let's do it the easy way round first: convert a `longint` to a Roman number string. Once we have that we'll have some test data to test the opposite routine automatically." I picked up a general reference book and turned to the section on Roman numbers. "I is one, V is five, X is ten, L is fifty, C is one hundred, D is five hundred and M is one thousand. The Romans used a bizarre system to denote 5,000 and above, so we'll ignore numbers above 3,999 (which is MMMCMXCIX). There's no zero as it was invented much later.

"The easiest way of doing this is to make the code table driven. Create tables for the single digits, the tens, the hundreds, and the thousands. Four tables in all. Then take the input value, peel off the powers of ten, and replace them with the entry in the relevant table." I typed away and produced Listing 8.

Father Christmas nodded and gestured at the GWBASIC code. "And now the other way."

"The code illustrates a table driven state machine," I started off confidently. "Since I've absolutely no desire to convert GWBASIC spaghetti code into Object Pascal, let's recreate the state machine, regenerate the table describing it, and the code will follow pretty quickly from that.

"To make this easier to start off with, let's assume that we only have the X, V, and I numerals and we'll design the state machine for numbers less than ten. Once we've analyzed this case we can easily expand our design to the rest. Look at single characters first. A V can only be followed by an I. An I can be followed by an X, a V or an I (to a maximum of three in a row). Now look at double characters. IX cannot be followed by anything, the same applying to IV, they are terminators. Double I cannot be followed by V or X, it can either stop there or it can be followed by another I. This gives us the state machine in Figure 2. $ means a terminating state. We can rewrite this as Table 1. Here the entries in the table are the amount to add to a sum as we parse a Roman number string followed by the state to move to. If we ever hit a dash our Roman number is invalid, or similarly if we have another letter after hitting state $.

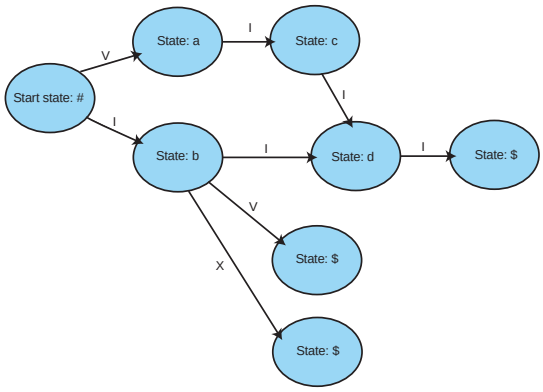
"Let's try it with IX. We start off in state # with sum 0. The first character is an I. We look along the state # line until we reach the I column. This says add 1 and move to state b. The next character is an X. Look along the state b line until we reach column X. This says add 8 and move to state $. We have no more characters and so we finish with a total of 9.

"If we try it with VV, we'll hit a dash, meaning the Roman number was invalid. If we try IIIX, we'll have an extra character after getting to state $, which again means we got an error."

Father Christmas suddenly interrupted. "I see what you're getting at now. The analysis we've done for the numbers 1 to 9, also applies to the tens from 10 to 90, using the characters C, L and X. And similarly the hundreds from

```
const
  RomanDigits : array [1..9] of string[4] =
     ('I', 'II', 'III', 'IV', 'V', 'VI', 'VII', 'VIII', 'IX');
  Roman10s : array [1..9] of string[4] =
     ('X', 'XX', 'XXX', 'XL', 'L', 'LX', 'LXX', 'LXXX', 'XC');
  Roman100s : array [1..9] of string[4] =
     ('C', 'CC', 'CCC', 'CD', 'D', 'DC', 'DCC', 'DCCC', 'CM');
  Roman1000s : array [1..3] of string[3] =
     ('M', 'MM', 'MMM');
function IntToRoman(aValue : integer) : string;
var Digit : integer;
begin
  if (aValue <= 0) or (aValue >= 4000) then
    RaiseBadNumberError(aValue);
  Digit := aValue div 1000;   {get 1000s digit and convert}
  if (Digit <> 0) then
    Result := Roman1000s[Digit]
  else
    Result := '';
  aValue := aValue mod 1000;  {get 100s digit and convert}
  Digit := aValue div 100;
  if (Digit <> 0) then
    Result := Result + Roman100s[Digit];
  aValue := aValue mod 100;   {get 10s digit and convert}
  Digit := aValue div 10;
  if (Digit <> 0) then
    Result := Result + Roman10s[Digit];
  Digit := aValue mod 10;     {get singles digit and convert}
  if (Digit <> 0) then
    Result := Result + RomanDigits[Digit];
end;
```

➤ *Listing 8: Converting integer to Roman number.*



➤ *Figure 2: A state machine for parsing Roman numbers between I and IX.*

Letter:	X	V	I
State: #	-	5,a	1,b
a	-	-	1,c
b	8,$	3,$	1,d
c	-	-	1,d
d	-	-	1,$

➤ *Table 1: The I to IX state machine as a table.*

```
const
  RomanNumerals : string[7] = 'MDCLXVI';
  RomanStateMc : array [0..17, 1..7] of word =
      {Numeral:        M      D     C     L     X     V    I}
      {State 0:} ((32001, 16004, 3205, 1609, 330, 174, 47),
           { 1:}  (32002, 16004, 3205, 1609, 330, 174, 47),
           { 2:}  (32003, 16004, 3205, 1609, 330, 174, 47),
           { 3:}  (    0, 16004, 3205, 1609, 330, 174, 47),
           { 4:}  (    0,     0, 3206, 1609, 330, 174, 47),
           { 5:}  (25608,  9608, 3207, 1609, 330, 174, 47),
           { 6:}  (    0,     0, 3207, 1609, 330, 174, 47),
           { 7:}  (    0,     0, 3208, 1609, 330, 174, 47),
           { 8:}  (    0,     0,    0, 1609, 330, 174, 47),
           { 9:}  (    0,     0,    0,    0, 331, 174, 47),
           {10:}  (    0,     0, 2573,  973, 332, 174, 47),
           {11:}  (    0,     0,    0,    0, 332, 174, 47),
           {12:}  (    0,     0,    0,    0, 333, 174, 47),
           {13:}  (    0,     0,    0,    0,   0, 174, 47),
           {14:}  (    0,     0,    0,    0,   0,   0, 48),
           {15:}  (    0,     0,    0,    0, 287, 127, 49),
           {16:}  (    0,     0,    0,    0,   0,   0, 49),
           {17:}  (    0,     0,    0,    0,   0,   0, 63));
    {-State machine table to convert from Roman numbers to
      Integers. Each entry is equal to (ValueToAdd * 32) +
      NextState for a given State/Roman numeral. State 0 is
      the initial state; state 31 is the terminator state.}
function RomanToInt(aValue : string) : integer;
var
  i, ChInx, State, StateValue : integer;
  Ch : char;
begin
  Result := 0;
  State := 0;
  for i := 1 to length(aValue) do begin
    if (State = 31) then
      RaiseBadRomanNumberError(aValue, i);
    {get the next character, check to see if it's valid}
    Ch := UpCase(aValue[i]);
    ChInx := Pos(Ch, RomanNumerals);
    if (ChInx = 0) then
      RaiseBadCharError(aValue, aValue[i], i);
    {get the value to add, if it's zero we've got a badly
      formed Roman number}
    StateValue := RomanStateMc[State, ChInx];
    if (StateValue = 0) then
      RaiseBadRomanNumberError(aValue, i);
    inc(Result, StateValue div 32);
    State := StateValue mod 32;
  end;
end;
```

➤ *Listing 9: Converting Roman number to integer.*

100 to 900 using the characters M, D, and C. Etcetera, etcetera." He scribbled on my legal pad. "In fact, the final state machine table will have 7 columns, one for each Roman letter and 18 rows, one for each state."

"I agree," I said. We alternately scribbled on the paper, rearranging rows and re-lettering states and compressing the table until we reached a consensus. "Now," I said, "the code is pretty simple." And so it was and together we produced Listing 9 after some testing. Luckily for us and you it didn't have a `GOTO` in sight.

We both yawned simultaneously. "Now, it's *really* time for bed," I said. He smiled and agreed and thanked me for my work. We ceremoniously polished off our cans of cola and tossed them into the wastepaper basket by the wall. Mine missed so I walked over to pick it up. When I looked back the room was empty.

Julian Bucknall wishes all his readers a Merry Christmas and a Happy New Year (or whatever they may celebrate). In 1999 he really will get to Part II of his graphs series. The code with this article is fun and freeware and can be used as-is in your own applications.

© Julian M Bucknall, 1998