7-1-2009

Computing sequences and series by recurrence

Stephen J. Sugden
Bond University, ssugden@bond.edu.au

Chapter 8

Computing Sequences and Series by Recurrence

8.1. Background

Many commonly-used mathematical functions may be computed via carefully-constructed recurrence formulas. Sequences are typically defined by giving a formula for the general term. *Series* is the mathematical name given to partial sums of sequences. In either case we may often take advantage of the great expressive power of recurrence relations to create code which is both lucid and compact. Further, this does *not* necessarily mean that we must use recursive code. In many instances, iterative code is adequate, and often more efficient.

A *recurrence* takes advantage of the fact that to compute the next term, we can often obtain it from the previous term with a little extra effort. To use a direct formula, we need to start from scratch each time. To use a recurrence formula, we just make an adjustment to the previous term. For example, consider the simple formula for a sequence of powers of two. We have $a_n = 2^n$. Now suppose you were given the value of $a_{15} = 2^{15} = 32768$ and were asked to compute $a_{16} = 2^{16}$. Most of the work is already done, since $2^{16} = 2 \times 2^{15} = 2 \times 32768 = 65536$. Here, we have implicitly made use of the simple recurrence $a_n = 2a_{n-1}$. The quantity a_n denotes the nth term, and a_{n-1} is the one before it. We can derive the recurrence as follows. To obtain a_{n-1} we simply replace each occurrence of n in the direct formula for a_n by $n-1$. To get the recurrence, take the quotient of a_n and its predecessor a_{n-1}.

$$\begin{aligned} a_n &= 2^n \\ a_{n-1} &= 2^{n-1} \\ \therefore \frac{a_n}{a_{n-1}} &= \frac{2^n}{2^{n-1}} = 2 \end{aligned}$$

As a general rule of thumb, if a direct formula involves power, factorials, or exponential functions, we can often obtain a recurrence by considering the ratio a_n/a_{n-1}. If the formula for a_n involves addition and subtraction, and perhaps polynomials, it is usually better to consider $a_n - a_{n-1}$.

As noted, even if a function may be defined by a series or by a recurrence, this does not mean that explicit recursive programming has to be used. Almost always, iteration is

entirely adequate. A simple example will illustrate this point. We compute the exponential function e^x by using the familiar power series of eq 8.1.

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots \tag{8.1}$$

Of course, we cannot add an infinite number of terms, so we truncate the series after a certain point. If x is moderate positive value, say 1 or 2, the series will be rapidly convergent and we obtain a good estimate of e^x even if we take about the first 7 or 8 terms. We may show that the *truncation error* committed by ignoring the rest of the series is very small. We have:

$$e^x \approx \sum_{k=0}^{n} \frac{x^k}{k!} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!}$$

We suppose that $x \geq 0$ and consider the truncation error, which is given by eq 8.2. The following steps, culminating in eq 8.3, give one possible upper bound for the truncation error, $E_n(x)$.

$$\begin{aligned}
E_n(x) &= \frac{x^{n+1}}{(n+1)!} + \frac{x^{n+2}}{(n+2)!} + \frac{x^{n+3}}{(n+3)!} + \ldots \qquad (8.2)\\
&= \frac{x^{n+1}}{(n+1)!}\left(1 + \frac{x}{n+2} + \frac{x^2}{(n+2)(n+3)} + \ldots\right)\\
&< \frac{x^{n+1}}{(n+1)!}\left(1 + \frac{x}{n+2} + \frac{x^2}{(n+2)^2} + \ldots\right)\\
&< \frac{x^{n+1}}{(n+1)!}\left(1 + \frac{x}{n+1} + \frac{x^2}{(n+1)^2} + \ldots\right)\\
&= \frac{x^{n+1}}{(n+1)!}\frac{n+1}{n+1-x}\\
&= \frac{x^{n+1}}{(n+1-x)\,n!} \qquad (8.3)
\end{aligned}$$

This last expression gives us a reasonable upper bound for the error introduced by ignoring the remainder of the power series. For example, suppose that we take $x = 2$ and $n = 10$. Our upper bound is then:

$$\frac{x^{n+1}}{(n+1-x)\,n!} = \frac{2^{11}}{9 \times 10!} = 6.2708 \times 10^{-5}$$

By increasing n to 15, we have, for $x = 2$:

$$\frac{x^{n+1}}{(n+1-x)\,n!} = \frac{2^{16}}{14 \times 15!} \simeq 3.5797 \times 10^{-9}$$

Using values of x greater than 1 is not wise as convergence is slower. Even to compute $e = e^1$ itself, it is better to compute $e^{0.5}$ and then square the result. Compare the convergence

for $x = 0.5$ compared with $x = 2$. Substituting $x = 0.5$ and $n = 10$ in eq 8.3, we have:

$$E_{10}(0.5) < \frac{0.5^{11}}{(10+1-0.5)10!} \simeq 1.2815 \times 10^{-11}$$

This is better accuracy than for 15 terms when $x = 2$. Practical algorithms to compute functions like $e^x, \sin x, \cos x$ first of all use properties of the function itself to reduce x to a suitable interval. Convergence is then much faster. We now consider how we can use a recurrence to compute our estimate for e^x.

To develop a recurrence for the terms of the sum, we first write each term as $t_k = x^k/k!$ Then, replacing t by $t-1$, we have:

$$\begin{aligned} t_k &= \frac{x^k}{k!} \\ t_{k-1} &= \frac{x^{k-1}}{(k-1)!} \end{aligned}$$

Taking the quotient of these two equations we have:

$$\frac{t_k}{t_{k-1}} = \frac{x^k}{k!} \div \frac{x^{k-1}}{(k-1)!} = \frac{x^k}{k!} \times \frac{(k-1)!}{x^{k-1}} = \frac{x}{k}$$

Therefore

$$\begin{aligned} e^x &\approx \sum_{k=0}^{n} t_k \\ &\text{where } t_k = \tfrac{x}{k} t_{k-1} \quad k \geq 1 \\ &\text{and} \qquad t_0 = 1 \end{aligned}$$

This is a very simple recurrence which can be used to generate the terms of the sum, up to say 10 or 15 terms. It may be coded, along with some code to test it out in Delphi as follows. A better approach would be to code it as a function, to reduce x to a suitable interval, and perhaps to make the number of terms a function of x. This is left as an exercise.

8.2. Computation of π

We continue with the idea of using recurrences to compute fundamental mathematical functions, or even constants. Perhaps the most famous constant is π, the ratio of any circle 's circumference to its diameter. There is an amazing variety of formulas to compute the first digits of π. In Delphi, π is itself also an intrinsic constant. Of course, the constant π is of fundamental importance in higher mathematics, as well as its relatively humble role in computing circumferences and areas of circles. The number 2π is the period of the circular (trigonometric) functions, and the five most important constants of all, viz., $0, 1, e, i, \pi$ are related via the equation $e^{i\pi} + 1 = 0$.

We give just a few formulas for computing π, code some of them into Delphi and then compare performances. Many more formulas (all infinite sums) are given in [15] and also at the site [58].

8.2.1. The Wallis Product

As well as infinite sums, there are infinite products for π.Perhaps the most famous is the *Wallis product*, given here by eq 8.4.

$$\pi = 2\prod_{n=1}^{\infty} \frac{(2n)^2}{(2n-1)(2n+1)} \tag{8.4}$$

This product is not very good for computation as it converges far too slowly. We should avoid it. We implement it here only for comparison purposes. It is extremely sluggish. Can you see why?

8.2.2. Machin-Like Sums of Arctangents

A very simple infinite sum for π is derived from the fact that $\pi = 4\arctan 1$. Using the infinite series, eq 8.5, for $\arctan x$, we have eq 8.6.

$$\arctan x = \sum_{n=1}^{\infty} \frac{(-1)^{n-1}x^{2n-1}}{2n-1} = x - \frac{1}{3}x^3 + \frac{1}{5}x^5 - \frac{1}{7}x^7 + O\left(x^9\right) \tag{8.5}$$

$$\pi = 4\left(x - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots\right) \tag{8.6}$$

The formula in eq 8.6 also has very poor convergence properties. We should avoid it, too. However, with a little algebra, we can find some very much better ones. Consider the formula of Machin, eq 8.7.

$$\frac{\pi}{4} = 4\arctan\frac{1}{5} - \arctan\frac{1}{239} \tag{8.7}$$

We use the $\arctan x$ intrinsic in Delphi to evaluate π, using eq 8.7.

8.2.3. Ramanujan's Sum

We compare both of these with an approximation using the infinite series of Ramanujan given by eq 8.8.

$$\frac{1}{\pi} = \sum_{n=0}^{\infty} \left(\frac{\binom{2n}{n}}{16^n}\right)^3 \frac{42n+5}{16} \tag{8.8}$$

In order to estimate $1/\pi$ using eq 8.8, we develop a recurrence for the general term, which we denote by c_n; that is,

$$c_n = \left(\frac{\binom{2n}{n}}{16^n}\right)^3 \frac{42n+5}{16}$$

Proceeding as before, we write the corresponding expression for c_{n-1}.

$$c_{n-1} = \left(\frac{\binom{2n-2}{n-1}}{16^{n-1}} \right)^3 \frac{42n-37}{16}$$

Since the formulas for c_n and c_{n-1} both have lots of powers and factorials, taking the quotient leads to lots of cancellation. We have:

$$\begin{aligned}
\frac{c_n}{c_{n-1}} &= \left(\frac{\binom{2n}{n}}{\binom{2n-2}{n-1}} \right)^3 \times \left(\frac{16^{n-1}}{16^n} \right)^3 \times \frac{42n+5}{42n-37} \\
&= \left(\frac{(2n)!\,(n-1)!\,(n-1)!}{n!n!\,(2n-2)!} \right)^3 \times \frac{1}{16^3} \times \frac{42n+5}{42n-37} \\
&= \left(\frac{2n\,(2n-1)}{n^2} \right)^3 \times \frac{1}{16^3} \times \frac{42n+5}{42n-37} \\
&= \frac{1}{512} \left(2 - \frac{1}{n} \right)^3 \frac{42n+5}{42n-37}
\end{aligned}$$

Thus, we have eq 8.9. Notice the massive amount of cancellation that led to it and the comparative simplicity of this final recurrence.

$$c_n = \left(2 - \frac{1}{n} \right)^3 \frac{42n+5}{42n-37} \frac{c_{n-1}}{512} \tag{8.9}$$

If we regard the cube in eq 8.9 as just two multiplications, we have nothing here but the four operations of arithmetic—no complex powers, factorials or exponentials.

8.3. Bernoulli Numbers

The previous subsections used various series to show how the powerful recurrence technique may be used to compute common mathematical functions or constants. A somewhat more challenging exercise along similar lines is to compute the so-called *Bernoulli numbers* b_m for $0 \leq m \leq M$. These numbers are important in many areas of mathematical investigation, including the derivation of compact formulas for sums of powers [8]. First we define $b_0 = 1$ and thereafter, for $m > 0$, b_m is defined by the recurrence of eq 8.10.

$$\sum_{j=0}^{m} \binom{m+1}{j} b_j = 0 \tag{8.10}$$

In eq 8.10, the factor $\binom{m+1}{j}$ is the usual binomial coefficient. For $0 \leq j \leq m$, it is given by:

$$\binom{m+1}{j} = \frac{(m+1)!}{j!\,(m+1-j)!} \tag{8.11}$$

Solving eq 8.10 for b_m, we have

$$b_m = -\sum_{j=0}^{m-1} \frac{1}{m+1}\binom{m+1}{j} b_j \tag{8.12}$$

We may now develop a recurrence for the coefficient of b_j in the sum of eq 8.12. Writing

$$c_j^{(m)} = \frac{1}{m+1}\binom{m+1}{j}$$

we have

$$c_{j-1}^{(m)} = \frac{1}{m+1}\binom{m+1}{j-1}$$

so that

$$\frac{c_j^{(m)}}{c_{j-1}^{(m)}} = \frac{\binom{m+1}{j}}{\binom{m+1}{j-1}} = \frac{m+2-j}{j} \tag{8.13}$$

Clearing fractions on the left side of eq 8.13, we now write

$$c_j^{(m)} = \frac{m+2-j}{j} c_{j-1}^{(m)} \tag{8.14}$$

This means that we can compute $c_j^{(m)}$ from the recurrence of eq 8.14. Delphi code for computing e^x, π and the Bernoulli numbers follows.

```
unit Unit1;
interface uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    Memo1: TMemo;
    BernoulliButton: TButton;
    ExitButton: TButton;
    ExpButton: TButton;
    PiRamanujanButton: TButton;
    ClearButton: TButton;
    PiButton: TButton;
    PiButtonEq5: TButton;
    WallisButton: TButton;
    procedure BernoulliButtonClick(Sender: TObject);
    procedure ExitButtonClick(Sender: TObject);
    procedure ExpButtonClick(Sender: TObject);
    procedure PiRamanujanButtonClick(Sender: TObject);
    procedure ClearButtonClick(Sender: TObject);
    procedure PiButtonClick(Sender: TObject);
    procedure PiButtonEq5Click(Sender: TObject);
    procedure WallisButtonClick(Sender: TObject);
  end;
var
  Form1: TForm1;
```

```
implementation
{$R *.dfm}
const tab = chr(9);
type float = double;

procedure TForm1.BernoulliButtonClick(Sender: TObject);
const mmax = 40;
var m, j : integer; c, b : array[0..mmax] of double;
sum : double;
begin
  memo1.lines.add('m' + tab + 'B[m]');
  b[0] := 1.0; b[1] := -0.5;
  for m := 2 to mmax do begin
    c[0] := 1/(m + 1);
    sum := c[0]; // this is the zeroth term c[0]*b[0]
    for j := 1 to m - 1 do begin
      c[j] := c[j - 1]*(m + 2 - j)/j;
      sum := sum + c[j]*b[j]
    end;
    b[m] := -sum;
    if not odd(m) then
      memo1.lines.add(inttostr(m) + tab + floattostr(b[m]))
  end;
    memo1.lines.add('')
end;

procedure TForm1.ExitButtonClick(Sender: TObject);
begin
  close
end;

procedure TForm1.ExpButtonClick(Sender: TObject);
var k, z : integer; x, term, sum, diff : double; s : string;
begin
  s := 'x' + tab + 'exp(x) approx' + tab + 'exp(x) exact' + tab + 'diff';
  memo1.lines.add(s);
  for z := 10 to 20 do begin
    x := z; x := x/10.0;
    term := 1.0;
    sum := term;
    for k := 1 to 10 do begin
      term := term*x/k;
      sum := sum + term;
    end;
    diff := sum - exp(x);
    s := floattostr(x) + tab + floattostr(sum);
    s := s + tab + floattostr(exp(x)) + tab + floattostr(diff);
    memo1.lines.add(s);
  end
end;

procedure TForm1.PiRamanujanButtonClick(Sender: TObject);
const nmax = 10;
var n : integer; p, term, sum, error : extended; s : string;
begin
```

```
  memo1.lines.add('#terms' + tab + 'pi estimate' + tab + tab + 'error');
  term := 5.0/16.0;
  sum := term;
  for n := 1 to nmax do begin
    term := term*(42*n + 5)/(42*n - 37)/512;
    term := term*(2 - 1/n)*(2 - 1/n)*(2 - 1/n);
    sum := sum + term;
    p := 1/sum;
    error := p - pi;
    s := floattostr(n) + tab + floattostrf(p, ffgeneral,22,20) +
      tab + floattostr(error);
    memo1.lines.add(s)
  end
end;

procedure TForm1.ClearButtonClick(Sender: TObject);
begin
  memo1.clear
end;

procedure TForm1.PiButtonClick(Sender: TObject);
// Salamin-Brent
var a, b, c, t, s, p, temp, error : extended; k : integer;
msg : string;
begin
  k := 0;
  a := 1.0;
  b := 1.0/sqrt(2.0);
  s := 0.5;
  t := 1.0;
  repeat
    inc(k);
    t := 2*t; // assert t = 2^k
    temp := (a + b)/2; // temp is new value of a
    b := sqrt(a*b);
    a := temp;
    c := a*a - b*b;
    s := s - t*c;
    p := 2*a*a/s;
    error := p - pi;
    msg := floattostr(k) + tab + floattostrf(p, ffgeneral,22,20);
    msg := msg + tab + floattostr(error);
    memo1.lines.add(msg)
  until abs(error)< 1.0e-16
end;

procedure TForm1.PiButtonEq5Click(Sender: TObject);
var i, j : integer; z, sum, f, error : extended;
msg : string;
begin
  i := 0;
  z := 1.0; // z is 16^(-i)
  sum := 0.0;
  repeat
    j := 8*i;
```

```
    f := 4/(j + 1) − 2/(j + 4) − 1/(j + 5) − 1/(j + 6);
    sum := sum + z∗f;
    error := sum − pi;
    msg := floattostr(i) + tab + floattostrf(sum, ffgeneral,22,20);
    msg := msg + tab + floattostr(error);
    memo1.lines.add(msg);
    inc(i);
    z := z/16.0
  until abs(error)< 1.0e−16
end;

procedure TForm1.WallisButtonClick(Sender: TObject);
var twon : integer; factor, product, error : extended;
msg : string;
begin
  product := 2.0;
  twon := 0;
  msg := 'n' + tab + 'pi estimate' + tab + tab + 'error';
  memo1.lines.add(msg);
  repeat
    twon := twon + 2;
    factor := sqr(twon)/(twon∗twon − 1);
    product := product∗factor;
    error := product − pi;
    msg := floattostr(twon div 2) + tab +
      floattostrf(product, ffgeneral,22,20);
    msg := msg + tab + floattostr(error);
    memo1.lines.add(msg);
  until twon = 500
end;

end.
```

8.3.1. Mutual Recursion

We consider further application of recurrences to generate sequences. The difference here is that two intertwined sequences are required. These are defined by so-called *mutual recurrences*. We illustrate the ideas by an example from calculus. A situation in which such recurrences occur frequently is in elementary integral calculus. It is here that we often encounter integrals such as:

$$I_n = \int_0^{\infty} x^n e^{-x} \tag{8.15}$$

or

$$J_n = \int_0^{2\pi} x \sin nx dx \tag{8.16}$$

or perhaps

$$K_n = \int_0^{\pi} x^n \sin x dx \tag{8.17}$$

The common features of these are that they contain an integer parameter, usually denoted n. In this chapter we consider the evaluation of certain trigonometric integrals using

mutual recursion and compare this approach with alternative techniques which make use of arrays and simple iteration. The integrals we study are defined by eqs 8.18 and 8.19.

$$s_n = \int_0^{\pi} x^n \sin x dx \qquad n \geq 0 \tag{8.18}$$

$$c_n = \int_0^{\pi} x^n \cos x dx \qquad n \geq 0 \tag{8.19}$$

The foregoing integrals are defined for all integers $n \geq 0$. Taking $n \geq 1$, and using integration by parts on the indefinite integral corresponding to eq 8.18, we have:

$$\begin{aligned} \int x^n \sin x dx &= \int x^n d(-\cos x) \\ &= -x^n \cos x + n \int x^{n-1} \cos x dx \end{aligned}$$

Applying the limits of integration, and noting the definition of c_n from eq 8.19, we have:

$$\begin{aligned} s_n &= \int_0^{\pi} x^n \sin x dx \\ &= [-x^n \cos x]_0^{\pi} + n \int_0^{\pi} x^{n-1} \cos x dx \\ &= -\pi^n \cos \pi + n c_{n-1} \\ &= \pi^n + n c_{n-1} \end{aligned}$$

Similar steps for eq 8.19 are:

$$\begin{aligned} \int x^n \cos x dx &= \int x^n d(\sin x) \\ &= x^n \sin x - n \int x^{n-1} \sin x dx \end{aligned}$$

Applying the limits of integration, and noting the definition of s_n from eq 8.18, we have:

$$\begin{aligned} c_n &= \int_0^{\pi} x^n \cos x dx \\ &= [x^n \sin x]_0^{\pi} - n \int_0^{\pi} x^{n-1} \sin x dx \\ &= 0 - n s_{n-1} \\ &= -n s_{n-1} \end{aligned}$$

Finally, in order to use the recurrences for computation, we need initial values. these are easily obtained by substituting $n = 0$ in eqs 8.18 and 8.19:

$$s_0 = \int_0^{\pi} x^0 \sin x dx = \int_0^{\pi} \sin x dx = 2 \tag{8.20}$$

$$c_0 = \int_0^{\pi} x^0 \cos x dx = \cos x dx = 0 \tag{8.21}$$

In summary, we have:

$$s_n = \pi^n + nc_{n-1}; \qquad n \geq 1 \tag{8.22}$$

$$s_0 = 2 \tag{8.23}$$

$$c_n = -ns_{n-1}; \qquad n \geq 1 \tag{8.24}$$

$$c_0 = 0 \tag{8.25}$$

Our task here is to write efficient code to evaluate the two sequences (s_n) and (c_n) for small to moderately large values of n. Since the sequences are defined recursively, it is natural, perhaps, to write recursive code. Noting that the term π^n may also be defined recursively, it is rather straightforward to develop the following recursive solution in Delphi.

The standard forward declaration is required because we need an exception to the usual Pascal (Delphi) rule that all objects must be declared before the first use or reference. Since Pascal is designed to be compilable in one-pass of the source text, the statement simply tells the compiler what kind of thing *cc* is, so that it may be checked both when referenced and defined later in the code.

Delphi Code

```
unit mutrecur;
interface uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Buttons;
type
  TForm1 = class(TForm)
    Memo1: TMemo;
    GenerateButton: TButton;
    ExitButton: TButton;
    DumpButton: TButton;
    RecursiveGenerationButton: TButton;
    Button1: TButton;
    procedure ExitButtonClick(Sender: TObject);
    procedure GenerateButtonClick(Sender: TObject);
    procedure DumpButtonClick(Sender: TObject);
    procedure RecursiveGenerationButtonClick(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  end;

var
  Form1: TForm1;

implementation
const nmax = 10;
var s,c,p:array[0..nmax] of double;
{$R *.DFM}

procedure TForm1.ExitButtonClick(Sender: TObject);
begin
  close
end;
```

```
procedure TForm1.GenerateButtonClick(Sender: TObject);
var n:integer;
begin
  c[0] := 0.0; s[0] := 2.0; p[0] := 1.0;
  for n := 1 to nmax do p[n] := pi*p[n - 1];
  for n := 1 to nmax do begin
    s[n] := p[n] + n*c[n - 1];
    c[n] := -n*s[n - 1]
  end
end;

procedure TForm1.DumpButtonClick(Sender: TObject);
var n:integer; msg:string;
begin
  memo1.lines.add('n' + #9 + 'c[n]' + #9 + 's[n]');
  for n:=0 to nmax do begin
    msg := inttostr(n) + #9;
    msg := msg + floattostrf(c[n],fffixed,15,5) + #9;
    msg := msg + floattostrf(s[n],fffixed,15,5);
    memo1.lines.add(msg)
  end
end;

procedure TForm1.RecursiveGenerationButtonClick(Sender: TObject);
var n:integer;

  function cc(n : integer) : double; forward;

  function pp(n : integer) : double;
  begin
    if n = 0 then result := 1
    else result := pi*pp(n - 1)
  end;

  function ss(n : integer) : double;
  begin
    if n = 0 then result := 2.0
    else result := pp(n) + n*cc(n - 1)
  end;

  function cc(n : integer) : double;
  begin
    if n = 0 then result := 0.0
    else result := -n*ss(n - 1)
  end;

begin
  for n := 0 to nmax do begin
    p[n] := pp(n);
    c[n] := cc(n);
    s[n] := ss(n)
  end
end;

procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
  memo1.clear
end;

end.
```

Comments on the Two Implementations

Note that two approaches to recursion have been used. The recursive code just shown is logically correct, simply by inspection. Unfortunately, it has a number of very serious shortcomings, foremost of which is gross inefficiency. In essence, the inefficiency arises from *recomputing the same quantities over and over again* without making use of them to evaluate subsequent elements of the two sequences. The second approach uses arrays to store the generated sequence elements. This method is in fact iterative, but certainly based on the mathematical recurrences. Note the compactness of the code. It is also worth noting that the iterative algorithm represented by this code may be easily implemented in a spreadsheet such as Excel.

Definition and Implementation

Here, we have a very clear example of the principle of trading memory for computing time. In essence, with extra storage, we can often reduce computation time. This is very definitely the case here. The simple expedient of not only storing the computed values in two arrays, but also making use of them for intermediate computations, saves us an enormous amount of computation. Thus, while correct, the fully recursive solution previously presented is a very good example of how *not* to solve this problem. In fact it illustrates a very important general principle: *the separation of definition from implementation*. Just because something is defined recursively does not mean it must be computed recursively.

Recursion is, without doubt, a very powerful concept indeed, both from the mathematical and computing viewpoints. From the mathematical standpoint, it is very closely related to the powerful proof technique of mathematical induction. This has widespread application in computer science, including algorithm correctness proofs and certain considerations of algorithm complexity. In many practical instances, recursively-defined sequences are better computed by using iteration, usually with arrays to store the generated terms, which may then be easily re-used to generate further terms. This rule of thumb has become increasingly valid as the cost of main memory has plummeted in recent years, and it has become common to have desktop PCs or laptops with several gigabytes of RAM.

8.4. Catalan Numbers

In this section we consider various means of computing the well-known *Catalan numbers*. This sequence of natural numbers may be defined by at least two quite different , as well as by a direct formula. Again, one of the main points we wish to make in this chapter is that many sequences defined by direct formulae are most easily computed via recurrence

relations. Just as with the sequences of Chapter 8, this does not mean that explicit recursion need be used to compute them. Indeed, it is often better to avoid this approach. What is meant is that, even if given a direct formula, a (simpler) recurrence can sometimes be found, and this then used with iteration, and sometimes arrays, may be used to very efficiently compute the sequences. Such an approach may be thought of as a well-designed combination of processing power and memory. The dictum that extra memory may be used to make an algorithm run faster certainly applies in the computation of sequences of numbers. It applies especially to the efficient computation of the Catalan numbers.

Definition 6 *The order of a recurrence relation (formula) is the difference between the highest and lowest values of the sequence index which appear in the relation. For example* $x_n = 2x_{n-1}$ *has order 1 (first-order), while* $a_m + 2a_{m-1}^2 - 5a_{m-2} = 0$ *is of order 2 (second-order). The Catalan recurrence just below is of variable order, as* x_n *depends on the entire history, from* x_0 *to* x_{n-1}.

Definition 7 *Given an integer,* $n \geq 0$*, we define the nth Catalan number,* x_n*, to be the number of topologically distinct binary trees with n nodes. Using this definition, we obtain,* $x_0 = 0$*, and by recursive reasoning, for* $n > 0$*:*

$$x_n = \sum_{r=0}^{n} x_r x_{n-r-1} \tag{8.26}$$

8.4.1. Applications of Catalan Numbers

Catalan numbers have application to enumeration of binary trees, parenthesizing arithmetic expressions, and elsewhere. A useful, modern introduction to this topic may be found, for example, in [35]. Just as strong induction is logically equivalent to ordinary induction, we transform a "variable order" recurrence (x_n apparently depends on all its predecessors) into one of first order (x_n depends on just its immediate predecessor). We observe that a recurrence with fixed initial conditions yields only one sequence, whereas any given sequence may be generated from arbitrarily many recurrences. Therefore, we have an *equivalence class* of recurrences: all generate the same sequence (Catalan numbers). A similar situation arises in the theory of Boolean expressions, where we usually seek one member of an equivalence class which is in some sense canonical or simplest. This is clearly related to simplicity of computer implementation in a modern conventional 3GL such as Delphi, or even a spreadsheet such as Microsoft Excel.

Next, we note that in order to find alternative recurrence relations, some mathematics (i.e., algebra) has to be done. For the Catalan recurrence, one may use the standard theory of generating functions to derive a simple closed form expression (direct formula) for the sequence.

This direct formula may now be used to derive a much simpler recurrence which is easy to implement in a spreadsheet such as Excel, and indeed, much more efficiently computable in a 3GL such as Delphi. More conventional recurrences, such as linear, second order are very easy to implement in a spreadsheet such as Excel. However, the defining recurrence for Catalan numbers given by eq 8.26 is not easily realized in Excel, or any other conventional

spreadsheet program, certainly not without use of Visual BASIC, which is just another 3GL anyway.

The fact the Catalan recurrence is hard to implement in Excel may give some clue as to its unsuitability for naive computational implementation in a conventional 3GL. If a naive recursive implementation is attempted in Delphi (or any 3GL supporting recursion), it is unimaginably inefficient. If one uses an array to store the sequence elements, then the recursive definition of eq 8.26 is perfectly acceptable as a guide to implementation. This example illustrates two important programming and mathematical principles. Firstly, we have, yet again, an instance of the ubiquitous principle of trade-off of memory and computation time. The time for computation is simply intolerable for the naive recursive implementation, but quite acceptable if an array is used to store already generated elements, and then either recursion or iteration used to obtain the next element in the sequence. Secondly, there is a much simpler recurrence (first order, in fact) available to compute the Catalan numbers.

8.4.2. Direct Formula for Catalan Numbers

As noted, by using the theory of *generating functions*, a direct formula for x_n is available. This work is beyond our scope here, but see, for example, [46]. The formula, valid for all $n \geq 0$ is given by eq 8.27. It may be written as explicit factorials, or in terms of a .

$$x_n = \frac{(2n)!}{n!\,(n+1)!} = \frac{1}{n+1}\binom{2n}{n} \tag{8.27}$$

How is this to be computed? It is not at all obvious that x_n, as defined by eq 8.27, will be integral for arbitrary integer $n \geq 0$. The numbers are all integers, and the numerator $(2n)!$ of eq 8.27 is huge, even for modest values of n. It will overflow if computed naively with integer variables in a conventional 16-bit 3GL for extremely small values of n; $n > 3$, to be precise. The denominator is similarly large. Once again, we need a recurrence, which is, at least in the case of first order, essentially a discrete derivative.

Another recurrence

Simply considering the ratio of successive terms in eq 8.27 yields, for $n \geq 1$:

$$\frac{x_n}{x_{n-1}} = \frac{(2n)!}{n!\,(n+1)!} \times \frac{n!\,(n-1)!}{(2n-2)!} \tag{8.28}$$

$$= \frac{(2n)!}{(2n-2)!} \times \frac{(n-1)!}{(n+1)!} \tag{8.29}$$

$$= \frac{2n(2n-1)}{1} \times \frac{1}{n(n+1)} \tag{8.30}$$

$$= \frac{4n-2}{n+1} \tag{8.31}$$

We end up with:

$$x_n = \left(\frac{4n-2}{n+1}\right) x_{n-1} \tag{8.32}$$

Note that this last recurrence is first order, linear and homogeneous! It is trivial to compute the Catalan numbers in Excel, in a 3GL such as Delphi, or even with simple calculator

using this recurrence. The lesson from this is that some mathematical insight may transform an intractable computation into something that is quite trivial. Thinking must precede computation, and mathematics is our systematic and precise way of thinking.

8.4.3. Asymptotics

Another benefit of eq 8.32 is that the asymptotic behaviour of x_n may be seen as roughly $x_n = O(4^n)$. We have, from eq 8.32

$$x_n = \left(\frac{4n-2}{n+1}\right) x_{n-1} = \left(4 - \frac{6}{n+1}\right) x_{n-1} \tag{8.33}$$

If not for the negative term $-6/(n+1)$, eq 8.33 would not just be approximately asymptotically 4^n, it would be *exactly* 4^n. This diminishing term tells us to expect that x_n is sub-asymptotic to 4^n. The real truth is given by the relationship (8.34). This means that x_n divided by the expression on the right approaches 1 as $n \to \infty$. Equivalently, $\ln(x_n) - n\ln 4 + 1.5\ln n + 0.5\ln\pi$ should approach zero. This may be obtained by taking logarithms of the quotient of the left and right sides of (8.34).

$$x_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}} \tag{8.34}$$

It is clear from eq 8.33, by induction on n, and the fact that $x_0 = 1 = 4^0$, that x_n must always be less than 4^n. This knowledge may be useful to programmers in order to estimate when overflow might occur. The asymptotic behaviour is also obtainable with considerable extra effort from eq 8.27 by use of Stirling's formula [9] and [54], but, once again, a knowledge of mathematics, not programming, is needed here. A simple Microsoft Excel model implementing eq 8.32 is shown in Figure 8.2 and the corresponding chart in Figure 8.1. An interesting aspect of the Catalan sequence which may be observed is that x_n does not overtake 3^n until n reaches 17.

8.4.4. Delphi Code to Compute Catalan Numbers

There is nothing too remarkable about the code, save for one point. When computing x_n from x_{n-1} using (8.32), it is necessary to do the (integer) division last. If this is not done, then some truncation may occur and then a false result will ensue.

```
unit Unit1;
interface uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    Memo1: TMemo;
    RunButton: TButton;
    ExitButton: TButton;
    ClearButton: TButton;
    procedure ExitButtonClick(Sender: TObject);
    procedure RunButtonClick(Sender: TObject);
    procedure ClearButtonClick(Sender: TObject);
```

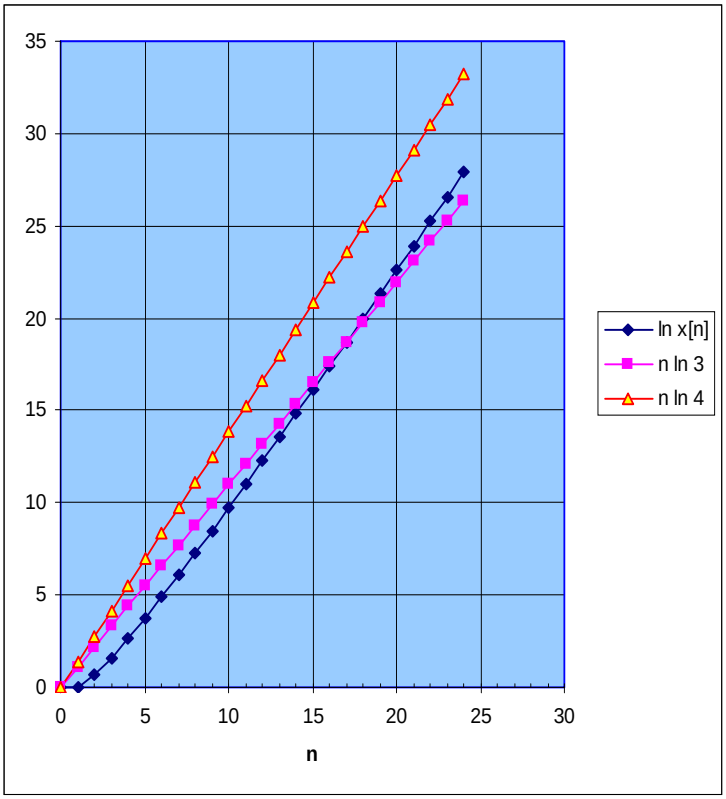

Figure 8.1. Graph of Catalan numbers versus exponentials.

```
  end;
var
  Form1: TForm1;

implementation
{$R *.dfm}

procedure TForm1.ExitButtonClick(Sender: TObject);
begin
  close
end;

procedure TForm1.RunButtonClick(Sender: TObject);
const nmax = 20; tab = chr(9);
var n : integer; x : int64; msg : string; z : extended;
begin
  n := 0;
  x := 1; // x[0] = 1
  msg := 'n' + tab + 'x[n]' + tab + tab + 'ln(x[n])' + tab;
  msg := msg + 'n*ln(3)' + tab + 'n*ln(4)';
  memo1.lines.add(msg);
  repeat
    msg := inttostr(n) + tab + inttostr(x);
```

n	x[n]	3^n	x[n] - 3^n	ln x[n]	n ln 3	n ln 4
0	1	1	0	0	0	0
1	1	3	- 2	0	1.099	1.386
2	2	9	- 7	0.693147181	2.197	2.773
3	5	27	- 22	1.609437912	3.296	4.159
4	14	81	- 67	2.63905733	4.394	5.545
5	42	243	- 201	3.737669618	5.493	6.931
6	132	729	- 597	4.882801923	6.592	8.318
7	429	2187	- 1758	6.061456919	7.69	9.704
8	1430	6561	- 5131	7.265429723	8.789	11.09
9	4862	19683	- 14821	8.489205155	9.888	12.48
10	16796	59049	- 42253	9.728896042	10.99	13.86
11	58786	177147	- 118361	10.98165901	12.08	15.25
12	208012	531441	- 323429	12.24535105	13.18	16.64
13	742900	1594323	- 851423	13.51831673	14.28	18.02
14	2674440	4782969	- 2108529	14.79925057	15.38	19.41
15	9694845	14348907	- 4654062	16.08710486	16.48	20.79
16	35357670	43046721	- 7689051	17.3810259	17.58	22.18
17	129644790	129140163	504627	18.68030888	18.68	23.57
18	477638700	387420489	90218211	19.98436515	19.78	24.95
19	1767263190	1162261467	605001723	21.29269797	20.87	26.34
20	6564120420	3486784401	3077336019	22.60488436	21.97	27.73
21	24466267020	10460353203	14005913817	23.92056115	23.07	29.11
22	91482563640	31381059609	60101504031	25.23941423	24.17	30.5
23	343059613650	94143178827	2.48916E+11	26.56117007	25.27	31.88
24	1289904147324	2.8243E+11	1.00747E+12	27.88558903	26.37	33.27

Figure 8.2. Tabulation of Catalan numbers versus exponentials.

```
    z := x; z := ln(z);
    if n < 16 then msg := msg + tab;
    msg := msg + tab + floattostrf(z,ffgeneral,5,3);
    msg := msg + tab + floattostrf(n*ln(3),ffgeneral,5,3);
    msg := msg + tab + floattostrf(n*ln(4),ffgeneral,5,3);
    memo1.lines.add(msg);
    inc(n);
    if n <= nmax then
      x := (4*n - 2)*x div (n + 1)
  until n > nmax
end;

procedure TForm1.ClearButtonClick(Sender: TObject);
begin
  memo1.clear
end;

end.
```

8.5. Exercises

1. Code the Delphi procedure of section 8.1. to use a function to estimate e^x.

2. Euler's constant is defined as

$$\gamma = \lim_{n\to\infty}\left(\left(\sum_{k=1}^{n}\frac{1}{k}\right)-\ln n\right)$$

Develop an algorithm in Delphi to estimate it, correct to 10 decimal places. This value is $\gamma \simeq 0.5772156649$. Convergence will be extremely slow. How to accelerate it? A much more rapidly-converging series is given at [3].

$$\gamma = \sum_{k=1}^{\alpha n}(-1)^{k-1}\frac{n^k}{k\times k!}-\ln n+O\left(e^{-n}\right)$$

In this equation, $\alpha \simeq 3.59$ and is the positive root of $\alpha(\ln\alpha-1)=1$. To determine γ to d decimal places, we require $n \simeq d\ln 10$. Develop a suitable recurrence to compute the sum, implement the formula and compare the value and rate of convergence with the estimate obtained from the definition.

3. The beta $B(m,n)$ function for integer parameters m and n is defined as follows:

$$B(m,n)=\frac{(m-1)!(n-1)!}{(m+n-2)!}$$

Develop suitable recurrence(s) to compute this function for $1 \le m \le n \le 10$. Implement the recurrence in Delphi and extend the program to tabulate the output to a memo box or textfile.

4. Extend the Delphi program for Bernoulli numbers to compute the Bernoulli polynomials $B_n(x)$. These may be defined in various ways. Perhaps the simplest for our purpose is:

$$B_n(x)=\sum_{k=0}^{n}\binom{n}{k}b_k x^{n-k}$$

5. By using integration by parts *or otherwise*, find recurrences for the following integrals, then develop efficient Delphi code to evaluate them.

(**a**)

$$\int_0^\infty x^n e^{-x} \tag{8.35}$$

(**b**)

$$\int_0^1 x^m(1-x)^n\,dx \tag{8.36}$$

(**c**)

$$\int_0^{\pi/2}\sin mx\cos nx dx \tag{8.37}$$

6. Prove eq 8.26.

7. Prove eq 8.27.

8. Use induction to show that $x_n > 3^n$ for $n \geq 17$.

9. Use the recurrence of eq 8.32 to compute the Catalan numbers in Excel for $n \leq 20$.

10. Find the smallest positive integer n for which $x_n \geq 3.5^n$, and then prove that the inequality is also true for all subsequent values of n by induction.

11. Earlier, we said that $\ln(x_n) - n\ln 4 + 1.5\ln n + 0.5\ln\pi$ should approach zero as $n \to \infty$. Extend the Delphi code to check this. Note that this expression cannot be computed for $n = 0$.

12. Write a Delphi program to compute the Catalan numbers up to x_{20} using the defining recurrence, eq 8.26, and an array to store the terms of the sequence. Use iteration, not recursion.