Evaluating Polynomials

Textbook Reading (from *Algorithms: Foundations and Design Strategies)*:

- Section 1.4, pp. 19-21

Evaluating Polynomials

A basic problem with many applications to engineering, mathematics and science is the problem of evaluating a polynomial

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

at a particular value of v of x.

Polynomials approximate differentiable functions

We can use the infinite Tayler series to obtain formula for differential functions, which can then be approximated with polynomials. For example,

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!} + \cdots$$

so we can approximate it with the polynomial

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!}$$

where our choice of *n* depends on how good an approximation we want.

Straightforward solution

function *PolyEvalNaive*(*a*[0:*n*],*v*)

Input: *a*[0:*n*] (an array of real numbers), *v* (a real number)

Output: the value of the polynomial $a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$ at $x = v$

```
    Sum ← a[0]
    Product ← 1
    for i ← 1 to n do
        Sum ← Sum + a[i] * Powers(v,i)
    endfor
    return(Sum)
end PolyEval
```

Analysis of *PolyEvalNaive*

Using an efficient version of *Powers,* which we've analyzed to perform at least $\log_2 n$ multiplications, the number of multiplications performed by *PolyEvalNaive* is at least

$$\begin{aligned}
&\log_2 1 + \log_2 2 + \cdots + \log_2 n \\
&\geq \log_2\left(\left\lfloor \tfrac{n}{2} \right\rfloor + 1\right) + \log_2\left(\left\lfloor \tfrac{n}{2} \right\rfloor + 2\right) + \cdots + \log_2 n \\
&\geq \log_2\left(\tfrac{n}{2}\right) + \log_2\left(\tfrac{n}{2}\right) + \cdots + \log_2\left(\tfrac{n}{2}\right) \\
&\geq \tfrac{1}{2} n \log_2\left(\tfrac{n}{2}\right) = \tfrac{1}{2} n \left(\log_2(n) - 1\right) \approx \tfrac{1}{2} n \log_2 n.
\end{aligned}$$

The number of additions performed is n.

Better Solution

function *PolyEval*(*a*[0:*n*],*v*)

Input: *a*[0:*n*] (an array of real numbers), *v* (a real number)

Output: the value of the polynomial $a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$ at $x = v$

```
    Sum ← a[0]
    Product ← 1
    for i ← 1 to n do
        Product ← Product * v
        Sum ← Sum + a[i] * Product
    endfor
    return(Sum)
end PolyEval
```

Analysis of *PolyEval*

PolyEval clearly does

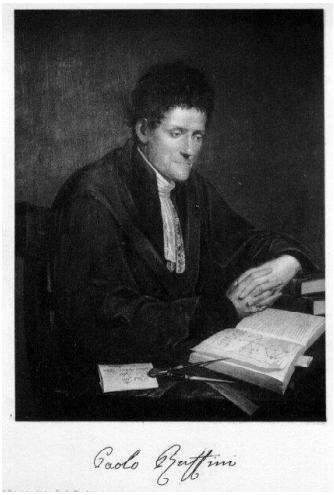
$2n$ multiplications

n additions.

This might seem the best that we can do. But it isn't. We can do much better.

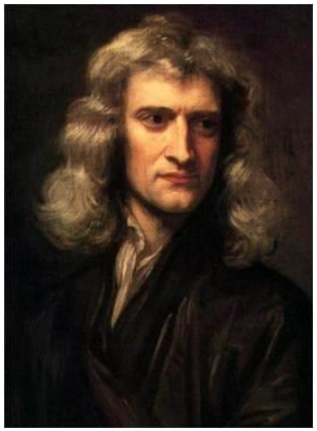
Horner's rule

There is a simple algorithm for polynomial evaluation that cuts the number of multiplications performed by *PolyEval* in half. This algorithm goes under the name of Horner's rule, since W. G. Horner popularized the method in 1819.



But the algorithm was already known to Paolo Ruffini in 1809 (Ruffini's rule)

to Isaac Newton in 1669



and the Chinese mathematician Zhu Shijie in the 14th century.

Example

A fourth-degree polynomial

$$a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

is rewritten as

$$((a_4*x + a_3)*x + a_2)*x + a_1)*x + a_0$$

This rewriting yields the algorithm known as Horner's rule.

Horner's rule

function *HornerEval*(*a*[0:*n*],*v*)

Input: *a*[0:*n*] (an array of real numbers), *v* (a real number)

Output: the value of the polynomial $a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$ at $x = v$

```
    Sum ← a[n]
    for i ← n – 1 downto 0 do
        Sum ← Sum * v + a[i]
    endfor
    return(Sum)
end HornerEval
```

Analysis of *HornerEval*

HornerEval performs

n multiplications

n additions

It can be proved that any algorithm for evaluating an *n*th-degree polynomial that only uses multiplications or divisions and additions or subtractions must perform at least *n* multiplications or divisions and at least *n* additions or subtractions. Hence, *HornerEval* is an optimal algorithm for evaluating polynomials.

Hope you had a good breakfast this morning.
What type of bagel can fly?

Answer:

a plain bagel