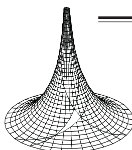


6/1 (2008), 173–185
tmcs@inf.unideb.hu
http://tmcs.math.klte.hu
Teaching Mathematics and Computer Science

Fibonacci beyond binary recursion

Michael A. Wirth

Abstract. The Fibonacci series is a classical algorithm taught in computer science, usually implemented in some programming language. It is hard to find a programming textbook which doesn't touch on Fibonacci, and it's most common use is in the illustration of binary recursion. There are also many ways of tailoring the basic algorithm in order to implement it. This paper discusses some novel algorithms, which help address some of the limitations of binary recursion, but also illustrate how differing algorithms can be pedagogically beneficial. We introduce a simple algorithm for accurately calculating any Fibonacci number.

Key words and phrases: Fibonacci, algorithms, binary recursion, accurate algorithms.

ZDM Subject Classification: M50, D40, I30.

1. Introduction

The Fibonacci numbers were conceived by European mathematician Leonardo of Pisa (1175–1250) who was called Fibonacci (fib-on-arch-ee), short for Filius Bonacci, "the son of Bonaccio", since his father's name was Guglielmo Bonacci. In 1202 Fibonacci wrote *Liber Abbaci*, or "The Book of the Abacus", which was to contain one of his most infamous problems: *paria coniculorum* — the rabbit problem. Suppose a newly-born pair of rabbits, one male, one female, are put in a field. Rabbits are able to mate at the age of one month so that at the end of its second month a female can produce another pair of rabbits. Suppose that our rabbits never die and that the female always produces one new pair (one male, one female) every month from the second month on. The puzzle that Fibonacci

Copyright © 2008 by University of Debrecen

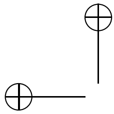
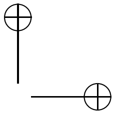
posed was: "How many pairs will there be in one year?". The answer to his problem involves a series of numbers:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89$$

to which his name has been applied. To Leonardo, this sequence was of little importance, as the central focus of *Liber Abbaci* was the introduction of the Hindu-Arabic numeric system. To the uninitiated, Fibonacci numbers appear as a progression of numbers with no real significance. However Fibonacci numbers appear in nature in a number of unexpected ways. The way in which the spiral patterns of sunflower seeds and pine cones grow is described by the sequence, and it is common for the number of petals on a flower to be a Fibonacci number. Four-leaved clovers are less common than three-leaved ones because three is in the Fibonacci sequence and four isn't! The arrangement of structures such as leaves around a stem, scales on a pine cone or on a pineapple, florets in the head of a daisy, and seeds in a sunflower are examples of an aspect of plant form known as *phyllotaxis*. Around the turn of the 18th century the well known Astronomer Johanne Kepler observed that the Fibonacci numbers are common in plants. The number of petals in many flowers, such as the daisy, can be represented as a Fibonacci number [1]. An iris has 3 petals, buttercups have 5, some delphiniums have 8, corn marigolds have 13, asters have 21, daises have 34, 55 or 89 petals. The pattern of leaves as they spiral up a stem, or Fibonacci phyllotaxis, affords optimal illumination to the photosynthetic surface of plants, since it allows for the least amount of overlap [2]. For example in phyllotaxis ratio revolutions/(leafs:buds) apple, apricot and cherry trees have a 2/5 ratio, a pear has 3/8, and an almond 5/13.

The number of spiral rows of fruitlets (eyes) in pineapples was studied as early as 1933 in an article by Linford [3] published in *The Pineapple Quarterly*, however no reference was made to Fibonacci numbers. In a follow-up study by Onderdonk [2] in 1970 it was found that the majority of pineapples had 8–13–21 rows of fruitlets. This was important as it was thought that a pineapple with more fruitlets for a given size would have a finer texture. One other place that plant life reflects the Fibonacci sequence is in the seed patterns of pinecones and sunflowers. The most perfect example of this is the common sunflower, *Helianthus annuus* [4], where two sets of spirals are present. In pinecones, scales are arranged in helical whorls, e.g. eight rows winding in one direction, with five in the other [5].

Outside nature, the Fibonacci series has applications in diverse fields including the generation of musical compositions [6], colour selection in planning a painting [7], and the conceptualization of Minoan architecture in Crete circa

1400–2000 BC [8]. The latter application is a prime example of the role of Fibonacci in the calculation of *golden ratios*.

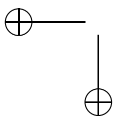
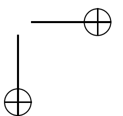
2. The Pedagogy of Fibonacci

Open any textbook on programming or algorithm analysis and you are sure to find some implementation of the Fibonacci algorithm. More often than not it is the archetypal example of binary recursion. Yet despite this, the Fibonacci algorithm is scarcely used in computer science to its fullest potential. It is the type of algorithm that due to its innate simplicity can be used in first year courses to introduce the concept of an algorithm. The algorithm can then be used to illustrate loops, demonstrate recursion, and make a case for modularization. In this capacity it offers us what we shall term a *cascading* case study. This is essentially a case study which can be used in a sequential manner, building upon the algorithm as new programming concepts are introduced. The various algorithms can also be used to demonstrate algorithm complexity. Although this may seem excessive in an introductory course, it gives students an understanding of how different algorithms behave. Lastly it provides an insight into the mathematical nuances of programming: generation of large integers, and overflow precision issues with the use of doubles. The algorithms described in this article were all written in C, and tested on a machine with a dual Pentium D 2.8GHz processor.

3. The Basic Algorithm

The characteristic of this sequence is that each number is the sum of its two immediate predecessors. In other words, the Fibonacci numbers f_n are generated by the simple recurrence:

$$\begin{aligned} f_1 &= 1, \\ f_2 &= 1, \\ f_n &= f_{n-1} + f_{n-2}, \quad n > 2 \end{aligned}$$

4. Why Not Binary Recursion?

The original formula lends itself to a natural, if somewhat naÏve, example of binary recursion, which is probably the most notorious implementation found in the literature. The algorithm works by returning one if $n = 1$ or 2, and the sum of f_{n-1} and f_{n-2} if $n > 2$. This algorithm certainly generates the correct answer, but what is its computational cost, i.e. how long does it take? If n is less than two, the function halts almost immediately. However for larger values of n, there are two recursive calls of the algorithm. This implies that the running time of the algorithm grows at exponential time, or $f_n = 2^{0.694n}$, which for $n = 200$ is 2^{140} operations. On the IBM Blue Gene/P, which has a speed of 10^{15} floating-point operations per second, the calculation of f_{200} would take at least 2^{92} seconds. To put this into context, on a cosmic scale, the universe has existed for approximately 2^{59} seconds, so calculation of f_{200} is inherently prohibitive. The biggest problem with using binary recursion to calculate the Fibonacci numbers is the time spent re-calculating already calculated Fibonacci numbers. For example, when calculating f_{40} using binary recursion, f_{39} is calculated once, f_{35} is calculated eight times, f_0 is calculated 165 580 141 times, for a total of 331 160 281 function calls. The calculation of f_{40} actually takes approximately 34 seconds. This is an interesting analysis, rarely made in textbooks. Few textbooks discuss alternatives to binary recursion for Fibonacci. Indeed, the use of Fibonacci numbers to illustrate binary recursion is a good example of when *not* to use recursion.

```
int fib_BinaryR(int n)
{
    if (n <= 2)
        return 1;
    else
        return fib_BinaryR(n-1) + fib_BinaryR(n-2);
}
```

5. Classic Algorithms for Fibonacci

The next most common algorithm makes allowances for the fact that efficiency isn't always the major protagonist in algorithm design. If an algorithm is slower, you can always run it for longer, and wait for the result. However, there is usually finite memory in which programs can run, so space becomes an issue in algorithm

design. Consider the recursive process outlined above. The space used by a recursive algorithm is the total space used by all recursive calls at a particular time. This includes space for variables, and function arguments, and overhead space for each call. An iterative algorithm incorporates some form of looping structure. Given the values of the first two Fibonacci numbers, a loop is used to calculate the $n-2$ remaining numbers. Each step through the loop uses only the previous two values of f_n which requires some swapping around of values so that everything stays in the appropriate places. So starting with f_1=1 and f_2=1, we compute each successive f_i such that $2 < i \leq n$ by adding f_{i-2} and f_{i-1}.

```
int fib_Iterative(int n)
{
    int f1 = 1, f2 = 1, f;
    for (int i = 3; i <= n; i=i+1) {
        f = f1 + f2;
        f1 = f2;
        f2 = f;
    }
    return f;
}
```

The one caveat with this approach is that there is no way of storing all the numbers. The next logical leap from this is to use a *dynamic programming* approach using an array to register previous results, rather than recomputing them.

```
int fib_Dynamic(int n)
{
    int f[n];
    f[0] = f[1] = 1;
    for (int i = 2; i < n; i=i+1)
        f[i] = f[i-1] + f[i-2];
    return f[n];
}
```

There is also a simple approach which uses a matrix [9]:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{bmatrix} \tag{1}$$

```
int fib_Matrix(int n)
{
    int i;
    int m[2][2] = {{1,1},{1,0}};
    int t[2][2];

    for (i=1; i<n; i=i+1) {
        t[0][0] = m[0][0];
        t[0][1] = m[0][1];
        t[1][0] = m[1][0];
        t[1][1] = m[1][1];
       m[0][0] = t[0][0] + t[0][1];
         m[0][1] = t[0][0];
         m[1][0] = t[1][0] + t[1][1];
         m[1][1] = t[1][0];
    }
    return m[0][0];
}
```

There are of course certain algorithmic limitations to this conventional approach to calculating Fibonacci numbers. The nth Fibonacci number must be defined in terms of the two before it, so to calculate the 100th you have to calculate the 99 numbers before it. Is there a formula which solves only the nth and does not need previous values? Yes, the formula is Binet's formula, derived by Jacques Philippe Marie Binet, a French mathematician in 1843. It uses the golden section number ($\phi = 1.618$) and is calculated using Binet's Algorithm:

$$f_n = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}} \tag{2}$$

```
int fib_Binet(int n)
{
    double phi;
    if (n < 2)
        return 1;
    phi = (1.0 + sqrt(5.0)) / 2.0;
    return (pow(phi,n) - pow(1-phi,n)) / sqrt(5.0);
}
```

All three of these algorithms execute in 0.0 seconds, for $n = 40$.

6. Other Recursive Algorithms

Although Fibonacci applied using binary recursion is not ideal, there is an alternative which uses simple linear recursion. Here the two previous Fibonacci numbers are stored as parameters to a recursive function.

```
int fib_LinearR(int a, int b, int n)
{
    if (n <= 2)
        return b;
    else if (n > 2)
        return fib_LinearR(b,a+b,n-1);
}
```

and is called using the nomenclature:

```
fib_LinearR(1,1,n);
```

Note the two parameters a and b hold two successive Fibonacci numbers. This linear recursive version takes linear time. For $n = 40$ binary recursion facilitates 204 668 309 function calls and takes approximately 37 seconds. Linear recursion conversely takes 39 calls and 0 seconds. Another interesting recursive solution is that recently posed by Rubio and Pajak [10] who offer up mutual recursion whereby two functions are defined in terms of each other. The solution is based on an analysis of Fibonacci's original rabbit problem and takes the form:

```
int fib_babies(int i)
{
    if (i == 1)
        return 1;
    else
        return fib_adults(i-1);
}

int fib_adults(int i)
{
    if (i == 1)
        return 0;
    else
        return fib_adults(i-1) + fib_babies(i-1);
}
```

and is called using the nomenclature:

```
fib = fib_adults(n) + fib_babies(n);
```

With $n = 40$ this form of recursion takes about 120 seconds, so it's performance is actually worse than binary recursion.

7. A Quick Algorithm

Shortt [11] introduced an iterative process for calculating Fibonacci numbers in 1978. A similar construct was posed by Dijkstra, in the same year [12]. Both are based on work by Varobyov [13] in 1966. Dijkstra begins his algorithm with $f_0 = 0$ and $f_1 = 1$ using the following constructs:

$$f_{n-1} = f_{n-1}^2 + f_n^2,$$
$$f_{2n} = (2f_{n-1} + f_n)f_n.$$

Therefore only f_n and f_{n-1} are required to compute both f_{2n} and f_{2n-1}. As an example, to calculate f_{200} , there are only 18 values of f needed. This naturally results in much less overhead. This problem can itself be solved in three ways: iteratively, recursively and a blending of recursion and dynamic programming. If we solve it recursively, for f_{200} there are 327 calls to the function which, like binary recursion is largely due to redundant calls. For example, f_0 is called 64 times, f_1, 100 times. Below is the recursive implementation of the algorithm. The equations have been decoupled using temporary variables to reduce recursive calculations.

```
long long fib_DijkstraR(int n)
{
        long long i,j,tempi,tempj;
        if (n == 0)
                return 0;
        if (n == 1)
                return 1;

        if (n % 2 == 0) { // even
                i = (n-1) / 2;
                j = n / 2;
                tempi = fib_DijkstraR(i);
```

```
                tempj = fib_DijkstraR(j);
                return (2 * tempi + tempj) * tempj;
        }
        else { // odd
                i = n / 2;
                j = (n+1) / 2;
                tempi = fib_DijkstraR(i);
                tempj = fib_DijkstraR(j);
                return tempi * tempi + tempj * tempj;
        }
}
```

With $n = 40$ this algorithm takes about 0.02 seconds, with 63 recursive calls.

8. A Precise Algorithm

One of the caveats of sequences such as Fibonacci is that they progress in a rapid fashion. The first five numbers are 1,1,2,3,5, and before you know it you're at the 200$^{\text{th}}$ number, whose value is 280571172992510140037611932413038677189525. One of the difficulties associated with computing Fibonacci numbers can be attributed to the fact that for any value of f_n ($n > 17$) the number of digits in f_n is at least $\frac{n}{5}$ and at most $\frac{n}{4}$ [13]. At some point, many of the classic algorithms will fail, usually as a result of integer overflow. This problem with large Fibonacci numbers has to do with going beyond the maximum value which can be stored. On a 32-bit processor, using a C compiler, the maximum value of a signed long long data type is defined as 9223372036854775807. Now if we try and calculate f_{100} we get to f_{91} and f_{92} and we get 4660046610375530309 and 7540113804746346429 respectively. Which is fine, however if we now try and calculate f_{93}, we get -6246583658587674878. Which means an overflow has occurred and the numbers have wrapped around. We could use a C type *double* of course, but we will then have problems with the accuracy of calculations. Actually everything works fine until we hit f_{40} whose result is 102334154.999999991559, instead of 102334155. Subsequently the errors will just compound. We could also move to a 64-bit architecture. However none of these solutions solve the basic problem. There is however another, trickier way. There are challenges with large Fibonacci numbers, in a similar fashion to those experienced by large factorials, or any sequence of numbers. The solution is to treat the numbers as a series of digits next

to one another. Now, we can store each of these digits as a right-justified series of elements in an array. Once we have broken the large numbers into numbers into single digit numbers, we can apply simple addition of adjoining rows in the array. Consider the example shown in Figure 1. If we add the numbers 610 and 987, the first step is store the numbers one above the other in a 2D holding array. Now, starting at the right, we add the 7 and 0 equaling 7, which implies write the 7 in the row below. Next add $8 + 1 = 9$, and finally add $6 + 9 = 15$, which implies write the 5, carry the 1 to the next column. It actually produces numerically accurate solutions which are easy to output.

	0					$n-1$
f_{15}	0	0	0	6	1	0
f_{16}	0	0	0	9	8	7
f_{17}	0	0	**1**	**5**	**9**	**7**

+

Figure 1. Fibonacci "addition" using array elements.

Potentially the limit of this algorithm using a C *int* array on a 32-bit system is a Fibonacci number with 2 147 483 647 elements. The neat thing about this approach is that is shows students an innovative way of tackling very large numbers, as well as calculating large Fibonacci numbers accurately.

```
void fib_P(int n)
{
    // 2D array to hold Fibonacci numbers up to 50 digits
    int fib[200][50];
    int i,j,k,r1,r2,s1,carry = 0,sum,temp = 0,t;

    // Set all values of the hold array to 0
    for (i=0; i<200; i++)
        for (j=0; j<50; j++)
            fib[i][j] = 0;
    // Assign the first two Fibonacci numbers
    fib[0][49] = 1;
    fib[1][49] = 1;
```

```
    // Cycle through F3 to F100
    for (k=2; k<=100; k++) {
        r1 = 0;
        // Find the start of number Fn-1
        while (fib[k-1][r1] == 0)
            r1 = r1 + 1;
        // Set the carry-over to zero
        carry = 0;

        // Start from the end of Fn-1 moving left
        for (s1=99; s1>=r1; s1=s1-1) {
            // Add two sequential digits
            sum = (fib[k-1][s1] + fib[k-2][s1]) + carry;
            // Deal with sums>10, calculate the carry over
            if (sum >= 10) {
                temp = sum % 10;
                carry = (sum - temp)/10;
                sum = temp;
                if (s1 == r1)
                    r1 = r1 - 1;
            }
            else
                carry = 0;
            // Set the digit in Fn to the value calculated
            fib[k][s1] = sum;
        }
        t = 0;
        // Print out Fn
        while (fib[k][t] == 0)
            t = t + 1;
        printf("%d ",k+1);
        for (j=t; j<50; j++)
            printf("%d",fib[k][j]);

        printf("\n");
    }
}
```

9. Conclusion

The basic algorithm for Fibonacci is fairly straightforward. It is the way that the algorithm can be extended to incorporate increases in speed, or less memory that makes Fibonacci attractive as an illustrative example. It can be introduced early on in CS1 and used throughout the course as a cascading case, to illustrate new programming constructs and illustrate the differing approaches to an algorithm. In CS2 it can be used to introduce the notion of algorithm complexity, allowing a comparison of the various algorithms to gauge running time and memory usage. In a course on data structures, the precise algorithm could be extended further to incorporate the use of pointers and linked lists as opposed to simple arrays. In retrospect, binary recursion offers a concept, but its realization is limited for large Fibonacci numbers. It should be relegated to an example of when not to use recursion. There are merits to more non-traditional algorithms. Indeed there is merit in the entire notion of cascading cases. By familiarizing themselves with an algorithm, students are able to differentiate facets of algorithm design. By introducing the various renditions of Fibonacci, students are confronted with a good example of algorithm diversity, with the most optimal solution depending on whether the task requires the least time, or storage frugality. It also illuminates problems with calculating large numbers.

References

[1] M. de Sales McNabb, Sister, Phyllotaxis, *The Fibonacci Quarterly* **1–2** (1963–64), 57–60.

[2] P. B. Onderdonk, Pineapples and Fibonacci numbers, *The Fibonacci Quarterly* **8** (1970), 507–508.

[3] M. B. Linford, Fruit quality studies II, eye number and weight, *Pineapple Quarterly* **3** (1933), 185–195.

[4] C. Sutton, Sunflower spirals obey laws of mathematics, *New Scientist* (April 18, 1992), 16.

[5] A. Brousseau, Brother, Fibonacci statistics in conifers, *The Fibonacci Quarterly* **7** (1969), 525–532.

[6] H. Norden, Proportions and the composer, *The Fibonacci Quarterly* **10** (1972), 319–322.

[7] M. Bicknell Johnson, Fibonacci Chromotology or how to paint your rabbit, *The Fibonacci Quarterly* **16** (1978), 426–428.

[8] D. A. Preziosi, Harmonic design in Minoan architecture, *The Fibonacci Quarterly* **6** (1968), 370–384.

[9] A. J. Martin, M. Rem, A presentation of the Fibonacci algorithm, *Information Processing Letters* **19** (1984), 67–68.

[10] M. Rubio, B. Pajak, Fibonacci numbers using mutual recursion, *Proc. 5th Finnish/Baltic Sea Conference on Computer Science Education* (2006), 174–177.

[11] J. Shortt, An interactive program to calculate Fibonacii numbers in O(log n) arithmetic operations, *Information Processing Letters* **7** (1978), 299–303.

[12] E. W. Dijkstra, In honour of Fibonacci, *University of Texas EWD654* (1978).

[13] N. N. Voroboyov, *The Fibonacci Numbers*, D. C. Heath and Company, Boston, IL, 1966.

MICHAEL A. WIRTH
DEPARTMENT OF COMPUTING AND INFORMATION SCIENCE
UNIVERSITY OF GUELPH
GUELPH, ONTARIO N1G 2W1
CANADA

E-mail: mwirth@uoguelph.ca

(Received January, 2008)