6 The interesting case of the Fibonacci sequence

Although it is now associated with Leonardo Fibonacci, who wrote about it in the thirteenth century, the Fibonacci sequence was described in Indian poetry books from the earliest times. It remains interesting for number theoretic reasons, for example, it is related to the so called Golden Mean. One of Alan Turing's interests at the end of his life was in Fibonacci Phyllotaxis.

The Fibonacci sequence is defined recursively. Using a notation where the nth Fibonacci number is $F(n)$ then the sequence is defined as having $F(0) = 0$, $F(1) = 1$ and the nth element given by

$$F(n) = F(n-1) + F(n-2) \tag{1}$$

This means that if you know all the elements below $F(n)$ you can work out $F(n)$; since you know $F(0)$ and $F(1)$ you can work out all the $F(n)$ one by one. This isn't the same as having a direct formula, a 'closed form' for $F(n)$ in terms of n. If you want $F(100)$, say, you need to first work out $F(2)$, $F(3)$ and so on all the way up to $F(98)$ and $F(99)$. In fact, if you try this you'll find $F(100)$ is very large. The first few terms are

$$\begin{array}{rcl} F(2) & = & 1+0=1 \\ F(3) & = & 1+1=2 \\ F(4) & = & 2+1=3 \\ F(5) & = & 3+2=5 \\ F(6) & = & 5+3=8 \\ F(7) & = & 8+5=13 \\ F(8) & = & 13+8=21 \\ F(9) & = & 21+13=34 \end{array} \tag{2}$$

and so on. It is in the Encyclopedia of Integer Sequences as `http://oeis.org/A000045`.

The Fibonacci Sequence does have a closed form,

$$F(n) = \frac{\tau^n - (1-\tau)^n}{\sqrt{5}} \tag{3}$$

where $\tau = (1+\sqrt{5})/2$. Finding this uses a standard technique in the study of recursive formula; you substitute $F(n) = r^n$ into the equation giving

$$r^n = r^{n-1} + r^{n-2} \tag{4}$$

Dividing across by r^{n-2} makes this

$$r^2 - r - 1 = 0 \tag{5}$$

and this is solved by

$$r = \frac{1 \pm \sqrt{5}}{2} \tag{6}$$

or $r = \tau$ and $r = 1 - \tau$. Since the recursion relation is linear this means it is solved by

$$F(n) = A\tau^n + B(1-\tau)^n \tag{7}$$

for any A and B. The values of A and B are fixed by comparing to the starting values $F(0) = 0$ and $F(1) = 1$. In fact, because this involves high powers of irrational numbers, the recursive formula is more convenient. This is often the case, lots of problems can be more conveniently

```
1 int fib(int n)
2 {
3   if(n==0||n==1){
4       return n;
5     }
6
7   return fib(n-1)+fib(n-2);
8
9 }
```

Table 1: A recursive function for calculating the Fibonacci Sequence. This function can be found included in the program `fib_recursion.c`. If n is 0 or 1 the program returns n, these are the stopping conditions, otherwise, the function calls itself with a smaller value. The computer will put more and more copies of the function on the stack with smaller and smaller values of n until the end condition is reached and it passes the values back down to the original copy of the function, popping off the stack as it goes, until it returns the answer to the main program. This isn't a particularly safe implementation, if it is passed a negative integer it will never reach a terminating condition and so it will eventually overflow the stack and give a segmentation error, this is done in `fib_recursion_notermination.c`; just replacing (n==0||n==1) would stop this since then it would always terminate, even if the result for $n < 0$ is not useful.

```
1 int fib(int n)
2 {
3   return (n < 2) & n : fib(n-1) + fib(n-2);
4 }
```

Table 2: A fancier recursive function for calculating the Fibonacci Sequence. This uses the ternary operator.

expressed in a recursive form. This is particularly true of problems that are going to be solved on a computer.

An example recursive function for the Fibonacci Sequence is given in Table 1 or in a fancier way in Table 2. As usual with a recursive function, this function consists of two parts, a base case which returns fib(n-1)+fib(n-2), and the terminating condition for fib(0) and fib(1).

Calculating the run time in the case of Fibonacci is more difficult. The run time is

$$T(n) = c + T(n-1) + T(n-2) \tag{8}$$

so it seems like, apart from a constant, the run time for the recursive algorithm is itself like a Fibonacci sequence. In fact, it isn't quite that simple since the Fibonacci sequence has particular starting values, $F(0) = 1$ and $F(1) = 1$, but this calculation makes it clear that the recursive Fibonacci algorithm is $O(\tau^n)$ because $(1-\tau)^n$ is very small for large n. This is terrible. The reason this algorithm is so poor is that it calculates lots of quantities lots of times, for example `fib(n-2)` is calculated when it is called by `fib(n)` and when it is called by `fib(n-1)`; `fib(n-3)` is called by both `fib(n-2)` and by `fib(n-1)` and so on.

There are better ways to calculate the Fibonacci sequence! Table 3 gives a better algorithm; this algorithm uses tail recursion and basically passes along the last two numbers. It is easy to calculate the running time; on the face of it

$$T(n) = c + T(n-1) \tag{9}$$

making this algorithm $O(n)$. Furthermore, this recursion algorithm is very similar to the more obvious version with a for loop. This isn't the end of it, there is a $O(\log n)$ algorithm which won't be discussed here which rephrases the recursion in terms of matrix multiplications and uses efficient algorithms to do the matrix multiplication.

However, there is another confusing point; Fibonacci numbers get very large, very quickly. In fact, for large n, $F(n) \approx \tau^n/\sqrt{5}$ so that

$$\log_\tau F(n) \approx n \tag{10}$$

or, by the laws of logs

$$\log_{10} F(n) \approx n \log_{10} \tau \approx 0.2090n \tag{11}$$

In other words, since $\lfloor log_{10} A \rfloor$ gives the number of digits in A, the funny bracket is 'floor', the number rounded down, $F(n)$ has about $n/5$ digits, or put another way, the number of digits in $F(n)$ is $O(n)$. A graph of the length of $F(n)$ against n is given in Fig. 1. All through our analysis we assume certain operations, including summing numbers, use a constant amount of time, but here that clearly breaks down, as n gets higher the time spent adding $F(n-1)$ and $F(n-2)$ is going to get longer, as is the time spend accessing their storage and over writing the previous values. This case shows just how complicated calculating run times can be.

```
1  int fib_r(int n, int a, int b)
2  {
3     if(n==0)
4       return a;
5     if(n==1)
6       return b;
7     if(n==2)
8       return a+b;
9
10      return fib_r(n-1,b,a+b);
11
12 }
13
14 int fib(int n)
15 {
16   return fib_r(n,0,1);
17 }
```

and here is a table for $n = 10$, \$1, \$2 and \$3, the three columns, represent the first, second and third argument of factorial(int n, int a, int b) when it is called.

\$1	\$2	\$3
10	0	1
9	1	1
8	1	2
7	2	3
6	3	5
5	5	8
4	8	13
3	13	21
2	21	34

and then it returns $21 + 34 = 55$ which is correct.

Table 3: A more efficient algorithm for calculating the Fibonacci sequence. This algorithm avoids the wasteful extra recursive calls that were a feature of the previous Fibonacci function. It does loose some of the naturalness though, the recursive structure is less obvious and it looks more like an attempt to shoe-horn the for-loop form into a recursive function. The for-loop form is given in Table 4.

```
1  int fib(int n)
2  {
3    if(n<2)
4       return n;
5
6    int last=1, old_last=0;
7    int i;
8
9    for(i=2;i<=n;i++){
10          int temp=last;
11          last=last+old_last;
12          old_last=temp;
13       }
14   return last;
15 }
```

Table 4: A simple algorithm for calculating Fibonacci numbers, this is implemented as part of `fib_loop.c`.

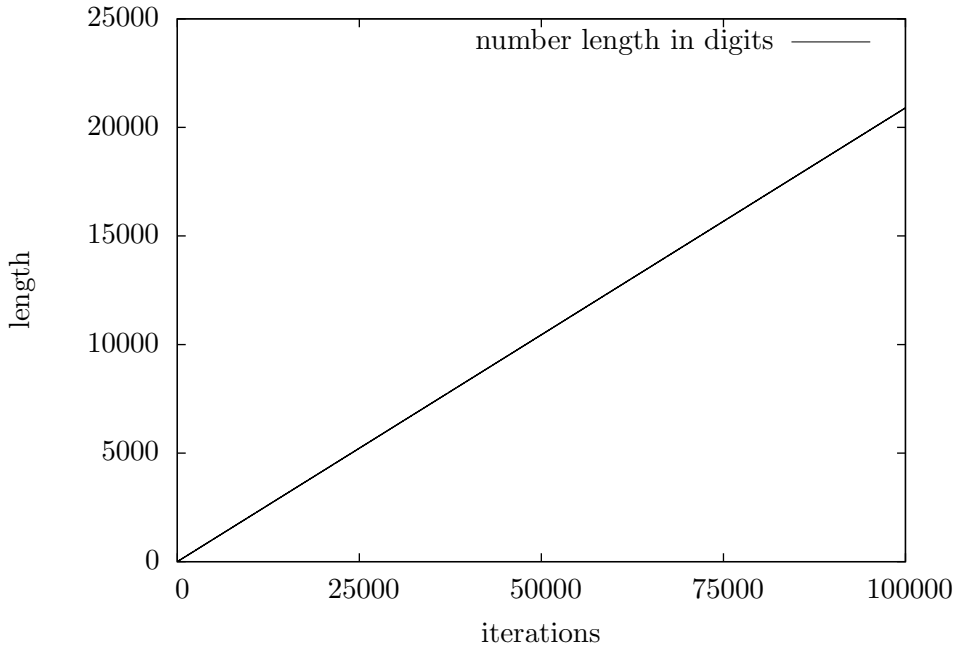

Figure 1: A plot of the length of $F(n)$ in digits against n.