Calculating Fibonacci numbers using the Binet formula without using floating point arithmetic

Faifel B.L.

SGTU by Yu.A. Gagarin,

Saratov

By now, many ways have been invented for calculating Fibonacci numbers: from direct recursion based on the formula:

$$F_n=F_{n-1}+F_{n-2}$$

to the matrix method described by D. Knuth

Binet's formula is located separately in this series of algorithms, which has the form:

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

This formula seems attractive at first glance, but it contains an irrational number, which in computer calculations we are forced to represent in the form of a floating point number (i.e. replace an infinite non-periodic fraction with a finite one).

This means that the calculations will not be accurate; a limitation error is introduced into them.

The author once came across a publication in which Binet's formula was used to calculate a very large Fibonacci number, but the implementation assumed the use of superhigh-bit floating arithmetic (so that the required number would completely fit into the mantissa).

We'll take a completely different path!

Consider a set of numbers of the form:

$$x = a + \sqrt{5} * b$$

where a and b are integers. It seems quite obvious that this set is algebraically closed with respect to the operations of ordinary addition and multiplication:

$$\left(a + b\sqrt{5}\right) + \left(c + d\sqrt{5}\right) = \left((a + c) + (b + d)\sqrt{5}\right)$$

$$\left(a + b\sqrt{5}\right)\left(c + d\sqrt{5}\right) = \left((ac + 5bd) + (ad + bc)\sqrt{5}\right)$$

In addition, zero and unit belong to the set under consideration in a trivial way:

$$1 \equiv (1 + \sqrt{5} * 0)$$

$$0 \equiv (0 + \sqrt{5} * 0)$$

Subtraction is quite naturally realized:

$$\left(a + b\sqrt{5}\right) - \left(c + d\sqrt{5}\right) = \left((a - c) + (b - d)\sqrt{5}\right)$$

Now you can implement arithmetic on a set of pairs (a, b), in which addition, subtraction and multiplication will be described by the formulas:

$$(a,b)+(c,d)=((a+c),(b+d))$$

$$(a,b)-(c,d)=((a-c),(b-d))$$

$$(a,b)*(c,d)=((ac+5bd),(ad+bc))$$

Thus, we can "safely forget" about √5 and implement a direct calculation using the Binet formula.

As a result, the numerator of the fraction will be a pair of the form (0, r√5) = r√5. Dividing this irrational number by √5 gives the desired integer result. Naturally, in reality, dividing is not required, it is enough to calculate (using the above-described pair arithmetic) two binomials:

$$A = \frac{(1+\sqrt{5})^n}{2^n} \quad и \quad B = \frac{(1-\sqrt{5})^n}{2^n}$$

and then subtract A-B

```python
def prod_pairs(a,b): # pairs multiplication
    return (a[0]*b[0]+5*a[1]*b[1],a[0]*b[1]+a[1]*b[0])
def sub_pairs(a,b):   # pairs subtracting
    return (a[0]-b[0],a[1]-b[1])
def pow_pair(a,n):   # exponentiation
    c=a
    for _ in range(n-1):
        c=prod_pairs(c,a)
    return c
def fib_bine(n): # Binet formula
    x1=pow_pair((1,1),n)
    x2=pow_pair((1,-1),n)
    z=sub_pairs(x1,x2)
    return z[1]//(2**n)
```

Is it possible to speed up this code? Yes, we can, if we speed up the exponentiation.

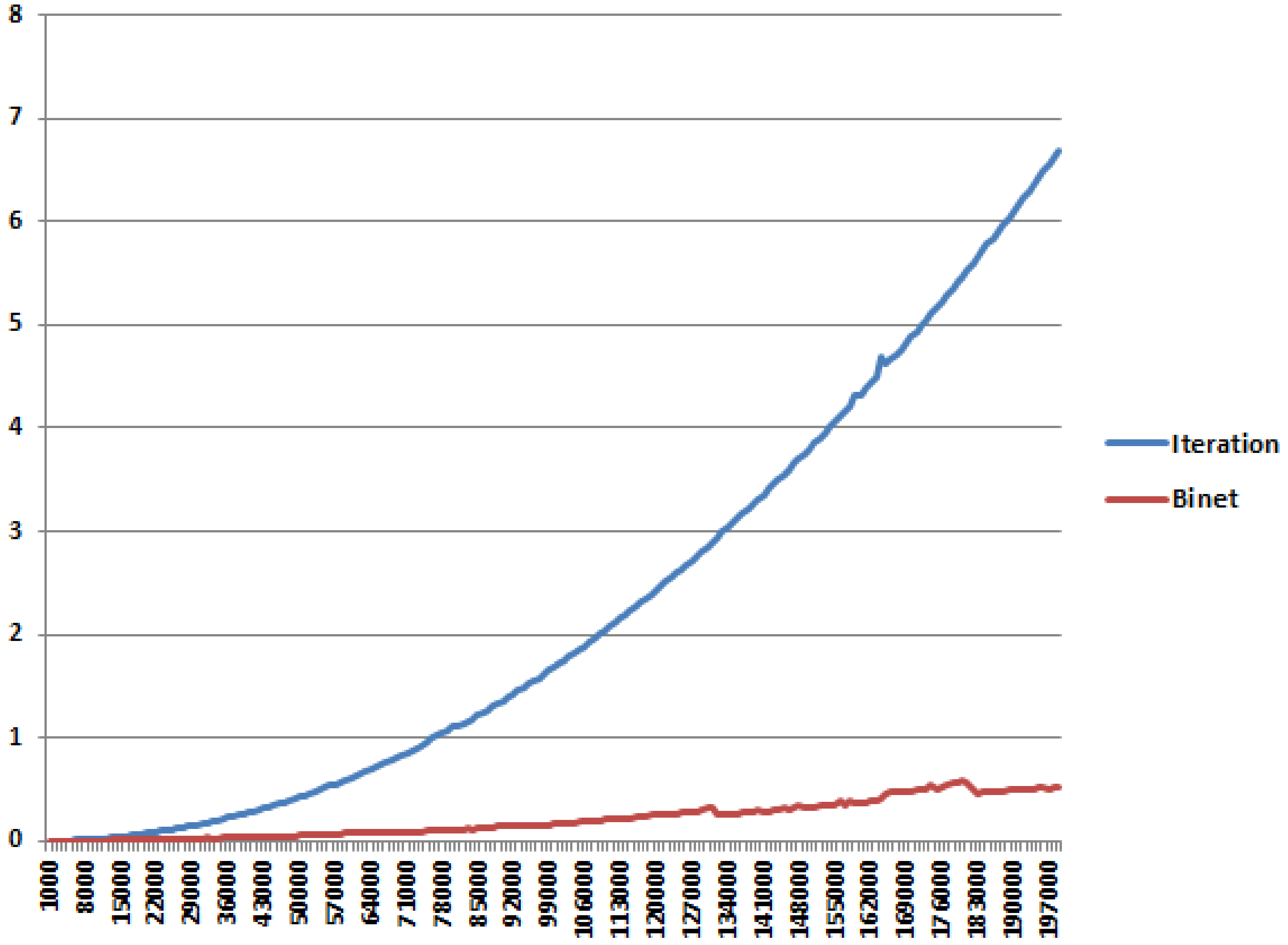
To speed up exponentiation, there is a standard approach, which is that to calculate x^n, the chain $x \to x^2 \to x^4 \to \ldots \to x^{2^k}$ is calculated until $2^k \le n$, and then $x^{(n - 2^k)}$.

```
def pow_pair(a,n):
    if (n==1):
        return a
    c=copy(a)
    k=1
    while k*2<=n:
        if k<=n:
            c=prod_pairs(c,c)
            k=k*2
    p=n-k
    if p>=1:
        tmp=pow_pair(a,p)
        return prod_pairs(tmp,c)
    else:
        return c
```

Using this technique allows us to calculate Fibonacci numbers in a logarithmic time using the Binet formula and without using floating point arithmetic.

Below are the test results comparing the computation time of Fibonacci numbers by simple iteration:

```
def fib_ite(n):
    c,p=0,1
    for _ in range(n):
        c,p=c+p,c
    return c
```


8
7
6
5
4
3
2
1
0
1000
8000
15000
22000
29000
36000
43000
50000
57000
64000
71000
78000
85000
92000
99000
106000
113000
120000
127000
134000
141000
148000
155000
162000
169000
176000
183000
190000
197000
Iteration
Binet

Despite the apparent simplicity of the fib_ite code, the fib_bine function performs significantly better.

Thanks for your attention!