A Super Fast Fibonacci Algorithm

The Fundamental Equation

Binet's formula is a great way to calculate any Fibonacci number without having to calculate all those that come before. It's given by

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

Notice that the quantities in the numerator are the famous golden ration φ and its conjugate $-1/\varphi$

Easy enough then. Let's calculate the Fibonacci sequence!

We'll perform the calculation with BigFloats (floating point numbers with arbitrary precision). It turns out that in order to get6 the correct answer, we'll need to increase the precision in proporton to the Fibonacci number n.

```
using Plots    , BenchmarkTools
```

```
fibonet_raw (generic function with 1 method)
```

```
function fibonet_raw(n)::BigInt
    setprecision(max(2,n)) # precision must increase as n increases, or errors
round
    return round((((1+√BigInt(5))/2)^n - ((1-√BigInt(5))/2)^n) / √BigInt(5))
end
```

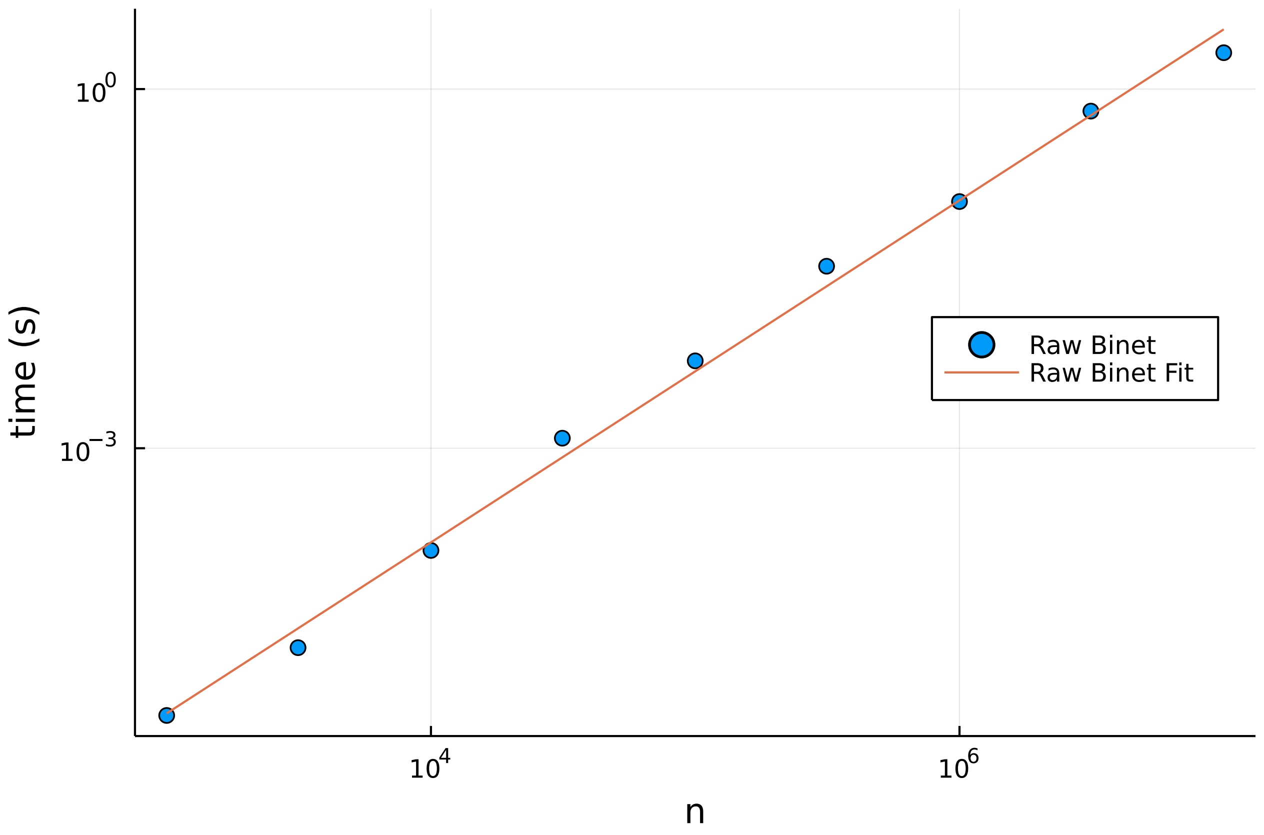
```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765
```

```
[fibonet_raw(i) for i ∈ 0:20]
```

This is the correct sequence. Now let's map the speeds we assume a relationship between computation time (t) and Fibonacci number (n) as follows:

$$t = cn^x$$

taking the log of both sides yields $\log t = x \log n + \log c$



we find that

$$t(n) = 10^{-22} n^{1.4}$$

Q: How long would it take to calculate the 1,000,000,000th Fibonacci number using this method?

A: 2268 seconds

Less than half an hour. Not terrible. But can we do better?

Forget the Floats

The first thing we notice is that, even though there are $\sqrt{5}$'s all over the Binet formula, the answer is always an integer. So that fact suggests there must be a clever way to perform the calculation using integers only.

Let's define an ordered pair $\phi(a, b)$ that represents numbers in the field $\mathbb{Z}(\sqrt{5})$. a and b are both integers, and the number represented is of the form $a + b\sqrt{5}$.

```julia
struct ϕField <: Number
    a::BigInt
    b::BigInt
end
```

Multiplying these ordered pair representations of $\mathbb{Z}(\sqrt{5})$ is done as follows:
$\phi(a_1, b_1) * \phi(a_2, b_2) = \phi(a_1 a_2 + 5 b_1 b_2, a_1 b_2 + a_2 b_1)$

```
one (generic function with 29 methods)
```

```julia
begin
    import Base: *, -, convert, promote_rule, one

    function *(num1::ϕField, num2::ϕField)::ϕField
        a = num1.a * num2.a + 5 * num1.b * num2.b
        b = num1.a * num2.b + num2.a * num1.b
        return ϕField(a, b)
    end

    function -(num1::ϕField, num2::ϕField)::ϕField
        a = num1.a - num2.a
        b = num1.b - num2.b
        return ϕField(a, b)
    end

    convert(::Type{ϕField}, x::Int) = ϕField(BigInt(x), 0)

    promote_rule(::Type{ϕField}, ::Type{<:Number}) = ϕField

    one(::Type{ϕField}) = ϕField(1,0) # enables use of built in ^
end
```

```
ϕField(6, 2)
```

```julia
let
    ϕ = ϕField(1,1)
    ϕ*ϕ
end
```

Extracting the Integers

Let's represent the Binet equation as

$$f(n) = \frac{\mathfrak{A}(\phi(1,1)^n - \phi(1,-1)^n) + \mathfrak{B}(\phi(1,1)^n - \phi(1,-1)^n)\sqrt{5}}{2^n\sqrt{5}}$$

where the function $\mathfrak{A}$ extracts the coefficient a from $a + b\sqrt{5}$, and the function $\mathfrak{B}$ extracts the coefficient b from $a + b\sqrt{5}$.

We expect the value of $\mathfrak{A}(\phi(1,1)^n - \phi(1,-1)^n)$ to be 0 for all n because there's a $\sqrt{5}$ term in the denominator that has to cancel. All Fibonacci numbers are integers, so no irrational terms should survive.

Let's check

```
B (generic function with 1 method)
```

```
begin
    A(ϕ::ϕField) = ϕ.a

    B(ϕ::ϕField) = ϕ.b
end
```

```
["0 + 2 √5", "0 + 4 √5", "0 + 16 √5", "0 + 48 √5", "0 + 160 √5", "0 + 512 √5", "0 + 16
```

```
begin
    fields = [prod([ϕField(1,1) for i in 1:n]) -
                  prod([ϕField(1,-1) for i in 1:n]) for n ∈ 1:10]
    ["$(A(ϕ)) + $(B(ϕ)) √5" for ϕ ∈ fields]
end
```

Yes! the a term disappears. Thus we can rewrite our Binet algorithm equation as

$$f(n) = \frac{\mathfrak{B}(\phi(1,1)^n - \phi(1,-1)^n)}{2^n}$$

Forget about $\phi(1,-1)$

Now let's examine the symmetry between $\phi(a_1,b_1)^n$ and $\phi(a_1,-b_1)^n$. We hypothesize that if $\phi(a_1,b_1)^n = \phi(a_n,b_n)$ then $\phi(a_1,-b_1)^n = \phi(a_n,-b_n)$ for all n

Proof by induction:

If

$$\phi(a_n, b_n) = \phi(a_{n-1}, b_{n-1}) \times \phi(a_1, b_1) = \phi(a_{n-1}a_1 + 5b_{n-1}b_1, a_{n-1}b_1 + b_{n-1}a_1)$$

$$\phi(a_n, -b_n) = \phi(a_{n-1}, -b_{n-1}) \times \phi(a_1, -b_1) = \phi(a_{n-1}a_1 + 5b_{n-1}b_1, -a_{n-1}b_1 - b_{n-1}a_1)$$

Then

$$\phi(a_{n+1}, b_{n+1}) = \phi(a_n, b_n) \times \phi(a_1, b_1) = \phi(a_n a_1 + 5b_n b_1, a_n b_1 + b_n a_1) =$$

$$\phi(a_{n-1}(a_1^2 + 5b_1^2) + 10b_{n-1}a_1b_1, 2a_{n-1}a_1b_1 + 5b_{n-1}(a_1^2 + 5b_1^2))$$

$$\phi(a_{n+1}, -b_{n+1}) = \phi(a_n, -b_n) \times \phi(a_1, -b_1) = \phi(a_n a_1 + 5b_n b_1, -a_n b_1 - b_n a_1) =$$

$$\phi(a_n a_1 + 5b_n b_1, -a_n b_1 - b_n a_1)$$

base case:

Let $n = 2$

$$\phi(a_2, b_2) = \phi(a_1, b_1)^2 = \phi(a_1^2 + 5b_1^2, a_1b_1 + a_1b_1)$$

$$\phi(a_2, -b_2) = \phi(a_1, -b_1)^2 = \phi(a_1^2 + 5b_1^2, -a_1b_1 - a_1b_1)$$

Q.E.D.

Demonstration for $n = 3$

$$\phi(a_3, b_3) = \phi(a_2, b_n2) \times \phi(a_1, b_1) = \phi(a_2a_1 + 5b_2b_1, a_2b_1 + b_2a_1) =$$

$$\phi(a_1^3 + 15a_1b_1^2, 3a_1^2b_1 + 5b_1^3)$$

$$\phi(a_3, -b_3) = \phi(a_2, -b_2) \times \phi(a_1, -b_1) = \phi(a_2a_1 + 5b_2b_1, -a_2b_1 - b_2a_1) =$$

$$\phi(a_1^3 + 15a_1b_1^2, -3a_1^2b_1 - 5b_1^3)$$

```
[(ϕField(2, 3), ϕField(2, -3)), (ϕField(49, 12), ϕField(49, -12)), (ϕField(278, 171), ϕ
```

```
[(prod([ϕField(2,3) for i ∈ 1:n]),  prod([ϕField(2,-3) for i ∈ 1:n])) for n ∈ 1:10]
```

With this fact in hand we can make the following simplification

$$\mathfrak{B}(\phi(1,1)^n - \phi(1,-1)^n) = 2\mathfrak{B}(\phi(1,1)^n)$$

We can reduce our computation budget by 50%. Our Binet algorithm equation has now become

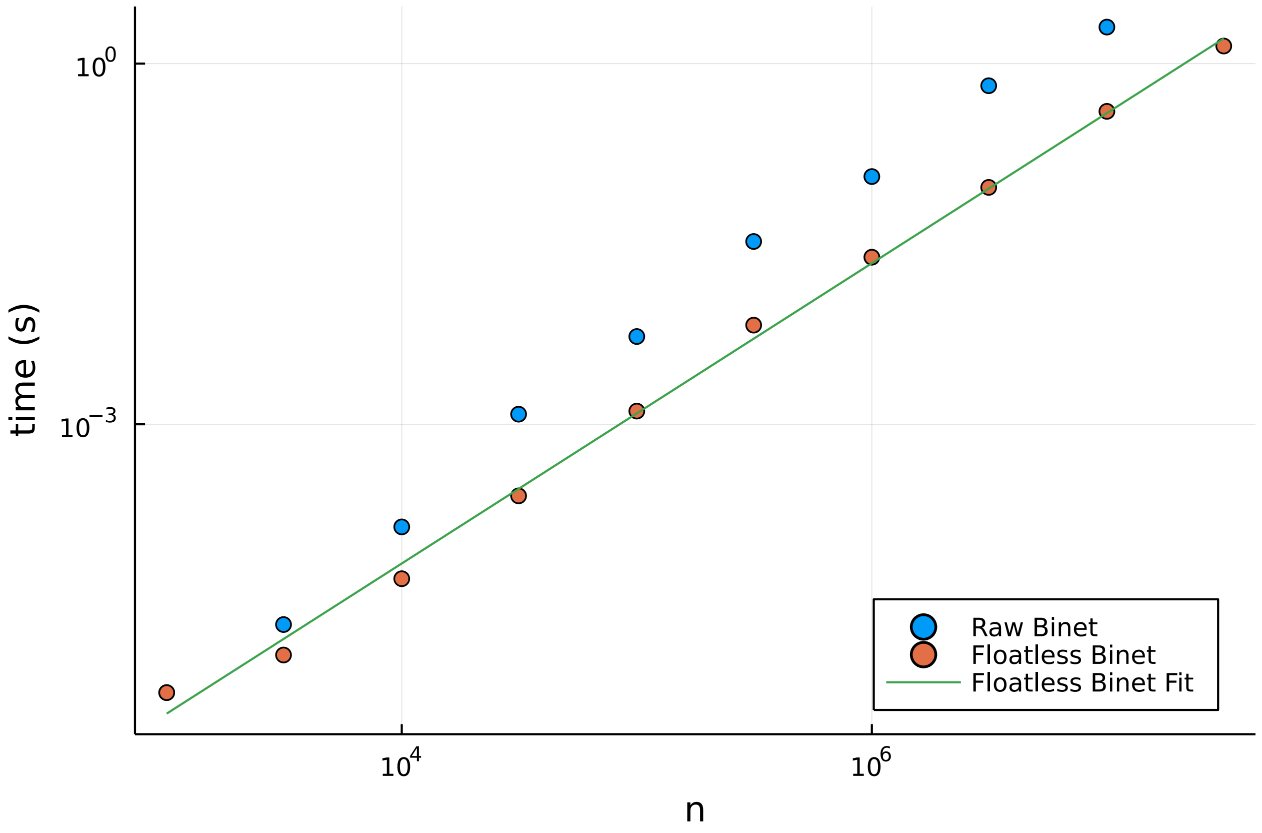
$$f(n) = \frac{2\mathfrak{B}(\phi(1,1)^n)}{2^n}$$

```
fibonet_floatless (generic function with 1 method)
```

```julia
function fibonet_floatless(n::Int)::BigInt
    return 2*B(ϕField(1,1)^n) ÷ BigInt(2)^n
end
```

```
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

```julia
[fibonet_floatless(i) for i ∈ 0:20]
```



$$t(n) = 10^{-22}n^{1.2}$$

The exponent is slightly better than with the raw Binet algorithm.

Optimized Bitshifting

The final improvement is to deal with the $s = 2^n$ inthe denominator. It turns out there's a really elegant optimization that substantially reduces computation time. Let's first understand how exponentiation (`^`) work.

Smart Exponentiation Through Doubling

Say we want to evaluate b^n. We could multiply b with itself $n - 1$ times. A better way is to use smart doubling. Here's an example for evaluating b^{21} recursively.

With every iteration we double the base. But we only multiply the result by the base when the trailing digit of the binary representation of n is 1:

$level$	$binary\ \ rep$	$base$	$trailing\, bit$	$result$
1	10101	b	1	b
2	1010	b^2	0	b
3	101	b^4	1	b^5
4	10	b^8	0	b^5
5	1	b^{16}	1	b^{21}

This saves on inefficient repeated doublings.

Brilliant Bitshifting

For the nth Fibonacci number we must perform the equivalent of $n-1$ bitshifts. We want to find the most efficient way of doing this. Here's that method.

Let's focus first on the base. Afterwards we'll deal with the in-process result. Practically, we can't bitshift until both a and b in $\phi(a,b)$ are even. Immediately after the first doubling we can and should bitshift once to the right. This divides the squared value by 2. Now, upon doubling again, that bitshift we performed "doubles" yielding 2 effective bitshifts. Double again and it becomes 4 effective bitshifts, and so on. For the nth Fibonacci number that would amount to $2^{\lfloor\log_2 n\rfloor-1}$ bitshifts.

But that's not enough. So after the second doubling we should bitshift again. This bitshift will also effectively double with each doubling, yielding an additional $2^{\lfloor\log_2 n\rfloor-2}$ bitshifts. Likewise after each additional doubling we should always perform one bitshift. The sum total of effective bitshifts we get by this algorithm is

$$2^{\lfloor\log_2 n\rfloor-1}+2^{\lfloor\log_2 n\rfloor-2}+2^{\lfloor\log_2 n\rfloor-3}+\cdots+2^{\lfloor\log_2 n\rfloor-\lfloor\log_2 n\rfloor}=$$

$$2^{\lfloor\log_2 n\rfloor}\left(\frac{1}{2}+\frac{1}{2^2}+\frac{1}{2^3}+\cdots+\frac{1}{2^{\lfloor\log_2 n\rfloor}}\right)=$$

$$2^{\lfloor\log_2 n\rfloor}\sum_{i=1}^{\lfloor\log_2 n\rfloor}\frac{1}{2^i}=$$

$$2^{\lfloor\log_2 n\rfloor}\left(1-\frac{1}{2^{\lfloor\log_2 n\rfloor}}\right)=$$

$$2^{\lfloor\log_2 n\rfloor}-1$$

So far that only accounts for the leading digit (in the binary representation of n). For every trailing digit whose value is 1 we must multiply the in-process result by the corresponding base.

Here's how it works for our example of b^{21}.

$level$	$binary\ rep$	$base\ bitshift$	$base$	$trailing\ bit$	$result$
1	10101	0	b	1	b
2	1010	1	$b^2/2$	0	b
3	101	1	$b^4/2^3$	1	$b^5/2^3$
4	10	1	$b^8/2^7$	0	$b^5/2^3$
5	1	1	$b^{16}/2^{15}$	1	$b^{21}/2^{18}$

The answer is close, but wrong. It appears we need to perform a bitshift to the in-process result every time we multiply the in-process result by the base

$level$	$binary\ rep$	$base\ bitshift$	$base$	$trailing\ bit$	$result\ bitshift$	$result$
1	10101	0	b	1	0	b
2	1010	1	$b^2/2$	0	0	b
3	101	1	$b^4/2^3$	1	1	$b^5/2^4$
4	10	1	$b^8/2^7$	0	0	$b^5/2^4$
5	1	1	$b^{16}/2^{15}$	1	1	$b^{21}/2^{20}$

We have reached our destination, a super-fast implementation of Binet's formula. I am unaware of any techniques to further improve performance. Any additional improvements are left to the reader.

Let's look at our algorithmic formulation of Binet's equation:

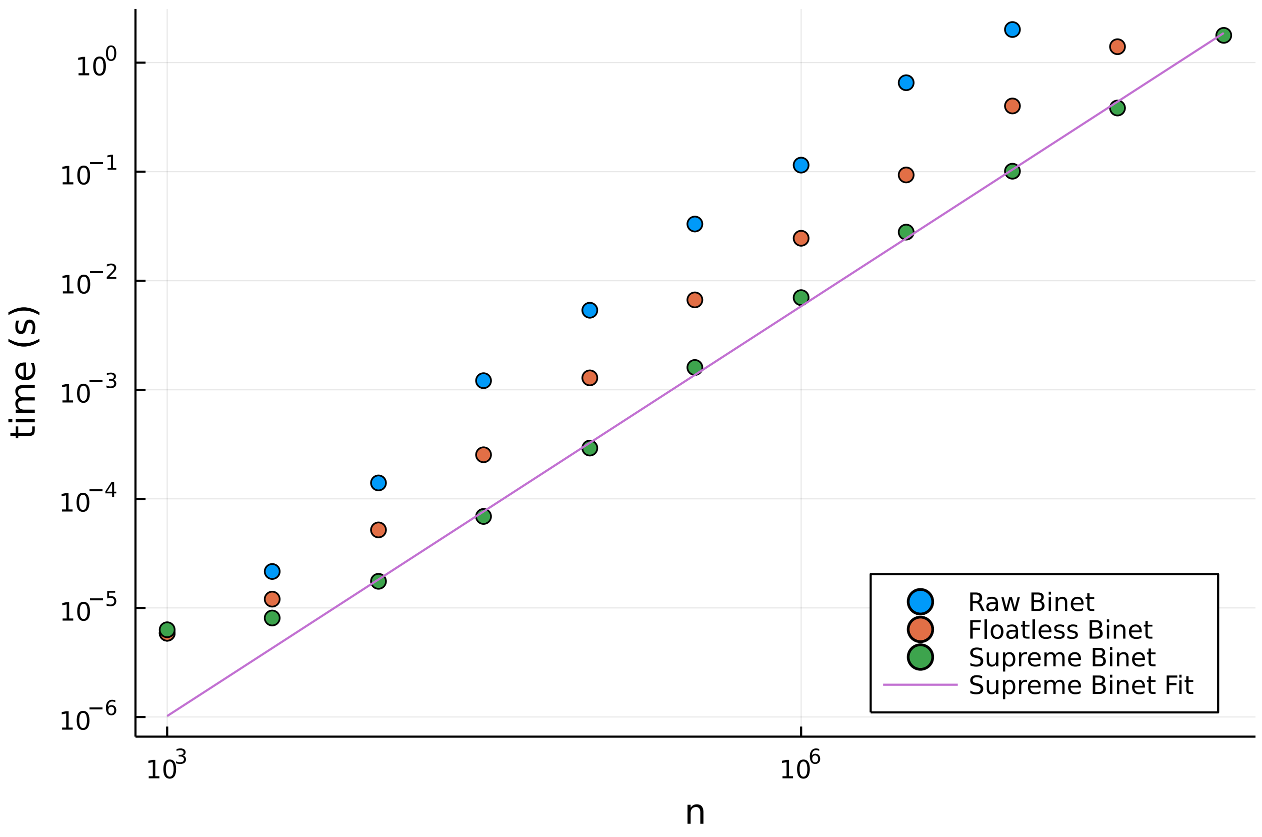
$$f(n) = \mathfrak{B}\left(\frac{\phi(a,b)^n}{2^{n-1}}\right)$$

```
>> (generic function with 9 methods)
```

```julia
begin
    import Base.>>

    function >>(num::ϕField, shift::Int)
        return ϕField(A(num)>>shift, B(num)>>shift)
    end
end
```

```
fibonet (generic function with 1 method)
```

```
begin
    import Base.^

    function ^(base::ϕField, n::Int, result=one(base), shift=0)::ϕField
        if n==0
            return result
        elseif n==1
            return ^(base, n>>1, (result*base)>>shift)
            # this penultimate level, don't square the base to safe compute
        elseif n%2==1
            return ^((base*base)>>1, n>>1, (result*base)>>shift, 1)
        else
            return ^((base*base)>>1, n>>1, result, shift)
        end
    end

    fibonet(n::Int)::BigInt = B(ϕField(1,1)^n)
end
```



```
BenchmarkTools.Trial: 1 sample with 1 evaluation.
 Single result which took 18.505 s (1.91% GC) to evaluate,
 with a memory estimate of 10.30 GiB, over 881633 allocations.
```

```
@benchmark fibonet(1_000_000_000)
```

That was a substantial improvement! In fact, we just calculated the 1,000,000,000th Fibonacci number in under 20 seconds.