A Julia Journey

In the beginning

I was doing some statistical experiments with *multiplicative* functions - these are the functions *f* defined on the integers such that $f(n,m) = f(n)f(m)$ whenever m and m are relatively prime. Initially, I was doing these experiments in *Mathematica*, but that was a little slow, so I decided to use these experiments as an excuse to learn more about **Julia** - a relatively new language targeted at technical and scientific computing designed at MIT - one attraction was that one of the people on the team was Alan Edelman - someone I have enormous respect for for his work on random matrix theory and related subjects. I also had a better algorithm in mind - computing multiplicative function for one number at a time requires factoring that number, and the idea was that computing it for all numbers in a range can be done more efficiently. The attraction of **Julia** over, say *Mathematica*, was that *Mathematica* is a shell language - it is essentailly impossible to implement something fast in *Mathematica* itself -either it is built in, or it is not. Since I have been doing a lot of work in *Mathematica* (and also in **Python**, and particularly in **Pandas** - the **Python** data science sub-language, which have the same problem), I got tired of the straight jacket, and wanted to try someone flexible enough to allow me to have a bit more fun.

First Try

I am new to Julia, and as my first stab at a mildly nontrivial program wrote the below, which computes the divisor sigma functions for all numbers in a range from 1 to N - the divisor sigma function $\sigma(k,n)$ computes the sum of kth powers of divisors of n (if $k = 0$, it is simply the number of divisors). Since $\sigma(k,n)$ is a multiplicative function of n, we write it as below (this, in principle, should be far more efficient than factoring each integer in our range):

```
using Primes
```

```
thefunc(i) = (a::Int, b::Int)-> sum(BigInt(a)^(k*i) for k in 0:b)
```

```
function powsfunc(x, n, func)
    bound = convert(Int, floor(log(n)/log(x)))
    return Dict{Int, Number}(x^i => func(x, i) for i in 0:bound)
end

dictprod2(d1, d2, bd)=Dict(i*j=> d1[i]*d2[j] for i in keys(d1), j in keys(d2) if i*j<bd)

function dosigma(k, n)
    primefunc = thefunc(k)
    combfunc = (a, b) -> dictprod2(a, b, n)
    allprimes = [powsfunc(p, n, primefunc) for p in primes(n)]
    trivdict = Dict{Int, Number}(1=>1)
    theresult = reduce(combfunc, trivdict, allprimes)
    return theresult
end
```

The good news is that the above works. The bad news is that it is horrifically slow, with `dosigma(0, 100000)` taking ten minutes of CPU time, and consing 150GB(!). Question is: why?

The light begins to dawn.

I have done some more thinking/coding/profiling (mostly on the very nice [JuliaBox][1] platform), and the one line summary is:

Julia is not LISP

which means that programming in functional style will only get you so far.

Now, let's get to the code (which is what we want to see anyhow). First, here is the reference "stupid" implementation:

```
using Primes
```

```
function sigma(k, x)
   fac = factor( x)
   return prod(sum(i^(j*k) for j in 0:fac[i]) for i in keys(fac))
   end

allsigma(k, n) = [sigma(k, x) for x in 2::n]
```

On JuliaBox (also on my laptop) it takes about 40 seconds for k=0, n=10000000. An astute reader will note that this will break for largish k because of overflow. The way I get around this is by replacing the function by:

```
function sigma(k, x)
   fac = factor( x)
   return prod(sum(BigInt(i)^(j*k) for j in 0:fac[i]) for i in keys(fac))
   end
```

This takes 131 seconds for the same computation (95 seconds on my desktop) so a little more than a factor of 3 slower. By contrast, the same computation in *Mathematica* done as:

```
foo = DivisorSigma[0, Range[10000000]] // Timing
```

takes 27 seconds (on the desktop), which indicates that, most likely, *Mathematica* first checks that the computation can be done in fixnums, and then does it in the obvious way.

Now we move on to the "intelligent" implementation. The working hypothesis here was that Julia is not at functional language level for operating with data, so, with that in mind, I avoided the creation and destruction of dictionaries for the following code:

```
using Primes
```

```
thefunc(i) = (a::Int, b::Int)-> sum(BigInt(a)^(k*i) for k in 0:b)
```

```
function biglist(n, func, thedict)
bot = Int(ceil(sqrt(n)))
theprimes = primes(bot, n)
    for i in theprimes
        thedict[i] = func(i, 1)
    end
    return theprimes
end

function powsfunc(x, n, func, motherdict)
    bound = convert(Int, floor(log(n)/log(x)))
    res = Int[]
    for i in 1:bound
        tmp = x^i
        push!(res, tmp)
        motherdict[tmp] = func(x, i)
    end
    return res
end

function makeprod(l1, l2, nn, dd)
    res = []
    for i in l1
        for j in l2
            if i*j <= nn
                dd[i*j] = dd[i] * dd[j]
                push!(res, i*j)
            end
        end
    end
    return vcat(l1, l2, res)
end

function dosigma3(n, k)
    basedict = Dict{Int, BigInt}(1=>1)
    ff = thefunc(k)
    f2(a, b) = makeprod(a, b, n, basedict)
    smallprimes = reverse(primes(Int(ceil(sqrt(n))) -1))
    bl = biglist(n, ff, basedict)
    for i in smallprimes
        tmp = powsfunc(i, n, ff, basedict)
        bl = makeprod(bl, tmp, n, basedict)
    end
    return basedict
end
```

now, `dosigma3(100000, 0)` takes 0.5 seconds, for a factor of 1500 speedup over my original code, and a factor of 150 speedup over the other answer.

Also `dosigma3(10000000, 0` takes too long to run on JuliaBox, but on the aforementioned desktop it takes 130 seconds, so within a factor of two of the dumb implementation.

An examination of the code shows that the `makeprod` routine suffers from not having the various inputs sorted, which could result in faster termination. To make them faster, we can introduce a merge step, thus:

```
function domerge(l1, l2)
   newl = Int[]
   while true
        if isempty(l1)
            return vcat(l2, reverse(newl))
        elseif isempty(l2)
            return vcat(l1, reverse(newl))
        elseif l1[end]>l2[end]
                tmp = pop!(l1)
                push!(newl, tmp)
        else
                tmp = pop!(l2)
                push!(newl, tmp)
        end
    end
end

function makeprod2(l1, l2, nn, dd)
    res = Int[]
    for i in l1
         restmp = Int[]
         for j in l2
            if i*j > nn
                break
            end
            dd[i*j] = dd[i] * dd[j]
            push!(restmp, i*j)
        end
        res = domerge(res, restmp)
    end
    return domerge(l1, domerge(l2, res))
end

function dosigma4(n, k)
    basedict = Dict{Int, BigInt}(1=>1)
    ff = thefunc(k)
    f2(a, b) = makeprod(a, b, n, basedict)
    smallprimes = reverse(primes(Int(ceil(sqrt(n))) -1))
    bl = biglist(n, ff, basedict)
    for i in smallprimes
        tmp = powsfunc(i, n, ff, basedict)
        bl = makeprod2(bl, tmp, n, basedict)
    end
    return basedict
end
```

However, this takes 15 seconds already for 100000, so I did not try it for 10000000, and indicates either my own incompetence in Julia (I am a novice, after all) or Julia's incompetence in managing memory.

Epiphany

I was depressed by the above - how could I be so dumb as to be unable to make a clever algorithm outperform a dumb one? As I was tossing and turning in bed one night, I finally realized the source of the problem - I was being too clever. The basic idea remained the same (as it has to be) - compute the multiplicative function on prime powers, but there was a layer missing - the power of the sieve was to factor all of the numbers in the range from 1 to N quickly (quickly means in time $O(N \log N)$, vs the dumb $O(N\sqrt{N})$, The additional benefit of this approach was that the factorization was produced once and then could be used for all imaginable multiplicative functions. Avoiding excessive cleverness immediately paid off: the code that ran in 130 seconds before, now ran in 26 seconds the first time we had to compute a multiplicative function, and in 2 seconds the second time (since no factoring was done the second time). The code, with all the bells and whistles can be found on github.com. It beats *Mathematica* handily (where by *Mathematica* we mean the C implementation *Mathematica* uses.)

Conclusion(?)

Using shell languages (like *Mathematica* and **Python** and **Pandas**) has been (and remains) a very effective way to get interesting experiments done, but it was taking all the fun out of the programming, because it was straightjacketing me into thinking the same way that the language creators did. *Julia* freed me from that (without the bit-twiddling and core dumps that programming in C would have involved, and without dealing with the ugliness that is C++), and I am very grateful. While *Julia* comes from MIT and has unmistakable roots in **Lisp** and **Scheme** (a big plus for me), it is not **Lisp**, neither is it **Scheme**, it is its own thing and as such requires a somewhat different way of thinking. The story told above also has a non-language-specific aspect (which is hopefully more interesting): keep it simple, and keep to the point. Getting too fancy will produce worse code, which will be of more interest to language implementors (my first attempts really ***should*** be faster) than to the author. On the other hand, moving around in circles is part of the human condition, and here, I did succeed in the end, and learned a lot more than I would have by taking a straight line path.