

## Re: Surrogate factoring, corrected algorithm

*Source:* <http://sci.tech-archive.net/Archive/sci.math/2005-02/6426.html>

---

*From:* Tim Peters ([tim.one\\_at\\_comcast.net](mailto:tim.one_at_comcast.net))

*Date:* 02/17/05

Date: Thu, 17 Feb 2005 01:04:16 -0500

[Tim Peters]

>> *Sorry, it didn't work for me. I started this time with my running*  
>> *example,  $M = 112554401 * 221667653$  ...*  
>> *This version didn't factor that  $M$  at any  $j$  in 1 thru 16, but did*  
>> *succeed once at  $j=17$ . In all, 62,320,128 gcds were tried through  $j=17$ ,*  
>> *with 1 success.*  
>>  
>>  *$j=17$  was painfully expensive. Then  $T^3 j^2 =$*   
>>  
>>  $2^9 3^6 5^3 7^3 11^3 17^2 191^3 7841^3 53633^3 56197^3$   
>>  $335417^3 14832053^3$   
>>  
>> *and so there are ... 55,050,240 gcds tried at  $j=17$  alone.*

[Bryan Olson]

> *With the "corrected" algorithm's suggested  $j = \text{floor}(M/2) (+1 \text{ if even})$ ,*  
> *I get a mere 24960 gcd trials. None of them work.*

Ya, afraid that's par for this course. In two previous versions of the algorithm, across a number of experiments the number of gcds required before finding a factor seemed worse than the expected number of gcds picking candidates at random would require. I didn't test that hypothesis rigorously, but had too much evidence in that direction piling up to ignore.

>> *Of course the most troublesome bit is that factors still aren't found*  
>> *at some  $j$ , so at least one of {proof, algorithm, implementation} is*  
>> *wrong.*

And in fact my implementation was wrong (added  $2j^2$  instead of  $2Tj^2$ ). Correcting that actually made the outcome for this  $M$  worse: it's now gone through all  $j$  in 1..18 without finding a factor of  $M$ .

> *I never saw a proof of the claim that one of the GCD's must be a proper*  
> *factor of  $M$ .*

James consistently claims to have proved this, although when I still tried to make sense of his "proof posts", I sure didn't see anything in them that

## sci.math: Re: Surrogate factoring, corrected algorithm

looked like a proof of this to me. I just run programs now — takes a lot less of my time, and the outcome has been the same anyway.

>> *Implementation note: it's easiest to generate all  $\langle f_1, f_2 \rangle$  splittings without trying to weed out reversals. For example, if  $\langle 3, 5 \rangle$  is generated, also generate  $\langle 5, 3 \rangle$ . But then trying to weed out reversals later, by keeping track of the ones already seen, can require an enormous amount of memory,*

> *How about requiring  $f_1 \leq f_2$ ? Just build  $f_1$ 's and toss any that get larger than the square-root of the product.*

What I have now does:

```
for all (f1_list, f2_list) in generate_splits(T3J2_list):
    f1 = product(f1_list)
    f2 = prodcut(f2_list)
    compute one gcd
```

That's very simple and regular. Saving a quarter of the product() calls would be a useful optimization, if I expected to run this enough in my lifetime to make up for the time I spent fiddling the code to do it <0.7 wink>.