Using C++ programming to implement a deterministic algorithm for integer factorization

Lillian Zeng, Dr. Ghaith A. Hiary

The Ohio State University – Department of Mathematics

Abstract

This research is an implementation of a deterministic algorithm for integer factorization (derived by Ghaith A. Hiary) using C++ programming. The most basic way for integer factorization is trial division. Though this algorithm provides a new deterministic approach for integer factorization, and can be used for the problem of factoring with high bits known. This implementation used the GNU MP Bignum Library to do arithmetic on very large numbers. Numbers from 15 digits to 27 digits were tested, and the new algorithm gradually beats trial division in speed competition. It successfully reduced the running time by about 67%.

Objectives

The main purpose of this research is to develop an efficient way for deterministic integer factorization, and to implement it using C++ programming language.

Techniques

- **Trial division**

  Trial division is a basic algorithm for integer factorization. In this research, trial division is not only used for speed comparison, but also is the first step in the new algorithm.

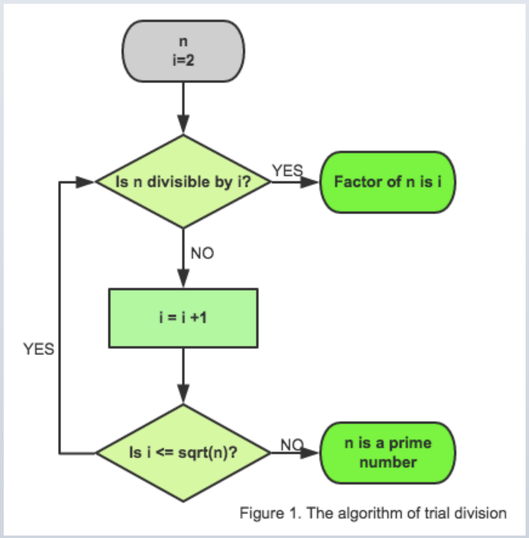

Figure 1. The algorithm of trial division

- **C++**

  C++ is a programming language that serves general purposes. It is available on many platforms and is widely used as software infrastructure and applications. This research uses C++ as the programming language because of its speed and flexibility.

- **GNU MP Bignum Library**

  GNU MP Bignum Library (i.e. GMP) is a well-developed library to compute huge operands. The precision limit only depends on the available memory in the machine GMP runs on. Because the main purpose of the algorithm is to find factors of huge operands, GMP is necessary

Comparison between Trial Division and New Algorithm

The main idea of the new algorithm is that it allows testing divisibility of n by all integers in a large block simultaneously rather than integer by integer like in trial division. Therefore, the new algorithm reduces the number of tests we need to run to find factor of n.

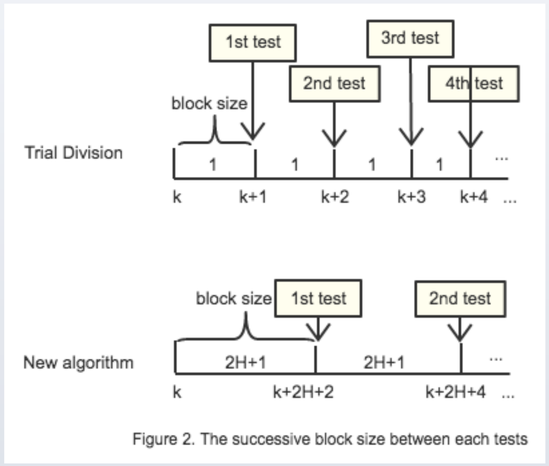

Figure 2. The successive block size between each tests

As shown in the Figure 2, for trial division, the block size between tests are 1. For the new algorithm, the block size is 2H + 1, where H is approximately sqrt(n)/(17n)$^{1/3}$ at the end, which is large for large n.

Main Function

```
int main()
{
  mpz_t n, sqrtN,x0, x, H, factor;
  mpz_init(factor);
  mpz_init_set_str (n,"1000000000013600000000003663",10);
  //x0 = min(ceil(pow(17*n),1/3.0)),sqrt(n))
    mpz_init (x0);
    mpz_mul_ui (x0,n,17);
    mpz_root (x0, x0,3);
    mpz_init (sqrtN);
    mpz_sqrt(sqrtN, n);
    if(mpz_cmp(x0, sqrtN)>0){mpz_set(x0,sqrtN);}
    //x = x0 + 2
    mpz_init (x);
    mpz_add_ui (x,x0,2);
    //H = 1
    mpz_init_set_str (H,"1",10);

    //trialDivision
    trialDivision (n, x0,factor);
    if(mpz_cmp_ui(factor,0) != 0){gmp_printf("The non-trivial factor of %Zd is %Zd \n",n,factor);}
    //The new algorithm
    else
      {
        findFactorLoop (x,n,sqrtN,H,factor);
        if(mpz_cmp_ui(factor,0) == 0){gmp_printf("%Zd is a prime number \n",n);}
        else{gmp_printf("The non-trivial factor of %Zd is %Zd \n",n,factor);}
      }
    mpz_clear(n);
    mpz_clear(sqrtN);
    mpz_clear(x0);
    mpz_clear(x);
    mpz_clear(H);
    mpz_clear(factor);
    return 0;
}
```

Figure 3. Main function of the program using new algorithm

Example Output

When input n is 1000000000013600000000003663:

```
The non-trivial factor of 1000000000013600000000003663 is 10000000000037
```

When input n is 1000000000000037:

```
100000000000037 is a prime number
```

Speed Competition between Trial Division and New Algorithm

The running time for both algorithms was tested. When the operands are less than 23 digits, trial division beats the new algorithm, though after 25 digits, the new algorithm gradually beats trial division. The running time of the new algorithm increases roughly like $n^{1/3}$ instead of sqrt(n) like in trial division. For example, if n is multiplied by 100, the running time of the new algorithm increases by a factor of $100^{1/3} = 4.64\ldots$, but the running time of trial division increases by a factor of sqrt(100) = 10. Thus, the new algorithm is much more efficient for operands with more than 25 digits.

Table 1. Running time for trial division and new algorithm

Digits		Trial Division	New Algorithm
15	10^{15}	0.13s	1.05s
17	10^{17}	1.26s	6.54s
19	10^{19}	14.16s	41.58s
21	10^{21}	2min52.98s	4min21.65s
23	10^{23}	11min26.37s	11min55.48s
25	10^{25}	1hr59min57.05s	1hr07min48.84s
27	10^{27}	20hrs(extrapolation)	6hr18min13.80s

Factoring with the high bits known

A practical application of the new algorithm is factoring with the high bits known. For example, when n is RSA-160 (a 160–digit composite number with 2 prime factors) and if we have already known the first 54 digits of a factor, then using this new algorithm, we only need to run one or two test(s) to find the factor.

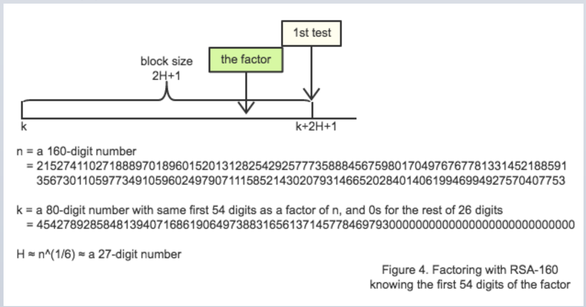

Figure 4. Factoring with RSA-160 knowing the first 54 digits of the factor

Conclusion

- Developed a more efficient way for integer factorization by reducing the running time by about 67%.
- Illustrated a comprehensive application of GNU MP Bignum Library
- Practical application: Factoring with the high bits known.

Reference

- Hiary, Ghaith A. "A Deterministic Algorithm for Integer Factorization." *Mathematics of Computation*. 85.300 (2016).