Report for project factoring, DD2440

Erik Helin
ehelin@kth.se

Jim Holmström
jimh@kth.se

November 16, 2011

Abstract

This report desribes the implementation of an algorithm that factors integers into prime factors. Factoring an integer into prime factors means finding a representation of an integer $n = p_1^{k_1} p_2^{k_2} \dots p_n^{k_n}$ where p_i are primes and k_i are integers. Three different algorithms were tried (separately and combined): trial division, Pollard's ρ algorithm and perfect power factorization. Pollard's ρ algorithm was tried with two different cycle detection algorithms. The algorithms where then benchmarked using 100 random integers. The report concludes that using Pollard's ρ algorithm with Floyd's cycle detection resulted in the best performance.

Contents

1 Introduction **3**

2 Problem statement **3**

3 Implementation **4**
3.1 Algorithms 4
3.1.1 Primality testing 4
3.1.2 General algorithm 4
3.1.3 Trial division 5
3.1.4 Pollard's ρ algorithm 5
3.1.5 Perfect power factorization 7
3.2 Implementation of algorithms 8
3.2.1 Handling failure 8
3.2.2 Implementing primality testing 8
3.2.3 Implementing trial division 9
3.2.4 Implementing Pollard's ρ algorithm 9
3.2.5 Implementing perfect power factorization 9

4 Results **9**
4.1 Benchmark 9
4.2 Test system 9
4.3 Test results 10
4.4 Kattis submission 12

5 Analysis **12**

6 Conclusion **13**

A Appendix A **15**

1 Introduction

The factoring project is about factoring integers into prime factors. That is, for a given number $n \in \mathbb{Z}$, n can be represented as $n = p_1^{k_1} p_2^{k_2} \dots p_m^{k_m}$ where p_i are prime numbers and k_i are integers. The task of the project is to find these prime numbers p_i and their exponents k_i. As an example, $20 = 2^2 \cdot 5^1$.

Today, there is no known (non-quantum) algorithm that effectively factors very large integers. In 2010, a group of researches managed to factor a number represented by 232 bits, but this took 2 years using hundreds of machines [1].

This report will describe the implementation of an algorithm for factoring integers. Section 2 states the exact problem defintion, then section 3 describes the various algorithms being tried. In section 4, these algorithms are benchmarked and the results are then analyzed in section 5.

2 Problem statement

Given 100 unknown integers, compute and print the prime number factorization for each integer. If number $n_i = p_1^{k_1} p_2^{k_2} \dots p_m^{k_m}$, then the program should print the following on `stdout`.

$$
\begin{array}{l}
\left.\begin{array}{l} p_1 \\ p_1 \\ \vdots \\ p_1 \end{array}\right\} \; k_1 \text{ times} \\
\left.\begin{array}{l} p_2 \\ p_2 \\ \vdots \\ p_2 \end{array}\right\} \; k_2 \text{ times} \\
\vdots \\
\left.\begin{array}{l} p_m \\ p_m \\ \vdots \\ p_m \end{array}\right\} \; k_m \text{ times}
\end{array}
$$

The time limit for the program is 15 seconds. The text `fail` must be printed on `stdout` if the program fails to factor a number. The performance of the program is measured in how many numbers the program are able to factor within the time limit. The program must print either the correct prime factorization or `fail` for each of the 100 numbers.

The program is *not* allowed to use the `time` system call.

3 Implementation

For the implementation of the program, several algorithms have been implemented (and combined in various ways). First, the algorithms will be described, then the implementations of the algorithms will discussed.

3.1 Algorithms

3.1.1 Primality testing

The input can contains prime numbers, and for these number, no factors can be found since they are prime. Therefore, before the input is factored, the input needs to be tested to see if it is prime. This was done by using the Miller-Rabin algorithm [2].

3.1.2 General algorithm

The task of factoring a given number n can be split into three tasks:

- Find a factor m of n
- Find the exponent k of factor m
- Print the number m k times

Therefore, the task of finding a prime factor m of n can be implemented in a subroutine *find-prime-factor*, resulting in algorithm 3.1.

Algorithm 3.1: FACTOR(n)

comment: General algorithm for factoring a number n

while $n \neq 1$
do
- $m \leftarrow$ *find-prime-factor*(n)
- $k \leftarrow 1$
- **while** $m \mid n$
  - **do** $k \leftarrow k + 1$; $n \leftarrow$ m / n
- **for** $i \leftarrow 0$ **to** k
  - **do** *print*(m)

3.1.3 Trial division

If an integer n is composite (not prime), then n must have a factor less than or equal to $\sqrt{n}$, since $\sqrt{n} \cdot \sqrt{n} = n$. Therefore, a naive algorithm for finding a prime factor to n is to try all primes less than or equal to $\sqrt{n}$ and check if any of them divides n. This yields algorithm 3.2.

Algorithm 3.2: TRIAL-DIVISION(n)

for $i \leftarrow 2$ **to** $\sqrt{n}$
do
- **if** $i \mid n$
  - **then return** (i)
- $i \leftarrow$ *next-prime*(i)

3.1.4 Pollard's ρ algorithm

In 1975, Jonh Pollard invented an algorithm that today is referred to as Pollard's ρ algorithm [3]. Pollard's ρ algorithm tries to find two numbers x and y such that $x \nmid n$ and $y \nmid n$, but $(x - y) \mid n$. Since n is composite, $n = pq$ for some p, q where $gcd(p, q) = 1$. By the Chinese Remainder Theorem [8], $x \bmod n$ can be represented by the pair $(x \bmod p, x \bmod q)$. When $x \equiv y \bmod p$, then $p \mid (x - y)$ and since $p \mid n$, $(x - y) \mid n$ and the only factor $(x - y)$ and n can share is p. Hence, $p = gcd(x - y, n)$. Therefore, when two numbers x and y are found such that $gcd(x - y, n) \neq 1$, a factor p

is found. Therefore, Pollard's ρ algorithm needs to generate numbers such that $x \not\equiv y \bmod n$ and $x \equiv y \bmod p$.

Iterating a polynomial formula $f(x) = x^2 + c \pmod{n}$ will yield numbers that might turn into a cycle, that is, the returned values will be repeated. The possibility that $g(x) = x^2 + c \pmod{p}$ will turn into a cycle is higher, since $p < n$. Therefore, by repeatedly applying $f(x)$ to some start value x_0, two numbers x and y such that $x = y \pmod{p}$ will be found if the function $f(x)$ enters a cycle. Then, since $p \leq \sqrt{n}$, there is a high probability that $x \not\equiv \bmod n$. The following two paragraphs describes two different ways for finding such a cycle.

Floyd's cycle detection Floyd's cycle detection is based on the idea of using two values x and y and for then repeatedly performing $x = f(x)$ and $y = f(f(y))$, that is, y will "move" twice as fast. If a cycle is found, then y will enter the cycle and then "catch up" with x when x enters the cycle. However, there is a change that $x = y = n$, and in this case, the algorithm fails.

Using Floyd's cycle detection results in algorithm 3.3.

Algorithm 3.3: POLLARD-FLOYD(n)

$d \leftarrow 1$
while $d = 1$
  do $\begin{cases} x \leftarrow f(x) \\ y \leftarrow f(f(y)) \\ d \leftarrow gcd(|y - x|, n) \end{cases}$
if $d = n$
  then return $(f)ail$
  else return (d)

Brent's cycle detection Brent's cycle detection is similar to Floyd's, but differ in how often y is updated. Instead of updating y at every iteration, y is updated whenever the number of iterations is a power of 2.

Using Brent's cycle detection results in algorithm 3.4.

Algorithm 3.4: POLLARD-BRENT(n)

$d \leftarrow 1$
$c \leftarrow 0$
while $d = 1$
do $\begin{cases} x \leftarrow f(x) \\ \textbf{if } c \equiv 0 \bmod 2 \\ \quad \textbf{then } y = x \\ d \leftarrow gcd(|y - x|, n) \\ c \leftarrow c + 1 \end{cases}$
if $d = n$
then return $(f)ail$
else return (d)

Note that the Pollard ρ algorithm does *not* find a prime factor of n, it justs a finds a factor. Therefore, in order to find a prime factor, the algorithm can be reapplied to the found factor in a recursive step. This results in algorithm 3.5.

Algorithm 3.5: POLLARD-PRIME(n)

$d \leftarrow pollard(n)$
while $is\text{-}prime(d) = false$
do $d \leftarrow pollard(d)$
return (d)

3.1.5 Perfect power factorization

Some composite numbers are perfect powers, that is, an integer $n = p^k$ for some prime p and some integer k. Such numbers can be factored by using Newtons method [4] by repeatedly trying to take jth root of n and check if it yields an integer for different integers j. Since n is a perfect power, j has to be larger than 2. j has to be less than $\lceil log_2(n) \rceil$, since $p \geq 2$. This

results in algorithm 3.6.

Algorithm 3.6: PERFECT-POWER-FACTORIZATION(n)

$j_{max} \leftarrow \lceil log_2(n) \rceil$
for $j \leftarrow j_{max}$ **to** 2
do $\begin{cases} r \leftarrow newton(n, j) \\ \textbf{if } r \in \mathbb{Z} \\ \quad \textbf{then return } (r) \end{cases}$

3.2 Implementation of algorithms

All algorithms were implemented in the C programming language using the GMP [7] library for all arithmetic operations involving large integers.

The different parts of the implementation will be discussed in the following sections.

3.2.1 Handling failure

Since it might be the case that the program won't be able to factor all the given numbers and an answer has to be given for all numbers, a way was needed to abort the factorization of a number.

Handling time-outs Since the program isn't allowed to use the `time` system call, this was implemented by using a cut-off value. Every iteration of the factoring algorithm decremented a counter with an initial max value. If the counter reached 0, that meant that the number couldn't be factored.

Handling partial factorization Since the program could find some of the factors, but not all of them due to a time-out, the factors couldn't be printed directly. Instead, they were stored in a pre-allocated array, and if the algorithm succeeded, the contents of the array was printed.

3.2.2 Implementing primality testing

The GMP `mpz_probab_prime_p` is using the Miller-Rabin test to test if a given number n is prime or not. `mpz_probab_prime_p` also takes a second parameter `reps` describing how many times Miller-Rabin should be run to increase the accuracy of the test. We used a value of 3.

3.2.3 Implementing trial division

Trial division was implemented by testing a fixed number of primes stored in an array. The array was iterated over in a loop, trying to divide the given number n with each prime p in the array.

3.2.4 Implementing Pollard's ρ algorithm

The implementation of Pollard's ρ algorithm followed algorithm 3.3 closely.

The algorithm used the polynomial function $f(x) = x^2 + 2 \pmod{n}$ as for generating pseudo-random numbers with the initial value of x set to 1.

When implementing the algorithm, the case when the algorithm fails due to $x = y = n$ has to be handled. The way this was done was to restart the algorithm, but with a different polynomial function $g(x) = x^2 + c \pmod{n}$ and a new initial value i. The values i and c were chosen randomly when restarting.

3.2.5 Implementing perfect power factorization

The implementation of perfect power hashing made use of the GMP function `mpz_root` for finding the n:th root, even though using this function was probably against the rules. The reason for this was to benchmark a solution making use of perfect power factorization, and if a performance increase noticed, implement Newton's method ourself. The implementation followed algorithm 3.6 closely.

4 Results

4.1 Benchmark

To benchmark the algorithm we measured the execution time for 100 randomly sampled semiprimes $p * q = n$ with a bitlength of $p, q = (u, v)$. The factorization is symmetric on p, q, therefore we only did measurements where $u \leq v$. The values $(u, v) \in 4[1, 12]^2$ where used in the tests.

We also did some comparisons between some algorithms and their parameters by directly getting points from `KATTIS` [5] on the problem `oldkattis:factoring` [6].

4.2 Test system

All measurements was made on a computer using Ubuntu 10.04 LTS (64-bit) on a Intel Xeon X3470 CPU @ 2.93GHz with 4 cores (8 threads) and with

16GB RAM. The code was compiled with `gcc-4.4.3` using the flag `-g` and the library `GMP-4.3.2`.

The execution time was measured with the `USER` time from the `time` command.

4.3 Test results

These tests were solely based on the score and running time on the `KATTIS` problem `oldkattis:factoring`.

Primality testing Using only the Miller-Rabin tests do determine if a number is prime and if so print the number. The results can be seen in table 1

Score	Running time
1	0.01

Table 1: Only using the Miller-Rabin test

Trial division and primality test The next version uses trial division (as described in algorithm 3.2) and the Miller-Rabin primality test. A fixed number of primes were used for trial division, but the number of primes we used varied. The results can be seen in table 2.

Pollard's ρ Pollard's ρ algorithm was tested in a few different ways. The different combinations used were:

A) Only Pollard's ρ with Floyd's cycle detection and primality test

B) Only Pollard's ρ with Brent's cycle detection and primality test

C) Pollard's ρ with Brent's cycle detection, trial division and primality test

D) Pollard's ρ with Floyd's cycle detection, handling perfect powers, trial division and primality test

The results can be seen in table 3.

Given that Floyd's cycle detection seemed superior to Brent's cycle detection, we, Pollard's ρ algorithm with Floyd's cycle detection was tested with two different random value generators. The first one used was GMPs `mp_urandomm` and the second being used was `rand`. The results can be seen in table 4.

Number of primes	Score	Running time
10000	21	0.05
7500	21	0.05
5000	21	0.04
2500	21	0.02
1875	21	0.02
1718	21	0.02
1562	20	0.02
1250	19	0.02
625	18	0.01
312	18	0.01
156	15	0.01
78	14	0.01
39	13	0.01
19	9	0.01
10	7	0.01
5	4	0.01
0	1	0.01

Table 2: Using trial division and primality test

Variant	Score
A	72
B	64
C	71
D	72

Table 3: Comparing different implementations Pollard's ρ algorithm

Variant	Score
`mp_urandomm`	72
`rand`	75

Table 4: Comparing different random generators

Algorithm cut-off As a final test, Pollard's ρ algorithm was tested with different cut-off values and the result can be seen in table 5 (see section for 3.2.1 more details about the cut-off value).

Note: this algorithm used `mp_urandomm`, therefore the maximum score was 72.

Cut-off value	Score	Running time
190000	72	14.32
172500	71	13.05
155000	71	11.93
137500	71	10.97
120000	70	10.01
110000	70	9.50
100000	69	8.70
90000	66	8.21
80000	65	7.48
70000	65	6.77
60000	63	6.00
50000	58	5.22
40000	56	4.28
35000	55	3.83
30000	50	3.35
25000	49	2.86
20000	44	2.37
15000	41	1.82
10000	38	1.25
5000	35	0.67
0	1	0.01

Table 5: Comparing different cut-off values

4.4 Kattis submission

The best Kattis submission has the id 256166.

5 Analysis

The results from table 1 shows that it is good to use a primality test as an initial test, since one number was prime. The running time also shows that is a fast operation.

The results in table 2 then shows that trial division alone does not suffice as an algorihtm for factoring integers. It it also worth pointing out that there was no gain in increasing the number of primes used from 1718 to 10000. This is because the prime factors of most of the number are larger than 10000th prime, rendering the trial division test useless.

When comparing the different implementations of Pollard's ρ algorithm, one clearly sees that in our case, Floyd's cycle detection method outperforms Brent's cycle detection. The reason for this might be that the number used at Kattis suits Floyd's cycle detection algorithm better (Brent's algorithm ended up with $x = y = n$ more often that Floyd's). Due to this, we decided to use Floyd's cycle detection.

It is also interesting that the combined use of trial division, perfect power factorization and Pollard's ρ algorithm shows no performance increase compated to using only Pollard's ρ algorithm. This is probably due to the fact that the numbers that trial division and perfet power factorization can factor can also be quickly factored with Pollard's ρ algorithm. When comparing Brent's cycle detection with and without all trial division and perfect power hashing, the two extra algorithms did help. This further justifies the thought that Brent's cycle deteciton is weaker than Floyd's on the number available on Kattis. Since there were no performance gains in using trial division and perfect power hashing with Floyd's cycle detection, we decided to remove these two steps.

Since we didn't gain any increased results from using perfect power hashing, we decicded to not implement Newton's method.

When comparing the different random algorithms used when restarting Pollard's ρ algorithm, the reason that `rand` is faster is probably because it is much simpler and doesn't deal with as large number as `mp_urandomm` does.

It is not surprising that the number of factored number increases as the cut-off increases, since this gives the factoring algorithm more time. These numbers also shows the importance of tweaking the implementation to be as fast as possible, since a fast implementation can use a higher cut-off.

6 Conclusion

The final algorithm being used was Pollard's ρ algorithm with Floyd's cycle detection toghether with `rand` for generating random numbers, since this combination achieved the highest result.

Given more time, we would have liked to invest more time in increasing the performance of the cycle detection part of Pollard's ρ algorihtm. As a start, the Wikipedia article [3] mentions some adjustments that can be done to speed up the cycle detection.

We would also have liked to come up with a better algorithm for noticing numbers that our algorihtm probably wouldn't be able to factor. This would have saved us time for factoring other numbers that we might haven been

able to factor.

References

[1] Kleinjung, Aoki, Franke et al., *Factorization of a 768-bit RSA modulus*, International Association for Cryptologic Research

[2] Wikipedia, *Miller-Rabin primality test*, `http://en.wikipedia.org/wiki/Miller-Rabin_primality_test`

[3] Wikipedia, *Pollard's rho algorithm*, `http://en.wikipedia.org/wiki/Pollard's_rho_algorithm`

[4] Wikipedia, *Newton's method*, `http://en.wikipedia.org/wiki/Newton's_method`

[5] Kattis, *Kattis*, `https://kth.kattis.scrool.se`

[6] Kattis, *Integer Factorization*, `https://kth.kattis.scrool.se/problems/oldkattis:factoring`

[7] GMP, *The GNU MP Bignum Library*, `http://http://gmplib.org/`

[8] Wikipedia, *Chinese remainder theorem*, `http://http://en.wikipedia.org/wiki/Chinese_remainder_theorem`

A Appendix A

```
#include <stdio.h>
#include <stdlib.h>
#include <gmp.h>

#define NUM_INPUT 100

#define MAX_FACTORS 512

#define MAX_NUMBER_OF_TRIES 260000 //190000 (with 300 (and with 900) trial division) /

#define NEXT_VALUE(r, x, n, c) \
    mpz_set_ui(cp, c);          \
    mpz_addmul(cp, x, x);       \
    mpz_mod(r, cp, n)

#define IS_ONE(x)  mpz_cmp_ui(x, 1)
#define IS_PRIME(x) mpz_probab_prime_p(x, 3)
#define FAILED(x) mpz_cmp_ui(x, 0) == 0
#define ADD_FACTOR(factors, num_factors, factor) \
    do {                                         \
        mpz_set( factors[num_factors], factor);  \
        num_factors++;                           \
    } while (0)

#define RAND ((int) (((double) rand()) / ((double)RAND_MAX)) * 100)
#include "primes.h"

mpz_t cp;

#include "primes.h"

void pollard_brent(mpz_t x_1, mpz_t x_2, mpz_t d, mpz_t n, int init, int c, int counte
{
    int i;
    for(i = 0; i < 10000; ++i) {
        if(mpz_divisible_ui_p(n, primes[i])) {
            mpz_set_ui(d, primes[i]);
            return;
        }
    }

    unsigned long power = 1;
    unsigned long lam = 1;

    mpz_set_ui(x_1, init);
```

```
    mpz_set_ui(d, 1);
    while(mpz_cmp_ui(d, 1) == 0) {
        if (power == lam) {
            mpz_set(x_2, x_1);
            power *= 2;
            lam = 0;
        }

        mpz_mul(x_1, x_1, x_1);
        mpz_add_ui(x_1, x_1, c);
        mpz_mod(x_1, x_1, n);

        mpz_sub(d, x_2, x_1);
        mpz_abs(d, d);
        mpz_gcd(d, d, n);

        lam++;

        if (counter < 0){
            mpz_set_ui(d, 0);
            return;
        } else {
            --counter;
        }
    }

    if(mpz_cmp(d, n) == 0) {
        return pollard_brent(x_1, x_2, d, n, rand(), rand(), counter);
    }
}

void pollard(mpz_t x_1, mpz_t x_2, mpz_t d, mpz_t n, int init, int c, int counter)
{
    if(mpz_divisible_ui_p(n, 2)) {
        mpz_set_ui(d, 2);
        return;
    }

    mpz_set_ui(x_1, init);
    NEXT_VALUE(x_1, x_1, n, c);
    NEXT_VALUE(x_2, x_1, n, c);

    mpz_set_ui(d, 1);
    while (mpz_cmp_ui(d, 1) == 0) {
        NEXT_VALUE(x_1, x_1, n, c);
        NEXT_VALUE(x_2, x_2, n, c);
        NEXT_VALUE(x_2, x_2, n, c);

        mpz_sub(d, x_2, x_1);
```

```
        mpz_abs(d, d);
        mpz_gcd(d, d, n);

        if (counter < 0){
            mpz_set_ui(d, 0);
            return;
        } else {
            --counter;
        }
    }

    if(mpz_cmp(d, n) == 0) {
        return pollard(x_1, x_2, d, n, primes[RAND], primes[RAND], counter);
    }
}

int factor(mpz_t x_1, mpz_t x_2, mpz_t d, mpz_t n, mpz_t tmp, mpz_t *factors)
{
    int num_factors = 0;

    while (IS_ONE(n) != 0 && IS_PRIME(n) == 0) {
        /*pollard(x_1, x_2, d, n, 2, 1, MAX_NUMBER_OF_TRIES);*/
        pollard_brent(x_1, x_2, d, n, 2, 3, MAX_NUMBER_OF_TRIES);

        if(FAILED(d)) {
            return 0;
        }

        while (IS_ONE(d) != 0 && IS_PRIME(d) == 0) {
            /*pollard(x_2, x_2, tmp, d, 2, 1, MAX_NUMBER_OF_TRIES);*/
            pollard_brent(x_1, x_2, d, n, 2, 3, MAX_NUMBER_OF_TRIES);

            if(FAILED(tmp)) {
                return 0;
            }

            mpz_set(d, tmp);
        }
        do {
            ADD_FACTOR(factors, num_factors, d);
            mpz_divexact(n, n, d);
        } while (mpz_divisible_p(n, d));
    }
    if(IS_ONE(n) != 0) {
        ADD_FACTOR(factors, num_factors, n);
    }
    return num_factors;
}
```

```
void print_factors(mpz_t *factors, int num_factors)
{
    int i;

    if (num_factors == 0) {
        printf("fail\n");
    } else {
        for (i = 0; i < num_factors; ++i) {
            gmp_printf("%Zd\n", factors[i]);
        }
    }
    putchar('\n');
}

int main(void)
{
    int i, num_factors;
    mpz_t n, d, tmp, x_1, x_2;
    mpz_init(n);
    mpz_init(d);
    mpz_init(tmp);
    mpz_init(x_1);
    mpz_init(x_2);
    mpz_init(cp);

    srand(1337);

    mpz_t *factors = malloc(sizeof(mpz_t) * MAX_FACTORS);

    for(i = 0; i < MAX_FACTORS; ++i) {
        mpz_init(factors[i]);
    }

    for (i = 0; i < NUM_INPUT; ++i) {
        mpz_inp_str(n, stdin, 10);
        num_factors = factor(x_1, x_2, d, n, tmp, factors);
        print_factors(factors, num_factors);
    }

    return 0;
}
```