Project 1: Factoring

by

Mårten Pålsson 8802030539 <mpals@kth.se>
Henrik Sohlberg 8805080531 <hsoh@kth.se>

KTH Computer Science and Communication
DD2440 - avalg12

2012-10-24

Abstract

The subject of this report is the creation of a program that factorizes numbers into their constituent primes and also different ways of doing so. The requirements of our program was that it had 15 seconds to factorize 100 numbers up to 100 bits in length using at most 32 MB of memory. Our solution used a combination of several methods to factorize integers. During the project we tried a few different variations but for our final version we used Pollard's rho algorithm using Floyds Circle detection algorithm together with Trial Division and finding Perfect Powers. What we discovered were that Pollard's rho is good for the factorization of integers with small prime factors and that the most important feature of a program of this type, besides the algorithm itself, is to be able to control how the limited execution time is spent.

Statement Of Collaboration

Both Henrik Sohlberg and Mårten Pålsson have participated equally in the process of finding a solution to the main problem of this project, how to factorize large numbers. Later in the process (the implementation phase) Mårten implemented the Trial Division method, Henrik implemented the Perfect Power and the Fermat's Factorization methods, while both wrote the code to the Pollard's rho method.

Both contributed to the content of the whole report. Mårten wrote the sections: Introduction, parts of Background, parts of Method, Results, Discussion, and Conclusion. Henrik wrote parts of the background and parts of the method.

Contents

1 Introduction

1.1 Introduction

The purpose of this project was to investigate different methods and algorithms for prime factorization. Prime factorization, or Prime decomposition as it is also called, is the practice of splitting an integer into a series of prime factors that when multiplied together form the original number [1]. There are many algorithms for factoring integers into primes but the one we have used and will discuss in the greatest detail in this report is Pollard Rho's algorithm for factoring numbers.

1.2 Problem Statement

As stated above the problem in this case consists of factoring numbers into their constituent primes. The number of integers to be factorized are 100 and the time limit of doing so is 15 seconds. The program will need to be able to handle integers up to 100 bits in length. There is also a limit of 32 MB of memory that can be allocated to the program.

2 Background

This section briefly discuss some algorithms used to factorize numbers.

2.1 Trial Division

Trial Division is a naive approach where an integer is divided by a series of precomputed prime numbers to discover small factors quickly. If the rest of such a division is 0 then we know that the prime in question is a factor of the given integer. The resulting quotient is then sent back to Trial Division to try to break it down even further. This is a quick way of finding small prime factors and reducing the magnitude of the given Integer.

2.2 Perfect Powers

Perfect power refers to integers that can be written on the form $N = m^k$ where $m > 1$ and $k \geq 2$ [2]. If N is found to be a perfect power it is only necessary to continue the factorization using m as the input and store any found factors k times before printing. This is because any factors found in m would have been found k times in N since m is the k^{th} root of N. This further lessens the magnitude of N and reduces thus the computation time.

2.3 Fermat's Factorization Method

Pierre de Fermat's method to factorize a number is to represent it as the difference between two squares.

$N = a^2 - b^2 = (a + b)(a - b)$

To factorize N, start with a value $\lfloor\sqrt{N}\rfloor + 1 = a$. Then $b = a^2 - N$, if b is a perfect square, you have found two factors $(a + b)$ and $(a - b)$ to N.

2.4 Cycle Detection

Cycle detection refers to the problem of finding a cycle in a sequence of iterated function values. [3]

2.4.1 Floyd's Cycle Finding Algorithm

Also known as "tortoise and the hare" algorithm [3] which finds a cycle in a sequence. It uses two pointer, where one pointer (the hare) moves with double the speed to the other pointer (the tortoise) through the elements of the sequence. The algorithm detects the occurrence when the hare has sprinted exactly one whole cycle further than the tortoise, and uses this information to calculate the length of the found cycle. If the hare reaches the end of the sequence first, there is no cycle.

2.5 Pollard's Rho Algorithm

Pollard's rho algorithm (PR) is especially effective at finding small prime factors and is based on the concept of the existence of an subgroup to the group N [4][5]. The algorithm first makes the assumption that for an integer N there exists a factor d. Secondly, it uses a pseudo-random function to generate "random" numbers together with an implementation of the Floyd's cycle finding algorithm, sec. 2.4.1. An iteration consist of generating two random numbers, x_k (x) and $x_{k/2}$ (y), and performing a greatest common divisor (GCD) operation between the absolute value $(x - y)$ and the number to be factorized, N. PR succeeds when $\text{GCD}(|x - y|, N) = d$ where

1. $d > 1$, and $d \neq N$

(*Behind the scene*), we know that N is a multiple of d. When $x = y \mod d \rightarrow (x - y)$ is a multiple of d. If $\text{GCD}(|x - y|, N) > 1)$ and $\neq N$ we find a factor.

The time complexity of PR using a random function $f \mod p$ to find the factor p is, according to Hastad [8], $O(\sqrt{p}) = O(\sqrt{N})$. Since p is unknown, this function is not possible to use, but experience shows that the "random" function $x_{i+1} \equiv {x_i}^2 + 1$ also behaves this way.

2.5.1 A Pollard And Brent Improvement

Pollard and Brent [4] found out that if $\text{GCD}(a, n) > 1$, then $\text{GCD}(ab, n) > 1$ for any positive integer b. This yields that instead of computing a GCD operation each iteration of PR, it is sufficient to only perform a GCD every 100th iteration using $\text{GCD}(z, n)$, where z is the product of 100 consecutive $|x - y| \mod N$.

2.6 GMP library

To be able to work with and factorize large integers we have used Bjorn Granlunds GmpLib. This library gives us the functionality we need to perform integer operations with very large integers. All integers to be factorized and are stored in *mpz_t* structs to allow them to be up to 100 bits in length. The two functions we have used from this library that are of special note are *mpz_probab_prime()* [6] and *mpz_perfect_power_p()* [7]. *mpz_probab_prime()* is used to find whether an integer is prime or not, and *mpz_perfect_power()* is used to find out if an integer is a perfect power or not. All other functions except integer operations we have written ourselves.

3 Method

This section describes our approach in text, sec. 3.1, and in pseudo code, sec. 3.2.

3.1 Our Approach

For our final implementation we have used PR, sec. 2.5, for prime factorization with some slight modifications. During the entire process when a factor is found it is added to a factor list that is printed at program termination.

First we check whether the integer received is prime or not. If it is not we use Trial Division, sec. 2.1, with the first 10000 primes to attempt to quickly break down the given integer N into smaller components. This is to reduce the magnitude of N and thus lessen the time spent factoring. If it is prime it is stored as the only factor of itself and the program stops.

Then we attempt to find out whether the resulting integer N from trial division is a perfect power, sec. 2.2, on the form $N = m^k$ or not. If N is prime it is stored as a factor and the the function returns. If it is not prime but it is a perfect power we send the resulting $\sqrt{m}$ to PR with k as a parameter specifying the order of the root. Otherwise we send N to PR with $k = 1$ as the order of the root.

Then when we have broken down N as much as we can we run it through PR. If N is prime when it reaches PR it is added as a factor k times and the function returns.

Since we have time limit on our program and we want to factor as many numbers as possible we have put a limit on the amount of time spent on factoring each integer N. This time limit is implemented by only allowing PR to run an already specified amount of times before returning fail instead of a prime factor. Also instead of calculating the GCD$(|x-y|, N)$ in every iteration of the loop we store 99 multiplications of $|x-y| \mod N$ and then calculating a single GCD. When (and if) PR finds a factor it sends it to our perfect square function with any parameter k it has picked up on the way and the process starts over. This process is repeated until either we have run the specified max amount of loops or we have factored N into its prime factors. If we run out of time or a factor cannot be found fail is returned and the program continues on to the next number. Otherwise the found factors are printed and the whole process is repeated for the next number.

3.2 Pseudo code

```
$Factorize\ a\ integer\ N$

Initialize a vector $V$
**for** $i \leftarrow 1 \rightarrow 100$ **do**
    $N \leftarrow$ input integer
    clear $V$
    $fail \leftarrow false$
    $Factorize(N, 1)$
    **if** $\neg fail$ **then**
        Print all the elements in $V$
    end if
end for
return


$Factorize(N, number)$
if $mpz\_probab\_prime(N, 10)$ then
    Add $N$, $number$ times, to $V$
    return
end if
$primes$ = vector of precalculated primes
for $i \leftarrow 0 \rightarrow sizeof(primes)$ do
    if $primes[i]|N$ then
        Add $primes[i]$ to $V$
        $N \leftarrow N/primes[i]$
        if $N = 1$ then
            return
        end if
    end if
end for
$perfect\_powers(N, number)$
return


$perfect\_powers(N, number)$
if $mpz\_probab\_prime(N, 10)$ then
    Add $N$, $number$ times, to $V$
    return
end if
if mpz_perfect_power_p(N) then
    $found \leftarrow false$
    $i \leftarrow 2$
    while $\neg found$ do
        if $\sqrt[i]{N}$ = a even integer then
            $found \leftarrow true$
            $perfect\_powers(\sqrt[i]{N}, i \cdot number)$
            return
        end if
        $i{+}{+}$
    end while
else
    $Pollard's\ rho\ (N, number)$
end if


$Pollard's\ rho\ (N, number)$
if $mpz\_probab\_prime(N, 10)$ then
    Add $N$, $number$ times, to $V$
    return
```

```
end if
$x \leftarrow 7$
$y \leftarrow 7$
$sum \leftarrow 1$
$d \leftarrow 1$
$counter \leftarrow 0$
$LOOPS \leftarrow 380{,}000$
while $d = 1 \wedge counter < LOOPS$ do
    $counter{+}{+}$
    $x \leftarrow f(x, N)$
    $y \leftarrow f(f(y, N), N)$
    $sum \leftarrow sum \cdot |x - y| \mod N$
    if $counter \mod 100 = 0$ then
        $d \leftarrow \mathrm{GCD}(sum, N)$
        $sum \leftarrow 1$
    end if
end while
if $d \neq N || counter = LOOPS$ then
    $fail \leftarrow true$
    return "fail"
else
    $perfect\_powers(N/d, number)$
    $perfect\_powers(d, number)$
    return
end if
```

```
$f(x, N)$
$x \leftarrow x^2$
$x \leftarrow x + 1$
$x \leftarrow x \mod N$
return
```

4 Results

Below are accounts of the changes in the submissions where mayor implementation changes were made during the course of this project, and the results of their Kattis [9] submissions. If something is not described in one of the implementation accounts it has remained as it was in the last implementation. E.g. the initialization values of x and y in our first and second implementation were the same and as such we only mention them in the first implementation, but not the second.

4.1 First Implementation

Kattis submission ID: 306422
Description: Our first implementation used a system timer of 14 seconds to make sure we did not fail because of time issues. We also had no perfect power check and we only used the primes found between $[0, 1000]$ for our Trial Division. The initialization values x and y in PR were $= 2$ and $c = 1$.

Number solved: 50
CPU time used: 13.80 seconds

4.2 Second Implementation

Kattis submission ID: Several submissions
Description: A perfect square check using Fermats factorization method was implemented. Perfect square checks if a square can be found from N and if so sends the two squares on to PR . The amount of primes used in Trial Division was increased to the first 3000 primes.

Number solved: ≤ 47
CPU time used: variation between $3 - 14$ seconds

4.3 Third Implementation

Kattis submission ID: 306439
Description:Fermats factorization method was removed from the Perfect Square check as well as using system time to limit the running time of our program. Instead a maximum of 500 loops for PR was implemented.

Number solved: 54
CPU time used: 9.81 seconds

4.4 Fourth Implementation

Kattis submission ID: 306483
Description: The max loops of PR were increased to 50 000 and the initialization values of x and y were set to 7.

Number solved: 59
CPU time used: 5.10 seconds

4.5 Fifth Implementation

Kattis submission ID: 306631
Description: PR was allowed to perform a maximum of 4 runs for a single number and for a maximum of 40,000 loops each. The initialization values of x and y were set to 8 - current rho iteration.

Number solved: 64
CPU time used: 13.44 seconds

4.6 Sixth Implementation

Kattis submission ID: 306665
Description: PR allowed to run a max of 4 times at 42,000 loops each. Trial Division uses first 10,000 prime numbers.

Number solved: 65
CPU time used: 13.41 seconds

4.7 Seventh Implementation

Kattis submission ID: 307128
Description: Instead of Perfect Square making two calls to PR (one with each square root) it only makes one call but with a parameter specifying how many roots PR is to add to the factor list. PR is only allowed to run 1 time for a number and using a maximum of 202,000 loops. x and y are initiated to 7.

Number solved: 74
CPU time used: 14.90 seconds

4.8 Eighth Implementation

Kattis submission ID: 307972
Description: Perfect square is changed to perfect power and finds roots up to order 10. A check is added so our program only tries to factorize numbers that are smaller than 100 bits. Maximum loops for PR is increased to 222,000.

Number solved: 71
CPU time used: 14.98 seconds

4.9 Ninth Implementation

Kattis submission ID: 308558 (our best submission)
Description: Perfect power finds roots up to order k. In the main PR loop 99 multiples of $|x - y| \mod N$ are stored and one GCD is calculated every 100 loops instead of calculating a GCD in every loop. PR is allowed to run for a maximum of 380,000 loops. Numbers of any bit length are allowed.

Number solved: 79
CPU time used: 14.58 seconds

5 Discussion

Changing from using system time to keep our execution time in check to using a limited amount of loops for PR freed a lot of time for our program. This becomes apparent between the 1st and 3rd implementation. This is because now every number we received get the same amount of time instead of the whole program only getting 14 seconds in which to run. This means that instead of potentially getting stuck on a large complicated number for a large amount of time we can instead solve several smaller less complicated numbers.

Increasing the amount of precomputed prime numbers used in Trial Division is useful to quickly factorize numbers with many small factors. It does not however affect the computation time to the same degree in the factorization of numbers with larger prime factors as can be seen between the 5th and 6th implementation. This conclusion can be drawn since between the 5th and 6th implementation we only factorize one more number in the same amount of time, significantly more numbers are only factorized when other optimization have been implemented. This means that most numbers that are greatly affected by Trial Division have already been factorized when only using 3000 precomputed primes.

Having the initial values of x and y changing dynamically did not have any greater effect on the solver either as we can see between the 4th and 5th implementation. While the amount of successfully factorized numbers increased the time doing so also increased greatly.

The single greatest time gain can be seen between the 6th and 7th implementation when we only allowed PR to run once. This meant that even less time was spent on numbers that our algorithm was unable to factorize and left more time for numbers that we could. Through testing we found the optimal starting value of x and y to be 7.

Only factorizing one of the roots when a perfect power is found is also a time saver as can be seen between the 6th and 7th implementation. This is because now you only have to factorize an integer with half the magnitude (in the case of a square root, less if a root of a higher order is found). This gains us a lot of time if a perfect power is found.

Limiting the solver to smaller numbers was found to be a bad idea as can be seen between the 7th and 8th implementation. This means that our solver does not have any major problems solving number up to 100 bits in length in the allotted time and time can be spared for larger and more complicated numbers.

The final time gain found was in storing up multiples of $|x-y| \mod N$ and calculating fewer GCD operations. Since finding the GCD of two numbers is time consuming it is important to this as few times as possible. Reducing the amount of times we calculated a GCD speeded up our algorithm significantly and allowed us to solve 5 more numbers.

When we implemented Fermats Factorization algorithm it only slowed our solution down. So while Fermats algorithm is an interesting concept it was found to be too costly to be effectively used in combination with our implementation and the input.

6 Conclusion

The single greatest factor of our success was the fact that we were able to control the time spent on each number. It was when we were able to do this that other optimizations started taking effect on our computation time. This is because we could then control the amount of time spent on numbers that we are never going to be able to solve, i.e we will not get stuck.

Another important conclusion we can draw is that Pollard's rho is best at factoring relatively simple numbers (numbers with small prime factors). We can see the proof of this when we limited the length of the given integers to 100 bits. When we did this we factorized 8 less numbers than before, which is a significant amount. This means that length is not a deciding factor for Pollard's rho. Also since we use Trial Division, numbers who are Perfect Powers with relatively small roots (not one of the first 10,000 primes) are not a big problem either. This leads to the conclusion that we have difficulties factoring composite numbers with large prime factors and large pseudo prime numbers.

References

[1] Prime Factorization [Homepage on the Internet]. [cited 10/10/23]. Available from: http://mathworld.wolfram.com/PrimeFactorization.html

[2] Perfect Power [Homepage on the Internet]. [cited 10/10/23]. Available from: http://mathworld.wolfram.com/PerfectPower.html

[3] Cycle Detection [Homepage on the Internet]. [cited 10/10/23]. Available from: http://en.wikipedia.org/wiki/Cycle_detection

[4] Pollard's rho algorithm [Homepage on the Internet]. [cited 10/10/23]. Available from: http://en.wikipedia.org/wiki/Pollard\%27s_rho_algorithm

[5] Pollard's Rho Method [Homepage on the Internet]. [cited 10/10/23]. Available from: http://www.csh.rit.edu/~pat/math/quickies/rho/

[6] Number Theoretic Functions [Homepage on the Internet]. [cited 10/10/24]. Available from: http://gmplib.org/manual/Number-Theoretic-Functions.html#Number-Theoretic-Functions

[7] Root Extraction Functions [Homepage on the Internet]. [cited 10/10/23]. Available from: http://gmplib.org/manual/Integer-Roots.html#Integer-Roots

[8] Notes for the course advanced algorithms [Homepage on the Internet]. [cited 10/10/23]. Available from: http://www.csc.kth.se/utbildning/kth/kurser/DD2440/avalg07/algnotes.pdf

[9] KATTIS [Homepage on the Internet]. [cited 10/10/24]. Available from: https://kth.kattis.scrool.se/

A C++ code

```cpp
#include <gmp.h>
#include <gmpxx.h>
#include <iostream>
#include <math.h>
#include <queue>
#include <vector>
#include <time.h>
#include "primes.h"
using namespace std;
void f(mpz_t x, mpz_t N);
void factorize(mpz_t N, int number);
void perfect_squares(mpz_t N, int number);
vector<mpz_class> output_mpz;
vector<int> output_int;
int fail;
static int LOOPS =  380000;

void addFactors(mpz_t N, int i) {
  for (int j = 0; j<i; j++) {
      output_mpz.push_back(mpz_class(N));
  }
}
void addIntFactors(int factor, int number) {
  for(int i = 0; i<number;i++) {
      output_int.push_back(factor);
  }
}

void pollard_roh(mpz_t  N, int number){
  if(mpz_probab_prime_p(N,10)){
    addFactors(N, number);
    mpz_clear(N);
    return;
  }
  mpz_t x,y,d, abs, sum;
  mpz_init_set_ui (x, 7);
  mpz_init_set_ui (y, 7);
  mpz_init_set_ui(d, 1);
  mpz_init (abs);
  mpz_init_set_ui(sum,1);
  int counter = 0; int i = 0;
  while(!mpz_cmp_ui(d,1) && counter < LOOPS ){
    f(x, N);
    f(y, N); f(y, N);
    mpz_sub(abs, x, y);
    mpz_abs(abs, abs);
    mpz_mul(sum, sum, abs);
    mpz_mod(sum, sum, N);
    i++;
    if(i == 100) {
      mpz_gcd(d, sum, N);
      mpz_set_ui(sum, 1);
      i = 0;
    }
    counter++;
  }
  mpz_clear(x); mpz_clear(y);
  if(!mpz_cmp(d,N) || counter == LOOPS) {
     mpz_clear(d); mpz_clear(abs);
    cout << "fail" << endl;
    mpz_clear(N);
    fail = 1;
    return;
  }
  mpz_cdiv_q(abs, N, d);
  mpz_clear(N);
  perfect_squares(abs, number); perfect_squares(d, number);
}

void f(mpz_t x, mpz_t N) {
  mpz_pow_ui(x, x, 2);
  mpz_add_ui(x,x,1);
  mpz_mod(x,x,N);
}
void perfect_squares(mpz_t N, int number) {
  if(mpz_probab_prime_p(N,10)){
    addFactors(N, number);
    mpz_clear(N);
    return;
  }
  if(mpz_perfect_power_p(N)) {
    mpz_t root; mpz_init(root);
    int found = 0; int i = 2;
    while(!found) {
      if(mpz_root(root, N, i)) {
        found = 1;
        mpz_clear(N);
        perfect_squares(root, number*i);
      }
      i++;
    }
  }
  else {
    pollard_roh(N, number);
  }
```

```
}

void factorize(mpz_t N, int number){
  if(mpz_probab_prime_p(N,10)){
    addFactors(N, number);
    mpz_clear(N);
    return;
  }
  mpz_t tmp; mpz_init(tmp);
  mpz_t curr; mpz_init_set(curr,N);
  for(int i = 0; i < NUM_PRIMES; i++){
    if(mpz_cdiv_q_ui(tmp,curr,primes[i]) == 0){
      addIntFactors(primes[i], number);
      if(!mpz_cmp_ui(tmp,1)){
        mpz_clear(tmp);
        mpz_clear(N);
        return;
      }
      mpz_set(curr,tmp);
      i= -1;
    }
  }
  mpz_clear(tmp);
  mpz_clear(N);
  perfect_squares(curr, number);
}
void print_output() {
  for(unsigned int i = 0; i<output_mpz.size(); i++) {
    cout<<output_mpz[i]<<endl;
  }
  for(unsigned int j = 0; j<output_int.size(); j++) {
    cout<<output_int[j]<<endl;
  }
}
int main(){
  for(int i = 0; i < 100;i++){
    mpz_t N; mpz_init (N);
    gmp_scanf("%Zd",N);
    fail = 0;
    output_mpz.clear();
    output_int.clear();
    factorize(N, 1);
    if(!fail) {
      print_output();
    }
    cout << endl;
  }
  return 0;
}
```