Factoring

Advanced Algorithms - Project 2

Johanna Bromark, bromark@kth.se
Jacob Björkman, jacobbj@kth.se
Isabelle Hallman, ihal@kth.se

Final result: 65 on Kattis, ID: 2388459

Table of contents

1. Introduction

The goal of this project was to implement a solution to the problem of factoring integers into prime numbers as given on Kattis (Austrin 2006), a portal for distributing and testing programming exercises used by KTH. This report describes the algorithms and methods used to solve this problem, as well as testing some of its properties.

1.1 Integer Factoring

According to the fundamental theorem of arithmetic, any non-prime positive integer can be uniquely factored into primes. Thus there is only one correct factoring for each such integer.

There are several different algorithms for finding the prime factors of an integer. These include trial division, Fermat's factorization method, Quadratic sieve and more. In this project, we have used the Pollard's Rho algorithm. Further description of this algorithm can be found in section 2.2.

2. Method

To find the factors of a composite positive integer, we use a recursive function which runs the Pollard's Rho algorithm until all prime factors have been found. To handle the large test cases provided by Kattis, we use the GMP library. We test our code on numbers composed by factors of different sizes, as well as on Kattis. The solution was implemented in C++.

2.1 GMP library

GMP is a free library that provides precision arithmetic with no practical length limitation other than the memory of the machine GMP runs on (Free Software Foundation 2016b). Since the test cases on Kattis could be up to 100 bits, and no basic number data type in C++ has that capacity, we have chosen to use this library. The library also provides functions for finding the greatest common divisor as well as to determine whether a number is prime or not with a certain probability. These functions are used in our implementation.

2.2 Pollard's Rho Algorithm

In this project, we have used the Pollard's Rho Algorithm (Pollard 1978). Its expected running time is proportional to the square root of the smallest factor of the integer, making it especially efficient for numbers where one factor is small.

Pollard's Rho Algorithm is, in short, based on generating a pseudo random sequence of numbers mod n and detecting cycles in the sequence (Weisstein 2017). The numbers are generated by the function $g(x) = (x^2 + 1) \bmod n$, which is commonly used for this algorithm (Brent 1980). The factor (and cycle) is detected by checking whether $\gcd(x_i - x_j, n) \neq 1$. If the resulting gcd is not 1, it is a factor to n. In some cases, the resulting gcd can be n, in which case the algorithm fails. If it does, we can try again with a different start value. In our code, we allow for this to happen 10 times.

2.2.1 The theory behind the algorithm

The above works because if $\gcd(x_i - x_j, n) \neq 1$, then $x_i - x_j$ is necessarily some multiple of one of the factors to n. This reduces the problem to finding any multiple of one of the factors of n, rather than the factor directly.

We could try to find such a multiple by randomizing single numbers, but by the birthday paradox, we are more likely to find any number if we take the difference between two numbers rather than just one random number. For example, if we are trying to find the number 2 from the set [1, 10], the probability that we find it directly by randomizing a single number is 1/10. If we, however, take the difference between two

numbers, then any of ((3, 1), (4, 2), (6, 4), (8, 6), (10, 8)) will give us the desired result. We are thus more likely to find a multiple of one of the factors, by choosing many random x mod n and checking whether any difference between two of these numbers is a multiple of a factor of n.

In Pollard's Rho Algorithm, we generate numbers one by one and check if they fulfill $\gcd(x_i - x_{i-1}, n) \neq 1$. As stated above, the numbers are generated by the function g(x) to be pseudo random. Since we choose $x_i = g(x_{i-1})$, the next number is always dependent on the one that comes before, and thus the sequence will cycle at some point. Cycles are related to the factors in that they, too, are identified by checking if gcd(xi - xj, n) ≠ 1. The sequence mod n (N) is related to the sequence mod p (P), where p is a factor of n (Weisstein 2017). If we find a cycle in N and not in any of its factors, gcd(xi - xj, n) will yield n. If we find a cycle in P, this will yield gcd = p. Since p < n, there are fewer possible unique values for P, and thus we are more likely to find a cycle in P before we find a cycle in N. We are therefore likely to find the factor before we start cycling in N (which will make the algorithm fail).

2.2.2 Running time

The above theory also gives us our running time as stated in 2.2: by the reasoning in 2.2.1, we are most likely to find the smallest factor p first. The birthday paradox gives us that the likelihood that this will happen is $O(\sqrt{p})$, which is also our expected running time. Since the smallest factor is at most $\sqrt{n}$, this can also be written as $O(n^{\frac{1}{4}})$.

2.3 Finding the factors

The Pollard's Rho algorithm returns once it has found one factor, which may not be prime. We therefore use a recursive method which works as such:

```
findFactors(input, list of factors)
If input is probably prime:
      add input to list of factors
      return

Run Pollard's rho algorithm and save
      result in variable f
If algorithm failed:
      run again with different start
            value, max 10 times.
If we still don't get any result:
      return 'fail'

If f is probably prime:
      Add f to list of factors
      findFactors(input / f, list of
            factors)
Else:
      findFactors(f, list of factors)
      findFactors(input / f, list of
      factors)
```

2.3.1 Time complexity

If n has k factors, each factor p_k is expected to be found in $O(\sqrt{p_k})$. We should therefore find all the factors in $O(\sum_{i=1}^{k-1} \sqrt{p_i})$, as we won't run the Pollard's Rho Algorithm on the last factor, which will be identified as prime. The last factor is also likely to be the largest.

We also need to check if each factor is prime, which is done by using the GMP library's mpz_probab_prime_p-function, which performs some trial divisions and then runs the Miller-Rabin probabilistic test for primality (Free Software Foundation 2016a). The Miller-Rabin test has a time complexity of $O(m \log^3 n)$ (Dietzfelbinger 2004), where m is the number of values that are tested and n is the number that is tested. The total time complexity for finding all the factors is thus $O((k+x)m \log^3 n + \sum_{i=1}^{k-1} \sqrt{p_i})$, where x is the number of non-prime factors we find. Since k, x and m are constants, this can be simplified to $O(\log^3 n + \sum_{i=1}^{k-1} \sqrt{p_i})$.

2.4 Test cases

We used two strategies for testing our algorithm, described in more detail below.

2.4.1 Testing in Kattis

The test cases provided by Kattis was limited to at most 100 bits but the exact test cases were unknown to us. Kattis gave a score from 0 to 100 depending on how many factors was returned for each input. Kattis also imposed a time limit of 15 seconds if the algorithm was not done in time the test failed. Here we tested our algorithm with different maximal limit for the input until the time was maximized.

2.4.2 Running time for different sized factors

Another test was conducted where the running time of the algorithm was measured for different sized factors. The input was composite numbers of 28 or 29 digits, products of different sets of prime factors (see Appendix I), primes in each set were of the same order of magnitude. The factors were of the order 100, 1000, 10 000, 100 000, 1 000 000, 100 000 000 and arbitrarily chosen. We ran each test ten times and took the average to account for hardware anomalies.

3. Results

When only using the data type “unsigned long long” for storing the number and resulting factors, the result on Kattis was 0, due to the limitation of 64 bits. This was the reason we used GMP library. To solve the problem within the time limit, a limit on the input was needed and here, not unexpectedly, larger limits resulted in a better score. In the end we used a limit of 999467440737095510000000000, which yielded a result of 65 points on Kattis, running for about 14 seconds.

For the second test case we can see the time it took for the algorithm to solve for the factors depending on the size of them. We can see that the time seems to increase exponentially for composite numbers with larger smallest factors (see figure 1).

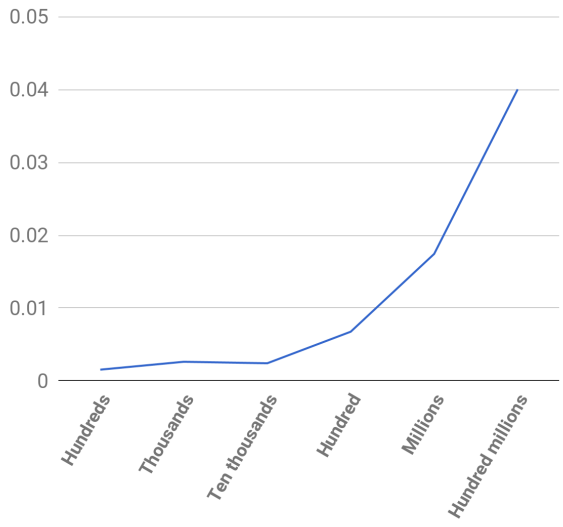

Figure 1: Running time in seconds for factors of different orders

4. Conclusions

As our second test case shows, integers with larger smallest factors takes longer time to factor, even though there were a fewer number of factors. This result is in line with what is expected for Pollard’s Rho algorithm, and illustrates why this algorithm might not be best suited for large factors.

It might be so, that the tests on Kattis uses a lot of composite numbers based only on large factors. If so, a different algorithm such as Quadratic sieve may have been more effective, perhaps yielding a higher score.

5. References

Austrin, P., 2006. Factoring. *KTH CSC Avancerade Algoritmer*. Available at: https://kth.kattis.com/problems/kth.avalg.factoring [Accessed November 20, 2017].

Brent, R.P., 1980. An improved Monte Carlo factorization algorithm. *BIT Numerical Mathematics*, 20(2), pp.176–184. Available at: https://doi.org/10.1007/BF01933190.

Dietzfelbinger, M., 2004. *Primality testing in polynomial time : from randomized algorithms to primes is in P*, Berlin, Germany ; Heidelberg, Germany : Springer.

Free Software Foundation, 2016a. Number Theoretic Functions. Available at: https://gmplib.org/manual/Number-Theoretic-Functions.html [Accessed December 1, 2017].

Free Software Foundation, 2016b. The GNU Multiple Precision Arithmetic Library. Available at: https://gmplib.org/.

Pollard, J.M., 1978. Monte Carlo methods for index computation $({\rm mod}\ p)$. *Mathematics of Computation*, 32(143), pp.918–924.

Weisstein, E.W., 2017. Pollard rho Factorization Method. *MathWorld - A Wolfram Web Resource*. Available at: http://mathworld.wolfram.com/PollardRhoFactorizationMethod.html [Accessed December 1, 2017].

Appendix I

Order 100
101 * 157 * 181 * 37 * 11 * 113 * 89 * 97 * 149 * 107 * 103 * 61 * 41 * 151 * 127 =
897502827512116958778885651 83

Order 1000
7537 * 2753 * 1627 * 2591 * 1453 * 1657 * 1153 * 1451 * 1699 =
59860165592827703318032059824 9

Order 10 000
18787 * 20047 * 18329 * 18313 * 18199 * 18503 * 19793 =
84257077408603175431014934601 3

Order 100 000
607727 * 591289 * 608789 * 610763 * 607819 =
81212357793684754896345979499

Order 1 000 000
9448093 * 10864759 * 11057027 * 11799089 =
13392174635060879013729574561

Order 100 000 000
982451653 * 920419823 * 899809343 =
81366877388467861666184571 7