Integer Factorization

Miska Kananen

December 2, 2018

Contents

1 Introduction

We will define the problem of integer factorization and present some applications of efficient factorization. Then we will look at some basic and some more efficient factorization algorithms, namely Pollard's Rho algorithm and the linear sieve, and implement them in practice.

1.1 Definitions

Definition 1.1. (Integer Factorization problem) Let n be a positive integer. Find an integer a ($1 < a < n$) such that $n = ab$ for some positive integer b or report that such integer does not exist. We call a a *factor* of n and the product ab a *factorization* of n.

If there exists a factorization of n, n is *composite*. Otherwise n is *prime* (or equals 1, in case of which it is neither). Note that it is not required that a is prime.

Example 1.2. (Factorization) Let $n = 36$. We can write $n = 4 \cdot 9$. Here 4 is a factor of 36 and the product $4 \cdot 9$ is a factorization of 36.

We are interested in solving two specific problems: finding a factor of an integer n and finding a factor for all integers in range $\{1, 2, \ldots, n\}$. Let's define these problems more formally.

Problem 1.3. *factor(n)*: find a factor of integer n and return it. If such factor does not exist, report it.

Problem 1.4. *factor-range(n)*: for each integer i in range $\{1, 2, \ldots, n\}$, find a factor of i and return it, or report that such factor does not exist.

In practice we will handle the case where a factor does not exist by returning -1.

Example 1.5. *factor(36)* $= 4$, *factor(5)* $= -1$

Example 1.6. *factor-range(10)* $= [-1, -1, -1, 2, -1, 2, -1, 2, 3, 2]$

1.2 Applications

2 Basic algorithms

In this section we look at some easy, but slightly inefficient algorithms for solving *factor* and *factor-range*. In later chapters we will find out how to solve the problems more efficiently.

2.1 Trial division

The easiest way to solve *factor(n)* is to iterate all integers in range $\{2, 3, \ldots, (n-1)\}$ and try whether one of them divides n. In this case, we have found a factor of n. This kind of algorithm is called *trial division*.

The correctness of the algorithm is easy to see: every possible factor of n is tried, and therefore if one exists, it will be found. The number of divisions done by the algorithm is in the worst case linear to the size of n, and the time complexity of the algorithm is thus $O(n)$. However, we can do better.

Proposition 2.1. *If n is composite, one of its factors is smaller than or equal to $\sqrt{n}$.*

Proof. Since n is composite, it can be written as $n = ab$. Assume that $a > \sqrt{n}$ and $b > \sqrt{n}$. Now $ab > \sqrt{n}^2 \implies ab > n$, which is a contradiction. Therefore either a or b must be $\leq \sqrt{n}$. □

According to Proposition 2.1, it suffices to try out the integers from 2 to $\sqrt{n}$ inclusive. If at this point we have not found a divisor, there are none. Now we do only $\sqrt{n}$ steps in the worst case, and the time complexity of the improved algorithm is $O(\sqrt{n})$. There is still room for improvement.

Proposition 2.2. *All primes p except 2 and 3 are of the form $6k \pm 1$ for some integer k.*

Proof. All integers can be represented as $6k + m$ for some integers k and $0 \leq m < 6$. $6k + 0$ is divisible by 6. $6k + 2$ and $6k + 4$ are divisible by 2. $6k + 3$ is divisible by 3. Thus, all primes must be of the form $6k + 1$ or $6k + 5 \equiv 6k - 1 \mod 6$. □

We can use Proposition 2.2 to reduce the amount of required divisions by only trying 2, 3 and the numbers of the form $6k \pm 1$. Here is our final trial division algorithm:

Algorithm 2.3. (Trial division)

1. Check if 2 or 3 divide n.
2. Iterate all integers of the form $6k \pm 1$ up to and including $\sqrt{n}$ and check if one of them divides n.
3. If no divisor was found in steps 1 and 2, report that there are none.

The time complexity of this algorithm is still $O(\sqrt{n})$, but it has a smaller constant factor than the previous one due to a reduced number of divisions.

2.2 Sieve of Eratosthenes

3 Pollard's Rho algorithm

Pollard's Rho algorithm is a randomized Monte Carlo algorithm that solves *factor(n)* in $O(\sqrt[4]{n})$ average.

3.1 Theory

Assume n is a composite integer and $n = ab$ for some integers a, b ($1 < a, b < n$). Without loss of generality, assume $a \leq b$. What kind of a randomized approach we could use to factorize n?

The first approach could be to randomly generate integers $x_1, x_2, \ldots, x_k$ from range $2 \ldots (n-1)$ and try if one of them divides n. There are $n-2$ integers in the range and if n is a product of two primes, the probability of finding a divisor is $\frac{2}{n-2}$. Therefore it takes roughly n attempts to find a divisor this way.

We can do better. Instead of trying to find an integer x_i that divides n, we can try to find an integer for which $\gcd(x_i, n) > 1$. In this case the greatest common divisor is a non-trivial factor of n. $\gcd(x_i, n) > 1$ for all multiples $a, 2a, \ldots, (b-1)a, b, 2p, \ldots, (a-1)b$ and there are $a + b - 2$ such multiples. The probability of finding such an integer is $\frac{a+b-2}{n} \approx \frac{\sqrt{n}}{n} = \frac{1}{\sqrt{n}}$ in the worst case when all factors are approximately equal, and thus it takes roughly $\sqrt{n}$ attempts to find a divisor.

Now we are on par with the trial division. To improve the algorithm further, we need to explore the concept of *birthday paradox*.

Proposition 3.1. *(Birthday paradox) When uniformly generating random integers $x_1, x_2, \ldots, x_k$ from the range $1 \ldots n$, we need to generate approximately $O(\sqrt{n})$ integers for the probability that two of the generated integers are equal to reach* 0.5.

Proof. Proved in CLRS[1], chapter 5.4.1. □

3.2 Implementation

4 Linear sieve

5 References

References

[1] Thomas H. Cormen; Charles E. Leiserson; Ronald L. Rivest; Clifford Stein. *Introduction to Algorithms, Third Edition*. 2009.