Integer Factorization

Miska Kananen

November 29, 2018

Contents

1 Introduction

We will define the problem of integer factorization and present some applications for efficient factorization. Then we will look at some trivial 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 for a to be 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 most trivial 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 is none. Now we do only $\sqrt{n}$ steps in the worst case, and the time complexity of the improved algorithm is $O(\sqrt{n})$.

2.2 Sieve of Eratosthenes

3 Pollard's Rho algorithm

4 Linear sieve

5 References