Integer Factorization

Miska Kananen

December 15, 2018

Contents

1 Introduction

First 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 or b are 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 or report that such factor does not exist.

In practice we can handle the case where a factor does not exist by, for example, 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

Many present-day cryptographic methods, such as the RSA cryptosystem, depend on the assumption that integer factorization is hard for sufficiently large integers. It's currently unknown whether a non-quantum algorithm that can solve *factor(n)* in polynomial time in relation to the bit length of n exists[6].

If such algorithm was found, RSA would be rendered insecure, since the attacker would be able to derive the private key used for decryption simply by factoring a part of the public key. Currently we can just increase the key size to render factorization infeasible with known algorithms, but if a polynomial-time algorithm existed, the required key sizes to make factorization slow would likely be too large to be practical.

2 Basic algorithms

In this section we look at some easy, but slightly inefficient algorithms for solving *factor* and *factor-range*. In later sections 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 positive 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

Sieve of Eratosthenes is an easy and actually quite efficient algorithm that solves *factor-range(n)*. The algorithm basically works as follows: in the beginning we assume that every integer in range $2 \ldots n$ is prime. Then we iterate through the integers from 2 to $\sqrt{n}$ and for each integer i we mark its multiples up to n composite by marking that i divides them.

The correctness is easy to see: if integer x is composite, it has a factor smaller than or equal to $\sqrt{x}$ according to Proposition 2.1. Since we iterate through the integers in increasing order, we mark x composite before reaching it.

What is the time complexity of the algorithm? We iterate through integers i from 2 to $\sqrt{n}$, and for every i we mark approximately $\frac{n}{i}$ multiples composite, so we are doing approximately $\frac{n}{2}+\frac{n}{3}+\cdots+\frac{n}{\sqrt{n}} = n\cdot(\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{\sqrt{n}})$ operations. The summation is called a *harmonic sum* and is asymptotic to $\log\sqrt{n} \approx \log n$.

Proof. The bound is proved in CLRS, Appendix A.2 [5]. □

Due to the upper bound, the sieve has a time complexity of $O(n \log n)$. However, we can improve this a bit. Instead of marking the multiples of all integers composite, it suffices that we only mark the multiples of primes composite. This works, because the set of multiples of a composite is a subset of the set of multiples of any of its prime factors.

Implemented like this, we are doing approximately $n \cdot (\frac{1}{2}+\frac{1}{3}+\frac{1}{5}+\frac{1}{7}+\ldots)$ (up to the largest prime p for which $\frac{1}{p} \leq \frac{1}{\sqrt{n}}$) operations. The sum is the sum of reciprocals of primes up to n, which is asymptotic to $\log\log\sqrt{n} \approx \log\log n$.

Proof. The bound is proved in [7]. □

By only marking the multiples of primes composite, the sieve has a time complexity of $O(n \log \log n)$.

Algorithm 2.4. (Sieve of Eratosthenes)

1. Zero-initialize a $(n+1)$-element array *fact*.
2. Iterate through integers from 2 to $\sqrt{n}$. For each integer i:
   (a) If $fact(i)$ is non-zero, i is composite. In this case we do nothing and continue iterating from $i+1$.
   (b) Now we know that i is prime. Iterate through integers $j \in \{2i, 3i, \dots\}$ up to n and for each j:
      i. Set $fact(j) = i$.
3. Return the array *fact*.

3 Pollard's Rho algorithm

Pollard's Rho algorithm is a randomized Monte Carlo algorithm that solves *factor(n)* in expected $O(\sqrt[4]{n})$ time and in $O(1)$ space. It was originally invented by John Pollard in 1975[4].

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 could we use to factorize n?

The first approach could be to randomly generate a sequence of integers $x_1, x_2, \dots, x_k$ from range $2 \dots (n-1)$ and try if one of them divides n. There are $n-2$ integers in the range and if n is a product of two distinct primes, the probability of finding a divisor is $\frac{2}{n-2}$, or $\frac{1}{n-2}$ if the primes are equal. 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 $a, 2a, \dots, (b-1)a, b, 2p, \dots, (a-1)b$ 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 randomly generating integers $x_1, x_2, \ldots, x_k$ with a uniform distribution 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, chapter 5.4.1[5]. □

We will utilize the birthday paradox as follows. When randomly generating the integers $x_1, x_2, \ldots, x_k$, we will check for all pairs (x_i, x_j) if $\gcd(x_i - x_j, n) > 1$. How soon can we expect to find a suitable pair?

When generating the sequence x_i, we are also implicitly generating an induced sequence $x'_i = x_i \mod a$. If we manage to generate two integers (x_i, x_j) such that $x'_i \equiv x'_j \mod a$, then a divides $(x'_i - x'_j)$ and it must hold that $\gcd(x_i - x_j, n) > 1$, since $n = ab$. Thus, the question reduces to finding how soon can we expect that $x'_i \equiv x'_j \mod a$.

We assumed that $a \leq b$, so according to Proposition 2.1 $a \leq \sqrt{n}$. So, we are generating random integers x'_i in range $1 \ldots \sqrt{n}$, and according to the birthday paradox we will likely find two equal integers in a list of $O(\sqrt{\sqrt{n}}) = O(\sqrt[4]{n})$ generated integers.

However, we have a problem. To check if there is a repetition in a list of k integers, we have to do approximately k^2 pairwise comparisons. If we implemented the algorithm this way, the time complexity would still be $O(\sqrt{n})$.

Instead of assuming a perfect random number generator, we will use a *pseudorandom number generator*, which is a function that maps each integer to an another seemingly random number, generating a sequence of apparently random integers. For our purposes, the recommended function is a polynomial of the form

$$f(x) = (x^2 + c) \mod n$$

where c is an arbitrary integer constant. The usual choice is $c = 1$. Values $c = 0$ and $c = -2$ are not recommended[4].

Using the function, we can generate the sequence from a starting value x_1 as follows: $x_2 = f(x_1), x_3 = f(x_2)$ and in general, $x_{k+1} = f(x_k)$.

Since we are taking the values mod n, we will find a repeated value at some point. Also, since each value depends only on the previous one, the sequence will enter a cycle after encountering a repeated value. The same properties hold for the induced sequence too, as proven in CLRS, chapter 31.9[5].

The average number of steps before entering a cycle is equal to the number of steps before the first repetition in the sequence, which we showed earlier to be approximately $O(\sqrt{n})$ for the x_i-sequence and $O(\sqrt[4]{n})$ for the x'_i-sequence.

So, if the x'_i-sequence enters a cycle, we have found a factor of n. However, if the x_i-sequence enters a cycle at the same time, the obtained factor is n, which is not useful. In this case, we have to change x_1 or c and restart the algorithm.

How can we detect whether a sequence has entered a cycle without doing pairwise comparisons and storing the earlier values of the sequence? We need *Floyd's algorithm*[2].

Algorithm 3.2. (Floyd's algorithm)

1. Initialize two pointers, $t = f(x_1)$ and $h = f(f(x_1))$.
2. While $t \neq h$, advance t by one step and h by two steps: $t = f(t), h = f(f(h))$.
3. When $t = h$, we are in a cycle.

In Pollard's Rho algorithm we will use Floyd's algorithm on the x_i-sequence in order to detect whether it has entered a cycle. Now we are finally ready to describe the entire Pollard's Rho algorithm.

Algorithm 3.3. (Pollard's Rho algorithm)

The integer to be factored is n and function used to generate the pseudorandom sequence is f.

1. Choose a starting value s from range $1 \ldots (n-1)$ and assign $t = f(s)$, $h = f(f(s))$.
2. While $\gcd(h - t, n) = 1$, repeat $t = f(t)$, $h = f(f(h))$.
3. If at this point $\gcd(h - t, n) = n$, the algorithm failed to find a non-trivial factor. Restart the algorithm and use different s or f.

4. Otherwise we found a non-trivial factor $\gcd(h - t, n)$, return it.

There are some points that are important to notice.

- The factor found by the algorithm is not necessarily prime.
- If n is prime, the algorithm might not terminate. Primality should be tested before using the algorithm.
- If n is a perfect square ($n = m^2$ for some integer m), the algorithm might not terminate.
- Depending on the choice of f, the algorithm might fail to find certain factors: for example the usual choice $f(x) = x^2 + 1$ fails to find the factor 2[1].
- In general, it's important to remember that the algorithm is *randomized*: it is neither guaranteed that it finds a factor nor that it even terminates[5].

3.2 Implementation

A basic implementation in Python 3.

```
1  def pollard_rho(n):
2          # Miller-Rabin, for example, can be used
3          if is_prime(n):
4                  return -1
5
6          # the polynomial x^2 + 1 fails to find factor 2
7          if n%2 == 0:
8                  return 2
9
10         # pay attention to floating-point precision issues
11         # it's recommended to implement these with integers
12         if is_square(n):
13                 return sqrt(n)
14
15         for s in range(1, n):
16                 t = f(s)
17                 h = f(f(s))
18
19                 while gcd(h-t, n) == 1:
20                         t = f(t)
```

```
21                          h = f(f(h))
22
23                  res = gcd(h-t, n)
24                  if res != n:
25                          return res
26
27          # n is composite, but we failed to find a factor
28          return -1
```

4 Linear sieve

The linear sieve is a deterministic algorithm that solves *factor-range(n)* in $O(n)$ time and space. It was originally invented by Gries and Misra in 1978[3].

4.1 Theory

What makes the sieve of Eratosthenes slightly inefficient is that many composite numbers are marked as such multiple times. If we can find a way to mark every composite number only once, we can get the time complexity down to $O(n)$.

The main idea is that if we can find a unique representation for every composite integer, then we can generate every such representation only once.

Theorem 4.1. *Every composite integer a has a unique representation $a = px$, where p is the smallest prime factor of a and x is an integer.*

Proof. Proved in Gries' and Misra's paper[3]. □

Corollary 4.2. *$x \geq p$ and x does not have prime factors that are smaller than p.*

Proof. If $x < p$, x is either prime or it must have a prime factor. In both cases we obtain a prime that is smaller than p. However, $a = px$ and we assumed that p is the smallest prime factor of a, so we have a contradiction.

Also, if $x \geq p$ and x has a prime factor smaller than p, we get the same contradiction again. □

We will denote the smallest prime factor of an integer i as $sp(i)$. The linear sieve works as follows: instead of iterating all the integer multiples of i up to the limit n, we will only iterate the representations pi: $2i$, $3i$, $5i$, ...,

$p_k i$, where p_k is the largest prime that is smaller or equal to $sp(i)$. We can't go any higher or we would violate Corollary 4.2 and thus the uniqueness of the representation.

This way we will find a factor for every composite number, and every composite number is processed exactly once.

Proof. We have to prove that we will find a factor exactly once for every composite a.

First we will prove that we will find a factor for a. Since a is composite, it can be uniquely written as $a = px$ according to Theorem 4.1. We process the integers in ascending order, and since $x < a$, we have processed x before processing a. While processing x, we marked all multiples $2x, 3x, \ldots, sp(x) \cdot x$ as composite, and px must have been one of these multiples, since according to Corollary 4.2 $p \leq sp(x)$.

We still have to prove that we will find a factor for a only once. The representation $a = px$ is unique, so there is exactly one x from which we can mark a composite, and when processing x, we mark every multiple only once. □

Algorithm 4.3. (Linear sieve)

1. Zero-initialize a $(n + 1)$-element array sp and create an empty list *primes*.
2. Set $sp(1) = 1$.
3. Iterate all integers i from 2 to n. For each integer i:
   (a) If $sp(i)$ is still 0, i is prime. Add i to *primes* and set $sp(i) = i$.
   (b) Iterate all integers p in *primes* while $p \leq sp(i)$ and $p \cdot i \leq n$. For each integer p:
      i. Set $sp(p \cdot i) = i$.
4. Return the array sp.

4.2 Implementation

Implementation in Python 3. In this implementation i is prime, if in the returned array $arr(i) = i$ instead of -1.

```
def linear_sieve(n):
        sp = (n+1)*[0]
        primes = []

        sp[1] = 1
        for i in range(2, n+1):
                if sp[i] == 0:
                        sp[i] = i
                        primes.append(i)

                for p in primes:
                        a = p*i
                        if p > sp[i] or a > n:
                                break
                        sp[a] = i

        return sp
```

References

[1] *Bad numbers for Pollard-Rho algorithm*. URL: https://math.stackexchange.com/questions/2855796/bad-numbers-for-pollard-rho-algorithm. (accessed: 02.12.2018).

[2] Antti Laaksonen. *Competitive Programmer's Handbook*. 2017.

[3] David Gries; Jayadev Misra. "A Linear Sieve Algorithm for Finding Prime Numbers". In: *Communications of the ACM* 21.12 (1978), pp. 999–1003.

[4] J. M. Pollard. "A Monte Carlo method for factorization". In: *BIT Numerical Mathematics* 15.3 (1975), pp. 331–334.

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

[6] William Stein. *Elementary Number Theory: Primes, Congruences, and Secrets*. 2011.

[7] *What is the sum of the reciprocal of primes?* URL: https://math.stackexchange.com/questions/674877/what-is-the-sum-of-the-reciprocal-of-primes-yes-it-diverges. (accessed: 13.12.2018).