

# Factoring integers

6.1600 Course Staff

Fall 2024

The problem of integer factorization was central to 20th-century cryptography. Breaking the one-wayness of the RSA trapdoor one-way function (??), for example, is no harder than factoring integers. In this chapter, we will see a couple of surprisingly powerful algorithms for factoring integers.

We will only consider factoring numbers of the form  $N = pq$ , for distinct odd primes  $p$  and  $q$ . (The general case is not too much more challenging.) Throughout, let  $n = \lceil \log_2 N \rceil$  be the bitlength of the number to factor.

**Disclaimer:** This set of notes is a work in progress. It may have errors and be missing citations. It is certainly incomplete. Please let the staff know of any errors that you find.

## 1 Background

*Trial division.* We can factor  $N$  by trying to divide  $N$  by each of the primes of size  $\leq \sqrt{N}$  and checking whether the result is an integer. If so, we have found a factor of  $N$ . Since at least one of the two factors of  $N$  is in  $\{1, \dots, \sqrt{N}\}$ , this algorithm (“trial division”) runs in time roughly  $\sqrt{N} = 2^{n/2}$ .

Trial division is an *exponential time* algorithm, since it runs in time  $2^{\Omega(n)}$ , where the bitlength  $n$  is the size of the number to be factored. The best known factoring algorithms run in *sub-exponential time*  $2^{O(n^c)}$ , for some constant  $c < 1$ .

*Euclid’s algorithm.* An important subroutine in almost all factoring algorithms is Euclid’s polynomial-time algorithm for computing the greatest common divisor of two integers  $x$  and  $y$ .

The principle of Euclid’s algorithm is that

$$\gcd(x, y) = \gcd(x, y \bmod x) \quad \text{and} \quad \gcd(x, 0) = x.$$

So, for example, if we want to compute  $\gcd(46, 12)$ , we can compute it as:

$$\gcd(46, 12) = \gcd(12, 10) = \gcd(10, 2) = 2.$$

*Difference of squares.* The second key idea is that, if we can find two numbers  $x, y \in \mathbb{Z}$  whose squares are congruent modulo  $N$ , we can

In this discussion, we draw on Arjen Lenstra’s very nice survey on factoring Arjen K Lenstra. “Integer factoring”. In: *Designs Codes, and Cryptography* 19.2/3 (2000).

use these numbers to factor  $N$ :

$$\begin{aligned} x^2 &= y^2 & (\text{mod } N) \\ x^2 - y^2 &= 0 & (\text{mod } N) \\ (x + y)(x - y) &= 0 & (\text{mod } N) \end{aligned}$$

If  $x = \pm y$ , then this relation is not helpful to us. But if  $x \neq \pm y$ , then we know that  $x + y \not\equiv 0 \pmod{N}$  and  $x - y \not\equiv 0 \pmod{N}$ . So we have:

$$(x + y)(x - y) = kN \quad \in \mathbb{Z},$$

for some positive integer  $k \in \mathbb{Z}$ . Then  $x + y$  must be a multiple of one of the factors of  $N$  (but not both), and  $\gcd(x + y, N)$  reveals a factor of  $N$ .

The goal of many factoring algorithms—including the one we will see today—is finding these integers  $x$  and  $y$  whose squares are congruent modulo  $N$ .

It is not necessarily obvious that the useful pairs  $(x, y)$  will ever exist. The key idea is that, modulo  $N = pq$ , every number in  $\mathbb{Z}_N^*$  either has four square roots or has none. If an element in  $\mathbb{Z}_N^*$  has four square roots then the roots are of the form  $r, -r, s, -s$ . In this case, a pair  $(\pm r, \pm s)$  yields the sort of relation that we need to factor.

## 2 Dixon's algorithm

Dixon's algorithm is one of the simplest sub-exponential-time factoring algorithms. It gives a fast method for finding two numbers whose squares are congruent modulo  $N$ . Once we have these squares, we can use them to factor as in Section 1

### 2.1 The idea

The principle of Dixon's algorithm is that we will pick many random numbers  $r \in \mathbb{Z}_N^*$  and square them modulo the integer  $N$  we would like to factor.

Say that we are somehow able to find numbers  $r, r'$  such that

$$\begin{aligned} r^2 &= 2 \cdot 3^2 \cdot 5 & (\text{mod } N) \\ r'^2 &= 2 \cdot 5 & (\text{mod } N), \end{aligned}$$

then we know that:

$$\begin{aligned} (rr')^2 &= 2^2 \cdot 3^2 \cdot 5^2 & (\text{mod } N) \\ (rr')^2 &= (2 \cdot 3 \cdot 5)^2 & (\text{mod } N) \end{aligned}$$

and now we have two numbers whose squares are congruent modulo  $N$ :

$$x = rr' \quad \text{and} \quad y = 2 \cdot 3 \cdot 5.$$

If we are lucky, this is the useful type of congruence that we can use to factor  $N$  (i.e.,  $rr' \not\equiv \pm 2 \cdot 3 \cdot 5 \pmod{N}$ ).

The principle of Dixon's algorithm is to generate many such  $rs$  and then use linear algebra to find a subset of them whose product modulo  $N$  is a perfect square.

## 2.2 The algorithm

*Input:* An integer  $N = pq$  for odd primes  $p$  and  $q$ . A parameter  $B \in \mathbb{N}$ , which we refer to as "the size of the factor base."

*Output:* The factors  $(p, q)$  of  $N$ .

1. **Collect linear relations.** Maintain a list  $L$  of pairs of (a) an element in  $\mathbb{Z}_N^*$  and (b) vectors over  $\mathbb{Z}_2^B$ . Repeat until  $L$  contains  $B + 1$  pairs:
  - Sample  $r \xleftarrow{\mathbb{R}} \mathbb{Z}_N^*$ .
  - Compute  $s \leftarrow (r^2 \bmod N)$ .
  - Attempt to write  $s$  as a product of the first  $B$  primes:

$$s = 2^{e_2} 3^{e_3} 5^{e_5} \dots$$

- If successful, add the pair  $(r, (e_2, e_3, e_5, \dots))$  to the list  $L$ .
2. **Solve linear system.** Let  $L = \{(r_1, \mathbf{v}_1), (r_2, \mathbf{v}_2), \dots\}$ . Find a non-zero combination of the vectors in  $L$  that sums to zero modulo 2. That is, find  $S \subseteq [B + 1]$  such that

$$\sum_{i \in S} \mathbf{v}_i = (0, 0, 0, \dots, 0) \in \mathbb{Z}_2^B.$$

Letting

$$(e_2, e_3, e_5, \dots) \leftarrow \sum_{i \in S} \mathbf{v}_i,$$

we then have a difference of squares:

$$\left( \sum_{i \in S} r_i \right)^2 = (2^{(\frac{e_2}{2})} \cdot 3^{(\frac{e_3}{3})} \cdot 5^{(\frac{e_5}{5})} \dots)^2 \pmod{N}.$$

3. **Use Euclid's algorithm to try to factor  $N$ .** We can take:

$$x = \left( \sum_{i \in S} r_i \right)$$

$$y = 2^{(\frac{e_2}{2})} \cdot 3^{(\frac{e_3}{3})} \cdot 5^{(\frac{e_5}{5})} \dots$$

and compute  $\gcd(x + y, N)$ . With probability roughly  $1/2$ , over the random choice of the  $rs$ , this will yield a factor of  $N$ .

If a number completely splits into prime factors  $\leq B$ , we say that the number is " $B$ -smooth."

We can find the set  $S$  using Gaussian elimination in roughly  $B^3$  time. Since the set of vectors will be extremely sparse, there are faster methods that implementers use in practice.

### 2.3 The analysis.

The costs of the three steps of the algorithms are:

1. Each iteration of the loop requires us to try to factor a number into primes  $\leq B$ . We can factor in this way by trial division using time roughly  $B$ .

The question then is how many trials it will take for us to find a single smooth number. For a smoothness bound  $B$ , let's say for now that it takes  $T(B)$  trials—we will look into the precise value of  $T(B)$  in a moment..

2. Solving the linear system using Gaussian elimination takes roughly  $B^3$  time.
3. Run Euclid's algorithm—the time required here is negligible compared to the time of the first two steps. This step runs in time  $\text{poly}(\log N) = \text{poly}(n)$ .

Putting everything together, we have that factoring an  $n$ -bit number with a factor base of size  $B$  takes time:

$$B \cdot T(B) + B^3 + \text{poly}(n). \quad (1)$$

*Smoothness probabilities.* The key question that we need to answer to complete the analysis is

"If we pick an integer uniformly at random from  $\{1, \dots, N\}$ , what is the probability that the integer will be  $B$ -smooth?"

The convention is to denote the number of  $B$ -smooth numbers in  $\{1, \dots, N\}$  as  $\Psi(N, B)$ . When  $B$  is "not too small," we have:

$$\Psi(N, B) \approx N \cdot u^{-u+o(1)} \quad \text{for } u = \frac{\log N}{\log B}.$$

The probability of a random number modulo  $N$  being  $B$ -smooth is then  $\Psi(N, B)/N$  and the expected number of tries it will take for us to find a smooth number is:

$$T(B) = 1/\Psi(N, B).$$

Now we can plug this estimate into the expression (1) for the running time of Dixon's algorithm and we can solve for the value of  $B$  that minimizes the running time. In particular, to minimize the running time we want:

$$B \approx T(B) \approx N/\Psi(N, B) = u^u,$$

I'm ignoring any  $\log B$  factors, which do matter very much in practice.

For many more details on these estimates, take a look at Granville's very nice survey on smooth numbers. Andrew Granville. "Smooth numbers: computational number theory and beyond". In: *Algorithmic number theory: lattices, number fields, curves and cryptography* 44 (2008), pp. 267–323

We are being slightly imprecise—we actually need to know the number of squares (quadratic residues) modulo  $N$  that are smooth. But heuristically, we can assume that quadratic residues behave like random integers modulo  $N$  for the purposes of smoothness.

for  $u = (\log N)/(\log B)$ .

$$\begin{aligned}
 B &= u^u \\
 \log B &= u \log u \\
 \log B &= \frac{\log N}{\log B} \log \frac{\log N}{\log B} \\
 \log^2 B &\approx \log N \log \log N \\
 \log B &\approx \sqrt{\log N \log \log N} \\
 B &\approx \exp(\sqrt{\log N \log \log N}).
 \end{aligned}$$

If we plug this value of  $B$  into Dixon's algorithm, we get a running time of

$$\exp(O(\sqrt{\log N \log \log N})) = 2^{O(\sqrt{n \log n})}.$$

### References

- Granville, Andrew. "Smooth numbers: computational number theory and beyond". In: *Algorithmic number theory: lattices, number fields, curves and cryptography* 44 (2008), pp. 267–323.
- Lenstra, Arjen K. "Integer factoring". In: *Designs Codes, and Cryptography* 19.2/3 (2000).