CSE291-14: The Number Field Sieve

https://cseweb.ucsd.edu/classes/wi22/cse291-14

Emmanuel Thomé

January 4, 2022

Part 1a

Introduction

Problem statement and motivations

Hardness

Algorithms: theory and practice

Course trivia

Plan

Problem statement and motivations

Hardness

Algorithms: theory and practice

Course trivia

The challenge

Integer factorization problem (IF)

Let $N = pq$ be a product of 2 primes with $\log p \approx \log q$. Factor N.
The hardness of IF increases with the size of N.

Discrete logarithm problem (DLP) in a group G

G is assumed to be cyclic. Given $a = g^x$ in G, find x.
The DLP hardness depends on the group G.

(If not absolutely comfortable with DLP, we'll talk about it in greater detail anyway.)

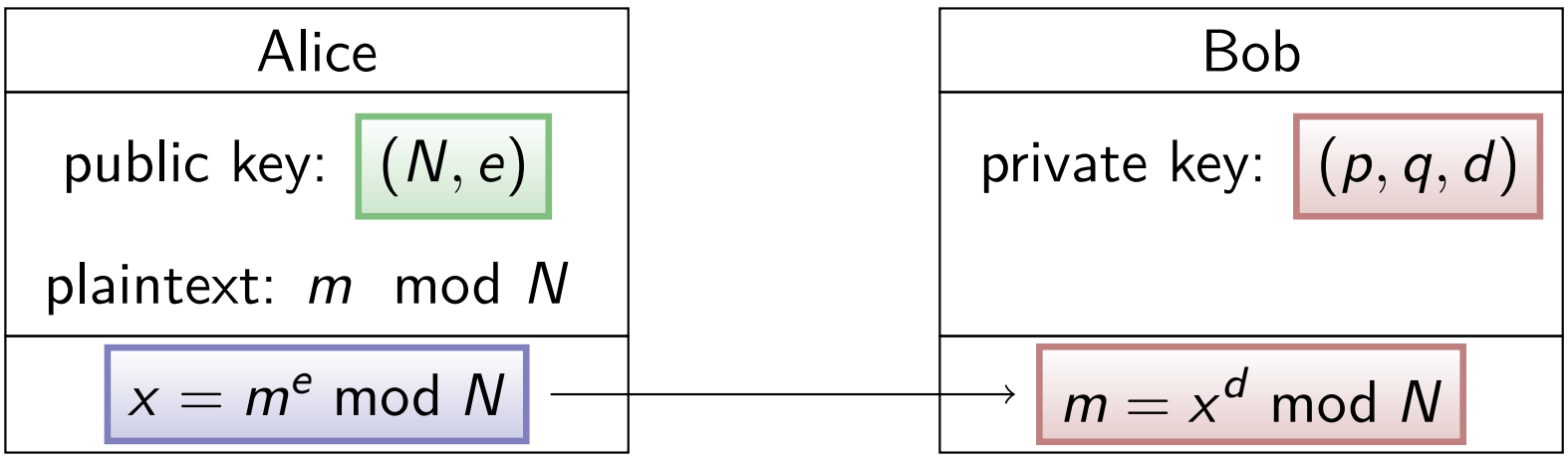
Why do we want to look at these problems?

Several motivating arguments.

- Cryptography and cryptanalysis.
  - IF is the hard problem that RSA relies on.
  - DLP is the hard problem that Diffie-Hellman relies on.
- Computational mathematics.
  - Many number-theoretic calculations use IF as a subroutine.
- Recreation.

RSA encryption

- N integer, and p, q (prime) such that $N = pq$.
- $\phi(N) = (p-1)(q-1) = \#(\mathbb{Z}/N\mathbb{Z})^*$.
- e coprime to $\phi(N)$, therefore invertible modulo $\phi(N)$.
- d computed such that $de \equiv 1 \mod \phi(N)$. $\rightarrow$ the trapdoor!



$$x^d \equiv m^{ed} \equiv m^{1+k\phi(N)} \equiv m \mod N.$$

RSA assumption

Bob keeps p, q secret, and publishes N.

$\Rightarrow$ Bob knows p, q, and can compute $\phi(N) = (p-1)(q-1)$.

$\Rightarrow$ He can therefore compute d.

$\Rightarrow$ and recover the m from the ciphertext $x = m^e \mod N$.

If an attacker can factor N, they can do the same.

Note: we have implications above.

The RSA assumption

The RSA assumption is: it is hard to recover m from x.

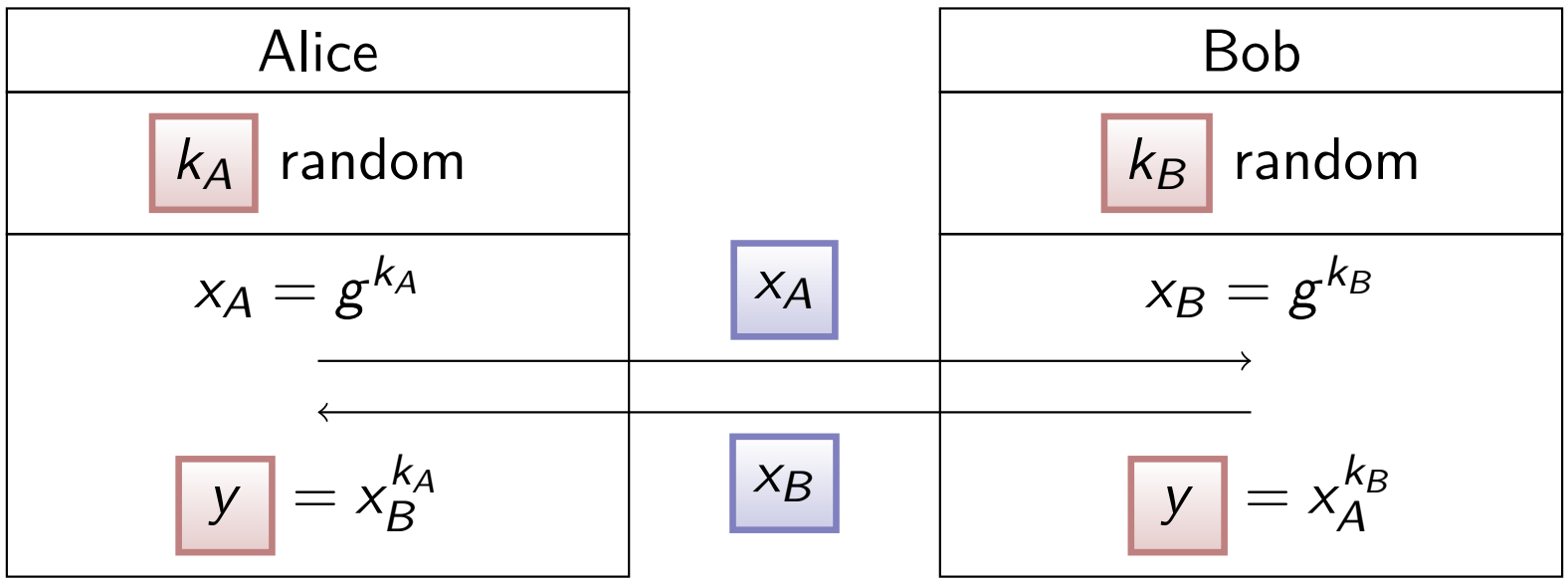
Of course it gets harder as N grows.

No one knows if hardness of IF and the RSA assumption are equivalent. (The common wisdom is that they are probably not.)

Diffie-Hellman

Very important primitive: key exchange.

Public data: a finite cyclic group $G = \langle g \rangle$.



Alice and Bob can then use y as a symmetric encryption key for their communication (e.g. with AES).

DH assumption

The goal of the attacker is to do something hard that neither Alice nor Bob has to do:

Knowing $x_A = g^{k_A}$ and $x_B = g^{k_B}$, find $y = g^{k_A k_B}$.

As before, if an attacker can solve DLP, he can do this.

The DH assumption

The DH assumption is that this is hard to find $g^{k_A k_B}$.
Of course it depends on the group G.

No one knows if DH and hardness of DLP are equivalent.
There are results suggesting that they might be.

Mathematical motivation

In some mathematical contexts, the prime factors of some integers hide some interesting structure.

Example: in a number field K, the only way to be sure that we correctly compute the ring of integers $\mathcal{O}_K$ is by factoring the discriminant of the defining polynomial of K.

Recreational aspect

Because IF is so easy to state, it has been looked at for a long time, and also attracts many enthusiasts.

- The Cunningham project has been running for decades.
  - Goal: collect factorization of numbers of the form $b^n \pm 1$.
  - Mathematical motivation does exist, but is quite dim.
- There is a sizable community of factoring enthusiasts (and big prime hunters).

A constructive cryptographical motivation

Discrete logarithm is a way to make a map explicit.

- Mathematician viewpoint: a cyclic group G is like $\mathbb{Z}/N\mathbb{Z}$. That's not interesting.

$$G \cong \mathbb{Z}/N\mathbb{Z}.$$

- However, this equivalence is precisely what DLP is all about!

Example of CSIDH-512:

- efficient uniform random sampling in a cyclic group via discrete logarithm computations.
- a DLP computation has helped make a cryptosystem proposal more efficient.

Plan

Problem statement and motivations

Hardness

Algorithms: theory and practice

Course trivia

Case-sensitiveness

A word of caution

As we talk about integer factorization, there's often an integer N that is floating around.
The bit size of the input N is of course $n = \log_2 N$.

Qualitatively, all complexities are understood as functions of n: an exponential algorithm is one that is exponential in n.

- A $O(\sqrt{N})$ algorithm is exponential in n.
- A $O(N^{0.001})$ algorithm is also exponential.
- A $O(\exp(\sqrt{n}))$ algorithm is sub-exponential.

Upper bounds on asymptotic complexity

Complexity (spoiler alert)

The Number Field Sieve (this class!) is the algorithm which solves these problems with the best asymptotic complexity:

- For an n-bit input, typically: $\exp\left(cn^{1/3}(\log n)^{2/3} \cdot (1 + o(1))\right)$ with an explicit constant c.
- This is sub-exponential.
- $(1 + o(1))$ hides a lot.
- What this means in concrete terms is hard to say.

We'll discuss the complexity in more detail as we review the different algorithms later on.

Hardness: converse problems

Note that of course, the converse problems are easy.

This situation is good for cryptography, of course.

The converse of DLP is ...

- ...
- efficiently done with ...
- which costs ... group operations for an n-bit input.

The converse of IF is ...

- ...
- efficiently done with ...
  - in practice: ...
  - in theory: ...

Complexity-wise, IF and DLP are in NP, but that doesn't say much.

Hardness: converse problems

Note that of course, the converse problems are easy.

This situation is good for cryptography, of course.

The converse of DLP is ...

- exponentiation in a group,
- efficiently done with ...
- which costs ... group operations for an n-bit input.

The converse of IF is ...

- ...
- efficiently done with ...
  - in practice: ...
  - in theory: ...

Complexity-wise, IF and DLP are in NP, but that doesn't say much.

Hardness: converse problems

Note that of course, the converse problems are easy.

This situation is good for cryptography, of course.

The converse of DLP is ...

- exponentiation in a group,
- efficiently done with square and multiply,
- which costs ... group operations for an n-bit input.

The converse of IF is ...

- ...
- efficiently done with ...
  - in practice: ...
  - in theory: ...

Complexity-wise, IF and DLP are in NP, but that doesn't say much.

Hardness: converse problems

Note that of course, the converse problems are easy.

This situation is good for cryptography, of course.

The converse of DLP is ...

- exponentiation in a group,
- efficiently done with square and multiply,
- which costs $O(n)$ group operations for an n-bit input.

The converse of IF is ...

- ...
- efficiently done with ...
  - in practice: ...
  - in theory: ...

Complexity-wise, IF and DLP are in NP, but that doesn't say much.

Hardness: converse problems

Note that of course, the converse problems are easy.

This situation is good for cryptography, of course.

The converse of DLP is ...

- exponentiation in a group,
- efficiently done with square and multiply,
- which costs $O(n)$ group operations for an n-bit input.

The converse of IF is ...

- integer multiplication,
- efficiently done with ...
  - in practice: ...
  - in theory: ...

Complexity-wise, IF and DLP are in NP, but that doesn't say much.

Hardness: converse problems

Note that of course, the converse problems are easy.

This situation is good for cryptography, of course.

The converse of DLP is ...

- exponentiation in a group,
- efficiently done with square and multiply,
- which costs $O(n)$ group operations for an n-bit input.

The converse of IF is ...

- integer multiplication,
- efficiently done with FFT-based methods.
  - in practice: ...
  - in theory: ...

Complexity-wise, IF and DLP are in NP, but that doesn't say much.

Hardness: converse problems

Note that of course, the converse problems are easy.

This situation is good for cryptography, of course.

The converse of DLP is ...

- exponentiation in a group,
- efficiently done with square and multiply,
- which costs $O(n)$ group operations for an n-bit input.

The converse of IF is ...

- integer multiplication,
- efficiently done with FFT-based methods.
  - in practice: Schönhage-Strassen (among others);
  - in theory: ...

Complexity-wise, IF and DLP are in NP, but that doesn't say much.

Hardness: converse problems

Note that of course, the converse problems are easy.

This situation is good for cryptography, of course.

The converse of DLP is ...

- exponentiation in a group,
- efficiently done with square and multiply,
- which costs $O(n)$ group operations for an n-bit input.

The converse of IF is ...

- integer multiplication,
- efficiently done with FFT-based methods.
  - in practice: Schönhage-Strassen (among others);
  - in theory: Harvey-van der Hoeven, $O(n \log n)$.

Complexity-wise, IF and DLP are in NP, but that doesn't say much.

Lower bounds on complexity

I am not aware of any non-trivial lower bound on:

- IF
- or any concrete instance of DLP.

Only result in the black box model

The only result is Nechaev-Shoup's: if you know nothing about a group G, then DLP in G costs $\Omega(\sqrt{\#G})$.

it is also $O(\sqrt{\#G})$, so it's $\Theta()$.

Heuristics and Probabilistic algorithms

This slide is some sort of a disclaimer.

As far as the cryptanalysis motivation goes, we ultimately don't mind if we achieve a security breach "heuristically", or "probabilistically". If it's broken, it's broken.

NFS does many wild heuristics, and is also a probabilistic algorithm. The GRH is only one of the many heuristics. This won't disturb us.

The search for proven algorithms for factoring or discrete logarithms is a very different topic. At this point, proven deterministic algorithms have exponential complexity.

Plan

Problem statement and motivations

Hardness

Algorithms: theory and practice

Course trivia

A zoo of algorithms

Part of this course will briefly review the pre-NFS algorithms, in particular because NFS did build on something!

It is also interesting from a historical perspective.

Some recursion is going on: NFS uses factorizations of moderate-size numbers as a subroutine.

Less advanced algorithms are better for these moderate sizes, and thus can be used within NFS!

What makes NFS stand out

Most of the pre-NFS algorithms needed mostly arithmetic with:

- finite fields $\mathbb{F}_p$,
- polynomials over $\mathbb{Z}$ or over $\mathbb{F}_p$,
- linear algebra.

In a way, it's the same with NFS. However NFS is inscribed in a broader mathematical context which can be frightening:

- number fields and algebraic number theory,
- a few other things touched upon in passing.

The cryptanalytic motivation has fueled the constant interest of people over several years/decades. NFS folklore is rich of multiple things.

Computational aspect

This course intends to present the NFS algorithm AND address its more practical aspects.

- What does it take to program NFS?
- What are the existing implementations and their shortcomings?
- How hard are the different steps of the algorithm?
- What implementation choices are important?

Required background

NFS touches upon many aspects, but very often on a fairly elementary level.

- Algebraic number theory.
  But one does not need to master ANT to understand NFS!
- Computer algebra and computer arithmetic.
  Mostly finite fields, polynomials.
- Complexity and analysis of algorithms.
- Analytic number theory.
  Prime number theorem is mostly enough.
- Basic principles of computer architecture.
- Programming.

Relevant literature

The course web page lists several resources, books, and articles. The immense majority is available online.

A few textbooks have spot-on chapters that deal with NFS, namely:

- V. Shoup, A Computational Introduction to Number Theory and Algebra.
- S. Galbraith, Mathematics of Public Key Cryptography.

Both are available online (links on course web page).

Early history and high-level view of NFS are best documented in

- A.K. Lenstra, H.W. Lenstra, Jr, The Development of the Number Field Sieve.

Plan

Problem statement and motivations

Hardness

Algorithms: theory and practice

Course trivia

About me

PhD in France some decades ago. Spent a year in Chicago as a visitor while I was beginning my PhD (previous millenium).

Full-time researcher at Inria since 2003, where I've been leading a research group since 2015. (group = 8 permanent full-time researchers + 2 permanent faculty. Small number of students).

Contributed several high-level things about NFS-like algorithms. Also contributed several low-level things and lots of code. Main author of Cado-NFS. Participated in many computational records.

Fulbright grant. Visiting professor in the CSE department this year. Enjoying San Diego!

Participation in records

I participated in most large NFS computations since 2010.

- 2010: Factorization of RSA-768
- 2014: Discrete logarithms in $\mathbb{F}_{2^{809}}$.
- 2015: Logjam attack, real-time 512-bit discrete logarithm computations and MitM exploit.
- 2016: Discrete logarithms modulo "special" 1024-bit p.
- 2019: Discrete logarithms modulo 795-bit p.
- 2019: Factorization of 795-bit RSA-240.
- 2020: Factorization of 829-bit RSA-250.

The Cado-NFS software

Cado-NFS is a full implementation of the Number Field Sieve, started in 2007. Cado-NFS is

- standalone,
- LGPL-licensed,
- open source and open development (open git since 2009),

  https://cado-nfs.inria.fr/
- mostly C and C++, hundreds of kLoc.

Contributors: 3 (core) + random variable in $[0, 5]$.

Cado-NFS was used in all my recent record computations.

Course organization

I'll try to put in perspective the theory aspects and the practical ones.

By "practical", I mean:

- Things that are definitely important in an implementation
- Things that are potentially useful in an implementation
- Considerations of how a typical implementation performs.

The main computer illustration platform will be Cado-NFS. It's probably wise to install it right away, if not done already.

Cado-NFS runs on linux.

pick the git master branch, please. It does break every now and then, please report!

Grading

Grading is primarily based on course participation.

If you want to maximize credit for this course, I expect that you turn in some work.

- No formal homework assignments, but some open questions or suggestions for independent investigation.
- Or you can pick a claim that appears in the courses, and work to document how true (or wrong!) it is. This can (but does not have to) entail experimental work.

CSE291-14: The Number Field Sieve

https://cseweb.ucsd.edu/classes/wi22/cse291-14

Emmanuel Thomé

January 4, 2022

Part 1b

Prime numbers and basic sieving

Primes and the prime counting function

Compositeness testing

Finding primes with sieving

Goals for today

Factoring is about splitting into primes.

Today: focus on primes.

Some techniques that already apparent in this context are also relevant for NFS.

Smooth numbers

Definition: smooth numbers

An integer N is B-smooth if it has all its prime factors are $\leq B$.
Example: $1152 = 2^7 \times 3^2$ is 3-smooth.

In NFS, most of the time ($> 50\%$) is spent in testing if certain numbers are smooth. This is very close to factoring.

Plan

Primes and the prime counting function

Compositeness testing

Finding primes with sieving

Primes !

There are infinitely many primes. Euclide knew that.

A much harder result:

Theorem (Prime number theorem, 1896)

$\pi(x) = \#\{p \text{ prime}, p < x\} \sim \dfrac{x}{\log x}$.

$\pi(x)$ *is known as the prime counting function.*

$\pi(x)$ is however not explicitly known.

Handwavy statement: a random integer around x has a probability $1/\log x$ of being a prime.

Handwavy statements like this are ok, but if anything more precise is wanted, things become hard pretty quickly.

$\pi(x)$ and $\mathrm{li}(x)$

Better approximations of $\pi(x)$ are known. The generalized Riemann hypothesis (GRH) implies (and is equivalent to):

$$\pi(x) - \mathrm{li}(x) = O(\sqrt{x}\log x)$$

where $\mathrm{li}(x)$ is the logarithmic integral function $\int_0^x \mathrm{d}t/\log t$.

In practice, we're ok with assuming GRH, and $\mathrm{li}(x)$ is quite good: relative error $< 10^{-5}$ for $x \approx 2^{32}$.

Note that $\mathrm{li}(x)$ is (almost) among the standard math functions in C++17:

```
double nprimes_interval(double p0, double p1)
{
    using namespace std;
    return expint(log(p1)) - expint(log(p0));
}
```

Dividing a prime range into pieces

We commonly have tasks to do like:

For all primes in $[a, b]$, do something.

In a multi-core, very parallel world, we want to divide this into pieces with roughly equal work. This is made easy with access to a decent approximation of $\pi(x)$.

The isPrime problem

Algorithmic problem: isPrime. Tell whether $x \in \mathbb{N}$ is prime or not.

Proved to be solvable in deterministic polynomial time only in 2002 (Agrawal, Kayal and Saxena).

Here: not only we don't care about deterministic, but we don't need a certified answer either.

Consequence: Stick to last century algorithms for compositeness testing.

Plan

Primes and the prime counting function

Compositeness testing

Finding primes with sieving

Compositeness testing

General idea

Start from a statement that says:

If N is prime then for all $x \mod N$, something(x) holds.

If we find an x such that something(x) doesn't hold, we have a proof that N is composite.

Testing something(x) is generally quick.

It is hopeless to try to prove primality with such statements.

People try to give bounds on the number of false negatives among witnesses x.

Fermat

If N is prime then it verifies Fermat's little Theorem:
for all x coprime to N, $x^{N-1} \equiv 1 \bmod N$

False negatives: $2^{340} \equiv 1 \bmod 341$.
Def. We say that 341 is a Fermat pseudoprime in base 2.

Even worse, this can happen for any base:

Def. A Carmichael number is a composite that is a Fermat pseudoprime in any base.

First examples: 561, 1105, 1729, 2465, 2821, 6601, 8911, 10585.

See A002997.

Thm. (Alford, Granville, Pomerance, 1994; later: Harman, 2008)
There are at least $n^{1/3}$ Carmichael numbers between 1 and n.
(the true count might be as large as $n^{1-o(1)}$).

Solovay–Strassen

If N is prime then $\left(\frac{x}{N}\right) = x^{(N-1)/2} \bmod N$.
note: The Jacobi symbol $\left(\frac{x}{N}\right)$ can be computed with the reciprocity law, which is close to the Euclidean algorithm.

Bound on false negatives: For a fixed composite N, the probability that N is a pseudoprime in base x is less than $1/2$.

Rem. This is very pessimistic.

Miller-Rabin

Idea. Refine previous test, using properties of the 2-Sylow subgroup of the multiplicative group of $\mathbb{Z}/N\mathbb{Z}$.
Assume that N is an odd prime; $N-1 = 2^s d$, where d is odd.
For any x coprime to N, one of the following holds:

- $x^d \equiv 1 \mod N$;
- $x^{d.2^r} \equiv -1 \mod N$, for an $0 \leq r \leq s-1$.

Proof. If N is an odd prime, there are 2 square roots of 1 modulo N, namely -1 and 1. Otherwise, there are more.

Thm. For any fixed composite N, the probability that N is a strong pseudoprime in base x is less than $1/4$.

Rem. Again, very pessimistic. Testing with $x = 2$ and $x = 3$ proves primality of N's up to $1,373,653$.

isPrime works

The bottom line is that testing a number for probable primality can be done:

- in a way that is satisfactory for "most purposes" (at the very least for all our potential uses within NFS),
- and in polynomial time, with minimal storage overhead.

There are several more advanced algorithms in this realm, some of which actually prove primality and give certificates. It is not useful to explore these in our usage context.

Plan

Primes and the prime counting function

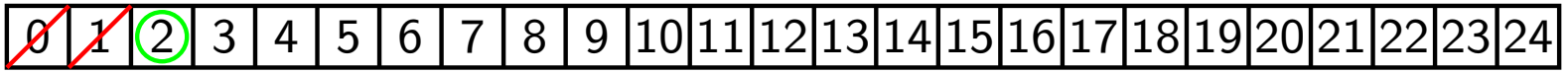
Compositeness testing

Finding primes with sieving

Eratosthenes' sieve

First goal: find all primes up to a certain bound B.

Memory requirement. Array of B bits with random access.

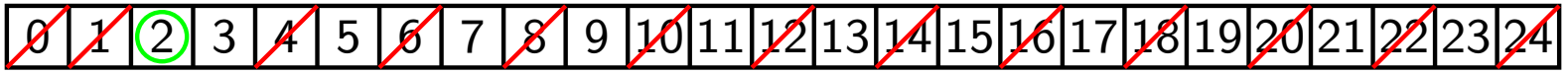


- Initialize the array with 1's everywhere. Let $P = 2$.
- While P is less than $\sqrt{B}$, do:

Eratosthenes' sieve

First goal: find all primes up to a certain bound B.

Memory requirement. Array of B bits with random access.

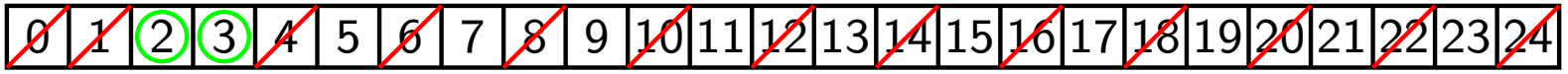


- Initialize the array with 1's everywhere. Let $P = 2$.
- While P is less than $\sqrt{B}$, do:
  - Zero out all array values at positions that are non-trivial multiples of P.

Eratosthenes' sieve

First goal: find all primes up to a certain bound B.

Memory requirement. Array of B bits with random access.

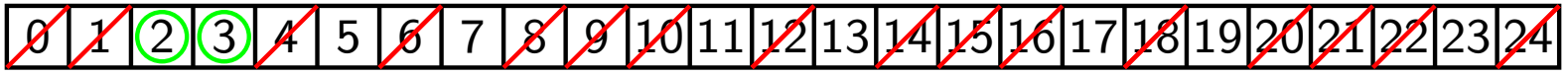


- Initialize the array with 1's everywhere. Let $P = 2$.
- While P is less than $\sqrt{B}$, do:
  - Zero out all array values at positions that are non-trivial multiples of P.
  - Advance P until the array value at P is 1.

Eratosthenes' sieve

First goal: find all primes up to a certain bound B.

Memory requirement. Array of B bits with random access.

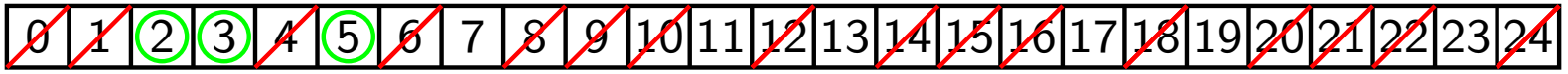


- Initialize the array with 1's everywhere. Let $P = 2$.
- While P is less than $\sqrt{B}$, do:
  - Zero out all array values at positions that are non-trivial multiples of P.
  - Advance P until the array value at P is 1.

Eratosthenes' sieve

First goal: find all primes up to a certain bound B.

Memory requirement. Array of B bits with random access.

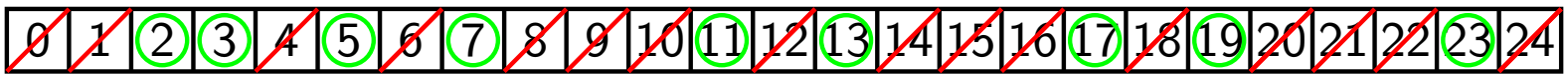


- Initialize the array with 1's everywhere. Let $P = 2$.
- While P is less than $\sqrt{B}$, do:
  - Zero out all array values at positions that are non-trivial multiples of P.
  - Advance P until the array value at P is 1.

Eratosthenes' sieve

First goal: find all primes up to a certain bound B.

Memory requirement. Array of B bits with random access.



- Initialize the array with 1's everywhere. Let $P = 2$.
- While P is less than $\sqrt{B}$, do:
  - Zero out all array values at positions that are non-trivial multiples of P.
  - Advance P until the array value at P is 1.
- Return the indices (≥ 2) of the 1's still in the array.

Eratosthenes' sieve

Time complexity.

The integer P in the loop takes all prime values up to $\sqrt{B}$.

For each P, we visit $\lfloor B/P \rfloor$ positions.

So the total number of operations is $\sum_{P<\sqrt{B} \text{ prime}} \lfloor B/P \rfloor$, which is essentially

$$B \sum_{P<\sqrt{B} \text{ prime}} \frac{1}{P}$$

By Mertens' theorem, this gives a cost of $O(B \log\log B)$ operations.

Caveat

The cost above is given in arithmetic operations (+ memory accesses). Arithmetic on integers below B has bit complexity $O(\log B)$ at least.

Wheel sieving

Starting idea. Most of the primes are odd. Sounds stupid to consider an even integer if we look for a prime (!)

This easily saves a factor of 2.

The Wheel sieve generalizes this. with all small primes up to k.

Let $M = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdots k$, and prepare an array of size M where the position i contains 1 iff i is coprime to M, and 0 otherwise.

Use it as a repeated mask to quickly kill all positions of the array that are divisible by a prime smaller than k.

Taking $k \approx \sqrt{B}$ yields

Thm. (Pritchard) All primes up to B can be computed in $O(B/\log\log B)$ operations.

Segmented sieve

It is not always necessary to allocate the full array beforehand.

- Simple sieves work well in a segmented way, requiring only $O(\sqrt{B})$ storage at a given time, with no significant penalty.
- Harder to do with more advanced version of the wheel sieve.

An interesting property is also random access.

- It is interesting to look for primes in $[a, b]$ and pay only $O(\sqrt{b})$ initialization cost.
- This is particularly useful when parallelizing.

(Cado-NFS: 4d61b4182, 12186c0ab; see also primegen)