Cryptography Week #7: Computational Complexity & Hard problems

Rogério Reis (rogerio.reis@fc.up.pt) 2024/2025 DCC FCUP

November, 8th 2024

Cryptography and absolute security

Given the recent evolution of capacity and speed of computers, is there a real waranty that some cryptographic scheme is realy secure?

A cryptographic algorithm, assumed to be cryptalalytical secure, with a key of lenght 256 bits is it secure? And, if yes, for how long?

Termodynamic limits of computation

A consequence from the **Second Law of Termodynamics** is that a minimum of energy is necessary to change the state of a single bit.

To alter the state of a bit a system takes, at least, kT energy, where T is the absolute temperature of the system, and k is Boltzman constant.

As $k = 1.38 \times 10^{-16}$ erg/K, and the average temperature of the Universe is 3.2 K, using an **idealy perfect** computer at that temperature, needs 4.4×10^{-16} erg to "erase" a bit.

The anual average radiation energy of the Sun is 1.21×10^{41} erg, which is enough to 2.7×10^{56} flips of bit state. Enough to put a register to count from 0 to 2^{187} .

Complexity & Cryptography

Given that the model of secrecy used by modern cryptography is not compatible with "secrecy by obfuscation" its security must rely on the Computational Complexity results.

Turing Machine



- I A Turing machine can both write on the tape and read from it.
- ② The read-write head can move both to the left and to the right.
- ③ The tape is infinite.
- ④ The special states for rejecting and accepting take effect immediately.

Turing-recognisable language

Call a language *L* **Turing-recognisable** if some Turing machine *A* recognises it. That is, if given a word $w \in L$ as input the TM always come to a stop giving a positive answer.

Turing-decidable language

Call a language Turing-decidable or simply decidable if some Turing machine decides it.

Some (actually the vast majority) languages are **not decidable** neither **not recognisable**. The obvious example is the language of Turing Machines that halt with an empty input. (The "halting problem").



Church–Turing thesis

All models of enough expressiveness are equivalent.

The computational complexity of a task is the mesure of the time necessary to accomplish the task as a function of the size of the input.

This is normally done using a Turing Machine using a single tape as reference, but (as long as they are deterministic) other models do not give results that correspond to different classes of complexity.

We will work on the assumption that the choice of the model of computation used is irrelevant in what complexity classification is $concern^1$.

¹Kind of a Church-Turing thesis v2.0.

Bachmann-Landau notation

he "big O" notation
$f:\mathbb{N}\longrightarrow\mathbb{R}$
$f(n) = \mathcal{O}(g(n))$
$\exists k > 0 \exists n_0 \forall n > n_0 : f(n) \le k g(n)$

When f(n) = O(g(n)), we say that g(n) is an upper bound for f(n), or more precisely, that g(n) is an asymptotic upper bound for f(n), to emphasise that we are suppressing constant factors.

Bachmann-Landau notation

- $f_1(n) = 5n^3 + 2n^2 + 55 = O(n^3)$
- $f_1(n) = \mathcal{O}(n^4)$
- $f_1(n) \neq \mathcal{O}(n^2)$, because, $\forall k \forall n_0 \exists n > n_0 : |f_1(n)| > k n^2$
- $f_2(n) = 3n \log_2 n + 5n \log_2 \log_2 n + 2 = O(n \log n)$

Bachmann-Landau Notation

The "small o" notation

f(n) = o(g(n))

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=0$$

Complexity Theory

Traditionally Complexity Theory considers only a special kind of programs that takes an input and and returns an answer of acceptance or rejection of that input. The set of accepted words by one of these programs is called *the language defined by the program* and we say that the program *decides the language* in the sense that it decides, for each word fed as input, if it belongs to the language or not.

Although this model does not cover all the problems that are necessary to study, most of the problems may be divided in components that follow in this definition of "decision program". Thus, we may see the study of the complexity of these "decision programs" as the study of lower bounds for complexity of a more general set of programs.

Complexity Classes

TIME

Let $f : \mathbb{N} \to \mathbb{R}^+$ be a function. Define the time complexity class, TIME(t(n)), to be the collection of all languages that are decidable by an $\mathcal{O}(t(n))$ time Turing machine.

SPACE

Let $t : \mathbb{N} \to \mathbb{R}^+$ be a function. Define the space complexity class, SPACE(t(n)), to be the collection of all languages that are decidable by an Turing machine using a $\mathcal{O}(t(n))$ space in their tape.

How important is the complexity in practice?



Complexity Classes

It is easy to see that if a function f is in TIME(g(n)) then it cannot but be in SPACE(g(n)).

```
\mathsf{TIME}(n^2) \subseteq \mathsf{SPACE}(n^2)
```

The most "important" of the complexity classes is P:

The class P

$$\mathsf{P} = \bigcup_{k \in \mathbb{N}} \mathsf{TIME}(n^k)$$

Thus P is the class of complexity of all the functions that can be decided in a polynomial time by a Turing Machine.

Examples of languages in P

- RELPRIME Given two integers, decide if their greatest common divider is 1, i.e. if they are coprime. The Euclides' algorithm solves the problem in time O(n).
 - PRIMEP Given an integer decide if it is a prime number. Proven in 2002 (*PRIMES is in P*, Agrawal, Kayal and Saxena, 2002). The AKS algorithm runs in time $O(n^{12} \log(n^{12})) = \tilde{O}(n^{12})^2$. Although this a polynomial asymptotic complexity, in practice, and for the size of numbers one wants to use it, the AKS is outperformed by many algorithms of non polynomial complexity.
 - Any CFL Any context-free language have a CFG that recognises it and CYK parser can thus operate in time $\mathcal{O}(n^3)$.

²This is the "soft-O" notation: $\tilde{\mathcal{O}}(f(n)) = \mathcal{O}(f(n)\log f(n))$.

Complexity Classes

In the same manner:

The class PSPACE

$$\mathsf{PSPACE} = \bigcup_{k \in \mathbb{N}} \mathsf{SPACE}(n^k)$$

PSPACE is the class of complexity of the functions that can be decided by a Turing Machine using polynomial space on its tape.

The second most "important" of the complexity classes is NP. NP does **not** stand for "non-polynomial" but for **"nondeterministic polynomial"**, that is languages that (these conditions are equivalent):

• can be decided by a nondeterministic Turing Machine in polynomial time;

• can be verified in polynomial time by a (deterministic) Turing Machine.

By "verified" one means that a Turing Machine with input $\langle w, c \rangle$ can decide if w is part of the language using c as additional information (normally called a witness). The time dependence of the machine only takes in account the size of w and not c.

Examples of languages in NP

Of course, $\mathsf{P}\subseteq\mathsf{NP},$ and thus all the previous examples are in NP.

- COMPOSITES Given an integer decide if it belongs to $\{n \mid \exists p, q, (p, q < n) \land (n = pq)\}$. Trivially is in NP because a verifier is straightforward to write using the list of factors as witness. Although many believe that the problem is in P, we still do not know any algorithm that ensure that.
 - CLIQUE Given a graph G and an integer k, decide if G has a k-clique as a subgraph. As a witness its enough to give the list of the vertices of the k-clique.

$$\text{SUBSET-SUM } \left\{ \langle S,t\rangle \mid S = \{x_1,\ldots,x_k\} \land S' \subseteq S \land \sum_{x \in S'} x = t \right\}$$

DISCRETELOG Given p, a and n decide if there is k s.t.

$$a^k \equiv n \pmod{p}.$$

Complexity Classes

NTIME

Let $f : \mathbb{N} \to \mathbb{R}^+$ be a function. Define the time complexity class, NTIME(t(n)), to be the collection of all languages that are decidable by an $\mathcal{O}(t(n))$ time nondeterministic Turing machine.

NΡ

$$\mathsf{NP} = \bigcup_{k \in \mathbb{N}} \mathsf{NTIME}(n^k)$$

$\mathrm{P}\ vs\ \mathrm{NP}$

- ${\rm P}~=~$ the class of languages with quick membership decision.
- $\mathrm{NP}~=~$ the class of languages with quick membership verification.



One of these is correct! Which one?

The best deterministic method currently known for deciding languages in NP uses exponential time.

EXPTIME

$$\mathsf{NP} \subseteq \mathsf{EXPTIME} = \bigcup_{k \in \mathbb{N}} \mathsf{TIME}\left(2^{n^k}\right)$$

SUPERPOLYTIME is not EXPTIME



Polynomial time computable function A function $f: \Sigma^* \to \Sigma^*$ is a **polynomial time computable function** if some polynomial time Turing machine M exists that halts with just f(w) on its tape, when started on any input w.

Polynomial time reduction

Language A is **polynomial time reducible** to language B, written $A \leq_P B$, if a polynomial time computable function $f : \Sigma^* \to \Sigma^*$, where

$$\forall w \ w \in A \iff f(w) \in B.$$

The function f is called the **polynomial time reduction of** A **to** B.



Theorem

If $A \leq_P B$ and $B \in P$, then $A \in P$.

NP-complete

A language B is NP-complete if it satisfies:

(1) B is in NP, and

② $\forall A \in NP(A \leq_P B)$. (we say that B is NP-hard)

Theorem

If B is NP-complete and $B \in P$, then P = NP.

Theorem

If B is NP-complete and $B \leq_P C$ for $C \in NP$, then C is NP-complete.

The set of NP-complete languages (problems) is the set of hardest languages (problems) in NP.

Examples of NP-complete languages

TRAVELING SALESMAN PROBLEM Given a graph with weights labelling each edge decide if there is a path through all the vertices with a total sum not exceeding a given x. CLIQUE

KNAPSACK Given set of integers S and two additional integers x and y decide if there is a $Q \subseteq S$ such that $x \leq \sum_{q \in Q} q \leq y$.

 $\mathsf{P}\subseteq\mathsf{NP}\subseteq\mathsf{PSPACE}=\mathsf{NPSPACE}\subseteq\mathsf{EXPTIME}$

The Factoring Problem

prime number

A number p is prime if its only positive factors are 1 and p.

Fundamental Theorem of Arithmetic

For any positive integer n there is a unique factorisation of n as a product of increasing primes.

The factoring problem consists of finding the prime numbers p and q given a large number, N = pq. The widely used RSA algorithms are based on the fact that factoring a number is difficult. In fact, the hardness of the factoring problem is what makes RSA encryption and signature schemes secure. This problem is indeed **hard**, yet probably **not NP-complete**.

How to factor an integer?

To factor the number *n* we can try to divide *n* by every $i \in \{2, ..., n-1\}$. This will take a time $\mathcal{O}(n)$.

But we can do better...

We can try to divide *n* by every $i \in \{2, ..., \lfloor \sqrt{n} \rfloor\}$. We have reduced the time to $\mathcal{O}\left(n^{\frac{1}{2}}\right)$ Still, we can improve.

We can only try to divide *n* by the primes that are smaller than \sqrt{n} . By the prime number theorem we know that the number of primes below \sqrt{n} is approximately $\frac{\sqrt{n}}{\log \sqrt{n}}$. This makes the task faster. But still we need 2¹²⁰ operations for a 256-bit integer. Not good enough!

The fastest factoring algorithm is the **general number field sieve (GNFS)**, with an average time to operate for a number n of

 $e^{1.91n^{\frac{1}{3}}(\log n)^{\frac{1}{3}}}.$

Factoring a 1	1024-bits integer	2 ⁷⁰ operations
Factoring a 2	2048-bits integer	2 ⁹⁰ operations
Factoring a 4	1096-bits integer	2 ¹²⁸ operations

- In 2005, after about 18 months of computation and thanks to the power of a cluster of 80 processors, with a total effort equivalent to 75 years of computation on a single processor—a group of researchers factored a 663-bit (200-decimal digit) number.
- In 2009, after about two years and using several hundred processors, with a total effort equivalent to about 2000 years of computation on a single processor, another group of researchers factored a 768-bit (232-decimal digit) number.

Thus the estimates are **very optimistic** regarding the possible performance of computers and algorithms.

So we have a problem that is in NP and that looks hard, but is it as hard as the hardest NP problems? In other words, is factoring NP- complete? **Probably not.**

Factoring may then be slightly easier than NP-complete in theory, but as far as cryptography is concerned, it's hard enough, and even more reliable than NP-complete problems. Indeed, it's **easier** to build cryptosystems on top of the factoring problem than NP-complete problems, because it's hard to know exactly how hard it is to break a cryptosystem based on some NP-complete problems—in other words, how many bits of security you'd get.

On the other hand... we may have to deal with quantum computers...

... and if we are not careful, factoring can became easy!

To factor 17976931348623159077293051907890247336179769789423065 7273430081157739343819933842986982557174198257278917258638193 7092658191860266261806597306650627109955565786394477156084151 8689565284169198292110720231716536912489048151238855803905342 7125099290315449262324709315263256083132540461407052872832790 915388014592 takes just a few seconds because its factors are: 2⁸⁰⁰, 641, 6700417, 167773885276849215533569 and 37414057161322375957408148834323969.

The Discrete Logarithm Problem

Consider the multiplicative group \mathbb{Z}_p^* (with p prime). The **DLP** consists, given g and x, in finding y such that $g^y = x$ in \mathbb{Z}_p^* , i.e.

 $g^y \equiv x \pmod{p}$.

In this conditions **DLP** seems as hard as the integer factoring problem.

