Cryptography
Week #7:
Computational Complexity & Hard problems

Rogério Reis, rogerio.reis@fc.up.pt
MSI/MCC/MIERSI - 2025/2026
DCC FCUP

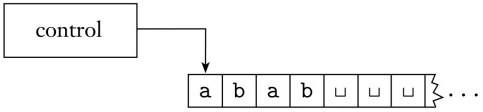
November, 7th 2025

Complexity & Cryptography

Given that the model of secrecy used by modern cryptography is not compatible with "secrecy by obfuscation"[1] its security must rely on the Computational Complexity results.

[1] Kerckhoffs' "law"

Turing Machine



1. 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.

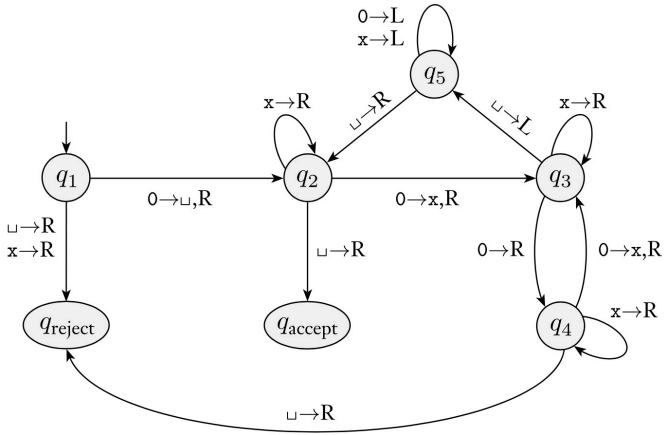
Turing-recognisable language

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").



Turing machine that decides $A = \{0^{2^n} | n \geq 0\}$.

Church–Turing thesis

All models of enough expressiveness are equivalent.

Turing Machine variants

ManyTapesInOneSimulator **Input:** $w = w_1 w_2 \cdots w_n$

1. The tape is initialised to simulate the k tapes of A as

$$\#w_1' w_2 \cdots w_n \#\_' \#\_' \# \cdots \#\_' \#$$

2. To simulate each step of A, sweeps the tape from the first # up to the $(k+1)$th # to get the symbols in the head positions of each tape. Then updates "the tapes" as established by A's program.
3. At any point if it moves a "virtual head" to a #, this means that A would have that head on a non written cell. Thus writes $\_'$ and shifts the content of the whole rest of the tape, one cell, to the right, and continues the simulation.

Universal Turing machine

Theorem

There is a TM $\mathcal{U}$ that, for any input (x, α) one has $\mathcal{U}(\langle x, \alpha \rangle) \equiv M_\alpha(x)$, where M_α is the TM such that $\langle M_\alpha \rangle = \alpha$. Furthermore, if M_α with input x stops after t steps, then $\mathcal{U}(\langle \alpha, x \rightarrow)$ stops after $ct \log t$ steps, where c is a constant depending solely of M_α (number of tapes, number of states and size of the alphabets).

The proof is quit straightforward using 3 tapes.

Non-deterministic Turing machines

The same way that a Finite automaton may have a nondeterministic behaviour definition (or a PDA), also a TM may have a non-deterministic behaviour. In that case

$$\delta : Q \times \Gamma \to 2^{Q \times \Gamma \times \{R,L\}}.$$

Of course, now, a TM can reach an accepting **and** a rejecting state with the **same** input. So we need a new semantic for the language recognised by a nondeterministic TM. The way is done is by valuing differently the positive and the negative answers. Thus, we say that TM M **accepts** input x if there is a branch (or if you prefer, an instance of the TM M) that ends in an accepting configuration. Of course it it does not make sense to have a **decisor** semantics for a nondeterministic TM, because a negative answer may never occur because some branches may simply enter in never-ending loops.

Surprisingly (or not!) the following theorem states that, in our most advanced model of computation, **non-determinism is (again) just a syntactic artifice**.

Theorem

For any nondeterministic TM N there is a deterministic TM D that simulates it.

Proof:

To prove this we will use a 3-tape TM (because we can!) to make its behaviour clearer.
Let the tapes of D be called T_1, T_2 and T_3.

- T_1 will contain the input w and will be used as a **read-only** tape.
- T_3 will be the "working tape", and it will be continuously erased and re-written, every time computation changes "branch".
- T_2 will contain the "branch address".

Let m be the maximum number of nondeterministic branches in any state (that is trivially bounded by $|\Sigma|.|\Gamma|.|Q|.2$).

What do we mean by "branch address"?

D **Input:** *w*

1. Initially T_1 contains w and T_2 and T_3 are empty.
2. Copies T_1 to T_3.
3. Uses T_3 to simulate N with input w in one of branches of its nondeterministic execution, in the following manner. Executes N using the symbols in T_2 to choose which nondeterministic option to assume. If the end of tape T_2 is reached, the address is invalid or reaches a rejecting configuration, jumps to 4. If it reaches an accepting configuration (of N), stops and accepts the input.
4. Re-writes T_2 inscribing the sucessor of the previous integer (in base m). Returns to 2.

□

Corollary

A language L is **recognisable** *if and only if there is a nondeterministic TM that recognise it.*

Enumerators

Another kind of device is one that has **two** tapes (T_1 and T_2) instead of just one tape. The T_1 is used as "working tape" and T_2 is used as an "output tape" (and thus, "write only"). This device inscribes in T_2 all the words of the corresponding language, using a new symbol "#" to delimit words. On T_2 the words do not need to appear in any pre-defined order.

Theorem

A language L is recognisable if, and only if, it exists an **enumerator** *for L.*

Proof: Suppose that L has an enumerator E, then one can construct a TM M that acts as a **recogniser** for L.

M **Input:** w

1. Executes E, step by step, and every time that "#" is written in T_2 compares w with the last word written on T_2.
2. If the comparison is successful, accepts w and stops.

Now, suppose that there is a recogniser R for L. We can construct an enumerator E for L using an enumerator E_Σ of all the words in $\Sigma^\star$.

E

For $i = 1, 2, 3, \ldots$ proceed in the following way:

1. Obtain, from E_Σ, the list of words $w_1, w_2, \ldots, w_i$ and for each of these words executes i steps of R with each of these words as input.
2. If in the ith step R accepts any of the words, put those words in T_2, and continues.

□

Examples of decidable languages

All problems that are expressed in terms of regular languages are decidable!

Theorem

$A_{CFG} = \{\langle G, w\rangle \mid G \text{ is a CFG}, w \in \mathcal{L}(G)\}$ *is decidable.*

Proof:

A decisor can be defined as

S **Input:** $\langle, G, w\rangle$

1. Writes G in CNF.
2. Lists all derivations with $2n-1$ steps ($|w| = n$).
3. Verifies if w is produced by G.

□

Theorem

$E_{CFG} = \{\langle G \rangle \mid \mathcal{L}(G) = \emptyset\}$ *is decidable.*

Proof:

R decides E_{CFG}:

R **Input:** $\langle G \rangle$

1. Marks all terminal symbols (including ε).
2. Repeats until saturated
   1. Marks all symbol A with rule $A \to U_1 U_2 \cdots U_n$ where U_i are marked, $\forall i$.
3. If the starting symbol is marked **rejects**, otherwise **accepts**.

□

Examples of recognisable languages

Theorem

$A_{TM} = \{\langle M, w\rangle \mid M \text{ is a } TM, w \in \mathcal{L}(M)\}$ is recognisable.

Proof:

It is enough to use the **Universal TM** $\mathcal{U}$:

$\mathcal{U}$ **Input:** $\langle M, w\rangle$

1. Simulates M with input w.
2. If M accepts, $\mathcal{U}$ **accepts**.

Theorem

$\overline{E_{TM}} = \{\langle M \rangle \mid \mathcal{L}(M) \neq \emptyset\}$ *is recognisable.*

Proof: Again, using $\mathcal{U}$:

R **Input:** $\langle M \rangle$

1. Let $s_1, s_2, \ldots$ be the sequence of the words with the alphabet of M.
2. For $i \in \mathbb{N}$
   1. Simulates i steps of M with input s_k for $k \leq i$.
   2. If any simulation reaches an acceptance configuration, stops, and **accepts**.

□

Theorem

A_{TM} *is* **undecidable**.

Proof:

By contradiction suppose that we have H deciding A_{TM}

H **Input:** $\langle M, w \rangle$

1. If M accepts w, **accept**.
2. Else, **rejects**.

Then we can construct D

D **Input:** $\langle M \rangle$

1. If H with input $\langle M, \langle M \rangle \rangle$ accepts, then D **rejects.**
2. If H with input $\langle M, \langle M \rangle \rangle$ rejects, then D **acepts.**

Thus

$$D(\langle M \rangle) \begin{cases} \textbf{accepts}, & \text{if } M \text{ does not accept } \langle M \rangle \\ \textbf{rejects}, & \text{if } M \text{ accepts } \langle M \rangle \end{cases}$$

What happens if we execute D with input $\langle D \rangle$?

$$D(\langle D \rangle) \begin{cases} \textbf{accepts}, & \text{if } D(\langle D \rangle) \text{ rejects,} \\ \textbf{rejects}, & \text{if } D(\langle D \rangle) \text{ accepts.} \end{cases}$$

Clearly an **absurd** resulting of the supposition of existence of H.

Thus H **cannot exist!**

□

We could have used Cantor's "diagonal argument" to construct D.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\cdots$	$\langle M_k \rangle$	$\cdots$
M_1	A	A	R	$\cdots$	R	$\cdots$
M_2	A	R	A	$\cdots$	A	$\cdots$
M_3	A	A	R	$\cdots$	A	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$D = M_k$	R	A	A	$\cdots$	?	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

Computational Complexity

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[2].

[2] Kind of a Church-Turing thesis v2.0.

Landau notation

The "big O" notation

$$f(n) = \mathcal{O}(g(n))$$

$$\exists k > 0 \, \exists n_0 \, \forall n > n_0 : |f(n)| \leq k \, g(n)$$

When $f(n) = \mathcal{O}(g(n))$, 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.

Landau notation

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

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 covers 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

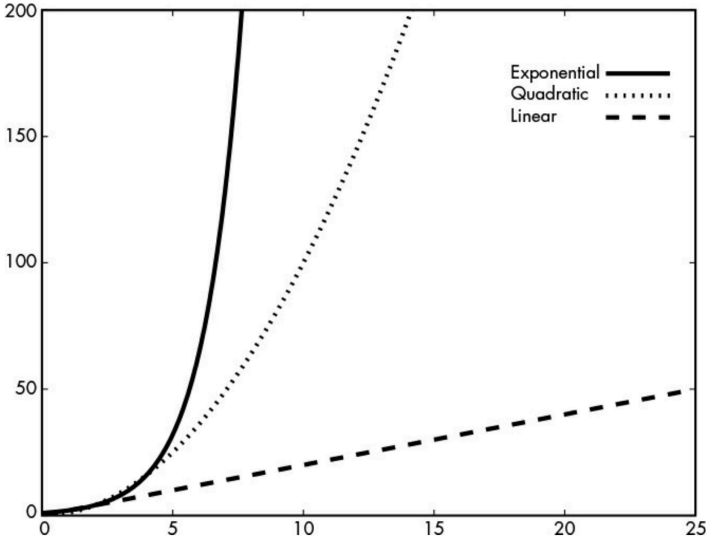
TIME

Let $t : \mathbb{N} \to \mathbb{R}^+$ 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 $\mathsf{TIME}(g(n))$ then it cannot but be in $\mathsf{SPACE}(g(n))$.

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

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 $\mathcal{O}(n)$.

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 $\mathcal{O}(n^{12}\log(n^{12})) = \tilde{\mathcal{O}}(n^{12})$[3]. 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 $\mathcal{O}(n^3)$.

[3] This 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

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

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

Complexity Classes

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

COMPOSITES Given an integer decide if it belongs to $\{n \mid \exists p, q, (p, q < n) \wedge (n = pq)\}$. 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.

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

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, $\mathsf{NTIME}(t(n))$, to be the collection of all languages that are decidable by an $\mathcal{O}(t(n))$ time nondeterministic Turing machine.

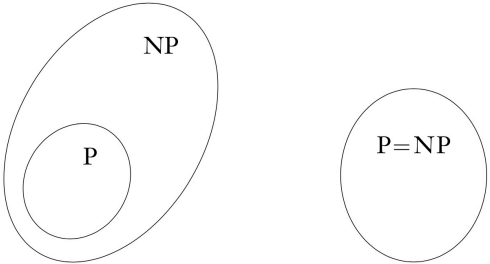
NP

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

P vs NP

P = the class of languages with quick membership decision.

NP = the class of languages with quick membership verification.



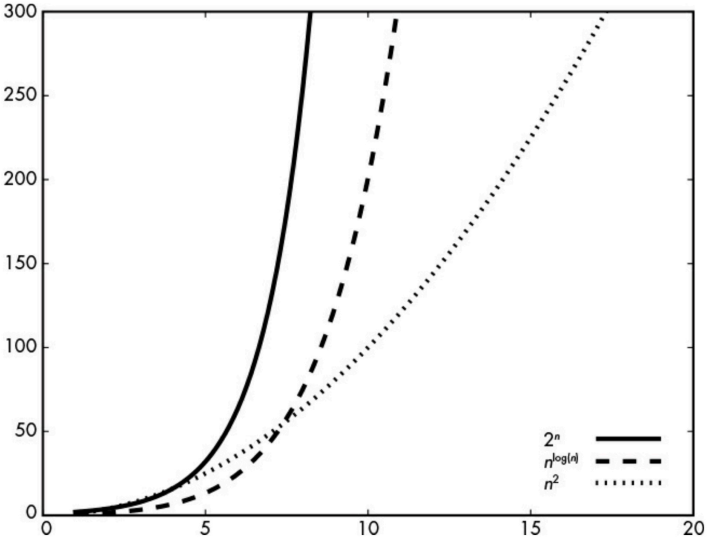
One of these is correct! Which one?

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



NP-Completness

Polynomial time computable function

A function $f : \Sigma^\star \to \Sigma^\star$ 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.

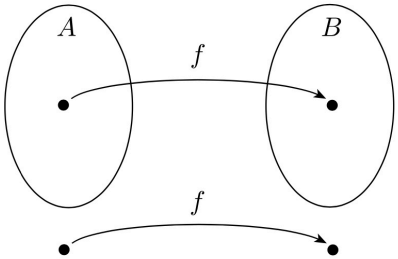
NP-Completness

Polynomial time reduction

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

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

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



NP-Completeness

Theorem

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

NP-complete

A language B is NP-complete if it satisfies:

1. B is in NP, and
2. $\forall A \in \mathrm{NP}(A \leq_P B)$. (we say that B is NP-hard)

Theorem

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

Theorem

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

NP-Completness

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 Given a graph G and an integer k to decide if G has as a sub-graph a clique of size k.

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

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

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 $i \in \{2, \ldots, n-1\}$. This will take a time $\mathcal{O}(n)$.
But we can do better...
We can try to divide n by every $i \in \{2, \ldots, \lfloor \sqrt{n} \rfloor\}$. 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 2^{120} 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

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

Factoring a 1024-bits integer	2^{70} operations
Factoring a 2048-bits integer	2^{90} operations
Factoring a 4096-bits integer	2^{128} operations

1. 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.
2. 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
72734300811577393438199338429869825571741982572789172586381 93
70926581918602662618065973066506271099555657863944771560841 51
86895652841691982921107202317165369124890481512388558039053 42
71250992903154492623247093152632560831325404614070528728327 90 915388014592 takes just a few seconds because
its factors are: 2^{800}, 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.