Asymmetrically hard

Notes for *Serious Cryptography* chapter 9

Jeffrey Goldberg
jeffrey@goldmark.org

August 28, 2024

About asymmetry

Encryption is easy

I can easily encrypt text with a key, say
`4f232f55df69d9f678814768658 0e854`

Encryption with known key

```
% echo -n "What key's used?" | \
  openssl enc -aes-128-ecb -nopad -a \
  -K 4f232f55df69d9f678814768658 0e854 \
  -out enc-demo.txt
```

This is ECB mode on a single, unpadded block. It will always produce the same result in file `enc-demo.txt`

Decryption is easy (with the key)

Decryption with the key

```
% openssl enc -d -aes-128-ecb -nopad -a \
  -K 4f232f55df69d9f6788147686580e854 \
  -in enc-demo.txt
What key's used?
```

Encryption is easy

Encryption with ephemeral key

```
% echo -n "What key's used?" | \
  openssl enc -aes-128-ecb -nopad -a \
  -K $(openssl rand 16 -hex)
FbREw7G228g2ScV2aWDTKQ==
```

The key is quickly forgotten and never seen. The output, this time, is `FbREw7G228g2ScV2aWDTKQ==`

About asymmetry

Hard to compute. Easy to verify

Can't find the key

Plaintext	`What key's used?`
Ciphertext	`FbREw7G228g2ScV2aWDTKQ==` (Base64 encoded)
Algorithm	AES-128, ECB, no-padding
Key	(Unfeasible to find; easy to verify)

Easy verification

If someone claims to have the key, anyone with the knowledge of the plaintext, ciphertext, and algorithm can easily see if the claim is true.

Nondeterministic Polynomial (NP)

NP problems

Definition (NP problems)

A problem is among the non-deterministic polynomial (**NP**) problems if it is easy to verify whether a claimed solution is correct.

Definition (P problems)

A problem is among the polynomial (**P**) problems if it is easy to compute a solution.

Asymmetrically hard

Nondeterministic Polynomial (NP)

NP problems

Definition (NP problems)

A problem is among the non-deterministic polynomial (**NP**) problems if it is easy to verify whether a claimed solution is correct.

Definition (P problems)

A problem is among the polynomial (**P**) problems if it is easy to compute a solution.

The use of distinct typography for **P** and **NP** is to highlight that these are mathematical objects representing sets of problems.

Assume the hardest

Technically, every thing in **P** is also in **NP**. In normal conversation, however, when people say that something is in **NP** they usually mean "in **NP** and not in **P**."

Implicature of Maximality

"Max has two dogs" is (pedantically) true even if Max has three dogs. But in most contexts it carries the implicature that Max has exactly two dogs.

Asymmetrically hard

Nondeterministic Polynomial (NP)

Assume the hardest

Technically, every thing in **P** is also in **NP**. In normal conversation, however, when people say that something is in **NP** they usually mean "in **NP** and not in **P**."

Implicature of Maximality

"Max has two dogs" is (pedantically) true even if Max has three dogs. But in most contexts it carries the implicature that Max has exactly two dogs.

In a context of Dad jokes, conversational implicatures go out the window.

Fame and fortune

- There is no proof that there are problems which are in **NP** but not in **P**.
- Fame and fortune awaits anyone who can remedy that situation with a proof either way.
- We all assume that there are **NP** problems that are not **P**. We have a very good ideas of what some of them are. A proof that $\mathbf{P} = \mathbf{NP}$ would be extremely surprising and really bad for cryptography.

Asymmetrically hard
Nondeterministic Polynomial (NP)
Fame and fortune

Fame and fortune

- There is no proof that there are problems which are in **NP** but not in **P**.
- Fame and fortune awaits anyone who can remedy that situation with a proof either way.
- We all assume that there are **NP** problems that are not **P**. We have a very good ideas of what some of them are. A proof that **P** = **NP** would be extremely surprising and really bad for cryptography.

The reasons for believing that **P** $\neq$ **NP** are not just wishful thinking.

Hard and easy

In one (useless) sentence

Definition (Hard)

A problem is hard if and only if there is no probabilistic polynomial time algorithm which can solve it for all valid input.

That definition is useless to those who don't already know what "hard" means, and it is pointless to repeat for those who do.

Hard and easy

Setting boundaries

Time and length

- It takes more time (steps) to multiply two 10-digit numbers than to multiply two 2-digit numbers.
- It takes more time to multiply two 100-digit numbers than to multiply two 10-digit numbers.
- It takes more time to multiply two 5000-digit numbers than to multiply two 100 digit numbers.

- It takes more time (steps) to multiply two 10-digit numbers than to multiply two 2-digit numbers.
- It takes more time to multiply two 100-digit numbers than to multiply two 10-digit numbers.
- It takes more time to multiply two 5000-digit numbers than to multiply two 100 digit numbers.

1. Should we count in digits or bits? It doesn't matter here as those just differ by a constant, $\log_2(10)$. We tend to use bits for other reasons.
2. Multiplication is sub-linear.

An example complexity

Suppose that the number of operations needed to perform some computation on input size n is something like this polynomial

$$f(n) = 4.5n^3 - 100n^2 + 50000n + 10000000 \tag{1}$$

A sort of runic rhyme

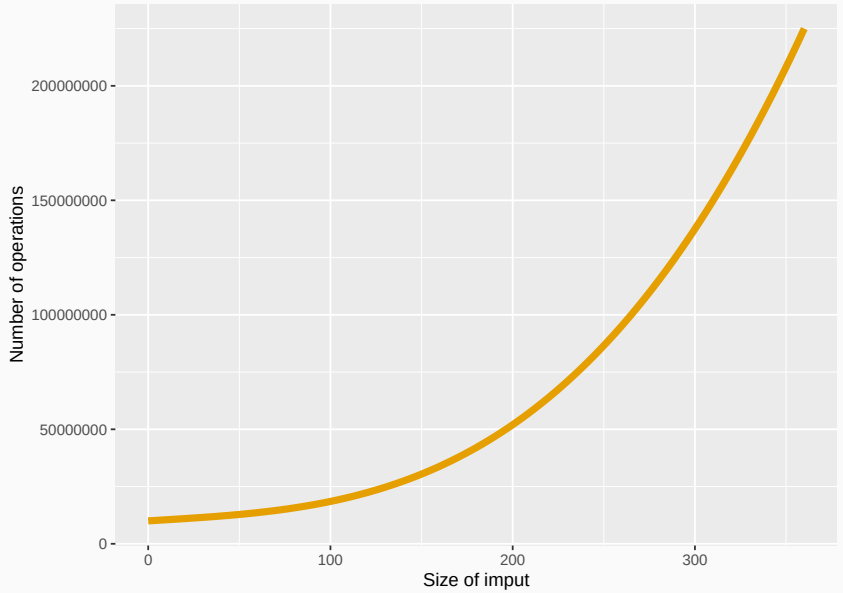

Figure 1: Time as a function of size of input

Setting an upper bound

An upper bound for $f(n)$ is

$$g(n) = 5n^3 \tag{2}$$

Beginning bounds

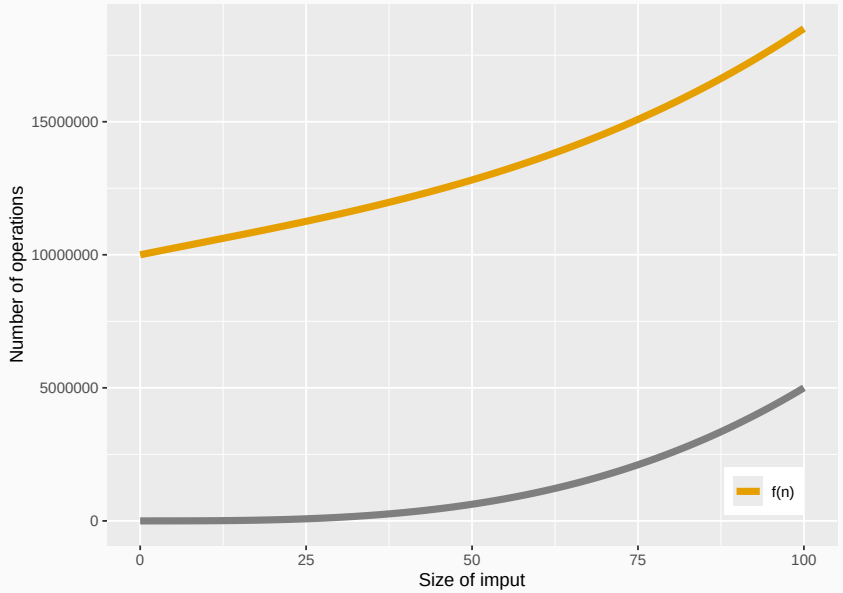

Figure 2: Time may exceed bound for small n

In the long run

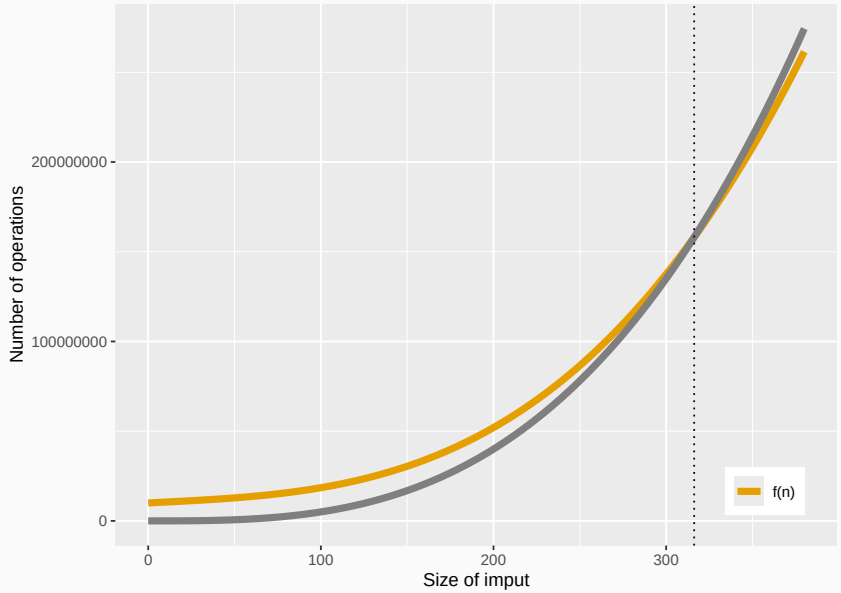

Figure 3: After a point, the bound is above

Big O

Repeating the example

So that I don't have to jump back and forth between slides

Example ($f(n)$ and $g(n)$)

$$f(n) = 4.5n^3 - 100n^2 + 50000n + 10000000 \tag{1}$$

$$g(n) = 5n^3 \tag{2}$$

There is some number, c

When we say that $f(n)$ is in $\mathcal{O}(n^3)$ we are saying that there is some number c such that cn^3 provides an upper bound to $f(n)$ in the way described in above examples.

In our example, c can be any number greater 4.5.

We don't care about c. We only care that it exists.

There is some number, $\hat{n}$

When we say that $f(n)$ is in $\mathcal{O}(n^3)$ we are also saying that there is some number $\hat{n}$ such that $f(n) \leq cn^3$ for all values of n that are greater than $\hat{n}$. In our example, $\hat{n}$ is around 316.23.

In our example, $f(n) \leq 5n^3$ for all $n > 100\sqrt{10}$.

We don't care about $\hat{n}$. We only care that it exists.

Learning not to care

Big O notation and many useful results in computational complexity requires us to not care about c and $\hat{n}$ beyond the fact that they must exist. It is hard not to care, especially because there are times when we have to, but

Theorem

Life goes easier for you if you learn not to care.

(Please do not quote me out of context.)

Learning not to care

Big O notation and many useful results in computational complexity requires us to not care about c and $\hat{n}$ beyond the fact that they must exist. It is hard not to care, especially because there are times when we have to, but

Theorem

Life goes easier for you if you learn not to care.

(Please do not quote me out of context.)

1. Rob Pike [Pik87] (when talking about unhelpful optimzations) once said, “fancy algorithms are slow when n is small, and n is usually small.” In cryptography we design systems with small enough c and $\hat{n}$ so that we do get to a place where we can used large enough n in practice to get the asymmetry we want.

When '$=$' $\neq$ "equals"

$f(n)$ is *in* $\mathcal{O}(n^3)$.

This is often written as "$f(n) = \mathcal{O}(n^3)$", but the equal sign should be read as "is in".

$$f(n) = \mathcal{O}(n^3)$$
$$g(n) = \mathcal{O}(n^3)$$
$$\text{But } f(n) \neq g(n)$$

Anything above

Any function in $\mathcal{O}(n^3)$ is also in $\mathcal{O}(n^4)$, which in turn is in $\mathcal{O}(n^{345})$, which in turn is in $\mathcal{O}(e^{\log n})$, which in turn is in $\mathcal{O}\left(e^{\sqrt{n}}\right)$, which in turn is in $\mathcal{O}(e^n)$, which in turn is in $\mathcal{O}\left(e^{n^2}\right)$, which in turn is in $\mathcal{O}(n!)$, and many more.

Implicature (again)

$h(n)$ is in $\mathcal{O}(n^5)$ is (pedantically) true even if $h(n)$ is in $\mathcal{O}(n^2)$. But in many contexts it carries the implicature that there is no reason to believe that $h(n)$ is in $\mathcal{O}(n^4)$.

Big O

Complexity classes

Polynomial time (and smaller)

A function is polynomial time if there exists a number k – the degree of a bounding polynomial – such that the function is in $\mathcal{O}(n^k)$. k must not depend on n.

Order	*Degree*	*Buzzword*
$\mathcal{O}(n^{84})$	$k = 84$	
$\mathcal{O}(n^2)$	$k = 2$	(quadratic time)
$\mathcal{O}(n)$	$k = 1$	(linear time)
$\mathcal{O}(\sqrt{n})$	$k = 0.5$	
$\mathcal{O}(\log n)$	unreal	(logarithmic time)
$\mathcal{O}(1)$	$k = 0$	(constant time)

Table 1: Some polynomial times, in decreasing size

Polynomial time (and smaller)

A function is polynomial time if there exists a number k – the degree of a bounding polynomial – such that the function is in $\mathcal{O}(n^k)$. k must not depend on n.

Order	*Degree*	*Buzzword*
$\mathcal{O}(n^{84})$	$k = 84$	
$\mathcal{O}(n^2)$	$k = 2$	(quadratic time)
$\mathcal{O}(n)$	$k = 1$	(linear time)
$\mathcal{O}(\sqrt{n})$	$k = 0.5$	
$\mathcal{O}(\log n)$	unreal	(logarithmic time)
$\mathcal{O}(1)$	$k = 0$	(constant time)

Table 1: Some polynomial times, in decreasing size

1. To the extent we can say something about k in the logarithic case, we can say that $k > 0$ and k is smaller than any positive real number.

Sub-exponential

Everything in polynomial time is also sub-exponential, but here we give examples often things that are sub-exponential and *not* polynomial.

- $\mathcal{O}(e^{\log n})$
- $\mathcal{O}\left(e^{\sqrt{n}}\right)$

These are of the form of $\mathcal{O}\left(e^{g(n)}\right)$ where $g(n) < \mathcal{O}(1)$. That is the exponent is a function of n that is sub-linear.

Exponential

Everything in sub-exponential is also in exponential time, but here we give examples often things that are exponential and *not* sub-exponential.

- $\mathcal{O}(e^n)$
- $\mathcal{O}\left(e^{n^2}\right)$

These are of the form of $\mathcal{O}\left(e^{g(n)}\right)$ where the order of $g(n)$ is greater than or equal to $\mathcal{O}(1)$. That is, the exponent is a function of n that is linear or greater.

Problems and algorithms

We've talked about the complexity (time) of particular algorithms, but we are interested in problems.

Definition (Problem complexity)

A problem has a particular complexity if there is at least one algorithm of that particular complexity which can solve the problem.

Example

The problem of listing all possible bit sequences of length n is $\mathcal{O}(2^n)$. There is no sub-exponential algorithm for doing so.

A not so useless definition

Definition (Hard problem)

A problem is hard if and only if there is no probabilistic polynomial time algorithm which can solve it for all valid inputs.

Definition (Hard problem)

A problem is hard if and only if there is no probabilistic polynomial time algorithm which can solve it for all valid inputs.

1. Cryptography has a slightly stronger requirement. We need a problem to be hard for most average inputs. A problem that is hard only for a small class of inputs is of no use.

NP (again)

NP

A problem is in **NP** (non-deterministic polynomial) if and only if there is a probabilistic polynomial time algorithm which can verify any candidate solution.

Big O

NP-complete

NP-complete

Two things to know about the **NP**-complete category of problems

1. A polynomial time algorithm that solves any **NP**-complete problem can be transformed into a polynomial time solution for any other **NP**-complete problem.
2. If there is a polynomial time algorithm to solve any **NP**-complete problem then there is a polynomial time algorithm for any **NP** problem.

Those two points are subtly different.

NP-COMPLETE

Two things to know about the **NP**-complete category of problems

1. A polynomial time algorithm that solves any **NP**-complete problem can be transformed into a polynomial time solution for any other **NP**-complete problem.
2. If there is a polynomial time algorithm to solve any **NP**-complete problem then there is a polynomial time algorithm for any **NP** problem.

Those two points are subtly different.

1. If someone finds a polynomial time algorithm to solve, say, 3-SAT (an **NP**-complete problem) then all problems in **NP** are in **P**, including factoring and the DLP. But a polyinomial time solution for 3-SAT will not necessarily give as a poly solution to factoring; it will only tell us that one exists. It will, however, give us a poly time algorithm to every other NP-complete problem because we can translate (in polynomial time) any **NP**-complete problem to 3-SAT.

NP-complete: Another take

Complete facts

1. All **NP**-complete problems are really just variants of a single problem.
2. That problem is at least as hard as anything in **NP**.

Are factoring and DLP NP-complete?

Factoring and the DLP do not smell like **NP**-complete problems, but there is no proof either way.

Generalizing the DLP

The cryptographically useful properties of the discrete logarithm problem requires a finite cyclic group (with a few other conditions on the group).

The group does not need to have integer elements and modular multiplication. It can be constructed from other things if it has the right structure. The term “generalized DLP” (GDLP) is sometimes used to talk about this generalization of the the DLP.

Happy groups

Все [конечные циклические группы] похожи друг на друга, каждая несчастливая группа [структурирована] по своему

All [finite cyclic groups] are alike, but each unhappy group is [structured] in its own way.

(with apologies to) Leo Tolstoy

Happy groups

Все [конечные циклические группы] похожи друг на друга, каждая несчастливая группа [структурирована] по своему

All [finite cyclic groups] are alike, but each unhappy group is [structured] in its own way.
(with apologies to) Leo Tolstoy

1. I spent way too much time on this slide. Particularly because I thought that ‘*г*’ and ‘*и*’ were some sort of error from a bad font mapping or encoding. It turns out that these are the correct Cyrillic letter forms for lowercase *italic* ‘г’ and ‘*и*’.
2. I really should have this about *abelian* finite cyclic groups, but I don’t want to bother Tim with more translation requests.

Elliptic curves

It is possible to define a group with the right structure using elliptic curves. See the computation-examples document for more information.

The operation is called “point addition” and so the notation is different.

Notation wars

Operation	$\mathbb{Z}_p^\times$	Elliptic Curve group
Op. name	mod. multiplication	point addition
	ab	$P + Q$
Repeated	exponentiation	scalar multiplication
	x^n	nP
From generator	$A = g^a$	$P = pG$
Logarithm	Find a given A, g	Find p given P, G
	$\log_g A$	`discrete_log`(P, G)

Table 2: Terms and notation for integer and elliptic curve groups

Notation wars

Operation	$\mathbb{Z}_p^\times$	Elliptic Curve group
Op. name	mod. multiplication	point addition
	ab	$P+Q$
Repeated	exponentiation	scalar multiplication
	x^n	nP
From generator	$A = g^a$	$P = pG$
Logarithm	Find a given A, g	Find p given P, G
	$\log_g A$	`discrete_log`(P, G)

Table 2: Terms and notation for integer and elliptic curve groups

1. ECs have a history in geometry, and so points are typically referred to using capital letters like P and Q, while in integer groups p is often used for the prime modulous.
2. When talking abstractly about groups, G is often used to refer to a group, but with elliptic curves, it is often used to describe the generator or base-point.
3. “Scalar” means ordinary number. n is a point.

DLP on elliptic curves

Given a suitably defined elliptic curve group $\mathcal{C}$ and a generator point G within the group and some integer d, a point P is easy to compute as in equation 3.

$$P = dG \qquad \text{(within group } \mathcal{C}\text{)} \tag{3}$$

Computing d from P is hard and is called finding the discrete logarithm of P.

GDLP Assymmetry

A carefully contrived elliptic curve group with approximately 20 bit security of

$$y^2 = x^3 + 522510265620x + 189830553521 \pmod{872078115079} \tag{4}$$

with generator $G = (842965367165, 470264993162)$ computing

$$P = pG \tag{5}$$

with $p = 266038276613$ took under 1 second, while computing p from P and G took about 25 seconds.

GDLP Assymmetry

A carefully contrived elliptic curve group with approximately 20 bit security of

$$y^2 = x^3 + 522510265620x + 189830553521 \pmod{872078115079} \tag{4}$$

with generator $G = (842965367165, 470264993162)$ computing

$$P = pG \tag{5}$$

with $p = 266038276613$ took under 1 second, while computing p from P and G took about 25 seconds.

The software that I happen to be using is not nearly as well optimized for elliptic curve operations as it is for integer operations.

Notes on the Reading

Notes on the Reading

Existence of a solution

Absense of a solution

> *To verify that no solution exists, you may need to go through all possible inputs. [Aum17, ch 9, §NP-time]*

I carefully contrived my initial encryption example to encrypt a single block in ECB mode. Because that is a pseudo-random permutation (PRP), a solution must exist.

A PRP implies a key

AES is a PRP on a 16-byte block. That means that there must exist a key that will encrypt a specific block to some other block.

Ptext `Can you find key`

Ctext `that comes to me`

Alg AES-128, ECB, no-padding

Key (The key will be hard to find, but easy to verify)

Notes on the Reading

Fields and Groups

SRP is weird

The PAKE (Password-based Authenticated Key Exchange) scheme we use for authentication in 1Password is the Secure Remote Password (SRP) protocol.

- Not all groups are fields
- All fields are groups
- Elliptic curves do not give us a fields
- $\mathbb{Z}_p^\times$ is a field.
- SRP requires its operations to be over a field.

Notes on the Reading

Miscellany

Math rendering

The ePub (when viewed with Apple Books app) doesn't always display the math correctly, while the print book does.

Space, Time, and Spacetime

To avoid yet another complexity, I pretended that running time (number of operations) as the only thing to count.

But the analogous thing should be done for memory requirements. Indeed, some problems are specifically designed to be memory hard. (Modern slow hashes are the prime example.)

Downgrade attacks

> *Not only did servers support a weak algorithm, but attackers could force a benign client to use that algorithm by injecting malicious traffic within the client's session [Aum17, ch 9, §small]*

Offering backwards compatibility with insecure systems tempting, but dangerous.

There are enough big primes

Theorem (Prime number theorem)

The number of prime numbers less than x approaches $\frac{x}{\ln x}$ as x gets larger.

Example (1024-bit primes)

There are about 2^{1013} 1024-bit primes.

$$\left(\frac{2^{1024}}{\ln 2^{1024}}\right) - \left(\frac{2^{1023}}{\ln 2^{1023}}\right)$$

Theorem (Prime number theorem)

The number of prime numbers less than x approaches $\frac{x}{\ln x}$ as x gets larger.

Example (1024-bit primes)

There are about 2^{1013} 1024-bit primes.

$$\left(\frac{2^{1024}}{\ln 2^{1024}}\right) - \left(\frac{2^{1023}}{\ln 2^{1023}}\right)$$

Hint for those doing that at home: $\ln x^m = m \ln x$. So the only logarithm you need to take is $\ln 2$.

3072 bits may be enough

Don’t forget about 3072-bit keys. The tools we use support it, and it is a good combination security and manageability.

Resources

References I

[Aum17] Jean-Philippe Aumasson. ***Serious cryptography: a practical introduction to modern encryption.*** No Starch Press, 2017.

[Pik87] Rob Pike. ***Notes on Programming in C.*** Sept. 1987. URL: https://doc.cat-v.org/bell_labs/pikestyle.