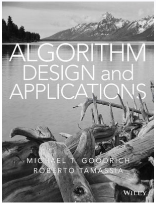
# COMP26120: Algorithms and Imperative Programming
## Introducing Complexity

Ian Pratt-Hartmann

Room KB2.38: email: ipratt@cs.man.ac.uk

2016–17

- You need this book: 


- Make sure you use the up-to-date edition. It is available on the course materials page:

  `http://studentnet.cs.manchester.ac.uk/ugt/2016/COMP26120/syllabus/`

- Read Ch. 1 (pp. 1–50).
- Pay particular attention to:
  - Pseudocode
  - Big-O notation and its relatives
  - The mathematical basics.
- Also read pp. pp. 689–690 and 695–696.

# Outline

Getting started: two ways of computing variance

Big-O notation

Some details: What is an operation, and how big is a number?

Example: Euclid's algorithm for finding highest common factors

Example: powers in modular arithmetic

- Let us begin with a simple example.
- Suppose we have a collection of numbers $x_1, \ldots, x_n$, and want to compute the *variance*, defined by the formula:

$$\sigma^2 = \frac{1}{n^2} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (x_i - x_j)^2 = \frac{1}{2n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} (x_i - x_j)^2.$$

- We could just do it:

$$
\begin{array}{l}
\texttt{var1}(x_1, \ldots, x_n) \\
\quad s := 0 \\
\quad \texttt{for } i \texttt{ from } 1 \texttt{ to } n-1 \\
\quad\quad \texttt{for } j \texttt{ from } i+1 \texttt{ to } n \\
\quad\quad\quad s := s + (x_i - x_j)^2 \\
\quad \texttt{return } s/n^2 \\
\texttt{end}
\end{array}
$$

- To see why this wouldn't be a good idea, let's count how much work is done.

```
var1(x_1, ..., x_n)
    s := 0
    for i from 1 to n-1
        for j from i+1 to n
            s := s + (x_i - x_j)^2
    return s/n^2
end
```

- We do $\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{1}{2}(n-1)n$ executions of the line $s := s + (x_i - x_j)^2$ plus one final squaring and division—about $\frac{3}{2}(n-1)n + 2$ operations.

- But suppose you notice that the variance of $x_1, \ldots, x_n$ is actually the mean squared distance from the mean, $\mu$. Noting that $\mu = \sum_{i=1}^{n} x_i/n$:

$$\sigma^2 = \sum_{i=1}^{n-1} \sum_{i=i+1}^{n} (x_i - x_j)^2/n^2 = \sum_{i=1}^{n} (x_i - \mu)^2/n.$$

Then the following algorithm will then work:

var2($x_1, \ldots, x_n$)
  $m := 0$
  for $i$ from 1 to $n$
    $m := m + x_i$
  $m := m/n$ % $m$ now holds the mean
  $s := 0$
  for $i$ from 1 to $n$
    $s := s + (x_i - m)^2$
  return $s/n$
end

- Now let's see how much work was done again:

```
var2(x_1, ..., x_n)
    m := 0
    for i from 1 to n
        m := m + x_i
    m := m/n                 % m now holds the mean
    s := 0
    for i from 1 to n
        s := s + (x_i − m)^2
    return s/n
end
```

- Here we do $n$ additions in the first loop, and $n$ subtractions, squarings and additions in the second loop, plus one division after each loop, making $4n + 2$ operations, much less (for large $n$) than $\frac{3}{2}(n - 1)n + 2$.

- Observe
  - Algorithms are given in pseudocode.
  - The correctness of the algorithm `var2` needs to be established. Specifically, we have to prove that

$$\frac{1}{n^2}\sum_{i=1}^{n-1}\sum_{i=i+1}^{n}(x_i - x_j)^2 = \frac{1}{n}\sum_{i=1}^{n}\left(x_i - \frac{1}{n}\sum_{i=1}^{n}x_i\right)^2 .$$

  - We could quibble endlessly about *exactly* how many operations are involved in these algorithms, but we'd rather not ...
  - Such quibbles are irrelevant, because `var2` is clearly superior to `var1`.
- This lecture is about how to articulate these ideas.

# Outline

Getting started: two ways of computing variance

Big-O notation

Some details: What is an operation, and how big is a number?

Example: Euclid's algorithm for finding highest common factors

Example: powers in modular arithmetic

- When comparing growth-rates of functions, it is often useful to ignore
  - small values
  - linear factors
- That is, we are interested in how functions behave in the long run, and up to a linear factor.
- This is essentially to enable us to abstract away from relatively trivial implementation details.

- The main device used for this is big-O notation. If $f : \mathbb{N} \to \mathbb{N}$ is a function, then $O(f)$ denotes the set of functions:

  $$\{g : \mathbb{N} \to \mathbb{N} \mid \exists n_0 \in \mathbb{N} \text{ and } c \in \mathbb{R}^+ \text{ s.t. } \forall n > n_0,\ g(n) \leq c \cdot f(n)\}.$$

- Thus, $O(f)$ denotes a *set* of functions.

- To see why this is useful, consider the sets of functions

$$O(n) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists n_0 \in \mathbb{N},\ c \in \mathbb{R}^+ \text{ s.t. } \forall n > n_0,\ g(n) \leq cn\}$$
$$O(n^2) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists n_0 \in \mathbb{N},\ c \in \mathbb{R}^+ \text{ s.t. } \forall n > n_0,\ g(n) \leq cn^2\}.$$

- The following should now be obvious:
  - The function $g_2(n) = 4n + 2$ is in $O(n)$.
  - The function $g_1(n) = \frac{3}{2}(n-1)n + 2$ is in $O(n^2)$.
  - The function $g_1(n)$ is not in $O(n)$.
- Notice, of course, that $O(n) \subsetneq O(n^2)$.

- So now we can express succinctly the difference between running times of our algorithms `var1` and `var2`:
  - The running time of `var1` is in $O(n^2)$ (but not in $O(n)$);
  - The running time of `var2` is in $O(n)$.
- Often, we forget that $O(f)$ is technically a set of functions, and say:
  - The running time of `var1` is $O(n^2)$ (or: is order $n^2$);
  - The running time of `var2` is $O(n)$ (or: is order $n$).

  But this is really just a manner of speaking.

- Of course, you can have $O(f)$ for any $f : \mathbb{N} \to \mathbb{N}$:
  - $O(\log n)$
  - $O(\log^2 n)$
  - $O(\sqrt{n})$
  - $O(n)$, $O(n^2)$, $O(n^3)$, $\ldots$
  - $O(2^n)$, $O(2^{n^2})$, $\ldots$
  - $O(2^{2^n})$, $O(2^{2^{2^n}})$, $\ldots$

- Sometimes you will see other asymptotic measures:
- If $f : \mathbb{N} \to \mathbb{N}$ is a function, then $\Omega(f)$ denotes the set of functions:

$$\{g : \mathbb{N} \to \mathbb{N} \mid \exists n_0 \in \mathbb{N} \text{ and } c \in \mathbb{R}^+ \text{ s.t. } \forall n > n_0,\ g(n) \geq c \cdot f(n)\}.$$

- Thus, $g \in \Omega(f)$ states that, asymptotically, $g$ grows as fast as $f$.
- $f \in \Omega(g)$ if and only if $g \in O(f)$.
- People also sometimes write

$$\Theta(f) = O(f) \cap \Omega(f).$$

- To say that $f \in \Theta(g)$ is to say that asymptotically, $f$ and $g$ grow as fast as each other.

- To think about:
  - Make sure you understand why $f(n) \leq g(n)$ for all $n$ implies $O(f) \subseteq O(g)$.
  - Why do you not hear people talking about $O(6n+7)$?
  - Give a succinct but accurate characterization of $O(1)$ in plain English.

# Outline

Getting started: two ways of computing variance

Big-O notation

Some details: What is an operation, and how big is a number?

Example: Euclid's algorithm for finding highest common factors

Example: powers in modular arithmetic

- We said that the time-complexity of `var2` is in $O(n)$, but what, exactly, does this mean?
- Answer: to say that an algorithm $A$ runs in time $g$ means the following.

  *Given an input of size $n$, the number of operations executed by $A$ is bounded above by $g(n)$.*

- This raises two important issues:
  - What is an operation?
  - How do we measure the size of the input?

- Deciding what to count as an operation is a bit of a black art. It depends on what you want your analysis for.
- For most practical applications, it is okay to take the following as operations:
    - arithmetic operations (e.g. +, ∗, /, %) on all the basic number types)
    - assignments (e.g. `a:= b`, `a[i] = t`, `t = a[i]`)
    - basic tests (e.g. `a = b`, `a` $\geq$ `b`
    - Boolean operations (e.g. `&`, `!`, `||`).
- Things like allocating memory, managing loops are often ignored—again, this may depend on the application.

- Note that, for some applications, this accounting régime might be misleading.
- Imagine, for example, an cryptographic algorithm requiring to perform arithmetic on numbers hundreds of digits long.
- In this case, we would probably want to count the number of logical operations involved.
- For example, to multiply numbers with $p$ bits and $q$ bits, we require in general about $pq$ logical operations.
- There is a formal model of computation, the Turing Machine, which specifies precisely what counts as a basic operation.
- But in this course, we shall not use the Turing machine model.

- The question of how to measure the size of the input is rather trickier.
- Officially, the input to an algorithm is a string.
- Often, that string represents a number, or a sequence of numbers, but it is still a string.
- What is the size of the following inputs?
  - The cat sat on the mat
  - 1
  - 13
  - 445
  - 65535
- The size of a positive integer $n$ (in canonical decimal representation) is $\lfloor \log_{10} n \rfloor + 1$, not: $n$.

# Outline

Getting started: two ways of computing variance

Big-O notation

Some details: What is an operation, and how big is a number?

Example: Euclid's algorithm for finding highest common factors

Example: powers in modular arithmetic

- Suppose you want to compute the highest common factor (hcf) of two non-negative integers $a$ and $b$.
- Note that the hcf is sometimes called the *greatest common divisor* (*gcd*).
- Assume $a \geq b$. A little thought shows that, letting $r = a \bmod b$, we have, for some $q$

$$\begin{aligned} a &= qb + r \\ r &= qb - a \end{aligned}$$

so that the common factors of $a$ and $b$ are the same as the common factors of $b$ and $r$. Hence:

$$\mathrm{hcf}(a, b) = \mathrm{hcf}(b, r).$$

- This gives us the following very elegant algorithm for computing highest common factors.

```
hcf(a, b)          (Assume 0 ≠ a ≥ b)
    if b = 0
        return a
    else
        r := a mod b
        return hcf(b, r)
end
```

- This is so simple, it hurts.

- How long does `hcf(a, b)` take to run?

  Well, let $a_1, a_2, \ldots, a_\ell$ be the first arguments of successive calls to `hcf` in the computation of `hcf(a, b)`. (Thus, $a = a_1$.)

  Certainly $a_1 > a_2 > \cdots > a_\ell$, so the algorithm definitely terminates.

  Assuming $\ell > 2$, consider $h$ in the range $1 \leq h \leq \ell - 2$. If $a_{h+1} \leq a_h/2$, then $a_{h+2} < a_h/2$. On the other hand, if $a_{h+1} > a_h/2$, then $a_{h+2} = a_h \mod a_{h+1} < a_h/2$.

  Either way, $a_{h+2} < a_h/2$. So the number of iterations is at most $\max(2, 2\lceil \log_2 a \rceil)$. That is, the algorithm is linear in the size of the input. (Actually, the algorithm performs slightly better than this.)

- This algorithm has an important place in computing history!

No. 4117 September 25, 1948 NATURE 487

colours were just distinguishable. The three colours 2.5G5/8, 2.5BG5/6 and 5B5/6, similarly mounted, appeared identical.

A second observer, completely unacquainted with the Munsell colour system, obtained results similar to those shown in the figure, though with more scatter. Apparatus is being constructed to check these findings by methods less open to objection on psychophysical grounds; but of the general picture there seems no doubt.

W. E. KNOWLES MIDDLETON

National Research Council,
Ottawa. April 13.

[1] Hartridge, H., *Phil. Trans.*, B, 592, 519 (1947).
[2] König, A., "Physiologische Optik", 396 (Leipzig, 1903), quoted in Wright, W. D., "Researches on Normal and Defective Colour Vision", 333 (London, 1946).

### Electronic Digital Computers

A SMALL electronic digital computing machine has been operating successfully for some weeks in the Royal Society Computing Machine Laboratory, which is at present housed in the Electrical Engineering Department of the University of Manchester. The machine is purely experimental, and is on too small a scale to be of mathematical value. It was built primarily to test the soundness of the storage principle employed and to permit experience to be gained with this type of machine before embarking on the design of a full-size machine. However, apart from its small size, the machine is, in principle, 'universal' in the sense that it can be used to solve any problem that can be reduced to a programme of elementary instructions; the programme can be changed without any mechanical or electro-mechanical circuit changes.

The essential parts of such a machine are: (1) a store for information and orders; (2) various arithmetical organs (for example, adders, multipliers); (3) a control unit[1].

The present machine contains the minimum set of facilities for a universal machine, namely: (*a*) If $x$ is any number in the store, $-x$ can be written into a central 'accumulator' $A$; or $x$ can be subtracted from what is in $A$. (*b*) The number $A$ can be written in an assigned address in the store. (By means of (*a*) and (*b*) addition or direct writing into $A$ can be programmed.) (*c*) The content of $A$ can be tested for whether $x \geqslant 0$, or $x < 0$; if $x < 0$ the order standing next in the store is passed over. This gives the essential power of branching routines. (*d*) Control can be shifted to an assigned order in the table. (*e*) The machine can be ordered to stop.

The capacity of the store is at present only 32 'words', each of 31 binary digits, to hold instructions, data and working. Hence only simple arithmetic routines devised to test the machine can be run. Examples of problems that have been carried out are: (1) Long division by the standard process. (For $(2^{30}-1)/31$, this took $1\frac{1}{2}$ seconds, the quotient being given to 39 significant binary figures of which the 13 least significant, to the left of the binary point, were zero, since 31 is a factor of $2^{30}-1$.) (2) H.C.F. by the standard process. (For 314,159,265 and 271,828,183, which are co-prime, approximately 0·5 second.) (3) Factorizing an integer. For (3) the method was deliberately chosen to give a long run the result of which could be easily checked. Thus the highest proper factor of $2^{18}$ was found by trying in a single routine every integer from $2^{18}-1$ downward, the necessary divisions being done not by long division, but by the primitive process of repeated subtraction of the divisor. Thus about 130,000 numbers were tested, involving some 3·5 million operations. The correct answer was obtained in a 52-minute run. The instruction table in the machine contained 17 entries.

It will, of course, be understood that it is intended to have other arithmetical facilities, as well as a much larger store, in a full-sized machine; and that even on the present machine quicker routines could have been used. At present routines are chosen with the sole object of testing the machine as thoroughly as possible.

The development of this machine has been very actively supported by the Telecommunications Research Establishment, Great Malvern.

F. C. WILLIAMS
T. KILBURN

Electrical Engineering Laboratories,
University, Manchester 13.
Aug. 3.

[1] For a fuller discussion see, for example, the Royal Society discussion on computing machines, summary in *Nature*, 161, 712 (1948).

### Excitation Probability Functions of Atomic and Molecular Energy-Levels

IN the correlation of the electrical and spectroscopic properties of a glow discharge in a molecular or atomic gas, the need for a rough quantitative analysis of the kinetics of the excitation of the gas soon becomes apparent. In all but the simplest cases, no more than a rough analysis can be attempted with the present scanty experimental and theoretical knowledge of the interaction of slow electrons with atoms and molecules. It is the purpose of this communication to present an empirical form for the excitation probability functions of the atomic and molecular levels which seems to fit the few experimental curves fairly closely, and at least provides a basis for calculation where previously only generalized discussion was possible.

In principle, the excitation probability functions of atomic or molecular levels may be calculated from quantum mechanics. However, in actual fact, very few atoms and molecules have been considered[1]. In any event, serious theoretical difficulties exist in calculating the excitation probabilities for collisions with slow electrons—the interesting case from the point of view of glow discharge—where the Born approximation becomes invalid. Apart from these difficulties, it is necessary that the wave functions of the ground and excited states should be known. The wave functions of very few of the energy states of atoms and molecules have, in fact, been calculated. The quantum-mechanical method does not express the excitation probability function in a form which is readily applicable to discharge problems. Blackett[2] made an interesting semi-classical approach to the subject using the principles of conservation of energy and momentum. This, however, was limited and exploratory.

The shapes of excitation functions observed experimentally for singlet and triplet atomic levels from a singlet ground state are different[3]. The singlet excitation function rises from zero at the energy of the level concerned to a broad maximum somewhat above it, while the triplet excitation function rises from zero at the energy of the level concerned to a

# Outline

Getting started: two ways of computing variance

Big-O notation

Some details: What is an operation, and how big is a number?

Example: Euclid's algorithm for finding highest common factors

Example: powers in modular arithmetic

- Recall the definition of $m \mod k$, for $k$ an integer greater than 1:

$$
\begin{aligned}
17 \mod 6 &= 5 \\
14 \mod 2 &= 0 \\
117 \mod 10 &= 7
\end{aligned}
$$

- When performing arithmetic mod $k$, we can stay within the numbers $0, \ldots, k-1$:

$$
\begin{aligned}
17 + 5564 \mod 10 &= 7 + 4 \mod 10 = 11 \mod 10 = 1 \\
17 \cdot 5564 \mod 10 &= 7 \cdot 4 \mod 10 = 28 \mod 10 = 8 \\
5564^{17} \mod 10 &= 4^{17} \mod 1 = 17179869184 \mod 10 = 4
\end{aligned}
$$

- Modular arithmetic is important in cryptography.

- Recall the definition of $m \mod k$, for $k$ an integer greater than 1:

$$
\begin{aligned}
17 &\mod 6 = 5\\
14 &\mod 2 = 0\\
117 &\mod 10 = 7
\end{aligned}
$$

- When performing arithmetic mod $k$, we can stay within the numbers $0, \ldots, k-1$:

$$
\begin{aligned}
&17 + 5564 \mod 10 = 7 + 4 \mod 10 = 11 \mod 10 = 1\\
&17 \cdot 5564 \mod 10 = 7 \cdot 4 \mod 10 = 28 \mod 10 = 8\\
&5564^{17} \mod 10 = 4^{17} \mod 1 = 17179869184 \mod 10 = 4
\end{aligned}
$$

- Modular arithmetic is important in cryptography.

- Modular arithmetic is particularly nice when the modulus is a prime number, $p$.
- If $1 \leq a < p$, then there is a unique number $b$ such that

$$a \cdot b = b \cdot a = 1 \mod p.$$

  In that case we call $b$ the inverse of $a$ (modulo $p$) and write $b = a^{-1}$.
- For example,

$$3 \cdot 5 = 5 \cdot 3 = 1 \mod 7,$$

  so 3 and 5 are inverses modulo 7.

- Here is an algorithm to compute $a^b \mod k$.
  (Note that we may as well assume that $a < k$.)

```
pow1(a, b, k)
    s := 1
    for i from 1 to b
        s := s · a mod k
    return s
end
```

- The number of operations performed here is clearly $O(b)$.
- Therefore the time complexity is . . . .

- Here is an algorithm to compute $a^b \mod k$.
  (Note that we may as well assume that $a < k$.)

$$
\begin{array}{l}
\texttt{pow1}(a, b, k) \\
\qquad s := 1 \\
\qquad \texttt{for } i \texttt{ from 1 to } b \\
\qquad\qquad s := s \cdot a \mod k \\
\qquad \texttt{return } s \\
\texttt{end}
\end{array}
$$

- The number of operations performed here is clearly $O(b)$.
- Therefore the time complexity is $O(2^n)$—i.e. exponential.

- Here is an algorithm to compute $a^b \mod k$.
  (Note that we may as well assume that $a < k$.)

```
pow1(a, b, k)
    s := 1
    for i from 1 to b
        s := s · a mod k
    return s
end
```

- The number of operations performed here is clearly $O(b)$.
- Therefore the time complexity is $O(2^n)$—i.e. exponential.
- That's right, exponential, not linear: the size of the input $b$ is $\log b$. (Note that $a$ and $k$ don't really matter here.)
- Reminder: $2^{\log_2 n} = n$.

- Here is a better algorithm to compute $a^b \mod k$.

```
pow2(a, b, k)
    d := a, e := b, s := 1
    until e = 0
        if e is odd
            s := s · d  mod k
        d := d^2  mod k
        e := ⌊e/2⌋
    return s
end
```

- The number of operations performed here is proportional to the number of times $d = b$ can be halved before reaching 0, i.e. at most $\lceil \log_2 b \rceil$. Thus, this algorithm has running time in $O(n)$, i.e. linear in the size $n$ of the input $b$. (Again, $a$ and $k$ don't really matter here.)

- To see how this number works, think of $b$ in terms of its binary representation $b = b_{n-1}, \ldots, b_0$, i.e.

$$b = \sum_{h=0}^{n-1} b_h \cdot 2^h,$$

so that

$$a^b = \prod_{h=0}^{n-1} a^{(b_h \cdot 2^h)}$$

And of course

$$a^{(b_h \cdot 2^h)} = \begin{cases} 1 & \text{if } b_h = 0 \\ a^{(2^h)} & \text{if } b_h = 1. \end{cases}$$

But the variable $e$ holds $a^{2^h}$ on entry to the $h$th iteration on the loop (counting from $h = 0$ to $h = n - 1$).

- Compute $7^{11} \mod 10$. (N.B. $7^{11}$ is actually 1977326743.)

```
pow2(a, b, k)
    d := a, e := b, s := 1
    until e = 0
        if e is odd
            s := s · d mod k
        d := d^2 mod k
        e := ⌊e/2⌋
    return s
end
```

Before loop:

$s \leftarrow 1$

$d \leftarrow 7$

$e \leftarrow 11$ (= Binary 1011)

Round 1:

$s \leftarrow 7$ ($e$ is odd and $1 \cdot 7 = 7 \mod 10$)

$d \leftarrow 9$ ($7^2 = 9 \mod 10$)

$e \leftarrow 5$

Round 2:

$s \leftarrow 3$ ($e$ is odd and $7 \cdot 9 = 3 \mod 10$)

$d \leftarrow 1$ ($7^4 = 9^2 = 1 \mod 10$)

$e \leftarrow 2$

Round 3:

$s \leftarrow 3$ ($e$ is even)

$d \leftarrow 1$ ($7^8 = 1^2 = 1 \mod 10$)

$e \leftarrow 1$

Round 4:

$s \leftarrow 3$ ($e$ is odd and $3 \cdot 1 = 1 \mod 10$)

$d \leftarrow 1$ ($7^{16} = 1^2 = 1 \mod 10$)

$e \leftarrow 0$

- At some point $d$ became 1. Do you see an optimization?

- Raising positive numbers to various powers modulo $k$ produces 1 more often than you think.
- This is of special interest when $k$ is some prime number, $p$.
- For example, set $k = p = 7$. In the following, all calculations are performed modulo 7.

$$
\begin{array}{llllll}
1^1 = 1 & & & & & \\
2^1 = 2 & 2^2 = 4 & 2^3 = 1 & & & \\
3^1 = 3 & 3^2 = 2 & 3^3 = 6 & 3^4 = 4 & 3^5 = 5 & 3^6 = 1 \\
4^1 = 4 & 4^2 = 2 & 4^3 = 1 & & & \\
5^1 = 5 & 5^2 = 4 & 5^3 = 6 & 5^4 = 2 & 5^5 = 3 & 5^6 = 1
\end{array}
$$

- Notice that, for $1 \leq a < p$, the smallest $k$ such that $a^k = 1 \mod p$ divides $p - 1$ (this is always true for $p$ prime).
- Hence $a^{p-1} \mod p$ is always 1 (this is Fermat's little theorem).
- However, there is always some number $a$ such that the various powers $g^i$ cover the whole of $\{1, 2, \ldots, p-1\}$ ($g$ is a primitive root modulo $p$).

- Let $p$ be a prime, and consider the equation

$$a^x = y \mod p.$$

- If $a$ is a primitive root modulo $p$, then, for every $y$ $(1 \leq y < p)$, such an $x$ $(1 \leq y < p)$ exists.
- In that case, the number $x$ is called the discrete logarithm of $y$ with base $a$, modulo $p$.
- Thus, the discrete logarithm is an inverse of exponentiation.
- We have seen that, for fixed $a$ and $p$, computing

$$y = a^x \mod p$$

for a given $x$ is very fast. However, no such fast algorithm is known for recovering $x$ from $y$.

- That is: modular exponentiation may be an example of a one-way function—easy to compute, hard to invert.

- Such one-way functions can be used for cryptography.
- Fix a prime, $p$ a primitive root $g$ modulo $p$.
- Choose a private key: $x$ $(1 \leq x < p - 1)$.
- Broadcast the public key: $(p, g, y)$, where $y = g^x$.
- Suppose someone wants to send you a message $M$ (assume $M$ is an integer $1 \leq M < p$).
- He picks $k$ relatively prime to $p - 1$, sets

$$a \leftarrow g^k \mod p$$
$$b \rightarrow My^k \mod p$$

  and sends the ciphertext $C = (a, b)$.
- To decode $C = (a, b)$, you set

$$M' \leftarrow b/(a^x) \mod p.$$

- To see that you get the proper message:

$$\begin{aligned} M' = b/(a^x) \mod p &= My^k(a^x)^{-1} \mod p \\ &= M(g^{xk})(g^{xk})^{-1} \mod p \\ &= M \end{aligned}$$

- To see that this is secure, notice that to encode, one needs the public key $y = g^x$, but to decode, one needs the private key $x$ (which only you have).
- In other words, to break the code, an enemy needs to be able to find the discrete logarithm of $y$ to the base $g$, modulo $p$.
- The existence of one-way functions is equivalent to the conjecture $P \neq NP$.