ALGORITHMS THAT CHANGED THE FUTURE

JOHN MACCORMICK

PRESENTATION BY TONY HÜRLIMANN

ALORITHMICS COURSE 2016
INTRODUCTION

• Great Ideas
  • 1930: British genius proves that certain problems cannot be solved by any computer to be built in the future, no matter how fast, powerful, or cleverly designed they are.
  • 1948, a scientist founds the field of information theory. His work will allow computers to transmit a message with perfect accuracy even when most of the data is corrupted by interference.
  • 1956, a group of academics founded the field of artificial intelligence. After many spectacular successes and numerous great disappointments, we are still waiting for a truly intelligent computer program to emerge.
  • 1969, a researcher at IBM discovers an elegant new way to structure the information in a database.
INTRODUCTION

• In 1974, researchers in the British government's lab for secret communications discover a way for computers to communicate securely even when another computer can observe everything that passes between them.
• 1996, two Ph.D. students at Stanford University decide to collaborate on building a web search engine. A few years later, they have created Google, the first digital giant of the Internet era.
ALGORITHM

• An algorithm is a precise recipe that specifies the exact sequence of steps required to solve a problem.

• Note the almost mechanical feel of the algorithm's steps. This is, in fact, one of the key features of an algorithm: each of the steps must be absolutely precise, requiring no human intuition or guesswork.
ALGORITHMS THAT CHANGE THE WORLD

- Web Search Engines: Matching and Ranking Algorithms
- Error Correction Code / Data Compression
- Public Key Cryptography / Digital Signature
- Pattern Recognition: Deep Learning
- Databases: The Quest of Consistency
- What is computable: the limits of algorithms
Google searches in billions of pages. How do they do this in a fraction of seconds?

1000s of Servers, high speed net, etc. but the most important factors are their clever algorithms for

- **Matching**: finding the appropriate pages
- **Ranking**: order them from the most important to less important

Google's beginnings in 1998 as the Ph.D. project of two graduate students Larry Page and Sergey Brin at Stanford University is impressive. Less than 10 years later, their company had become the greatest digital giant to rise in the internet age.
SEARCH ENGINE: TRY GOOGLE

- Amazing how Google finds **relevant** pages:
  - Try: «London bus timetable»
  - Try also: «Lamm am Kreuzgrill», «console parameters of libreoffice», «add file types in preview pane in windows», «software for drawing trees»,

Note: Many queries on real search engines have hundreds, thousands, or even millions of hits. And the users of search engines generally prefer to look through only a handful of results,
SEARCH ENGINE: ALTAVISTA

- Altavista launched 1995: technological breakthrough with an old concept: **indexing**
- Same on the web: each «page» gets a number:

1. Make a list of all words that appear
2. Run through the pages word by word and note the page number

**Sidenote:** Important tasks for indexing:
- We will learn:
  1. How to find a list of all different words
  2. Sort this list
  3. Search in the list efficiently
SEARCH ENGINE: ALTAVISTA

- Query **cat**: search in the list, return all pages with that index
- Query **cat sat**: search both words, return the intersection
- Query **«cat sat»**: a phrase query, difficult to do: we need the **word-location trick**: for each occurrence on a page note also where on the page the word appears:
SEARCH ENGINE: ALTAVISTA

• The word-location trick can also be used for near-word locations. Query: **cat NEAR dog** (was possible in Altavista) [is also used internally all the time by Google for ranging pages]

• Suppose you have two pages: which one is more relevant for the cause of malaria? In the first page «cause» and «malaria» are close together!

<table>
<thead>
<tr>
<th>1</th>
<th>By far the most common cause of malaria is being bitten by an infected mosquito, but there are also other ways to contract the disease.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Our cause was not helped by the poor health of the troops, many of whom were suffering from malaria and other tropical diseases.</td>
</tr>
</tbody>
</table>
The metaword trick was a true invention by Altavista. Altavista became the leader. All pages have a structure with title, header, body etc. in HTML we have tags: `<h1>`, `<title>`, `<body>`.

- Query «dog»
The word-location trick and especially the metaword trick did help AltaVista—where other had failed—in finding matches to the entire web.

We know this because the metaword trick is described in a 1999 U.S. patent filing by AltaVista, entitled “Constrained Searching of an Index.”

However: Matching is only half the work, AltaVista was eclipse by Google’s new type of ranging algorithm.
SEARCH ENGINE: GOOGLE

• The PageRank algorithm launched Google.
• Two students at Stanford, Larry Page and Sergey Brin, moved the operation into the now-famous Menlo Park garage. They must have been doing something right, because only three months after its legal incorporation as a company, Google was named by PC Magazine as one of the top 100 websites for 1998.
• PC Magazine: “uncanny knack for returning extremely relevant results.”
SEARCH ENGINE: GOOGLE

• So what is the PageRank algorithm?
• It's an algorithm that ranks web pages, but it's also the ranking algorithm of Larry Page, its chief inventor. Page and Brin published the algorithm in 1998, in an academic conference paper, “The Anatomy of a Large-scale Hypertextual Web Search Engine.”
• The paper is in fact a complete description of the Google system as it existed in 1998.
SEARCH ENGINE: GOOGLE

- *The hyperlink trick*: A hyperlink is a link on a page that points to another page, (a old idea 1945 by enggineer Vannevar Bush)
- First idea: The more links point to a page the more relevant it is!
- Query: «scrambled egg recipes». 
SEARCH ENGINE: GOOGLE

- Problems with the *hyperlink trick*:
  - Link to «bad» rather than good pages: «Erni’s recipe is awful»
  - All incoming links are treated equally
- *The authority trick*: A link coming from an “expert” should count more than from a novice. Suppose Alice Water is an expert in scramble egg recipes.
- Bert’s link is ranked higher.
SEARCH ENGINE: GOOGLE

• How can a computer determine that Alice Water is a greater authority on scrambled eggs?

• Combine the hyperlink trick with the authority trick!
  • All pages start with a authority score of 1
  • Calculate the authority by adding up all incoming links
SEARCH ENGINE: GOOGLE

- Upps! There are cycles!
- This can be resolved by the random surfer trick.
- Start with a page, click on a link randomly, continue this way or start a new page with a low probability of 15%.
- Start with A go till B, start with C etc.

The percentages calculated from random surfer simulations turn out to be exactly what we need to measure a page's authority. So let's define the surfer authority score of a web page to be the percentage of time that a random surfer would spend visiting that page.
SEARCH ENGINE: GOOGLE

- The percentages calculated from random surfer simulations turn out to be exactly what we need to measure a page's authority.
SEARCH ENGINE: GOOGLE

- Surfer authority scores for the scrambled egg example
Well, there are many complicating factors. One of them: The hyperlink trick can be abused—this strikes in the heart of the algorithm: artificially inflate the ranking of a page: create 10’000 pages on the fly that point to my page (web spam).

Of course, modern engines use many other criteria: Google's own website states that “more than 200 signals” are used in assessing the importance of a page.

Nevertheless, RankPage is still an important part and helped to dethrone AltaVista.
For a computer, being accurate 99.9999% of the time is not even close to good enough. Computers must be able to store and transmit literally billions of pieces of information without making a single mistake.

But computers have to deal with communication problems just like other devices. They don’t transmit information perfectly, because they often suffer from distortions, static, or other types of noise.

Electrical wires are subject to all sorts of fluctuations; wireless communications suffer interference all the time; and physical media such as hard disks, CDs, and DVDs can be scratched, damaged, or simply misread because of dust or other physical interference.

How on earth can we hope to achieve an error rate of less than one in many billions, in the face of such obvious communication errors?
The Repetition Trick: repeat the transmission. Pick the value that occurs most often. The probability of a false guess can be made arbitrarily small.

Example: bank account $5213.75 with error probability 20%.

The moral of the story is that by repeating an unreliable message often enough, you can make it as reliable as you want.

The repetition trick is much more vulnerable for malicious attack.
The Repetition Trick is not good enough for modern computer systems:
  • Transmitting a 200MB Software 1000 times would be impractical
  • Malicious attacks are still possible.

The Redundancy Trick: Basic principle: send something more than the bare bone message (the repetition trick sends multiple copies).

There are many way of sending “something more” – that is – redundancy – making the message somewhat longer.
ERROR CORRECTION CODE

• **The Redundancy Trick**: Each “character is encoded as a pattern.”1” becomes “one”, “2” becomes two, “5” becomes “five”.

• If “fique” is received we know that this must be “5”.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Decoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 → one</td>
<td>five → 5 (exact match)</td>
</tr>
<tr>
<td>2 → two</td>
<td>fique → 5 (closest match)</td>
</tr>
<tr>
<td>3 → three</td>
<td>twe → 2 (closest match)</td>
</tr>
<tr>
<td>4 → four</td>
<td>five → 5 (exact match)</td>
</tr>
<tr>
<td>5 → five</td>
<td>five → 5 (exact match)</td>
</tr>
</tbody>
</table>
ERROR CORRECTION CODE

• Mathematicians have worked out fancier codewords than the English-language ones we were using as an example, but otherwise the workings of reliable computer communication are the same.

• Example the (7,4) Hamming code, the one discovered by Richard Hamming at Bell Labs in 1947, in response to the weekend computer crashes.

• In storing we have the same problem as in transmitting!
The Checksum Trick: We saw methods for simultaneously detect and correct errors. In many applications it is enough to detect an error (example: TCP packets).

Simplest type: message is: 4 6 7 5 6. Add all digit and take modulo 10: 4+6+7+5+6=28 , modulo 10 is 8, add 8 to the message.

To good to be true: can only detect one error, not two.

However, if there is only one error, a simple checksum is absolutely guaranteed to detect it.
ERROR CORRECTION CODE

• Simplest checksum: just sum up, another is the staircase checksum: sum the digits multiplied by the position:
  \[4 \times 1 + 6 \times 2 + 7 \times 3 + 5 \times 4 + 6 \times 5 = 87,\] modulo 10 gives 7.

• if you include both the simple and staircase checksums, then you are guaranteed to detect any two errors in any message.

• Sometimes as many as 150 different checksums are used. The overhead is still small.

• A cryptographic hash function is even better and protects also from malicious changes.

<table>
<thead>
<tr>
<th>original message</th>
<th>4</th>
<th>6</th>
<th>7</th>
<th>5</th>
<th>6</th>
<th>8</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>message with one error</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>message with two errors</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>message with two (different) errors</td>
<td>2</td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>message with two (again different) errors</td>
<td>6</td>
<td>5</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>7</td>
</tr>
</tbody>
</table>
ERROR CORRECTION CODE

• **The Pinpoint Trick:** Detect and correct an error, another method.

• Translate the message all in numbers, break it into chunks of 16 digits, and rearrange them in a square, so 4837543622563997 gives 483725436822565399784306.

• Then calculate row and columns checksum.

• **Send now:** 483725436822565399784306.

• On reception calculate the checksums and the error can be identified and corrected.

• Then extract the correct 16 digits.

• The pinpoint trick is named “two-dimensional parity”.

\[
\begin{array}{cccc}
4 & 8 & 3 & 7 \\
5 & 4 & 3 & 6 \\
2 & 2 & 5 & 6 \\
3 & 9 & 9 & 7 \\
\end{array}
\]
Error correction was needed and developed before computers: Bell Telephone

However, the two Heroes are Hamming and Shannon, who proved that it was possible, in principle, to achieve surprisingly high rates of error-free communication over a noisy, error-prone link.

Hamming code was used in earliest computer; another is Reed-Solomon code

Staircase checksum and 2-dim parity are used in CDs, DVDs and others.

Ethernet (TCP) uses checksum for each packet.

Popular crypto-checksum are MD5 and SHA-1

New low density parity-check codes are used in high-definition satellite TV and deep space communication.
DATA COMPRESSION

• For travel you **compress** clothes into a suitcase, after travel you **decompress** them and (hopefully) in their original size and shape.

• The same can be done with information. Compressing files is often done behind the scene (voice over the telephone, more obviously: download a ZIP-file p.e., videos etc.)

• Compression can be **lossless** (the ultimate free lunch) or **lossy**.

• Lossless: Your calendar has 40 cells (8*5). For an appointment you do not say: “Monday 1. hour full, 2. hour full,…” you say: “Monday and Tuesday full”

• The message AAAAAAAAAAAAAAAAAAAABCBCBCBABCBCBABCBCBCBCAAADEFDEFDEF can be lossless be “compressed” into 21A, 10BC, 6A, 3DEF

• This **trick** is called **run-length encoding**. (Only useful for very special data)
Many other compression tricks have been invented, two of them are the **same-as-earlier trick** and the **shorter-symbol trick**.

These two tricks are the only used in the popular ZIP file format.

**The same-as-earlier Trick:** suppose the message (dashes are not part of the message): VJGDNQMYLH-KW-VJGDNQMYLH-ADXSGF-O- VJGDNQMYLH-ADXSGF-VJGDNQMYLH-EW-ADXSGF

Compressed it is (“bx” means “go back x positions”, “cy” means “copy y positions”): VJGDNQMYLH-KW-b12c10-ADXSGF-O-b17c16-b16c10-EW-b18c6 (44 against 63 characters)

What is : FG-b2c14 ? It is : FG-FG-FG-FG-FG-FG-FG-FG-FG-FG-FG

What is : Ab1c250 ? It is : 251 A!
The Shorter-Symbol Trick: Idea: each symbol is transformed into a Bit-Stream of variable length depending on its frequency in a text.

As a contrast to this idea, take the ASCII-code: each char is coded as a 1-byte, each code has the same length.

We use the shorter-symbol trick all the time: the 24-letter phrase “United States of America” is abbreviated as “USA”, how would you abbreviated the 24-letter phrase: “The sky is blue in color”? There is no need! Why? Because the first is much more frequently used then the second.

Let’s use digit instead of bits: “e” and “t” are more frequent than all other letters (in English), so let’s abbreviated them by a one digit say “8” and “9”, and all other letters by a 2-digit sequence and specials by a 3-digit sequence.
However the idea only works, if no 2-digit or 3-digit begins with “8” or “9” and all 3-digit begin with “7” and no 2-digit begins with “7”. Why?

So, how is a ZIP-file generated?

1. Use same-as-earlier trick to the original data
2. Examine the frequencies of the transformed data and establish a transformation table.
3. Apply the mapping to the transformed data and store the mapping table also in the result file.
4. The original file often is broken into chunks and each is treated separately.

ZIP-files can be compressed greatly using only these two tricks, it’s a free meal (lossless)!
Lossy compression: not a free meal but a very good deal: uncompress cannot reconstruct the original but is very similar to it.

Frequently used for images and audio data for the human eye and ear (low-quality videos)

The Leave-it-Out Trick: In a image, remove each second pixel in a row and a column, getting 25%, repeat again, generates an image of size 6%.

The trick is more sophisticated in various file compression algorithms.
DATA COMPRESSION

- JPEG divides the image into 8x8 pixels, compressing each square separately: if “almost” all pixels have the same color then the square is compressed into a single “color-pixel”.
- MP3 and AAC a popular audio compression formats. These format exploit also the facts about the human ear, certain types of sounds have little effect on human listeners and can be left out without reducing the quality of the output.
Error-correction codes and compression algorithms are two sides of the same coin. It all comes down to the notion of redundancy.

Error correction add redundancy while compression removes redundancy from the data.

In practice, compression and error correction do not cancel each other out. Good compression algorithms remove inefficient types of redundancy, while error-correction encoding adds a different, more efficient type of redundancy.

Huffman, a student of Shannon, found a method that yields the best possible compression for each individual symbol. (Shannon and Fano already found a very good and efficient way (1948), and proved that there is a better compression, but could not find it.)
Humans can communicate a secret by whispering in their friend's ear.

Computer cannot whisper, sending a credit card number through the Internet is like sending the number on a postcard: anyone coming in contact with the postcard can read and copy it.

Because any message on the Internet typically travels through numerous computers called routers, the contents of the message can be seen by anyone with access to the routers—and this includes potentially malicious eavesdroppers.
Quick fix of this postcard problem: Just use a secret code to encrypt each message before writing it on a postcard!

.Hmmm…! This works just fine if you *already* know the person to whom you’re sending the postcard. You agreed *in the past*, what secret code to use.

The real power of public key cryptography is that it allows you to employ a secret code that only the recipient can decrypt—despite the fact that you had no chance to secretly agree on what code to use.

If you buy on Amazon.com, you must transfer the credit card number at least the first time and you never contacted Amazon before!

It is like a paradox: The recipient will see exactly the same information as the postal workers, but somehow the recipient will learn how to decode the message, whereas the postal workers will not. Public Key Cryptography provides a resolution to this paradox.
PUBLIC KEY CRYPTOGRAPHY

• Let’s start with a **thought experiment**: Verbal communication in a room with three persons: You, friend Arnold and enemy Eve. Only communication: speaking loud! You want to transmit your credit card number (7) to Arnold, but Eve should not know it.

• First, try to think of some number that Arnold knows but Eve doesn't. You say: “Hey Arnold, remember the number of my family's house on Pleasant Street where we used to play as children? Well, if you take that house number, and **add** on the 1-digit credit card number I'm thinking of right now, you get 329.”

• Arnold can—if he remembers that number—**decrypt** by **subtracting** the shared secret (322), but Eve cannot, because she does not know the shared secret (322).
• Believe it or not, if you understood this simple “addition trick” of adding a shared secret to a private message like a credit card number, then you already understand how the vast majority of encryption on the internet actually works!

• A few important details need to be clarified:
  • The hard part: How to establish a “shared secret” which we will call the “key”. (“Shared secret” is cheating…but there is an ingenious way to solve this problem: the Diffie-Hellman key exchange.)
  • The key must be much longer than 3-digits, otherwise Eve can try all numbers 0-999. You heard about “128-bit encryption” (that is 38-digit number)
  • Addition/Subtraction are probably not the appropriate operations, because these symmetric operations are easy to do. We shall see why!
  • Long messages are broken into blocks, typically of 10-15 chars. And each block is transformed according to a fixed set of rules: “add the first half of the key to the second half of the block…etc.”. The most popular block cipher is Advanced Encryption Standard (AES).
PUBLIC KEY CRYPTOGRAPHY

• **The Paint-Mixing Trick:** You, Arnold and Eve, each has a large number of pots of paints.
  - Objective: You share a secret color with Arnold and Eve does not know which one.
  - Rules: (1) Each can mix various colors to create a new one. (2) Each has a hidden corner where he mixes the colors, (3) The middle of the room is public and each can carry color pots there and back. That is the unique place where communication can take place.

  **Step I:** You and Arnold both choose a private color (Y and A).
  **Step II:** Somebody announces a public color (P).
  **Step III:** You mix one pot of your private color with one pot of the public color, and Arnold also mixes his private color with the public color. Both mixture are announced publicly: both «public-private mixtures». The colors are YP and AP.
  **Step IV:** You take the mixture of Arnold (AP) back to your hidden corner and mix it with Y giving the mixture APY. Arnold does the same, taking your mixture (YP) and mix it with his private color, giving the mixture YPA. In fact, the colors APY and YPA are the same, each containing a pot of the three colors A P and Y!
• Amazing, you share a secret color APY with Arnold and Eve has no change to know it. She only knows P (the public color), and AP and YP (the two public-private mixtures).

• If she mixes the public with one of the mixture, she get APP or YPP.

• There is no way to “unmix” paint.

• All communication took place publicly and still you and Arnold “share a secret”.

• Let’s redo the trick with numbers instead with colors now!
We replace **colors** by **numbers** (here single digits to make calculation easy).

**Mixing** colors is replaced by **multiplying** numbers.

Note: mixing colors and multiplying number are commutative and associative operations (the order does not matter!)

**BUT:** This scheme is “flawed” !! There is a problem … Which one? What is unmixing … ?

**PUBLIC KEY CRYPTOGRAPHY**

**Step I**: You and Arnold both choose a private number (4 and 6).

**Step II**: Somebody announces a public color (7).

**Step III**: You multiply your private number with the public number, and Arnold also multiply his private number with the public number. Both multiplication are announced publicly: both «public-private mixtures». The numbers are 28 and 42.

**Step IV**: You take the «mixture» of Arnold (42) back to your hidden corner and multiply it with 4 giving the number 168. Arnold does the same, taking your «mixture» (28) and multiply it with his private number, giving the number 168. You and Arnold **share now a secret number** (like the house number above)!
PUBLIC KEY CRYPTOGRAPHY

• Unfortunately, Eve can find out the secret. She can divide the public “mixtures” by the public number to find out the private keys: $28/7=4$ and $42/7=6$.

• Now she can multiply the public “mixture” with the private key to obtain the “shared secret”!

• In contrast to mixing color, where unmixing colors is difficult, “unmixing” numbers is easy, the inverse operation of multiplication is division.

• What we need is a real-life math operation that is easy to do (like mixing paint) but practically impossible to undo (like unmixing paint).

• Hence we replace (multiplication,division) by called (discrete exponentiation,discrete logarithm). Discrete exponentiation is easy, but there is no easy known method to discrete logarithm.
PUBLIC KEY CRYPTOGRAPHY

- We replace multiplication by discrete exponentiation and division by discrete logarithm.
- We need modulo arithmetic. Example: $3^4 \mod 7 = 4$

**Step I:** You and Arnold both choose a private number (8 and 9).
**Step II:** Somebody announces two public color (11,2),
**Step III:** You do $2^8 \mod 11$ giving 3, and Arnold does $2^9 \mod 11$ giving 6, the two public-private «mixture».
**Step IV:** You take the «mixture» of Arnold (6) back to your hidden corner and do $6^8 \mod 11$ giving 4. Arnold does the same by $3^9 \mod 11$ also miraculously giving 4, your shared secret number!
This time, Eve cannot find out the shared secret. Of course the clock size (11) must be much, much larger, otherwise Eve could try all possible numbers by brute force to find out the right one.

She knows the public numbers (11,2) and the “mixtures” 3 and 6, but she cannot compute the shared secret in reasonable time, there is no known fast method for that operation.

This method is called the Diffie-Hellmann key exchange method and is used when you access a https:// web-site, for example.

Once the shared secret is established, the two computers can encrypt all their communication using a variant of the addition trick described earlier.
Again, large number must be chosen and these numbers must be chosen with care:

- The clock size number (11 in the example) should be a large prime number. Generating a large prime number is not a trivial task, but there are (also probabilistic) methods.
- The base (2 in the example) (the second public number) must be a primitive root of the clock size, this means that the powers of the base eventually cycle through every possible value on the clock. The table shows that 2 and 6 are primitive roots of 11, but 3 is not. (3 cycles through the values 3, 9, 5, 4, 1 and misses 2, 6, 7, 8, and 10).
Summary: For centuries, mathematicians “played” around with “useless” prime number theory. They had no other use then intellectual satisfaction. Suddenly this popped into commercial applications…

Public key cryptography is used to exchange secure information on the Internet. The two computers first establish a “shared secret key” using the described method (the house number above was 322), then the information can be encrypted using any methods, (we used the addition method $322 + 7 = 329$, 7 being the encrypted message, for example, a credit card number, to transmit).

The vast majority of the online transactions we perform every day could not be completed securely without public key cryptography.
“Digital Signature”, a paradoxical concept? Anything that is digital can be copied, on the other hand, the whole point of a “signature” is that it can be read, but cannot be copied.

Paper signatures are used to sign checks and other legal documents. Are digital signatures also used for the same kind of things: making online payments, etc.? No! Online banking and credit card payments require passwords and other means to identify the person, but no signatures.

So, what are digital signature used for? Instead of you signing material that is sent to others, it is typically others who sign material before sending it to you, for example, software to be installed on your computer.
Handwritten signatures are used to authenticate documents: “I promise to pay $100 to Francoise. Signed, Ravi”.

How can you verify that Ravi really signed this document? The answer is that you need some trusted repository of signatures, a bank or other trusted institution.

Assumptions: (1) We assume that the bank can be trusted, (2) It is not possible to forge the signature.
PUBLIC KEY CRYPTOGRAPHY
DIGITAL SIGNATURE

- **Physical Padlock Trick:** Alternative to paper signature: a physical padlock
  - We need (1) padlocks, (2) keys, and (3) locked boxes.

- Each participant receives a large numbers of (1), (2) and (3). For each participants they are all identical. Padlocks are exclusive (containing biometric sensors, for example): they can only be *locked* by their owners.

- Padlocks are there to lock boxes, key to open them.
Example: Ravi owes Francoise $100, and Francoise would like to record that fact in a verifiable way. How to proceed?

**Step I:** Ravi makes a document saying “Ravi promises to pay $100 to Francoise,” and doesn’t bother to sign it.

**Step II:** Ravi makes a copy of the document and places it in the lockbox and locks it with one of his padlock.

**Step III:** Ravi gives the lockbox to Francoise. (The lockbox is the signature.)

Francoise can now authenticate Ravi’s document at any time: If anyone tries to deny the authenticity, Francoise can say “Okay Ravi lend me one of your keys for a minute.” She opens the padlock (in presence of Ravi and other witnesses and displays the document.

Since only Ravi could have locked the padlock, the document must be authentic and produce by himself.
Problems with the physical padlock.

(1) When Ravi locks the box, Francoise is well advised to call in a trusted witness, otherwise Ravi could cheat by putting a different document into the box.

(2) More serious: Francoise needs Ravi cooperation to proof the authentication – that is, to open the box. Ravi could refuse to open the box, or he could pretend to cooperate and give her a different key, that cannot open the box.

Solutions:

We still need to resort to a trusted third party (a bank). The bank will store the keys to open the padlocks.

Whenever Francoise need to proof the authentication she takes the box to the bank and opens the box there under the eyes of witnesses.
• **Multiplicative padlocks**: padlocks, keys, and boxes (signature) are replaced by numbers.

• The operation of locking and unlocking is represented by mathematical operations, first by multiplication under modulo.
  - Examples: \( 7 \times 5 \mod 11 = 2 \), \( 5 \times 6 \mod 11 = 8 \), \( 8 \times 2 \mod 11 = 5 \)

• The document’s message is also replaced by a number, in our case as a digit— an incredible short message (to keep thing as simple as possible).
PUBLIC KEY CRYPTOGRAPHY
DIGITAL SIGNATURE

**Step I:** Ravi transforms the document’s message into a digit (5).
**Step II:** Ravi chooses a padlock key (6) and a clock size (11). He keeps the padlock key secret. We call it the **private key**.
**Step III:** Given the padlock key and the clock size, Ravi also calculates a key number to be able to unlock the “box” (the signature) later on. The key for our example is 2, because it unlock the signature by using multiplication again to extract the message.
**Step IV:** The clock size number (11) together with the key number (2) is given to the trusted authority. We call it the **public key**.
**Step V:** Ravi «locks» the padlock by multiplying the message by the padlock key modulo the clock size. This gives the signature (8) and gives it to Francoise.
**Step VI:** Francoise (or anyone) can now authentify the message by «unlocking» the signature using the public key and modulo multiplication, giving 5 (the original message).
A note on the trusted authority!

Ravi can publish the public key (the unlock key and the clock size) even on his homepage. Would this mean that we do not need a trusted authority anymore to store the public key?

No! Because without it Ravi could distribute a false key, or Ravi’s enemies could create a new padlock/key, make a website announcing that this is Ravi’s key.

The trusted authority is needed not to keep the key secret but to confirm the validity of the clock size number and the unlock key.

The only number that Ravi must keep secret is the padlock key.
PUBLIC KEY CRYPTOGRAPHY
DIGITAL SIGNATURE

• **A note on the size of the message:**
  
  • What if the message is larger than just a digit?
  
  • Of course, we must choose a clock size much, much larger than 11, say a number with 100 digits.
  
  • Then we can split the message into blocks of 100 digits and lock (sign) them separately.
  
  • However, we may also build a checksum of the message and sign the checksum, a transformation called cryptographic hash function.
  
  • (For the purpose of simplicity, we only use message of size of one digit.)
• **A note on the numerical approach:**
  
  • How does Ravi generate the “right” unlock key given the padlock and the clock size?
  
  • This is a very old mathematical problem, called the extended Euclid algorithm.
  
  • In our case, it is especially easy to find the unlock key: We need to solve:
    \[ a \times x \mod c = 1 \], that is, \[ 6 \times x \mod 11 = 1 \], and we find \( x = 2 \).
  
  • But the whole **multiplicative approach is fundamentally flawed** anyway, because by the same method anyone can generate the secret padlock key, given the public clock size and the unlock key. In our case, we only need to solve:
    \[ y \times 2 \mod 11 = 1 \], and we find \( y = 6 \).
An Exponent Padlock: Let’s upgrade our flawed multiplicative system to a new digital signature scheme, known as RSA (Ronald Rivest, Adi Shamir, and Leonard Adleman). We use exponentiation in place of the multiplication.

The system works exactly the same way as multiplicative approach. In the example we use: clock size = 22, (secret) padlock key = 3, unlock key = 7.

For the message 4 the signature 20 is calculated as: $4^3 \mod 22 = 20$. Unlocking the signature is done by: $20^7 \mod 22 = 4$. 
• As in the multicative padlocks, we use a similar procedure:

**Step I:** Ravi transforms the document’s message into a digit (4).

**Step II:** Ravi chooses a padlock key (3) and a clock size (22). He keeps the padlock key secret. We call it the **private key**.

**Step III:** Given the padlock key and the clock size, Ravi also calculates a key number to be able to unlock the “box” (the signature) later on. The key for our example is 7, because it unlock the signature by using exponentiation again to extract the message.

**Step IV:** The clock size number (11) together with the key number (2) is given to the trusted authority. We call it the **public key**.

**Step V:** Ravi «locks» the padlock by exponentiation the message by the padlock key modulo the clock size. This gives the signature (20) and gives it to Francoise.

**Step VI:** Francoise (or anyone) can now authentify the message by «unlocking» the signature using the public key and modulo exponentiation, giving 4 (the original message).

**Exponent padlocks**
Behind the exponent padlock scheme are some deep mathematical theorems had cannot be explained here. We only give the procedure to find the unlock key.

Ravi chooses a padlock key, here 3. However, Ravi chooses carefully the clock size: he generate two (large) prime numbers p and q (in our case p=2 and q=11). He keeps the two primes secret! Multiplying them give the clock size (pq=22).

Then he calculates (p-1)(q-1) which is 10 in our example. Interestingly, then he can generate the unlock key using our old method: Euclid’s extended algorithm: $3 \times x \mod 10 = 1$, it follows that the unlock key is $x = 7$.

So far, nobody can generate the padlock key given the clock size and the unlock key. The security reside on the fact that factorization of pq is difficult. There is no known fast method so far!
COMPUTABILITY

• Is there anything that computer (as we know them) cannot do? Is there any problem that cannot be translated into an algorithm?

• Of course, there are plenty of problems that computer still cannot do: driving a car, translating natural language, etc.

• Other problems are probably «hard» forever—meaning: take a long time to solve: shortest traveling salesman tour, etc.

• But: are there problems that can never be solved by computers?
The existence of uncomputable problems is striking enough on its own, but the story of their discovery is even more remarkable.

The existence of such problems was known before the first electronic computers were ever built! Independently found by Alonzo Church and Alan Turing 1936.
COMPUTABILITY:
A PROGRAM-ANALYZING PROGRAM

• A concrete problem: it is provably impossible for any software-checking tool to detect all possible crashes in all programs.

• In other words: **There does not exist a program that checks whether another programs crashes or not.**

• We shall prove this!
First: What do we mean by “provably impossible”. Proof by contradiction: “S implies T, but T is false, therefore S is false.”

It is provably impossible that a multiple of 10 ends with the digit 3.

The number of primes cannot be finite.

The three facts contradict each other. So one phrase cannot be. Why?
1. The U.S. Civil War took place in the 1860s.
2. Abraham Lincoln was president during the Civil War
3. Abraham Lincoln was born in 1520.”
COMPUTABILITY

- What does this mean: “a program analyses another program”?
- A program always slavishly follows the same sequence of well-defined instructions. Right? Wrong! Well, it depends.
- My MS-Word program behaves differently on different open docx-files, no?
  - Try on the console: “winword a.docx” (it opens the file “a.docx”)
  - Try now: “winword picture.jpg” (it opens garbage, no?)
  - Try now: “winword winword.exe” (even more garbage)
- Crazy idea! But since programs are files…
COMPUTABILITY

- Some program cannot exist! Find one!

- Some simple Yes-No programs:
  - ProgramA.exe: This simple program **always** output “Yes”.
  - ProgramB.exe: This simple program **always** output “No”.
  - SizeChecker.exe: This program takes a file as input and outputs “Yes” if the file is bigger than 10kB, otherwise it outputs “No”.
  - NameSize.exe: This program takes a file as input and outputs “Yes” if the filename is at least one character long, otherwise it outputs “No”.

- Do these programs exist? Of course, it is very easy to implement them!
COMPUTABILITY

- Make sure you understood these programs:

<table>
<thead>
<tr>
<th>program run</th>
<th>input file</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProgramA.exe</td>
<td>address-list.docx</td>
<td>yes</td>
</tr>
<tr>
<td>ProgramA.exe</td>
<td>ProgramA.exe</td>
<td>yes</td>
</tr>
<tr>
<td>ProgramB.exe</td>
<td>address-list.docx</td>
<td>no</td>
</tr>
<tr>
<td>ProgramB.exe</td>
<td>ProgramA.exe</td>
<td>no</td>
</tr>
<tr>
<td>SizeChecker.exe</td>
<td>mymovie.mpg (50MB)</td>
<td>yes</td>
</tr>
<tr>
<td>SizeChecker.exe</td>
<td>myemail.msg (3KB)</td>
<td>no</td>
</tr>
<tr>
<td>SizeChecker.exe</td>
<td>NameSize.exe (8KB)</td>
<td>no</td>
</tr>
<tr>
<td>SizeChecker.exe</td>
<td>SizeChecker.exe (12KB)</td>
<td>yes</td>
</tr>
<tr>
<td>NameSize.exe</td>
<td>mymovie.mpg</td>
<td>yes</td>
</tr>
<tr>
<td>NameSize.exe</td>
<td>ProgramA.exe</td>
<td>yes</td>
</tr>
<tr>
<td>NameSize.exe</td>
<td>NameSize.exe</td>
<td>yes</td>
</tr>
</tbody>
</table>
COMPUTABILITY

- **Freeze.exe**: a program that does one of the most annoying things a computer program can do: it “freezes” (no matter what its input is).

- **AlwaysYes.exe**: This program examines the input file it is given and outputs “yes” if the input file is itself a yes-no program that always outputs “yes.” Otherwise, the output of AlwaysYes.exe is “no.”

```
<table>
<thead>
<tr>
<th>input file</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>address-list.docx</td>
<td>no</td>
</tr>
<tr>
<td>mymovie.mpg</td>
<td>no</td>
</tr>
<tr>
<td>WINWORD.EXE</td>
<td>no</td>
</tr>
<tr>
<td>ProgramA.exe</td>
<td>yes</td>
</tr>
<tr>
<td>ProgramB.exe</td>
<td>no</td>
</tr>
<tr>
<td>NameSize.exe</td>
<td>yes</td>
</tr>
<tr>
<td>SizeChecker.exe</td>
<td>no</td>
</tr>
<tr>
<td>Freeze.exe</td>
<td>no</td>
</tr>
<tr>
<td>AlwaysYes.exe</td>
<td>no</td>
</tr>
</tbody>
</table>
```
**YesOnSelf.exe**: This is a simpler variant of AlwaysYes.exe. It outputs “yes” if the input file outputs “yes”, *when run on itself*, otherwise it outputs “no”.

Note **YesOnSelf.exe** is very smart: It analyses SizeChecker.exe, for example, and it knows that when SizeChecker is run on itself, it outputs «yes» (since the size of SizeChecker is larger than 10kB,) so YesOnSelf outputs «yes».

Such a program would be quite interesting and very powerful!

Note also: **YesOnSelf.exe** is just a simplified or particular version of **AlwaysYes.exe**.
• What would YesOnSelf.exe output when it is given YesOnSelf.exe as an input? (last line in previous table)

• Luckily, there are only two possibilities: “yes” or “no”.
  • Suppose it is “yes”: “yes,” we know that (according to the definition of YesOnSelf.exe), YesOnSelf.exe should output “yes” when run on itself. Everything is perfectly consistent, so the right answer is “yes”? 
  • But suppose it is “no”: Well, it would mean that (again, according to the definition of YesOnSelf.exe) YesOnSelf.exe should output “no” when run on itself. Again, this statement is perfectly consistent!

• It seems like YesOnSelf.exe can actually choose what its output should be. As long as it sticks to its choice, its answer will be correct. This mysterious freedom in its behavior will soon turn out to be the innocuous tip of a rather treacherous iceberg. Hmm…What now?
COMPUTABILITY

- **AntiYesOnSelf.exe**: Is just the opposite of YesOnSelf.exe, if YesOnSelf.exe outputs “yes” then AntiYesOnSelf.exe outputs “no” and vise versa.

- So the answer in the previous table to YesOnSelf.exe is just reversed.

- But what is now the output of AntiYesOnSelf.exe if applied on itself? Fortunately, the answer is either yes or no. So let’s analyse both cases.
Before proceeding let’s again define what **AntiYesOnSelf.exe** is doing:

**AntiYesOnSelf.exe** when given itself as input, answers the question:

*Will AntiYesOnSelf.exe, when run on itself, output “no”?*

- **Case I:** The output is “yes”: If the output is “yes,” then the answer to the question in bold in the box above is “no.” But the answer to the bold question is, by definition, the output of AntiYesOnSelf.exe (read the whole box again to convince yourself of this)—and therefore, the output must be “no.” Contradiction! So the answer cannot be “yes”.

- **Case II:** The output is “no”: If the output is “no,” then the answer to the question in bold in the box above is “yes.” But, just as in case I, the answer to the bold question is, by definition, the output of AntiYesOnSelf.exe—and, therefore, the output must be “yes.” In other words, we just proved that if the output is “no,” then the output is “yes.” Again contradiction! So the answer cannot be “no”.

What now?
**COMPUTABILITY**

- **AntiYesOnSelf.exe** was defined to be a yes-no program—a program that *always* terminates and produces one of the two outputs “yes” or “no.” And yet we just demonstrated a particular input for which AntiYesOnSelf.exe does not produce either of these outputs!

- This contradiction implies that our initial assumption was false:

- Thus, it is not possible to write a yes-no program that behaves like AntiYesOnSelf.exe

- Since AntiYesOnSelf.exe cannot exist, also YesOnSelf.exe and AlwaysYes.exe cannot exist, because if AntiYesOnSelf.exe would exist so would YesOnSelf.exe, we just need to reverse the output in the last before quitting, and YesOnSelf.exe is just a particular case of AlwaysYes.exe when applied on itself.
What do we want? We are not interested in these rather obscure programs AntiYesOnSelf.exe and YesOnSelf.exe. They are applied only on themselves! Well, AlwaysYes.exe would be quite interesting, but it cannot exist!

Our goal: to prove that the (very useful) program, that analyses any other program whether it crashes or not, cannot exist!

We are close to this goal!

Well, at least we proved that there are programs that cannot exist.
Ok, let us define a program **CanCrash.exe** which can analyze other programs and tell us whether or not they can crash.

- **CanCrash.exe**: outputs “yes”, if input can crash, outputs “no” if input never crashes.
- **CanCrashWeird.exe**: crashes itself, if input can crash, outputs “no” if input never crashes.
- **CrashOnSelf.exe**: crashes, if input crashes when run on itself, outputs “no” if input doesn’t crash when run on itself.
- **AntiCrashOnSelf.exe**: outputs “yes”, if input crashes when run on itself, crashes, if input doesn’t crash when run on itself.
COMPUTABILITY

- What will AntiCrashOnSelf.exe do when given itself as input? According to its own description, it should output “yes” if it crashes (a contradiction, since it can't terminate successfully with the output “yes” if it has already crashed). And again according to its own description, AntiCrashOnSelf.exe should crash if it doesn't crash— which is also self-contradictory. We've eliminated both possible behaviors of Anti-CrashOnSelf.exe, which means the program could not have existed in the first place.
• Conclusion: CanCrash.exe cannot exist, since otherwise we could implement CanCrashWeird.exe (just before output “yes” let it crash), and we could also implement CrashOnSelf.exe (just apply CanCrashWeird.exe on itself), and from it we could write AntiCrashOnSelf.exe (just do the reverse what CrashOnSelf.exe does with respect of its output.

• We now know that AntiCrashOnSelf.exe cannot exist, so CanCrash.exe cannot exist either.
• **Practical consequences** of the existence of undecidable problems?
  • We are more concerned about how long takes a solution
  • We still can implement programs that analyze other programs (program verification is a very active research topic)

• **Theoretical consequences** (brain=mind hypothesis)?
  • Does the existence of undecidable problems have implications for human thought processes?
  • If we believe that the human brain could, in principle, be simulated by a computer, then the brain is subject to the same limitations as computers.
  • That all computers, and probably humans too, have equivalent computational power — is known as the *Church-Turing thesis*.

Whether the *Church-Turing thesis* is true, is heavily debated