Bitcoin

The bitcoin *is a dematerialized and decentralized currency. It is based on two computer principles: public key cryptography and proof of work. To understand this second principle, you will create a simple model of bitcoin.*

Activity 1 (Proof of work)**.**

Goal: understand what "proof of work" means by studying a simple model. This activity is independent from the rest of the chapter. The idea is to find a problem that is difficult to solve but easy to check. Like sudokus for example: it only takes ten seconds to check that a grid is filled correctly, but it takes more than ten minutes to solve it.

The mathematical problem is as follows: you are given a prime number p and an integer y; you must find an integer x such that:

$$x^2 = y \pmod{p}$$

In other words, x is a square root of y modulo p. Be careful, there is not always a solution for x.

Examples.

- For $p = 13$ and $y = 10$, a solution is $x = 6$: because $x^2 = 6^2 = 36$ and $36 = 2 \times 13 + 10$ so $x^2 = 10 \pmod{13}$.
- The solution is not necessarily unique. For example, check that $x = 7$ is also a solution.
- There is not always a solution, for example for $p = 13$ and $y = 5$, a solution does not exist.
- Exercise: for $p = 13$, by hand find two solutions, x, to the problem $x^2 = 9 \pmod{13}$; also find a solution, x, to the problem $x^2 = 3 \pmod{13}$.

The prime number p is fixed for the rest of the activity. For easy examples we will use $p = 101$, for medium examples $p = 15\,486\,869$ and for difficult examples $p = 2\,276\,856\,017$. The larger the integer p is, the more difficult it is to obtain proof of work.

1. **Verification (easy).** Write a `verification(x,y)` function that returns `True` if x is the solution to the problem $x^2 = y \pmod{p}$ and `False` otherwise.
   Check that $x = 6\,543\,210$ is a solution when $y = 8\,371\,779$ and $p = 15\,486\,869$. Determine the calculation time required for this verification. (See the explanations after this activity.)
2. **Search for a solution (difficult).** To find a solution x, there is really no other choice for us than to test all x starting with $x = 0$, $x = 1\ldots$ Program a `square_root(y)` function that returns the first solution x to the problem for a given y (or `None` if there is no solution).
   - For $p = 101$ and $y = 17$, find x such that $x^2 = y \pmod{p}$.
   - For $p = 15\,486\,869$ and $y = 8\,371\,779$, find the solution x of the first question. How long did the search take?
   - Find a solution for $p = 15\,486\,869$ and $y = 13\,017\,204$.

Conclusion: we found a problem that was difficult to solve, but for which it is easy to check that a

given solution is suitable. For higher values of p, the search for a solution, x, can be much too long and may not succeed. We'll see how we can adjust the difficulty of the problem.

3. Instead of looking for an exact solution to our problem $x^2 = y \pmod p$, which is equivalent to $x^2 - y \pmod p = 0$, we can look for an approximate solution, i.e. one that checks:

$$x^2 - y \pmod p \leqslant m.$$

For example if $m = 5$, then you can have (modulo p): $x^2 - y = 0$, $x^2 - y = 1$, $x^2 - y = 2$,... or $x^2 - y = 5$. Of course there are now many possible solutions, x.
Program an `approximate_square_root(y,margin)` function that finds an approximate solution to our problem $x^2 = y \pmod p$.
How long does it take to find one solution to the problem when $p = 15\,486\,869$, $y = 8\,371\,779$ and $m = 20$? Choose a large prime number p and find a margin of error m so that finding a solution to the approximate problem requires about 30 to 60 seconds of calculations (for any y).

Here are some examples of prime numbers you can use for your tests:

101 1097 10651 100109 1300573 15486869

179426321 2276856017 32416187567

Lesson 1 (Stopwatch)**.**
The `time` module allows you to measure the execution time and also to know the date and time (see also the `timeit` module explained in the "Arithmetic – While loop – I" chapter). Here is a small script to measure the calculation time of an instruction.

```
import time

start_chrono = time.time()

time.sleep(2)

end_chrono = time.time()

total_chrono = end_chrono - start_chrono
print("Execution time (in seconds):", total_chrono)
```

Explanations.

- The module is named `time`.
- The function that returns the time is also named `time()`, so we call it using `time.time()`!
- The `time()` function returns a number that we are not interested in. What interests us is the difference between two values of this function.
- `start_chrono = time.time()` is like starting a stopwatch.
- `end_chrono = time.time()` is like stopping it.
- `total_chrono = end_chrono - start_chrono` is the total time taken by our script in seconds.
- The `time.sleep(duration)` function pauses the program for a certain amount of seconds.

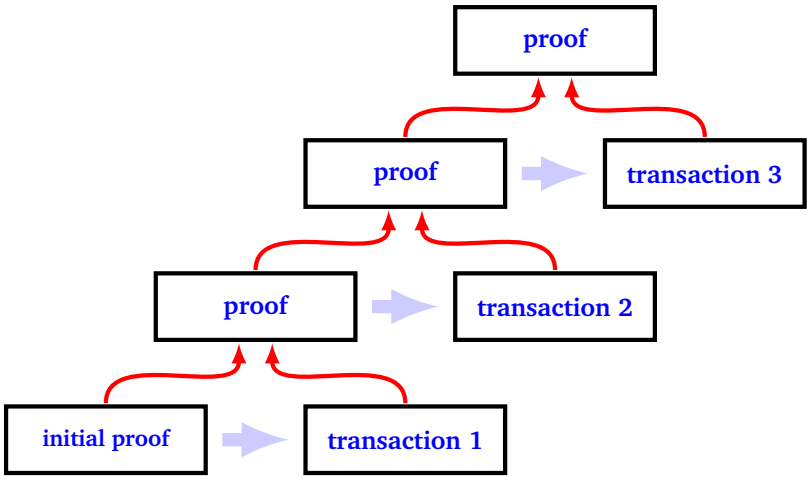
Lesson 2 (Bitcoin and blockchain)**.**
Bitcoin is a dematerialized currency. Transactions are recorded in a ledger called *blockchain*. We will build a (very simplified) model of such an account book.
Imagine a group of friends who want to share the group's expenses in the simplest possible way. At the beginning everyone has 1000 bitcoins and everyone's expenses and income are recorded as they go along. At the end of the holidays, they will regularize the situation.
The list of expenses/revenues is noted in the account book, for example:

- "Amir spent 100 bitcoins"
- "Barbara received 45 bitcoins"
- etc.

Just look through the account book to see how much everyone has received or spent since the beginning.

To prevent someone from falsifying the account book, after each transaction a certification based on a "proof of work" is added to the book. Here is what is written in the ledger:

1. We start with any initial proof of work. For us it will be `[0,0,0,0,0,0]`.
2. The first transaction is written (for example `"Amir -100"`).
3. A proof of work is calculated and written in the book, which will serve as a certificate. It's a list (for example `[56,42,10,98,2,34]`) obtained after many calculations taking into account the previous transaction and the previous proof of work.
4. For each new transaction (for example `"Barbara +45"`), someone calculates a proof of work for the last transaction associated with the previous proof. We write the transaction, then the proof of work.



The proof of work that is calculated depends on the previous transaction but also on the previous proof of work, which itself depends on the previous data... Thus, each new proof of work actually depends on everything that has been written since the beginning (even if the calculation explicitly refers only to the last two entries).
Someone who would like to fake a past transaction should recalculate all the proofs of work that come next. This is not possible for a single person: there are several proofs of work to calculate and each proof requires a lot of calculations.

Activity 2 (Tools for lists)**.**

Goal: build useful functions to manipulate lists of integers for the following activities.

Our lists are lists of integers between 0 and 99.

1. Program an `addition(mylist1,mylist2)` function that adds two lists of the same length, element by element, and applies modulo 100 to each term. For example, `addition([1,2,3,4,5,6],[1,1,1,98,98,98])` returns `[2,3,4,2,3,4]`.
2. A list `mylist` is smaller than the `max_list` list if each element of `mylist` is less than or equal to it's corresponding element of `max_list`. We will look for lists that start with zeros (or zeros and then rather small numbers).
   For example, the list `[0,0,1,2,3,4]` is smaller than the list `[0,0,5]`. This is not the case for the list `[0,10,0,1,1]`. Another example: being smaller than the list `[0,0,0]` means starting with three zeros. Being smaller than the list `[0,0,1]` means starting with `[0,0,0]` or `[0,0,1]`.
   Program an `is_smaller(mylist,max_list)` function that returns `True` when `mylist` is smaller than `max_list`.
3. We will need to transform a sentence into a list of numbers. In addition, we will split our lists into blocks of size N (with $N = 6$), we will add zeros at the beginning of the list so that its length is a multiple of N.
   Write a `sentence_to_list(sentence)` function that converts a string into a list of integers between 0 and 99 and if necessary adds leading zeros so that the list has the correct size.
   The formula to use to convert a character to an integer strictly less than 100 is:
   ```
   ord(c) % 100
   ```
   For example: if `sentence = "Be happy!"` then the function returns:
   ```
   [0, 0, 0, 66, 1, 32, 4, 97, 12, 12, 21, 33]
   ```
   The character `"e"` has ASCII/unicode code 101 so, modulo 100, the number is 1. Note that the function adds three 0's at the beginning of the list to have a length that is a multiple of $N = 6$.

Activity 3 (Hash function)**.**

Goal: create a hash function. From a long message we calculate a short fingerprint. It's hard to find two different messages with the same fingerprint.

In this activity, our message is a list of integers (between 0 and 99) of any length that is a multiple of $N = 6$. We transform it into a list of length $N = 6$: its fingerprint, named *hash*. Here are some examples of what our hash function will do:

- the list `[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]` has a hash:
  `[10, 0, 58, 28, 0, 90]`
- the slightly different list `[1, 1, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]` has a hash:
  `[25, 14, 29, 1, 19, 6]`

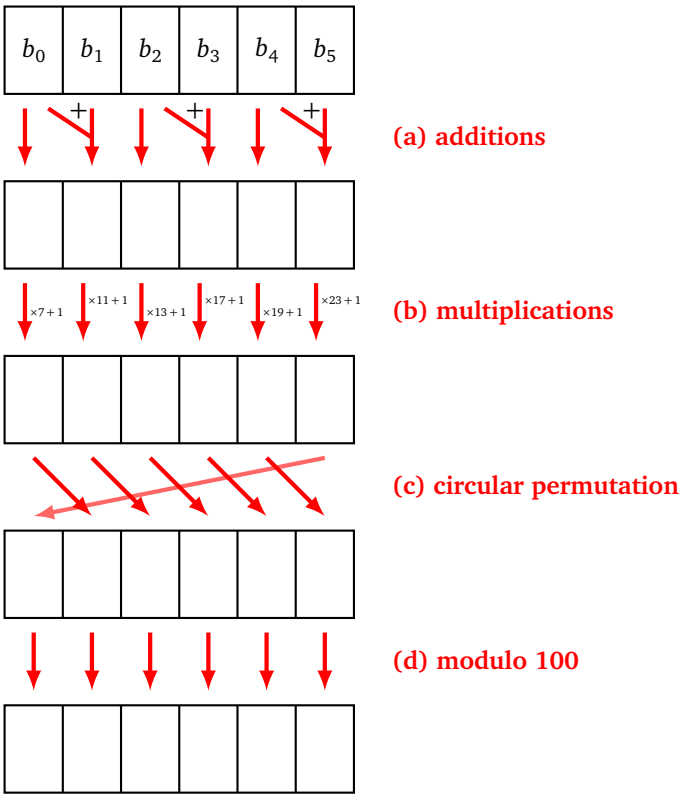
The idea is to mix the numbers per block of $N = 6$ integers, then combine this block with the next and start again, until you get a single block.

1. **One round.**
   For a block $[b_0, b_1, b_2, b_3, b_4, b_5]$ of size $N = 6$, *doing a round* consists in making the following operations:
   (a) We add some integers:
   $$[b'_0, b'_1, b'_2, b'_3, b'_4, b'_5] = [b_0, b_1 + b_0, b_2, b_3 + b_2, b_4, b_5 + b_4]$$
   (b) We multiply these integers by prime numbers (in order $7, 11, 13, 17, 19, 23$) and add 1:
   $$[b''_0, b''_1, b''_2, b''_3, b''_4, b''_5] = [7 \times b'_0 + 1, 11 \times b'_1 + 1, 13 \times b'_2 + 1, 17 \times b'_3 + 1, 19 \times b'_4 + 1, 23 \times b'_5 + 1]$$

(c) We make a circular permutation (the last one goes first):

$$[b_0''', b_1''', b_2''', b_3''', b_4''', b_5'''] = [b_5'', b_0'', b_1'', b_2'', b_3'', b_4'']$$

(d) We reduce each integer modulo 100 to get integers between 0 and 99.



Starting from the block $[0, 1, 2, 3, 4, 5]$, we have successively completed:

(a) additions: $[0, 1, 2, 5, 4, 9]$

(b) multiplications: $[7 \times 0 + 1, 11 \times 1 + 1, 13 \times 2 + 1, 17 \times 5 + 1, 19 \times 4 + 1, 23 \times 9 + 1] = [1, 12, 27, 86, 77, 208]$

(c) permutation: $[208, 1, 12, 27, 86, 77]$

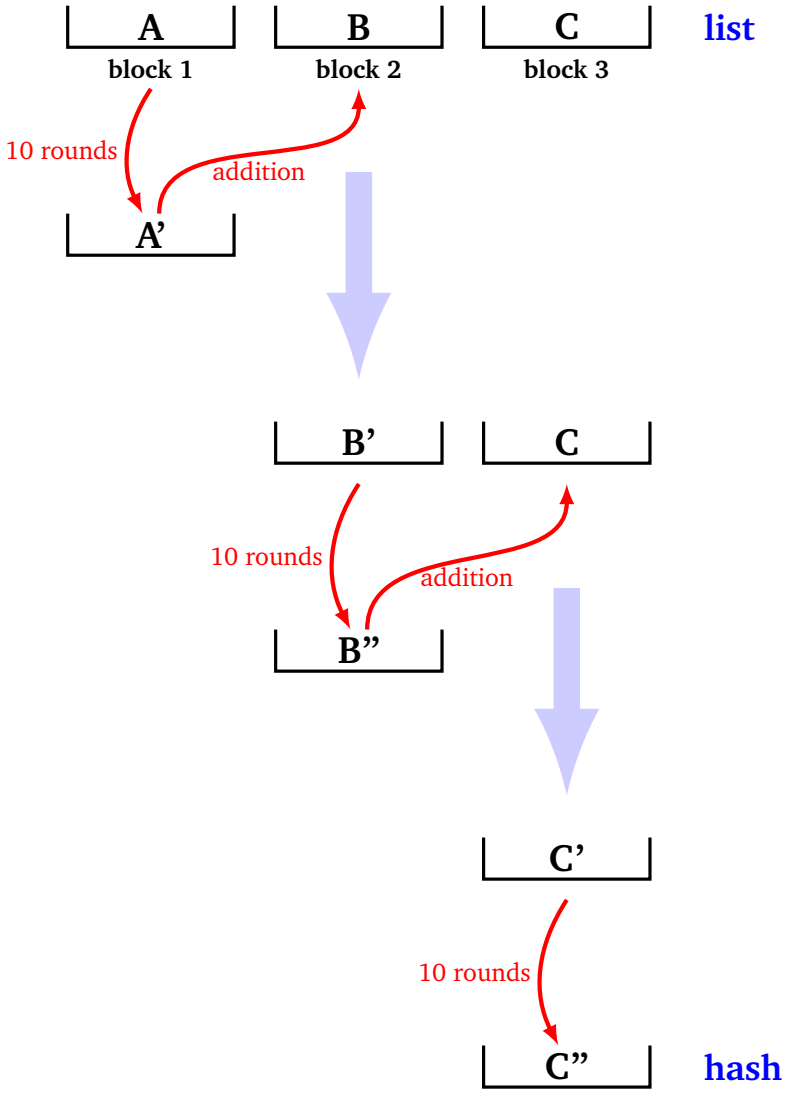
(d) reduction modulo 100: $[8, 1, 12, 27, 86, 77]$

Program an `one_round(block)` function which returns the transformation of the block after these operations. Verify that the block $[1, 1, 2, 3, 4, 5]$ is transformed into $[8, 8, 23, 27, 86, 77]$.

2. **Ten rounds.**
   To mix each block well, program a `ten_rounds(block)` function that iterates the previous operations ten times. After 10 rounds:
   - the block $[0, 1, 2, 3, 4, 5]$ becomes $[98, 95, 86, 55, 66, 75]$,
   - the block $[1, 1, 2, 3, 4, 5]$ becomes $[18, 74, 4, 42, 77, 42]$.

   Two nearby blocks are transformed into two very different blocks!
3. **Hash of a list.**
   Starting from a list whose length is a multiple of $N = 6$, it is split into blocks of length 6 and the hash of this list is calculated according to the following algorithm:
   - The first block of the list is extracted, then mixed for 10 rounds.
   - We add term to term (and modulo 100), the result of this mix to the second block.
   - We start again from the new second block.
   - When there is only one block left, we mix for 10 rounds, the result is the hash of the list.

Here is the diagram of a situation with three blocks: first there are three blocks (A, B, C); next there are only two blocks (B' and C) left; at the end there is only one block (C"): this is the hash!



Example using the list $[0, 1, 2, 3, 4, 5, 1, 1, 1, 1, 1, 1, 10, 10, 10, 10, 10, 10]$.

- The first block is $[0, 1, 2, 3, 4, 5]$, its mix after 10 rounds is $[98, 95, 86, 55, 66, 75]$.
- This mix is added to the second block $[1, 1, 1, 1, 1, 1]$ (see the `addition()` function in activity 2).
- The remaining list is now $[99, 96, 87, 56, 67, 76, 10, 10, 10, 10, 10, 10]$.
- Let's do it again. The new first block is $[99, 96, 87, 56, 67, 76]$, its mix after 10 rounds is $[60, 82, 12, 94, 6, 80]$, it is added to the last block $[10, 10, 10, 10, 10, 10]$ to get (modulo 100) $[70, 92, 22, 4, 16, 90]$.
- A last mix is done for 10 rounds to obtain the hash: $[77, 91, 5, 91, 89, 99]$.

Program a `bithash(mylist)` function that returns the fingerprint of a list. Test it on the examples given at the beginning of the activity.

Activity 4 (Proof of work - Mining)**.**

Goal: build a proof of work mechanism using our hash function.

We are going to build a complicated problem to solve, for which, if someone gives us the solution, then it is easy to check if it is correct.

Problem to solve. We are given a list, we need to find a block such that, when you add it to the list, it produces a hash starting with zeros. More precisely, given a list `mylist` and a maximum target of `max_list`, we have to find a block, `proof`, which, concatenated to the list and then hashed, is smaller than the list of `max_list`, i.e.:

`bithash(mylist + proof)` smaller than `max_list`

The list is of any length (a multiple of $N = 6$), the proof is a block of length N, the objective is to find a list starting with 0's (see activity 2).

For example: let `mylist = [0,1,2,3,4,5]` and `max_list = [0,0,7]`. Which `proof` block can I concatenate to `mylist` to solve my problem?

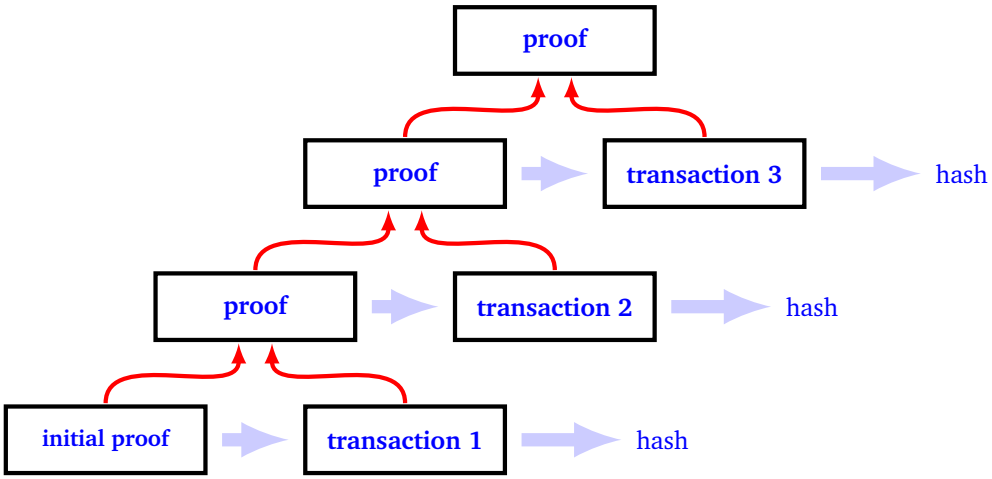
- `proof = [12,3,24,72,47,77]` is appropriate because concatenated to our list it gives `[0,1,2,3,4,5,12,3,24,72,47,77]` and the hash of this whole list gives `[0,0,5,47,44,71]` which starts with `[0,0,5]` smaller than the target.
- `proof = [0,0,2,0,61,2]` is also suitable because after concatenation we have `[0,1,2,3,4,5,0,0,2,0,61,2]` whose hash gives `[0,0,3,12,58,92]`.
- `[97,49,93,87,89,47]` is not suitable, because after concatenation and hashing we get `[0,0,8,28,6,60]` which is greater than the desired objective.

1. **Verification (easy).**
   Program a `verification_proof_of_work(mylist,proof)` function that returns true if the proposed solution `proof` is suitable for `mylist`. Use the `is_smaller()` function from activity 2.
2. **Search for a solution (difficult).**
   Program a `proof_of_work(mylist)` function that looks for a `proof` block that is a solution to our problem for the given list.
   Hints.
   - The easiest method is to take a `proof` block of random numbers and start again until a solution is found.
   - You can also systematically test all blocks starting with `[0,0,0,0,0,0]`, then `[0,0,0,0,0,1]`... and stop at the first appropriate one.
   - You adjust the difficulty of the problem by changing the objective: easy with `max_list = [0,0,50]`, medium with `max_list = [0,0,5]`, difficult with `max_list = [0,0,0]`, very difficult with `max_list = [0,0,0,0]`.
   - As there are several solutions, you do not necessarily get the same solution for each search.
3. **Calculation time.**
   Compare the calculation time of a simple check against the time of searching for a solution. Choose the `max_list` objective so that the search for a proof of work requires about 30 to 60 seconds of calculations.

For the bitcoin, those who calculate proofs of work are called the *miners*. The first one to find a proof wins a reward. The difficulty of the problem is adjusted so that that the calculation time taken by the winner (among all the miners) to find a solution is about 10 minutes.

Activity 5 (Your bitcoins)**.**

Goal: create an account book (called blockchain *for bitcoin) that records all transactions, this ledger is public and certified. It is practically impossible to falsify a transaction that has already been registered.*



1. **Initialization and addition of a transaction.**
   (a) Initialize a global `blockchain` variable which is a list and contains at the beginning a zero proof: `blockchain = [ [0,0,0,0,0,0] ]`.
   (b) A *transaction* is a string including a name and the amount to add (or deduct) to your account. For example `"Abel +25"` or `"Barbara -45"`.
   Program an `add_transaction(transaction)` function that adds the string `transaction` to the list `blockchain`. For example after initialization `add_transaction("Camille +100")`, `blockchain` is `[ [0,0,0,0,0,0], "Camille +100" ]`. Careful, to be able to modify `blockchain` you must start the function with: `global blockchain`.
2. As soon as a transaction is added, a proof of work must be calculated and added to the account book. Program a `mining()` function, without parameters, that adds a proof of work to the book. Here's how to do it:
   - We take the last transaction `transaction`, we transform it into a list of integers by the `sentence_to_list()` function from activity 2.
   - We also need `prev_proof`, the previous proof of work just before this transaction.
   - We form the `mylist` list composed, first of elements of `prev_proof`, then of elements of the list of integers obtained by converting the string `transaction`.
   - A proof of work is calculated from this list.
   - This proof is added to the account book.

   For example, if the book ends with:
   `[3,1,4,1,5,9], "Abel +35"`
   then after calculation of the proof of work the book ends with, for example:
   `[3,1,4,1,5,9], "Abel +35", [32,17,37,73,52,90]`
   It should be remembered that a proof of work is not unique and that it also depends on the `max_list` objective.
   Only one person at a time adds a proof of work. However, everyone has the opportunity to verify that the proposed proof is correct (and should do it). Write a `verification_blockchain()` function, without parameters, that checks that the last proof added to the `blockchain` is valid.
3. Write a blockchain that corresponds to the following data:
   - We take `max_list = [0,0,5]` and start with `blockchain = [ [0,0,0,0,0,0] ]`.

> - `"Alfred -100"` (Alfred owes 100 bitcoins).
> - Barnabe receives 150.
> - Chloe wins 35 bitcoins.
>
> Conclusion: let's imagine that Alfred wants to cheat, he wants to change the account book in order to receive 100 bitcoins instead of having a debit, so he has to change the transaction concerning him to `"Alfred +100"` but then he has to recalculate a new proof of work which is complicated, moreover he has to recalculate the proof of the Barnabe transaction and also that of the Chloe transaction!
> Someone who wants to modify a transaction must modify all the following proofs of work. If each proof requires sufficient computing time this is impossible. For the bitcoin each proof requires a lot of calculations (too much for a single person) and a new proof is to be calculated every 10 minutes. It is therefore impossible for a person to modify a past transaction.

The other aspect of the bitcoin that we didn't address is to make sure that each person involved is who they say they are, so that no one can get someone else's money. This is made possible by private/public key cryptography (RSA system). Each account is identified by a public key (two very large integers), which guarantees anonymity. But above all, only the one who has the private key to the account (a large integer) can access his bitcoins.