# CSCI 120 Introduction to Computation
# Introduction (draft)

Saad Mneimneh
Visiting Professor
Hunter College of CUNY

## 1 About the name

The name of this course as officially listed in the catalog is "Introduction to Computers". I prefer to call it "Introduction to Computation" (or computing). The reason is beyond the semantics of the title or even the content of the course. Here's why. It has become clear to me over the years that, whenever students see the word "computer", they visualize this thing:

Figure 1: Computer ?

Well, more or less... Of course, the figure that will register in your mind depends on the setting and on the current technology (which is very dynamic by the way). For instance, I am sure that, to you, a computer would range from being a laptop, to possibly an advanced PDA (Personal Digital Assistant) or a smart phone (hence I will not ask you to turn off your cell phone in class).

Nevertheless, Figure 1 is definitely a standard interpretation of the word computer. Next thing you think is that you are going to learn how to use Windows and Microsoft Office, or how to fix your computer if it crashes. This course is not about that. This course is about concepts and notions that are used to build computing devices, with eventually an emphasis of course on the current technologies (today's computers).

Here's another reason for using the word computation instead of computers. Let's start with a (rather unclear) definition:

**com.pu.ta.tion:** *noun* **a:** the act or action of computing, **b:** the use or operation of a computer (Merriam-Webster Dictionary)

Although computation is carried out by a computer (meaning **b**), computation is also an abstract notion (meaning **a**). It is a process by itself that we find everywhere in nature. To some extent is it independent form the underlying physical machine (e.g. the computer). Computation is what makes the field of computer science, a science. We will explore this notion further when we talk about *Algorithms*.

For the time being, let me attempt to *shake* or *blur* Figure 1 in your mind. I will do that by practicing what is called "thinking outside the box", the box of Figure 1 that is. I will draw a parallel between a biological process and a few components of a "computer" that should be familiar to a general audience. This will help interpret the biological process as a computation. But more importantly, this computation, being biological in nature, does not require the computer of Figure 1.

## 2 Who said the first computer was invented in the 1940s?

An important question in biological sciences is what makes life? A simple answer is Proteins and Nucleic Acids. Proteins are responsible for almost all body functions. Nucleic acids encode information necessary to produce the proteins and pass this "recipe" to subsequent generations (permanent storage). We have two kinds of nucleic acids: ribonucleic acids RNAs and dioxy-ribonucleic acids DNAs.

A DNA is a chain of simpler molecules, namely sugar molecules. Each sugar molecule is attached to a base and this is what makes it different from the other sugar molecules. We have 4 bases, A, G, C, and T, thus the DNA can be viewed as a long sequence of 4 letters.

The DNA is actually a double stranded helix (discovered in 1953). The two strands hold together because each base in one strand bonds with a complementary base in the other. $A \leftrightarrow T$ and $C \leftrightarrow G$. This makes it more stable and suitable as a storage device. The RNA is similar to the DNA but it is single stranded (hence less stable), and every occurrence of a T is replaced by a U. U also bonds with A.

The genome is a collection of long DNAs that are called chromosomes. A gene is a stretch along a chromosome that encode the information for producing a specific protein. Only certain stretches on the chromosomes are genes. If fact, it was believed that 90% of the our DNA is *junk* (well, not anymore!).

How does a gene produce a protein? A process called *transcription* creates an RNA by separating the DNA strands around the gene area (see figure below). The RNA is then synthesized into a protein. This latter process is called *translation*. The protein interacts with other proteins, with RNAs, and with DNA to perform several important functions in the body. The following figure illustrates the two processes of transcription and translation.

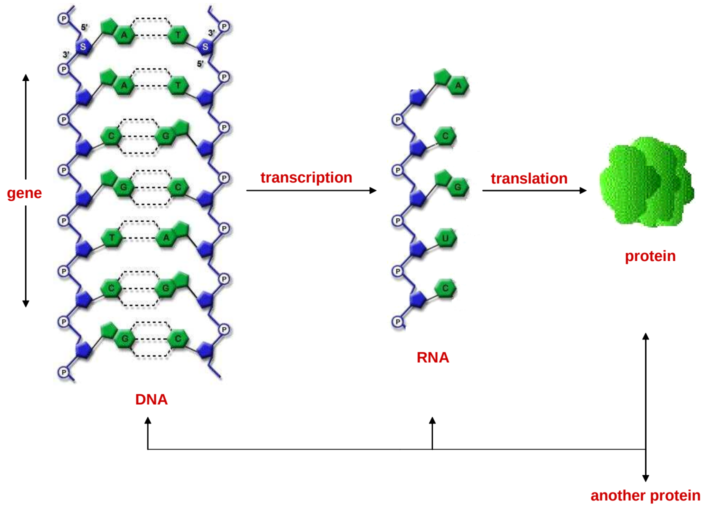

Figure 2: Biology in one picture

Here's a parallel of the biological process that is seen in computers. Much like DNA, the hard disk of a computer permanently stores information for later retrieval. The information is stored in some form; for instance, if it is a code, it is usually in a textual format, following a certain syntax, like C++ or Java. The code plays the role of an RNA. While an RNA is synthesized into a protein, the code is compiled into an executable program that is loaded into memory. Now the program is our protein. The program runs and performs some tasks while interacting with other programs (e.g. networks), reading and writing data from and to the hard disk, and possibly updating other codes.

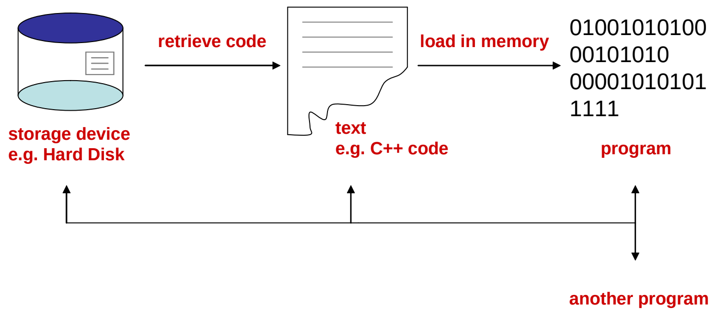

Figure 3: Biological process in computer jargon

The following table summarizes this parallel between a *biological computer* and the *electronic computer* that we know:

| Biology | Electronics |
|---|---|
| DNA (chemical) | Hard disk (magnetic) |
| Transcription | Data retrieval |
| RNA (chemical data) | Code (textual data) |
| Translation | Compilation |
| Protein | Program |

Therefore, one can claim that the biological process is actually a computation. Moreover, this makes our body a computer! What happens to Figure 1 now?

To expose the concept of computation even further, let us consider the following question: If DNA provides a permanent storage, how come we are all different? The answer: DNA is not really permanent across generations. A child inherits different portions of the DNAs of the mother and father; hence receiving a new DNA. This process is called *recombination*. Moreover, a DNA portion may change randomly due to a *mutation*. Computer scientists sometimes explore the concepts of recombination and mutation to solve difficult computational problems. Here's how:

- given a problem, randomly generate a number of solutions (could be bad solutions)
- encode solutions into strings (like DNA)
- generate more solutions (as offspring) by using *recombination* and *mutation* (create new population)
- make sure to keep the better solutions in the new population (natural selection)
- repeat

The process described above is often referred to as a genetic algorithm. Sometimes it is possible to efficiently search the space of solutions in this way, instead of the brute force alternative of considering every possible solution. We might have a chance to look at this later, but here's an example that I prepared especially for this lecture: The objective is to ultimately learn the concept of a square. First, I need a way to **represent** shapes. I started with a number of random shapes, each encoded as strings of 0's and 1's (that's my DNA except that the alphabet is {0,} instead of {A, G, C, T}). Each string is interpreted as follows: hold a pen and start tracing a path of points on the paper. Whenever you see a 1, turn right; otherwise, keep going in the same direction. Therefore, a string represents a drawing. Moreover, that drawing may actually cover a square area on the paper. However, most likely this is not going to happen with the randomly generated strings, without previous knowledge of the concept we are trying to learn. So I performed a genetic algorithm using a natural selection criterion of keeping those strings that achieve a high score, with the score being the number of pairs of points that are adjacent on the paper. Intuitively, a square should have a lot of such adjacent pairs (on average, each point has 4 adjacent points: up, down, left, and right). I ended up with a population of strings that produce squares (and rectangles) by spiraling around (because they can only turn right).

For instance, I started with a population of random strings like

$$1000110100101001010001110$$

Then I repeatedly picked two strings from the population and *mated* them to produce a new string using recombination, e.g. *100011010* 0101001010001110 and 001001110 *1010010001110010* may recombine to produce

*100011010 1010010001110010* and 001001110 0101001010001110

The recombination point is chosen randomly. Then some random characters are mutated. Here's how the population evolved (the last one learned the square!):

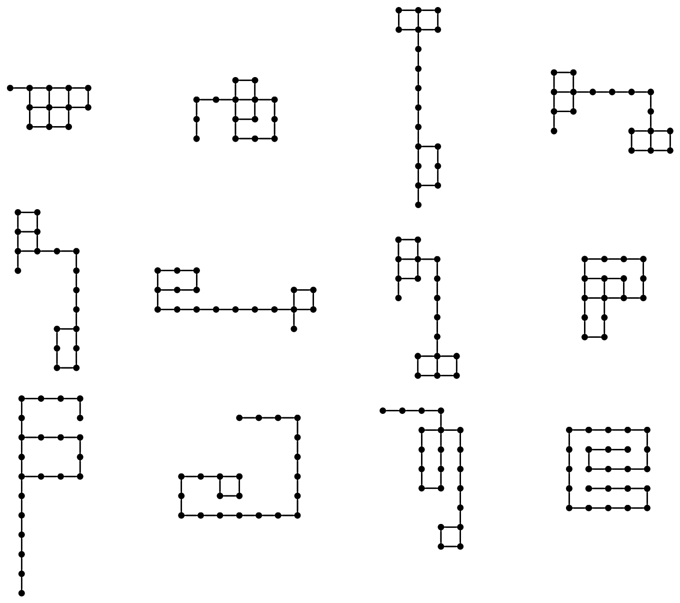

Figure 4: Learning a square

Although genetic algorithms do not rely on any biological component, they borrow the concept from biology. Therefore, this shows that, in principle, the computation is the important aspect and not the machine on which it runs. This also shows the power of abstraction. The human mind is able to abstract concepts from observations and apply those concepts in possibly different situations. Abstraction is a very important aspect in computer science. For instance, one may abstract the notion of a whole computer as follows:

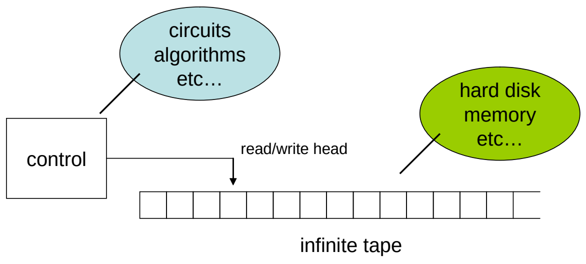

Figure 5: Turing machine

This is known as the Turing machine, after Alan Turing. How does this machine compute? The tape is initialized with some input. The head is originally pointing to the beginning of the tape. Repeatedly, the machine reads the symbol under the head. Depending on the symbol read, and on the state of the control, the machine decides how to update the symbol using the write capability of the head, changes state, and then either moves the head left or right, or stay. This is repeated until the control enters a special *halt* state. The final output is read from the tape. Although actual computers are not exactly Turing machines, they are equivalent in power: anything that can be done by a computer can be done by a Turing machine and vice versa. So the Turing machine can be regarded as an abstract representation of a computer.

## 3 So what is a computer then?

Well, a computer is a *device* (now electronic) that carries out a computation (circular definition?). Without plunging deeply into linguistics, a computer is simply a device that can store, retrieve, and process data. According to this definition, your brain is a computer! Don't be surprised because this is definitely true (I will prove it to you shortly).

Since we often require a computer to accomplish a certain task, a computation is not just a number of random operations consisting of data storage, retrieval, and processing. The most fundamental concept in computer science is the *Algorithm* [1]. Informally speaking, an algorithm is a well defined sequence of computational steps that accomplish a certain task. The task can be visualized as a relation between an input and an output. Therefore, an algorithm is a well defined sequence of computational steps that transforms some input into the desired output. Here are some examples:

- sorting numbers: the input is a set of numbers, the output is the same set of numbers in sorted order (from smallest to largest). The algorithm in this case is a sorting algorithm.
- cooking: the input is a set of ingredients, the output is the dish. The algorithm in this case is a recipe.
- addition: the input is two numbers, the output is one number which is their sum. The algorithm in this case is an adder.

Therefore, before a computer can perform a task, an algorithm for performing that task must be discovered. In fact, designing algorithms is one of the most important fields in computer science. The problem of finding algorithms goes back to the 1930s when mathematicians like Kurt Gődel, Alan Turing, Alonzo Church, and others were trying to answer the fundamental question whether every task has an algorithm (the answer is NO) It is actually then that the stage was set for the emergence of a new discipline known as computer science.

With the power of abstraction, once an algorithm for a particular task has been found, and proven correct, then it can be used safely as a black box or as a building block, without having to worry about how or why this algorithm works (unless otherwise you are interested in such knowledge). Of course, issues such as writing the code, finding an efficient implementation, or adapting to a special machine will eventually arise.

To materialize the notion of algorithm, let's look for a moment at the task of adding two numbers. We all know how to add two numbers (hopefully). Let's try the following (thanks to Yale Patt for the idea):

Add 12 and 8:

$$\begin{array}{rr} & 12 \\ + & 8 \\ \hline \end{array}$$

[1]The word algorithm comes from the arabian mathematician **Alko**w**arizm**i 780 - 850 A.D.

Add 129 and 17:

$$\begin{array}{rr} & 129 \\ + & 17 \\ \hline \end{array}$$

Read in reverse to see the answers. .641=71+921, 02=8+21 :srewsnA

Now recall that I promised to prove to you that your brain is a computer. If everything works as I expect, you will actually perform two different algorithms, one for each addition. For the first addition, you are likely to write down the digit 2 followed by the digit 0. The answer is 20. For this addition, you perform a simple *lookup*. The answer to 12+8 is somehow stored permanently in your brain. When you see the pattern 12+8, you access that area in your brain and retrieve the answer. For the second addition, you are likely to write down the digit 6 first, followed by the digit 4, followed by the digit 1. The answer is 146. For this addition, you perform the standard addition procedure that ripples a carry. Therefore, you add 9 and 7, which gives 16, you write down the 6 and you carry 1, etc... You use your short term memory (or in this case maybe the paper) to keep track of the carry. In either case, you are doing what a computer would do. We will see later how computers represent numbers and perform arithmetic operations on them.

You may be unaware of the *representation* of the addition algorithm in your brain, but it is definitely encoded into your brain cells. Similarly, a computer needs a way to represent algorithms in a form that is compatible with the machine. A representation of an algorithm is often called a *program*. You may have seen these programs, written in C++ or Java for instance, and printed on a piece of paper, or displayed on the screen. These however are then encoded in a manner compatible with the machine, using bits of 0's and 1's (analogy: translation from RNA to protein). We will talk about bits later. The whole process of writing the program and encoding it is called *programming*. Programs and the algorithms they represent are called *software*. The machine itself is referred to as *hardware*. Therefore, the software must be compatible with the hardware for it to work. For instance, a PC and a MAC require different software.

## 4 Summary of concepts

Here's a summary of the concepts we have seen so far:

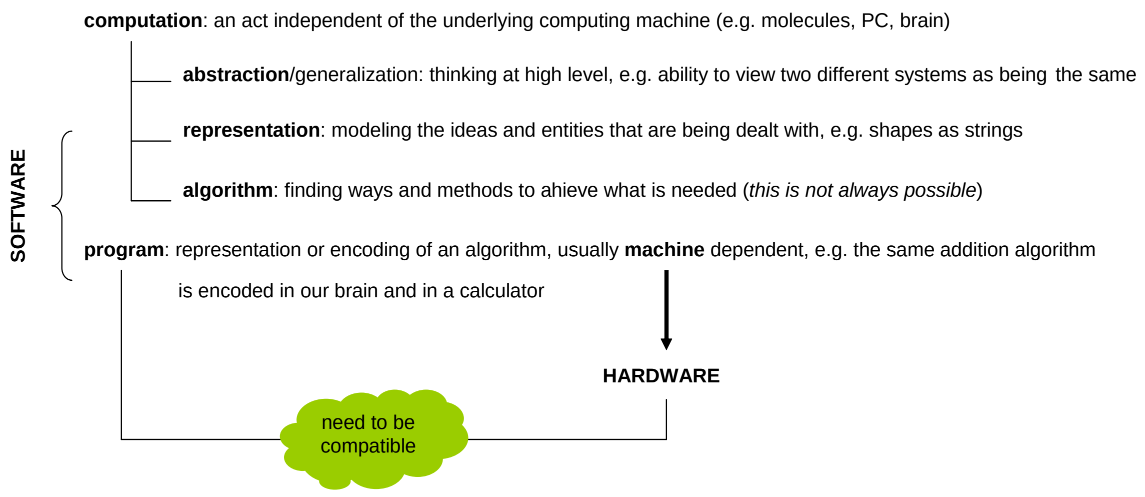
**computation**: an act independent of the underlying computing machine (e.g. molecules, PC, brain)
**abstraction**/generalization: thinking at high level, e.g. ability to view two different systems as being the same
**representation**: modeling the ideas and entities that are being dealt with, e.g. shapes as strings
**algorithm**: finding ways and methods to ahieve what is needed (*this is not always possible*)
SOFTWARE
**program**: representation or encoding of an algorithm, usually **machine** dependent, e.g. the same addition algorithm is encoded in our brain and in a calculator
HARDWARE
need to be compatible

Figure 6: Summary of concepts