Chapter 1

Introduction to computer science

1. Brief history of computer science

1.1. From the beginning to von Neumann architecture

Since antiquity, mankind has tried to count and calculate. Knots, ropes, fingers, and other artifacts were employed by our ancestors to make counting easier. Humans have also observed natural events and used some of them to measure time. We hence see some numeration systems based on the solar cycle and time. For example, the Sumerians used a sexagesimal numeration system, which is still used today to count minutes and seconds. Fingers have produced numerous numeral systems, including the decimal and duodecimal systems.

Calculations are typically complicated and time-consuming. As a result, mankind attempted to develop more sophisticated calculating methods and technologies in order to speed things up. Perhaps the abacus (see figure 1.1)was the first successful device to accomplish such a task. The origin of this device is unknown, although it has been extensively utilized, particularly in China (where its earliest record dates from the second century B.C.) and Japan (via the well-known Soroban).

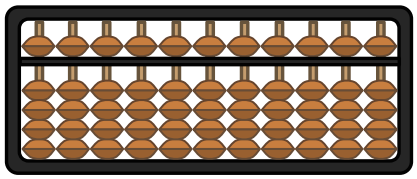

Figure 1.1 - the abacus has been to make calculating quite fast

Actually, computations may be carried out in a variety of ways. For example, Euclidean geometry can be used through the Pythagorean theorem and Thales' theorem for instance (can you figure out how ?). Nevertheless, doing out calculations in this way still takes a lot of time and effort.

During the Industrial Revolution, people began developing increasingly complex machines to automate hard tasks. However, most of the inventions were mainly focused with products manufacture. It was not until the 19th century that we saw the first attempts to create mechanical devices capable of doing computations. Charles Babbage was the first to describe a programmable machine that is theoretically comparable to current computers. Although the machine was never fully built, first programs were already written for this machine by Ada Lovelace.

In the 19th century, George Boole made great theoretical achievements in calculation theory by inventing *Boole algebra*. Thanks to that formalism based on proposition logic, calculations can be described using boolean functions, which subsequently formed the basis of contemporary computers.

Herman Hollerith, an American engineer, created the punched card tabulating machine in 1884. This electro-mechanical and semiautomatic system used punched cards to increment counters and record information. Thanks to this discovery, the US Census Bureau could conduct a census in 6 years rather than 8 years (which was viewed as a significant improvement).

Alan Turing described the first mathematical universal machine (known as the *Turing machine*) in 1936. This abstract machine can do any computable calculation. This machine has influenced nearly every programming language and has had a significant impact on computer design up to this day. Other researchers, like Alonzo Church (who is regarded as the father of functional programming languages), have also contributed to similar efforts. The expression **Turing complete** is now used to describe a machine or a language that can describe and execute any computable calculation. Turing was a visionary scientist who was among the first to investigate the development of intelligent systems. He even defined a test, known as the Turing test, to determine whether a machine is really intelligent. We may confidently state that the fates of computer science and artificial intelligence have been sealed since the beginning.

The attempts to create an all-electric computing machine went eventually successful in 1937. The ABC, a rudimentary and non-programmable computer, was made with vacuum tubes (electric and primitive devices that can control electric flows). During World War 2, many special-purpose machines were built in order to encrypt or break encrypted messages.

The first general-purpose computer, known as ENIAC (for Electronic Numerical Integrator and Computer), was built at the University of Pennsylvania and utilized 18000 vacuum tubes. Later on, several comparable computers were also built (EDVAC, BINAC, UNIVAC 1). These computers are quite primitive in comparison to modern ones, yet they were considered great innovations. In 1945, John von Neumann described the first modern computing architecture, which had a significant effect on computer design. Von Neumann architecture enabled computers to be programmed in a highly flexible manner. This era saw the emergence of concepts and principles for programs, subroutines, and programming languages.

Von Neumann architecture is built around the concept of stored programs. It includes the following components:
- A **processing unit** (also called Arithmetic Logical Unit) that contains arithmetic and boolean registers. It can calculate any computable calculations through a set of simple instructions.
- A **control unit** that controls the program flow. It uses an instruction register and a program counter.
- An **internal memory** for storing instructions and data. Data may be read or written on it.
- A large **external mass storage**
- **Input and output** capabilities

The CPU (Central Processing Unit) is composed of the processing unit and the control unit, as shown in figure 1.2 .

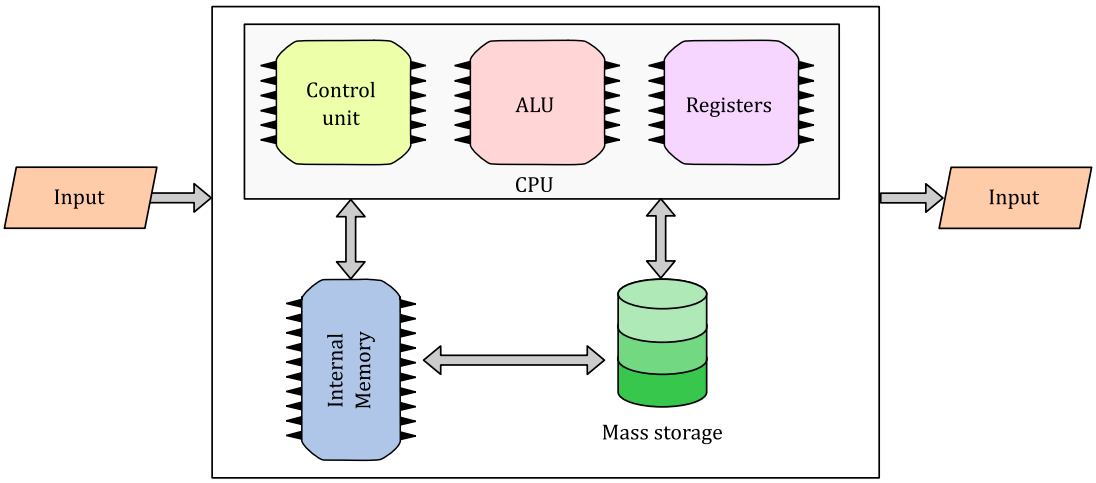

Figure 1.2 - the architecture of von Neumann

1.2. Computer generations

It should be noted that the term "computer" can apply to both *hardware* and *software* components. Any device, circuit, or material artifact that a system uses to perform computations, receive, and transfer data is regarded as hardware. Software refers to programs that execute on a computer at any abstraction level. Hardware and software have improved significantly as computers have evolved.

Computers can typically be categorized into three generations:

First generation computers

This generation, produced between 1937 and 1946, were built by using vacuum tubes, which resulted in low performance in general. Magnetic tapes and disks were utilized for storage, with printers serving as output. Rudimentary *operating systems* were developed, and programming languages were very simple and difficult for people to learn since instructions had to be hard-coded using front panel switches as shown in figure figure 1.3 .

Figure 1.3 - the front panel to control the computer

Second generation computers

This generation, produced between 1946 and 1962, saw the introduction of the first commercial computers (UNIVAC1, IBM650, IBM700) and the use of transistors in computer construction. As computers became more powerful, more sophisticated programming languages emerged, including second generation languages (such as assembly language) and third generation languages (high-level languages like Fortran, which, despite being developed in 1957, is still widely used, particularly in scientific computations requiring heavy mathematical operations).

Some worth-naming operating systems were developed. GMOSIS has been developed for IBM computers where punch cards were used to submit tasks to the system in order to execute them.

Third generation computers

This generation, produced between 1963 and the present, is built by using integrated circuits and even more advanced technologies, including nanoparticules, biological systems, and quantum physics. Computer hardware has undergone incredible advancements, reducing in size while increasing memory capacity, speed, and reliability (in PCs, smartphones, embedded systems, etc.). This allowed outstanding software evolution, starting with contemporary and highly performant operating systems (such as UNIX) that got more user-friendly over time (MSDOS, Windows, Linux, MacOS, Android, iOS). Programming languages have advanced significantly since the fourth generation (support for databases, GUIs, web development, and so on) to modern natural language processing, which allows programming computers to understand everyday spoken languages. In parallel, significantly theoretical research has been conducted, resulting in highly performant algorithms. Parallelism becomes an inherent part of this generation (concurrency, distributed systems, cloud computing, and so on).

2. Introduction to algorithms

Computer science is all about solving problems using computers. A Turing complete machine can execute any computable task (referred to as a *decidable problem*), but it cannot generally create its own programs. The Turing machine has a very basic construction yet is capable of solving very complicated problems. Yet it should have the right *recipe*.

An algorithm is any well-defined computational procedure that solve a well-defined problem. Problem specification generally consists of a relationship between certain values, known as the *input*, and other values, known as the *output*. The relationship is often described in terms of logical properties involving the input and the output. The algorithm defines a computational procedure that can be applied on arbitrary values that satisfies the initial conditions of the algorithm. Notice that the word *algorithm* is derived from the name of the Muslim mathematician Al-Khawarizmi.

Ill-defined problems are those that cannot be properly defined, resulting in complex or even infeasible algorithms. Ill-defined problems are typically ambiguous and/or lack information. Consider, for instance, the problem of computing the *fair wage* of an employee after commissions deductions (such as income tax). The word fair is ambiguous, and the income tax rate has not been defined. In addition, creativity problems are typically ill-defined. Just consider how to build an algorithm that draws a beautiful picture.

Example 1.4 : the maximum value problem

Consider a set of values $a_1, a_2, ..., a_n$, all of them are integers (i.e., $\forall i : a_i \in \mathbb{N}$). We would like to find the maximum value of these values. So we identify the input and the output as:

input: A set of values $a_1, a_2, ..., a_n$ such that $\forall i : a_i \in \mathbb{N}$.

output: A value m among the values $a_1, a_2, ..., a_n$ such that $\forall i : a_i \leq m$.

The algorithm should define a series of processing steps that allows computing m. If the algorithm is given the values $4, 8, 0, 5, 1$ (we call it an *instance* of the problem), then the output must be $m = 8$. However, the algorithm should not just work in this particular example; it should get the correct result for any set of values.

Many questions arise when dealing with problems and algorithms. Of course, we are just dealing here with well-specified problems.

- **Can any problem be solved with an algorithm?** The answer is no. Some problems have no algorithms to solve them; they are called *undecidable problems*. Consider a well-specified problem and a algorithm for solving it (at least in certain situations). If the problem has an algorithm that solves every instance in a finite amount of steps, then it is *decidable*.
- **If a problem is decidable, how can we create an algorithm to solve it?** Problem-solving is a challenging field. Algorithms are sometimes easy to find, yet they may be extremely hard to design for others. Many approaches have been established to help programmers in writing their programs (such as problem decomposition, divide and conquer, and so on).
- **At what cost does an algorithm solve a problem?** Algorithms are generally executed on computer systems (via programming languages). To execute a program, resources are required. For example, if someone does not own a computer, he should rent one in order to run his programs. Resource consumption has a cost that should be reduced, as it typically requires money and labor. This is referred to as *algorithmic complexity*. In general, the most essential cost to be reduced is time. However, many other criteria could and/or should be considered to define costs (what are they?)

2.1. Examples of problems

Real world problems can often be solved by algorithms, while others are unsolvable. We will examine some examples.

Example 1.5 : Security of e-commerce

E-commerce is the use of the Internet for making transactions involving products. Security is one of the most crucial requirements, among many others. Encrypting algorithms which transform information into unreadable

form provide general privacy protection. Decryption algorithms change the unreadable form back into information if and only if the right credentials are provided. This is a decidable problem.

Example 1.6 : Production management

A company produces goods that are sold with certain profit. Producing goods need raw materials, labor, and manufacturing machinery that require energy and maintenance. The company wants to know how many products will be manufactured in order to reduce costs while maximizing profits. This is a decidable problem.

Example 1.7 : The halting problem

The CPU doesn't understand programming languages intrinsically. To achieve this, *compilers* are employed. A compiler is a program that analyzes other programs and translates them into a format that the CPU understands. Suppose we wish to create an algorithm that determines if programs can stop (they don't run indefinitely). There is no such algorithm since the halting problem is undecidable.

2.2. Data structures

Algorithms are procedures that consumes data to produce data. Computers have to understand processed data before they can perform computations on it. Furthermore, because memory is always limited, algorithms should explicitly declare the amount of memory to allocate in order to store data.

Data coding refers to how data is represented in binary format. For example, integer numbers are not encoded the same way as text. The efficiency of algorithms is determined by a variety of factors, one of which is the organization of data. Therefore, the first step in using computers to solve problems is to define the data structure, or how the data is organized. It should be noted that each problem has its own set of data structures; nonetheless, some structures are commonly used.