Computer Science the study of *algorithms*, including

- Their formal and mathematical properties
- Their hardware realizations
- Their linguistic realizations
- Their applications

N. Gibbs and A. Tucker. "A Model Curriculum for a Liberal Arts Degree in Computer Science"; CACM 29, No 3 (March 1986)

So what is an *algorithm*?

**Definition of Algorithm** (after Al Kho-war-iz-mi a 9th century Persian mathematician) - an *ordered sequence* of *unambiguous and well-defined instructions* that *performs some task* and *halts in finite time*

Let’s examine the four parts of this definition more closely

1. an *ordered sequence* means that you can number the steps (it's socks then shoes!)
2. *unambiguous and well-defined instructions* means that each instruction is clear and do-able
3. *performs some task*
4. *halts in finite time* (algorithms terminate!)

In other words, an algorithm is like a *recipe*.

Algorithms can be executed by a computing agent which is not necessarily a *computer*. A computer program is an implementation of an algorithm, “*a detailed step-by-step set of instructions telling a computer exactly what to do*” (Zelle p. 2); a computer is an electronic device for executing an algorithm.

A *computer* is an electronic device, operating under the control of *instructions stored in its own memory unit*, that

1. accepts data (input),
2. processes data arithmetically and logically,
3. produces output (information) from the processing, and/or
4. stores the results for future use.

The phrase *instructions stored in its own memory unit* is the *stored program concept*. Because a computer makes no distinction between data and code a computer has the ability to *modify its own programming*. This is what makes the computer so versatile and powerful.
The computer is also a *universal machine* in the sense that it is a general-purpose symbol-manipulating machine capable of executing any sequence of instructions (a program) stored in its main memory.

“A modern computer can be defined as a ‘*machine that stores and manipulates information under the control of a changeable program*’” (Zelle p. 1)

The above definition(s) for a computer naturally leads to the physical make-up or *hardware* of a computer.

The **Hardware Components** of a computer consist of

1. Input Device(s): keyboard and/or mouse.
2. Central Processing Unit (CPU) which consists of two sub-units  
   - program control unit which fetches and executes instructions  
   - arithmetic-logic unit (ALU) which performs arithmetic and logical operations
3. Main Memory which stores both the *code* and *data* for the current program
4. Output Device(s): monitor screen and/or printer
5. Secondary Storage device(s): hard drives (disks), floppy drives (diskettes), jump drives, CD’s, DVD’s and/or network drives. Note that these are really I/O devices

Note how the hardware components of a computer parallel the definition of a computer given above. Of course by itself *hardware* is nothing without the *software* to drive it.

![Functional View of a Von Neumann Computer](image-url)
Another view of the computer is the system bus model. Here the components of the computer are seen as connected by a bus (a common carrier). Note the data lines, address lines and control lines. Input, Output, and Secondary Storage devices are seen simply as generic I/O (input/output) devices attached to the bus. Note that some of the address bus arrows are one-way.

The Importance of Binary

At the lowest level, a computer is constructed out of (inexpensive) bi-stable electronic components, components that maintain one of two states (off or on) which is represented by 0 or 1. The 0/1 is called a \textit{bit} which is a contraction of \textit{binary digit}. This affects everything about computers.

Main Memory

The memory of a computer is organized as a \textit{numbered list of fixed-sized} locations or "cells". By \textit{numbered list} we mean each cell in memory is identified by a numeric address. Memory is \textit{addressable} in that to access the contents of a cell, you must know its address. By \textit{fixed size} we mean each cell contained the same number of \textit{bits}. That is

1. Data and code is stored in binary representation; as 0’s and 1’s.
2. Main memory is partitioned into fixed length units called \textit{memory cells} (the generic name). Each memory cell is associated with a unique (numeric) identifier called its \textit{address}.
3. Memory cell sizes have been 6, 8, 12, 18, 24, 32, 40, 48, and 60 \textit{bits} long. Today the standard memory cell is an 8-bit cell size called a \textit{byte}. A \textit{word} is usually 4 bytes long (32-bits), a word being the amount of memory used to store an integer. Early micro-computers used a smaller 16-bit, 2 byte word size.
4. The \textit{address space} is the maximum number of possible cell addresses (hence the maximum size of memory). An n-bit address field generates $2^n$ addresses. 32-bit addressing allows memory sizes of up to 4 Gigabytes (a \textit{Gigabyte} is approximately 1 billion bytes)
5. Accesses to memory are done by specifying the *address*, not the *contents*. A memory access either fetches (reads from memory) or stores (writes to memory) the *complete contents* of a memory cell; you can’t access just part of a memory cell. A memory cell is the smallest unit of information that can be fetched from or stored to memory.

6. As noted above, a distinction is made between the *address* of a cell and its *contents* (very much like the distinction between the *variable* x and the *value* of the variable x).

7. When using a *bus* to connect memory to the CPU, there are separate *Data, Address* and *Control* lines. Access time is the *same* for all cells. Today access time is very fast, anywhere from 50 to 200 nanoseconds (one nanosecond equals 10 to the minus 9 seconds or 1 billionth of a second).

8. Physically memory can be visualized as a numbered list of cells, similar to an array or list where each component or entry is uniquely identified by its position (number) i.e. the address in the list.

```
0  ...  
1  ...  
2  ...  
3  ...  
4  ...  
...  ...  
...  ...  
...  ...  
```

**Cache Memory:** Despite the fast speed of memory access, the CPU (Central Processing Unit) is faster. Faster memory is possible but expensive. Cache memory is a small but faster (5 - 10 times faster) memory *interposed* between main memory and the CPU. When the CPU fetches the contents of a memory address from main memory, the contents and the contents of the surrounding cells are copied to cache memory (called a *cache line*). Thereafter if the CPU tries to access that same cell (or a cell close by), it can obtain the contents from the faster cache memory. The reason this works due to the *Principle of Locality* which is the observation that when a computer accesses a memory location the probability is high that in the near future it will access that memory location again (temporal locality) or a memory location which is nearby (spatial locality). Memory accesses are done in *parallel* to both the main memory and the cache. If the access is "satisfied out of cache", the slower main memory access is aborted. **Cache hit ratio** is the fraction of memory accesses satisfied out of cache. A cache hit ratio of 90% or above is not uncommon.

**Computer Software**

It’s all about software. Since a computer operates under the control of *instructions stored in its memory unit*, (stored program concept), a computer is only as good as the software that controls it.
Recall: At the lowest level, a computer is constructed out of bi-stable electronic components, components that maintain one of two states (off or on) which is represented by 0 or 1. The 0/1 is called a *bit* which is a contraction of *binary digit*. All instructions (and data) that reside in memory are in binary. This machine code is very low level and is very specific to that particular type of CPU (so a Mac G4 CPU understands a different binary code from an Intel CPU). Note that Macs and Window machines also run under different operating systems - another kind of difference. There are at least three levels of computer languages: machine code, assembler, high level languages.

**Machine Code**

Binary machine code has three problems

1. It’s written in binary, 0’s and 1’s, so it’s very hard to read and write
2. The instructions are very low level so writing any kind of moderately complicated application is both complicated and time consuming.
3. Code that runs on one type of CPU will not run on a different type of CPU. We say the code is not *portable*.

**Assembler Language**

Assembler language is mnemonic codes for machine codes. For example instead of using the Intel 80x86 machine code `00000001 11011000` we use `add ax, a`; that is, add the contents at address a to the contents of the AX register.

1. Assembler language is easier to program than machine code
2. It’s easy to write an Assembler, a program which translates assembler language to machine code.
3. Assembler language is logically no more powerful than machine code; its instructions are still very low level so writing any kind of moderately complicated application is still complicated and time consuming. Like machine code assembler language is “machine oriented”, not “human oriented”
4. The assembler language code is not portable between different types of CPU’s.

**High Level Languages**  
a.k.a. High Order Languages or HOL’s

A high level language is *procedure oriented* meaning that you code the procedure used to solve the problem. An algorithm is coded in a high level language which is “human friendly” then translated by a special computer program (either a *compiler* or an *interpreter*) to machine code.

A *compiler* translates the high level language code (called the *source* code) completely into machine code (called *object* code). Then the object code is executed by the computer.

An *interpreter* translates one source code statement into machine code then executes it. This is repeated.
With a compiler you make the translation first then execute the object code; with an interpreter you interleave translation and execution. With a compiler you pay the translation cost “up front” and only once. Thereafter you can execute the resulting object code as many times you wish with no additional cost. With an interpreter every time you re-run the program you have to re-translate each line. However an interpreter (a translation program which is easier to write) is more flexible, easier to work with, but slower. Python is interpreted.

1. It’s easier to write a program in a high-level language
2. Given the omnipresence of translation programs (a compiler/interpreter must be written for each different type of CPU) high level language programs are portable.

Some other terms used when discussion programming languages

Semantic Gap: the distance between the low level machine code executed by a computer and the high level languages humans used to code/express algorithms. In 1956 FORTRAN (the first high level language) showed that efficient language translators could be written to bridge this gap. Since then we’ve learned enough about compiler writing to be able to teach it in college courses.

Syntax: the structure of a language (not to be confused with a tax on alcohol and tobacco); think of diagramming a sentence: subject – verb – predicate. Computers are very sensitive to syntax

Semantics: the meaning of a language. Computers don’t think in the way humans do so any meaning we attach to a program makes no sense to a computer. For example “distance equals rate times time” has meaning to us. All that a computer sees is \( c = a \times b \). So while the computer has no problem with \( 500 = -50 \times -10 \) we might balk at the idea that if you travel at minus 50 mph for minus 10 hours (assuming you go backward in time), you have traveled 500 miles.

Steps to write compile and execute a computer program

1. Using a text editor create the source code file
2. Using a compiler translate your source code file into object code. If there are syntax errors found by the compiler, go back to step 1 and correct them
3. Using a linker link any other object code modules needed by your object code. This creates an executable file (denoted by an .exe extension). Many language compilers provide libraries of subroutines that your program can use. Why re-invent the wheel? A linker links these subroutines into your code.
4. Using a loader (supplied by the operating system) load your executable code into main memory. It has to be in main memory to run.
5. Run your code (the operating system gives control the computer to your program).

OR

Use an Integrated Development Environment (IDE) which handles/incorporates all the overhead listed above. This is what Python does except it’s called IDLE.py after Eric Idle one of the founding members of Monty Python’s Flying Circus.