# Some Basic Computer-Science Concepts for Data Scientists

Aris Anagnostopoulos

October 5, 2022

In these notes we describe some very basic computer-science concepts. They are just only a small part of what we have said in class, and they may contain some additional information. Depending on my available time, I may update these notes by adding more material. I am sure that there are a lot of errors, so I welcome any errors that you find, or any other comments that you may have.

## 1 Binary and Hexadecimal Number Systems

In everyday life, we represent numbers in base 10. For example the number 3051 is the value

$$1 \cdot 10^0 + 5 \cdot 10^1 + 0 \cdot 10^2 + 3 \cdot 10^3.$$

As in computers we use bits, numbers are represented in the *binary system*. For example:

$$(01101101)_2 = 2^0 + 2^2 + 2^3 + 2^5 + 2^6 = 109.$$

(I put the subscript 2 in the left to make explicit that it is a binary number.) This is how we can convert a number from binary to decimal. To convert number from decimal to binary there are a couple of ways, for example, you can do it by keep dividing the decimal number by 2 and looking at the remainder; search online for the details.

When humans need to work with RAM addresses (rare if you program in Python) the decimal system is not very convenient and the binary is hard to read. Then we use the *hexadecimal system*, which has 16 digits: 0 to 9 and the letters $A$ to $F$, with $A$ corresponding to 10, $B$ to 11, and so on. Then we have:

$$(F07D)_{16} = 14 \cdot 16^0 + 7 \cdot 16^1 + 15 \cdot 16^3 = 61566.$$

The hexadecimal system allows to easily convert from binary to hexadecimal and vice versa. Each hexadecimal digit corresponds to 4 binary digits; see Table 1. Using the table, we can easily convert between two binary and hexadecimal numbers.

$$(0110111010110001)_2 = (0110\ 1110\ 1011\ 0001)_2 = (6EB1)_{16} = 6\text{EB1}\,\text{h}.$$

As you can see, sometimes we add the letter "h" to indicate that a number is written in hexadecimal. In Python, to represent a binary number we preceed it by `0b` (or `0B`), and a hexadecimal by `0x` (or `0X`). For example the command `y=0xABCD` assigns to $y$ the value 43981.

## 2 Representation of Information

Numbers, characters, and so on are represented in memory as a sequence of *bits* (see the binary number system in Section 1). Each bit can be either 0 or 1. A sequence of 8 bits is called a *byte*. With a byte we can represent $2^8 = 256$ different numbers. A byte was originally used to represent one character but now

| Hexadecimal | Binary | Hexadecimal | Binary |
|---|---|---|---|
| 0 | 0000 | 8 | 1000 |
| 1 | 0001 | 9 | 1001 |
| 2 | 0010 | A | 1010 |
| 3 | 0011 | B | 1011 |
| 4 | 0100 | C | 1100 |
| 5 | 0101 | D | 1101 |
| 6 | 0110 | E | 1110 |
| 7 | 0111 | F | 1111 |

Table 1: Correspondence between digits in the hexadecimal and the binary number systems.

the unicode standard uses more than one bytes per character. A *word* is a sequence of $x$ bits, where $x$ is the length of the numbers that the processor works with, so it depends on the architecture of the processor. Typical values of $x$ are 16, 32, 64, and most processors today have $x = 64$ (8 bytes).

Let us now see how information is represented in memory without entering into details:

**Unsigned integers:** Representing nonnegative integers is straightforward: we use the binary representation. For example, 13 in decimal is represented as 1101 in binary; using Python's syntax, we can write $10 = 0b1101$. Similarly, $143 = 0b10001111$. With $n$ bits we can represent $2^n$ different numbers. Therefore, with 1 byte we can represent numbers from 0 to 255, with 4 bytes 0 to 65,535, and with 8 bytes from 0 to $2^{64} - 1 = 18,446,744,073,709,551,615$.

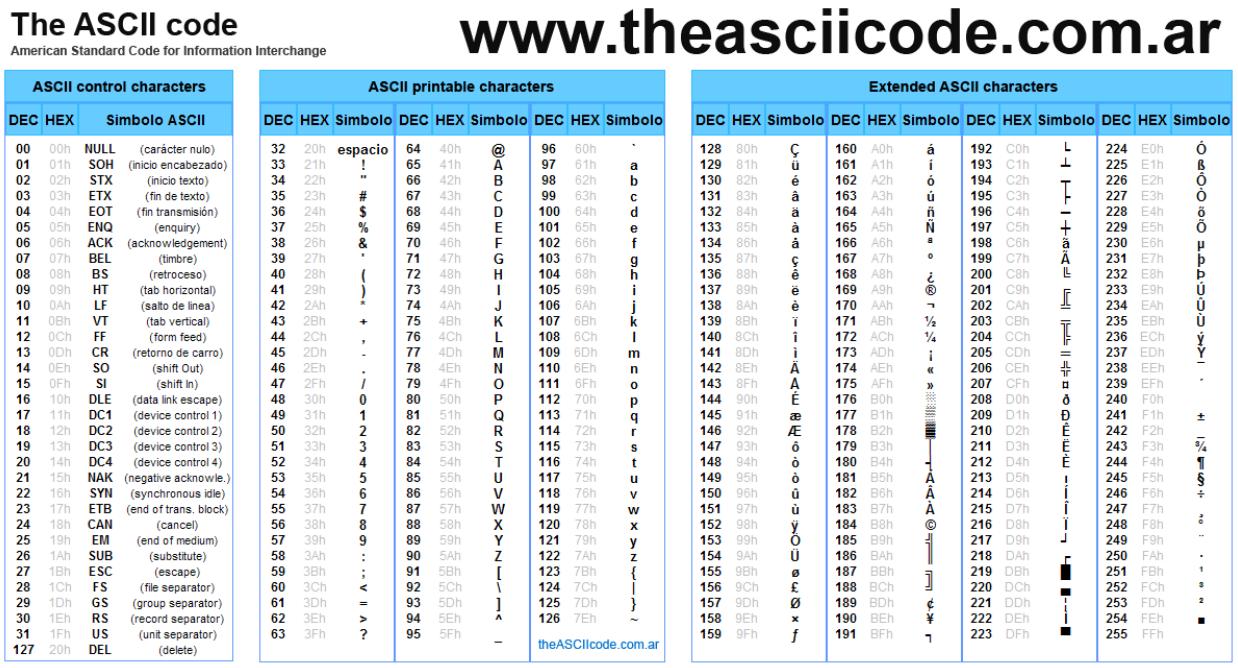
**(Signed) integers:** To be able to represent both positive and negative numbers, we use the leftmost bit as the sign of the number. Assume that we use 8bits to represent numbers, the 0000 0000 to 0111 1111 are all positive numbers (and zero) and 1000 0000 to 1111 1111 are all negative (the space in between is only for readability). With $n$ bits we can represent numbers from $-2^{n-1}$ up to $2^{n-1} - 1$. For positive numbers we use the binary representation of the number, as we did with unsigned integers. For negative numbers we use what is called the *two's complement representation*. In the two's complement representation, if we use 8 bits, 1000 0000 represents -128, 1000 0001 is -127, up to 1111 1111 which is -1. We will not explain here the logic behind two's complement representation and why we use it, it has to do with the fact that it allows the processor to perform easily addition and subtraction between positive and negative numbers.[1]

**Floating point:** Of course we cannot represent real numbers, which may have infinite number of digits. However we can represent fractional numbers up to some precision, using the *floating-point representation*. To represent them, we use some bits to represent the size of the number and some bits to represent the value. We will not enter into details about how we do that[2] but we will mention that if we use more bits we can represent numbers with higher absolute value and with higher precision. So, for example, with 32 bits we have *single precision* and with 64 bits we have *double precision*.

**Characters:** Characters are also represented as sequence of bits. The most common standard to represent characters was the ASCII encoding, where each character was represented as one byte and there was a mapping from each byte to a symbol (see Figure 1). This allowed to represent 256 different characters, which was enough to represent upper and lower english characters, characters with accents, numbers, punctuation points and in addition some "control characters" and some other symbols. You may still come across the ASCII endocing when working with text. However 256 slots were not enough for representing characters in all languages in the world, so recently we have started using the *Unicode* standard, where each character

[1] Check `https://en.wikipedia.org/wiki/Two's_complement` if you are interested.

[2] Check `https://en.wikipedia.org/wiki/Floating-point_arithmetic`.



**The ASCII code**

American Standard Code for Information Interchange

**www.theasciicode.com.ar**

**ASCII control characters**

| DEC | HEX | Simbolo ASCII | |
|---|---|---|---|
| 00 | 00h | NULL | (carácter nulo) |
| 01 | 01h | SOH | (inicio encabezado) |
| 02 | 02h | STX | (inicio texto) |
| 03 | 03h | ETX | (fin de texto) |
| 04 | 04h | EOT | (fin transmisión) |
| 05 | 05h | ENQ | (enquiry) |
| 06 | 06h | ACK | (acknowledgement) |
| 07 | 07h | BEL | (timbre) |
| 08 | 08h | BS | (retroceso) |
| 09 | 09h | HT | (tab horizontal) |
| 10 | 0Ah | LF | (salto de linea) |
| 11 | 0Bh | VT | (tab vertical) |
| 12 | 0Ch | FF | (form feed) |
| 13 | 0Dh | CR | (retorno de carro) |
| 14 | 0Eh | SO | (shift Out) |
| 15 | 0Fh | SI | (shift In) |
| 16 | 10h | DLE | (data link escape) |
| 17 | 11h | DC1 | (device control 1) |
| 18 | 12h | DC2 | (device control 2) |
| 19 | 13h | DC3 | (device control 3) |
| 20 | 14h | DC4 | (device control 4) |
| 21 | 15h | NAK | (negative acknowle.) |
| 22 | 16h | SYN | (synchronous idle) |
| 23 | 17h | ETB | (end of trans. block) |
| 24 | 18h | CAN | (cancel) |
| 25 | 19h | EM | (end of medium) |
| 26 | 1Ah | SUB | (substitute) |
| 27 | 1Bh | ESC | (escape) |
| 28 | 1Ch | FS | (file separator) |
| 29 | 1Dh | GS | (group separator) |
| 30 | 1Eh | RS | (record separator) |
| 31 | 1Fh | US | (unit separator) |
| 127 | 20h | DEL | (delete) |

**ASCII printable characters**

| DEC | HEX | Simbolo | DEC | HEX | Simbolo | DEC | HEX | Simbolo |
|---|---|---|---|---|---|---|---|---|
| 32 | 20h | espacio | 64 | 40h | @ | 96 | 60h | ` |
| 33 | 21h | ! | 65 | 41h | A | 97 | 61h | a |
| 34 | 22h | " | 66 | 42h | B | 98 | 62h | b |
| 35 | 23h | # | 67 | 43h | C | 99 | 63h | c |
| 36 | 24h | $ | 68 | 44h | D | 100 | 64h | d |
| 37 | 25h | % | 69 | 45h | E | 101 | 65h | e |
| 38 | 26h | & | 70 | 46h | F | 102 | 66h | f |
| 39 | 27h | ' | 71 | 47h | G | 103 | 67h | g |
| 40 | 28h | ( | 72 | 48h | H | 104 | 68h | h |
| 41 | 29h | ) | 73 | 49h | I | 105 | 69h | i |
| 42 | 2Ah | * | 74 | 4Ah | J | 106 | 6Ah | j |
| 43 | 2Bh | + | 75 | 4Bh | K | 107 | 6Bh | k |
| 44 | 2Ch | , | 76 | 4Ch | L | 108 | 6Ch | l |
| 45 | 2Dh | - | 77 | 4Dh | M | 109 | 6Dh | m |
| 46 | 2Eh | . | 78 | 4Eh | N | 110 | 6Eh | n |
| 47 | 2Fh | / | 79 | 4Fh | O | 111 | 6Fh | o |
| 48 | 30h | 0 | 80 | 50h | P | 112 | 70h | p |
| 49 | 31h | 1 | 81 | 51h | Q | 113 | 71h | q |
| 50 | 32h | 2 | 82 | 52h | R | 114 | 72h | r |
| 51 | 33h | 3 | 83 | 53h | S | 115 | 73h | s |
| 52 | 34h | 4 | 84 | 54h | T | 116 | 74h | t |
| 53 | 35h | 5 | 85 | 55h | U | 117 | 75h | u |
| 54 | 36h | 6 | 86 | 56h | V | 118 | 76h | v |
| 55 | 37h | 7 | 87 | 57h | W | 119 | 77h | w |
| 56 | 38h | 8 | 88 | 58h | X | 120 | 78h | x |
| 57 | 39h | 9 | 89 | 59h | Y | 121 | 79h | y |
| 58 | 3Ah | : | 90 | 5Ah | Z | 122 | 7Ah | z |
| 59 | 3Bh | ; | 91 | 5Bh | [ | 123 | 7Bh | { |
| 60 | 3Ch | < | 92 | 5Ch | \ | 124 | 7Ch | \| |
| 61 | 3Dh | = | 93 | 5Dh | ] | 125 | 7Dh | } |
| 62 | 3Eh | > | 94 | 5Eh | ^ | 126 | 7Eh | ~ |
| 63 | 3Fh | ? | 95 | 5Fh | _ | | | |

theASCIIcode.com.ar

**Extended ASCII characters**

| DEC | HEX | Simbolo | DEC | HEX | Simbolo | DEC | HEX | Simbolo | DEC | HEX | Simbolo |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 128 | 80h | Ç | 160 | A0h | á | 192 | C0h | └ | 224 | E0h | Ó |
| 129 | 81h | ü | 161 | A1h | í | 193 | C1h | ┴ | 225 | E1h | ß |
| 130 | 82h | é | 162 | A2h | ó | 194 | C2h | ┬ | 226 | E2h | Ô |
| 131 | 83h | â | 163 | A3h | ú | 195 | C3h | ├ | 227 | E3h | Ò |
| 132 | 84h | ä | 164 | A4h | ñ | 196 | C4h | ─ | 228 | E4h | õ |
| 133 | 85h | à | 165 | A5h | Ñ | 197 | C5h | ┼ | 229 | E5h | Õ |
| 134 | 86h | å | 166 | A6h | ª | 198 | C6h | ã | 230 | E6h | µ |
| 135 | 87h | ç | 167 | A7h | º | 199 | C7h | Ã | 231 | E7h | þ |
| 136 | 88h | ê | 168 | A8h | ¿ | 200 | C8h | ╚ | 232 | E8h | Þ |
| 137 | 89h | ë | 169 | A9h | ® | 201 | C9h | ╔ | 233 | E9h | Ú |
| 138 | 8Ah | è | 170 | AAh | ¬ | 202 | CAh | ╩ | 234 | EAh | Û |
| 139 | 8Bh | ï | 171 | ABh | ½ | 203 | CBh | ╦ | 235 | EBh | Ù |
| 140 | 8Ch | î | 172 | ACh | ¼ | 204 | CCh | ╠ | 236 | ECh | ý |
| 141 | 8Dh | ì | 173 | ADh | ¡ | 205 | CDh | ═ | 237 | EDh | Ý |
| 142 | 8Eh | Ä | 174 | AEh | « | 206 | CEh | ╬ | 238 | EEh | ¯ |
| 143 | 8Fh | Å | 175 | AFh | » | 207 | CFh | ¤ | 239 | EFh | ´ |
| 144 | 90h | É | 176 | B0h | ░ | 208 | D0h | ð | 240 | F0h | |
| 145 | 91h | æ | 177 | B1h | ▒ | 209 | D1h | Ð | 241 | F1h | ± |
| 146 | 92h | Æ | 178 | B2h | ▓ | 210 | D2h | Ê | 242 | F2h | ‗ |
| 147 | 93h | ô | 179 | B3h | │ | 211 | D3h | Ë | 243 | F3h | ¾ |
| 148 | 94h | ò | 180 | B4h | ┤ | 212 | D4h | È | 244 | F4h | ¶ |
| 149 | 95h | ò | 181 | B5h | Á | 213 | D5h | ı | 245 | F5h | § |
| 150 | 96h | û | 182 | B6h | Â | 214 | D6h | Í | 246 | F6h | ÷ |
| 151 | 97h | ù | 183 | B7h | À | 215 | D7h | Î | 247 | F7h | ¸ |
| 152 | 98h | ÿ | 184 | B8h | © | 216 | D8h | Ï | 248 | F8h | ° |
| 153 | 99h | Ö | 185 | B9h | ╣ | 217 | D9h | ┘ | 249 | F9h | ¨ |
| 154 | 9Ah | Ü | 186 | BAh | ║ | 218 | DAh | ┌ | 250 | FAh | · |
| 155 | 9Bh | ø | 187 | BBh | ╗ | 219 | DBh | █ | 251 | FBh | ¹ |
| 156 | 9Ch | £ | 188 | BCh | ╝ | 220 | DCh | ▄ | 252 | FCh | ³ |
| 157 | 9Dh | Ø | 189 | BDh | ¢ | 221 | DDh | ¦ | 253 | FDh | ² |
| 158 | 9Eh | × | 190 | BEh | ¥ | 222 | DEh | Ì | 254 | FEh | ■ |
| 159 | 9Fh | ƒ | 191 | BFh | ┐ | 223 | DFh | ▀ | 255 | FFh | |

Figure 1: The ASCII table. For example, "L" is represented as $76 = 4\mathrm{C}\,\mathrm{h}$.

can be represented with more than one byte.[3] In Unicode its character has a code and then some *encoding scheme* is used to represent it. Python 3 uses Unicode and it supports a variety of encoding schemes, such as UTF-8 and UTF-16.[4]

# 3 Super Basic Computer Architecture

A computer has multiple components required to function. Here we give a high-level description of the basic ones, which is required to understand the material of this text.

**Central processing unit (CPU):** The CPU (or simply processor), is the component that performs all the arithmetic and logical operations. It reads information from the memory into its internal variables called *registers* and it performs the operations on these registers. A processor is characterized by the dimension of its registers, which determine the magnitude of numbers that it performs operation on and the dimension of the memory that it processor can address. Thus we talk about processors of 16 bits, 32 bits, and 64 bits, with 64-bit being the most common nowadays (both on PCs and on smartphones). Another important characteristic is the frequency it runs at (e.g., the Intel® Core i7-9700K Processor has a speed of 3.60 GHz). Over the years, processors have had multiple enhancements, for example, multiple *cores*, which allow to perform operations in parallel and *cache* which is some small memory inside the CPU chip, which allows to load information from the main memory and then work on it fast without having to fetch the data from the main memory for every single operation.

[3] You can find a nice article describing Unicode at a high level at `https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses/`.

[4] A quick article on the topic can be found at `https://realpython.com/python-encodings-guide/`.

**Internal memory (RAM):** Random-access memory (RAM), often called *main memory* or just *memory* is where the data that the processor operates on reside. When a program is loaded, then its instructions are loaded into the memory. Variables also created during the execution of the program are also in memory. Then the processor reads the instructions from the memory and executes them, operating on data that it also reads from the memory. The main memory is also temporary: data are expected to be replaced when they are not needed any more and data are lost once the computer is switched off. The main memory is limited (typically some gigabytes), so the operating system and programs often need to resort to external memory while they are running.

**External memory:** External memory (typically hard disks or hard drives) are being used to store permenently information. It is also used to store data while a program is executed, if there is not enough space in the RAM. External memory is much larger than RAM and it is a permanent storage (i.e., data do not get erased when the computer power is off) but it is much slower. Therefore, programs should try to use it as little as possible for temporal storage. Traditional hard disks are based on magnetic metal disks (hence the name), however newer hard drives are based on solid-state technology and are much faster.

### 3.1 A View of Memory

Assume that we have a 64-bit CPU. This means that in the CPU registers can store numbers up to $2^{64} - 1$, so when the CPU uses registers to access memory it can access up to $2^{64}$ locations, from 0 h up to FFFFFFFFFFFFFFFF h; see Figure 2(a). This means that, theoretically, a 64-bit CPU could support RAM of size up to $2^{64}$ bytes, but it is not needed, so most processors support less (e.g., $2^{52}$ bytes = 4 petabytes).

Recall that bytes can be grouped into words (Figure 2(b)). Typically the values are not stored anywhere in RAM, but they are *aligned*. This means that if we want to store some value, we will store it in memory locations that are multiples of the word size (number of bytes per word, 8 in Figure 2). Thus, often we represent memory as a collection of words, as in Figure 2(c), and memory addresses will be multiples of the word size. Of course, if we want to store values that occupy less than 8 bytes, we also write inside a word; for example we can use one 8-byte word to represent 8 ASCII characters. Fortunately, the programming language takes care of all these issues.

## 4 Python Types and Memory

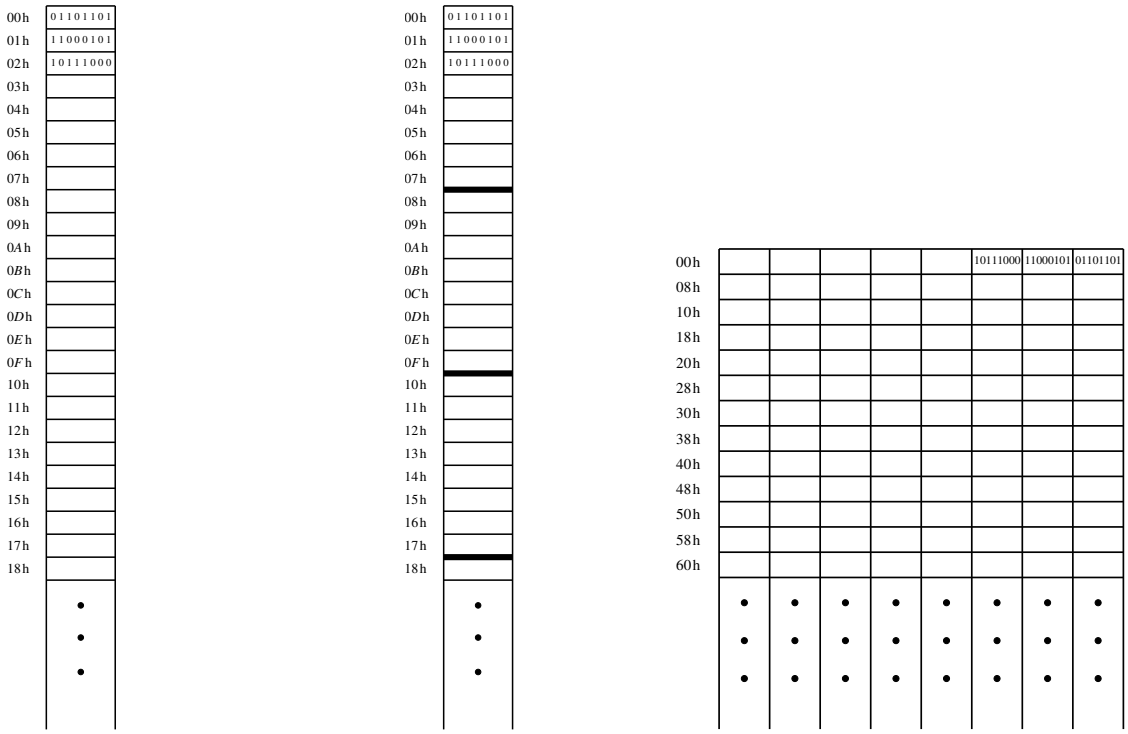
We are familiar with different types of values, which we can assign to a variable; for example, integer, floating point, string, tupple, list, dictionary, and so on. When we assign a value then Python creates an object. An *object* consists of the data that it contains, along with additional information that Python uses to handle these data. For example the object for the list `[1, 2, -9]` contains the numbers 1, 2, and −9 as well as information about the length of the list (three in this case), and information about functions that are tied to this object (e.g., `append(8)`, which will add 8 to the list).

Therefore, an object that we create has a given type. Python has some *built-in* types (e.g., integer, string, etc.) and we can also create new types by using *classes*.

In Table 2 we see the most common built-in types that Python has, along with the name that Python uses for that type.

Let us see now in more detail, how Python stores these data in memory. When a Python (or some other language) program is executed it uses the memory (1) to store its code, which will be executed, (2) to store information required while the code is being executed, and (3) to store variables.

Briefly, the first part refers to the commands that the programmer has written (e.g., `if`, `for`, `print` statements) as well as code stored in the libraries that the program uses. The CPU reads the code from the memory and executes it. The second part refers to information that is being stored while the program runs and is required to run properly. For example, it uses it to store the arguments that are passed when a function is called. The third part refers to the memory that is created when the programmer defines some




(a) A diagram of computer memory. Each locations stores a byte.

(b) The same memory viewed with the words separated.

(c) The same memory but viewed with the words aligned.

Figure 2: Three views of the memory of a 64-bit CPU architecture: just as a seaquence of bytes, separated into words, and aligned into words. Words have length of 8 bytes.

| Name | Python name |
|---|---|
| Integer | `int` |
| Floating point | `float` |
| Boolean | `bool` |
| Tuple | `tuple` |
| List or array | `list` |
| String | `str` |
| Dictionary | `dict` |
| Set | `set` |

Table 2: Some of them most used Python built-in data types.

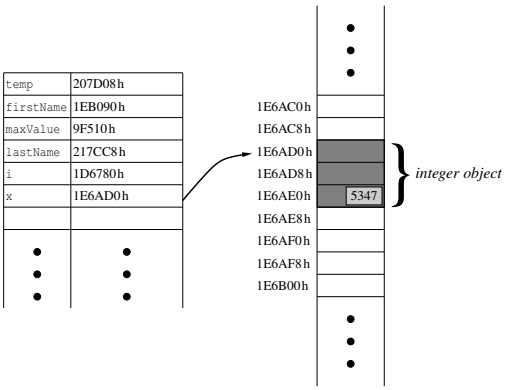



Figure 3: In the left we see the dictionary mapping the variable names to memory locations and on the right a part of the memory. When we do the assignment `x = 5347`, Python first creates an integer object and stores it into some location in the memory. In this example it occupies the locations 1E6AD0 h up to 1E6AE7 h (note that the last word consists of the bytes at 1E6AE0 h to 1E6AE7 h). The integer object contains internal information needed to Python, as well as the number 5347. Of course, the dictionary is also stored in the memory in some other location.

variables. The rest of this section refers to this last part, which is important to understand because it will allow us to do more informed choices when we use variables.

First of all, to handle variables, Python creates a type of a dictionary where each dictionary key is a variable name, and the dictionary value is the memory address where the value assigned to the variable is stored. Actually, Python has more than one such dictionaries, one for the global variables (variables defined outside all functions) and one for each new fuction that is called. In this way, a variable `x` used inside a function is different from a variable `x` defined in another function or a global variable `x`. This is captured by the concept of *namespace* and we will not discuss about it here. For the rest, let us assume that we talk about global variables, variables defined outside any function.

We now consider some types and we see how Python stores them internally. Knowing this, will allow us to understand whether an operation that we perform can be performed fast or not, and make the right choice of how to store the data. I want to make two comments first:

- The exact details depend on the particular Python interpreter[5] and actually change over time as the language evolves. For example, the exact way that information is stored in Python 2.7 may be different from that of Python 3.8. Nevertheless, the high-level procedure is typically the same.
- Most of what we say does not hold only for Python, but also for most other programming languages. The biggest difference of Python from many other languages is that it represents integers and other basic data types as objects. We discuss later about this when we talk about lists.

**Integers.** Consider a Python assignment such as `x = 5347`. This command does two things.[6] See Figure 3.

1. It creates an *integer object* at some free location in the memory, where it stores the value 5347. Except for this value, the integer object has some more internal information that are useful to Python.
2. In the dictionary with the variable names, it stores as value the address where the integer object is located.

[5]The *interpreter* is the program that reads a Python program and converts it into instructions for the CPU.

[6]Things are a bit different for small integer values ($\{-5, \ldots, 256\}$), for which Python tries to be more efficient, but let us ignore this for this discussion. They are also different for very large integer values, where it uses more space.

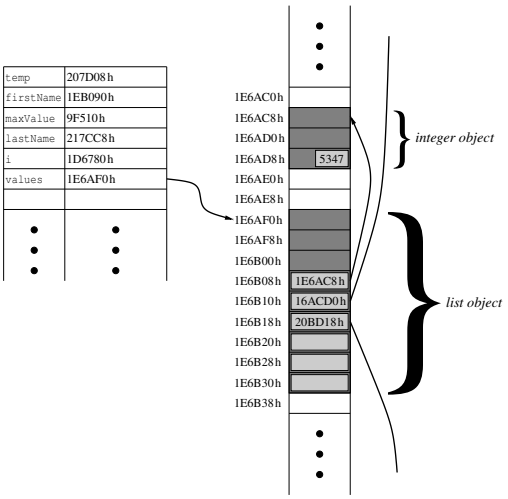



Figure 4: In the left we see the dictionary mapping the variable names to memory locations and on the right a part of the memory. When we do the assignment `values = [5347, 3971, 510]`, Python creates a list object and stores it into some location in the memory. In this example it occupies the locations 1E6AC8 h up to 1E6B37 h (note that the last word consists of the bytes at 1E6B30 h to 1E6B37 h). The list object contains internal information needed to Python (e.g., the size, 3), as well as some memory locations where the numbers are stored. For instance, the memory location 1E6B08 h contains the first element of `values`: it has stored the address (1E6AC8 h) of the (newly created) integer object where the value 5347 is stored. The list also contains three empty locations, in case we decide to append some elements.

**Floats, booleans, tuples, and strings.** These types are stored in memory in a way similar to integers. For example, when we assign a tuple to a variable, Python will create a *tuple object*, it will store in it the values in the tuple, and it will store in the dictionary of variables the memory address of the newly created tubple Although each of these types has its own details about how it is stored in memory (strings especially), the high-level procedure is the same.

**Lists.** Consider a Python assignment such as `values = [5347, 3971, 510]` See Figure 4. Python first creates and stores in memory integer objects for each of the three numbers. It then creates e *list object*, and it stores the memory addresses of those numbers, along with some internal information that is needed to Python, such as the size of the list (3 in our example). Also, when the list is created, Python will reserve some extra memory slots, in case we decide to append some value to the list. In the case that we do a lot of appends and those slots become full, then Python will allocate some more memory space (e.g., with double the size), probably in some other memory location, it will copy the old values and then it will continue appending. The exact details of this depend on the paricular Python implementation and are not important for us.[7]

One may wonder why we cannot store directly the numbers 5347, 3971, and 510 inside the memory reserved for the list and instead Python creates new memory objects. Isn't this a waste of space and time? Indeed it is, but it has other advantages; for example it allows to store in the list elements of different type—here, the second value, instead of 3971 could have been the string `"OK"`, and the only difference would have been that the memory address in location 1E6B10 h would have pointed to a string object. As a matter of fact, most programming languages (C++, Java, etc.) have made the choice, differently from Python, to store directly the numbers. Also the arrays provided by Python's `numpy` library store directly the numbers

[7]This shows why indexing for lists starts at 0 instead of 1: when in our program we try to access the element `values[i]`, internally Python will translate it to simply performing a lookup at memory location indicated at position 1E6B08 h + 8$i$ (Figure 4).

```
1. Function MaxFind(A)
2. Input: A: Array of size n
3. Output: The maximum among the values in A
4.   max ← A[1]
5.   for i = 2 to n:
6.     if A[i] > max:
7.       max ← A[i]
8.     end if
9.   end for
10.  return max
```

Figure 5: Algorithm MaxFind: Returns the maximum among the elements of list $A$.

and not memory locations, so I advise you to use `numpy` if you need to perform a lot of vector operations on large data.

We can now see why lists are inefficient for some operations. Assume that we want to remove an elements, for example the second one. Then all the elements from the third one up to the end of the list have to be copied one position above. Thus, removing an element from the middle of the list requires $\Theta(n)$ steps in the worst case, where $n$ is the current number of elements.[8] Similarly, to add an element may require to copy the entire list to some other memory location, and may require $\Theta(n)$ steps. That's why Python reserves some extra space when creating a list, or when it increases its size by copying it elsewhere. Finally, searching or finding the maximum also require $\Theta(n)$ steps: we need to go to every single location and examine the elements. Therefore, if we need to perform a lot of these operations then probably a list is not the best way to store the data. Luckily there are more complicated *data structures* that allow to perform these operations much faster.

**Sets and dictionaries.** These two types require to be able to access efficiently their keys. To do that, Python uses a data structre called a *hash table*, which we will describe later.

# 5 Complexity of Algorithms

We will now start seeing algorithms. An *algorithm* is a sequence of instructions for solving a computational problem. Often, we describe an algorithm using *pseudocode*, which is a high-level description of the algorithmic steps, and which a programmer can turn into a program written is some programming language. We will see various pseudocode examples in this text.

## 5.1 A Simple Example

We start by showing a very simple algorithm, MaxFind, for finding the maximum among a list of elements. We give the description in pseudocode in Figure 5.

The first thing that we check once we design an algorithm is *correctness*: whether it gives always the correct result. Here it is straightforward (one could prove it by induction), although for more complicated algorithms sometimes we really need to prove correctness formally to be sure of the result; we do this in Section 5.5 for algorithm BinarySearch.

The next thing that we typically need to estimate is its efficiency, its speed. Of course the running time of a program that implements this algorithm will depend on many factors: the input that we use, the programming language, how efficient the implementation is, and of course, the machine on which it will be executed. To be able to evaluate the efficiency of the algorithm in a way that it does not depend on the

[8] See Section 5.2 for the meaning of $\Theta(n)$.

```
1. Function MaxFind(A)
2. Input: A: Array of size n
3. Output: The maximum among the values in A
4.    max ← A[1]
5.    i ← 2
6.    while i ≤ n:
7.       if A[i] > max:
8.          max ← A[i]
9.       end if
10.      i ← i + 1
11.   end while
12.   return max
```

Figure 6: Algorithm MaxFind: Returns the maximum among the elements of list $A$. It uses a **while** loop instead of a **for** loop.

particular hardware or on how it is implemented, we instead measure the *complexity of the algorithm*, which is the number of steps that it performs, as a function of the input size.

Here our input is an array of $n$ elements, so the input is of size $n$. Lines 4 and 10 are executed once. The loop will be executed $n-1$ times, so line 6 will be executed $n-1$ times. Line 7 depends, it can be executed from 0 times if ($A[1]$ is the largest element) to $n-1$ times (if all the elements are different and $A$ is sorted in increasing order). Therefore, in the worst case the number of steps is $3(n-1)+2 = 3n-1$.

Note that I could have written the pseudocode in a different way, using a **while** loop instead of a **for** loop; see Figure 6. If we measure now the number of steps we obtain $4(n-1)+3 = 4n-1$. Even though the two algorithms are exactly the same, we obtain a different result based on how we describe it. In any case, the important fact is that in both cases the running time is shown to grow linear in $n$. Thus we say that the algorithm is a linear algorithm. If we define $T(n)$ to be the worst-case running time of MaxFind when the input is of size $n$, we write that $T(n) = O(n)$, indicating that the order of growth of the running time is $n$ (linear). We now make a digression to define and explain this notation.

## 5.2 Big O Notation

To measure the time complexity of algorithms, computer scientists use the *big O notation* or *asymptotic notation*, which allows to look at the growth rate of a function, ignoring the constants. Assume that we are interested in calculating the growth rate of a function $T(n)$.

We say that a function $T(n)$ is of the order of $g(n)$ and we write $T(n) = O(g(n))$ if there exist constants $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{Z}^+$ such that for all $n \geq n_0$ we have that $T(n) \leq cg(n)$. Because we are interested on what happens when $n$ is *sufficiently large*, we call this also asymptotic notation.

To understand the definition, let us look at the function $T(n) = 7n^3 + 50n^2 - 10n + 4$. We will show that $T(n) = O(n^3)$.

- We have the term $7n^3$
- For $n \geq 50$ we have that $n^3 \geq 50n^2$
- For $n \geq 0$ we have that $0 \geq -10n$
- For $n \geq 2$ we have that $n^3 \geq 4$

Therefore, for $n \geq 50$ we have that

$$T(n) \leq 7n^3 + n^3 + 0 + n^3 = 9n^3,$$

```
import numpy as np
import matplotlib.pyplot as plt
n = np.arange(1., 1000., 1)
plt.plot(n, np.log(n), 'k--', n, n, 'r--',
         n, n*np.log(n), 'g--', n, 200*n**2, 'b--', n, n**3, 'y--')
#plt.yscale('log')
plt.show()
```

Figure 7: Sample code for plotting function growth.

which means that $T(n) = O(n^3)$ (with $c = 9$ and $n_0 = 50$; note that if we were more precise we could have obtained a smaller value of $c$ but we don't really care, we just want to show that there exists some $c$ and some $n_0$).

Usually, it is easy to compute the growth order of the function, as it equals to the term that grows faster in $n$ (the term $7n^3$ in our example). This will eventually dominate.

The following rules allow to handle most of the cases that we see in practice. Here, to make the presentation simple, I write $f(n) \leq g(n)$, to indicate $f(n) = O(g(n))$. $a, b, c$ and $d$ indicate positive constants, independent of $n$.

- $n^c \leq n^d$ for $c \leq d$.
- $\log^c n (= (\log n)^c) \leq n^d$.
- $n^c \leq d^n$, for $d > 1$.
- More generally, $n^c \leq d^{an^b}$, for $d > 1$.

In the following we order some functions that we may encounter in practice when analyzing algorithms:

$$1 \leq \log\log n \leq \log n \leq \log^2 n \leq \sqrt{n} \leq n \leq n\log\log n \leq n\log n \leq n\sqrt{n} \leq n^2 \leq n^3 \leq 2^n$$

I invite you to visualize some functions to see visual the difference in the growth. You can find some sample code to do it with the `mathplotlib` library in Figure 7.

Note that when we say that $T(n) = O(g(n))$, then $g(n)$ is an *upper bound* on the complexity of $T(n)$. This means, for example, that if $g(n) = O(h(n))$, then we also have $T(n) = O(h(n))$. Usually when we analyze algorithms we want to upper bound its complexity, so we use $O()$.

What if we want to *lower bound*? Then there exists the $\Omega()$, defined analogously:

We say that a function $T(n)$ is $T(n) = \Omega(g(n))$ if there exists constants $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{Z}^+$ such that for all $n \geq n_0$ we have that $T(n) \geq cg(n)$.

When we analyze algorithms, there are some cases that we want to prove a lower bound on the running time, that is, we would like to show that no algorithm can achieve a running time smaller than $g(n)$. Then we would say that we have the lower bound $T(n) = \Omega(g(n))$.

Finally, if we know that the precise order growth of a function is $g(n)$, then we can write $T(n) = \Theta(g(n))$. Thus we have that a function is $T(n) = \Theta(g(n))$ if it is both $T(n) = O(n)$ and $T(n) = \Omega(n)$.

To make these notions clear, for the function $T(n) = 7n^3 + 50n^2 - 10n + 4$ that we saw earlier, we have, for example:

- $T(n) = O(n^3), T(n) = O(n^4), T(n) = O(2^n)$
- $T(n) = \Omega(n^3), T(n) = \Omega(n^2), T(n) = \Omega(\log n)$
- $T(n) = \Theta(n^3)$

```
1. PrefixSumFunction MaxFind(A)
2. Input: A: Array of size n
3. Output: S: Prefix-sum array of A
4.   for i = 1 to n:
5.     S[i] ← 0
6.     for j = 1 to i:
7.       S[i] ← S[i] + A[j]
8.     end for
9.   end for
10.  return S
```

Figure 8: Algorithm PrefixSum: Returns the prefix-sum array of array $A$.

For your information, there exist also the $T(n) = o(n)$ and $T(n) = \omega(n)$, but we will not use them here; you can find more information online.[9]

## 5.3 Back to the Analysis of Algorithms

Let us get back to the MaxFind example of Section 5.1. There we showed that $T(n) \leq 3n - 1$, therefore we have that $T(n) = O(n)$. Note that the big O notation allows to avoid the minor problem we had in comparing the pseudocodes in figures 5 and 6, in both cases we have $T(n) = O(n)$ (and, of course, $T(n) = \Theta(n)$).

The downside of this notation is that it hides the constants, and sometimes they can be important, that is, two algorithms may have the same complexity asymptotically, but in practice they may differ a lot because of the constants. Thus there are cases where we may want to perform a more precise analysis. However, for the vast majority of the cases, the asymptotic notations is the one that we use, as it allows us to obtain a good idea of the efficiency of the algorithms, at a theoretical—and often at a practical—level.

We have shown that our algorithm is correct and we have computed its run-time complexity. The final step is to see if it is optimal. In other words, is there any other algorithm for finding the maximum with asymptotic running time less than $n$? The answer is of course no, because any algorithm must at least read the entire input and this requires $n$ steps. Thus for any (correct) algorithm that compute the maximum, if $T(n)$ is its running time, we have that $T(n) = \Omega(n)$. Therefore our the MaxFind algorithm is optimal, asymptotically.

## 5.4 Prefix Sum

To understand better the concepts of the analysis of algorithms, we turn to another simple problem, that of computing the prefix sums of an array. The input to the array is an array $A$ of length $n$ and we want to compute, for each $i \in [n]$ the sum $A[1] + A[2] + \cdots + A[i]$, that is we want to compute all the values[10]

$$S[i] = \sum_{j=1}^{i} A[j].$$

We call $S$ the prefix-sum array of $A$.

The most obvious algorithm is to follow the definition. We present the algorithm in Figure 8.

As with MaxFind we want to show correctness and compute the running time complexity. The correctness is straightforward, as we apply directly the definition to compute $S[i]$. For the running time, we notice that the step that is executed most of the times is step 7, which is inside the two **for** loops. How many times

[9] We say that $T(n) = o(g(n))$ if $T(n) = O(g(n))$ but $T(n)$ is not $\Omega(g(n))$. In other words if $T(n)$ is strictly smaller that $g(n)$, meaning that $T(n)/g(n)$ converges to 0 as $n \to \infty$. We can define similarly $\omega(g(n))$.

[10] We define $[n] = \{1, 2, \ldots, n\}$.

```
1. Function PrefixSumFast(A)
2. Input: A: Array of size n
3. Output: S: Prefix-sum array of A
4.    S[1] ← A[1]
5.    for i = 2 to n:
6.       S[i] ← S[i − 1] + A[i]
7.    end for
8.    return S
```

Figure 9: Algorithm PrefixSumFast: Returns the prefix-sum array of array $A$ in linear time.

is it executed? For a given $i$ it is executed $i$ times (**for** $j = 1$ to $i$) and $i$ ranges from 1 to $n$. Therefore the total number of executions is

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2).$$

The next question that we ask is: "Is this algorithm optimal?" In the same way as with MaxFind, we can argue that the running time is $\Omega(n)$: we need to at least read all the values in $A$. But there is a gap between the lower bound $n$ and the running time of our algorithm, $n^2$. When there is such a gap, there are three cases that may happen, maybe more than one at the same time:

1. The analysis of the algorithm is not tight, and the actual running time is lower than the one calculated (remember that we compute an upper bound).
2. There exists a faster, asymptotically, algorithm.
3. The lower bound is not tight and one could show a stronger (higher) lower bound.

The area of analysis of algorithms if full of problems for which we have a gap between the best lower bound that we have proven and the best algorithm that we have designed, and researchers try to close this gap by attacking all three fronts:

1. Here there are two paths: (1) We try to make the analysis more precise, more tight as we say, and decrease the complexity upper bound that we prove, or (2) we try to find some input instance for which the running time required matches the one that we have computed; in that case our analysis is tight.
2. We try to design a better algorithm. Sometimes this may be relatively easy by combining techniques that work on other similar problems, but other times it requires ingenuity, a deep understanding of the problem, and probably the necessity to invent new algorithmic techniques.
3. We try to improve the lower bound, that is, we try to show a higher upper bound. Sometimes we can do this by simple combinatorial arguments, but other times we may need to use tools from information theory, probability, communication complexity, and other more advanced tools.

The more one works on algorithms, the more intuition she develops to guide her towards which direction to follow, and often, pursuing one direction may show that we should attack another one. For example, while trying to design a better algorithm we may not succeed and this may allow us to understand that the problem is harder than we thought and guide us toward showing a better lower bound. On the other hand, there have been some problems that have been open for years before some researchers come up with a new idea and do some progress.

In any case, the prefix-sum problem is easy. Clearly there is a faster algorithms that we can design: we don't need to compute the summation all the time form scratch, but we can notice that $S[i] = S[i-1] + A[i]$. This gives the algorithm PrefixSumFast, shown in Figure 9.

The correctness of PrefixSumFast is again straightforward (one could prove it with induction on $n$). Its time complexity is of course $\Theta(n)$.

1. **Function** FACTORIAL($n$)
2. **Input**: $n$: Nonnegative integer
3. **Output**: $n!$: The factorial of $n$
4. **if** $n = 0$:
5. **return** 1
6. **end if**
7. **return** $n \cdot$ FACTORIAL$(n-1)$

Figure 10: Algorithm FACTORIAL: Given $n$ it computes $n!$.

## 5.5 Recursion

Recursion is a technique that, although not strictly necessary, it sometimes makes the description of algorithms and the corresponding programs much more natural and easy. In recursion, we define a function that may call itself. (More generally, a function may call another function, which may call another function, an so on, until we call the original function, closing the cycle. Here however we only deal with the most common type of recursion, a function calling itself.)

Let us consider the problem of computing the factorial of a given number: $n!$ We can define the recursive function to compute it, shown in Figure 10. The function starts by checking in line 4 if the input is 0, in which case it returns a value. Every recursive function needs to check a boundary condition, otherwise it will keep calling itself for ever. If the input is greater than 0 then it will continue and it will return $n \cdot (n-1)!$, where $(n-1)!$ is calculated, recursively, by calling the same function FACTORIAL.

The algorithm obviously returns the correct result, given that $n! = n \cdot (n-1)!$. Let us now compute the running time, $T(n)$. Here the input is only one value, $n$, but we notice that the running time depends on the value $n$ (and not on the number of input elements, as was the case for the previous problems). Therefore, we let $T(n)$ be the running time when FACTORIAL is called with input value $n$.

Of course we cannot compute $T(n)$ as before; its running time depends on itself! But we can write a recursive definition:

$$T(n) = \begin{cases} 2, & \text{if } n = 0, \\ T(n-1) + 3, & \text{otherwise.} \end{cases}$$

To see why this is the case, note that we first perform 1 operation (line 4) and if $n = 0$ we perform 1 more (line 5), instead if $n > 0$, in line 7 we call FACTORIAL with input $n-1$, requiring time $T(n-1)$, and then we multiply by $n$ (1 step) and we return the result (1 step).[11] The particular values, 2 and 3, are not important as we will use the Big O notation, the important thing is that there is a constant work that we account for.

There are a few ways to solve a recursive equation like this, here we will see the most immediate, which is to expand and try to understand the main form. Then, if we want to be rigorous, we use the one that we guessed and we substitute to prove it. Let us see this procedure in practice. For large $n$ we have that

[11] We assume that each multiplication requires a constant amount of time. However, as $n!$ becomes larger, which happens quite fast, $n!$ cannot fit in a single CPU register and multiplication requires time that depends on the length of the two numbers that are being multiplied. This is a more advanced issue, and we ignore it here.

$T(n) = T(n-1) + 3$, so if we apply this equation multiple times, and in the end $T(0) = 2$, we obtain:

$$\begin{aligned}
T(n) &= T(n-1) + 3 \\
&= (T(n-2) + 3) + 3 = T(n-2) + 3 \cdot 2 \\
&= (T(n-3) + 3) + 3 \cdot 2 = T(n-3) + 3 \cdot 3 \\
&= (T(n-4) + 3) + 3 \cdot 3 = T(n-4) + 3 \cdot 4 \\
&= (T(n-5) + 3) + 3 \cdot 4 = T(n-5) + 3 \cdot 5 \\
&= \ldots \\
&= T(n-i) + 3i \\
&= \ldots \\
&= T(n-n) + 3n = 2 + 3n.
\end{aligned}$$

We thus showed that $T(n) = 3n + 2$. If really want to prove it, we can use induction and the recursive formula: For $n = 0$ we get that $T(0) = 2$ and for $n \geq 1$ we assume that the formula is correct for up to $n-1$ and for $n$ we have:

$$T(n) = T(n-1) + 3 = (3(n-1) + 2) + 3 = 3n + 2,$$

proving that the formula $T(n) = 3n + 2$ holds for every $n \in \mathbb{Z}^+$. Therefore, we have that $T(n) = O(n)$.

Let us consider now the problem of computing the $n$th Fibonacci number. Recall that the Fibonacci sequence $\{F_n; n \in \mathbb{Z}^+\}$, is defined as

$$F_n = \begin{cases} n, & \text{if } n = 0 \text{ or } 1, \\ F_{n-2} + F_{n-1}, & \text{otherwise,} \end{cases}$$

so that the sequence is the

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

Algorithm FIBONACCI in Figure 11 solves the problem applying directly the definition. The algorithm applies directly the definition, so it is correct. Let us now compute the running time. Following the procedure that we saw for FACTORIAL, we can write:

$$T(n) = \begin{cases} 2, & \text{if } n \leq 1, \\ T(n-2) + T(n-1) + 3 & \text{otherwise.} \end{cases}$$

This seems to be similar to the running time that we saw earlier; yet, the fact that we have those two terms instead of one, and in which $n$ decreases by a constant amount, makes $T(n)$ be exponential! Here is an easy way to see it. For general $n$, and using the easy to prove fact that $T(n-1) \geq T(n-2)$, we have that:

$$\begin{aligned}
T(n) &= T(n-2) + T(n-1) + 3 \\
&\geq T(n-2) + T(n-2) \\
&= 2 \cdot T(n-2) \\
&\geq 2 \cdot 2 \cdot T(n-4) = 4 \cdot T(n-4) \\
&\geq 4 \cdot 2 \cdot T(n-6) = 8 \cdot T(n-6) \\
&\geq 8 \cdot 2 \cdot T(n-8) = 16 \cdot T(n-8) \\
&\geq \ldots \\
&\geq 2^i \cdot T(n-2i) \\
&\geq \ldots \\
&\geq 2^{n/2} \cdot T(n-2(n/2)) \\
&= 2^{n/2+1},
\end{aligned}$$

```
1. Function Fibonacci(n)
2. Input: n: Nonnegative integer
3. Output: F_n: The nth Fibonacci number
4.   if n ≤ 1:
5.     return n
6.   end if
7.   return Fibonacci(n − 2) + Fibonacci(n − 1)
```

Figure 11: Algorithm Fibonacci: Given $n$ it computes the $n$th Fibonacci number $F_n$, starting from 0.

```
1. Function FibonacciIterative(n)
2. Input: n: Nonnegative integer
3. Output: F_n: The nth Fibonacci number
4.   if n ≤ 1:
5.     return n
6.   end if
7.   previousValue ← 0
8.   currentValue ← 1
9.   for i = 2 to n:
10.    newValue ← previousValue + currentValue
11.    previousValue ← currentValue
12.    currentValue ← newValue
13.  end for
14.  return currentValue
```

Figure 12: Algorithm FibonacciIterative: Given $n$ it computes the $n$th Fibonacci number $F_n$, using an iterative algorithm.

assuming than $n$ is even (if $n$ is odd we obtain a similar expression).

Therefore it turns out that this algorithm is *very* inefficient! If we reflect upon it a little bit, we can see that it is because by simply using the recursion we recalculate things again and again. For instance, calculating the value $T(n)$ and $T(n-1)$ both require the calculation of $T(n-2)$. Thus we calculate the value $T(n-2)$ *twice* instead of once. But *each* of these calculations, for the same reasoning, will require to perform *twice* the calculation of $T(n-4)$, for a total of four times. This can explain why we obtain an exponential time.

What is the solution? There are different ways. One would be to keep the recursive algorithm, but when we compute a result store it in memory and not recompute it (a technique called *memoization* but it is too early for us to explain more). Other approaches can also work but are also not important to learn at this point (e.g., a technique called *tail recursion*). The easiest way is to not use recursion but compute it with a for loop. For example, the algorithm FibonacciIterative in Figure 12 computes the $n$th Fibonacci value in time $O(n)$.

We now move to the last example of recursion, which is also the most interesting. We want to create an algorithm that, given an array $A$ that is sorted with values from the smallest to the largest, and a value $x$, returns the index of $x$ in $A$, or $-1$ if $A$ does not contain $x$. Of course we can find if the element exists, in linear time, by scanning one by one the array $A$. However the fact that $A$ is sorted can help us tremendously (think about how you search for a word in a dictionary). This can give us the idea of *binary search*: we look at the middle element of $A$, and if it equals $x$ we are done. If not, then if $x$ is smaller than the middle element, we restrict our search on the first half of $A$. If $x$ is greater, then we restrict our search on the second half of $A$. In this way, in every step we can ignore half of the remaining elements. We present the pseudocode of the algorithm BinarySearch in Figure 13.

```
1. Function BinarySearch(A, low, high, x)
2. Input: A: Array of size n, sorted in nondecreasing order
3.        low: The lowest index of A in which we search
4.        high: The highest index of A in which we search
5.        x: Real value
6. Output: The index of A where x can be found, or −1 if x does not exist within A[low], ... A[high]
7.   if high < low:
8.     return −1
9.   end if
10.  mid = ⌊(low + high)/2⌋
11.  if x = A[mid]:
12.    return mid
13.  else if x < A[mid]:
14.    return BinarySearch(A, low, mid − 1, x)
15.  else:
16.    return BinarySearch(A, mid + 1, high, x)
17.  end if
```

Figure 13: Algorithm BinarySearch: It accepts array $A$ of length $n$, sorted in nondecreasing order, two integers *low* and *high*, and a value $x$. If $x$ exists in $A[low], A[low+1], \ldots, A[high-1], A[high]$, then it returns an index of $A$ where $x$ belongs. If $x$ does not exists in this range, it returns $-1$. In the first call, to search for $x$ in the entire array $A$, the function call is BinarySearch$(A, 1, n, x)$.

This is an example of how to prove the correctness of the algorithm when it is not completely obvious as in the previous examples that we saw. We argue as follows:

- If $x \neq A$, first note that the condition in line 11 can never be satisfied and, therefore, line 12 will never be executed. Furthermore, notice that, as long as $high \geq low$, the difference $high - low$ decreases in every recursive call. This means that eventually *high* will become less than *low* and then line 7 will evaluate to True and the function will return $-1$.
- On the other hand, if $x \in A$, then by examining every line (esp. lines 13–16) we can see that the algorithm BinarySearch maintains the invariant that $A[low] \leq x \leq A[high]$. This, combined with the fact that $A$ is sorted in nondecreasing order, means that we will never have that $high < low$, so line 7 will always evaluate to False and line 8 will never be executed.. On the other hand, in each recursive call the difference $high - low$ decreases. These two facts, imply that eventually we will have $A[mid] = x$ (at the latest when the function gets called with $low = high$) and then the condition in line 11 will evaluate to True and subsequently in line 12 it will return the correct index.

Let us now compute the running time, $T(n)$. The fact that we use the Big O notation allows us to be a bit sloppy. For $n = 1$ we have $T(1) = c'$ for some constant $c'$. Also, as long as $n > 1$ we have that

$$T(n) \leq T\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + c,$$

where $c$ is some constant that counts the constant amount of work we do at each step. Note also that we have that $\lceil (n-1)/2 \rceil \leq n/2$. Therefore, we obtain

$$\begin{aligned} T(n) &= T\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + c \leq T\left(\frac{n}{2}\right) + c \leq T\left(\frac{n}{4}\right) + 2c \leq T\left(\frac{n}{8}\right) + 3c \\ &\leq T\left(\frac{n}{16}\right) + 4c \leq \cdots \leq T\left(\frac{n}{2^i}\right) + c \cdot i \leq \cdots \leq T\left(\frac{n}{2^{\log_2 n}}\right) + c \log_2 n \\ &= T(1) + c \log_2 n = c' + c \log_2 n = O(\log n). \end{aligned}$$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | | | 7 | | | | |
| 6 | | | 1 | 9 | 5 | | | |
| | 9 | 8 | | | | | 6 | |
| 8 | | | | 6 | | | | 3 |
| 4 | | | 8 | | 3 | | | 1 |
| 7 | | | | 2 | | | | 6 |
| | 6 | | | | | 2 | 8 | |
| | | | 4 | 1 | 9 | | | 5 |
| | | | | 8 | | | 7 | 9 |

(a) A sudoku problem instance.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | 4 | 6 | 7 | 8 | 9 | 1 | 2 |
| 6 | 7 | 2 | 1 | 9 | 5 | 3 | 4 | 8 |
| 1 | 9 | 8 | 3 | 4 | 2 | 5 | 6 | 7 |
| 8 | 5 | 9 | 7 | 6 | 1 | 4 | 2 | 3 |
| 4 | 2 | 6 | 8 | 5 | 3 | 7 | 9 | 1 |
| 7 | 1 | 3 | 9 | 2 | 4 | 8 | 5 | 6 |
| 9 | 6 | 1 | 5 | 3 | 7 | 2 | 8 | 4 |
| 2 | 8 | 7 | 4 | 1 | 9 | 6 | 3 | 5 |
| 3 | 4 | 5 | 2 | 8 | 6 | 1 | 7 | 9 |

(b) A solved sudoku problem instance.

Figure 14: An unsolved and solved sudoku problem instance. (Source: `https://en.wikipedia.org/wiki/Sudoku`)

Notice that the exact constants $c$ and $c'$ do not matter if we want to show that $T(n) = O(\log n)$.

Thus we see that binary search is exponentially faster than linear search. Of course it requires to have the input already sorted.

Is it optimal? The answer is yes, but we will not prove it here.[12]

# 6 NP-Completeness

All the problems that we have seen until now can be solved by algorithms that require polynomial time. Yet there are problems that we do not know how to solve in polynomial time or whether they can even be solved in polynomial time. (By the way, there are even problems that we know that can never be solved by any algorithm no matter how long it takes, but that's another story.)

Computer science has developed the area of *complexity theory* to study the hardness of problems. Computational problems belong into *complexity classes* based on their hardness. Two very important classes are the following two:

**Class $\mathcal{P}$:** This includes the problems that can be solved in polynomial time.

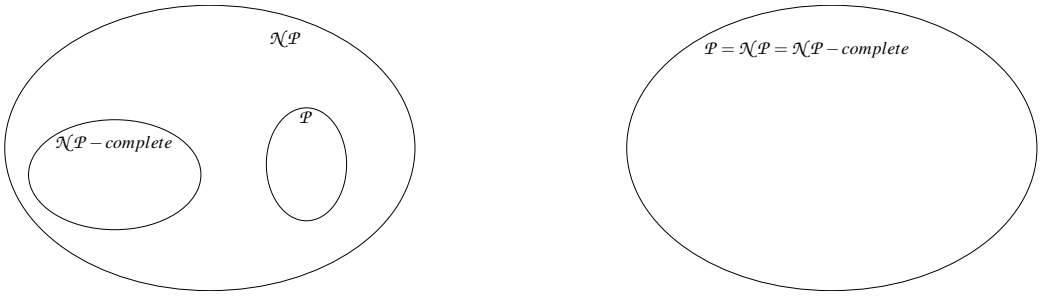
**Class $\mathcal{NP}$:** This includes the problems for which, if a solution is given, we can verify in polynomial time whether the solution is correct or not or.

Class $\mathcal{P}$ is easy to understand and we have seen various examples (finding the maximum element of an array, finding the index of a given nuber in a sorted array, finding the $n$th Fibonacci number, To understand the class $\mathcal{NP}$, think of the problem of sudoku (see Figure 14). Although sudoku might be hard to solve, it is very easy to verify if a given solution is correct: it is enough to check that every row, column, and block contains all digits 1–9. Sudoku can actually be generalized to having $n = k^2$ numbers, instead of just 9, and given a generalized problem intance of sudoku and a candidate solution, it is easy to verify whether a solution is correct, in time $O(n^2)$. Thus the problem of (generalized) sudoku is in $\mathcal{NP}$.

Note that we have $\mathcal{P} \subseteq \mathcal{NP}$: If a problem can be solved in polynomial time then we can certainly also verify in polynomial time if a given solution is correct.[13]; see Figure 15.

[12] The proof is based on *information theory*: We need to identify one index out of $n$ possible indices, so we need $\log_2 n$ bits of information, and each comparison that we perform gives as at most one bit of information, so we need to peform at least $\log_2 n$ comparisons.

[13] This is just the intution. To really prove that $\mathcal{P} \subseteq \mathcal{NP}$ we need to become more technical and it is not needed at this point.




(a) Scenario 1: $\mathcal{P} \neq \mathcal{NP}$.

(b) Scenario 2: $\mathcal{P} = \mathcal{NP}$.

Figure 15: Relationship between the $\mathcal{NP}$ and the $\mathcal{P}$ problem classes. One of the two possible scenarios holds but we do not know which one.

One may wonder whether there exist problems that are in $\mathcal{NP}$ but are not in $\mathcal{P}$. Actually, after 50 years of research, we still don't know the answer! There are two potential scenarios in our universe:

1. $\mathcal{P} \neq \mathcal{NP}$ (see Figure 15(a)): This means that there exist problems in $\mathcal{NP} \setminus \mathcal{P}$; these are problems that cannot be solved by a polynomial-time algorithm, but for which if we are given a solution we can verify in polynomial time whether the solution is correct.

2. $\mathcal{P} = \mathcal{NP}$ (see Figure 15(b)): Every problem in $\mathcal{NP}$ can be solved in polynomial time.

Which of the two scenarios is true? This is arguably the most important unsolvable problem in theoretical computer science and one of the most important for all mathematics. It is actually so important that the Clay institute categorized it as one of the seven millenium problems in mathematics, for which a solution would give one million U.S. dollars to the solver.[14] Other millenium problems are the Poincaré conjecture (solved in 2003) and the Riemann hypothesis. If it was shown that $\mathcal{P} = \mathcal{NP}$, this would have very profound implications in many fields, such as mathematics, cryptography, economics, and philosophy.

Most researchers though believe that $\mathcal{P} \neq \mathcal{NP}$ and there are good reasons for this. One is the following. Inside $\mathcal{NP}$ there exist some problems with the property that if we could solve any of them in polynomial time, then we could solve any problem in $\mathcal{NP}$ in polynomial time. We call this class of problems $\mathcal{NP}$-complete problems. Let us elaborate a bit more.

In complexity theory we have the concept of a *polynomial-time reduction* between two problems. We say that a problem $P_1$ can be reduced in polynomial time to another problem $P_2$, if there exists a way that transforms in polynomial time an instance of the $P_1$ problem to an instance of the $P_2$ problem. Thus, if we have a polynomial algorithm for $P_2$ and a reduction from $P_1$ to $P_2$ then we also have a polynomial-time algorithm for $P_1$. For example, finding the maximium among $n$ numbers can be polynomially reduced (trivially) to the problem of sorting $n$ numbers. But also the opposite is true: sorting $n$ numbers can be polynomially reduced to finding the maximum (find out how). Then we say that *a problem $P \in \mathcal{NP}$ is $\mathcal{NP}$-complete if every problem in $\mathcal{NP}$ can be polynomially reduced to $P$.*

Although a priori it is not clear why such problems should exist, It turns out that they do![15] Problems such as TSP, set cover, knapsack, max clique, which we will see in Section 6.1, are all $\mathcal{NP}$-complete. Notice that this indicates that these problems are very hard: for 50 years, researchers have been trying unsuccessfully to find a polynomial-time algorithm for problems in $\mathcal{NP}$. If any $\mathcal{NP}$-complete problem could be solved in polynomial time, this would imply a polynomial-time algorithm for all those problems that all these smart people have not been able to solve. Given this, most researchers believe that $\mathcal{P} \neq \mathcal{NP}$ and therefore the reality is as depicted in Figure 15(a). If however, at some point, someone would find a polynomial-time

[14] `https://www.claymath.org/millennium-problems`

[15] Cook in 1971 and Levin in 1973 proved, independently, that SAT, the problem of deciding whether a boolean formula that contains $n$ binary variables can be evaluated to True for some assignment of those $n$ variables, is $\mathcal{NP}$-complete. Then, to prove that some other problem $P$ is $\mathcal{NP}$-complete, it suffices to find a polynomial-time reduction from SAT to $P$.

algorithm for an $\mathcal{NP}$-complete problem, then all the three problem classes would collapse to one, as shown in Figure 15(b)

It is important of course to recognize if a problem that we come across is $\mathcal{NP}$-complete. In this case it is almost always useless to try to solve the problem optimally and we can decide to solve it (nonoptimally) with some other approach; see Section 6.2. The more you will work with algorithms, the more experienced you will become into recognizing whether a problem is $\mathcal{NP}$-complete or not.

Another related class of problems are the problems that are $\mathcal{NP}$-hard. A problem is $\mathcal{NP}$-hard if any problem in $\mathcal{NP}$ can be reduced to it. The difference from $\mathcal{NP}$-complete problems is that an $\mathcal{NP}$-hard problem does not have to be in $\mathcal{NP}$, it can be even harder![16]

$$\mathcal{NP}\text{-complete} = \mathcal{NP}\text{-hard} \cap \mathcal{NP}.$$

One last note. The concept of hardness that we discuss here, has to do with the *worst-case* scenario. In other words, when we say that a problem $P$ can be solved in polynomial time, it means there exists an algorithm such that for every instance of $P$ the algorithm can produce a solution in polynomial time. On the contrary, if a problem is not polynomial it means that there exists some problem instance that cannot be solved in polynomial time. Note that this does not mean that all problem instances are hard! Luckily, it turns out that some of the input instances to $\mathcal{NP}$-complete problems that we encounter in practice are easy instances to solve. However, we cannot be sure that this will always be the case, that is, that we will always come across easy instances.

## 6.1 Some Common $\mathcal{NP}$-Complete Problems

There are various $\mathcal{NP}$-complete problems that we encounter in practice. For a large list consult the book of Garey and Johnson [2] and the web [1].

Here is a list of some common problems, which I will expand at some point, for now you can search them online for more details.

- Travaling salesman problem (TSP): Given $n$ points and the distances between them, find the tour that visits all the points and minimizes the sum of the distances that belong to the tour.
- Hamiltonian path: Given a graph, find a path that visits each node exactly once.
- Knapsack: We are given a budget $W \in \mathbb{Z}^+$ and a collection $U = \{e_1, \ldots, e_n\}$ of $n$ elements. For each element $e_i$ we have a value $v_i$ and a weight $w_i$, with $v_i, w_i \in \mathbb{Z}^+$. The goal is to select a subset of elements $S \subseteq U$, such that,
$$\sum_{e_i \in S} w_i \leq W$$
that maximizes
$$\sum_{e_i \in S} v_i.$$
- Set cover: We have a univerver $U = \{e_1, e_2, \ldots, e_n\}$ of $n$ elements and $m$ sets $\mathcal{S} = \{S_1, \ldots S_m\}$, where $S_i \subseteq U$. The goal is to find the minimum number of sets that can cover all the elements of $U$, that is, find a collection $\mathcal{C} \subseteq \mathcal{S}$ that contains as few sets as possible, such that
$$\bigcup_{S \in \mathcal{C}} S = U.$$
- Maximum clique: Given a graph, find a subgraph of maximum size in which each node is directly connected with every other node in the subgraph.
- Subset sum: Given a list of nunbers, find a subset whose sume equals to 0.

[16] There is also another technical difference, the class $\mathcal{NP}$-complete contains the decision version of the problems, whereas $\mathcal{NP}$-hard may contain the optimization version.

### 6.2 Approximation Algorithms and Heuristics

Unfortunately, often we come across an $\mathcal{NP}$-complete problem that we need to solve (e.g., DHL every day computes tours for delivering packages). Even though we cannot hope for a polynomial-time algorithm that solves the problem, we still need to provide some solution. There are two main approaches that we follow.

**Approximation algorithms.** An *approximation algorithm* to a problem, is an algorithm that returns a solution that is guaranteed to be within a factor of the optimal solution. For example, even though we cannot solve the TSP exactly in polynomial time, there are approximation algorithms that if the input distances satisfy the triangle inequality guarantee a solution that is at most 50% worse than the optimal. There is a whole area in theoretical computer science dedicated to the study of such algorithms. The downside, is that, although polynomial, often these algorithms are inpractical in practice (they have a very high exponent in the polynomial running time) and then they are interesting only from a theoretical point of view.

**Heuristics.** *Heuristics* are typically simple, natural algorithms that attempt to solve the problem. Usually they are very easy to implement, and very often they return good solution in practice. Their main downside is that they do not provide some guarantee for the solution that they return. *Metaheuristics* are generic approaches that can be used to solve optimization problems. Examples of metaheuristic algorithms are *simulated annealing* (inspired from statistical physics), *ant-colony optimization*, *taboo search*, and other *local-search* approaches; these last ones, work by finding some or more candidate solutions and then try to improve them by searching for solutions that are similar to the one(s) already found.

## 7 Data Structures

As we saw in Section 4, arrays are not very efficient for all types of operations; for example, they are not efficient for searching for a given element by value—unless the elements are sorted. For this reason, computer scientists have designed *data structures*, which are ways of storing information in memory such that some operation can be performed efficiently. Each data structure may be more suitable for some type of operation and less suitable for another; therefore, it is up to the analyst and the programmer to decide which data structre is more appropriate for the desired application.

In this section we will present some om the most basic data structures, and their complexity in performing operations. Note that often, even though two data structures may have the same asymptotic complexity, one may be faster *in practice* for some operations because it may be simpler: for example, one data may require one memory lookup for a given operation, whereas another one may require three. Nevetheless, here we deal with the big O complexity and most of the time we ignore these differences; but as one starts designing real-world systems becomes more aware of these issues.

**Abstract and Concrete Data Structures** First we distinguish between two concepts, that of an abstract data type and that of a concrete data structure. Sometimes there can be a confusion between the two concepts, because there is not a standard terminology (the choice of the two terms that I use is not standard) and because they often share the same names, but it is important to know the difference.

An *abstract data type* (ADT) or *abstract data structure* is a description of the operations that a data structure must support. It is essentially an interface description. For example, the dictionary abstract data type, specifies that a data structure that supports it must offer ways to create it, insert and remove elements, search for an element, and so on.

A *(concrete) data structure* is a description of how the data are stored in memory and how various operations, such as inserting elements into it, removing elements from it, and accessing its elements are performed. A data structure can be used to implement some interface and different data structures can implement the same interface. For example, the dictionary abstract data type can be implemented by a balanced tree or by a hash table, each of which has different properties.

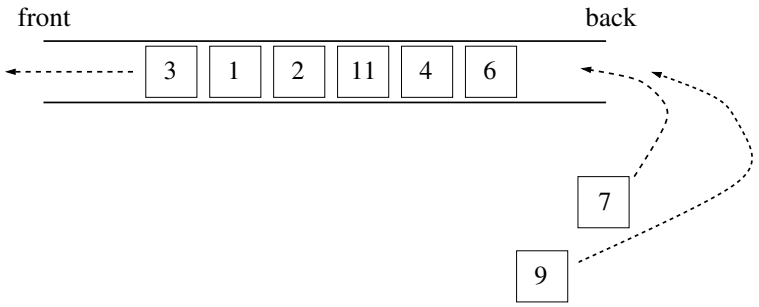



Figure 16: An example of a queue $Q$. Elements are inserted to the back of $Q$ and returned from the front. $Q$.peek() will return 3, and leave $Q$ as is. $Q$.dequeue() will return 3 and remove it, so that the next element in the front of the queue will be 1. Therefore, the sequence $Q$.dequeue(); $Q$.peek(); $Q$.dequeue(); $Q$.dequeue(), will return the values 3, 1, 1, and 2. The sequence of $Q$.enqueue(7); $Q$.enqueue(9), will put 7 to the back, after 6, and then 9 after 7.

To understand the difference, think of a car. A car is the analogue of an abstract data type: It specifies that it has four wheels, a gas and break pedals, a steering wheel used to turn, and so on. But there are different ways to implement a car: it can be based on a traditional internal combustion engine, it can be hybrid, gas-powered, electric, and so on. Each of the different tecnologies has its own properties and can be thought of as the analogue of a data structure.

## 7.1 Abstract Data Structures

We start by describing four abstract data structures: the queue, the stack, the dictionary, and the priority queue.

### 7.1.1 Sequence

A sequence $S$ supports the following operations.

- $Q$.enqueue($x$): Insert element $x$ to the back of $Q$.
- $Q$.dequeue(): Remove the element in the front of $Q$ and return it.
- $Q$.peek(): Return the elements in the front of $Q$.
- $Q$.isEmpty() : Return `True` if $Q$ is empty.

### 7.1.2 Queue

A *queue* $Q$ is a data structure that supports the *first in first out* (FIFO) paradigm: elements can be inserted to $Q$ one after the other, and the first one to be inserted is the last one to be extracted. See Figure 16.

A queue $Q$ supports the following operations.

- $Q$.enqueue($x$): Insert element $x$ to the back of $Q$.
- $Q$.dequeue(): Remove the element in the front of $Q$ and return it.
- $Q$.peek(): Return the elements in the front of $Q$.
- $Q$.isEmpty() : Return `True` if $Q$ is empty.

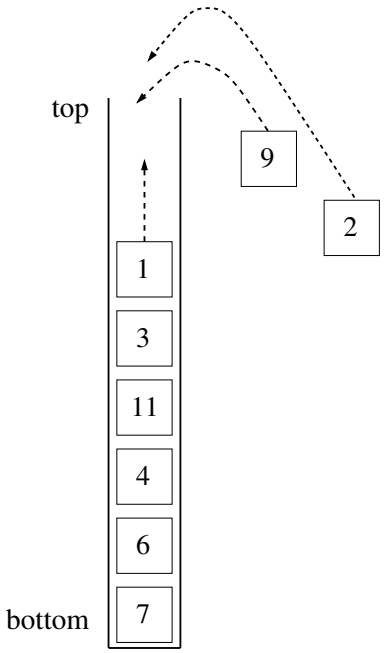



Figure 17: An example of a stack $S$. Elements are inserted on the top of $S$ and removed from the top. $S$.pop() will remove 1 from $S$ and return it. The sequence $S$.push(9); $S$.push(2), will insert 9 on top of 1 and 2 on top of 9, and will not return any value. Therefore, the sequence $S$.push(9); $S$.peek(); $S$.push(2); $S$.pop(); $S$.pop(); $S$.peek(); $S$.pop(); $S$.pop(), will return the values 9, 2, 9, 1, 1, and 3.

### 7.1.3 Stack

A *stack* is a data structure that supports the *last in first out* (LIFO) paradigm: elements can be inserted one after the other, and the last element inserted is the first element to be extracted.

A stack $S$ supports the following operations.

- $S$.push($x$): Insert element $x$ on the top of $Q$.
- $S$.pop(): Remove element form the top of $Q$ and return it.
- $S$.peek(): Return the element on the top of $Q$.
- $S$.isEmpty() : Return `True` if $Q$ is empty.

### 7.1.4 Dictionary

The *dictionary* ADT is a data structure that allows to associate a key with a value. Note that this is what Python's dictionaries implement. But we should not confuse the two concepts. The dictionary that we refer to in this section is an ADT, and Python's dicitionaries are data structures that implement the dictionary ADT. But one could implepent the dictionary ADT in some other way (e.g., with a *red-black tree.*)

A dictionary $D$ supports the following operations.

- $D$.set($k, v$): Set the value of $k$ in $D$ equal to $k$. Replace the existing value associated with $k$ if $k$ already exists in $D$.
- $D$.get($k$): Return the value associated with key $k$ in $D$.
- $D$.exists($k$): Return `True` if $k$ exists in $D$, `False` if it does not.

- $D$.delete($k$): Remove the key $k$ from $D$ and its associated value.
- $D$.isEmpty() : Return `True` if $D$ is empty.

#### 7.1.5 Priority Queue

The *priority queue* ADT allows to insert elements each of which has a priority, and it allows to extract the one with the maximum priority.

A priority queue $Q$ supports the following operations.

- $Q$.insert($x$): Insert element $x$ to the back of $Q$.
- $Q$.removeMax(): Remove the maximum element of $Q$ and return it.
- $Q$.peekMax(): Return the maximum elements in $Q$.
- $Q$.isEmpty() : Return `True` if $Q$ is empty.

As an example, assume that we execute the following sequence of operations, in an initially empty priority queue $Q$:

1. $Q$.isEmpty()
2. $Q$.insert(7)
3. $Q$.insert(2)
4. $Q$.insert(13)
5. $Q$.insert(4)
6. $Q$.insert(7)
7. $Q$.peekMax()
8. $Q$.removeMax()
9. $Q$.removeMax()
10. $Q$.isEmpty()
11. $Q$.insert(5)
12. $Q$.removeMax()

Then, the values contained in the queue and the results returned will be:

0. $Q = \{\}$
1. $Q = \{\}$, value returned: `True`
2. $Q = \{7\}$, no value returned
3. $Q = \{2, 7\}$, no value returned
4. $Q = \{2, 7, 13\}$, no value returned
5. $Q = \{2, 4, 7, 13\}$, no value returned
6. $Q = \{2, 4, 7, 7, 13\}$, no value returned

7. $Q = \{2, 4, 7, 7, 13\}$, value returned: 13
8. $Q = \{2, 4, 7, 7\}$, value returned: 13
9. $Q = \{2, 4, 7\}$, value returned: 7
10. $Q = \{2, 4, 7\}$, value returned: `False`
11. $Q = \{2, 4, 7\}$, value returned: `False`
12. $Q = \{2, 4, 5, 7\}$, no value returned
13. $Q = \{2, 4, 5\}$, value returned: 7

# 8 Algorithmic Techniques

Designing algorithms requires to understand well the problem, and with experience we become better in it. Whereas often we need to come up with original ideas, there are some standard techniques that in some cases can be applied to solve our problem or, at least, some part of our problem. Thus, knowing them, can make our life easier when we try to solve the problem. Here we present some of them, starting with dynamic programming.

## 8.1 Dynamic Programming

*Dynamic programming* is a very powerful technique, which, in the cases that it can be applied, it can provide a polynomial-time algorithm to a problem that might look as requiring exponential time. We will expose it with the example of the edit distance.

We define as the *edit distance* between two strings $S$ and $T$ as the minimum number of insertions or deletions required (from anywhere in the string) to transform $S$ to $T$. There are many other definitions for the edit diestance that you may find, for example allowing also substitutions, or allowing also transposition, but we stick with the simpler definition that only allows insertions and deletions.

For example the edit distance between strings "*ALGORITHMS*" and "*LOGARITHM*" is 5: We remove the letters "*A*," "*O*," and "*S*" and we insert the letters "*O*' and "*A*" in the right locations. Note that there are many ways to transform "*ALGORITHMS*" to "*LOGARITHM*". A trivial way is to remove all the character from "*ALGORITHMS*" and then insert all the characters of "*LOGARITHM*," resulting in 19 operations. The edit distance though is defined as the *minimum* number of operations in which this can be achieved. Note also that in general there may be more than one ways that this minimum number can be achieved.

It is easy to verify that the edit-distace definition satisfies all the properties of a distance function.

Let $S[i]$ be the $i$th character of string $S$, and define $S^i$ to be the prefix of length $i$ of $S$, that is

$$S^i = S[1] \circ S[2] \circ \cdots \circ S[i],$$

where $\circ$ denotes string concatenation. Define analogously for $T$. Let $n$ and $m$ be the lengths of $S$ and $T$, respectively, so we have that $S = S^n$ and $T = T^m$.

Let $\mathrm{ED}(S^i, T^j)$ be the edit distance between strings $S^i$ and $T^j$. Then we can inductively prove that, for $i \in \{0\} \cup [n], j \in \{0\} \cup [m]$, we have

$$\mathrm{ED}(S^i, T^j) = \begin{cases} i, & \text{if } j = 0 \\ j, & \text{if } i = 0 \\ \mathrm{ED}(S^{i-1}, T^{j-1}), & \text{if } i, j > 0 \text{ and } S[i] = T[j] \\ 1 + \min\{\mathrm{ED}(S^{i-1}, T^j), \mathrm{ED}(S^i, T^{j-1})\} & \text{if } i, j > 0 \text{ and } S[i] \neq T[j]. \end{cases} \tag{1}$$

1. **Function** EDITDISTANCERECURSIVE($S, T$)
2. **Input**: $S$: String of length $i$
3. $T$: String of length $j$
4. **Output**: The edit distance between $S$ and $T$
5. $i \leftarrow \text{len}(S)$
6. $j \leftarrow \text{len}(T)$
7. **if** $i = 0$:
8. **return** $j$
9. **else if** $j = 0$:
10. **return** $i$
11. **else if** $S[i] = T[j]$:
12. **return** EDITDISTANCERECURSIVE($S^{i-1}, T^{j-1}$)
13. **else**:
14. **return** $1 + \min\{\text{EDITDISTANCERECURSIVE}(S^{i-1}, T^{j}), \text{EDITDISTANCERECURSIVE}(S^{i}, T^{j-1})\}$
15. **end if**

Figure 18: Algorithm EDITDISTANCERECURSIVE: Given two strings $S$ and $T$, return the edit distance $\text{ED}(S, T)$ between them, computing it recursively.

Given this recursive formula, it is straightforward to implement the function recursively. In Figure 18 we we see algorithm EDITDISTANCERECURSIVE.

Let us compute the running time for lengths $T(i, j)$. The worst case, is the last case, when we need to perform two recursive calls. In that case we have that $T(i, j) = c+T(i-1, j)+T(i, j-1)$, for some constance $c \geq 1$. Let $N = i + j$, and let $T(N) = T(i, j)$. Then we can write

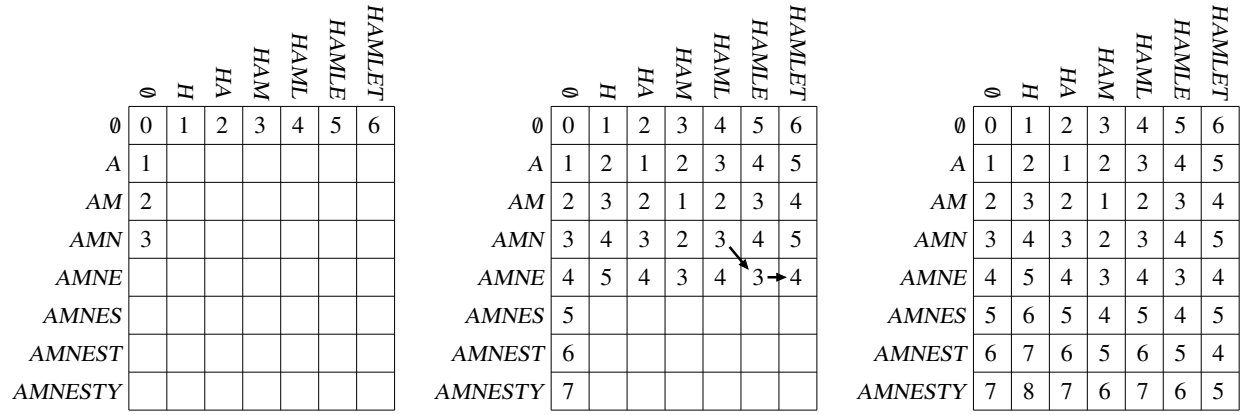
$$T(N) = 2 \cdot T(N-1) + c \geq 2 \cdot T(N-1) \geq 4 \cdot T(N-2) = 8 \cdot T(N-3) = \cdots = 2^{\ell} \cdot T(N-\ell).$$

We can continue until one of the two strings becomes the empty string, which will happen after at least $\ell \geq \min\{n, m\}$ steps. Therefore, in the worst case we have that $T(n, m) = \Omega(2^{\min\{n,m\}})$, that is, it requires exponential time.

The reason that this happens, is the same as with the algorithm FIBONACCI: the recursive definition leads to recomputing again and again the edit distance between the same substrings.

Luckily here we can use the technique of dynamic programming. We want to compute the optimal solution of the problem of transforming string $S$ to string $T$. Equation (1) shows us, that to compute this optimal solution, it suffices to use the optimal solution of three subinstances (depending on the case), namely the pairs of strings $(S^{i-1}, T^{j-1})$, $(S^{i-1}, T^{j})$, and $(S^{i}, T^{j-1})$, and that these optimal solutions can be reused. Thus, instead of recomputing them each time that we need them, we can store them. There are various ways to implement this, and one way is by building a *dynamic-programming table*.

See Figure 19. We construct a table with $n+1$ rows and $m+1$ columns, with the $i$th row (starting from 0) corresponding to the prefix $S^i$ and the $j$th column (starting from 0) corresponding to the prefix $T^j$. The value at cell $(i, j)$ equals to $\text{ED}(S^i, T^j)$. We can complete the table using Equation (1). Row and column 0 equal to the length of the nonempty string. Then, notice that because $\text{ED}(S^i, T^j)$ depends only on the corresponding values of the substrings, to compute the value at cell $(i, j)$, we just need values of the up–left diagonal cell (if $S[i] = T[j]$), or the values in the cells just left and just above (if $S[i] \neq T[j]$). Therefore, we can complete the table from top to bottom and from left to right. The edit distance $\text{ED}(S, T)$ is given at the bottom–right cell. We can see a description of algorithm EDITDISTANCEDP in Figure 20. The running time of this algorithm is $O(nm)$.




(a) We start by completing the first row and column. These just depend on the prefix lengths.

(b) We compute the values moving from top to bottom and from left to right.

(c) The table completed. The value at the bottom-right corner gives the edit distance of the two strings.

Figure 19: The dynamic-programming table for computing the distance between the strings $S =$ "*AMNESTY*" and $T =$ "*HAMLET*". Each row corresponds to a prefix $S^i$ and each column to a prefix $T^j$. Starting from top left, we fill in the table using Equation (1). In (a) we start by completing the first row and column. For instance, we have that ED("*AMN*", "") = 3. In (b) we use the previous values to complete the table. For example, we have that ED("*AMNE*", "*HAMLE*") = ED("*AMN*", "*HAML*") = 3. Also, ED("*AMNE*", "*HAMLET*") $= 1 + \min\{\text{ED}(\text{“}AMNE\text{”}, \text{“}HAMLE\text{”}), \text{ED}(\text{“}AMN\text{”}, \text{“}HAMLET\text{”})\} = 1 + \min\{3, 5\} = 4$. In (c) we have completed the table and we see that ED("*AMNESTY*", "*HAMLET*") = 5.

# References

[1] P. Crescenzi and V. Kann. A compendium of NP optimization problems. `http://www.csc.kth.se/~viggo/problemlist/compendium.html`, 2020 (accessed October 27, 2020).

[2] M. R. Garey and D. S. Johnson. *Computers and Intratability: A Guide to the Theory of NP-Completeness*. W. H. Freeman cnd Company, 1979.

```
1. Function EditDistanceDP(S, T)
2. Input: S: String of length n
3.        T: String of length m
4. Output: The edit distance between S and T
5.   n ← len(S)
6.   m ← len(T)
7.   edTable ← Z^{(n+1)×(m+1)}
8.   for j = 0 to m:
9.     edTable[0][j] ← j
10.  end for
11.  for i = 1 to n:
12.    edTable[i][0] ← i
13.  end for
14.  for i = 1 to n:
15.    for j = 1 to m:
16.      if S[i] = T[j]:
17.        edTable[i][j] ← edTable[i − 1][j − 1]
18.      else:
19.        edTable[i][j] ← 1 + min{edTable[i − 1][j], edTable[i][j − 1]}
20.      end if
21.    end for
22.  end for
23.  return edTable[n][m]
```

Figure 20: Algorithm EditDistanceDP: Given two strings $S$ and $T$, return the edit distance $\mathrm{ED}(S, T)$ between them, computing it by filling in the dynamic-programming table edTable. To simplify notation, we assume that indexing in the table starts at 0 and indexting in $S$ and $T$ starts at 1.