Basics of algorithmic complexity

Processing math: 100%

Each code instruction consists in several elementary operations that involve different components of the computer:

- CPU/GPU: the fastest
- memory: quite fast
- disk: very slow

The running time of each operation is variable. **Minimizing the algorithmic complexity** is chosing the instructions that:

- run fast
- do what we want to do

Processing math: 100%

Overhead and variable running times

Processing math: 100%

Key idea

In a typical scientific problem, the running time increases depending on some parameters but a constant time always exists.

Processing math: 100%

Simple example

In this part, a fake problem is built and it's code is analysed in order to understand how each instruction contributes to the overall algorithmic complexity.

Case study definition

Let's define a **naive** function that sums the first `n` integers.

In [1]:

```
def sum_integers(n):
    print('Summing the n first integers')
    total = 0
    for k in range(n):
        total += k
    print(f'The sum is: {total}')
```

Processing math: 100%

Time complexity decomposition

Intuitive approach

What is the time complexity of this code? There are two types of operations:

- operations that do not depend on `n`, i.e. the **size of the problem**
  - calls to `print` at the beginning and at the end
  - the creation of a variable `total`
  - the creation of a `range` instance
- `n` iterations in the `for` loop. Each iteration includes:
  - an incrementation of `k`
  - the update of `total`

Thus one understands that running time will never be near-zero even for small `n` values. Conversely, the overhead run time is negligible for large `n` values.

This shows that the optimization of a code depends on the typical usage one intend to have of this code.

Math approach

The `sum_integers` function performs:

- C_1 operations
- C_2 opérations for each iteration of the `for` loop

Eventually time complexity can be formulated this way:

$$C = C_1 + C_2 \times n$$

What is of interest in most cases is the evolution of the complexity with `n`, thus C becomes $C(n)$. We note that, asymptotically, for large values of `n`:

$$C(n) = O(n)$$

Advanced example

Let's add to the previous example a second `for` loop inside the first one:

In [2]:
```
def sum_integers_difficult(n):
    total = 0
    total2 = 0
    for k in range(n):
        total += k
        for j in range(k+1):
            total2 += j
    return total
```

The following operations are performed:

- C_1 operations (overhead)
- for each iteration in the first loop (`n` iterations):
    - C_2 operations (incrementation of `total`)
    - for each iteration in the second loop (`k+1` iterations): C_3 operations (incrementation of `total2`)

Processing math: 100%

Thus time complexity becomes:

$$C(n) = C_1 + \sum_{k=0}^{n-1} \left[C_2(k) + \sum_{j=0}^{k} C_3(k, j) \right]$$

But $C_2(k)$ is almost independant from k ($C_2(k) \simeq C_2$) and similarly $C_3(k, j) \simeq C_3$. Thus:

$$C(n) = C_1 + n \times C_2 + \frac{n(n+1)}{2} \times C_3$$

$$C(n) = C_1 + \left(C_2 + \frac{C_3}{2} \right) \times n + \frac{C_3}{2} \times n^2$$

Processing math: 100%

Thus **asymptotically**:

$$C(n) = O(n^2)$$

When `n` is doubled, running time is multiplied by 4.

Hidden operations

In the previous examples all instructions were rather explicit: simple loops and incrementations, no advanced function calls.

A complex code can include several calls to unknown functions. Thus the preferred way is to work at **the function level**:

- **User defined functions**: think about the time complexity of your function: overhead and asymptotic behaviour.
- **Other functions**: if the documentation says nothing about time complexity, you can make simple hypothesis to define lower and upper bounds of this complexity. For instance, the sum of a `numpy` array of `n` elements using `array.sum()` implies as many operations as there are elements in the array (`n`), thus one can expect a time complexity $O(n)$.