Computational cost aka computational complexity

Algorithms and programs must be correct and also *efficient*. That is, they must save *resources*:

- Time
- Memory
- Network traffic
- ...

Time is the most important but it depends on several factors:

- Hardware
- Implementation
- Input (size and content)

Simplification 1: Removing the hardware dependency

$$\text{Elementary instructions require a unit of time}$$
$$\Downarrow$$
$$\text{minimizing the running time} \Leftrightarrow \text{minimizing number of elementary instructions}$$

Simplification 2: Removing the implementation dependency

A block of a constant number of elementary operations requires a fixed constant c unit of time independently by the number of operations. Running a constant number of operations is equivalent to performing only one.

Simplification 3: Removing the input content dependency

If n is the size of the input, the number of elementary operations performed by the algorithm must me evaluated with respect the **worst case input** (instance) of size n.

Computational cost function

It remains a measure $T(n)$ that only depends on the size n of the input. We are interested in the order of magnitude of $T(n)$.

Sometime it is considered the value that $T(n)$ assumes with *hight probability* or in the *average case*.

- **High probability**: $T(n)$ takes that value 'almost always'. Its value can be larger, but rarely.
- **Average case**: It is the total cost of a sequence of m operations divided by m. The cost of a single expensive operation can be amortized by the execution of several cheap operations.

Big-O notation

A mathematical tool for defining the order of magnitude of a function.

Definition: Function $t(n)$ is $O(f(n))$ if there exist two constants c and n_0 such that for each $n \geq n_0$

$$t(n) \leq c \cdot f(n).$$

If $T(n)$ is $O(f(n))$, the order of growth of $T(n)$ is at most the one of $f(n)$. Observe that constant functions are $O(1)$.

Example

For all $n \geq 0$

$$T(n) = \sum_{i=1}^{i \leq n} i = \frac{n(n+1)}{2} \leq n^2$$

This function is $O(n^2)$.

Summarizing

If $T(n)$ is the computational cost (number of performed elementary operations) of an algorithm A on instances of size n and $T(n)$ is $O(f(n))$ for some function f, we say the $O(f(n))$ is the computational cost of A.

Examples: on lists of size n

- The cost of searching an item is $O(n)$
- The cost of indexing is $O(1)$
- The cost of cloning is $O(n)$
- The cost of sorting with Bubble sort is $O(n^2)$
- The cost of the binary search is $O(\log_2 n)$

The computational costs of some common Python statements or functions, n is the size of the input structure.

	Lists	Dictionaries	Sets
`a[k] = v`	$O(1)$	$O(1)^*$	
`x = a[k]`	$O(1)$	$O(1)^*$	
len`(a)`	$O(1)$	$O(1)$	$O(1)$
`a.`**append**`(x)`	$O(1)^*$		
Slicing/Cloning	$O(n)$		
del `a[k]`	$O(n)$	$O(1)^*$	
`a.`**get**`(k)`		$O(1)^*$	
in	$O(n)$	$O(1)^*$	$O(1)^*$
sorting	$O(n \log_2 n)^{**}$		
`a.`**add**`(x)`			$O(1)^*$
`a.`**remove**`(x)`			$O(1)^*$
union, intersection, difference			$O(n)^*$

* the average cost; ** with high probability.