UNIVERSITÉ
PARIS SCIENCES
& LETTRES

Quality of an Algorithm & Python Collections

The Art of Computer Programming

Pierre Senellart

28 September 2026

Outline

UNIVERSITÉ PARIS SCIENCES & LETTRES

Algorithmic complexity

Termination, correctness, efficiency

Lecture 1 defined **algorithmics** as the design of algorithms, the analysis of their performance, and the proof of their correctness. What makes an algorithm a **good** algorithm?

Termination, correctness, efficiency

Lecture 1 defined **algorithmics** as the design of algorithms, the analysis of their performance, and the proof of their correctness. What makes an algorithm a **good** algorithm?

Termination an algorithm **terminates** if the computation ends in a finite number of steps, regardless of the input

Termination, correctness, efficiency

Lecture 1 defined **algorithmics** as the design of algorithms, the analysis of their performance, and the proof of their correctness. What makes an algorithm a **good** algorithm?

Termination an algorithm **terminates** if the computation ends in a finite number of steps, regardless of the input

Correctness an algorithm is **correct** if the value returned is the solution to the problem, regardless of the input

Termination, correctness, efficiency

Lecture 1 defined **algorithmics** as the design of algorithms, the analysis of their performance, and the proof of their correctness. What makes an algorithm a **good** algorithm?

Termination an algorithm **terminates** if the computation ends in a finite number of steps, regardless of the input

Correctness an algorithm is **correct** if the value returned is the solution to the problem, regardless of the input

Efficiency how to define that?

How to measure the efficiency of an algorithm?

- Attempt to **characterize**, from the description of an algorithm, the **efficiency** of a program implementing it; or, from the description of a problem, the efficiency of a program implementing an algorithm solving that problem
- **Different notions** of efficiency, different notions of complexity:

Time complexity	execution **time** of a program
Space complexity	memory **space** used by a program
Communication complexity	volume of **data exchanged** by a distributed system
Kolmogorov complexity	**size** of the smallest **program** producing a given output
Circuit complexity	**size** of the smallest **circuit** computing the function, for each input size

- In this course: only the first two (and mainly the first!) – **algorithmic complexity**

How to compute time complexity

- We assume that each **elementary operation** appearing in the description of an algorithm:
  - arithmetic operations
  - variable assignments
  - comparisons
  - tests
  - etc.

  takes an **elementary time**, bounded by a constant C
- We compute the sum of the number of elementary operations performed, **as a function of the input size *n***, e.g., $42 \times n$
- We then deduce an upper bound, here $42 \times n \times C$, on the **total** time of the algorithm

Elementary operations

- Not formally defined yet

Elementary operations

- Not formally defined yet
- But we can think, for example:

Elementary operations

- Not formally defined yet
- But we can think, for example:
  - of the number of bytecode instructions that the interpreter executes (bounding the time taken by one instruction by the **maximum time** taken by any bytecode instruction)

Elementary operations

- Not formally defined yet
- But we can think, for example:
  - of the number of bytecode instructions that the interpreter executes (bounding the time taken by one instruction by the **maximum time** taken by any bytecode instruction)
  - of the number of assembly instructions that will be executed by the processor, one per **CPU cycle** (3 billion per second for a 3 GHz processor), once compiled

Elementary operations

- Not formally defined yet
- But we can think, for example:
  - of the number of bytecode instructions that the interpreter executes (bounding the time taken by one instruction by the **maximum time** taken by any bytecode instruction)
  - of the number of assembly instructions that will be executed by the processor, one per **CPU cycle** (3 billion per second for a 3 GHz processor), once compiled
  - In the listings of lecture 1, one iteration of the search loop is 9 bytecode instructions, or 7 assembly instructions

Elementary operations

- Not formally defined yet
- But we can think, for example:
  - of the number of bytecode instructions that the interpreter executes (bounding the time taken by one instruction by the **maximum time** taken by any bytecode instruction)
  - of the number of assembly instructions that will be executed by the processor, one per **CPU cycle** (3 billion per second for a 3 GHz processor), once compiled
  - In the listings of lecture 1, one iteration of the search loop is 9 bytecode instructions, or 7 assembly instructions
- In practice, **more complicated than that**: pipelining, multi-cycle instructions, speculative execution, etc.

Elementary operations

- Not formally defined yet
- But we can think, for example:
  - of the number of bytecode instructions that the interpreter executes (bounding the time taken by one instruction by the **maximum time** taken by any bytecode instruction)
  - of the number of assembly instructions that will be executed by the processor, one per **CPU cycle** (3 billion per second for a 3 GHz processor), once compiled
  - In the listings of lecture 1, one iteration of the search loop is 9 bytecode instructions, or 7 assembly instructions
- In practice, **more complicated than that**: pipelining, multi-cycle instructions, speculative execution, etc. No big deal! We can always bound elementary operations by a **sufficiently large constant**.

Elementary operations

- Not formally defined yet
- But we can think, for example:
  - of the number of bytecode instructions that the interpreter executes (bounding the time taken by one instruction by the **maximum time** taken by any bytecode instruction)
  - of the number of assembly instructions that will be executed by the processor, one per **CPU cycle** (3 billion per second for a 3 GHz processor), once compiled
  - In the listings of lecture 1, one iteration of the search loop is 9 bytecode instructions, or 7 assembly instructions
- In practice, **more complicated than that**: pipelining, multi-cycle instructions, speculative execution, etc. No big deal! We can always bound elementary operations by a **sufficiently large constant**.
- For theoretical reasoning, the notion can be made more **formal** (Turing machine, von Neumann machine); we skip that for now

$O()$, $\Omega()$, $\Theta()$ notation

- Let $f : \mathbb{N} \to \mathbb{R}_+$, $g : \mathbb{N} \to \mathbb{R}_+$ be two functions

$O()$, $\Omega()$, $\Theta()$ notation

- Let $f : \mathbb{N} \to \mathbb{R}_+$, $g : \mathbb{N} \to \mathbb{R}_+$ be two functions
- We write $f(n) \in O(g(n))$ (or $f(n) = O(g(n))$) if

$$\exists N \in \mathbb{N}, \exists \alpha \in \mathbb{R}_+^*, \forall n > N \quad f(n) \leqslant \alpha g(n)$$

$O()$, $\Omega()$, $\Theta()$ notation

- Let $f : \mathbb{N} \to \mathbb{R}_+$, $g : \mathbb{N} \to \mathbb{R}_+$ be two functions
- We write $f(n) \in O(g(n))$ (or $f(n) = O(g(n))$) if

$$\exists N \in \mathbb{N}, \exists \alpha \in \mathbb{R}_+^*, \forall n > N \quad f(n) \leqslant \alpha g(n)$$

- We write $f(n) \in \Omega(g(n))$ (or $f(n) = \Omega(g(n))$) if

$$\exists N \in \mathbb{N}, \exists \alpha \in \mathbb{R}_+^*, \forall n > N \quad f(n) \geqslant \alpha g(n)$$

$O()$, $\Omega()$, $\Theta()$ notation

- Let $f : \mathbb{N} \to \mathbb{R}_+$, $g : \mathbb{N} \to \mathbb{R}_+$ be two functions
- We write $f(n) \in O(g(n))$ (or $f(n) = O(g(n))$) if

$$\exists N \in \mathbb{N}, \exists \alpha \in \mathbb{R}_+^*, \forall n > N \quad f(n) \leqslant \alpha g(n)$$

- We write $f(n) \in \Omega(g(n))$ (or $f(n) = \Omega(g(n))$) if

$$\exists N \in \mathbb{N}, \exists \alpha \in \mathbb{R}_+^*, \forall n > N \quad f(n) \geqslant \alpha g(n)$$

- We write $f(n) \in \Theta(g(n))$ (or $f(n) = \Theta(g(n))$) if

$$f(n) = O(g(n)) \quad \text{and} \quad f(n) = \Omega(g(n))$$

$O()$, $\Omega()$, $\Theta()$ notation

- Let $f : \mathbb{N} \to \mathbb{R}_+$, $g : \mathbb{N} \to \mathbb{R}_+$ be two functions
- We write $f(n) \in O(g(n))$ (or $f(n) = O(g(n))$) if

$$\exists N \in \mathbb{N}, \exists \alpha \in \mathbb{R}_+^*, \forall n > N \quad f(n) \leqslant \alpha g(n)$$

- We write $f(n) \in \Omega(g(n))$ (or $f(n) = \Omega(g(n))$) if

$$\exists N \in \mathbb{N}, \exists \alpha \in \mathbb{R}_+^*, \forall n > N \quad f(n) \geqslant \alpha g(n)$$

- We write $f(n) \in \Theta(g(n))$ (or $f(n) = \Theta(g(n))$) if

$$f(n) = O(g(n)) \quad \text{and} \quad f(n) = \Omega(g(n))$$

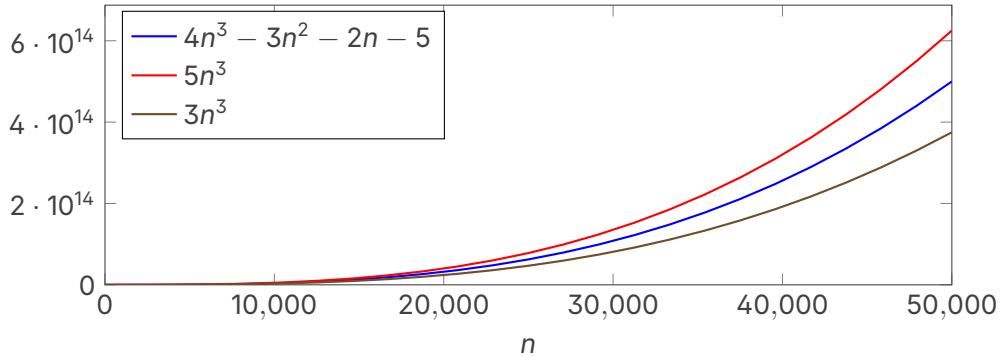
- In words: up to a constant factor and for n large enough, f is **at most** g, **at least** g, **the same as** g

Examples of $O()$, $\Omega()$, $\Theta()$ (1/2)

If P is a polynomial of degree k (with a positive leading coefficient), $P(n) \in \Theta(n^k)$

Examples of $O()$, $\Omega()$, $\Theta()$ (1/2)

If P is a polynomial of degree k (with a positive leading coefficient), $P(n) \in \Theta(n^k)$



Examples of $O()$, $\Omega()$, $\Theta()$ (2/2)

- $\log n \in O(n)$ but $\log n \notin \Omega(n)$

Examples of $O()$, $\Omega()$, $\Theta()$ (2/2)

- $\log n \in O(n)$ but $\log n \notin \Omega(n)$
- $n \log n \in O(n^2)$, $n \log n \in \Omega(n)$, but $n \log n \notin \Omega(n^2)$

Examples of $O()$, $\Omega()$, $\Theta()$ (2/2)

- $\log n \in O(n)$ but $\log n \notin \Omega(n)$
- $n \log n \in O(n^2)$, $n \log n \in \Omega(n)$, but $n \log n \notin \Omega(n^2)$
- $42n^2 \log n + 23n^2 \in \Theta(n^2 \log n)$

Examples of $O()$, $\Omega()$, $\Theta()$ (2/2)

- $\log n \in O(n)$ but $\log n \notin \Omega(n)$
- $n \log n \in O(n^2)$, $n \log n \in \Omega(n)$, but $n \log n \notin \Omega(n^2)$
- $42n^2 \log n + 23n^2 \in \Theta(n^2 \log n)$
- for any polynomial P, $P(n) \in O(2^n)$ but $P(n) \notin \Omega(2^n)$

Examples of $O()$, $\Omega()$, $\Theta()$ (2/2)

- $\log n \in O(n)$ but $\log n \notin \Omega(n)$
- $n \log n \in O(n^2)$, $n \log n \in \Omega(n)$, but $n \log n \notin \Omega(n^2)$
- $42n^2 \log n + 23n^2 \in \Theta(n^2 \log n)$
- for any polynomial P, $P(n) \in O(2^n)$ but $P(n) \notin \Omega(2^n)$
- If $f(n) \in O(\varphi(n))$ and $g(n) \in O(\psi(n))$, then $f(n) + g(n) \in O(\varphi(n) + \psi(n))$ and $f(n) \times g(n) \in O(\varphi(n) \times \psi(n))$

Examples of $O()$, $\Omega()$, $\Theta()$ (2/2)

- $\log n \in O(n)$ but $\log n \notin \Omega(n)$
- $n \log n \in O(n^2)$, $n \log n \in \Omega(n)$, but $n \log n \notin \Omega(n^2)$
- $42n^2 \log n + 23n^2 \in \Theta(n^2 \log n)$
- for any polynomial P, $P(n) \in O(2^n)$ but $P(n) \notin \Omega(2^n)$
- If $f(n) \in O(\varphi(n))$ and $g(n) \in O(\psi(n))$, then $f(n) + g(n) \in O(\varphi(n) + \psi(n))$ and $f(n) \times g(n) \in O(\varphi(n) \times \psi(n))$
- If $f(n) \in \Omega(\varphi(n))$ and $g(n) \in \Omega(\psi(n))$, then $f(n) + g(n) \in \Omega(\varphi(n) + \psi(n))$ and $f(n) \times g(n) \in \Omega(\varphi(n) \times \psi(n))$

Additional remarks

- Since $\log_b n = \frac{\ln n}{\ln b}$ by definition, $\log_b n \in \Theta(\ln n)$ for any $b > 1$: we can therefore omit the base of the logs and write $\Theta(\log n)$, $O(\log n)$, $\Omega(\log n)$

Additional remarks

- Since $\log_b n = \frac{\ln n}{\ln b}$ by definition, $\log_b n \in \Theta(\ln n)$ for any $b > 1$: we can therefore omit the base of the logs and write $\Theta(\log n)$, $O(\log n)$, $\Omega(\log n)$
- If $f(n) \in O(g(n))$, then $g(n) \in \Omega(f(n))$

Additional remarks

- Since $\log_b n = \frac{\ln n}{\ln b}$ by definition, $\log_b n \in \Theta(\ln n)$ for any $b > 1$: we can therefore omit the base of the logs and write $\Theta(\log n)$, $O(\log n)$, $\Omega(\log n)$
- If $f(n) \in O(g(n))$, then $g(n) \in \Omega(f(n))$
- If $f(n) \in \Omega(g(n))$, then $g(n) \in O(f(n))$

Additional remarks

- Since $\log_b n = \frac{\ln n}{\ln b}$ by definition, $\log_b n \in \Theta(\ln n)$ for any $b > 1$: we can therefore omit the base of the logs and write $\Theta(\log n)$, $O(\log n)$, $\Omega(\log n)$
- If $f(n) \in O(g(n))$, then $g(n) \in \Omega(f(n))$
- If $f(n) \in \Omega(g(n))$, then $g(n) \in O(f(n))$
- If $f(n) \in \Theta(g(n))$, then $g(n) \in \Theta(f(n))$

Additional remarks

- Since $\log_b n = \frac{\ln n}{\ln b}$ by definition, $\log_b n \in \Theta(\ln n)$ for any $b > 1$: we can therefore omit the base of the logs and write $\Theta(\log n)$, $O(\log n)$, $\Omega(\log n)$
- If $f(n) \in O(g(n))$, then $g(n) \in \Omega(f(n))$
- If $f(n) \in \Omega(g(n))$, then $g(n) \in O(f(n))$
- If $f(n) \in \Theta(g(n))$, then $g(n) \in \Theta(f(n))$
- Usually, we put in $O()$, $\Omega()$, $\Theta()$ expressions "as simple as possible": $O(n^2)$ rather than $O(3n^2 + 2n)$

Additional remarks

- Since $\log_b n = \frac{\ln n}{\ln b}$ by definition, $\log_b n \in \Theta(\ln n)$ for any $b > 1$: we can therefore omit the base of the logs and write $\Theta(\log n)$, $O(\log n)$, $\Omega(\log n)$
- If $f(n) \in O(g(n))$, then $g(n) \in \Omega(f(n))$
- If $f(n) \in \Omega(g(n))$, then $g(n) \in O(f(n))$
- If $f(n) \in \Theta(g(n))$, then $g(n) \in \Theta(f(n))$
- Usually, we put in $O()$, $\Omega()$, $\Theta()$ expressions "as simple as possible": $O(n^2)$ rather than $O(3n^2 + 2n)$
- Writing $f(n) = O(g(n))$ instead of $f(n) \in O(g(n))$ is an abuse of notation (it is not an equality)... but everyone does it

Asymptotic (time) complexity

- We use the notation $O()$, $\Omega()$, $\Theta()$ and the established bounds to indicate the complexity of an algorithm while **ignoring** C and other constants
- For example, if the time τ of each elementary operation is **bounded** by:

$$C_1 \leqslant \tau \leqslant C_2$$

- ... and if **for all inputs** of size n, algorithm $\mathcal{A}$ performs $42 \times n$ elementary operations, then the **running time** $T(\mathcal{A}, n)$ of $\mathcal{A}$ on inputs of size n satisfies:

$$42 \times C_1 \times n \leqslant T(\mathcal{A}, n) \leqslant 42 \times C_2 \times n$$

and thus $T(\mathcal{A}, n) = \Theta(n)$

Worst-case and average-case complexity

- Usually, we look for an upper bound on the time of an algorithm, and consider the **worst-case** complexity: an upper bound that holds for any input of size *n*
- Sometimes too restrictive, so we look at the **average case**: on average, over all inputs of a given size, what is an upper bound on the complexity?
- This average case assumes that all inputs have the **same probability**, which is **debatable**
- Occasionally, we also mention the **best case**: a lower bound on the time, reached for some (usually lucky) inputs; not to be confused with Ω(), which can be used for any of the three

Asymptotic complexity vs. real efficiency

- Asymptotic complexity **matters**. An algorithm in $\Theta(n)$ is slower than one in $O(\log n)$ *if n is large enough*

Asymptotic complexity vs. real efficiency

- Asymptotic complexity **matters**. An algorithm in $\Theta(n)$ is slower than one in $O(\log n)$ *if n is large enough*
- Sometimes, an algorithm in $\Omega(n^2)$ (i.e., $\geqslant \alpha n^2$) can be more efficient than one in $O(n)$ (i.e., $\leqslant \beta n$) for a **practical input size *n***, because $\alpha \ll \beta$

Asymptotic complexity vs. real efficiency

- Asymptotic complexity **matters**. An algorithm in $\Theta(n)$ is slower than one in $O(\log n)$ *if n is large enough*
- Sometimes, an algorithm in $\Omega(n^2)$ (i.e., $\geqslant \alpha n^2$) can be more efficient than one in $O(n)$ (i.e., $\leqslant \beta n$) for a **practical input size *n***, because $\alpha \ll \beta$
- Sometimes, the **worst-case** complexity is high, but the average-case complexity is low, and that is what matters in practice

Asymptotic complexity vs. real efficiency

- Asymptotic complexity **matters**. An algorithm in $\Theta(n)$ is slower than one in $O(\log n)$ *if n is large enough*
- Sometimes, an algorithm in $\Omega(n^2)$ (i.e., $\geqslant \alpha n^2$) can be more efficient than one in $O(n)$ (i.e., $\leqslant \beta n$) for a **practical input size *n***, because $\alpha \ll \beta$
- Sometimes, the **worst-case** complexity is high, but the average-case complexity is low, and that is what matters in practice
- The **programming language used** has a real impact on performance (but usually only by a constant factor, potentially large: see the Python vs. C measurements later)

Asymptotic complexity vs. real efficiency

- Asymptotic complexity **matters**. An algorithm in $\Theta(n)$ is slower than one in $O(\log n)$ *if n is large enough*
- Sometimes, an algorithm in $\Omega(n^2)$ (i.e., $\geqslant \alpha n^2$) can be more efficient than one in $O(n)$ (i.e., $\leqslant \beta n$) for a **practical input size n**, because $\alpha \ll \beta$
- Sometimes, the **worst-case** complexity is high, but the average-case complexity is low, and that is what matters in practice
- The **programming language used** has a real impact on performance (but usually only by a constant factor, potentially large: see the Python vs. C measurements later)
- Time complexity says nothing about the potential for parallelization or distribution of an algorithm – **other** notions of complexity are needed

In practice: solving a problem efficiently

- **Design** (mentally or on paper) the most efficient algorithm possible, using the most appropriate data structures (second half of this lecture, and later ones)

In practice: solving a problem efficiently

- **Design** (mentally or on paper) the most efficient algorithm possible, using the most appropriate data structures (second half of this lecture, and later ones)
- If not obvious, prove its **termination** and **correctness**

In practice: solving a problem efficiently

- **Design** (mentally or on paper) the most efficient algorithm possible, using the most appropriate data structures (second half of this lecture, and later ones)
- If not obvious, prove its **termination** and **correctness**
- Compute its **algorithmic complexity** ($O()$ or even $\Theta()$), in the worst case (and possibly in the average case)

In practice: solving a problem efficiently

- **Design** (mentally or on paper) the most efficient algorithm possible, using the most appropriate data structures (second half of this lecture, and later ones)
- If not obvious, prove its **termination** and **correctness**
- Compute its **algorithmic complexity** ($O()$ or even $\Theta()$), in the worst case (and possibly in the average case)
- **Implement** the algorithm in a programming language as faithfully as possible, taking into account the specificities of the environment

In practice: solving a problem efficiently

- **Design** (mentally or on paper) the most efficient algorithm possible, using the most appropriate data structures (second half of this lecture, and later ones)
- If not obvious, prove its **termination** and **correctness**
- Compute its **algorithmic complexity** ($O()$ or even $\Theta()$), in the worst case (and possibly in the average case)
- **Implement** the algorithm in a programming language as faithfully as possible, taking into account the specificities of the environment
- **Test** the correctness and efficiency of the algorithm on small examples and real examples, experimentally checking the algorithmic complexity

In practice: solving a problem efficiently

- **Design** (mentally or on paper) the most efficient algorithm possible, using the most appropriate data structures (second half of this lecture, and later ones)
- If not obvious, prove its **termination** and **correctness**
- Compute its **algorithmic complexity** ($O()$ or even $\Theta()$), in the worst case (and possibly in the average case)
- **Implement** the algorithm in a programming language as faithfully as possible, taking into account the specificities of the environment
- **Test** the correctness and efficiency of the algorithm on small examples and real examples, experimentally checking the algorithmic complexity
- Perform **profiling** to determine the parts of the program where the most time is spent; optimize them (possibly by going back to the algorithm design for those parts)

Space complexity

- Similar to time complexity, except that we measure the **elementary uses of memory** instead of the time of elementary operations
- We often make **simplifying assumptions**, such as assuming that any integer fits in constant space
- We **do not count** the space required to represent the input
- We also use $O()$, $\Omega()$, $\Theta()$ to summarize the **asymptotic** complexity
- For example, for linear search in an array (next section), the space complexity is $\boldsymbol{O(1)}$: we only need to store the variable i in memory (in addition to the inputs T and x), which requires a constant amount of space, independent of the input size
- For a **recursive** algorithm, each pending call occupies space on the **call stack** (lecture 2): the recursion depth counts in the space complexity

UNIVERSITÉ PARIS SCIENCES & LETTRES

Examples of complexity calculation

First example: search in an array

How many elementary operations?

Input: Array T with n distinct elements, an element x
Output: the position of x in T

```
1: for i ← 0 to n − 1 do
2:     if T[i] = x then
3:         return i
4:     end if
5: end for
6: return not found
```

(the example algorithm of lecture 1)

First example: search in an array

How many elementary operations?

Input: Array T with n distinct elements, an element x
Output: the position of x in T

```
1: for i ← 0 to n − 1 do
2:     if T[i] = x then
3:         return i
4:     end if
5: end for
6: return not found
```

(the example algorithm of lecture 1)

Worst case

First example: search in an array

How many elementary operations?

Input: Array T with n distinct elements, an element x
Output: the position of x in T

```
1: for i ← 0 to n − 1 do
2:     if T[i] = x then
3:         return i
4:     end if
5: end for
6: return not found
```

(the example algorithm of lecture 1)

Worst case (x absent or in last position)

First example: search in an array

How many elementary operations?

Input: Array T with n distinct elements, an element x
Output: the position of x in T

```
1: for i ← 0 to n − 1 do
2:     if T[i] = x then
3:         return i
4:     end if
5: end for
6: return not found
```

(the example algorithm of lecture 1)

Worst case (x absent or in last position)

- n assignments of i

First example: search in an array

How many elementary operations?

Input: Array T with n distinct elements, an element x
Output: the position of x in T

```
1: for i ← 0 to n − 1 do
2:     if T[i] = x then
3:         return i
4:     end if
5: end for
6: return not found
```

(the example algorithm of lecture 1)

Worst case (x absent or in last position)

- n assignments of i
- n comparisons of i with n

First example: search in an array

How many elementary operations?

Input: Array T with n distinct elements, an element x
Output: the position of x in T

```
1: for i ← 0 to n − 1 do
2:     if T[i] = x then
3:         return i
4:     end if
5: end for
6: return not found
```

(the example algorithm of lecture 1)

Worst case (x absent or in last position)

- n assignments of i
- n comparisons of i with n
- n accesses to $T[i]$

First example: search in an array

How many elementary operations?

Input: Array T with n distinct elements, an element x
Output: the position of x in T

```
1: for i ← 0 to n − 1 do
2:     if T[i] = x then
3:         return i
4:     end if
5: end for
6: return not found
```

(the example algorithm of lecture 1)

Worst case (x absent or in last position)

- n assignments of i
- n comparisons of i with n
- n accesses to $T[i]$
- n comparisons of $T[i]$ with x

First example: search in an array

How many elementary operations?

Input: Array T with n distinct elements, an element x
Output: the position of x in T

```
1: for i ← 0 to n − 1 do
2:     if T[i] = x then
3:         return i
4:     end if
5: end for
6: return not found
```

(the example algorithm of lecture 1)

Worst case (x absent or in last position)

- n assignments of i
- n comparisons of i with n
- n accesses to $T[i]$
- n comparisons of $T[i]$ with x
- 1 return

First example: search in an array

How many elementary operations?

Input: Array T with n distinct elements, an element x
Output: the position of x in T

```
1: for i ← 0 to n − 1 do
2:     if T[i] = x then
3:         return i
4:     end if
5: end for
6: return not found
```

(the example algorithm of lecture 1)

Worst case (x absent or in last position)

- n assignments of i
- n comparisons of i with n
- n accesses to $T[i]$
- n comparisons of $T[i]$ with x
- 1 return

about $4n$, i.e., ***O(n)***

First example: search in an array

How many elementary operations?

Input: Array T with n distinct elements, an element x
Output: the position of x in T

```
1: for i ← 0 to n − 1 do
2:     if T[i] = x then
3:         return i
4:     end if
5: end for
6: return not found
```

(the example algorithm of lecture 1)

Worst case (x absent or in last position)

- n assignments of i
- n comparisons of i with n
- n accesses to $T[i]$
- n comparisons of $T[i]$ with x
- 1 return

about $4n$, i.e., ***O(n)***

Average case

First example: search in an array

How many elementary operations?

Input: Array T with n distinct elements, an element x
Output: the position of x in T

```
1: for i ← 0 to n − 1 do
2:     if T[i] = x then
3:         return i
4:     end if
5: end for
6: return not found
```

(the example algorithm of lecture 1)

Worst case (x absent or in last position)

- n assignments of i
- n comparisons of i with n
- n accesses to $T[i]$
- n comparisons of $T[i]$ with x
- 1 return

about $4n$, i.e., ***O(n)***

Average case (x present, on average at position $n/2$)

First example: search in an array

How many elementary operations?

Input: Array T with n distinct elements, an element x
Output: the position of x in T

```
1: for i ← 0 to n − 1 do
2:     if T[i] = x then
3:         return i
4:     end if
5: end for
6: return not found
```

(the example algorithm of lecture 1)

Worst case (x absent or in last position)

- n assignments of i
- n comparisons of i with n
- n accesses to $T[i]$
- n comparisons of $T[i]$ with x
- 1 return

about $4n$, i.e., ***O(n)***

Average case (x present, on average at position $n/2$)

- $n/2$ assignments of i

First example: search in an array

How many elementary operations?

Input: Array T with n distinct elements, an element x
Output: the position of x in T

```
1: for i ← 0 to n − 1 do
2:     if T[i] = x then
3:         return i
4:     end if
5: end for
6: return not found
```

(the example algorithm of lecture 1)

Worst case (x absent or in last position)

- n assignments of i
- n comparisons of i with n
- n accesses to $T[i]$
- n comparisons of $T[i]$ with x
- 1 return

about $4n$, i.e., ***O(n)***

Average case (x present, on average at position $n/2$)

- $n/2$ assignments of i
- $n/2$ comparisons of i with n

First example: search in an array

How many elementary operations?

Input: Array T with n distinct elements, an element x
Output: the position of x in T

```
1: for i ← 0 to n − 1 do
2:     if T[i] = x then
3:         return i
4:     end if
5: end for
6: return not found
```

(the example algorithm of lecture 1)

Worst case (x absent or in last position)

- n assignments of i
- n comparisons of i with n
- n accesses to $T[i]$
- n comparisons of $T[i]$ with x
- 1 return

about $4n$, i.e., ***O(n)***

Average case (x present, on average at position $n/2$)

- $n/2$ assignments of i
- $n/2$ comparisons of i with n
- $n/2$ accesses to $T[i]$

First example: search in an array

How many elementary operations?

Input: Array T with n distinct elements, an element x
Output: the position of x in T

```
1: for i ← 0 to n − 1 do
2:     if T[i] = x then
3:         return i
4:     end if
5: end for
6: return not found
```

(the example algorithm of lecture 1)

Worst case (x absent or in last position)

- n assignments of i
- n comparisons of i with n
- n accesses to $T[i]$
- n comparisons of $T[i]$ with x
- 1 return

about $4n$, i.e., ***O(n)***

Average case (x present, on average at position $n/2$)

- $n/2$ assignments of i
- $n/2$ comparisons of i with n
- $n/2$ accesses to $T[i]$
- $n/2$ comparisons of $T[i]$ with x

First example: search in an array

How many elementary operations?

Input: Array T with n distinct elements, an element x
Output: the position of x in T

```
1: for i ← 0 to n − 1 do
2:     if T[i] = x then
3:         return i
4:     end if
5: end for
6: return not found
```

(the example algorithm of lecture 1)

Worst case (x absent or in last position)

- n assignments of i
- n comparisons of i with n
- n accesses to $T[i]$
- n comparisons of $T[i]$ with x
- 1 return

about $4n$, i.e., ***O(n)***

Average case (x present, on average at position $n/2$)

- $n/2$ assignments of i
- $n/2$ comparisons of i with n
- $n/2$ accesses to $T[i]$
- $n/2$ comparisons of $T[i]$ with x
- 1 return

First example: search in an array

How many elementary operations?

Input: Array T with n distinct elements, an element x
Output: the position of x in T

```
1: for i ← 0 to n − 1 do
2:     if T[i] = x then
3:         return i
4:     end if
5: end for
6: return not found
```

(the example algorithm of lecture 1)

Worst case (x absent or in last position)

- n assignments of i
- n comparisons of i with n
- n accesses to $T[i]$
- n comparisons of $T[i]$ with x
- 1 return

about $4n$, i.e., $\boldsymbol{O(n)}$

Average case (x present, on average at position $n/2$)

- $n/2$ assignments of i
- $n/2$ comparisons of i with n
- $n/2$ accesses to $T[i]$
- $n/2$ comparisons of $T[i]$ with x
- 1 return

about $2n$, i.e., $\boldsymbol{O(n)}$

Second example: binary search

Input: **Sorted** array T with n elements, an element x
Output: a position of x in T

```
 1: beg ← 0
 2: end ← n
 3: while beg < end do
 4:     mid ← ⌊(beg + end)/2⌋
 5:     if T[mid] > x then
 6:         end ← mid
 7:     else if T[mid] < x then
 8:         beg ← mid + 1
 9:     else
10:         return mid
11:     end if
12: end while
13: return not found
```

- If the array is **sorted**, much better than linear search: compare x with the **middle** element, then continue in the **left** or **right** half
- Interval still to be searched: $T[beg \ldots end - 1]$, initially the whole array, halved at each iteration
- Example: $T = [2, 3, 5, 7, 11, 13, 17, 19]$, $x = 13$: $11 < 13$, search $[13, 17, 19]$; $17 > 13$, search $[13]$; found

Proof of termination and correctness

- **Finite number of steps**: each time we enter the loop, the quantity $end - beg$ strictly decreases, until $beg = end$, so we cannot go into an infinite loop (and if x is absent, we correctly return *not found*)

Proof of termination and correctness

- **Finite number of steps**: each time we enter the loop, the quantity $end - beg$ strictly decreases, until $beg = end$, so we cannot go into an infinite loop (and if x is absent, we correctly return *not found*)
- Assume there exists i such that $T[i] = x$; we want to show that binary search returns a position of x

Proof of termination and correctness

- **Finite number of steps**: each time we enter the loop, the quantity $end - beg$ strictly decreases, until $beg = end$, so we cannot go into an infinite loop (and if x is absent, we correctly return *not found*)
- Assume there exists i such that $T[i] = x$; we want to show that binary search returns a position of x
- **Loop invariant**: at each step of the loop, we have $beg \leqslant i < end$ (proved by induction on the number of iterations, using the fact that T is sorted)

Proof of termination and correctness

- **Finite number of steps**: each time we enter the loop, the quantity $end - beg$ strictly decreases, until $beg = end$, so we cannot go into an infinite loop (and if x is absent, we correctly return *not found*)
- Assume there exists i such that $T[i] = x$; we want to show that binary search returns a position of x
- **Loop invariant**: at each step of the loop, we have $beg \leqslant i < end$ (proved by induction on the number of iterations, using the fact that T is sorted)
- Therefore, the **while** condition ($beg < end$) can never become false and the program **always eventually returns a value**

Proof of termination and correctness

- **Finite number of steps**: each time we enter the loop, the quantity $end - beg$ strictly decreases, until $beg = end$, so we cannot go into an infinite loop (and if x is absent, we correctly return *not found*)
- Assume there exists i such that $T[i] = x$; we want to show that binary search returns a position of x
- **Loop invariant**: at each step of the loop, we have $beg \leqslant i < end$ (proved by induction on the number of iterations, using the fact that T is sorted)
- Therefore, the **while** condition ($beg < end$) can never become false and the program **always eventually returns a value**
- When the value *mid* is returned, $T[mid] = x$, so **we indeed return a correct value**

(Worst-case) time complexity of binary search

- Assignments, tests, accesses to a cell of T are elementary operations ($O(1)$)

(Worst-case) time complexity of binary search

- Assignments, tests, accesses to a cell of T are elementary operations ($O(1)$)
- So the complexity depends **only on the number of times we enter the loop**

(Worst-case) time complexity of binary search

- Assignments, tests, accesses to a cell of T are elementary operations ($O(1)$)
- So the complexity depends **only on the number of times we enter the loop**
- If at the start of a loop iteration $end - beg = \ell$, at the next iteration, we have $end' - beg' = \ell'$ with one of the two cases:

(Worst-case) time complexity of binary search

- Assignments, tests, accesses to a cell of T are elementary operations ($O(1)$)
- So the complexity depends **only on the number of times we enter the loop**
- If at the start of a loop iteration $end - beg = \ell$, at the next iteration, we have $end' - beg' = \ell'$ with one of the two cases:
  - $\ell' = \left\lfloor \frac{beg+end}{2} \right\rfloor - beg \leqslant \frac{beg+end}{2} - beg = \frac{\ell}{2}$

(Worst-case) time complexity of binary search

- Assignments, tests, accesses to a cell of T are elementary operations ($O(1)$)
- So the complexity depends **only on the number of times we enter the loop**
- If at the start of a loop iteration $end - beg = \ell$, at the next iteration, we have $end' - beg' = \ell'$ with one of the two cases:
  - $\ell' = \left\lfloor \frac{beg+end}{2} \right\rfloor - beg \leqslant \frac{beg+end}{2} - beg = \frac{\ell}{2}$
  - $\ell' = end - \left\lfloor \frac{beg+end}{2} \right\rfloor - 1 \leqslant end - \frac{beg+end}{2} = \frac{\ell}{2}$

(Worst-case) time complexity of binary search

- Assignments, tests, accesses to a cell of T are elementary operations ($O(1)$)
- So the complexity depends **only on the number of times we enter the loop**
- If at the start of a loop iteration $end - beg = \ell$, at the next iteration, we have $end' - beg' = \ell'$ with one of the two cases:
  - $\ell' = \left\lfloor \frac{beg+end}{2} \right\rfloor - beg \leqslant \frac{beg+end}{2} - beg = \frac{\ell}{2}$
  - $\ell' = end - \left\lfloor \frac{beg+end}{2} \right\rfloor - 1 \leqslant end - \frac{beg+end}{2} = \frac{\ell}{2}$
- Thus we decrease ℓ at each iteration by a factor $\geqslant 2$

(Worst-case) time complexity of binary search

- Assignments, tests, accesses to a cell of T are elementary operations ($O(1)$)
- So the complexity depends **only on the number of times we enter the loop**
- If at the start of a loop iteration $end - beg = \ell$, at the next iteration, we have $end' - beg' = \ell'$ with one of the two cases:
  - $\ell' = \left\lfloor \frac{beg+end}{2} \right\rfloor - beg \leqslant \frac{beg+end}{2} - beg = \frac{\ell}{2}$
  - $\ell' = end - \left\lfloor \frac{beg+end}{2} \right\rfloor - 1 \leqslant end - \frac{beg+end}{2} = \frac{\ell}{2}$
- Thus we decrease ℓ at each iteration by a factor $\geqslant 2$
- So there are $\leqslant \log_2(n) + 1 =$ ***O*(log *n*)** iterations

(Worst-case) time complexity of binary search

- Assignments, tests, accesses to a cell of T are elementary operations ($O(1)$)
- So the complexity depends **only on the number of times we enter the loop**
- If at the start of a loop iteration $end - beg = \ell$, at the next iteration, we have $end' - beg' = \ell'$ with one of the two cases:
  - $\ell' = \left\lfloor \frac{beg+end}{2} \right\rfloor - beg \leqslant \frac{beg+end}{2} - beg = \frac{\ell}{2}$
  - $\ell' = end - \left\lfloor \frac{beg+end}{2} \right\rfloor - 1 \leqslant end - \frac{beg+end}{2} = \frac{\ell}{2}$
- Thus we decrease ℓ at each iteration by a factor $\geqslant 2$
- So there are $\leqslant \log_2(n) + 1 =$ ***O*(log *n*)** iterations
- We conclude with complexity $O(\log n)$: for $n = 10^6$, at most 20 iterations, against up to 10^6 for linear search

Binary search in Python and C

Python

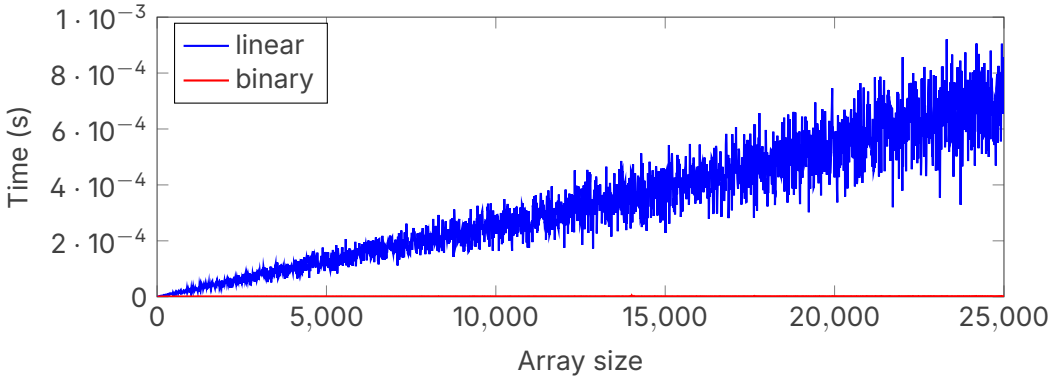
```python
def binary_search(t, x):
    # t must be sorted
    beg, end = 0, len(t)
    while beg < end:
        mid = (beg + end) // 2
        if t[mid] > x:
            end = mid
        elif t[mid] < x:
            beg = mid + 1
        else:
            return mid
    return -1
```

C

```c
// t must be sorted
int binary_search(int t[], int n, int x) {
  int beg = 0, end = n;
  while (beg < end) {
    int mid = beg + (end - beg) / 2;
    if (t[mid] > x) end = mid;
    else if (t[mid] < x) beg = mid + 1;
    else return mid;
  }
  return -1;
}
```

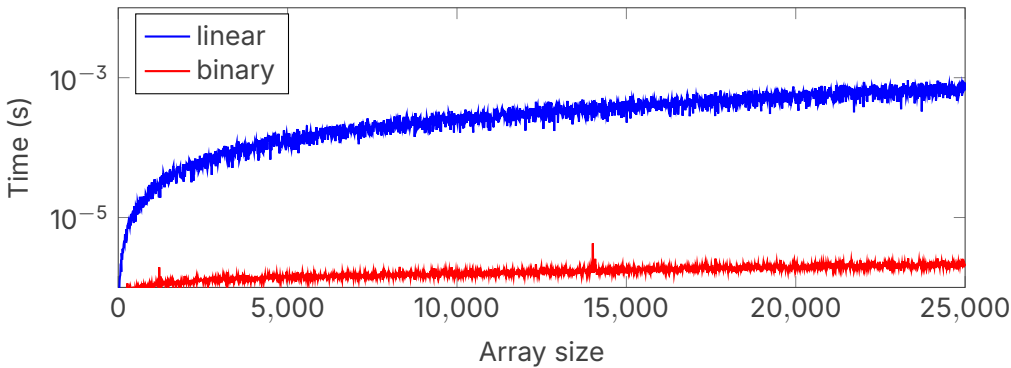
- In C, an array is passed together with its size `n` (arrays in C: next lecture); `beg + (end - beg) / 2` avoids the **integer overflow** (lecture 1) of `(beg + end) / 2` on very large arrays
- Binary search can also be written **recursively** (lecture 2) on the subarray $T[beg \ldots end - 1]$: same time complexity, but $O(\log n)$ space on the call stack

Linear search vs. binary search



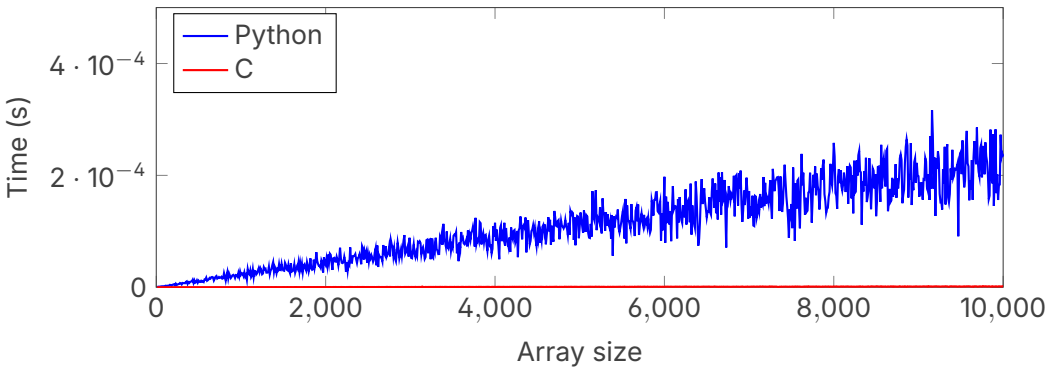
Python 3.10, average over 10 random searches in a sorted list of random integers

Linear search vs. binary search



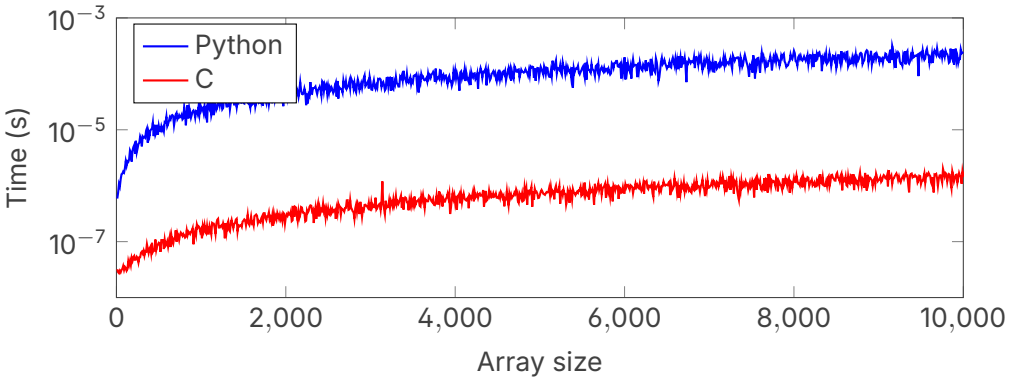
Python 3.10, average over 10 random searches in a sorted list of random integers; logarithmic scale on the *y* axis

Linear search: Python vs. C



Same algorithm, same complexity $O(n)$, but a constant factor of about 150 between Python 3.10 and C compiled with `gcc -O2`

Linear search: Python vs. C



Same algorithm, same complexity $O(n)$, but a constant factor of about 150 between Python 3.10 and C compiled with `gcc -O2`; logarithmic scale on the y axis

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Collections

From arrays to collections

- Both search algorithms work on an **array**: a fixed number of elements, each accessible by its position in $O(1)$
- But an algorithm often needs more: **add** or **remove** elements, test whether a value is **present**, look up a value **by a key**, etc.; and the cost of these operations depends on how the elements are stored
- In computer science, a **collection** is a group of **elements** stored together and accessible through a unified interface; collections allow us to store multiple values, organize them logically, and access, modify, and process them efficiently
- **Different kinds** of collections support different operations and are used for different purposes; so far we have only glimpsed the array T of lecture 1 and the Python list of the `append` example of lecture 2
- Plan: the **abstract** kinds of collections, then the collections of Python, with the **complexity** of each of their operations

Abstract kinds of collections

Ordered sequences elements are arranged in order; two subkinds:

- with **random access**, i.e., the possibility of directly accessing any element of the sequence by its position (**arrays**)
- without random access, i.e., only specific elements can be directly accessed, such as the first or the last

Value-based collections (e.g., **sets**): no underlying sequential order of the elements, we only care about whether an element is present in the collection

Associative arrays or **maps** or **key–value stores**: each element is a pair of a key and a value, access to elements is by their key, duplicate keys are (usually) not allowed

Mutable vs. immutable, fixed-size vs. dynamic

- **Mutable collections**: contents can be changed after creation
  - elements can be updated, and potentially added or removed
- **Immutable collections**: contents cannot be changed once created
  - any modification creates a new collection

- **Fixed-size collections**: the number of elements is determined at creation and cannot grow or shrink
- **Dynamic collections**: can change size during execution

Abstract collections vs. concrete data structures

- The kinds of collections we have just seen are **abstract**: they specify **which operations** are possible, not how (remember the definition of a **data structure** in lecture 1: an abstract object, its operations, and algorithms realizing them)
- They can be implemented in different ways using specific **data structures**; for example:
  - an ordered sequence may be implemented by an array or by a linked list
  - a set or an associative array may be implemented by a hash table or by a balanced tree
- The choice of data structure impacts the **complexity** of operations, which we now know how to measure
- We will present these data structures in a later lecture

Collections in programming languages

- Modern programming languages provide implementations of these collections in their **standard libraries**; for example:
  - in Python: `list`, `set`, `dict`…
  - in C++: `std::vector`, `std::set`, `std::map`…
  - in C: only fixed-size arrays; everything else is to be written by the programmer, or found in third-party libraries
- These implementations rely on efficient data structures under the hood
- Programmers must choose the right collection for the right task
- Other implementations are available in **third-party libraries** (e.g., NumPy arrays in Python)

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Python collections

Collections in Python

- Python includes a number of collections as **standard types** in the language
- Some are **mutable**, some are **immutable**
- Some are **dynamic**, some are **fixed-size**
- Usually flexible enough for most situations; occasionally, collections from third-party libraries are needed
- Their operations have a **complexity** that a Python programmer needs to know, to write efficient programs

Iterating over collections

Python

```
for x in collection:
    # Repeat the following code for each element
    # x of the collection (of type list, tuple,
    # set...). The variable x contains each element
    # in sequence
    ...
```

As with other loops, inside the `for` loop, you can use `continue` to skip to the next element or `break` to exit the loop early

Python collections

Name	Abstract type	Mutable?	Size
`tuple`	Random-access ordered sequence	No	Fixed
`range`	Random-access ordered sequence (integers)	No	Fixed
`list`	Random-access ordered sequence	Yes	Dynamic
`str`	Random-access ordered sequence (characters)	No	Fixed
`bytes`	Random-access ordered sequence (bytes)	No	Fixed
`bytearray`	Random-access ordered sequence (bytes)	Yes	Dynamic
`set`	Value-based collection	Yes	Dynamic
`frozenset`	Value-based collection	No	Fixed
`dict`	Associative array	Yes	Dynamic

Complexity of collections

- We are interested in the possible operations on Python collections (lists, sets, etc.) and their **complexity**
- Complexity is not specified by the language and may depend on the Python interpreter! But in practice, all major implementations agree; the complexities given below are those of CPython [Python Software Foundation, 2024]
- Complexity sometimes **amortized**: some rare operations are expensive, but the average cost over a long sequence of operations is what is given (see later lecture)
- For sets and dictionaries, the $O(1)$ given is an **average** over the possible values: the worst case is $O(n)$, but rare (see the lecture on hash tables)
- We will see how some of these data structures can be implemented in a later lecture; for now, one hint: in spite of their name, Python lists are implemented as **arrays**, not as linked lists

Tuples

Type	`tuple`	
Definition	ordered sequence of fixed size n, immutable	
Literals	`()` `(1,)` `(1, 2, 3)` `('a', 1, 3.14)`	
Access	`t[i]`	$O(1)$
Membership test	`x in t`	$O(n)$
Concatenation	`t1 + t2`	$O(n_1 + n_2)$
Length	`len(t)`	$O(1)$
Loop	`for x in t:`	$O(n)$

Elements of different types are allowed; the tuple itself cannot be changed, but a mutable element (e.g., a list) inside it can

Ranges

Type	`range`	
Definition	ordered sequence of integers, of fixed size *n*, immutable; from `start` (included) to `stop` (excluded) by steps of `step` (lecture 2)	
Construction	`range(stop)` `range(start, stop)` `range(start, stop, step)`	
Access	`r[i]`	***O*(1)**
Membership test	`x in r` (integer `x`)	***O*(1)**
Length	`len(r)`	***O*(1)**
Loop	`for i in r:`	***O*(*n*)**
Conversion	`list(r)`	***O*(*n*)**

The elements are **not stored** in memory but computed on demand from `start`, `stop`, and `step`: a range takes *O*(1) space whatever its length, which is why the `for i in range(n):` loops of lecture 2 cost nothing beyond the loop itself

Python “lists”

Type	`list`	
Definition	ordered sequence of variable size n, mutable	
Literals	`[]` `[1]` `[1, 2, 3]` `['a', 1, 3.14]`	
Access	`l[i]`	$O(1)$
Modification	`l[i] = 42`	$O(1)$
Append at end	`l.append(42)`	$O(1)$ (amortized)
Insert elsewhere	`l.insert(16, "toto")`	$O(n)$
Remove at end	`l.pop()`	$O(1)$
Remove elsewhere	`del l[16]`	$O(n)$
Membership test	`x in l`	$O(n)$
Concatenation	`l1 + l2`	$O(n_1 + n_2)$
Length	`len(l)`	$O(1)$
Loop	`for x in l:`	$O(n)$

List slices

Python also allows accessing a **slice** of a list, i.e., a sublist: `l[i:j]` denotes all elements of the list between positions *i* (inclusive) and *j* (exclusive). If *i* is omitted, it defaults to 0; if *j* is omitted, it defaults to `len(l)`

Access	`l[i:j]`	$O(j - i)$
Modification	`l1[i:j] = l2`	$O(n_1 + n_2)$
Deletion	`del l[i:j]`	$O(n)$

Accessing a slice creates a **new list**, a copy of the elements: `l[:]` (or `l.copy()`) is the way to copy a list. By contrast, `l2 = l1` does **not** copy: both variables refer to the **same** list (as for parameters passed by object reference, lecture 2), and modifying one modifies the other

Strings and byte arrays

- Strings (`str`) and byte arrays (`bytearray`, `bytes`) behave very similarly to lists, but only store characters or bytes
- But `str` and `bytes` are **immutable**, **fixed-size** sequences
- Access, slicing operations: **same operations, same complexity** as lists
- Consequence: building a string by repeated concatenation `s = s + c` in a loop copies the string each time, $O(n^2)$ in total in general (CPython sometimes optimizes this away, but do not rely on it); collect the pieces in a list and use `"".join(pieces)` instead, $O(n)$

Sets

Type	`set`	
Definition	value-based collection of variable size *n*	
Literals	`{1}` `{1, 2, 3}` `{'a', 1, 3.14}` (empty set: `set()`)	
Membership test	`x in s`	$O(1)$
Add	`s.add(42)`	$O(1)$
Remove	`s.remove(42)`	$O(1)$
Union	`s3 = s1 | s2`	$O(n_1 + n_2)$
Intersection	`s3 = s1 & s2`	$O(\min(n_1, n_2))$
Length	`len(s)`	$O(1)$
Loop	`for x in s:`	$O(n)$

Note that `{}` is an empty *dictionary*, not an empty set; elements must be **hashable** (explained in a later lecture), in practice **immutable** (numbers, strings, tuples of those… but not lists). `frozenset` is the immutable variant of `set`, same operations and complexities minus the modifications

Dictionaries

Type	`dict`	
Definition	associative array of variable size *n*, mapping keys to values	
Literals	`{}` `{'a': 1, 'b': 2}`	
Access	`d['a']`	***O*(1)**
Membership test	`'a' in d`	***O*(1)**
Insert/Update	`d['c'] = 42`	***O*(1)**
Remove	`del d['b']`	***O*(1)**
Length	`len(d)`	***O*(1)**
Loop	`for key in d:`	***O*(*n*)**

Keys must be **hashable**, as for set elements; values can be anything. Iterating over `d.items()` yields (key, value) pairs; since Python 3.7, keys are enumerated in **insertion order**

Comprehensions

- **Comprehension:** a way to define a collection by not giving the list of its elements (**extension**) but a characterization of them (**intension**), as in the mathematical notation $\{x^2 \mid x \in \mathbb{N}, x \text{ odd}\}$
- In Python, an expression building a list, a set, or a dictionary from any collection
- **Syntax:** `[expression for x in collection if condition]` for a list; with braces, `{expression for ...}` for a set and `{key: value for ...}` for a dictionary
- More concise than writing a loop!
- Same complexity as the corresponding loop

Python

```
marriages = {"Titi": "Tata", "Toto": None, "Tata": "Titi"}
singles = [p for p in marriages if marriages[p] is None]
# Multiplication table skipping every other row
[(x, y, x * y) for x in range(10) for y in range(10) if x % 2 == 0]
# Set of the squares of the odd numbers below 100
{x * x for x in range(100) if x % 2 == 1}
```

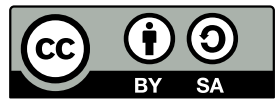
UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Coda

Wednesday's tutorial and references

- Wednesday's tutorial: computing by hand the complexity of small algorithms (loops, nested loops, loops with a variable doubling at each step), then implementing them in Python and C, **timing** them for growing input sizes, and comparing with the predicted complexity; practicing Python lists, sets, dictionaries, and comprehensions
- Reference: Cormen et al. [2022], chapters 2 (analyzing an algorithm) and 3 (asymptotic notation); Lutz [2025], the chapters on Python's core object types (lists, dictionaries, tuples, sets)

Licensing



Bibliography I

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 4th edition, 2022. ISBN 978-0-262-04630-5. URL https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/.

Mark Lutz. *Learning Python: Powerful Object-Oriented Programming*. O'Reilly Media, 6 edition, 2025. ISBN 1098171306. URL https://www.oreilly.com/library/view/learning-python-6th/9781098171301/.

Python Software Foundation. Time complexity. Python Wiki, https://wiki.python.org/moin/TimeComplexity, 2024.