Quality of an Algorithm & Python Collections

The Art of Computer Programming

Pierre Senellart

PSL UNIVERSITÉ PARIS

29 September 2025

What are Collections?

- 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
  - Access, modify, and process them efficiently
- Different kinds of collections support different forms of operations and are used for different purposes

Abstract Kinds of Collections

Ordered sequences Elements are arranged in order; two subkinds:

- With random access, i.e., possibility of directly accessing any element of the sequence (arrays)
- Without random access, i.e., only possible to directly access specific elements, such as the first or the last

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

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

Mutable vs. Immutable Collections

- 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 vs. Dynamic Collections

- Fixed-size collections:
  - Number of elements is determined at creation
  - Cannot grow or shrink
- Dynamic collections:
  - Can change size during execution

Abstract Collections vs. Concrete Data Structures

- The collections we have seen are abstract
- 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 a balanced tree
- The choice of data structure impacts the complexity of operations
- 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
- Examples:
  - In Python: `list`, `set`, `dict`...
  - In C++: `std::vector`, `std::set`, `std::map`...
- 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

Outline

Termination, Correctness, Efficiency

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

Termination, Correctness, Efficiency

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

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
  Descriptive complexity size of the smallest program
  Circuit complexity size of the smallest circuit implementing the program
- 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 in one CPU cycle (3 billion per second for a 3 GHz processor), once assembled

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 in one CPU cycle (3 billion per second for a 3 GHz processor), once assembled
- 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 in one CPU cycle (3 billion per second for a 3 GHz processor), once assembled
- 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 in one CPU cycle (3 billion per second for a 3 GHz processor), once assembled
- 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 of elementary operation can be made more formal (with the notion of a Turing machine or a Von Neumann machine), but we skip that for now

Outline

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

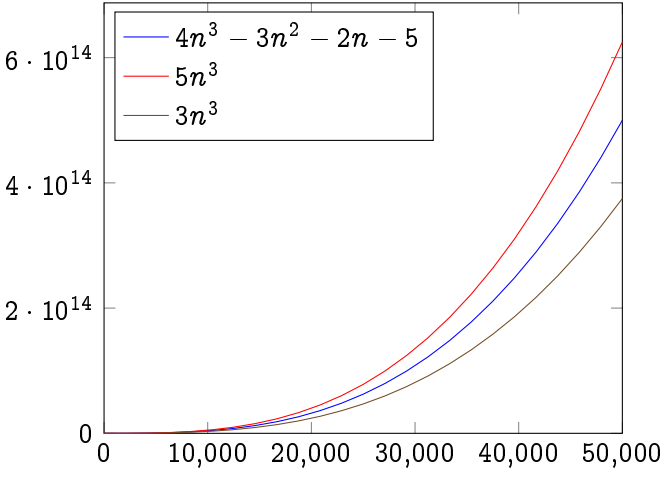
$$f(n) = O(g(n)) \quad \text{and} \quad f(n) = \Omega(g(n))$$

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

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

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

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



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: 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: 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: 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: 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: 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"

Additional remarks

- Since $\log_b n = \frac{\ln n}{\ln b}$ by definition, $\log_b n \in \Theta(\ln n)$ for any b: 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"
- 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

Outline

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, algorithm A performs $42 \times n$ elementary operations, then:

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

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
- Sometimes too restrictive, so we look at the average case: on average, for 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

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)

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)
- 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

In Practice: Solving a Problem Efficiently

- Design (mentally or on paper) the most efficient algorithm possible, using the most appropriate data structures
- If not obvious, prove its termination and correction

In Practice: Solving a Problem Efficiently

- Design (mentally or on paper) the most efficient algorithm possible, using the most appropriate data structures
- If not obvious, prove its termination and correction
- 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
- If not obvious, prove its termination and correction
- 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
- If not obvious, prove its termination and correction
- 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
- If not obvious, prove its termination and correction
- 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, the space complexity is $O(1)$: 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 input size

Outline

First Example: Search in an Array

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
```

How Many Elementary Operations?

First Example: Search in an Array

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
```

How Many Elementary Operations?

Worst case

First Example: Search in an Array

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

1: **for** $i \leftarrow 0$ **to** $n-1$ **do**
2: **if** $T[i] = x$ **then**
3: **return** i
4: **end if**
5: **end for**

How Many Elementary Operations?

Worst case

(x in last position)

First Example: Search in an Array

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
```

How Many Elementary Operations?

Worst case

(x in last position)

- n assignments of i

First Example: Search in an Array

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

1: **for** $i \leftarrow 0$ **to** $n-1$ **do**
2: **if** $T[i] = x$ **then**
3: **return** i
4: **end if**
5: **end for**

How Many Elementary Operations?

Worst case

(x in last position)

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

First Example: Search in an Array

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
```

How Many Elementary Operations?

Worst case

(x in last position)

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

First Example: Search in an Array

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
```

How Many Elementary Operations?

Worst case

(x 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

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
```

How Many Elementary Operations?

Worst case

(x 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

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
```

How Many Elementary Operations?

Worst case

(x 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

$4n + 1$, i.e., $O(n)$

First Example: Search in an Array

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
```

How Many Elementary Operations?

Worst case (x in last position) Average case

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

$4n + 1$, i.e., $O(n)$

First Example: Search in an Array

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
```

How Many Elementary Operations?

Worst case

(x 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

$4n + 1$, i.e., $O(n)$

Average case

(x on average at position $n/2$)

First Example: Search in an Array

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
```

How Many Elementary Operations?

Worst case

(x 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

$4n + 1$, i.e., $O(n)$

Average case

(x on average at position $n/2$)

- $n/2$ assignments of i

First Example: Search in an Array

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
```

How Many Elementary Operations?

Worst case

(x 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

$4n + 1$, i.e., $O(n)$

Average case

(x on average at position $n/2$)

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

First Example: Search in an Array

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
```

How Many Elementary Operations?

Worst case

(x 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

$4n + 1$, i.e., $O(n)$

Average case

(x 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

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
```

How Many Elementary Operations?

Worst case

(x 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

$4n + 1$, i.e., $O(n)$

Average case

(x 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

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
```

How Many Elementary Operations?

Worst case

(x 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

$4n + 1$, i.e., $O(n)$

Average case

(x 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

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
```

How Many Elementary Operations?

Worst case

(x 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

$4n + 1$, i.e., $O(n)$

Average case

(x 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

$2n + 1$, i.e., $O(n)$

Second Example: Binary Search

Input: Array T assumed to be sorted with n elements, an element x
Output: the 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
```

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

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
- Assume there exists i such that $T[i] = x$, we want to show that binary search returns this 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
- Assume there exists i such that $T[i] = x$, we want to show that binary search returns this x
- Loop invariant: at each step of the loop, we have $beg \leqslant i < end$ (proved by induction)

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
- Assume there exists i such that $T[i] = x$, we want to show that binary search returns this x
- Loop invariant: at each step of the loop, we have $beg \leqslant i < end$ (proved by induction)
- 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
- Assume there exists i such that $T[i] = x$, we want to show that binary search returns this x
- Loop invariant: at each step of the loop, we have $beg \leqslant i < end$ (proved by induction)
- 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 the 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 $\geqslant 2$

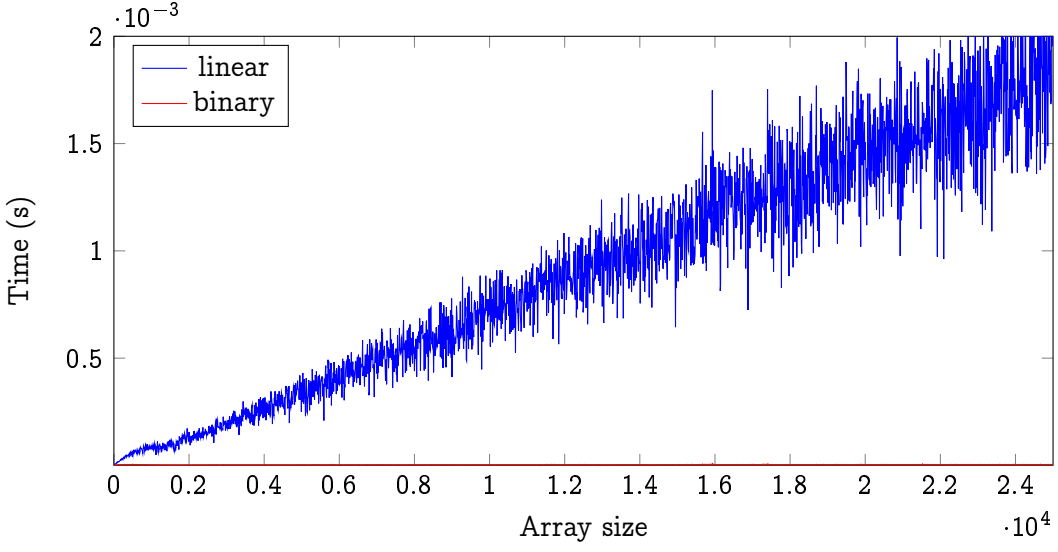
(Worst-Case) Time Complexity of Binary Search

- 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

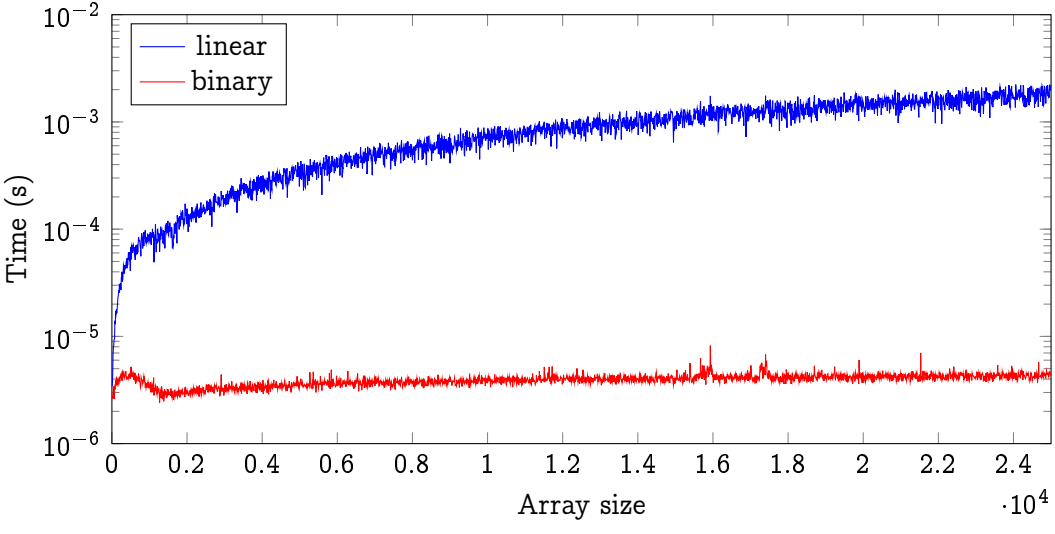
(Worst-Case) Time Complexity of Binary Search

- 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 $O(\log n)$

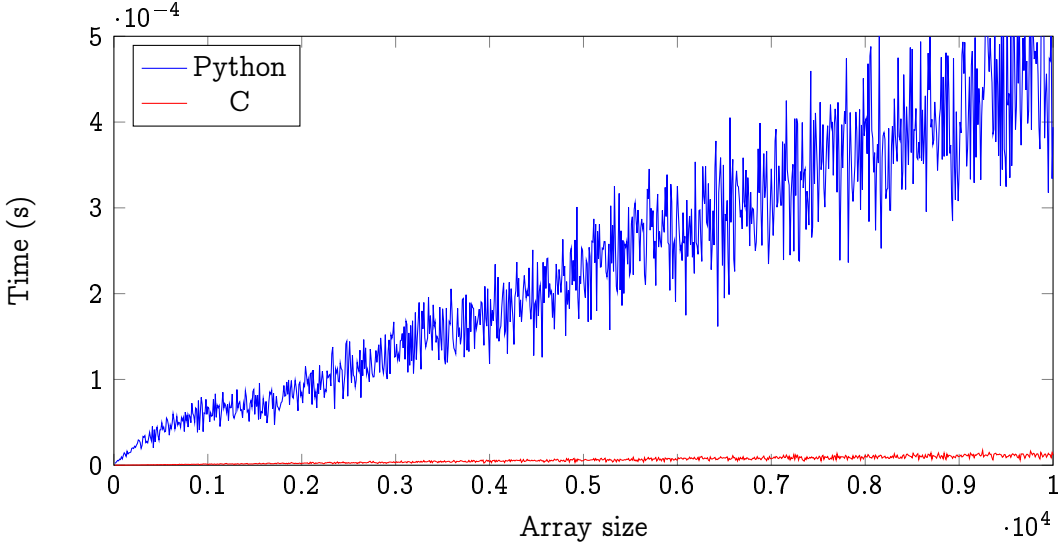
Linear search vs binary search



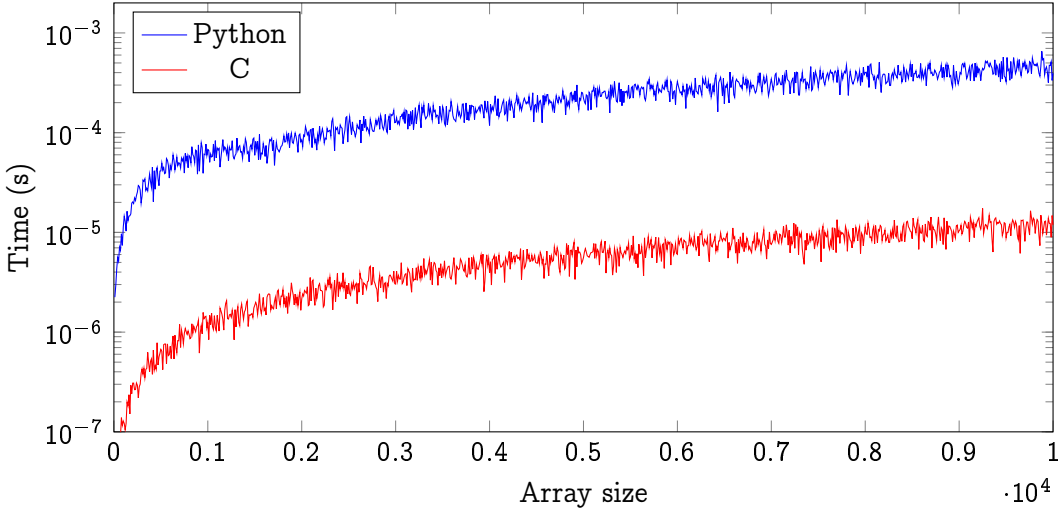
Linear search vs binary search



Linear search: Python vs C



Linear search: Python vs C



Outline

Collections in Python

- Python includes a number of collections as standard types in the language
- Most are mutable, but some are immutable
- Most are dynamic, but some are fixed-size
- Usually flexible enough for most situations, occasionally need collections from third-party libraries

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
`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
`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, no difference
- Complexity sometimes amortized (see later lecture)
- We will see how some of these data structures can be defined in a later lecture

Tuples

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

Python "Lists"

Type	`list`	
Definition	Ordered sequence of variable size n, mutable elements	
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)`	$O(1)$
Insert elsewhere	`l.insert(16,"toto")`	$O(n)$
Remove at end	`l.pop()`	$O(1)$
Remove elsewhere	`del l[16]`	$O(n)$
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, a sublist of the list: `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]`	$O(n)$

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

Sets

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

Dictionaries

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

List Comprehensions

- Comprehension: a way to define a collection by not giving the list of its elements (extension) but a characterization of them
- In Python, an expression defining complex lists
- Syntax: `[expression for x in collection if condition]`
- 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] == 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]
```

Licensing