Introduction to algorithmics and data structures
L3 Algorithmics and Programming

Pierre Senellart

27 September 2018
Outline

Algorithmics & programming

Algorithmic complexity

Elementary data structures

Amortized complexity

References
Algorithmics & programming

- Comes from the name of محمد بن موسى الخوارزمي
Algorithmics & programming

- Comes from the name of محمد بن موسى الخوارزمي (Muhammad ibn Musa al-Khwarizmi), a Persian scientific from the 9th century
Algorithmics & programming

- Comes from the name of محمد بن موسى الخوارزمی (Muhammad ibn Musa al-Khwarizmi), a Persian scientific from the 9th century
- ... who also gave the word algebra (الجبر, bone-setting, rejoining in Arabic), from the title of one of his book on solving equations
Algorithmics & programming

- Comes from the name of محمد بن موسى الخوارزمی (Muhammad ibn Musa al-Khwarizmi), a Persian scientific from the 9th century
- ... who also gave the word algebra (الجبر, bone-setting, rejoining in Arabic), from the title of one of his book on solving equations
- An algorithm is the formal specification of the way to solve a given problem: from some input, how to produce the output corresponding to the solution of a problem via elementary operations
Algorithmics & programming

- Comes from the name of محمد بن موسى الخوارزمي (Muhammad ibn Musa al-Khwarizmi), a Persian scientific from the 9th century
- ... who also gave the word algebra (الجبر, bone-setting, joining in Arabic), from the title of one of his book on solving equations
- An algorithm is the formal specification of the way to solve a given problem: from some input, how to produce the output corresponding to the solution of a problem via elementary operations
- Algorithmics is the study of algorithms: algorithm design, analysis of their performance, etc.
Algorithmics & programming

- Comes from the name of محمد بن موسى الخوارزمي (Muhammad ibn Musa al-Khwarizmi), a Persian scientific from the 9th century
- ... who also gave the word algebra (الْجَبَر, bone-setting, rejoining in Arabic), from the title of one of his book on solving equations
- An algorithm is the formal specification of the way to solve a given problem: from some input, how to produce the output corresponding to the solution of a problem via elementary operations
- Algorithmics is the study of algorithms: algorithm design, analysis of their performance, etc.
- Programming is the way to turn an algorithm into code in a computer language, so as to execute the algorithm on concrete data
Algorithmics vs programming

- Every algorithm is **implementable**: it must be described in precise enough terms so that the programming the algorithm is unambiguous
- ... but this does **not** mean that the program implementing this algorithm is **easy to write**, as the programmer must take into account machine limits, quirks of the programming language, low-level objects vs high-level concepts, etc.
- **Algorithm**: abstraction of what is **implementable**
- The programming language has no influence on what is implementable; all usual programming languages have the same **expressive power** (Turing-complete)
- ... but a programming language has an impact on **ease** (cf. http://pierre.senellart.com/travaux/languages/languages.xml) or **efficiency** of implementation
**Data structure**

- **Basic** element used in more complex algorithms, reused in various algorithms to solve various problems
- Formal specification of an abstract mathematical **object** (list, set, function, graph, matrix, etc.), of possible **operations** on this object (insertion, enumeration, inversion, etc.) and of **algorithms** realizing them
- Implementable **building block**, often in the form of a **class** in object-oriented programming
- Often possible to design different data structures for the same mathematical object, with different **efficiency**
Outline

Algorithmics & programming

Algorithmic complexity

Elementary data structures

Amortized complexity

References
How to measure the efficiency of an algorithm?

- Attempt at characterizing, from the description of an algorithm, the efficiency of a program that implements it; or, from the description of a problem, the efficiency of a program that implements an algorithm solving the problem.

- Different notions of efficiency, different notions of complexity:

  - **Time complexity** computation time of a sequential program
  - **Space complexity** memory space used by a program program
  - **Communication complexity** volume of data exchanged by a distributed system
  - **Descriptive complexity** shortest program length
  - **Circuit complexity** size of the electronic circuit implementing the algorithm

- In this course: first two only (and mostly the first!)
How to compute time complexity

- One assumes every elementary operations appearing in the description of an algorithm:
  - arithmetic operations
  - variable assignments
  - comparisons
  - tests
  - etc.

  takes elementary time, bounded by a constant $C$

- One sums the number of elementary operations made, as a function of the input size $n$, e.g., $42 \times n$

- On deduces a bound, here $42 \times n \times C$, on the total time of the algorithm
How to compute time complexity

- One assumes every elementary operations appearing in the description of an algorithm:
  - arithmetic operations
  - variable assignments
  - comparisons
  - tests
  - etc.

  takes elementary time, bounded by a constant $C$

- One sums the number of elementary operations made, as a function of the input size $n$, e.g., $42 \times n$

- On deduces a bound, here $42 \times n \times C$, on the total time of the algorithm

- Elementary operation times can be made formal (with the notions of Turing machines, or of Von Neumann machines), but we will skip this
Let $f : \mathbb{N} \to \mathbb{R}_+$, $g : \mathbb{N} \to \mathbb{R}_+$ be two functions

One writes $f(n) \in O(g(n))$ (or $f(n) = O(g(n))$) if

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

One writes $f(n) \in \Omega(g(n))$ (or $f(n) = \Omega(g(n))$) if

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

One writes $f(n) \in \Theta(g(n))$ (or $f(n) = \Theta(g(n))$) if

$$f(n) = O(g(n)) \quad \text{and} \quad f(n) = \Omega(g(n))$$
**Asymptotic time complexity**

- One uses the $O()$, $\Omega()$, $\Theta()$ notation and bounds that have been established to indicate the complexity of an algorithm while **neglecting** $C$ and other constants.
- For example, if the elementary operation time $\tau$ is bounded by:

  $$C_1 \leq \tau \leq C_2$$

- ... and if on all inputs, algorithm A makes $42 \times n$ operations, then:

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

  so that $T(\mathcal{A}, n) = \Theta(n)$
Complexity in the worse case, average case

- Usually, one looks for an upper bound on the time of an algorithm, and one looks at the **worst case** complexity: an upper bound that holds on any input.
- Sometimes too restrictive, and then one looks at the **average case**: on average, for all inputs of a given size, what is a bound on the complexity?
- Assumes that all inputs have the same probability, which is debatable.
Simple example: searching an array

**Input:** Array $T$ with $n$ distinct elements, an element $x$ within $x$

**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?
Simple example: searching an array

Input: Array \( T \) with \( n \) distinct elements, an element \( x \) within \( x 

Output: the position of \( x \) in \( T \)

1: \textbf{for} \( i \leftarrow 0 \) to \( n - 1 \) \textbf{do}
2: \hspace{1em} \textbf{if} \( T[i] = x \) \textbf{then}
3: \hspace{2em} \textbf{return} \( i \)
4: \hspace{1em} \textbf{end if}
5: \hspace{1em} \textbf{end for}

How many elementary operations?

Worst case

\((x \text{ in last position})\)

\( n \) 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., \( O(n) \)
Simple example: searching an array

**Input:** Array $T$ with $n$ distinct elements, an element $x$ within $x$

**Output:** the position of $x$ in $T$

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

How many elementary operations?

<table>
<thead>
<tr>
<th>Worst case</th>
<th>Average case</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(x$ in last position)</td>
<td>$(x$ in expected position $n/2$)</td>
</tr>
<tr>
<td>$n$ assignments of $i$</td>
<td>$n/2$ assignments of $i$</td>
</tr>
<tr>
<td>$n$ comparisons of $i$ with $n$</td>
<td>$n/2$ comparisons of $i$ with $n$</td>
</tr>
<tr>
<td>$n$ accesses to $T[i]$</td>
<td>$n/2$ accesses to $T[i]$</td>
</tr>
<tr>
<td>$n$ comparisons of $T[i]$ with $x$</td>
<td>$n/2$ comparisons of $T[i]$ with $x$</td>
</tr>
<tr>
<td>1 return</td>
<td>1 return</td>
</tr>
</tbody>
</table>

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

$2n + 1$, i.e., $O(n)$
Asymptotic complexity vs real efficiency

In practice:

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

• Sometimes, an algorithm in $O(n^2)$ (i.e., $\leq \alpha n^2$) may be more efficient than an algorithm in $O(n)$ (i.e., $\leq \beta n$) for common input size $n$, because $\alpha \ll \beta$.

• Sometimes, **worst-case** complexity is high, but average-case complexity is low and may be the only thing that matters in practice.

• The **programming language used** has a real impact on performance (but, generally, by a constant factor, potentially large).

• Time complexity does not say anything on the potential for parallelization or distribution of an algorithm – **other complexity notions** are necessary.
Linear vs binary search in a sorted array

![Graph showing linear vs binary search times](image-url)
Linear vs binary search in a sorted array

![Graph showing linear vs binary search times](image)

**X-axis:** Array size

**Y-axis:** Time (s)

- **Linear**
- **Binary**

- **Time comparison:**
  - Linear search time increases significantly with array size.
  - Binary search time remains relatively constant.

**Analysis:**
- Linear search is inefficient for large datasets.
- Binary search is more efficient and scalable.
Linear search: Python vs C

![Graph showing comparison between Python and C for linear search time vs array size](image-url)
Linear search: Python vs C

![Graph showing comparison between Python and C for linear search](chart.png)
Space complexity

- Same as time complexity, except one counts elementary uses of memory space instead of time of elementary operations.
- One often makes simplifying assumptions, such as the fact that any integer fits in constant space.
- One does not count the space needed to represent the input.
- One also uses $O()$, $\Omega()$, $\Theta()$ as a summary of asymptotic complexity.
- For example, array search, space complexity of $O(1)$: just store the variable $i$ in memory (in addition to the inputs $T$ and $x$), which requires an elementary space, independent of the size of the input.
Outline

Algorithmics & programming

Algorithmic complexity

Elementary data structures

Amortized complexity

References
Containers

- Data structures storing a list $L = (l_1, \ldots, l_n)$ of elements (e.g., integers, floating-point numbers, complex objects, etc.)

- Diverse specifications of such a structure, allowing different operations, with different efficiency:
  - Random access: given $i$, access $l_i$
  - Access at the beginning ($l_1$), at the end ($l_n$)
  - Insertion at the beginning (before $l_1$), at a random position (between $l_i$ and $l_{i+1}$), at the end (after $l_n$)
  - Deletion of the first element ($l_1$), of a random element ($l_i$), of the last element ($l_n$)
  - Variant: assuming an ordering on the elements, we suppose $l_1 \leq \ldots \leq l_n$. Access to an ordered element, insertion while following order.
Fixed array

- Contiguous memory area, of a fixed size, pre-allocated
- Random access in $O(1)$
- Insertion, deletion impossible
- As compact as possible, no lost space
- Corresponds to classic arrays of programming languages:
  - bracketed arrays in C or Java
  - std::array in C++ 2011
  - numpy.array in Python
Linked list

- Each element is stored in a link, which also contains a pointer to the next element
- Also maintain a pointer to the first link
- Access to first element in $O(1)$
- Random access or access to last element in $O(n)$
- Insertion in a random position (once the previous element has been accessed) in $O(1)$
- Deletion of the first element in $O(1)$
- Deletion in $O(1)$ if the previous element is known, in $O(n)$ otherwise
- std::forward_list in C++ 2011
Doubly linked list

- Each element is stored in a link, which also contains pointers to the previous and next elements
- Also maintain a pointer to the first and last links
- Access to first/last element in $O(1)$
- Random access in $O(n)$
- Insertion in a random position (once the element has been accessed) in $O(1)$
- Deletion of a random element (once it has been accessed) in $O(1)$
- In programming languages:
  - std::list in C++
  - java.util.LinkedList in Java
Stack (or LIFO for last-in-first-out)

- **Abstract** data structure, that can be implemented in different ways, e.g., with a singly linked list
- **Only possible operations:**
  - Access to **first** element in $O(1)$
  - Insertion **at the beginning** in $O(1)$
  - Deletion of the **first** element in $O(1)$
- **In programming languages:**
  - `std::stack` in C++
  - not explicit in Python, but standard lists can be used
  - `java.util.Stack` in Java
Queue (or FIFO for first-in-first-out)

- **Abstract** data structure, that can be implemented in different ways, e.g., with a doubly linked list
- **Only possible operations:**
  - Access to last element in $O(1)$
  - Insertion at the beginning in $O(1)$
  - Deletion of the last element in $O(1)$
- In programming languages:
  - `std::queue` in C++
  - not explicit in Python, but `collections.deque` can be used
  - `java.util.Queue` interface in Java
Unbalanced binary search tree

- Stores an ordered list of elements
- Binary tree: every node points towards at most two children, a pointer to the root is kept
- For every node storing element \( l \), the subtree rooted at the left child (if it exists) contains elements \( \leq l \), and the subtree rooted at the right child elements \( \geq l \)
- Access, insertion, deletion of a given element: \( O(d) \) where \( d \) is the depth of the tree
- No bound on this depth!
Outline

Algorithmics & programming

Algorithmic complexity

Elementary data structures

Amortized complexity

References
Amortized complexity

- Up to this point, complexity is measured separately for each operation on a data structure.
- Sometimes, not possible to bound the time of each individual operation, but possible to bound the average time within a sequence of operations.
- Consider a sequence of \( n \) operations \( o_1, \ldots, o_n \) on a data structure, with cost \( c_1, \ldots, c_n \).
- Compute the average complexity of an operation \( o_i \), i.e.,

\[
\frac{1}{n} \sum_{i=1}^{n} c_i
\]

- This is called the amortized complexity.
- Only makes sense if a precise definition of the sequence type considered is given.
Potential method

- Associate a potential $\Phi(X)$ (real number) to a data structure $X$
- Consider a sequence of $n$ operations $o_1, \ldots, o_n$, of real costs $c_1, \ldots, c_n$, and corresponding data structures $X_0, \ldots, X_n$
- Define $\hat{c}_i := c_i + \Phi(X_i) - \Phi(X_{i-1})$
- Then: $\frac{1}{n} \sum_{i=1}^{n} \hat{c}_i = \frac{1}{n} \sum_{i=1}^{n} c_i + \frac{1}{n} (\Phi(X_n) - \Phi(X_0))$
- If $\hat{c}_i = O(f(|X|))$, then we also have $\frac{1}{n} \sum_{i=1}^{n} c_i = O(f(|X|))$
  if for all $n$, $\Phi(X_n) \geq \Phi(X_0)$
- Often, one takes $X_0$ the empty data structure and $\Phi(X_0) := 0$, which gives the condition $\Phi(X_n) \geq 0$
Application: dynamic array

- Pointer towards a classic array of capacity $c + \text{integer } n \leq c$ storing the size really used
- **Random access** in $O(1)$
- Deletion of the last element in $O(1)$ by decreasing $n$
- Insertion at the end in **amortized $O(1)$ complexity** (see further)
- Called vector, array, array list in programming languages:
  - `std::vector` in C++
  - lists and array in standard Python, or `numpy.ndarray`
  - `java.util.Vector` and `java.util.ArrayList` in Java
Insertion at the end of a dynamic array

Input: array $T$ of capacity $c$ and size $n$, element $x$
Output: array $T$ of size $n + 1$ with $T[n] = x$

if $n = c$ then
allocate new array $T'$ of size max$(2 \times c, 1)$
for $i \leftarrow 0$ to $n - 1$ do
  $T'[i] \leftarrow T[i]$
end for
deallocate array pointed by $T$
make $T$ point to $T'$
c $\leftarrow$ max$(2 \times c, 1)$
end if
$T[n] \leftarrow x$
n $\leftarrow n + 1$
Analysis of amortized complexity

Blackboard analysis for a sequence of $n$ insertions, discussion of the case of arbitrary sequences.
Insertion in a std::vector in C++

Number of inserted elements

Cumulated time (s)
Outline

Algorithmics & programming

Algorithmic complexity

Elementary data structures

Amortized complexity

References
References

- **Generalities** on algorithmics: Chap. 1 of [Cormen et al., 2009, 2010]
- Basics of **complexity analysis** of an algorithm: Chap. 2 and 3 of [Cormen et al., 2009, 2010]
- Elementary **data structures**: Chap. 10 of [Cormen et al., 2009, 2010]
- **Amortized complexity**: Chap. 17 of [Cormen et al., 2009, 2010]

Used resources

The image of a queue is due to Vegpuff (Wikimedia), CC-BY-SA-3.0.