# Algorithmics for Biology

Département Génie Biologique
GB4 – year 2023–2024

Jean-Paul Comet

Université Côte d'Azur

19/01/2024





# Teaching organization

- Lectures : 9 sessions of 1 hours 30
- TDs : 9 sessions of 1 hours 30
- teachers :
  Jean-Paul Comet    Jean-Paul.Comet@univ-cotedazur.fr
  Lisa Guzzi    lisa.guzzi@etu.univ-cotedazur.fr

| | Date | hours | Lecture | TDs |
|---|---|---|---|---|
| 1 | 19/01/2024 | 13h30-16h45 | JPC | LG |
| 2 | 09/02/2024 | 8h30-11h45 | JPC | LG |
| 3 | 16/02/2024 | 8h30-11h45 | JPC | LG |
| 4 | 08/03/2024 | 8h30-11h45 | JPC | LG |
| 5 | 12/03/2024 | 15h15-16h45* | JPC | LG |
| 6 | 29/03/2024 | 8h30-11h45 | JPC | LG |
| 7 | 05/04/2024 | 13h30-16h45 | JPC | LG |
| 8 | 11/04/2024 | 8h30-11h45 | JPC | LG |
| 9 | 19/04/2024 | 8h30-11h45 | JPC | LG |

- Evaluation : Final exam (3 hours), 23/04/2024 13h30-16h30    70%
- 4 TD report, to finish at home    30%
- course material + TD + annals :
  https://www.i3s.unice.fr/~comet/SUPPORTS/





# Plan







# Generality

An algorithm is a sequence of actions to be performed by a machine or automaton in a finite amount of time, to achieve the desired result.

- finite sequence of instructions
- inputs / outputs

sort an array – ***insertion sort***

```
insertion_sort (double A[], int n)
{
   for (j=1; j<n; j++) {
      key = A[j];
      /* insertion of A[j] in the sorted sequence A[0...(j-1)] */
      i=j-1;
      while (i>=0) && (A[i]>key) {
         A[i+1] = A[i];
         i = i-1;
      }
      A[i+1] = key;
   }
}
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| j=1 : | 5 | **2** | 4 | 1 | 3 | 2 | 4 |
| j=2 : | 2 | 5 | **4** | 1 | 3 | 2 | 4 |
| j=3 : | 2 | 4 | 5 | **1** | 3 | 2 | 4 |
| j=4 : | 1 | 2 | 4 | 5 | **3** | 2 | 4 |
| j=5 : | 1 | 2 | 3 | 4 | 5 | **2** | 4 |
| j=6 : | 1 | 2 | 2 | 3 | 4 | 5 | **4** |
| j=7 : | 1 | 2 | 2 | 3 | 4 | 4 | 5 |

# Complexity analysis

- Complexity analysis shows whether one algorithm is more efficient than another.
- This analysis must be independent of the physical resources used (processor, memory access time).

**Complexity** ≡ number of steps **required** to solve the problem for an input of a given size.

What's the point of complexity ?

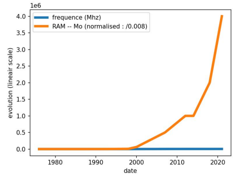
- Plan the resources required for an algorithm
- What are the critical resources ?
  the time, the memory, (the bandwidth of a communication)
- Complexity will depend on the machine model. Generally
  - random access memory (RAM)
  - a single processor
  If this model changes, so does the complexity, since you may have to take into account communication times between processors and/or the time it takes to access information in memory.

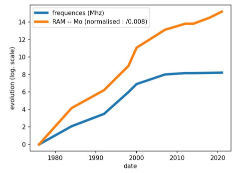




# Complexity analysis

| | | | |
|---|---|---|---|
| 1976 | 1 Mhz | 8 Ko | 1 core |
| 1984 | 8 Mhz | 512 Ko | 1 core |
| 1992 | 33 Mhz | 4 Mo | 1 core |
| 1998 | 400 Mhz | 64 Mo | 1 core |
| 2000 | 1 Ghz | 512 Mo | 1 core |
| 2007 | 3 Ghz | 4 Go | 1/2 cores |
| 2012 | 3.5 Ghz | 8 Go | 1/2/4 cores |
| 2014 | 3.5 Ghz | 8 Go | 2/4/8 cores |
| 2018 | 3.6 Ghz | 16 Go | 8 cores |
| 2021 | 3.7 Ghz | 32 Go | 10 cores |




Insertion sort execution time depends on the input :

- on the number of elements to be sorted
- on the nature of the array :
  - if the elements are already sorted, very quickly
    the shifting is no longer necessary, and the # of comparisons is very low.
  - if sorted in reverse : much longer





- In general, execution time increases with input size.
  execution time = $f$(input size)
- input *size*
  - for an array : number of elements
  - for a graph : (number of vertices, number of arcs) ...
- To estimate execution time :
  - execution time for each elementary instruction

Example : `Tri_insertion`

```
def Tri_insertion (array):
  n = len(array)
  for j in range(n):
    key = array[j]
    i=j-1
    while(i>=0) and (A[i]>key):
      A[i+1] = A[i]
      i = i-1
    A[i+1] = key
  return(array)
```

| cost | no. of passes |
|---|---|
| $c_2$ | 1 |
| $c_3$ | $n-1$ |
| $c_4$ | $n-1$ |
| $c_5$ | $n-1$ |
| $c_6$ | $\Sigma(j-1)$ |
| $c_7$ | $\Sigma(j-1)$ |
| $c_8$ | $\Sigma(j-1)$ |
| $c_9$ | $n-1$ |
| $c_{10}$ | 1 |

The overall execution time is then given by the formula :

$$t = c_1(n-1) + c_2(n-1) + \ldots$$





Remarks :

- If the array is already sorted : complexity ***linear.***
  This is the most favorable case.
- If the array is sorted in reverse : exact complexity calculable.
  - time proportional to the square of $n$.
  - The algorithm is said to be ***quadratic***.

As execution time depends $\begin{cases} \text{on the size of the input} \\ \text{on the nature of the input} \end{cases}$,
it is difficult to give a complexity independent of the input.
We're interested in execution time ***in the worst case*** :

- upper bound for any input of the same size,
- for certain algorithms, the worst happens quite often (e.g., if you're looking for information in a database that doesn't contain it),
- the average case is often as bad as the worst case (e.g. insertion sorting).

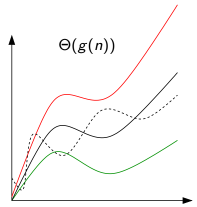
Simplification of the expression found :

- The real cost of each instruction is neglected,
- We neglect the abstract cost ($c_i$) of each instruction,
- We're interested in the order of magnitude of the execution time. We retain only the dominant term when $n$ is very large.
- Finally, we neglect the coefficient in front of this term.





# Notations




1. Notation $\Theta(g(n))$ : Asymptotic Approximate Bound

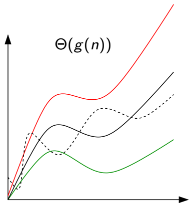
$$\Theta(g(n)) = \left\{ f(n) \mid \begin{array}{l} \exists c_1 > 0 \\ \exists c_2 > 0 \\ \exists n_0 > 0 \end{array}, \text{ s.t. } \begin{array}{c} 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \\ \forall n \geq n_0 \end{array} \right\}$$

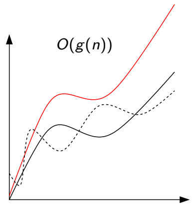
Note that $f(n) = \Theta(g(n))$ for $f \in \Theta(g(n))$.
« $f(n)$ is equal to $g(n)$ to within one constant factor. »
$g(n)$ is an ***approximate asymptotic bound for*** $f$.





# Notations




2. Notation $O(g(n))$ : Asymptotic Upper Bound

$$O(g(n)) = \left\{ f(n) \mid \begin{array}{l} \exists c > 0 \\ \exists n_0 > 0 \end{array}, \text{ s.t. } \begin{array}{c} 0 \leq f(n) \leq c g(n), \\ \forall n \geq n_0 \end{array} \right\}$$

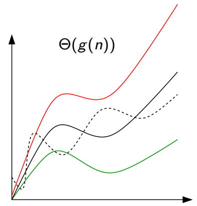
This is an ***upper bound to within one constant***.

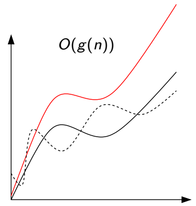
- $\Theta(g(n)) \subset O(g(n))$
- $\Theta(n) \subset O(n^2)$. Be careful.

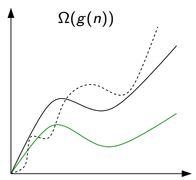




# Notations



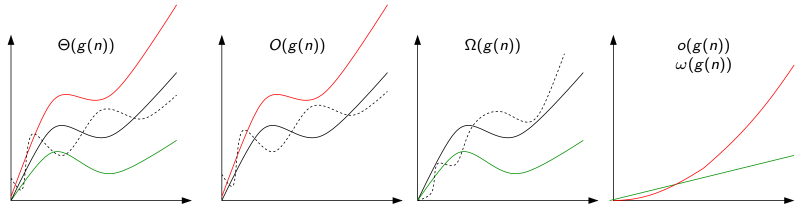




3. Notation $\Omega(g(n))$ : Asymptotic Lower Bound

$$\Omega(g(n)) = \left\{ f(n) \mid \begin{array}{l} \exists c > 0 \\ \exists n_0 > 0 \end{array}, \text{ s.t. } \begin{array}{c} 0 \leq c g(n) \leq f(n), \\ \forall n \geq n_0 \end{array} \right\}$$

This is an ***lower bound to within one constant***.

# Notations




4. Notation $o(g(n))$ : non-asymptotically approximated upper bound

$$o(g(n)) = \left\{ f(n) \mid \begin{array}{l} \forall c > 0 \\ \exists n_0 > 0 \end{array} \text{ s.t. } \begin{array}{c} 0 \le f(n) < cg(n), \\ \forall n \ge n_0 \end{array} \right\}$$

$f(n)$ becomes negligible in front of $g(n)$ as $n$ tends to $+\infty$. Examples :
For example, $2n = o(n^2)$ and $2n = O(n^2)$.
On the other hand, $2n^2 \neq o(n^2)$ and $2n^2 = O(n^2)$.

5. Notation $\omega(g(n))$ : non-asymptotically approximated lower bound

$$f(n) \in \omega(g(n)) \iff g(n) \in o(f(n))$$





# Divide and Conquer

Many algorithms have a recursive structure :

- recursive calls to very similar sub-problems,
- these calls separate the problem into several similar subproblems of smaller size.
- they solve the sub-problems recursively
- then combine the solutions of the sub-problems to calculate the solution to the problem.

There are three steps to each level :

- Divide the problem,
- Reign in the sub-problems by solving them recursively,
- Combine sub-problem solutions.





# Divide and conquer : Merge sorting.

- Divide : divide the array into 2 sub-arrays of approximately same size.
- Reign : sort each of the sub-arrays.
- Combine : merge the two previously sorted sub-arrays.

**Remark :**

a- If size(table) $\leq 1$ : it's sorted, nothing to do. (basic case)

b- Main stage : the ***fusion***.

`MERGE(A,p,q,r)` where `A` is an array, `p,q,r` are s.t. $p \leq q < r$.

- `A[p..q]` and `A[q+1..r]` are supposed to be sorted.
- it merges them to form `A[p..r]` sorted.

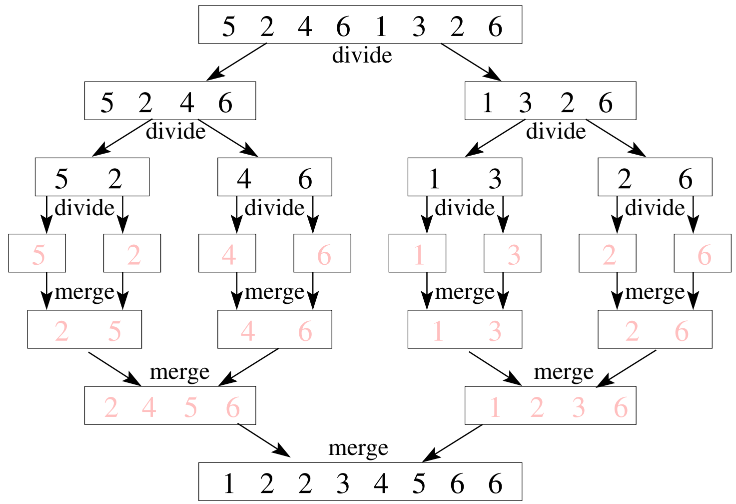
It's easy to write an algorithm in $\Theta(r - p + 1)$ that performs this fusion (leave as an exercise).

```
def Merge_sort (A, p, r):
   if (p<r) :
      q = (int) (q+p)/2
      Merge_sort(A,p,q)
      Merge_sort(A,q+1,r)
      MERGE(A,p,q,r)
```





# Divide and conquer : Merge sorting.



# Complexity of divide-and-conquer algorithms

Execution time can often be written as a recurrence equation that describes the overall execution time for a problem of size $n$ as a function of the execution time for smaller inputs.

Let $T(n)$ be the execution time for an input of size $n$.

- If the size is reduced ($n \leq n_0$) : direct solution, calculable in $Theta(1)$.
- Assume that :
  - we divide the problem into $a$ ss-pb each of size $n/b$.
  - we need $D(n)$ to divide the problem, and
  - we need $C(n)$ to construct the final solution.

The recurrence is then :

$$T(n) = \begin{cases} \Theta(1) & \text{if} \quad n \leq n_0 \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

**Merge sort :**

- Divide : center index calculation : $D(n) = \Theta(1)$
- Reign : $2 \times T(n/2)$
- Combine : $C(n) = \Theta(n)$





# 3 methods for solving recurrence equations

1. Substitution method : Only if we have an idea of the solution.
   We replace one of the terms in the equation by the solution presented.

$$\begin{array}{lll} T(\frac{n}{2}) & \leq & c(\frac{n}{2})log_2(\frac{n}{2}) \\ T(n) & \leq & 2T(\frac{n}{2}) + C(n) \quad = \quad 2(c(\frac{n}{2})log_2(\frac{n}{2})) + kn \\ & \leq & cnlog_2(\frac{n}{2}) + kn \\ & \leq & cnlog_2(n) - cnlog_2(2) + kn \qquad \text{on a :} \quad log_2(2) = 1 \\ & \leq & cnlog_2(n) - cn + kn \\ & \leq & cnlog_2(n) + n(k-c) \end{array} \qquad (1)$$

We find the solution for $n$ (only if $c > k$).
We must also check that this property is also valid at the limits, i.e. that we can can choose $c$ such that $T(n) \leq cnlog_2(n)$ also holds at the limit.
There may be a few problems. For $n = 1$ we have :

$$\begin{cases} T(1) = 1 \\ T(1) \leq c \times 1 \times log_2(1) = 0 \end{cases}$$

The property must therefore be verified for $n \geq n_0$. From the recurrence, we have $T(2) = 5$ and $T(3) = 9$ and we must choose $c$ such that, :

$$\begin{cases} 5 = T(2) \leq c \times 2 \times log_n(2) \\ 9 = T(3) \leq c \times 3 \times \underbrace{log_2(3)}_{=1.58} \end{cases}$$

$c \geq 2$ is a sufficient condition.





# 3 methods for solving recurrence equations

2. Iterative method : we iterate the recurrence until we obtain the solution.
   To simplify : $n$ is assumed to be a power of 2.

$$\begin{array}{lll} T(n) & = & 2T(n/2) + \underbrace{n}_{fusion} + \underbrace{1}_{diviser} \\ & = & 2\left(2T(n/4) + \underbrace{n/2} + \underbrace{1}\right) + \underbrace{n} + \underbrace{1} \\ & = & 4T(n/4) + \underbrace{2/2n + n} + \underbrace{1+2} \\ & = & 2\left(4T(n/8) + \underbrace{n/4} + \underbrace{1}\right) + \underbrace{2/2n + n} + \underbrace{1+2} \\ & = & 8T(n/8) + \underbrace{4/4n + 2/2n + n} + \underbrace{1+2+4} \end{array} \qquad (1)$$

Iteration leads to T(1) when $n/2^i = 1$, i.e. when $i \geqslant log_2(n)$.

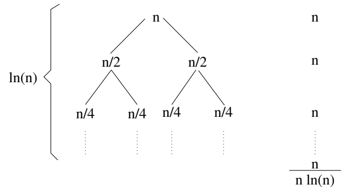
$$\begin{array}{lll} T(n) & = & 2^i T(1) + \underbrace{n + 2/2n + 4/4n + ... + 2^i/2^i n} + \underbrace{\sum_{i=0}^{k=log_2(n)-1} 2^i} \\ & = & nT(1) + \underbrace{nlog_2(n)} + \underbrace{2^{log_2(n)-1+1} - 1} \\ & = & nT(1) + \underbrace{nlog_2(n)} + \underbrace{n-1} \\ & = & O(n.log_2(n)) \end{array} \qquad (2)$$





# 3 methods for solving recurrence equations

2. Iterative method :




**A reminder of a remarkable identity :**

- $(A^n - B^n) = (A - B)(A^{n-1} + A^{n-2}B + A^{n-3}B^2 + \ldots + AB^{n-2} + B^{n-1})$
- $A^n - 1 = (A-1)(A^{n-1} + A^{n-2} + A^{n-3} + \ldots + A + 1)$
- $2^n - 1 = (2-1)(2^{n-1} + 2^{n-2} + 2^{n-3} + \ldots + 2 + 1)$

**A reminder of logarithms**

- $\log_a(A)$ **is the number** $x$ **such that** $a^x = A$
- **log neperian, :** $\ln = \log_e$ **where** $e = 2.718...$
- **log decimal :** $\mathrm{Log}(x) = \log_{10}(x) = \ln(x)/\ln(10)$
- **Properties :**

$\ln(ab) = \ln(a) + \ln(b) \qquad \ln(a/b) = \ln(a) - \ln(b)$
$\ln(a^b) = b\ln(a) \qquad \log_b(a) = \ln(a)/\ln(b)$ **because** $b^x = e^{x\ln(b)}$

**In fact, I'm looking for $x$ such that $b^x = a$, i.e. such that $e^{xln(b)} = a$. Taking the logarithm, we have $xln(b) = ln(a)$.**

## 3 methods for solving recurrence equations

3. General method :

**Theorem**

*Let $a \geq 1$, $b > 1$, $f(n)$ be a positive function and let $T(n)$ be defined by the recurrence :*

$$T(n) = aT(n/b) + f(n)$$

*Then $T(n)$ can be asymptotically bounded as follows :*

1. *if $f(n) = O(n^{\log_b a - \epsilon})$ for a constant $\epsilon > 0$ then*
$$T(n) = \Theta(n^{\log_b a})$$
2. *if $f(n) = \Theta(n^{\log_b a})$ then*
$$T(n) = \Theta(n^{\log_b a} ln(n))$$
3. *if $f(n) = \Omega(n^{\log_b a + \epsilon})$ for a constant $\epsilon > 0$ and if $af(n/b) \leq cf(n)$ for $c < 1$ and for any $n$ large enough, then*
$$T(n) = \Theta(f(n))$$

Please note that some possible situations are not covered.



Algorithmics for Biology
Jean-Paul Comet
Complexity
Generality
Complexity analysis
Notations
Divide and Conquer
Pat. Matching
Graphs
Dyn.Prog.
Sequences

## 3 methods for solving recurrence equations

3. General method : **Example of using the theorem**
   1. $T(n) = 9T(n/3) + n$
   2. $T(n) = T(\frac{2}{3}n) + 1$
   3. $T(n) = 3T(\frac{n}{4}) + n.ln(n)$
   4. $T(n) = 2T(n/2) + n.ln(n)$



Algorithmics for Biology
Jean-Paul Comet
Complexity
Generality
Complexity analysis
Notations
Divide and Conquer
Pat. Matching
Graphs
Dyn.Prog.
Sequences

## An atypical sorting algorithm

A sorting algorithm **NOT based** on the comparison of its elements :

- Assumption : the array to be sorted is composed only of integers $\in [0, 63]$.

1. An array of size 64 is created (initialized to 0).
2. We browse the initial array, and when we find the value $k$, we update the array $C$ : `C[k]++;`.
3. The sorted array is then reconstructed.

Complexity : $O(n)$.