Programmation avancée

Pierre Geurts

E-mail : p.geurts@ulg.ac.be
URL : http://www.montefiore.ulg.ac.be/~geurts/pa.html
Bureau : I 141 (Montefiore)
Téléphone : 04.366.48.15 — 04.366.99.64

Contact

- Chargé de cours :
  - Pierre Geurts, p.geurts@ulg.ac.be, I141 Montefiore, 04/3664815
- Assistant :
  - Jean-Michel Begon, jm.begon@student.ulg.ac.be, GIGA-R (B34,+1), CHU, 04/3662766
- Sites web du cours :
  - http://www.montefiore.ulg.ac.be/~geurts/pa.html

Objectifs du cours

Introduction à l'étude systématique des algorithmes et des structures de données

Deux objectifs :

- Vous fournir une boîte à outils contenant :
  - Des structures de données permettant d'organiser et d'accéder efficacement aux données
  - Les algorithmes les plus populaires
  - Des méthodes génériques pour la modélisation, l'analyse et la résolution de problèmes algorithmiques
- Vous former à la résolution de problèmes algorithmiques nouveaux en vous basant sur cette boîte à outils

On insistera sur la généralité des algorithmes et structures de données et on les étudiera de manière formelle

Les projets visent à vous familiariser à la résolution de problèmes

Dans votre cursus

Pré-requis :

- Introduction à la programmation (INFO0946) : premier pas avec le c et l'algorithmique
- Compléments de programmation (INFO0947) : construction formelle de programmes, récursivité, types de données abstraits, et premières structures avancées

Co-requis :

- Théorie des graphes (MATH0499) : algorithmique sur les graphes

Après cela :

- Techniques de programmation (INFO0027) : approfondissement sur les techniques de résolution de problèmes et les structures de données, programmation parallèle.
- Programmation orienté-objet (INFO0067)
- Programmation fonctionnelle (INFO0054)
- ...

Organisation du cours

- Cours théoriques :
  - Les vendredis de 13h30 à 15h30, 2.93, Bâtiment B28 (Institut Montefiore).
  - 10-12 cours
- Répétitions :
  - Certains vendredis de 15h30 à 17h30, 2.93, Bâtiment B28 (Institut Montefiore).
  - Exercices portant sur la matière théorique + debriefing des projets
- Projets (sujet à modifications) :
  - Trois projets tout au long de l'année, de difficulté croissante
  - Les deux premiers individuels, le troisième en binôme
  - En C

Evaluation

- Première session :
  - Projets : 30%
  - Examen écrit : 70%
  - La réalisation des projets est obligatoire pour pouvoir accéder à l'examen écrit.

- Deuxième session :
  - La cote des projets peut être reportée en seconde session (pour 10% ou 30%, au choix)
  - Un projet de rattrapage est proposé. Il est obligatoire si les projets n'ont pas été faits pendant l'année, facultatif sinon. S'il est réalisé, il intervient pour 10% de la cote finale.
  - Examen écrit : 90% ou 70%, en fonction des projets.

Notes de cours

- Transparents disponibles sur la page web du cours.
- Pas de livre de référence obligatoire mais le cours se base fortement sur l'ouvrage suivant :
  - **Introduction to algorithms, Cormen, Leiserson, Rivest, Stein, MIT press, Third edition, 2009.**
    - `http://mitpress.mit.edu/algorithms/`
- Autres références :
  - Algorithms, Sedgewick and Wayne, Addison Wesley, Fourth edition, 2011.
    - `http://algs4.cs.princeton.edu/home/`
  - Data structures and algorithms in Java, Goodrich and Tamassia, Fifth edition, 2010.
    - `http://ww0.java4.datastructures.net/`
  - Algorithms, Dasgupta, Papadimitriou, and Vazirani, McGraw-Hill, 2006.
    - `http://cseweb.ucsd.edu/users/dasgupta/book/index.html`
    - `http://www.cs.berkeley.edu/~vazirani/algorithms/all.pdf`

Cours sur le web

Ce cours s'inspire également de plusieurs cours disponibles sur le web :

- Antonio Carzaniga, Faculty of Informatics, University of Lugano
  - `http://www.inf.usi.ch/carzaniga/edu/algo/index.html`
- Marc Gaetano, Polytechnique, Nice-Sophia Antipolis
  - `http://users.polytech.unice.fr/~gaetano/asd/`
- Robert Sedgewick, Princeton University
  - `http://www.cs.princeton.edu/courses/archive/spr10/cos226/lectures.html`
- Charles Leiserson and Erik Demaine, MIT.
  - `http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/index.htm`
- Le cours de 2009-2010 de Bernard Boigelot

Contenu du cours

Partie 1: Introduction
Partie 2: Outils d'analyse
Partie 3: Algorithmes de tri
Partie 4: Structures de données élémentaires
Partie 5: Dictionnaires
Partie 6: Résolution de problèmes

Partie 1

Introduction

Plan

1. Algorithms + Data structures = Programs (Niklaus Wirth)

2. Rappel sur la récursivité

Algorithmes

- Un **algorithme** est une suite *finie* et *non-ambiguë* d'opérations ou d'instructions permettant de résoudre un *problème*
- Provient du nom du mathématicien persan *Al-Khawarizmi* ($\pm$820), le père de l'algèbre
- Un problème algorithmique est souvent formulé comme la transformation d'un ensemble de valeurs, **d'entrée**, en un nouvel ensemble de valeurs, **de sortie**.
- Exemples d'algorithmes :
  - Une recette de cuisine (ingrédients $\rightarrow$ plat préparé)
  - La recherche dans un dictionnaire (mot $\rightarrow$ définition)
  - La division entière (deux entiers $\rightarrow$ leur quotient)
  - Le tri d'une séquence (séquence $\rightarrow$ séquence ordonnée)

Algorithmes

- On étudiera essentiellement les algorithmes **corrects**.
  - Un algorithme est (totalement) *correct* lorsque pour chaque instance, il se termine en produisant la bonne sortie.
  - Il existe également des algorithmes *partiellement corrects* dont la terminaison n'est pas assurée mais qui fournissent la bonne sortie lorsqu'ils se terminent.
  - Il existe également des algorithmes *approximatifs* qui fournissent une sortie inexacte mais néanmoins proche de l'optimum.

- Les algorithmes seront évalués en termes d'*utilisation de ressources*, essentiellement par rapport aux **temps de calcul** mais aussi à l'utilisation de la **mémoire**.

Algorithmes

Un algorithme peut être spécifié de différentes manières :

- en langage naturel,
- graphiquement,
- en pseudo-code,
- par un programme écrit dans un langage informatique
- ...

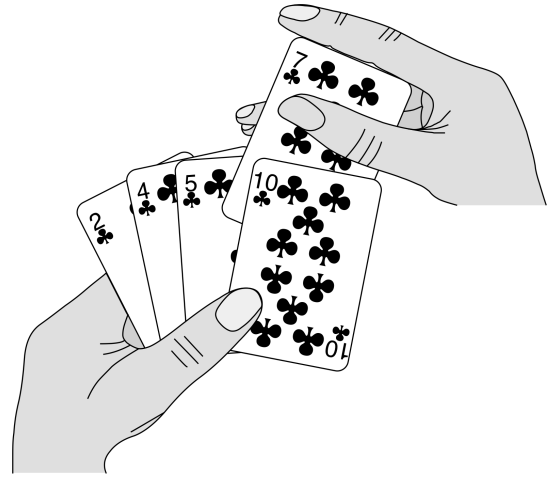
La seule condition est que la description soit précise.

Exemple : le tri

- Le problème de tri :
  - Entrée : une séquence de n nombres $\langle a_1, a_2, \ldots, a_n \rangle$
  - Sortie : une permutation de la séquence de départ $\langle a'_1, a'_2, \ldots, a'_n \rangle$ telle que $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

- Exemple :
  - Entrée : $\langle 31, 41, 59, 26, 41, 58 \rangle$
  - Sortie : $\langle 26, 31, 41, 41, 58, 59 \rangle$

Tri par insertion



Description en langage naturel :

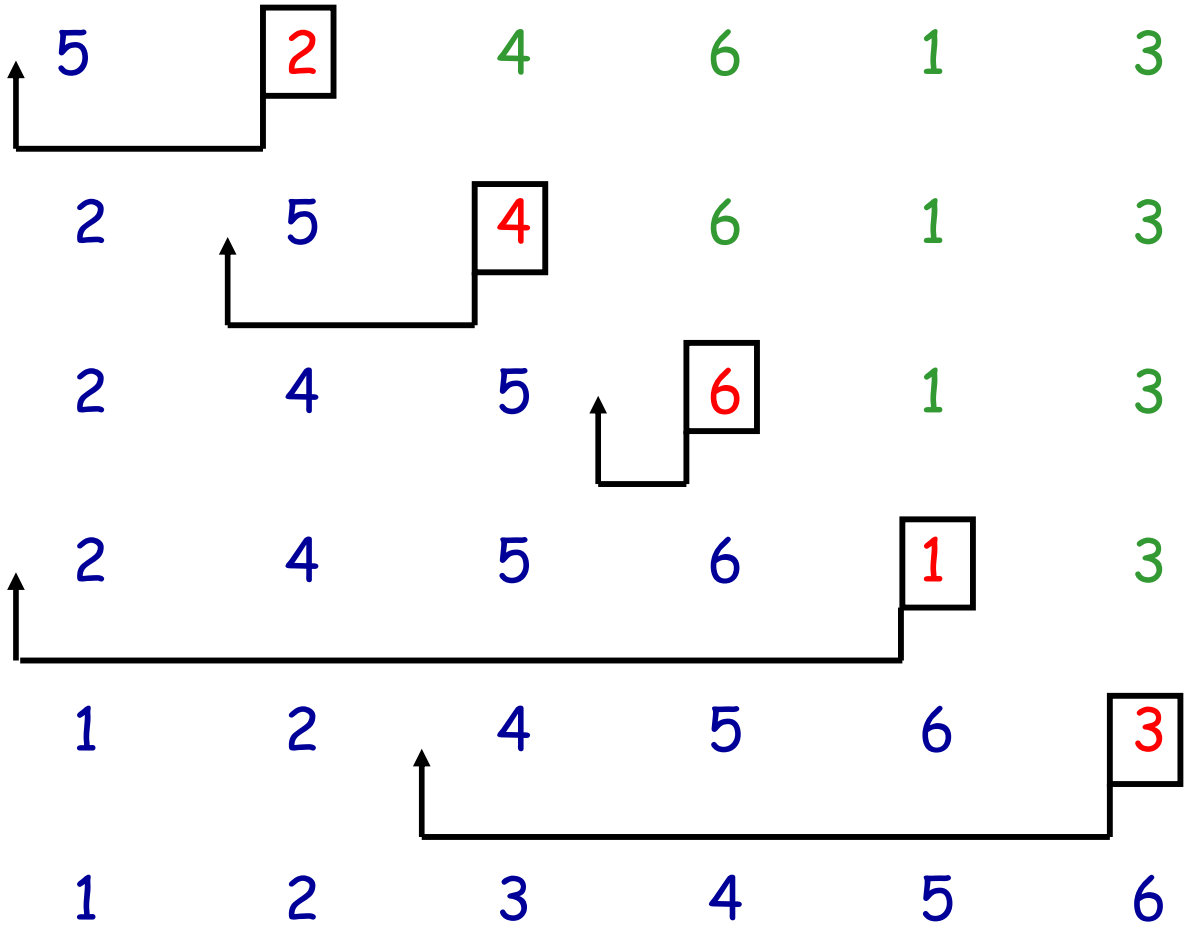
On parcourt la séquence de gauche à droite

Pour chaque élément a_j :

- On l'insère à sa position dans une nouvelle séquence ordonnée contenant les éléments le précédant dans la séquence.

On s'arrête dès que le dernier élément a été inséré à sa place dans la séquence.

Tri par insertion



Tri par insertion

Description en C (sur des tableaux d'entiers) :

```
void InsertionSort (int *a, int length) {
  int key, i;
  for(int j = 1; j < length; j++) {
    key = a[j];
    /* Insert a[j] into 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;
  }
}
```

Insertion sort

Description en **pseudo-code** (sur des tableaux d'entiers) :

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1..j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Pseudo-code

Objectifs :

- Décrire les algorithmes de manière à ce qu'ils soient compris par des humains.
- Rendre la description indépendante de l'implémentation
- S'affranchir de détails tels que la gestion d'erreurs, les déclarations de type, etc.

Très proche du C (langage procédural plutôt qu'orienté objet)

Peut contenir certaines instructions en langage naturel si nécessaire

Pseudo-code

Quelques règles

- Structures de blocs indiquées par l'indentation
- Boucles (**for**, **while**, **repeat**) et conditions (**if**, **else**, **elseif**) comme en C.
- Le compteur de boucle garde sa valeur à la sortie de la boucle
- En sortie d'un **for**, le compteur a la valeur de la borne max+1.

$$\textbf{for } i = 1 \textbf{ to } Max \quad Code \quad \Leftrightarrow \quad i = 1 \quad \textbf{while } i \leq Max \quad Code \quad i = i + 1$$

- Commentaires indiqués par //
- Affectation (=) et test d'égalité (==) comme en C.

Pseudo-code

- Les variables (i, j et *key* par exemple) sont locales à la fonction.
- $A[i]$ désigne l'élément i du tableau A. $A[i..j]$ désigne un intervalle de valeurs dans un tableau. $A.length$ est la taille du tableau.
- L'indexation des tableaux commence à 1.
- Les types de données composés sont organisés en *objets*, qui sont composés d'attributs. On accède à la valeur de l'attribut *attr* pour un objet x par $x.attr$.
- Un variable représentant un tableau ou un objet est considérée comme un pointeur vers ce tableau ou cet objet.
- Paramètres passés par valeur comme en C (mais tableaux et objets sont passés par pointeur).
- ...

Trois questions récurrentes face à un algorithme

1. Mon algorithme est-il correct, se termine-t-il ?
2. Quelle est sa vitesse d'exécution ?
3. Y-a-t'il moyen de faire mieux ?

Exemple du tri par insertion

1. Oui $\rightarrow$ technique des invariants (INFO0947 et partie 2)
2. $O(n^2) \rightarrow$ analyse de complexité (partie 2)
3. Oui $\rightarrow$ il existe un algorithme $O(n \log n)$ (partie 1)

Correction de Insertion-Sort

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      i = j − 1
4      while i > 0 and A[i] > key
5          A[i + 1] = A[i]
6          i = i − 1
7      A[i + 1] = key
```

- Invariant : (pour la boucle externe) le sous-tableau $A[1\,..\,j-1]$ contient les éléments du tableau original $A[1\,..\,j-1]$ ordonnés.
- On doit montrer que
  - l'invariant est vrai avant la première itération
  - l'invariant est vrai avant chaque itération suivante
  - En sortie de boucle, l'invariant implique que l'algorithme est correct

Correction de Insertion-Sort

- Avant la première itération :
  - $j = 2 \Rightarrow A[1]$ est trivialement ordonné.
- Avant la $j$ème itération :
  - Informellement, la boucle interne déplace $A[j-1]$, $A[j-2]$, $A[j-3] \ldots$ d'une position vers la droite jusqu'à la bonne position pour *key* ($A[j]$).
- En sortie de boucle :
  - A la sortie de boucle, $j = A.length + 1$. L'invariant implique que $A[1 \ldots A.length]$ est ordonné.

Complexité de Insertion-Sort

```
INSERTION-SORT(A)
1   for j = 2 to A.length
2       key = A[j]
3       i = j − 1
4       while i > 0 and A[i] > key
5           A[i + 1] = A[i]
6           i = i − 1
7       A[i + 1] = key
```

- Nombre de comparaisons $T(n)$ pour trier un tableau de taille n ?
- Dans le pire des cas :
  - La boucle **for** est exécutée $n - 1$ fois ($n = A.length$).
  - La boucle **while** est exécutée $j - 1$ fois

Complexité de Insertion-Sort

- Le nombre de comparaisons est borné par :

$$T(n) \leq \sum_{j=2}^{n} (j-1)$$

- Puisque $\sum_{i=1}^{n} i = n(n+1)/2$, on a :

$$T(n) \leq \frac{n(n-1)}{2}$$

- Finalement, $T(n) = O(n^2)$

(borne inférieure ?)

Structures de données

- Méthode pour stocker et organiser les données pour en faciliter l'accès et la modification
- Une structure de données regroupe :
  - un certain nombre de données à gérer, et
  - un ensemble d'opérations pouvant être appliquées à ces données
- Dans la plupart des cas, il existe
  - plusieurs manières de représenter les données et
  - différents algorithmes de manipulation.
- On distingue généralement l'interface des structures de leur implémentation.

Types de données abstraits

- Un type de données abstrait (TDA) représente l'interface d'une structure de données.
- Un TDA spécifie précisément :
  - la nature et les propriétés des données
  - les modalités d'utilisation des opérations pouvant être effectuées
- En général, un TDA admet différentes implémentations (plusieurs représentations possibles des données, plusieurs algorithmes pour les opérations).

Exemple : file à priorités

- Données gérées : des objets avec comme attributs :
  - une clé, munie d'un opérateur de comparaison selon un ordre total
  - une valeur quelconque

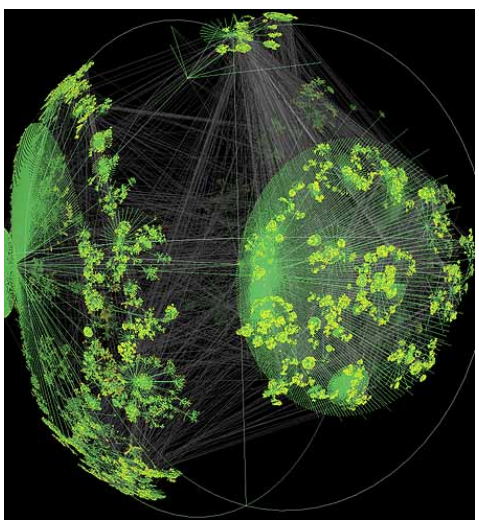
- Opérations :
  - Création d'une file vide
  - $\textsc{Insert}(S, x)$: insère l'élément x dans la file S.
  - $\textsc{Extract-Max}(S)$: retire et renvoie l'élément de S avec la clé la plus grande.

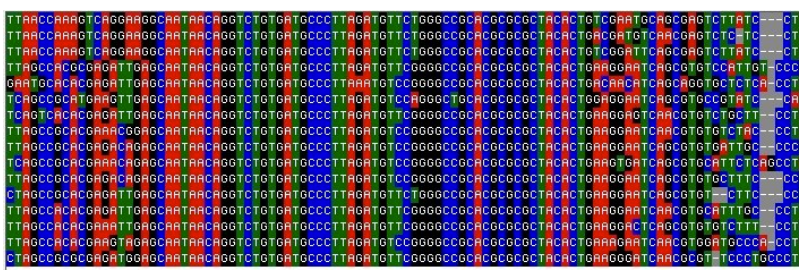
- Il existe de nombreuses façons d'implémenter ce TDA :
  - Tableau non trié ;
  - Liste triée ;
  - Structure de tas ;
  - ...

  Chacune mène à des complexités différentes des opérations $\textsc{Insert}$ et $\textsc{Extract-Max}$

Structures de données et algorithmes en pratique

- La résolution de problème algorithmiques requiert presque toujours la combinaison de structures de données et d'algorithmes sophistiqués pour la gestion et la recherche dans ces structures.
- D'autant plus vrai qu'on a à traiter des volumes de données importants.
- Quelques exemples de problèmes réels :
  - Routage dans les réseaux informatiques
  - Moteurs de recherche
  - Alignement de séquences ADN en bio-informatique





Un exemple (B. Boigelot)

- Un laboratoire de génie génétique désire développer un programme capable de repérer des répétitions de longueur M dans une séquence de nucléotides S de longueur N (avec $N >> M$) :

$$ACTG\underline{CGAC}GGTACGCTT\underline{CGAC}TTAG...(M = 4)$$

- Première approche :
  - Un indice i variant de 2 à $N - M + 1$
  - Un indice j variant de 1 à $i - 1$
  - Pour tout $k \in [0, \ldots, M - 1]$, on teste si $S[i + k] = S[j + k]$.
- Efficacité : le nombre de comparaisons à effectuer est égal à :

$$\begin{aligned} M \cdot (1 + 2 + \ldots + (N - M)) &= \frac{M(N - M + 1)(N - M)}{2} \\ &\approx 4,510^{21} \text{ pour } N = 3 \cdot 10^9 \text{ et } M = 1000 \\ &\approx 143.000 \text{ ans au rythme de } 10^9 \text{ opérations/s.} \end{aligned}$$

Une meilleure solution

1. On construit une table à $N - M + 1$ lignes et M colonnes dont la k-ème ligne contient la sous-séquence de longueur M commençant à la position k dans S :

   ACTG
   CTGC
   TGCG
   GCGA
   CGAC
   ⋮

2. On trie les lignes de cette table par ordre lexicographique ;
3. On parcourt la table triée afin de déterminer si elle contient deux lignes consécutives identiques

Note : Lors de la comparaison de deux lignes, on s'arrête à la première différence ($\Rightarrow$ moins de 4/3 comparaisons en moyenne).

Efficacité

- Construction de la table : $M(N - M + 1)$ opérations de copie.
- Tri par ordre lexicographique (algorithme de tri rapide, voir partie 3) :

$$\leq \frac{8}{3} N \ln N \text{ opérations de comparaison en moyenne}$$

- Détection de lignes consécutives :

$$\leq \frac{4}{3}(N - M) \text{ opérations de comparaison en moyenne}$$

En supposant des coûts identiques pour toutes les opérations, on obtient :

$$\begin{aligned} & N(M + \frac{8}{3} \ln N + \frac{4}{3}) - M(M + \frac{1}{3}) \\ \approx \quad & 3,179.10^{12} \text{ opérations pour } N = 3.10^9 \text{ et } M = 1000. \\ \approx \quad & 53 \text{ minutes au rythme de } 10^9 \text{ opérations/s.} \end{aligned}$$

Remarques

- Utiliser un ordinateur plus puissant ne permet généralement pas de résoudre les problèmes d'efficacité !
  Avec un ordinateur 1000 fois plus puissant : 143 ans pour la première approche, 3,2s pour la deuxième.
- La deuxième solution est plus rapide mais elle est très gourmande en espace mémoire (M fois plus que la première !)

- Exercice pour plus tard : proposez une solution plus efficace encore en utilisant les structures de données vues au cours.

Plan

1. Algorithms + Data structures = Programs (Niklaus Wirth)

2. Rappel sur la récursivité

Algorithmes récursifs

Un algorithme est **récursif** s'il s'invoque lui-même directement ou indirectement.

Motivation : Simplicité d'expression de certains algorithmes

Exemple : Fonction factorielle :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

```
FACTORIAL(n)
1  if n == 0
2        return 1
3  return n · FACTORIAL(n − 1)
```

Algorithmes récursifs

```
FACTORIAL(n)
1  if n == 0
2        return 1
3  return n · FACTORIAL(n − 1)
```

Règles pour développer une solution récursive :

- On doit définir un cas de base ($n == 0$)
- On doit diminuer la "taille" du problème à chaque étape ($n \to n-1$)
- Quand les appels récursifs se partagent la même structure de données, les sous-problèmes ne doivent pas se superposer (pour éviter les effets de bord)

Exemple de récursion multiple

Calcul du $n$ième nombre de Fibonacci :

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ \forall n \geq 2 : F_n &= F_{n-2} + F_{n-1} \end{aligned}$$

Algorithme :

```
FIBONACCI(n)
1  if n ≤ 1
2        return n
3  return FIBONACCI(n − 2) + FIBONACCI(n − 1)
```

Exemple de récursion multiple

```
FIBONACCI(n)
1  if n ≤ 1
2        return n
3  return FIBONACCI(n − 2) + FIBONACCI(n − 1)
```

1. L'algorithme est-il correct ?
2. Quelle est sa vitesse d'exécution ?
3. Y-a-t'il moyen de faire mieux ?

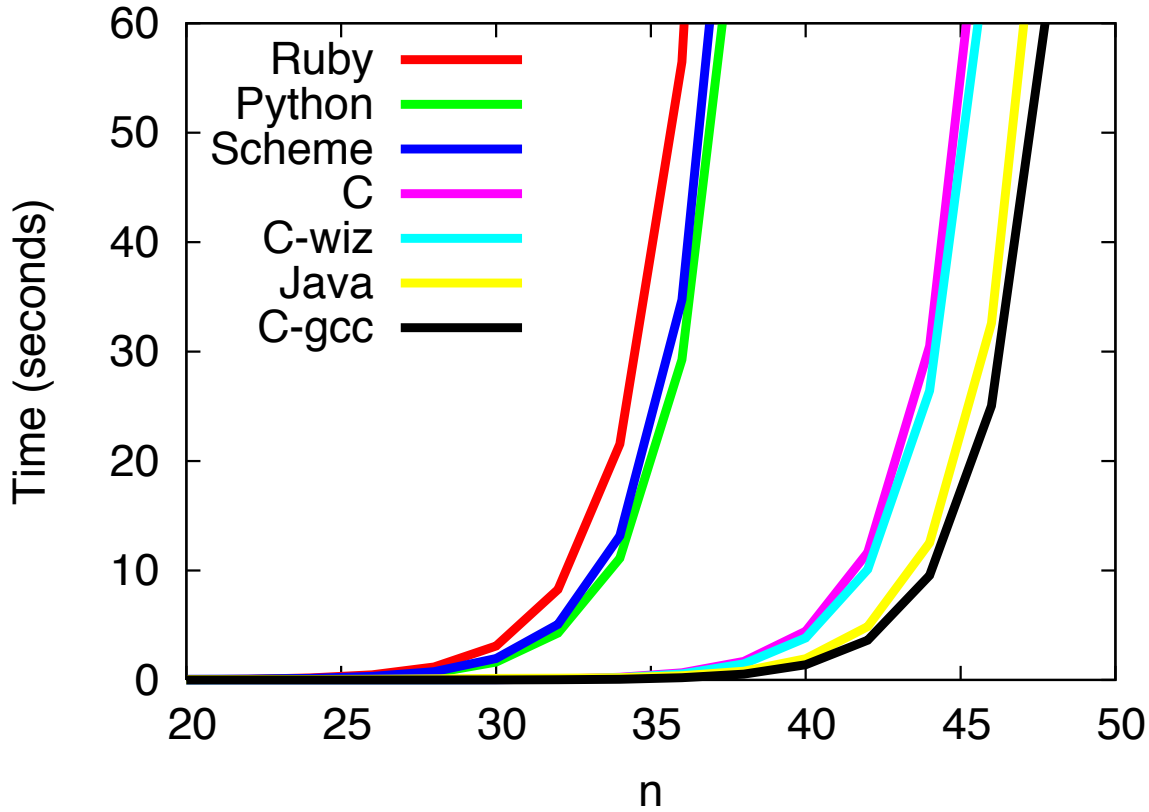
Exemple de récursion multiple

```
FIBONACCI(n)
1  if n ≤ 1
2        return n
3  return FIBONACCI(n − 2) + FIBONACCI(n − 1)
```

1. L'algorithme est correct ?
   - Clairement, l'algorithme est correct.
   - En général, la correction d'un algorithme récursif se démontre par induction.
2. Quelle est sa vitesse d'exécution ?
3. Y-a-t'il moyen de faire mieux ?

Vitesse d'exécution

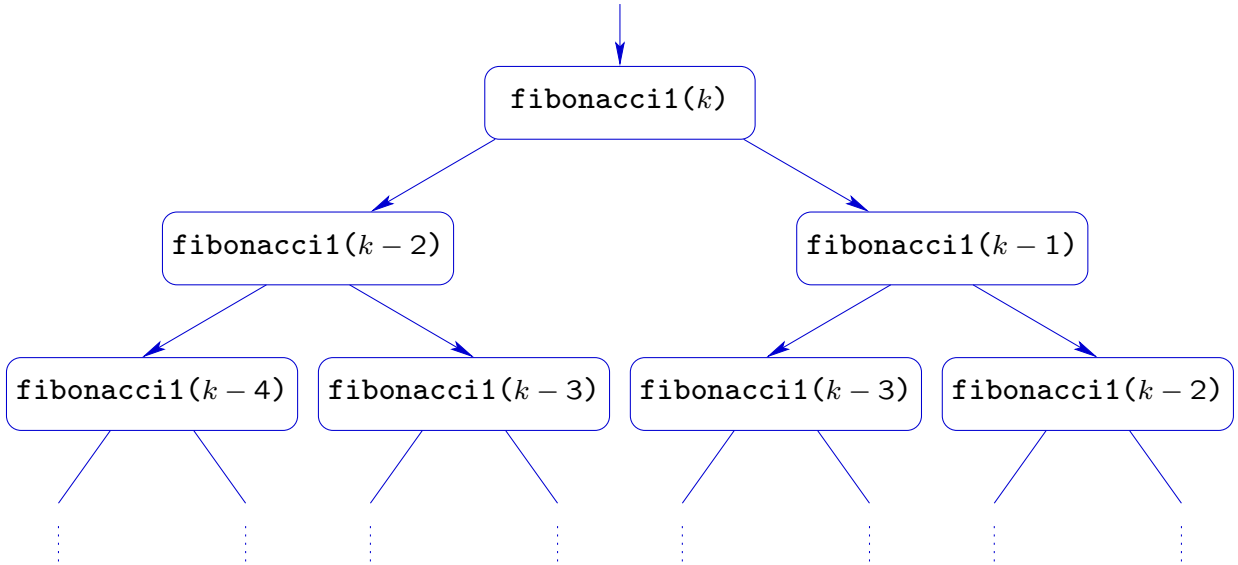
- Nombre d'opérations pour calculer $\textsc{Fibonacci}(n)$ en fonction de n
- Empiriquement :



(Carzaniga)

- Toutes les implémentations atteignent leur limite, plus ou moins loin

Trace d'exécution



(Boigelot)

Complexité

```
FIBONACCI(n)
1  if n ≤ 1
2        return n
3  return FIBONACCI(n − 2) + FIBONACCI(n − 1)
```

- Soit $T(n)$ le nombre d'opérations de base pour calculer FIBONACCI(n) :

$$\begin{aligned} T(0) &= 2, T(1) = 2 \\ T(n) &= T(n-1) + T(n-2) + 2 \end{aligned}$$

- On a donc $T(n) \geq F_n$ ($=$ le $n$ème nombre de Fibonacci).

Complexité

- Comment croît F_n avec n ?

$$T(n) \geq F_n = F_{n-1} + F_{n-2}$$

Puisque $F_n \geq F_{n-1} \geq F_{n-2} \geq \ldots$, on a :

$$F_n \geq 2F_{n-2} \geq 2(2F_{n-4}) \geq 2(2(2F_{n-6})) \geq 2^{\frac{n}{2}-1}F_2 = \frac{(\sqrt{2})^n}{2}$$

si $n \geq 2$ est pair et

$$F_n \geq 2F_{n-2} \geq 2(2F_{n-4}) \geq 2(2(2F_{n-6})) \geq 2^{\frac{n-1}{2}}F_1 = \frac{(\sqrt{2})^n}{\sqrt{2}} \geq \frac{(\sqrt{2})^n}{2}$$

si $n \geq 1$ est impair.

Et donc

$$T(n) \geq \frac{(\sqrt{2})^n}{2} \approx \frac{(1.4)^n}{2}$$

- $T(n)$ croît exponentiellement avec n

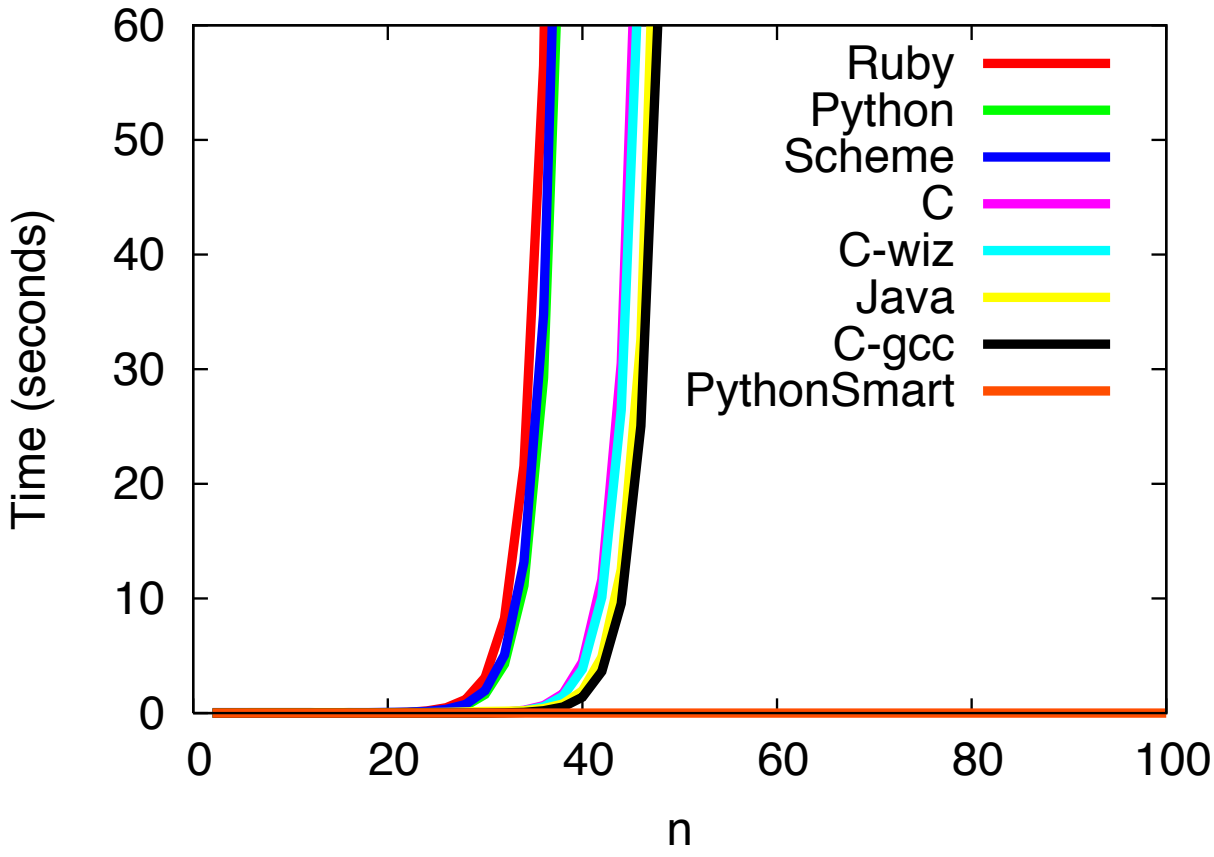
- Peut-on faire mieux ?

Solution itérative

```
FIBONACCI-ITER(n)
 1  if n ≤ 1
 2        return n
 3  else
 4        pprev = 0
 5        prev = 1
 6        for i = 2 to n
 7              f = prev + pprev
 8              pprev = prev
 9              prev = f
10        return f
```

Vitesse d'exécution

Complexité : $O(n)$



(Carzaniga)

Tri par fusion

Idée d'un tri basé sur la récursion :

- on sépare le tableau en deux sous-tableaux de la même taille
- on trie (récursivement) chacun des sous-tableaux
- on fusionne les deux sous-tableaux triés en maintenant l'ordre

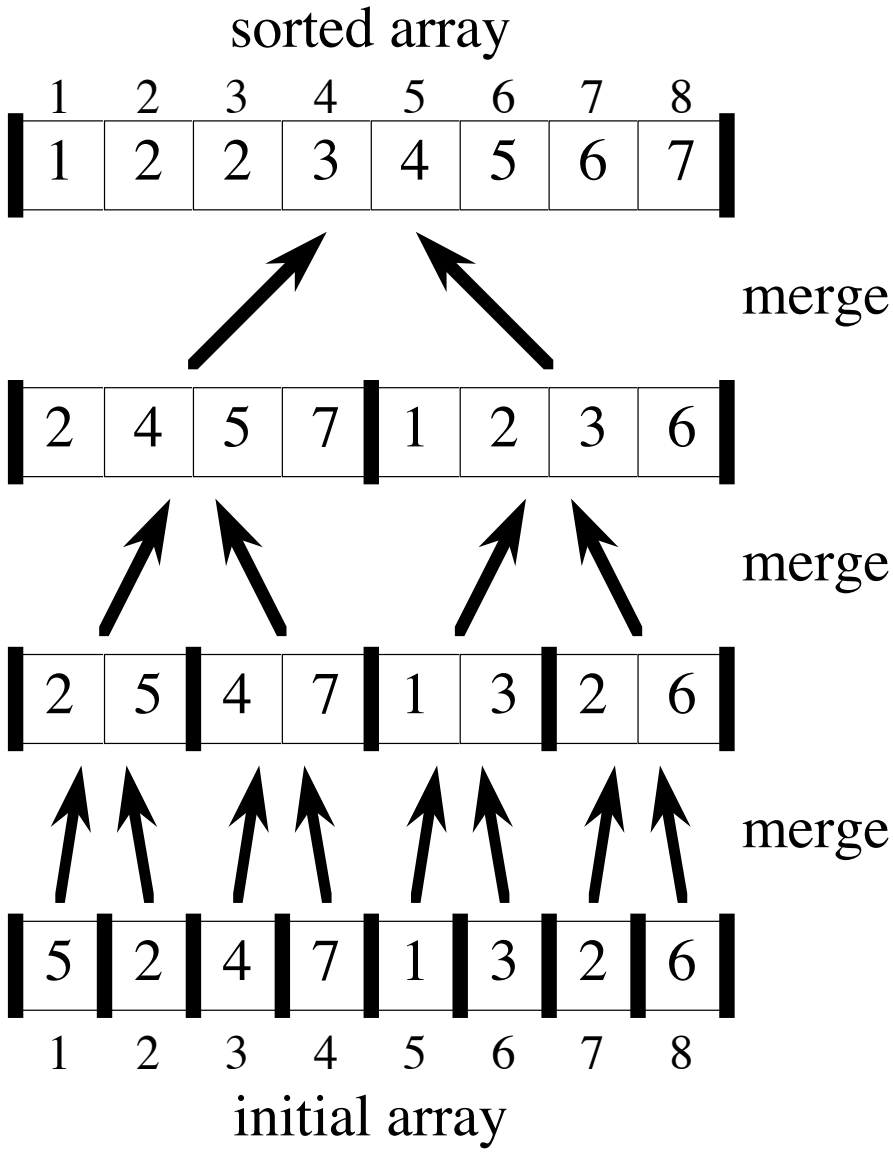
Le cas de base correspond à un tableau d'un seul élément.

```
MERGE-SORT(A, p, r)
1  if p < r
2        q = ⌊(p+r)/2⌋
3        MERGE-SORT(A, p, q)
4        MERGE-SORT(A, q + 1, r)
5        MERGE(A, p, q, r)
```

Appel initial : MERGE-SORT$(A, 1, A.length)$

Exemple d'application du principe général de "diviser pour régner"

Tri par fusion : illustration



Fonction MERGE

MERGE(A, p, q, r) :

- **Entrée :** tableau A et indice p, q, r tels que :
  - $p \leq q < r$ (pas de tableaux vides)
  - Les sous-tableaux $A[p\,..\,q]$ et $A[q+1\,..\,r]$ sont ordonnés
- **Sortie :** Les deux sous-tableaux sont fusionnés en un seul sous-tableau ordonné dans $A[p\,..\,r]$

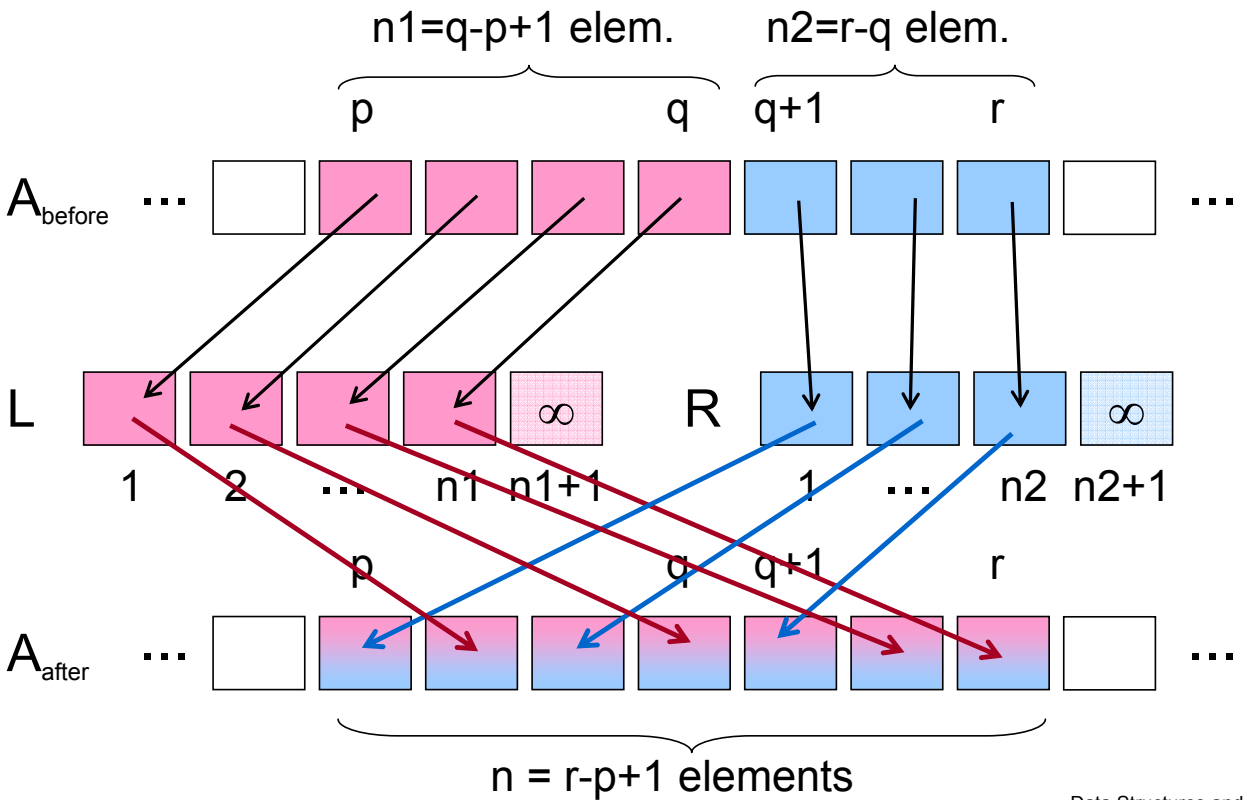
Idée :

- Utiliser un pointeur vers le début de chacun des sous-tableaux ;
- Déterminer le plus petit des deux éléments pointés ;
- Déplacer cet élément vers le tableau fusionné ;
- Avancer le pointeur correspondant

Fusion : algorithme

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1 ; n2 = r − q
 2  Soit L[1..n1 + 1] et R[1..n2 + 1] deux nouveaux tableaux
 3  for i = 1 to n1
 4        L[i] = A[p + i − 1]
 5  for j = 1 to n2
 6        R[j] = A[q + j]
 7  L[n1 + 1] = ∞ ; R[n2 + 1] = ∞                    // Sentinels
 8  i=1 ;j=1
 9  for k = p to r
10        if L[i] ≤ R[j]
11              A[k] = L[i]
12              i = i + 1
13        else
14              A[k] = R[j]
15              j = j + 1
```

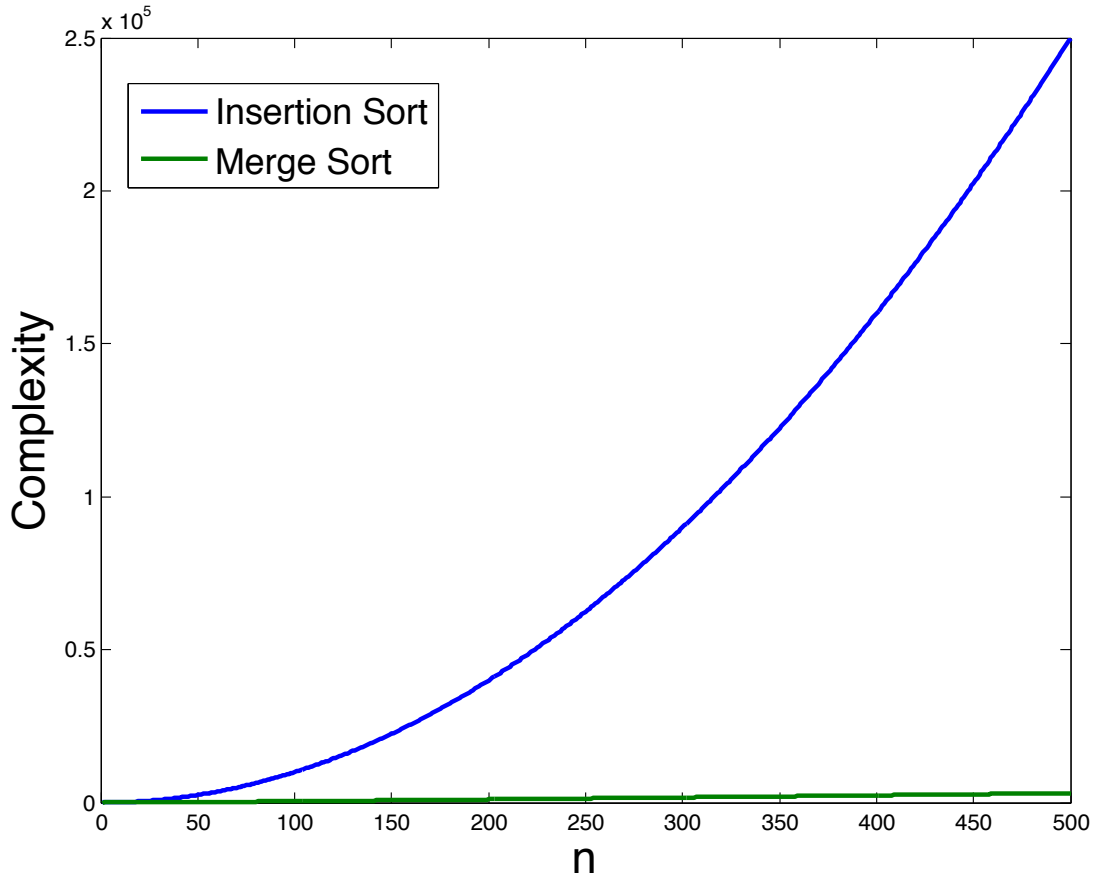
Fusion : illustration



Complexité : $O(n)$ (où $n = r - p + 1$)

Vitesse d'exécution

Complexité de MERGE-SORT : $O(n \log n)$ (voir partie 2)



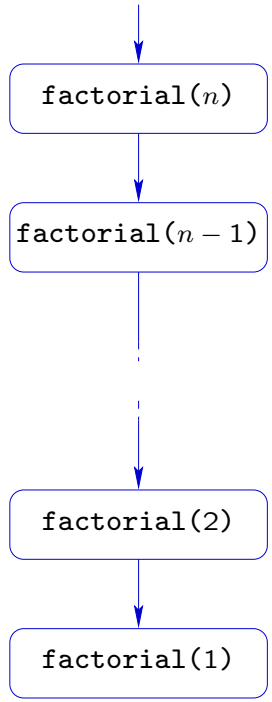
Remarques

- La fonction MERGE nécessite d'allouer deux tableaux L et R (dont la taille est $O(n)$). Exercice (difficile) : écrire une fonction MERGE qui ne nécessite pas d'allocation supplémentaire.
- On pourrait réécrire MERGE-SORT de manière itérative (au prix de la simplicité)
- Version récursive du tri par insertion :

```
INSERTION-SORT-REC(A, n)
1  if n > 1
2        INSERTION-SORT-REC(A, n − 1)
3        MERGE(A, 1, n − 1, n)
```

Note sur l'implémentation de la récursivité

- Trace d'exécution de la factorielle



- Chaque appel récursif nécessite de mémoriser le contexte d'invocation
- L'espace mémoire utilisé est donc $O(n)$ (n appels récursifs)

Récursivité terminale

- Définition : Une procédure est récursive terminale (“tail recursive”) si elle n’effectue plus aucune opération après s’être invoquée récursivement.

- Avantages :
  - Le contexte d’invocation ne doit pas être mémorisé et donc l’espace mémoire nécessaire est réduit
  - Les procédures récursives terminales peuvent facilement être converties en procédures itératives

Version récursive terminale de la factorielle

```
FACTORIAL2(n)
1  return FACTORIAL2-REC(n, 2, 1)
```

```
FACTORIAL2-REC(n, i, f)
1  if i > n
2        return f
3  return FACTORIAL2-REC(n, i + 1, f · i)
```

Espace mémoire utilisé : $O(1)$ (si la récursion terminale est implémentée efficacement)

Ce qu'on a vu

- Définitions générales : algorithmes, structures de données, structures de données abstraites...
- Analyse d'un algorithme itératif (Insertion-Sort)
- Notions de récursivité
- Analyse d'un algorithme récursif (Fibonacci)
- Tri par fusion (MergeSort)