LICENCE D'INFORMATIQUE

Sorbonne Université

LU3IN003 – Algorithmique

Cours 1 : Rappels sur les preuves d’algorithmes et l’analyse de complexité

Année 2024-2025

Responsables et chargés de cours
Fanny Pascual
Olivier Spanjaard

Equipe pédagogique, supports de TD, site web de l'UE

Chargés de cours et de TD :

Fanny Pascual, Olivier Spanjaard

fanny.pascual@lip6.fr olivier.spanjaard@lip6.fr

Chargés de TD :

Manuel Amoussou, Mohamed Ouaguenouni

Fascicules de TD :

La distribution sera faite en séance de travaux dirigés.

https://moodle-sciences-24.sorbonne-universite.fr/

Evaluation

- 50% CC, 50% Examen
- Contrôle continu :
  - Projet avec rapport et soutenance (15%)
    (un logiciel de **détection de plagiat** est appliqué sur tous les projets soumis)
  - Partiel (35%)
    (un exercice du fascicule de TD, ou une portion d’exercice, fera partie du sujet)

Ouvrages

Algorithmique de Cormen, Leiserson, Rivest, Stein.
DUNOD, 3$^{\text{ième}}$ édition, série Sciences Sup, 2010.

Eléments d'algorithmique de Berstel, Beauquier, Chrétienne.
MASSON, collection MIM.
http://www-igm.univ-mlv.fr/~berstel/Elements/Elements.pdf

155 exercices et problèmes corrigés d'algorithmique
Baynat, Chrétienne, Munier, Kedad-Sidhoum, Hanen, Picouleau.
DUNOD, Sciences Sup, 2010 (3^{e} édition).

Algorithms de Dasgupta, Papadimitriou, Vaziran.
McGraw Hill Higher Education, 2006.

Algorithm design de Kleinberg et Tardos.
Pearson, 2005.

Algorithms de Erickson.
https://jeffe.cs.illinois.edu/teaching/algorithms/

Sondage : Formation suivie

Quelle formation suivez-vous ?

A) Licence Informatique (parcours monodisciplinaire)
B) Double licence Biologie-Informatique
C) Double licence Electronique-Informatique
D) Double licence Informatique-Mathématiques
E) Double licence Lettres-Informatique
F) Autre

Pour participer au sondage et au quiz, à l'aide d'un smartphone :

socrative.com
∟ **Login**
  ∟ **Student Login**
    ∟ **Room Name : LU3IN003**
      ∟ **JOIN**

Contenu de l’UE

Rappels : Preuve et complexité d’algorithmes

Partie 1 : Programmation récursive
Diviser pour régner
Algorithmes d’exploration d’arbres d’énumération

Partie 2 : Algorithmes de parcours et applications
Rappels sur les graphes
Parcours en profondeur
Algorithmes de Dijkstra et de Prim

Partie 3 : Conception et analyse d’algorithmes gloutons

Partie 4 : Programmation dynamique

Calcul du *n*ième terme de la suite de Fibonacci

La suite de Fibonacci est définie ainsi :

$F_1 = 1$

$F_2 = 1$

$F_n = F_{n-1} + F_{n-2}$

Ce sont les nombres de Fibonacci :

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

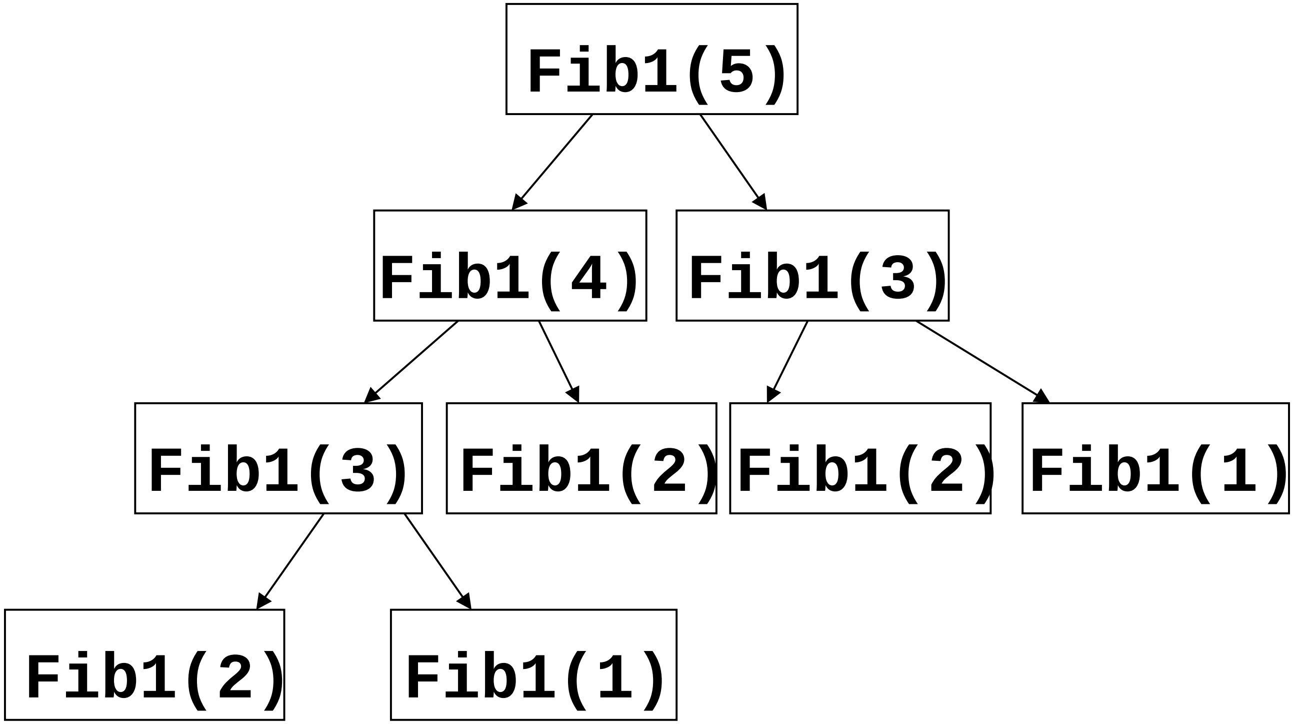
Ils croissent *très* vite: $F_{30} > 10^6$!

Leonardo da Pisa, dit Fibonacci

En fait, $F_n \approx 2^{0.694n}$, croissance exponentielle.

Un premier algorithme (récursif)

```
fonction Fib1(n)
si n = 1 retourner 1
si n = 2 retourner 1
retourner Fib1(n-1) + Fib1(n-2)
```



Analyse d'un algorithme

Analyser un algorithme, c'est répondre aux trois questions suivantes :

- Terminaison : Est-ce que l'algorithme se termine ?
- Validité : Est-ce que l'algorithme retourne le résultat attendu ?
- Complexité : Quelle est le nombre d'opérations élémentaires que réalise l'algorithme ?

Terminaison et validité de Fib1

```
fonction Fib1(n)
si n = 1 retourner 1
si n = 2 retourner 1
retourner Fib1(n-1) + Fib1(n-2)
```

Par **récurrence** : HR_n « Fib1(n) se termine et retourne F_n. »

Cas de base. OK pour *n=1* et *n=2* car Fib1(1) et Fib1(2) se terminent et retournent bien $1=F_1=F_2$.

Etape inductive. Montrons que :

Pour tout n ≥ 3, HR_{n-2} et HR_{n-1} vérifiées => HR_n vérifiée.

Fib1(n-1) se termine et retourne F_{n-1} d'après HR_{n-1}.

Fib1(n-2) se termine et retourne F_{n-2} d'après HR_{n-2}.

Donc Fib1(n) se termine et retourne $F_{n-1}+F_{n-2}=F_n$.

Conclusion. Pour tout n ≥ 1, Fib1(n) se termine et retourne F_n.

Quiz : Correction d’un algorithme

On appelle **suite de Syracuse** d'un entier n>0 une séquence d'entiers définie comme suit : on part de n ; s’il est pair, on le divise par 2 ; s’il est impair, on le multiplie par 3 et on ajoute 1. On s’arrête dès qu’on atteint 1.

L'algorithme ci-dessous, dont la spécification est de retourner le nombre de termes de la suite de Syracuse de n>0, est-il correct ? (correction = terminaison + validité)

```
Syracuse(n)
   x:=n
   nb:=0
   tant que x≠1
      nb:=nb+1
      si x est pair alors
         x:=x/2
      sinon
         x:=3x+1
   retourner nb+1
```

A) Oui
B) Non
C) Je ne sais pas

Complexité de Fib1

```
fonction Fib1(n)
si n = 1 retourner 1
si n = 2 retourner 1
retourner Fib1(n-1) + Fib1(n-2)
```

Soit $T(n)$ = nombre d'additions requises pour calculer *Fib1(n).*

Alors:

$T(1)=0$, $T(2)=0$

$T(n) = T(n-1) + T(n-2) + 1$

n	**1**	**2**	**3**	**4**	**5**	**6**	**7**
F_n	1	1	2	3	5	8	13
T(n)	0	0	1	2	4	7	12

D'où $T(n) = F_n - 1 \approx 2^{0.694n}$!

Complexité exponentielle.

Complexité exponentielle

$2^{0.694n}$ additions requises pour calculer F_n.

C'est-à-dire que le calcul de F_{200} requiert de l'ordre de 2^{140} additions.

Combien de temps cela prend sur une machine rapide ?

Frontier (Oak Ridge National Laboratory, Etats-Unis)

Ce supercalculateur américain occupe la première place du classement des supercalculateurs (juin 2024), avec une puissance de 1206 pétaflops, soit 1206×10^{15} opérations/sec.

Complexité exponentielle

$1206 \text{x} 10^{15} \approx 1206 \text{x} 2^{50} \approx 2^{60}$ opérations/sec.

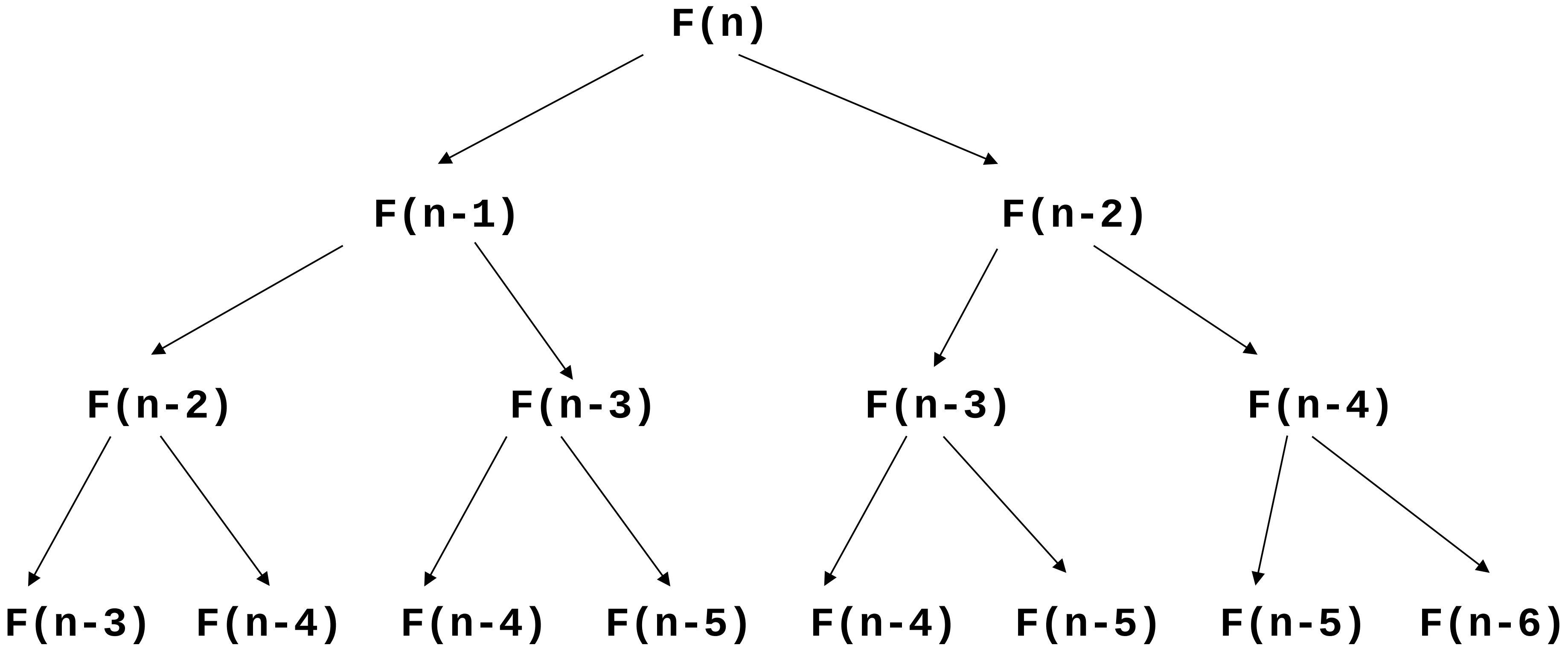
Calcul de F_{200} requiert ≈ 2^{140} opérations

→ 2^{80} secondes pour calculer F_{200} avec Frontier

Temps en secondes	Interprétation
2^{10}	17 minutes
2^{20}	12 jours
2^{30}	32 ans
2^{40}	33000 ans

Pourquoi **`Fib1`** est-il si mauvais ?

Observons l'arbre de récursion...



Les mêmes sous-problèmes sont résolus un grand nombre de fois !

Un autre algorithme (itératif)

Il y a *n* sous-problèmes F_1, F_2, …, F_n. Stocker les résultats intermédiaires plutôt que de relancer les calculs.

```
fonction Fib2(n)
 Créer un tableau fib[1..n]
 fib[1] = 1
 fib[2] = 1
 pour i = 3 à n:
     fib[i] = fib[i-1] + fib[i-2]
 retourner fib[n]
```

Les trois question usuelles :

1. Terminaison ? (évidente)
2. Validité ? (**invariant de boucle**)
3. Complexité ?

Invariant de boucle

- **Invariant de boucle** : propriété P qui, si elle est valide avant l'exécution d'un tour de boucle, est aussi valide *après* l'exécution du tour de boucle.
- On vérifie alors que les conditions initiales rendent la propriété P vraie en entrée du premier tour de boucle (cas de base) et on prouve l'invariant par récurrence.
- Un bon choix de la propriété P prouvera qu'on retourne bien ce que l'on recherche en sortie du dernier tour de boucle.
- Invariant de boucle pour **`Fib2`** :
  « **`fib[i]`** contient F_i à l'issue de l'itération i. »

  (*Preuve par récurrence omise ici*)

Quiz : Invariant de boucle

Quel invariant de boucle (vérifié à l'issue de chaque itération i) permet de prouver la validité de la fonction Inverse(n) ci-dessous, qui inverse l'ordre des digits d'un entier n = $D_1D_2...D_m$ pour obtenir un entier rev ?

```
Inverse(n)
   rev:=0
   tant que n>0 faire
        rev:=rev*10+n%10
        n:=n//10
   retourner rev
```

A) $n=D_1D_2...D_{m-i}$ et $rev=D_mD_{m-1}...D_{m-i+1}$

B) $n \neq rev$

C) $n=D_{m-i+1}...D_{m-1}D_m$ et $rev=D_{m-i}...D_2D_1$

D) $n=D_1D_2...D_m$ et $rev=D_mD_{m-1}...D_1$

Complexité de **`Fib2`**

Le contenu de la boucle consiste en une addition, et la boucle est itérée *n − 2* fois.

→ le nombre d'additions réalisées par **`Fib2`** est linéaire en *n*.

```
fonction Fib2(n)
  Créer un tableau fib[1..n]
  fib[1] = 1
  fib[2] = 1
   pour i = 3 à n:
       fib[i] = fib[i-1] + fib[i-2]
   retourner fib[n]
```

Autrement dit, le nombre d'opérations est proportionnel à *n*.

La complexité semble être O(n).

Complexité d’un algorithme

La complexité (temporelle) d’un algorithme est une évaluation du nombre d’instructions élémentaires (réalisées **en temps constant**) en fonction de la **taille de codage** des paramètres d’entrée (souvent notée ***n***), et en utilisant les notations de Landau (ordres de grandeur).

Complexité pire cas : on évalue le nombre d’instructions dans le pire des cas (**majorant** sur le nombre d’instructions).

On identifie généralement la complexité d’un algorithme avec son pire cas.

Complexité de **`Fib2`** (révisée)

```
fonction Fib2(n)
  Créer un tableau fib[1..n]
  fib[1] = 1
  fib[2] = 1
  pour i = 3 à n:
      fib[i] = fib[i-1] + fib[i-2]
  retourner fib[n]
```

Attention : la complexité de **`Fib2`** est-elle vraiment linéaire ?

Il est raisonnable de traiter une addition comme une opération élémentaire (en temps constant) si des petits nombres sont sommés, par exemple, des entiers sur 32 bits.

Mais le *n*ième nombre de Fibonacci comporte environ *0.694n* bits, ce qui peut largement dépasser 32 quand *n* augmente.

On va compter le nombre d'additions de bits.

Addition

Additionner deux nombres de *n* bits de long

[22]		1	0	1	1	0
[13]			1	1	0	1
	--					
[35]	1	0	0	0	1	1

Cela prend *O(n)* opérations… et on ne peut espérer mieux. L'addition prend un temps *linéaire*.

Complexité de **Fib2** (révisée)

```
fonction Fib2(n)
  Créer un tableau fib[1..n]
  fib[1] = 1
  fib[2] = 1
  pour i = 3 à n:
      fib[i] = fib[i-1] + fib[i-2]
  retourner fib[n]
```

Chaque addition nécessite de l'ordre de *i* opérations élémentaires (**`fib[i]`** comporte de l'ordre de *i* bits). On le fait pour *i=3* à *n* :

$$\sum_{i=3}^{n} i = \frac{n(n+1)}{2} - 3$$

De l'ordre de n^2 opérations élémentaires → O(n^2)

Cela <u>semble</u> être une complexité quadratique en *n*.

Taille d'une instance d'un problème

Plusieurs définitions de la taille d'une instance sont possibles dans la mesure où une même instance peut s’énoncer de différentes manières.

Exemple :
110 (base 2) et 20 (en base 3) sont deux représentations possibles de l’entier 6 (en base 10).

La plupart des représentations raisonnables d’une instance conduisent à des tailles similaires.

Plutôt que de formaliser cette notion, nous la préciserons pour chaque problème traité.

Exemples :
Multiplication de deux entiers sur n bits → taille : n
Tri d'un tableau A[1...n] → taille : n
etc.

Complexité de **`Fib2`** (révisée bis)

```
fonction Fib2(n)
  Créer un tableau fib[1..n]
  fib[1] = 1
  fib[2] = 1
  pour i = 3 à n:
       fib[i] = fib[i-1] + fib[i-2]
  retourner fib[n]
```

La complexité de **`Fib2`** est $O(n^2)$ mais la donnée d'entrée est n, qui requiert de l'ordre de $log_2 n$ bits pour être stockée en mémoire.

On a $n=2^{log2(n)}$ d'où $n^2=4^{log2(n)}$.

La complexité en fonction de la taille $log_2 n$ de la donnée d'entrée s'écrit donc $\boldsymbol{O(4^{log2(n)})}$, autrement dit elle est exponentielle en la taille de la donnée d'entrée.

Toutefois, tout algorithme calculant F_n aura une complexité exponentielle car F_n comporte de l'ordre de $0.694n$ bits, une sortie de taille exponentielle par rapport à la taille $log_2 n$ de l'entrée.

Quiz : Complexité

Quelle est la complexité de la fonction Fun ci-dessous, qui prend en entrée un entier $n \geq 0$?

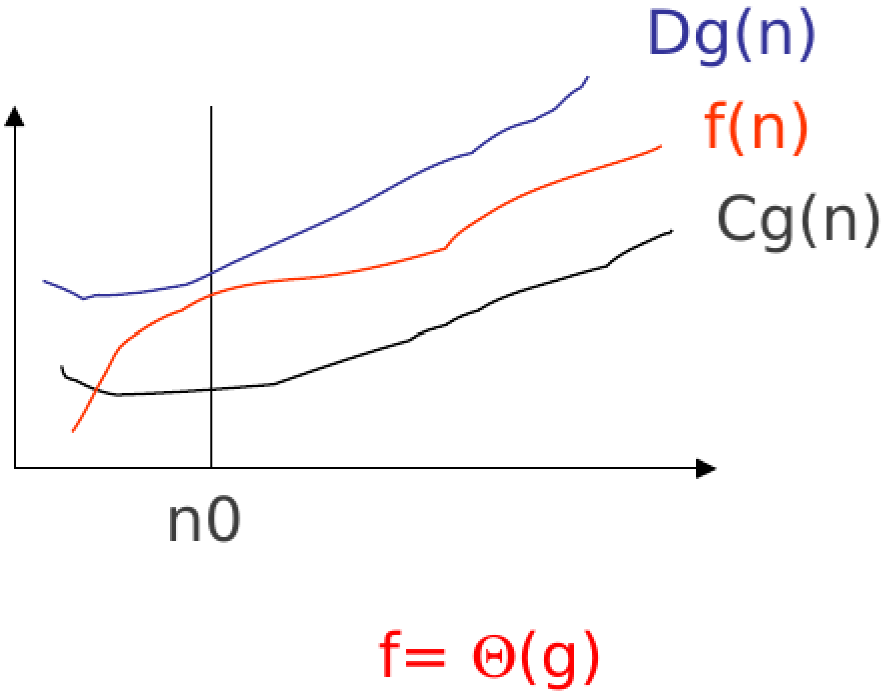
```
Fun(n)
   si $n \leq 1$ alors retourner n
   sinon retourner Fun(n//2)
```

A) O(log log n)
B) O(log n)
C) O(n)
D) $O(2^n)$

Rappel sur les notations de Landau

Soient f et g deux fonctions de N dans N.

$f \in O(g)$
s’il existe une constante D positive et un entier n_0 tels que:
$n > n_0$, $f(n) \leq Dg(n)$.



$f \in \Omega(g)$
s’il existe une constante C positive et un entier n_0 tels que:
$n > n_0$, $Cg(n) \leq f(n)$.

$f \in \Theta(g)$
s’il existe 2 constantes C et D positives et un entier n_0 tels que:
$n > n_0$, $Cg(n) \leq f(n) \leq Dg(n)$.

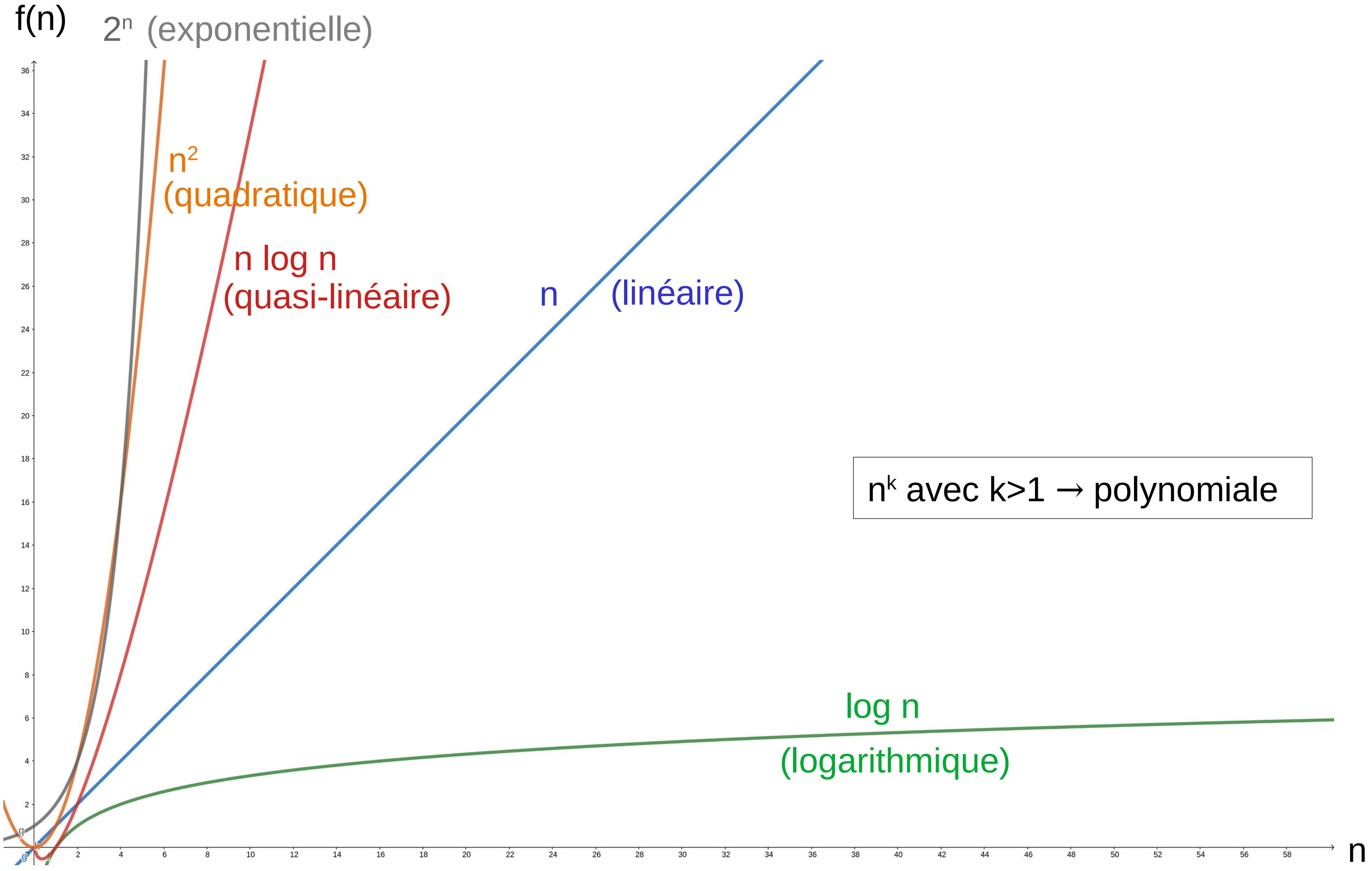
Exemples : $100n^2 + 4n\log_2(n) \in O(n^2)$; $2n + n^{10} \in O(2^n)$;
Il n’existe aucun entier K tel que $2^n \in O(n^K)$

Quelques règles utiles

Les quelques règles suivantes permettent de simplifier les complexités en omettant des termes dominés :

- Les coefficients peuvent être omis : $14n^2$ devient n^2
- n^a domine n^b si $a > b$: par exemple, n^2 domine n
- Une exponentielle domine un polynôme : 3^n domine n^5 (cela domine également 2^n)
- De même, un polynôme domine un logarithme : n domine $(\log n)^3$. Cela signifie également, par exemple, que n^2 domine $n \log n$.

Courbes de complexités fréquentes



n^k avec k>1 → polynomiale

Quiz : Notation O(·)

Laquelle des expressions suivantes n’est pas en $O(n^2)$?

A) $15^{10} * n + 12099$
B) $n^{1.98}$
C) $n^3/\sqrt{n}$
D) $2^{20} * n$

Quiz : Complexités croissantes

Quelle séquence correspond bien à l’ordre des complexités croissantes ?

A) $\Theta(n^2)$, $\Theta(n \log n)$, $\Theta(n^{\sqrt{n}})$, $\Theta(2^n)$
B) $\Theta(n^2)$, $\Theta(n \log n)$, $\Theta(2^n)$, $\Theta(n^{\sqrt{n}})$
C) $\Theta(n \log n)$, $\Theta(n^2)$, $\Theta(n^{\sqrt{n}})$, $\Theta(2^n)$
D) $\Theta(n \log n)$, $\Theta(n^2)$, $\Theta(2^n)$, $\Theta(n^{\sqrt{n}})$

Attention !

LU2IN003 (Algorithmique I)

Ω(n) : minorant du nombre d’opérations sur toutes les instances
O(n) : majorant du nombre d’opérations sur toutes les instances
Θ(n) : si le minorant et le majorant du nombre d’opérations sur toutes les instances coïncident.

LU3IN003 (Algorithmique II)

Ω(n) : minorant du nombre d’opérations sur les pires instances
O(n) : majorant du nombre d’opérations sur les pires instances
Θ(n) : si le minorant et le majorant du nombre d’opérations sur les pires instances coïncident.

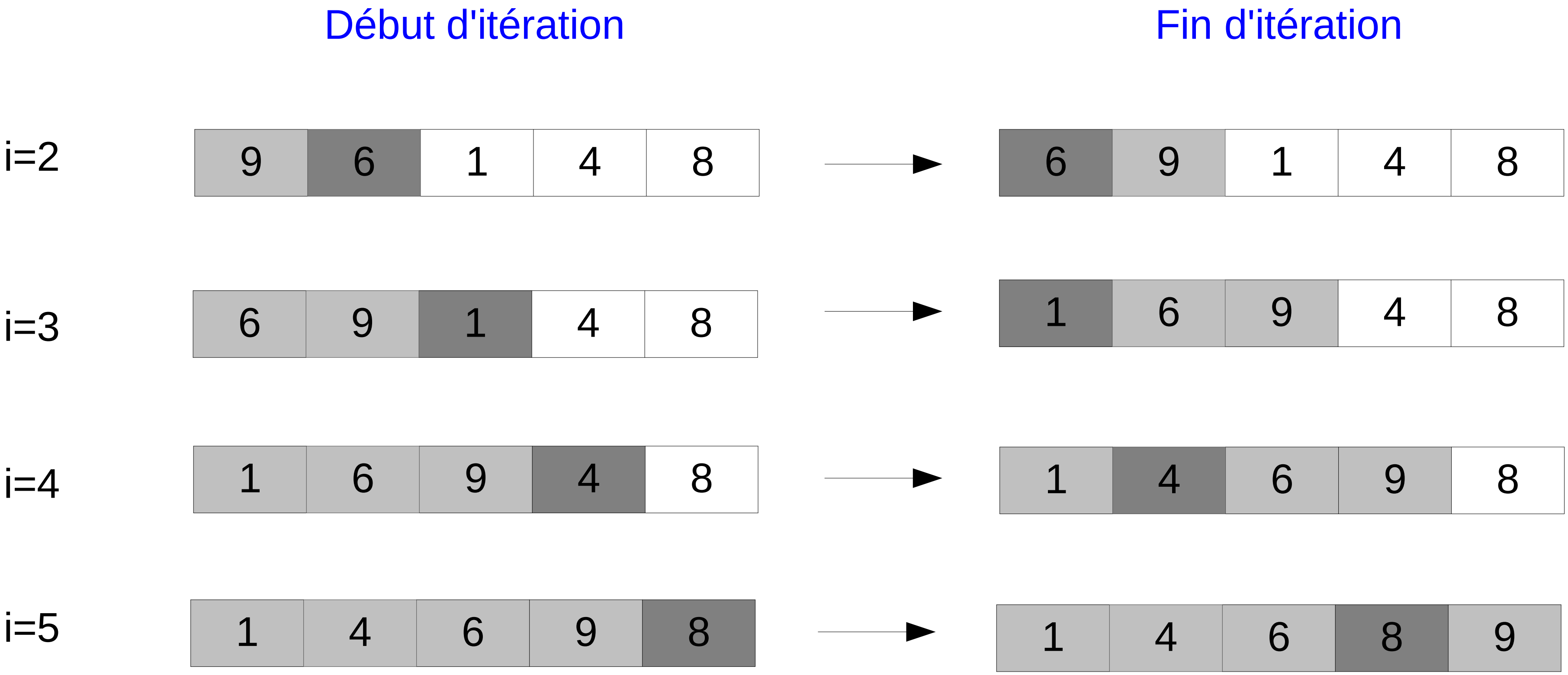
Illustrons cette différence en analysant le tri par insertion.

Exemple : complexité de **TRI_INS**

Algorithme de tri d’un tableau T[1...n].
Supposons que seules les comparaisons de 2 entiers du tableau soient comptées.

```
Procédure TRI_INS(T : tableau d’entiers, n : entier);
                                        # comparaisons
Pour i de 2 à n faire ------------------------------- 0
     z:=T[i]; k:=i-1; ------------------------------ 0
     Tantque k>0 et T[k]>z faire --------------------- $N_2+N_3+…+N_n$
          T[k+1]:=T[k]; k:=k-1 ---------------------- 0
     Fintantque;
     T[k+1]:=z; ------------------------------------------ 0
Finpour.
```

Exemple : un déroulement de **`TRI_INS`**



	Début d'itération						Fin d'itération				
i=2	9	6	1	4	8	→	6	9	1	4	8
i=3	6	9	1	4	8	→	1	6	9	4	8
i=4	1	6	9	4	8	→	1	4	6	9	8
i=5	1	4	6	9	8	→	1	4	6	8	9

Déterminer la complexité « en Θ » de `TRI_INS`

Comme $N_i \leq i-1$, on a: **TRI_INS**(n) ≤ 1/2 n(n-1).
Donc **TRI_INS**(n) $\in O(n^2)$.

Si les éléments du tableau sont initialement rangés dans l'ordre décroissant strict :
on a $N_i = i-1$ pour tout i de 2 à n.
Or pour un énoncé quelconque de taille n, on a :
$N_i \leq i-1$ pour i de 2 à n,
Il en résulte que : **TRI_INS**(n)= 1/2 n(n-1).
Donc **TRI_INS**(n) $\in \Omega(n^2)$.

L'algorithme **TRI_INS** est donc de complexité pire cas $\Theta(n^2)$.

Retour sur la remarque précédente

LU2IN003 (Algorithmique I)
$\Omega(n)$: minorant du nombre d’opérations sur toutes les instances.
$O(n)$: majorant du nombre d’opérations sur toutes les instances.
$\Theta(n)$: si le minorant et le majorant du nombre d’opérations sur toutes les instances coïncident.

Le tri par insertion est en $\Omega(n)$ (si le tableau est trié en ordre croissant en entrée) et en $O(n^2)$ (en majorant le nombre d’opérations). Le minorant et le majorant ne coïncident pas et le tri par insertion n’est donc pas en $\Theta(n)$ car il ne réalise pas le même nombre de comparaisons sur toutes les instances.

LU3IN003 (Algorithmique II)
$\Omega(n)$: minorant du nombre d’opérations sur les pires instances.
$O(n)$: majorant du nombre d’opérations sur les pires instances.
$\Theta(n)$: si le minorant et le majorant du nombre d’opérations sur les pires instances coïncident.

Le tri par insertion est en $\Omega(n^2)$ (si le tableau est trié en ordre décroissant en entrée) et en $O(n^2)$ (en majorant le nombre d’opérations). Le minorant et le majorant coïncident et le tri par insertion est donc en $\Theta(n^2)$, c’est à dire qu’il réalise exactement de l’ordre de n^2 opérations sur les pires instances.

Quiz : Notations de Landau

Parmi les affirmations suivantes, laquelle est exacte ?

A) Un algorithme en $\Theta(n^2)$ est plus lent sur **toutes** les données qu'un algorithme en $\Theta(n)$.

B) Un algorithme en $\Theta(n^2)$ est plus lent sur **certaines** données qu'un algorithme en $\Theta(n)$.

C) Un algorithme en $O(n^2)$ est plus lent sur **toutes** les données qu'un algorithme en $O(n)$.

D) Un algorithme en $O(n^2)$ est plus lent sur **certaines** données qu'un algorithme en $O(n)$.