

Cours n° 1 : introduction à l'algorithmique

Christophe Gonzales



HUGO3 — Algorithmique

2 questions :

- ▶ Êtes-vous prêt à attendre 5 minutes que votre GPS calcule votre itinéraire ?
- ▶ Êtes-vous prêt à attendre 10 minutes pour obtenir une réponse de google ?

⇒ Besoin de logiciels « efficaces »

But du cours d'algorithmique

Donner quelques clefs pour réaliser de tels logiciels.

2 points clefs pour des logiciels efficaces :

- ▶ Méthodes de « résolution » rapides : 1ère partie du cours
- ▶ Accéder rapidement à l'information : 2ème partie du cours

- ▶ Note finale = 30% contrôle continu + 70% examen
- ▶ Contrôle continu : 2 examens en cours de semestre

- ① En route vers l'algorithmique
- ② En route vers un critère d'efficacité
- ③ Complexité et notations de Landau

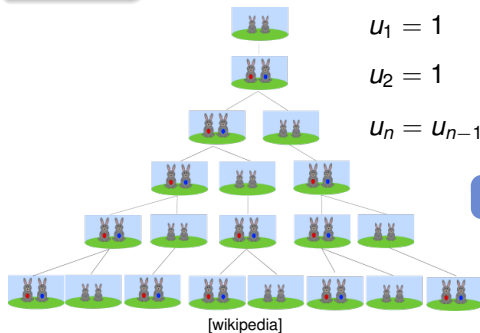
① En route vers l'algorithmique

Exemple récréatif : la reproduction des lapins

Problème (Leonardo da Pisa, dit Fibonacci, 1202) :

- ▶ Couple de lapins dans un lieu isolé.
- ▶ Commence à enfanter 2 mois après leur naissance.
- ▶ Engendre tous les mois un nouveau couple.

Question : Combien de couples après un an ?



Suite de Fibonacci

Suite de Fibonacci : 1er programme C

$$\begin{cases} u_1 = 1 \\ u_2 = 1 \\ u_n = u_{n-1} + u_{n-2} \text{ pour tout } n \geq 3 \end{cases}$$

```
#include <stdio.h>
#include <stdlib.h>

// renvoie la valeur de  $u_n$ 
int fib1 (int n) {
    if (n <= 2) return 1;
    return fib1(n-1) + fib1(n-2);
}

int main (int argc, char** argv) {
    int n = atoi (argv[1]);
    printf ("fibonacci(%d) = %d\n", n, fib1(n));
    return 0;
}
```


Suite de Fibonacci : 2ème programme C

$$u_1 = 1 \quad u_2 = 1 \quad u_n = u_{n-1} + u_{n-2} \text{ pour tout } n \geq 3$$

```
#include <stdio.h>
#include <stdlib.h>

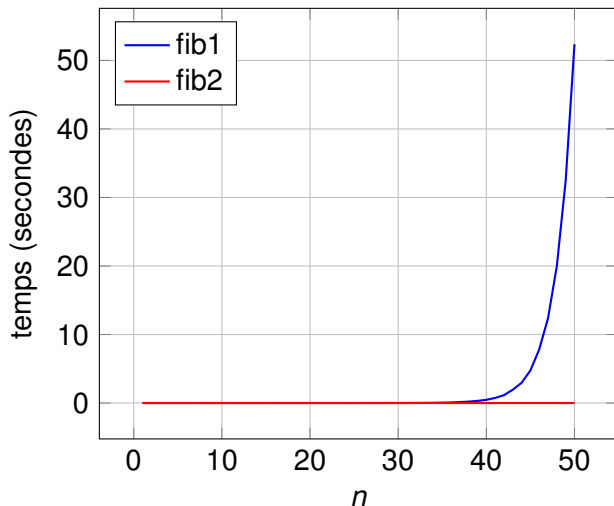
// renvoie la valeur de U_n
int fib2 (int n) {
    int u_n          = 1;
    int u_n_moins_1 = 1;
    int u_n_moins_2;

    for (int i = 3; i <= n; i++) {
        u_n_moins_2 = u_n_moins_1;
        u_n_moins_1 = u_n;
        u_n          = u_n_moins_1 + u_n_moins_2;
    }

    return u_n;
}

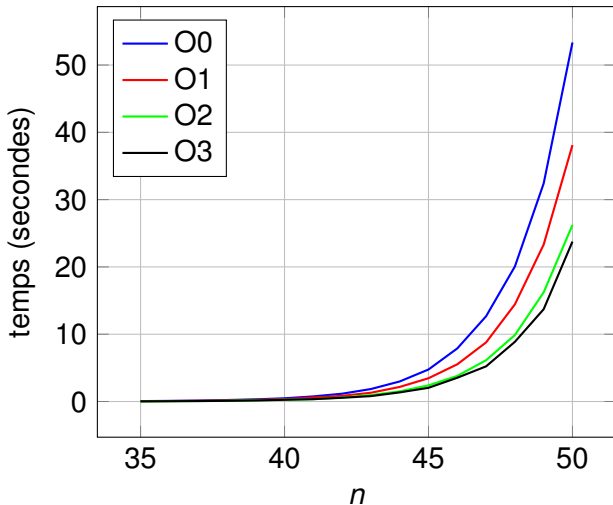
int main (int argc, char** argv) {
    int n = atoi (argv[1]);
    printf ("fibonacci(%d) = %d\n", n, fib2(n));
    return 0;
}
```

Comparaison de fib1 et fib2



Temps d'exécution \implies fib2 « meilleur » que fib1

Autres programmes pour Fibonacci(n)



Compilation de fib1 avec `gcc -Ox -o fib1 fib1.c!`

- ▶ Temps d'exécution = mauvais critère car :
 - ① Prend en compte l'**exécutable**, pas le **code source**
 - ② Dépend :
 - ▶ du compilateur (gcc, clang, mingw, mvsc, *etc.*)
 - ▶ des options de compilation
 - ▶ de l'architecture du processeur
 - ▶ du système d'exploitation
 - ▶ du langage de programmation
 - ▶ *etc.*
- ▶ Se concentrer sur le code source
- ▶ Trouver un critère d'efficacité sur ce code

Variations sur les langages de programmation

```
int fib1 (int n) {  
    if (n <= 2) return 1;  
    return (fib1(n-1) + fib1(n-2));  
}
```

langage C

```
let rec fib1 n =  
    if n <= 2 then 1  
    else fib1(n-1) + fib1(n-2)
```

langage Ocaml

```
def fib1 (n):  
    return fib1(n-1) + fib1(n-2) if n>=2 else 1
```

langage Python

► Se concentrer sur le raisonnement : langage algorithmique :

```
fonction fib1 (n) :  
    si n <= 2 alors  
        retourner 1  
    sinon  
        retourner fib1(n-1) + fib1(n-2)  
    finsi
```

langage algorithmique

Algorithmique : késako ?

Définition : Algorithmique

Étude, science des algorithmes.

Définition : Algorithme

[Knuth (2011), page 1]

Suite de règles :

- ▶ définies de façon précise
 - ▶ disant comment produire des informations de sortie
 - ▶ en fonction d'informations d'entrée
- ▶ **Exemple GPS** : Infos d'entrée : 2 points A et B sur une carte
Infos de sortie : chemin rapide de A vers B

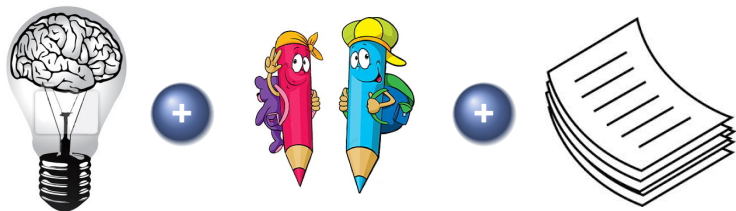


Règles \neq programme informatique !

\implies indépendant du langage de programmation

\implies raisonnement/recette (en français)

Implanter l'algorithme = le programmer (en C par exemple)



« Ce qui se conçoit bien s'énonce clairement
et les mots pour le dire arrivent aisément. »

Nicolas Boileau

① **Spécification formelle du problème :**

Entrées, sorties, lien entre elles.

② **Séparation du problème en morceaux plus petits :**

Spécifier chaque sous-problème et les regrouper à la fin.

③ **Construction d'algorithmes :**

Contraintes d'efficacité, vérification, preuve.

④ **Construction de structures de données :**

Lien entre algorithme et implantation.

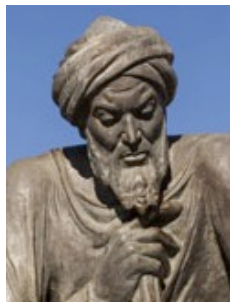
⑤ **Implantation**

Culture : problèmes classiques en algorithmique

- ▶ Algèbre et arithmétique (racines carrées, polynômes, fonctions, calcul matriciel, PGCD, *etc.*)
- ▶ Tri et recherche d'éléments
- ▶ Parcours de graphe (plus court chemin, tri topologique, flot maximum, *etc.*)
- ▶ Cryptographie
- ▶ Chaînes de caractères (recherche de sous-séquence, plus longue sous-séquence commune, *etc.*)
- ▶ Compression de données (Huffman, *etc.*)
- ▶ Géométrie algorithmique (enveloppe convexe, deux points les plus proches, *etc.*)

⇒ facile de trouver de bons algorithmes pour ces problèmes.

- ▶ Étymologie : mathématicien perse Muhammad Ibn Musa **Al-Khwârizmî** (IXe siècle).
- ▶ Né à Khiva, Ouzbékistan (cf. statue).
- ▶ Vit à Bagdad pendant la dynastie abbasside.
- ▶ Un des pères de l'algèbre : traité *Kitab **al jabr w'al muqabalah*** \implies méthodes d'extraction de racines carrées. . .
- ▶ Traité sur le système de numération décimale \implies méthodes de calcul d'additions. . .



② En route vers un critère d'efficacité

Retour sur Fibonacci : comptage d'opérations

```
1 fonction fib2 (n) :
2    $U_n \leftarrow 1$ 
3    $U_{n-1} \leftarrow 1$ 
4   pour i variant de 3 à n faire
5      $U_{n-2} \leftarrow U_{n-1}$ 
6      $U_{n-1} \leftarrow U_n$ 
7      $U_n \leftarrow U_{n-1} + U_{n-2}$ 
8   fait
9
10  retourner  $U_n$ 
```

Ligne	Opération	# exécutions
2	affectation	1
3	affectation	1
4	affectation	$n - 1$
4	comparaison	$n - 1$
5	affectation	$n - 2$
6	affectation	$n - 2$
7	addition	$n - 2$
7	affectation	$n - 2$

$$\text{Total} = 6 \times n - 8$$

$$\text{Total} \rightarrow 6 \times n \text{ quand } n \rightarrow +\infty$$

Règle n° 1 : on ne garde que le terme dominant.

Code en C

```
int func1 ( int n) {  
    int u_n = 2;  
    return u_n;  
}  
  
int func2 ( int n) {  
    if (n == 2) return 0;  
    else return 1;  
}
```

Code assembleur

```
func1(int) :  
    movl    $2, %eax  
    ret  
  
func2(int) :  
    xorl    %eax, %eax  
    cmpl   $2, %edi  
    setne  %al  
    ret
```

⇒ 1 opération \neq 1 instruction machine

⇒ se contenter d'un ordre de grandeur sur le nombre d'opérations

Règle n° 2 : on ne garde pas les constantes multiplicatives.

⇒ # opérations de fib2 ($\approx 6 \times n$) de l'ordre de n

Résumé des règles n° 1 et 2

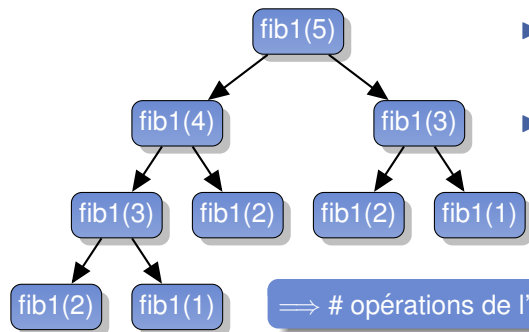
```
1 fonction fib2 (n) :  
2    $U_n \leftarrow 1$   
3    $U_{n-1} \leftarrow 1$   
4   pour i variant de 3 à n faire  
5      $U_{n-2} \leftarrow U_{n-1}$   
6      $U_{n-1} \leftarrow U_n$   
7      $U_n \leftarrow U_{n-1} + U_{n-2}$   
8   fait  
9  
10  retourner  $U_n$ 
```

- ▶ boucle \implies grossièrement de l'ordre de n itérations
- ▶ intérieur de la boucle \implies pas d'appel de fonctions
 \implies nombre constant d'opérations
 \implies compte pour 1
- ▶ boucle \implies globalement de l'ordre de n opérations
- ▶ avant la boucle \implies nombre constant d'opérations \implies 1
- ▶ n grand $\implies n \gg 1 \implies$ ordre de grandeur de $\text{fib2} = n$

Retour vers fib1

```
1 fonction fib1 (n) :  
2   si n <= 2 alors  
3     retourner 1  
4   sinon  
5     retourner fib1(n-1) + fib1(n-2)  
6   fin
```

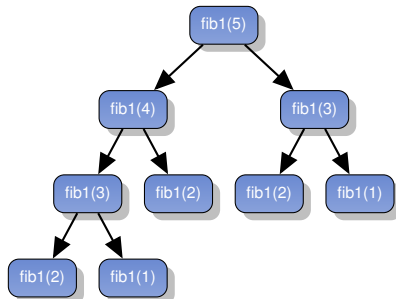
Exécution de fib1(5) :



- ▶ Boîtes fib1(1) et fib1(2) :
1 test
- ▶ Autres boîtes :
1 test + 2 appels + 1 addition

⇒ # opérations de l'ordre du nombre d'appels à fib1

Calcul du nombre de boîtes de fib1



▶ $T(n) = \#$ boîtes pour fib1(n)

▶ $T(1) = T(2) = 1$

▶ $T(n) = 1 + T(n-1) + T(n-2)$
pour $n \geq 3$

▶ $T(n) \approx \frac{2}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^n - 1$

opérations $\approx \left(\frac{1 + \sqrt{5}}{2} \right)^n \approx 2^{0,694n}$

n	T(n)	# opérations fib1	# opérations fib2
5	9	11	5
10	109	123	10
20	13529	15127	20
30	1664079	1860498	30
40	204668309	228826127	40
50	25172538049	28143753123	50

Calcul des ordres de grandeur du nombre d'opérations :

- ▶ **Règle n° 1** : on ne garde que le terme dominant.
- ▶ **Règle n° 2** : on ne garde pas les constantes multiplicatives.
- ▶ **Règle n° 3** : ensemble fini (constant) d'opérations (hors appel de fonction) : ordre = 1
- ▶ **Règle n° 4** : ordre d'un appel de fonction = ordre de grandeur du # d'opérations de la fonction
- ▶ **Règle n° 5** : boucle : de l'ordre du nombre d'itérations \times l'ordre du # d'opérations à l'intérieur de la boucle

Multiplication de 2 matrices

- ▶ 2 matrices : $A_{N \times M} = (a_{i,k}) \in \mathbb{R}^{N \times M}$, $B_{M \times P} = (b_{k,j}) \in \mathbb{R}^{M \times P}$
- ▶ $M_{N \times P} = (m_{i,j}) \in \mathbb{R}^{N \times P} = A_{N \times M} \otimes B_{M \times P}$
- ▶ $m_{(i,j)} = \sum_{k=1}^M a_{(i,k)} \times b_{(k,j)}$ pour tout i, j

```
fonction produit ( $A_{N \times M}$ ,  $B_{M \times P}$ ) :  
  pour i variant de 1 à N faire  
    pour j variant de 1 à P faire  
       $m_{(i,j)} \leftarrow 0$   
      pour k variant de 1 à M faire  
         $m_{(i,j)} \leftarrow m_{(i,j)} + a_{(i,k)} \times a_{(k,j)}$   
      fait  
    fait  
  fait
```

- ▶ Ordre de grandeur du # d'opérations de produit =

Puissance d'un nombre

$$\blacktriangleright x^n = \begin{cases} 1 & \text{si } n = 0 \\ x^{n/2} \times x^{n/2} & \text{si } n \text{ est pair} \\ x \times x^{\lfloor n/2 \rfloor} \times x^{\lfloor n/2 \rfloor} & \text{si } n \text{ est impair} \end{cases}$$

```
fonction puissance (x, n) :  
  si n = 0 alors  
    retourner 1  
  sinon  
    x_n2 ← puissance (x, [n/2])  
    si n est pair alors  
      retourner x_n2 × x_n2  
    sinon  
      retourner x × x_n2 × x_n2  
  finsi  
finsi
```

► Ordre de grandeur du # d'opérations de puissance =

Ordres de grandeur \implies « Complexité asymptotique »

- ▶ complexité de puissance = $O(\log(n))$
- ▶ complexité de produit = $O(N \times M \times P)$

2 types de complexité intéressantes :

- ▶ Complexité en temps : coût du temps d'exécution
- ▶ Complexité en espace : coût en consommation mémoire



Exprimées en fonction des paramètres d'entrée !

③ Complexité et notations de Landau

Différentes complexités

```
1 fonction bizarre (n) :  
2   si n est pair alors  
3     retourner fib1 (n)  
4   sinon  
5     retourner fib2 (n)  
6   finsi
```

- 1 Complexité au pire cas : borne supérieure du nombre d'instructions. Ex : $O(\text{bizarre}) = O(\text{fib1}) = O(2^{0,694n})$
- 2 Complexité dans le meilleur cas : borne inférieure du nombre d'instructions. Ex : $O(\text{bizarre}) = O(\text{fib2}) = O(n)$
- 3 Complexité en moyenne : moyenne du coût pour l'ensemble des valeurs possibles d'entrée



nécessite une distribution de probabilité des valeurs d'entrée.

Ex : si 1 chance sur 3 que n soit pair :

$$O(\text{bizarre}) = O\left(\frac{1}{3}2^{0,694n} + \frac{2}{3}n\right) = O(2^{0,694n})$$

En pratique : souvent, complexité dans le pire cas.

Idée : comparaison de 2 fonctions f et g

- ▶ fonction f = « vrai » temps d'exécution
- ▶ fonction g = « approximation » de f
= ordre de grandeur vue précédemment
- ▶ Notation de Landau \implies comparaison de f et g

4 ordres de grandeurs / comparaisons :

- ▶ O (grand O) : borné supérieurement
- ▶ Ω : borné inférieurement
- ▶ Θ : borné supérieurement et inférieurement
- ▶ o (petit O) : dominé (négligeable)

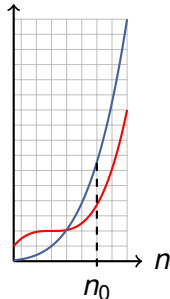


Complexités asymptotiques !

Définition du grand O

- ▶ f et g deux fonctions de \mathbb{R} dans \mathbb{R}
- ▶ $f \in O(g)$: f est asymptotiquement bornée supérieurement par g ssi :

$$\exists c \in \mathbb{R}_*^+, \exists n_0 \in \mathbb{R}^+ \text{ tels que } \forall n > n_0, f(n) \leq c \times g(n).$$



Exemples :

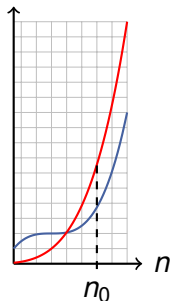
- ▶ $10n \in O(n^2)$: $c = 1, n_0 = 10$
- ▶ $n^2 + 10n \in O(n^2)$: $c = 11, n_0 = 0$
- ▶ $10n \in O(n)$: $c = 10, n_0 = 0$
- ▶ $n \notin O(\log(n))$

⇒ Règles n° 1 et n° 2

Définition du Ω

- ▶ f et g deux fonctions de \mathbb{R} dans \mathbb{R}
- ▶ $f \in \Omega(g)$: f est asymptotiquement bornée inférieurement par g ssi

$$\exists c \in \mathbb{R}_*^+, \exists n_0 \in \mathbb{R}^+ \text{ tels que } \forall n > n_0, f(n) \geq c \times g(n).$$

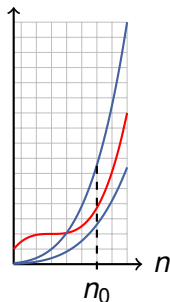


- ▶ Exemple : $n^2 \in \Omega(10n)$
- ▶ Remarque n° 1 : $f \in \Omega(g) \iff g \in O(f)$
- ▶ Remarque n° 2 : ici, définition de Knuth \neq définition de Hardy-Littlewood

Définition du Θ

- ▶ f et g deux fonctions de \mathbb{R} dans \mathbb{R}
- ▶ $f \in \Theta(g)$: f est asymptotiquement bornée supérieurement et inférieurement par g ssi $f \in O(g)$ et $f \in \Omega(g)$:

$$\exists c, d \in \mathbb{R}_*^+, \exists n_0 \in \mathbb{R}^+ \text{ t.q. } \forall n > n_0, d \times g(n) \leq f(n) \leq c \times g(n)$$

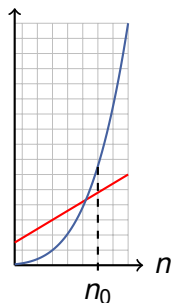


- ▶ Exemple : $3n^2 + 5n \in \Theta(n^2)$:
 $d = 1, c = 10, n_0 = 1$

Définition du petit o

- ▶ f et g deux fonctions de \mathbb{R} dans \mathbb{R}
- ▶ $f \in o(g)$: f est négligeable asymptotiquement devant g ssi :

$$\forall c \in \mathbb{R}_*^+, \exists n_0 \in \mathbb{R}^+ \text{ t.q. } \forall n > n_0, f(n) \leq c \times g(n)$$



- ▶ Exemple : $n \in o(n^2)$: $n_0 = 1/c$

Interprétations

- ▶ $f \in O(g) \implies f/g$ bornée supérieurement : $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} < +\infty$
- ▶ $f \in \Omega(g) \implies f/g$ bornée inférieurement : $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} > 0$
- ▶ $f \in \Theta(g) \implies f/g$ bornée : $0 < \lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} < +\infty$
- ▶ $f \in o(g) \implies f/g$ tend vers 0 : $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = 0$

Propriétés

- ▶ $f_1 \in O(g_1)$ et $f_2 \in O(g_2)$ alors :
- ▶ $f_1 + f_2 \in O(\max(g_1, g_2))$: Règle n° 1
- ▶ $f_1 \times f_2 \in O(g_1 \times g_2)$
- ▶ $cst \times f_1 \in O(cst \times g_1) = O(g_1)$: Règle n° 2

Calcul du O (ou Θ) d'une fonction :

- ▶ **Règle n° 1** : on ne garde que le terme dominant.
- ▶ **Règle n° 2** : on ne garde pas les constantes multiplicatives.
- ▶ **Règle n° 3** : ensemble fini (constant) d'opérations (hors appel de fonction) : $O(1)$
- ▶ **Règle n° 4** : appel de fonction : $O(\text{fonction})$
- ▶ **Règle n° 5** : $O(\text{boucle}) = \text{nombre d'itérations} \times O(\text{instructions à l'intérieur de la boucle})$

Complexité de fib1

```
1 fonction fib1 (n) :  
2   si n <= 2 alors  
3     retourner 1  
4   sinon  
5     retourner fib1(n-1) + fib1(n-2)  
6   finsi
```

► $T(n) = \# \text{ boites pour fib1}(n)$

► $\begin{cases} T(1) = T(2) = 1 \\ T(n) = 1 + T(n-1) + T(n-2) \text{ pour } n \geq 3 \end{cases}$

► $T(n) \approx \frac{2}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^n - 1 \propto 2^{0,694n}$



ici : complexité pire cas = meilleur cas = moyenne

fib1 $O(2^{0,694n})$, fib1 $\Omega(2^{0,694n})$, fib1 $\Theta(2^{0,694n})$

Complexité de fib2

```
1 fonction fib2 (n) :  
2    $u_n \leftarrow 1$   
3    $u_{n-1} \leftarrow 1$   
4   pour i variant de 3 à n faire  
5      $u_{n-2} \leftarrow u_{n-1}$   
6      $u_{n-1} \leftarrow u_n$   
7      $u_n \leftarrow u_{n-1} + u_{n-2}$   
8   fait  
9  
10  retourner  $u_n$ 
```

► Nombre d'opérations = $6 \times n - 8$



ici : complexité pire cas = meilleur cas = moyenne

fib2 $O(n)$, fib2 $\Omega(n)$, fib2 $\Theta(n)$

Fonctions induisant des temps de réponse « raisonnables » :

- ▶ Complexité constante (indépendante de n) : $O(1)$
- ▶ Complexité sous-linéaire (e.g., logarithmique) : $O(\log(n))$
- ▶ Complexité linéaire : $O(n)$
- ▶ Complexité quasi linéaire : $O(n \log(n))$
- ▶ Complexité quadratique : $O(n^2)$
- ▶ Complexité polynomiale : $O(n^k)$


Fonctions induisant des temps de réponse « prohibitifs » :

- ▶ Complexité exponentielle : $O(2^n)$, $O(\exp(n))$
- ▶ Complexité factorielle : $O(n!)$, asymptotiquement équivalente à $O(n^n)$

Temps d'exécutions possibles pour $n = 100$:

- ▶ Complexité constante : $O(1) = 1\text{ms}$
- ▶ Complexité sous-linéaire $O(\log(n)) = 6,6\text{ secondes}$
- ▶ Complexité linéaire : $O(n) = 100\text{ secondes}$
- ▶ Complexité quasi linéaire : $O(n \log(n)) = 11\text{ minutes}$
- ▶ Complexité quadratique : $O(n^2) = 7\text{ heures}$
- ▶ Complexité polynomiale : $O(n^k) = 11\text{ jours}$
- ▶ Complexité exponentielle : $O(2^n) = 4 \times 10^{22}\text{ années}$
- ▶ Complexité factorielle : $O(n!) = 10^{148}\text{ années}$

Complexité de problème

- ▶ Difficulté intrinsèque du problème, indépendamment des algorithmes qui existent pour le résoudre.
 - ▶ Hiérarchie (polynomiale) : hiérarchie de difficulté.
 - ▶ Complexité polynomiale : problèmes « faciles ».
-
- ▶ **Exemple** : Recherche d'un élément dans un tableau = complexité polynomiale.
-
- ▶  Complexité de problème \neq complexité d'algorithme :
 - ▶ Programmation linéaire : complexité polynomiale.
 - ▶ Algo du simplexe : complexité pire cas exponentielle.
 - ▶ Complexité de problème : pas le but du cours d'algorithmique.

Analyse d'algorithme : les questions qui se posent :

- ▶ Complexité : combien opérations élémentaires ?
- ▶ Terminaison : est-ce que l'algorithme se termine ?
- ▶ Validité : est-ce que l'algorithme retourne le résultat attendu ?