LICENCE D'INFORMATIQUE
Sorbonne Université

3I003 – Algorithmique
Cours 1 : Preuve et complexité d'algorithmes

Année 2018-2019

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 :

Nawal Benabbou, Anne-Elisabeth Falq, Pierre Fouilhoux, Maryse Pelletier, Lionel Tabourier.

Fascicules de TD :

à retirer, la distribution aura lieu en salle **14-15/506** (ALIAS)

Site web de l'UE :

https://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2018/ue/3I003-2018oct

Evaluation

- 40 % CC, 60 % Examen
- Contrôle continu :
  - 2 interrogations en TD (5%)
  - Mini-projet avec rapport et soutenance (15%)
  - Partiel (20%)

Ouvrages

Algorithmique
Cormen, Leiserson, Rivest, Stein
DUNOD, 3ième édition, série Sciences Sup, 2010.

Eléments d'algorithmique
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 (3e édition).

Algorithms
Dasgupta, Papadimitriou, Vazirani
McGraw Hill Higher Education, 2006.

Algorithm design
Kleinberg, Tardos
Pearson, 2005.

Contenu de l'UE

Rappels : Preuve et complexité d'algorithmes

Partie 1 : Programmation récursive

Introduction aux graphes

Partie 2 : Algorithmes de parcours et applications

Partie 3 : Conception et analyse d'algorithmes gloutons

Partie 4 : Programmation dynamique

Introduction

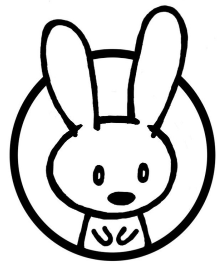
- Le mot *algorithme* est un dérivé du nom d'un illustre savant musulman d'origine iranienne **Muhammad Ibn Musa Al Khawarizmi** qui vécut au neuvième siècle de l'ère chrétienne, sous le règne du calife abbasside Al-Ma'mun.
- Al Khwarizmi a exposé les méthodes de base pour l'addition, la multiplication, la division, l'extraction de racines carrées, le calcul des décimales de π
- Ces méthodes sont précises, sans ambiguïté, mécanique, efficace, correcte

→ Ces méthodes sont des algorithmes !

Comparaison de deux algorithmes résolvant un même problème

Question

On suppose qu'un lapin devient fertile en exactement un mois, après quoi il produit un enfant par mois, pour toujours. En commençant avec un lapin, combien il y a de lapins après n mois ?



Evolution de la population de lapins

	Fertile	Non fertile
Initialement		
Un mois		
Deux mois		
Trois mois		
Quatre mois		
Cinq mois		

Suite de Fibonacci

Soit F_n = nombre de lapins au mois n

$F_1 = 1$

$F_2 = 1$

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

Ce sont les nombres de Fibonacci :

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

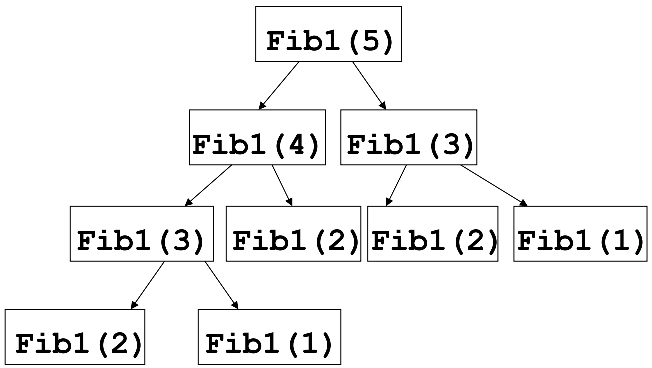
Ils croissent *très* vite: $F_{30} > 10^6$!

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

Leonardo da Pisa, dit Fibonacci

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.

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:

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

Mais rappelons $F_n = F_{n-1} + F_{n-2}$. D'où $T(n) > F_n \approx 2^{0.694n}$!

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 ?

Tianhe-2 (Université nationale de technologie de la défense, Chine)

Ce supercalculateur chinois occupe la quatrième place du classement des supercalculateurs (juin 2018), avec une puissance de 33.86 pétaflops, soit 33.86×10^{15} opérations/sec.

Complexité exponentielle

$33.86 \times 10^{15} \approx 33.86 \times 2^{40} \approx 2^{45}$ opérations/sec.

Calcul de F_{200} requiert $\approx 2^{140}$ opérations

$\rightarrow$ 2^{95} secondes pour calculer F_{200} avec Tianhe-2

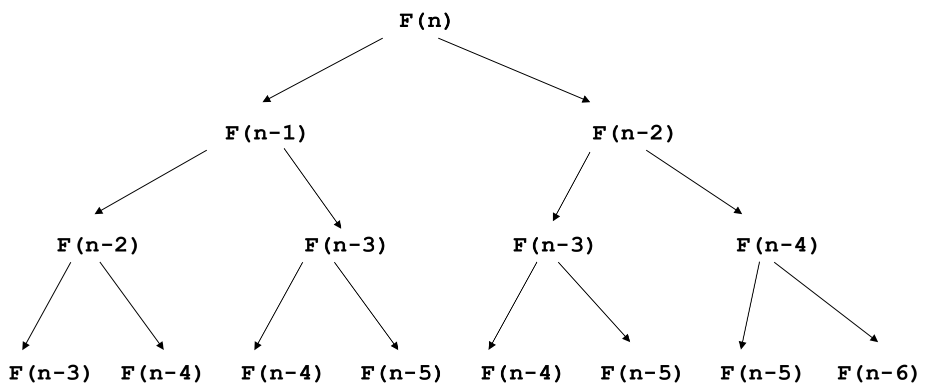
Temps en secondes	Interprétation
2^{10}	17 minutes
2^{20}	12 jours
2^{30}	32 ans
2^{40}	33000 ans

Et la loi de Moore ?

- Selon la loi de Moore, la vitesse des ordinateurs double tous les 18 mois
- $2^{0.694n} \approx (1.6)^n$ additions pour le calcul de F_n
- Soit 1.6 fois plus de temps pour calculer F_{n+1} que F_n
- Donc si l'on peut raisonnablement calculer F_{100} avec la technologie de cette année, l'année prochaine ce sera F_{101}
- Juste un nombre de Fibonacci de plus par an !

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*)

Complexité de `Fib2`

Le contenu de la boucle consiste en une addition, et la boucle est itérée $n - 1$ 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]
```

Nombre d'opérations élémentaires proportionnel à n.

Mais quelle est la constante : *2n, 3n...* ?

La constante dépend :

- de l'unité de temps – minutes, secondes, millisecondes, …
- des spécificités de l'architecture de l'ordinateur.

Elle est *beaucoup* trop complexe à déterminer exactement. De plus, elle importe beaucoup moins que l'énorme fossé entre n et 2^n. On dit donc simplement que **la complexité est *O(n)*.**

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.

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 → quadratique en n.

Complexité et notations de Landau

Polynomial vs. exponentiel

Les complexités comme

n, n^2, n^3, sont *polynomiales*.

Les complexités comme

2^n, e^n, $2^{\sqrt{n}}$ sont *exponentielles*.

Ce qu'il faut retenir en gros :

les complexités polynomiales sont raisonnables

les complexités exponentielles ne sont pas raisonnables

C'est la dichotomie la plus fondamentale en algorithmique.

Complexité d'un algorithme

La complexité (temporelle) d'un algorithme est une évaluation du nombre d'instructions élémentaires pour une exécution de l'algorithme.

Elle est exprimée en fonction de la taille de codage des paramètres de l'algorithme, et en utilisant les notations de Landau (ordres de grandeur).

Complexité pire cas : on évalue le nombre d'instructions dans le pire des cas (borne supérieure) ;

Complexité meilleur des cas : on évalue le nombre d'instructions dans le meilleur des cas (borne inférieure) ;

On identifie généralement le complexité d'un algorithme avec son pire cas.

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.

En toute rigueur, l'efficacité d'un algorithme devrait prendre en compte non pas l'instance mais sa représentation fournie en entrée de l'algorithme.

Cependant 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 pire cas : motivations

Evaluer le temps d'exécution d'un algorithme en fonction de la longueur de l'énoncé.

Comparer les performances de différents algorithmes résolvant le même problème.

Evaluer la taille maximale des énoncés qu'un algorithme peut traiter.

Mesure du temps indépendante des machines → on compte le nombre d'instructions.

Evaluation de la complexité

On est souvent incapable de calculer la complexité exacte TA(n) d'un algorithme A.

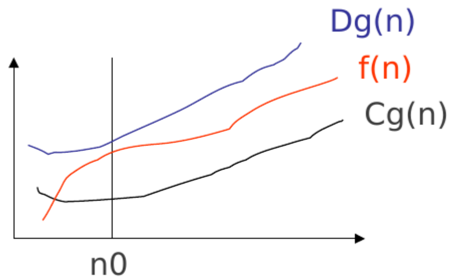
Ce qui est important, c'est le comportement de TA(n) pour les grandes valeurs de n.

On cherche donc à encadrer le taux de croissance de TA(n) pour n suffisamment grand.

Notations de Landau: Θ, O et Ω
relations dans l'ensemble des fonctions de N dans N.

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:
$n > n_0$, $Cg(n) \leq f(n)$.

$f = \Theta(g)$

$f \in \Theta(g)$
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 $n\ log\ n$.

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 -------------------- $N_2+N_3+\ldots+N_n$
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) = $O(n^2)$.

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).

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