Module EA4 – Éléments d'Algorithmique II

Outils pour l'analyse des algorithmes

Dominique Poulalhon
`dominique.poulalhon@irif.fr`

Université Paris Diderot
L2 Informatique & Math-Info
Année universitaire 2019-2020

Points à retenir du cours précédent

algorithme = méthode systématique pour résoudre un problème

Points à retenir du cours précédent

algorithme = méthode systématique pour résoudre un problème

nécessite une preuve de correction : pour chaque entrée, l'algorithme doit terminer en produisant la bonne sortie

Points à retenir du cours précédent

algorithme = méthode systématique pour résoudre un problème

nécessite une preuve de correction : pour chaque entrée, l'algorithme doit terminer en produisant la bonne sortie

il peut exister plusieurs algorithmes pour le même problème

Points à retenir du cours précédent

algorithme = méthode systématique pour résoudre un problème

nécessite une preuve de correction : pour chaque entrée, l'algorithme doit terminer en produisant la bonne sortie

il peut exister plusieurs algorithmes pour le même problème

pour les comparer, il faut étudier leur complexité en temps et en espace

Exemple : comment calculer le 1 000 000e terme de la suite de Fibonacci ?

suite définie par : $F_0 = 0, \quad F_1 = 1 \quad \text{et} \quad \forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

Exemple : comment calculer le 1 000 000e terme de la suite de Fibonacci ?

suite définie par : $F_0 = 0, \quad F_1 = 1 \quad \text{et} \quad \forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

Préambule : à quel point est-ce gros ? 0, 1, 1, 2, 3, 5, 8, 13...

Exemple : comment calculer le 1 000 000e terme de la suite de Fibonacci ?

suite définie par : $F_0 = 0$, $F_1 = 1$ et $\forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

Préambule : à quel point est-ce gros ? 0, 1, 1, 2, 3, 5, 8, 13...

- la suite (F_n) est positive, (donc) croissante,

Exemple : comment calculer le 1 000 000e terme de la suite de Fibonacci ?

suite définie par : $F_0 = 0, \quad F_1 = 1 \quad \text{et} \quad \forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

Préambule : à quel point est-ce gros ? 0, 1, 1, 2, 3, 5, 8, 13...

- la suite (F_n) est positive, (donc) croissante,
- donc la relation de récurrence implique : $\forall n \geqslant 2,\ F_n \geqslant 2F_{n-2}$

Exemple : comment calculer le 1 000 000e terme de la suite de Fibonacci ?

suite définie par : $F_0 = 0, \quad F_1 = 1 \quad \text{et} \quad \forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

Préambule : à quel point est-ce gros ? 0, 1, 1, 2, 3, 5, 8, 13...

- la suite (F_n) est positive, (donc) croissante,
- donc la relation de récurrence implique : $\forall n \geqslant 2,\ F_n \geqslant 2F_{n-2}$
- (or $F_6 = 8 = 2^{6/2}$ et $F_7 = 13 \geqslant 2^{7/2}$) donc $\forall n \geqslant 6,\ F_n \geqslant 2^{n/2}$

Exemple : comment calculer le 1 000 000e terme de la suite de Fibonacci ?

suite définie par : $F_0 = 0, \quad F_1 = 1 \quad$ et $\quad \forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

Préambule : à quel point est-ce gros ? 0, 1, 1, 2, 3, 5, 8, 13...

- la suite (F_n) est positive, (donc) croissante,
- donc la relation de récurrence implique : $\forall n \geqslant 2,\ F_n \geqslant 2F_{n-2}$
- (or $F_6 = 8 = 2^{6/2}$ et $F_7 = 13 \geqslant 2^{7/2}$) donc $\forall n \geqslant 6,\ F_n \geqslant 2^{n/2}$
- donc $\forall n \geqslant 6,\ F_n$ a au moins $\frac{n}{2}$ chiffres (en binaire)

Exemple : comment calculer le 1 000 000e terme de la suite de Fibonacci ?

suite définie par : $F_0 = 0$, $F_1 = 1$ et $\forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

Préambule : à quel point est-ce gros ? 0, 1, 1, 2, 3, 5, 8, 13...

- la suite (F_n) est positive, (donc) croissante,
- donc la relation de récurrence implique : $\forall n \geqslant 2,\ F_n \geqslant 2F_{n-2}$
- (or $F_6 = 8 = 2^{6/2}$ et $F_7 = 13 \geqslant 2^{7/2}$) donc $\forall n \geqslant 6,\ F_n \geqslant 2^{n/2}$
- donc $\forall n \geqslant 6,\ F_n$ a au moins $\frac{n}{2}$ chiffres (en binaire)
- similairement, $F_n \leqslant 2^n$, donc a au plus n chiffres (en binaire)

Exemple : comment calculer le 1 000 000e terme de la suite de Fibonacci ?

suite définie par : $F_0 = 0, \quad F_1 = 1 \quad$ et $\quad \forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

Préambule : à quel point est-ce gros ? 0, 1, 1, 2, 3, 5, 8, 13...

- la suite (F_n) est positive, (donc) croissante,
- donc la relation de récurrence implique : $\forall n \geqslant 2,\ F_n \geqslant 2F_{n-2}$
- (or $F_6 = 8 = 2^{6/2}$ et $F_7 = 13 \geqslant 2^{7/2}$) donc $\forall n \geqslant 6,\ F_n \geqslant 2^{n/2}$
- donc $\forall n \geqslant 6,\ F_n$ a au moins $\frac{n}{2}$ chiffres (en binaire)
- similairement, $F_n \leqslant 2^n$, donc a au plus n chiffres (en binaire)

Donc $F_{1\,000\,000}$ a entre 500 000 et 1 000 000 chiffres en binaire, soit (environ) entre 150 000 et 300 000 chiffres en décimal

Exemple : comment calculer le 1 000 000e terme de la suite de Fibonacci ?

suite définie par : $F_0 = 0, \quad F_1 = 1 \quad$ et $\quad \forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

Préambule : à quel point est-ce gros ? 0, 1, 1, 2, 3, 5, 8, 13...

- la suite (F_n) est positive, (donc) croissante,
- donc la relation de récurrence implique : $\forall n \geqslant 2,\ F_n \geqslant 2F_{n-2}$
- (or $F_6 = 8 = 2^{6/2}$ et $F_7 = 13 \geqslant 2^{7/2}$) donc $\forall n \geqslant 6,\ F_n \geqslant 2^{n/2}$
- donc $\forall n \geqslant 6,\ F_n$ a au moins $\frac{n}{2}$ chiffres (en binaire)
- similairement, $F_n \leqslant 2^n$, donc a au plus n chiffres (en binaire)

Donc $F_{1\,000\,000}$ a entre 500 000 et 1 000 000 chiffres en binaire, soit (environ) entre 150 000 et 300 000 chiffres en décimal

Plus précisément, on peut montrer (admis) :

$$F_n \sim \varphi^n, \text{ avec } \varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$$

Exemple : comment calculer le 1 000 000e terme de la suite de Fibonacci ?

suite définie par : $F_0 = 0, \quad F_1 = 1 \quad$ et $\quad \forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

Exemple : comment calculer le 1 000 000^{e} terme de la suite de Fibonacci ?

suite définie par : $F_0 = 0, \quad F_1 = 1 \quad$ et $\quad \forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

1. utiliser directement la définition par récurrence

```
def fibo_1(n) :
  if n <= 0 : return 0
  if n <= 2 : return 1
  return fibo_1(n-1) + fibo_1(n-2)
```

Exemple : comment calculer le 1 000 000e terme de la suite de Fibonacci ?

suite définie par : $F_0 = 0, \quad F_1 = 1 \quad$ et $\quad \forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

1. utiliser directement la définition par récurrence

```
def fibo_1(n) :
  if n <= 0 : return 0
  if n <= 2 : return 1
  return fibo_1(n-1) + fibo_1(n-2)
```

Preuve de terminaison : par récurrence (forte)

- cas de base : si $n \leqslant 2$, `fibo_1(n)` termine
- hérédité : soit $n \geqslant 2$ t.q. `fibo_1(k)` termine pour tout $k < n$
  alors `fibo_1`$(n-1)$ et `fibo_1`$(n-2)$ terminent, donc `fibo_1(n)` termine

donc `fibo_1(n)` termine pour tout n

Exemple : comment calculer le 1 000 000e terme de la suite de Fibonacci ?

suite définie par : $F_0 = 0, \quad F_1 = 1 \quad$ et $\quad \forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

1. utiliser directement la définition par récurrence

```
def fibo_1(n) :
  if n <= 0 : return 0
  if n <= 2 : return 1
  return fibo_1(n-1) + fibo_1(n-2)
```

Preuve de correction : par récurrence (forte)

- cas de base : si $n \leqslant 2$, `fibo_1`(n) retourne F_n
- hérédité : soit $n \geqslant 2$ t.q. `fibo_1`(k) retourne F_k pour tout $k < n$
  alors `fibo_1`(n) retourne
  `fibo_1`$(n-1)$ + `fibo_1`$(n-2) = F_{n-1} + F_{n-2} = F_n$

donc `fibo_1`(n) retourne F_n pour tout n

Exemple : comment calculer le 1 000 000e terme de la suite de Fibonacci ?

suite définie par : $F_0 = 0, \quad F_1 = 1 \quad \text{et} \quad \forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

1. utiliser directement la définition par récurrence

```
def fibo_1(n) :
  if n <= 0 : return 0
  if n <= 2 : return 1
  return fibo_1(n-1) + fibo_1(n-2)
```

(gros) inconvénient : recalcul permanent de valeurs déjà calculées

Exemple : comment calculer le 1 000 000e terme de la suite de Fibonacci ?

suite définie par : $F_0 = 0, \quad F_1 = 1 \quad$ et $\quad \forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

1. utiliser directement la définition par récurrence
2. garder un tableau de toutes les premières valeurs

```
def fibo_2(n) :
  if n <= 0 : return 0
  liste = [0, 1] + [0] * (n-1)
  for i in range(2, n+1) :
    liste[i] = liste[i-1] + liste[i-2]
  return liste[n]
```

(on appelle cette technique « *programmation dynamique* »)

Exemple : comment calculer le 1 000 000e terme de la suite de Fibonacci ?

suite définie par : $F_0 = 0, \quad F_1 = 1 \quad$ et $\quad \forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

1. utiliser directement la définition par récurrence
2. garder un tableau de toutes les premières valeurs
3. garder seulement les deux dernières valeurs

```
def fibo_3(n) :
  if n <= 0 : return 0
  previous, last = 0, 1
  for i in range(2, n+1) :
    previous, last = last, previous + last
  return last
```

Exemple : comment calculer le 1 000 000e terme de la suite de Fibonacci ?

suite définie par : $F_0 = 0, \quad F_1 = 1 \quad$ et $\quad \forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

1. utiliser directement la définition par récurrence
2. garder un tableau de toutes les premières valeurs
3. garder seulement les deux dernières valeurs

```
def fibo_3(n) :
  if n <= 0 : return 0
  previous, last = 0, 1
  for i in range(2, n+1) :
    previous, last = last, previous + last
  return last
```

Preuve de terminaison : $n - 1$ tours de boucle

Exemple : comment calculer le 1 000 000ᵉ terme de la suite de Fibonacci ?

suite définie par : $F_0 = 0$, $F_1 = 1$ et $\forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

1. utiliser directement la définition par récurrence
2. garder un tableau de toutes les premières valeurs
3. garder seulement les deux dernières valeurs

```
def fibo_3(n) :
  if n <= 0 : return 0
  previous, last = 0, 1
  for i in range(2, n+1) :
    previous, last = last, previous + last
  return last
```

Preuve de correction : à l'aide de l'invariant :

« *après le tour de boucle d'indice* i, `previous` $= F_{i-1}$ *et* `last` $= F_i$ »

Exemple : comment calculer le 1 000 000e terme de la suite de Fibonacci ?

suite définie par : $F_0 = 0, \quad F_1 = 1 \quad \text{et} \quad \forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

1. utiliser directement la définition par récurrence
2. garder un tableau de toutes les premières valeurs
3. garder seulement les deux dernières valeurs
4. utiliser :

$$\forall n \geqslant 1, \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

```
def fibo_4(n) :
  M = puissance_matrice_2_2 ([ [1, 1], [1, 0] ], n-1)
  return M[0][0]
```

Exemple : comment calculer le 1 000 000e terme de la suite de Fibonacci ?

suite définie par : $F_0 = 0, \quad F_1 = 1 \quad$ et $\quad \forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

1. utiliser directement la définition par récurrence
2. garder un tableau de toutes les premières valeurs
3. garder seulement les deux dernières valeurs
4. utiliser :

$$\forall n \geqslant 1, \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

Exemple : comment calculer le 1 000 000e terme de la suite de Fibonacci ?

suite définie par : $F_0 = 0, \quad F_1 = 1 \quad$ et $\quad \forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

1. utiliser directement la définition par récurrence
2. garder un tableau de toutes les premières valeurs
3. garder seulement les deux dernières valeurs
4. utiliser :

$$\forall n \geqslant 1, \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

Lemme : les 4 algorithmes sont corrects

Exemple : comment calculer le 1 000 000e terme de la suite de Fibonacci ?

suite définie par : $F_0 = 0, \quad F_1 = 1 \quad$ et $\quad \forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$

1. utiliser directement la définition par récurrence
2. garder un tableau de toutes les premières valeurs
3. garder seulement les deux dernières valeurs
4. utiliser :

$$\forall n \geqslant 1, \quad \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

Lemme : les 4 algorithmes sont corrects

Quelle est la meilleure méthode ?

Complexité en espace

= quantité de mémoire nécessaire pour effectuer le calcul

Complexité en espace

= quantité de mémoire nécessaire pour effectuer le calcul

mémoire incompressible : nécessaire pour stocker les données et le résultat

mémoire auxiliaire : pour les calculs intermédiaires

Complexité en espace

= quantité de mémoire nécessaire pour effectuer le calcul

mémoire incompressible : nécessaire pour stocker les données et le résultat

mémoire auxiliaire : pour les calculs intermédiaires

pour comparer entre eux des algorithmes résolvant le même problème, seule la mémoire auxiliaire est pertinente

Complexité en espace – exemple

```
def fibo_2(n) :
  if n <= 0 : return 0
  liste = [0, 1] + [0] * (n-1)
  for i in range(2, n+1) :
    liste[i] = liste[i-1] + liste[i-2]
  return liste[n]
```

utilise un tableau de n (grands) entiers pour en calculer un seul (plus un (petit [1]) entier comme indice de boucle)

1. petit = valeur de l'ordre de n *vs* grand = de l'ordre de n chiffres

Complexité en espace – exemple

```
def fibo_2(n) :
  if n <= 0 : return 0
  liste = [0, 1] + [0] * (n-1)
  for i in range(2, n+1) :
    liste[i] = liste[i-1] + liste[i-2]
  return liste[n]
```

utilise un tableau de n (grands) entiers pour en calculer un seul (plus un (petit [1]) entier comme indice de boucle)

donc la place nécessaire est (à peu près) la somme des tailles des n valeurs calculées, donc (supérieure à) $\sum_{i=0}^{n} \frac{i}{2} = \frac{n(n+1)}{4}$ bits

(soit à peu près 250 000 000 000 pour $n = 1\,000\,000$, c'est-à-dire 30 Go)

1. petit = valeur de l'ordre de n *vs* grand = de l'ordre de n chiffres

Complexité en espace – exemple

```
def fibo_3(n) :
  if n <= 0 : return 0
  previous, last = 0, 1
  for i in range(2, n+1) :
    previous, last = last, previous + last
  return last
```

utilise seulement une variable auxiliaire de type (grand) entier
(plus un (petit) entier comme indice de boucle)

Complexité en temps

= temps nécessaire pour mener le calcul à son terme

Complexité en temps

= temps nécessaire pour mener le calcul à son terme

plus difficile à quantifier précisément, essentiellement car ce temps dépend de la machine utilisée

Complexité en temps

= temps nécessaire pour mener le calcul à son terme

plus difficile à quantifier précisément, essentiellement car ce temps dépend de la machine utilisée

convention : on estime ce temps par le nombre d'opérations élémentaires effectuées : sur une machine donnée, le temps d'exécution sera (à peu près) proportionnel à ce nombre

opération élémentaire = opération dont le temps d'exécution peut être considéré comme constant

exemple : affectation, comparaison, opération arithmétique (sur des nombres de taille bornée)...

Complexité et ordres de grandeur

Nombre de cycles effectués par un processeur monocœur à 1 GHz :

en 1 seconde	10^9
en 1 heure	$3 \cdot 10^{12}$
en 1 jour	$9 \cdot 10^{13}$
en 1 an	$3 \cdot 10^{16}$

(pour un quadricœur à 2.5GHz, rajouter juste un zéro)

Complexité et ordres de grandeur

n	10	100	10^3	10^6	10^9	10^{12}
$\log_2 n$	4	7	10	20	30	40
$10n$	100	10^3	10^4	10^7	10^{10}	10^{13}
$n \log_2 n$	34	665	10^4	$2 \cdot 10^7$	$3 \cdot 10^{10}$	$4 \cdot 10^{13}$
n^2	100	10^4	10^6	10^{12}	10^{18}	10^{24}
n^3	10^3	10^6	10^9	10^{18}	10^{27}	10^{36}
2^n	10^3	10^{30}	10^{301}	...	...	...

(rappel : en 1 an, un processeur à 1 GHz effectue $3 \cdot 10^{16}$ cycles)

Complexité et ordres de grandeur

n	10	100	10^3	10^6	10^9	10^{12}
$\log_2 n$	4	7	10	20	30	40
$10n$	100	10^3	10^4	10^7	10^{10}	10^{13}
$n \log_2 n$	34	665	10^4	$2 \cdot 10^7$	$3 \cdot 10^{10}$	$4 \cdot 10^{13}$
n^2	100	10^4	10^6	10^{12}	10^{18}	10^{24}
n^3	10^3	10^6	10^9	10^{18}	10^{27}	10^{36}
2^n	10^3	10^{30}	10^{301}	...	...	...

(rappel : en 1 an, un processeur à 3.2 GHz effectue 10^{17} cycles)

Complexité et ordres de grandeur

n	10	100	10^3	10^6	10^9	10^{12}
$\log_2 n$	4	7	10	20	30	40
$10n$	100	10^3	10^4	10^7	10^{10}	10^{13}
$n \log_2 n$	34	665	10^4	$2 \cdot 10^7$	$3 \cdot 10^{10}$	$4 \cdot 10^{13}$
n^2	100	10^4	10^6	10^{12}	10^{18}	10^{24}
n^3	10^3	10^6	10^9	10^{18}	10^{27}	10^{36}
2^n	10^3	10^{30}	10^{301}	...	...	...

(rappel : en 1 an, un processeur à 3.2 GHz effectue 10^{17} cycles)

Notations utilisées

$f \in O(g) \iff$ f « ne grandit pas plus vite que » g (« *grand O* »)

$f \in \Omega(g) \iff$ f « grandit au moins aussi vite que » g (« *Oméga* »)

$f \in \Theta(g) \iff$ f et g « grandissent à la même vitesse » (« *Théta* »)

Notations utilisées

$f \in O(g) \iff$ f « ne grandit pas plus vite que » g (« *grand O* »)

« (pour $m \leqslant n$ assez grands) si $g(n) \leqslant \alpha g(m)$, alors $f(n) \leqslant \alpha f(m)$ »

$f \in \Omega(g) \iff$ f « grandit au moins aussi vite que » g (« *Oméga* »)

$f \in \Theta(g) \iff$ f et g « grandissent à la même vitesse » (« *Théta* »)

Notations utilisées

$f \in O(g) \iff f$ « ne grandit pas plus vite que » g (« *grand O* »)

« (pour $m \leqslant n$ assez grands) si $g(n) \leqslant \alpha g(m)$, alors $f(n) \leqslant \alpha f(m)$ »

$f \in \Omega(g) \iff f$ « grandit au moins aussi vite que » g (« *Oméga* »)

« (pour $m \leqslant n$ assez grands) si $g(n) \geqslant \alpha g(m)$, alors $f(n) \geqslant \alpha f(m)$ »

$f \in \Theta(g) \iff f$ et g « grandissent à la même vitesse » (« *Théta* »)

Définition (grand O, grand Oméga, grand Théta)

Soit f et g deux fonctions de $\mathbb{N}$ dans $\mathbb{N}$. On dit que :

- $f \in O(g)$ (ou $f(n) \in O(g(n))$, ou $f(n) = O(g(n))$) si :

$$\exists c > 0,\ \exists n_0 \in \mathbb{N},\quad \forall n \geqslant n_0,\quad f(n) \leqslant c \cdot g(n)$$

- $f \in \Omega(g)$ (ou $f(n) \in \Omega(g(n))$) si :

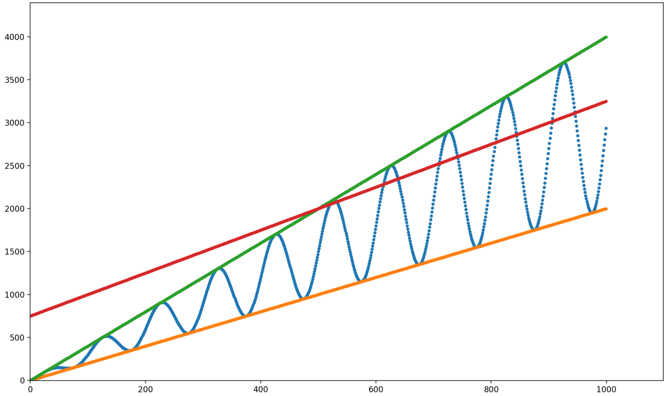
$$\exists c > 0,\ \exists n_0 \in \mathbb{N},\quad \forall n \geqslant n_0,\quad f(n) \geqslant c \cdot g(n)$$

- $f \in \Theta(g)$ (ou $f(n) \in \Theta(g(n))$) si :

$$f \in O(g) \quad \text{et} \quad f \in \Omega(g)$$

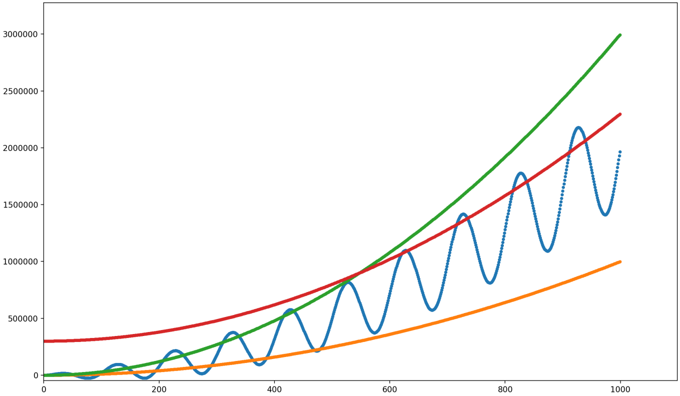
autrement dit :

$$\exists c_1, c_2 > 0,\ \exists n_0 \in \mathbb{N},\quad \forall n \geqslant n_0,\quad c_1 \cdot g(n) \leqslant f(n) \leqslant c_2 \cdot g(n)$$



Fonctions appartenant à $\Theta(n)$

Exemples



Fonctions appartenant à $\Theta(n^2)$

Complexité des calculs de F_n

utilisation naïve de la récurrence $\Longrightarrow \Theta(\varphi^n)$ additions

```
def fibo_1(n) :
  if n <= 2 : return 1
  return fibo_1(n-1) + fibo_1(n-2)
```

Complexité des calculs de F_n

utilisation naïve de la récurrence $\Longrightarrow$ $\Theta(\varphi^n)$ additions

calcul itératif des n premières valeurs $\Longrightarrow$ $\Theta(n)$ additions

```
def fibo_3(n) :
  previous, last = 1, 1
  for i in range(2, n+1) :
    previous, last = last, previous + last
  return last
```

Complexité de l'exponentiation binaire

```
def puissance(a, n) :
  if n == 0 : return 1
  tmp = puissance(a, n//2)
  carre = tmp * tmp             # une multiplication
  if n%2 == 0 : return carre
  else : return a * carre       # une multiplication
```

Complexité

$\Theta(\log_2 n)$ multiplications de la forme $a^k \cdot a^\ell$, $k \in \{1, \ell\}$

Complexité de l'exponentiation binaire

Complexité

$\Theta(\log_2 n)$ multiplications de la forme $a^k \cdot a^\ell$

Complexité de l'exponentiation binaire

Complexité

$\Theta(\log_2 n)$ multiplications de la forme $a^k \cdot a^\ell$

si ces multiplications ont un coût constant, *i.e.* si les opérandes ont une taille constante, complexité en $\Theta(\log_2 n)$

c'est le cas avec l'arithmétique modulaire ou l'arithmétique flottante utilisées usuellement : tous les nombres sont codés sur exactement 32 (ou 64) bits, donc le coût d'une multiplication est constant

Complexité de l'exponentiation binaire

Complexité

$\Theta(\log_2 n)$ multiplications de la forme $a^k \cdot a^\ell$

si ces multiplications ont un coût constant, *i.e.* si les opérandes ont une taille constante, complexité en $\Theta(\log_2 n)$

sinon il faut tenir compte du coût de ces multiplications ; par exemple en arithmétique exacte sur des entiers :

valeur	taille (en bits)	coût du calcul naïf du carré
a	$\log_2 a$	$\Theta((\log_2 a)^2)$
a^k	$k \cdot \log_2 a$	$\Theta(k^2 \cdot (\log_2 a)^2)$

Complexité des calculs de F_n

utilisation naïve de la récurrence $\Longrightarrow$ $\Theta(\varphi^n)$ additions

calcul itératif des n premières valeurs $\Longrightarrow$ $\Theta(n)$ additions

calcul de $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$ $\Longrightarrow$ $\Theta(\log_2 n)$ multiplications...
de matrices 2×2
(chacune impliquant 4 additions et 8 multiplications d'entiers)

Complexité des calculs de F_n

utilisation naïve de la récurrence $\Longrightarrow$ $\Theta(\varphi^n)$ additions

calcul itératif des n premières valeurs $\Longrightarrow$ $\Theta(n)$ additions

calcul de $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$ $\Longrightarrow$ $\Theta(\log_2 n)$ multiplications...
de matrices 2×2
(chacune impliquant 4 additions et 8 multiplications d'entiers)

MAIS... comme $F_n \in \Theta(\varphi^n)$, les opérations arithmétiques se font sur des entiers de taille $\Theta(n)$ (c'est-à-dire de $\Theta(n)$ chiffres)

$\Longrightarrow$ additions en $\Theta(n)$ opérations élémentaires,
multiplications en $O(n^2)$ (coût de l'algo naïf)