Programmation et algorithmique

IN101
ENSTA Paris - TC 1ère année

François Pessaux

U2IS

2022-2023

prenom.nom@ensta-paristech.fr

Complexité

Court vs long

- Addition de 2 entiers (diapos 28 et 29 première séance).

```
int addition (int x, int y) {
  return (x + y) ;
}
```

```
int addition (int x, int y) {
  int res = x ;
  for (int i = 0; i < y; i++)
    res = res + 1 ;
  return res ;
}
```

- L'un semble plus court que l'autre.

Comparaison

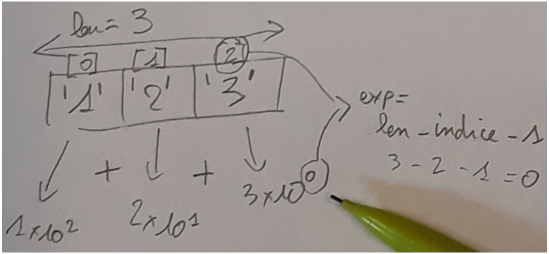
```
int addition (int x, int y) {
  return (x + y) ;
}
```

```
int addition (int x, int y) {
  int res = x ;
  for (int i = 0; i < y; i++)
    res = res + 1 ;
  return res ;
}
```

- Pour x et y donnés :
  - Algorithme de gauche : 1 instruction exécutée.
  - Algorithme de droite : plus de y instructions exécutées.
- L'un est plus court que l'autre.
- Surtout : l'un est plus efficace que l'autre.
- $\nearrow$ instructions $\Rightarrow$ $\nearrow$ temps.
- Efficacité temporelle.

Chaîne vers entier (1/2)

- Conversion chaîne vers entier : `tonum ("123")` → 123.
- Chaîne = tableau de caractères.
- Codes ASCII (entier) des caractères sont ordonnés.



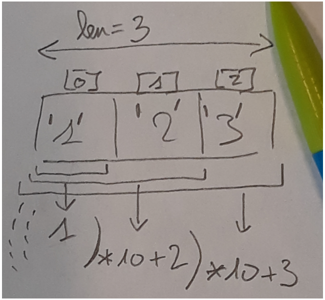

```
#include <math.h>

/* char = entier 8 bits dont la valeur est le code ASCII du caractère. */
unsigned int tonum (char *str) {
  unsigned int res = 0 ;
  int slen = strlen (str) ;

  for (int i = 0; i < slen; i++)
    res = res + (str[i] - '0') * pow (10, (slen - i - 1)) ;
  return res ;
}
```

Chaîne vers entier (2/2)

- Conversion chaîne vers entier : `tonum ("123")` → 123.




```
/* char = entier 8 bits dont la valeur est le code ASCII du caractère. */
unsigned int tonum (char *str) {
  unsigned int res = 0 ;
  for (int i = 0; str[i] != '\0'; i++)
    res = res * 10 + (str[i] - '0') ;
  return res ;
}
```

⇒ RIP `pow` ☺

Court vs court

```c
unsigned int tonum (char *str) {
  unsigned int res = 0 ;
  int slen = strlen (str) ;
  for (int i = 0; i < slen; i++)
    res = res + (str[i] - '0') *
          pow (10, (slen - i - 1)) ;
  return res ;
}
```

```c
unsigned int tonum (char *str) {
  unsigned int res = 0 ;
  for (int i = 0; str[i] != '\0'; i++)
    res = res * 10 + (str[i] - '0') ;
  return res ;
}
```

- Longueurs quasiment identiques.
- Algorithme de gauche : utilisation de `pow` (puissance).
- `pow` est une fonction : plusieurs instructions (boucle en version naïve) .
- `pow` est une fonction sur les flottants.
- Appel pour chaque exposant.
- ⇒ Algorithme de gauche plus coûteux en temps.

Qu'est-ce que la complexité ?

- Complexité d'un algorithme : mesure de son efficacité intrinsèque :
  - en fonction de la taille des données à traiter,
  - asymptotiquement,
  - dans le pire cas ou en moyenne.
- ⇒ Notion d'efficacité indépendante de la vitesse de la machine.
- Deux formes de complexité :
  - Temporelle : s'intéresse au temps passé dans dans l'exécution.
  - Spatiale : s'intéresse à l'espace mémoire nécessaire lors de l'exécution.
- On s'intéresse le plus souvent à la complexité temporelle.
- Pratiquement, pour une entrée de taille n :
  - On compte le nombre d'opérations « *de base* » nécessaires.
  - On regarde comment ce nombre évolue asymptotiquement.

Pourquoi s'intéresser à la complexité ?

- Imaginez. . .
  - Une recherche *Google* prenant 5 minutes.
  - Les simulations météo de la veille terminées le lendemain.
  - *Counter Strike* à 10 images/seconde.
  - Acheter de la RAM à chaque mise à jour de votre application favorite.

- Entre plusieurs algorithmes, on va préférer le plus efficace.

Un premier exemple (1)

```
unsigned int sumint (unsigned int n) {
  unsigned int res = 0 ;
  for (unsigned int i = 1; i <= n; i++)
    res = res + i ;
  return res ;
}
```

- Simple boucle `while`
  - Taille de l'entrée : n.
  - La boucle va « *tourner* » n fois.
  - La fonction va faire n additions ...+ n pour l'incrémentation de la boucle, + n tests $\Rightarrow 3\,n$ instructions.
  - En général, on ne s'intéresse qu'aux opérations calculant le résultat.
  - Complexité linéaire en la taille de l'entrée.

Un premier exemple (2)

- Un peu de réflexion :

$$\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & n=5 \\ + & + & + & + & + & + \\ n=5 & 4 & 3 & 2 & 1 & 0 \\ \hline \end{array}$$

$$\underbrace{n + n + n + n + n + n}_{n+1} = 2 * \overbrace{(1 + 2 + \ldots + 5)}^{\displaystyle \sum_{n=0}^{5} n \; =}$$

$$= n * (n + 1)$$

- Donc, $\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \rightarrow$
  - 1 addition +
  - 1 multiplication +
  - 1 division.
  - Et même pas de tests ou autres incrémentations autour !
  - $\Rightarrow$ Calcul en temps constant.

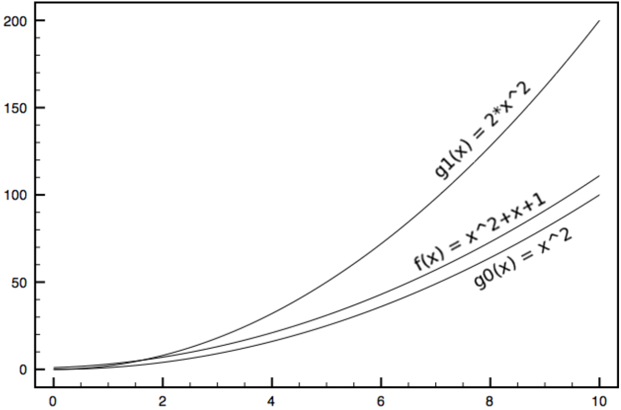
Notations de complexité

- Évaluation de la complexité, de l'efficacité d'un algorithme :
  - $\Rightarrow$ Encadrer le nombre d'opérations qu'il fait.
  - $\Rightarrow$ Borne supérieure et borne inférieure.
  - Lorsque la taille de l'entrée tend vers $+\infty$.
- $O(g)$: ensemble des fonctions f telles que $0 \leq f(x) \leq k \times g(x)$
  - Avec $k \in \mathbb{R}_+^*$
  - Et $\exists x_0 \in \mathbb{N}^*, \forall x \geq x_0$ ($\rightsquigarrow$ comportement asymptotique).
- $\theta(g)$: ensemble des fonctions f telles que $k_1 \times g(x) \leq f(x) \leq k_2 \times g(x)$
  - Avec $k_1, k_2 \in \mathbb{R}_+^*$
  - Et $\exists x_0 \in \mathbb{N}^*, \forall x \geq x_0$ ($\rightsquigarrow$ comportement asymptotique).
- Notation malheureuse : on écrit $f = O(g)$ au lieu de $f \in O(g)$.

Exemple : θ

$\theta(g)$: ensemble des fonctions f telles que $k_1 \times g(x) \leq f(x) \leq k_2 \times g(x)$

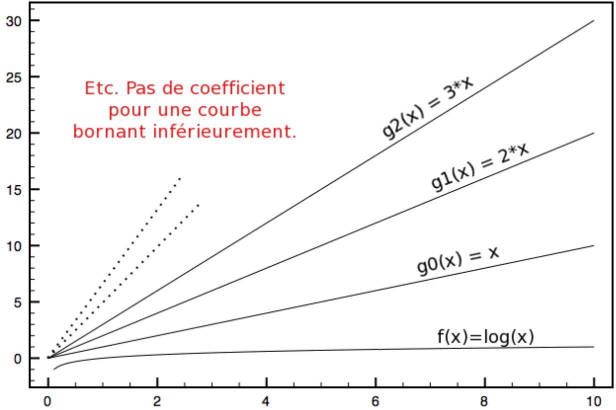
- $f(x) = x^2 + x + 1 \ \in \ \theta(n^2)$



Exemple : O

$O(g)$: ensemble des fonctions f telles que $0 \leq f(x) \leq k \times g(x)$

- $f(x) = log(x) \in O(x)$



Classement de complexités

$$\log(n) \ll \sqrt{n} \ll n \ll n\log(n) \ll n^2 \ll n^3 \ll 2^n \ll exp(n) \ll n! \ll n^n \ll 2^{2^n}$$

- Définitions : complexité
  - linéaire : $f(x) = O(x) \rightarrow$ réalisable
  - quadratique : $f(x) = O(x^2) \rightarrow$ réalisable
  - polynomiale : $\exists k > 0,\ f(x) = O(x^k) \rightarrow$ souvent réalisable
  - exponentielle : $\exists b > 1,\ f(x) = O(b^x) \rightarrow$ en général irréalisable
  - doublement exponentielle, par exemple : $f(x) = O(2^{2^x})$.
  - sous-exponentielle, par exemple : $f(x) = O(2^{\sqrt{x}})$.
- Quelques exemples :
  - Algorithme de tri par tas : $O(n\ log\ n)$.
  - Calcul de la matrice d'accessibilité d'un graphe : $O(n^3)$.

Quelques ordres de grandeur

- (2010) - **AMD FX-8150 (8-core) @ 3.6 GHz** : $\approx 1.0\ 10^{11}$ instr/s.
- (2011) - **Intel Core i7 2600K@ 3.6 GHz** : $\approx 1.2\ 10^{11}$ instr/s.
- (2016) - **Intel Core i7 6950X @ 3 GHz** : $\approx 3.2\ 10^{11}$ instr/s.
- (2017) - **AMD Ryzen 7 1800X@ 3.6 GHz** : $\approx 3.0\ 10^{11}$ instr/s.
- (2019) - **AMD Ryzen 9 3950X@ 4.6 GHz** : $\approx 7.5\ 10^{11}$ instr/s.

- PC standard, 2^{40} (10^{12}) instructions : pas un problème.
- Records actuels : un peu au-delà de 2^{60} (10^{18}) op. binaires (réalisable par des gens « motivés » → NSA, Folding@home ...)
- En cryptographie, actuellement 2^{80} (10^{24}) opérations binaires sont considérées comme inatteignables ...aujourd'hui ...
- ⇒ Clefs de 128 bits sûres pour quelques dizaines d'années.

Fibonnaci

- On cherche à calculer le $n^{\text{ième}}$ nombre de la suite de Fibonacci
- $Fib_0 = 0$ et $Fib_1 = 1$
- $Fib_n = Fib_{n-1} + Fib_{n-2}$ pour $n > 1$
- $\rightarrow$ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233 ...

Calcul direct

- Récurrence linéaire à coefficients constants.
- $\Rightarrow$ Utilisation de résultats d'algèbre linéaire.
- Polynôme caractéristique : $x^2 - x - 1$
- On recherche ses racines :
  - Racine 1 : $\varphi = \frac{1}{2}(1 + \sqrt{5})$
  - Racine 2 : $\overline{\varphi} = \frac{1}{2}(1 - \sqrt{5})$
- $\Rightarrow\ Fib_n = \frac{1}{\sqrt{5}}(\varphi^n - \overline{\varphi}^n)$
- $\overline{\varphi} \simeq -0.62$
  - $\Rightarrow$ Pour $n > 1$ asymptotiquement, Fib_n = entier le plus proche de $\frac{\varphi^n}{\sqrt{5}}$.

```
#include <math.h>

unsigned long int fib (unsigned int n) {
  /* L'utilisation des float donne parfois un résultat erroné par arrondis.
     Ex : fib (71) -> 308061521170130 au lieu de 308061521170129. */
  long double phi = 1. / 2. * (1 + sqrt (5)) ;
  return (round (powl (phi, n) / sqrt (5))) ;
}
```

Version récursive naïve

$Fib_0 = 0,\ Fib_1 = 1,\ Fib_n = Fib_{n-1} + Fib_{n-2}$

```
unsigned int fib (unsigned int n) {
  if (n < 2) return (n) ;
  else return (fib (n - 1) + fib (n - 2)) ;
}
```

- Calcul de la complexité en nombre d'appels à `fib`.
- Se réduit à une somme de Fib_0 et de Fib_1.

$$
\begin{aligned}
Fib_4 &= Fib_3 + Fib_2 \\
&= (Fib_2 + Fib_1) + (Fib_1 + Fib_0) \\
&= (Fib_1 + Fib_0) + Fib_1 + Fib_1 + Fib_0
\end{aligned}
$$

- Nombre d'appels récursifs : $2 \times Fib_{n+1} - 1$.
- $Fib_n = \frac{\varphi^n}{\sqrt{5}} \Rightarrow$ complexité $= \theta(Fib_n) = \theta(\varphi^n)$: exponentielle.

Version itérative simple

$Fib_0 = 0,\ Fib_1 = 1,\ Fib_n = Fib_{n-1} + Fib_{n-2}$

- Calcul de plusieurs fois la même valeur Fib_i : à éviter.
- $\Rightarrow$ Utilisation d'un tableau pour mémoriser les Fib_i (*« mémoïzation »*).

```
unsigned int fib (unsigned int n) {
  unsigned int i ;
  /* Allocation dynamique. */
  unsigned int *fibs = malloc (sizeof (unsigned int) * (n + 1))
      ;
  fibs[0] = 0 ;
  fibs[1] = 1 ;
  for (i = 2; i < n + 1; i++) {
    fibs[i] = fibs[i - 1] + fibs[i - 2] ;
  }
  return (fibs[n]) ;
}
```

- Complexité temporelle en $\theta(n)$.
  - Accès case de tableau : temps constant.
- **Mais** complexité spatiale aussi en $\theta(n)$.

Version itérative plus rusée

$Fib_0 = 0,\ Fib_1 = 1,\ Fib_n = Fib_{n-1} + Fib_{n-2}$

- Seules 2 valeurs sont lues dans le tableau à chaque itération.
- `fibs[i - 1]` et `fibs[i - 2]`.
- $\Rightarrow$ Utiliser seulement 2 variables.

```
unsigned int fib (unsigned int n) {
  unsigned int i ;
  unsigned int fib0 = 0 ;
  unsigned int fib1 = 1 ;
  if (n < 2) return (n) ;
  for (i = 2; i < n + 1; i++) {
    fib1 = fib0 + fib1 ;
    fib0 = fib1 - fib0 ;
  }
  return (fib1) ;
}
```

- Complexité temporelle toujours en $\theta(n)$.
- **Mais** complexité spatiale maintenant en $\theta(1)$.

Version itérative optimale

- Calcul plus efficace et sans nombres flottants (pas de $\sqrt{5}$).
- Pour $n \geq 2$, on écrit *Fib* sous la forme :

$$\begin{array}{rcl} Fib_n & = & 1 \times Fib_{n-1} + 1 \times Fib_{n-2} \\ Fib_{n-1} & = & 1 \times Fib_{n-1} + 0 \times Fib_{n-2} \end{array}$$

- Donc on a :

$$\begin{array}{rcl} \begin{pmatrix} Fib_n \\ Fib_{n-1} \end{pmatrix} & = & \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} Fib_{n-1} \\ Fib_{n-2} \end{pmatrix} \\ & = & \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \times \begin{pmatrix} Fib_1 \\ Fib_0 \end{pmatrix} \end{array}$$

- $\Rightarrow$ Simple problème d'exponentiation matricielle.
- Complexité temporelle : $\theta(\log(n))$.
- Complexité spatiale : $\theta(1)$.

Comparaison des temps de calcul

- Ordinateur d'il y a une dizaine d'années, avec 4 Go de mémoire.
- Machines actuelles permettent d'aller plus loin, mais ne changent pas l'évolution finale.

	n	40	2^{25}	2^{28}	2^{31}
`fibo1`	$\theta(\varphi^n)$	31s	Calcul irréalisable		
`fibo2`	$\theta(n)$	< 1s	18s	`Segmentation fault`	
`fibo3`	$\theta(n)$	< 1s	4s	25s	195s
`fibo4`	$\theta(log(n))$	< 1s	< 1s	< 1s	< 1s

Théorie de la complexité

- But : classifier des problèmes en fonction de la complexité du meilleur algorithme pour les résoudre.
- « *Meilleur* » algorithme, « *pas meilleur connu* » !
- On ne sait pas toujours prévoir la complexité optimale.
- Pour certains problèmes, on peut prouver la complexité optimale :
  - Tri par comparaison : $\theta(n\ log(n))$.