Les divisions entières par 2

Il arrive fréquemment que les équations de récurrences soient du type :

$$c(n) = c\left(\frac{n}{2}\right) + f(n)$$

Théorème

Soit $c(n)$ une fonction croissante de $I\!N$ vers $I\!R$. Si, pour tout n assez grand de la forme $n = 2^p$, $c(n) = \mathcal{O}(n^\alpha(\log n)^\beta)$ avec $\alpha \geq 0$ et $\beta \geq 0$, alors l'égalité reste vraie pour tout n assez grand.

Cela signifie que si on résout l'équation pour des n de la forme 2^p alors la solution sera valable pour n'importe quel n, pourvu qu'il soit grand.

Le tri fusion

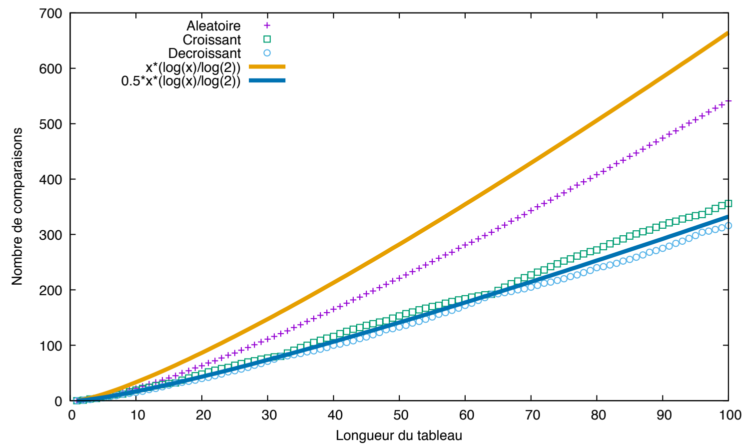
- principe, exemple
- code

```
def merge_sort (t):
    n = len(t)
    if n <= 1:
        # cas de base
        return copy.deepcopy(t)
    else:
        # cas general
        t1 = merge_sort((t[0:((n-1)//2+1)]))
        t2 = merge_sort((t[((n-1)//2+1):n]))
        return merge(t1,t2)
```



```
def merge (t1,t2):
    n1 = len(t1)
    n2 = len(t2)
    t = [ 0 for i in range(0,n1+n2)]
    i = j = k = 0
    while i < n1 and j < n2:
        if cmp(t1[i],t2[j]):
            t[k] = t1[i]
            i = i + 1
        else:
            t[k] = t2[j]
            j = j + 1
        k = k + 1
    while i < n1:
        t[k] = t1[i]
        i = i + 1
        k = k + 1
    while j < n2:
        t[k] = t2[j]
        j = j + 1
        k = k + 1
    return t
```

Tri fusion - analyse



Equations de partition

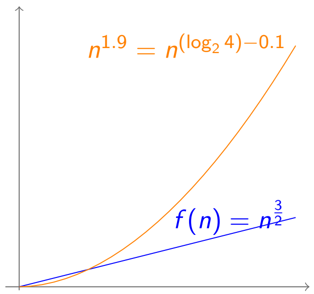
On s'intéresse aux équations de la forme

$$c(n) = a \times c\left(\frac{n}{b}\right) + f(n)$$

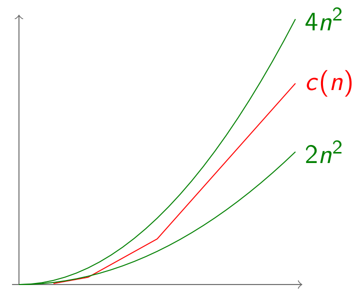
> ! **Suivant le comportement asymptotique de $f(n)$ par rapport à $n^{\log_b a}$, on va pouvoir prédire le comportement asymptotique de $c(n)$.**

Cas 1

$$c(n) = 4c\left(\frac{n}{2}\right) + n^{\frac{3}{2}}, a = 4, b = 2, f(n) = n^{\frac{3}{2}}, \log_b a = 2$$



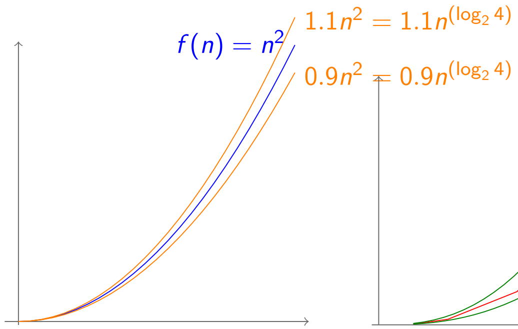
si $f(n) = \mathcal{O}(n^{\log_b a - \varepsilon})$



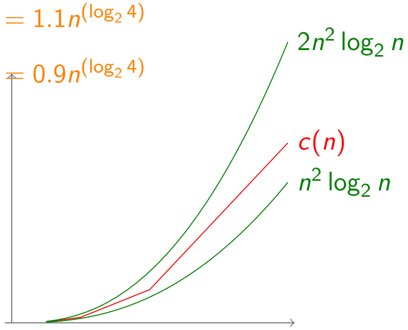
alors $c(n) = \Theta(n^{\log_b a})$

Cas 2

$$c(n) = 4c\left(\frac{n}{2}\right) + n^2, a = 4, b = 2, f(n) = n^2, \log_b a = 2$$



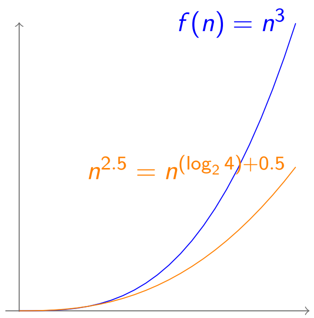
si $f(n) = \Theta(n^{\log_b a})$



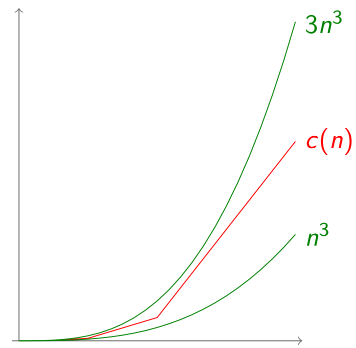
alors $c(n) = \Theta(n^{\log_b a} \log_2 n)$

Cas 3

$$c(n) = 4c\left(\frac{n}{2}\right) + n^3, a = 4, b = 2, f(n) = n^3, \log_b a = 2$$



si $f(n) = \Omega(n^{\log_b a + \varepsilon})$



alors $c(n) = \Theta(f(n))$

Equations de partitions

Théorème général

Soient $a \geq 1$ et $b > 1$ deux constantes, soit $f(n)$ une fonction et soit $c(n)$ définie pour les entiers non négatifs par la récurrence

$$c(n) = a \times c\,(n/b) + f(n),$$

où l'on interprète n/b comme étant $\lfloor \frac{n}{b} \rfloor$ ou $\lceil \frac{n}{b} \rceil$. $c(n)$ peut alors être bornée asymtotiquement de la façon suivante.

1. si $f(n) = \mathcal{O}(n^{\log_b a-\varepsilon})$ pour une certaine constante $\varepsilon > 0$, alors
$$c(n) = \Theta(n^{\log_b a}).$$
2. si $f(n) = \Theta(n^{\log_b a})$, alors
$$c(n) = \Theta(n^{\log_b a} \log_2 n).$$
3. si $f(n) = \Omega(n^{\log_b a+\varepsilon})$ pour une certaine constante $\varepsilon > 0$, et si $a \times f(n/b) \leq k \times f(n)$ pour une certaine constante $k < 1$ et pour n suffisamment grand, alors
$$c(n) = \Theta(f(n)).$$

Complexité des tris

Tri	Nombre d'opérations de comparaison		Complexité
	pire des cas	meilleur des cas	
Bulle	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$	$\Theta(n^2)$
Sélection	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$	$\Theta(n^2)$
Insertion	$\frac{n(n-1)}{2}$	$n-1$	$\mathcal{O}(n^2)$,$\Omega(n)$
Insertion dicho.	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$
Fusion	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$

II Faisons le point

d'un point de vue pratique

- les tris récursifs
- le paradigme diviser pour régner
- les points forts et les faiblesses des différents tris

d'un point de vue théorique

- théorème général qui permet d'avoir facilement une approximation asymptotique des fonctions de complexité qui s'expriment sous la forme d'équations de partition

Rappel : présence d'un élément dans un tableau

```
(* version recursive *)
def contains (t,v):
  n = len(t)
  if n == 0:
    return False
  else
    tt = copy.deepcopy(t[1:n])
    return t[0] == v or contains(tt,v)
```

- expression du nombre de comparaisons

$$c(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 + c(n-1) & \text{sinon} \end{cases}$$

- expression de l'espace mémoire occupé

$$e(n) = \begin{cases} 0 & \text{si } n = 0 \\ n - 1 + e(n-1) & \text{sinon} \end{cases}$$

Cas des équations linéaires d'ordre 1

La solution d'une équation linéaire d'ordre 1 de la forme :

$$c(n) = a \times c(n-1) + f(n)$$

est :

$$c(n) = a^n \left(c(0) + \sum_{i=1}^{n} \frac{f(i)}{a^i} \right)$$

II Faisons le point

notons qu'il arrive fréquement qu'on compte le nombre d'appels récursifs pour évaluer la complexité d'un algorithme récursif

```
def f(n) =
  if n == 0 then
    return 0
  else:
    return f(n-1)
```

$$a(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 + a(n-1) & \text{sinon} \end{cases}$$

```
   f(10) -> f(9) -> f(8) -> f(7) -> f(6) -> f(5)
-> f(4)  -> f(3) -> f(2) -> f(1) -> f(0)
```

$$a(n) = 1^n \left(1 + \sum_{i=1}^{n} \frac{1}{1^i} \right) = 1 + n$$

on vérifie $a(10) = 11$

II Faisons le point

on sait obtenir une approximation asymptotique pour :

- des équations de partition mais bien identifier et prouver le cas
- des équations linéaires du 1er ordre

on peut aussi utiliser la méthode de substition/arbre pour les cas "simples", mais attention aux erreurs de calcul dans la hauteur de l'arbre, sur les termes additionnels, etc.

On est prêt pour l'analyse en complexité de tous les algorithmes que l'on va écrire