2019-2020, Semestre d'automne
L3, Licence Sciences et Technologies
Université Lyon 1

# LIFAP6: Algorithmique, Programmation et Complexité

**Chaine Raphaëlle (responsable semestre automne)**
E-mail : raphaelle.chaine@liris.cnrs.fr
http://liris.cnrs.fr/membres?idn=rchaine



## Conception et analyse des algorithmes

Décomposition de l'activité de programmation :

1. Exposé d'un problème
2. Modélisation et formalisation de ce problème (spécification)
3. Construction d'une solution algorithmique
4. Vérification justesse de l'algorithme (preuve)
5. Analyse complexité
6. Construction du programme
7. Exécution du programme pour résoudre le problème



1. **Exposé du problème**
   - Exemple : tri d'une séquence
2. **Modélisation et formalisation d'un problème** :
   - Étant donné un ensemble de valeurs ou l'état d'un système (entrée),
   - comment produire un ensemble de valeurs ou modifier l'état du système (sortie)

   avec relation désirée entre l'entrée et la sortie (peut faire appel à des prédicats et des fonctions d'accès)

- Exemple
  - entrée : une séquence de n nombres $\{a_1, a_2, \ldots, a_n\}$
  - sortie : une permutation de la séquence $\{a_1', a_2', \ldots, a_n'\}$
  - telle que

$$a_1' \leq a_2' \leq \ldots \leq a_n'$$



3. **Algorithme :**
   - composition d'un nombre fini d'étapes dont chacune **devra** pouvoir être
     - définie de façon rigoureuse et non ambiguë
     - effectivement réalisable sur une machine

     … **lors de la mise en œuvre** dans un langage de programmation
   - Remarque
     - L'algorithme peut être exprimé dans un modèle abstrait du monde
     - … qui se précisera au fur et à mesure que l'on affinera son degré de précision



- Exemple :
  - Algorithme informel de tri :

```
tant que il existe i < j t. que ai> aj alors
    échanger ai et aj
fin tantque
```

  - Opérations élémentaires : comparaison et échange
  - On peut d'ores et déjà prouver la justesse de cet algorithme, cependant, les différents raffinements de cet algorithme pourront conduire à des algorithmes d'efficacité potentiellement différentes
    - Tri à bulle : échange d'éléments consécutifs
    - Tri par sélection du minimum : échange du premier élément de la partie non triée avec son minimum
    - Le tri par insertion peut également être vu comme une variante de cet algorithme de tri informel



4. **Preuve :**
   - Un algorithme est dit **correct** si pour n'importe quelle entrée satisfaisant les spécifications il **s'arrête** avec la sortie correcte
   - Un algorithme correct **résout** un problème de calcul

   Il existe une théorie de la **calculabilité** qui caractérise les problèmes qui peuvent être résolus à l'aide d'un algorithme (MIF15)

- *Exemple* : le problème de l'arrêt
  - Il ne peut pas exister d'algorithme qui pour tout programme P et toute donnée D répond oui ou non à la question :
  - « P se termine-t-il pour D ? »

Démonstration:

- Supposons que vous réussissez à construire un algorithme **TestTermine** qui pour tout programme P et toute donnée répond **oui** ou **non** à la question : « P se termine-t-il pour D? »
- Exécutez **TestTermine** sur la procédure récursive suivante (qui contient elle-même un appel à **TestTermine**)
  - Procedure **Maligne**()
    Debut
      si **TestTermine(Maligne)**
      alors **Maligne**
      finsi
    Fin

  L'exécution de Maligne termine si et seulement si TestTermine répond en un temps fini qu'elle ne termine pas! Ce qui est contradictoire!

87

- Preuve de l'algorithme informel de tri
  - Définitions
    - Inversion dans une séquence a : couple (i,j) t.que i<j et $a_i > a_j$
    - Soit inv(a) le nombre d'inversions de a
  - Soit (i,j) une inversion : i<j et $a_i > a_j$
    - On décompose a en plusieurs parties :

$$a = \{sa_1, \mathbf{a_i}, sa_2, \mathbf{a_j}, sa_3\}$$

    - Après permutation de $a_i$ et $a_j$ on obtient a'

$$a' = \{sa_1, \mathbf{a_j}, sa_2, \mathbf{a_i}, sa_3\}$$

    - Montrons que inv(a') < inv(a)

88

- En effet :
  - le nombre d'inversions de $a_i$ (resp. $a_j$) avec $sa_2$ dans a' ne peut être que plus petit que le nombre d'inversions de $a_j$ (resp. $a_i$) avec $sa_2$ dans a
  - L'inversion entre $a_i$ et $a_j$ a disparu
  - Le nombre d'inversions faisant intervenir $a_j$ ou $a_j$ avec un élément de $sa_1$ ou $sa_3$ ne change pas
  - Le nombre d'inversions ne faisant intervenir ni $a_i$ ni $a_j$ ne change pas
- A chaque passage dans la boucle, inv(a) décroît strictement
- La séquence est triée quand inv(a) s'annule

89

## 5. Analyse de complexité

- Analyser un algorithme, c'est prévoir les ressources nécessaires à son exécution
  - mémoire utilisée
  - temps d'exécution
  - largeur de bande d'une communication
  - …
- Comment mesurer l'efficacité d'un algorithme ?
  - Mesure du temps d'exécution d'une mise en œuvre de l'algorithme?

  NON car dépendance
  - à la machine
  - au langage de programmation
  - au compilateur
  - aux données

90

- On aimerait pouvoir dire :
  - Sur toute machine et quel que soit le langage de programmation, l'algorithme A1 est meilleur que l'algorithme A2 pour des données « de grande taille »

  ➡ On compte le nombre d'opérations élémentaires de l'algorithme

Grande taille?

- Fibo(n: entier) : ici grande taille de la valeur de l'entier
- Addition (n1, n2 : entiers codés sur n bits)
  ici grande taille du nombre de bits n
- Tri(n : entier, tab[1..n] de entiers)
  ici grande taille du nombre n d'éléments à trier

91

- « de grande taille »?
  - Exemples :
    - Un algo qui s'exécute sur un ensemble de données en nombre de plus en plus grand
    - Un algo qui s'exécute sur un nombre n stocké en mémoire en utilisant un nombre de bits de plus en plus grand (par exemple n de plus en plus grand ou de plus en plus précis)
    - Un algo qui s'exécute sur un nombre qui implique un nombre d'instructions de plus en plus grand
    - ….

92

- Complexité en temps :
  - T(algo) : temps d'exécution de l'algorithme algo
  - T(algo(d)) : temps d'exécution de algo appliqué aux données d
- Règles utilisées (architecture RAM)
  - T(opération élémentaire) = Constante
  - Séquence d'instructions
    $T(\{i_1,i_2\}) = T(i_1)+T(i_2)$ (somme)
  - Branchements conditionnels

$$T(\textbf{si } C \textbf{ alors } I_1 \textbf{ sinon } I_2) \leq T(C) + max(T(I_1), T(I_2))$$

  - Boucles

$$T(\textbf{pour } i \textbf{ de } i_1 \textbf{ a } i_2 \textbf{ faire } P(i)) = \sum_i T(P(i))$$

  - Temps de calcul de procédures récursives
    - Solution d'équations de récurrence



- T(algo(d)) dépend en général de d et notamment de sa taille n :
  - nombre d'éléments constituant d
  - nombre de bits pour représenter d
  - nombre de sommets et d'arêtes d'un graphe, etc.
- Complexité au pire

$$T_{MAX}(\text{algo}, \text{n}) = \max\{T(\text{algo}, \text{d}) : \text{d de taille n}\}$$

- Complexité au mieux

$$T_{MIN}(\text{algo}, \text{n}) = \min\{T(\text{algo}, \text{d}) : \text{d de taille n}\}$$

- Complexité en moyenne

$$T_{MOY}(\text{algo}, \text{n}) = \sum_{\text{d de taille n}} \text{p(d)} T(\text{algo}, \text{d})$$

Où p(d) probabilité d'avoir la donnée d de taille n



# Comportement asymptotique

- Généralement, on compare l'efficacité des algorithmes sur des données de taille importante
  - La manière dont évolue le temps d'exécution d'un algorithme est alors pertinente
  - Etude de l'efficacité **asymptotique** des algorithmes
- Les outils utilisés doivent permettent de comparer les profils de cette évolution
- Idée :
  - Indiquer que le profil asymptotique d'un temps d'exécution est meilleur, moins bon ou similaire à un profil type



# Ordres de grandeurs asymptotiques

- Borne asymptotique supérieure : « grand O »
  T(algo,n)=O(f(n)) ssi
  il existe C>0 et N>0
  tels que pour tout n>N T(algo,n) ≤ Cf(n)
- Borne asymptotique inférieure « grand Oméga »
  T(algo,n)=Ω(f(n)) ssi
  il existe C>0 et N>0
  tels que pour tout n>N T(algo,n) ≥ Cf(n)
- Borne approchée asymptotique « grand théta »
  T(algo,n)=θ(f(n)) ssi
  il existe $C_1$>0, $C_2$>0 et N>0
  tels que pour tout n>N $C_1$f(n) ≤ T(algo,n) ≤ $C_2$f(n)
- Propriété

$$f = O(g) \quad et \quad f = \Omega(g) \Rightarrow \quad f = \Theta(g)$$



- Exemple :
  A-t-on

$$\frac{1}{2}n^2 - 3n = \Theta(n^2)?$$

cela revient à montrer que

$$\exists c_1, \exists c_2, \exists n_0 \; tels\ que \quad \forall n \geq n_0, \quad c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2, \text{soit}$$

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

$$\text{pour } n \geq 1, \frac{1}{2} \geq \frac{1}{2} - \frac{3}{n}$$

$$\text{pour } n \geq 7, \frac{1}{2} - \frac{3}{n} \geq \frac{1}{14}$$

$$\text{donc } c_1 = \frac{1}{14}, c_2 = \frac{1}{2} \text{ et } \quad n_0 = 7$$



- Exemple

$$f(n) = \begin{cases} n+1 & \text{si} \quad n \quad \text{pair} \\ n^2 - n + 1 & \text{si} \quad n \quad \text{impair} \end{cases}$$

$$f(n) = O(n^2), \text{car pour } n \geq 2, n^2 \geq n^2 - n + 1 \geq n + 1$$

$$f(n) = \Omega(n), \text{car pour } n \geq 1, n^2 - n + 1 \geq n \text{ et } n + 1 \geq n$$

- Un outil supplémentaire

  T(algo,n)=o(f(n)) ssi
  pour tout ε>0, il existe N>0
  tel que pour tout n>N   T(algo,n) ≤ εf(n)

- Propriété

  si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0, \text{alors } f = o(g)$

- Si f=o(g) alors f+g=θ(g)

99

- Utilisation des notations

$$\bullet\ 2n^2 + 3n + 1 = 2n^2 + \Theta(n)$$
$$\bullet\ 2n^2 + \Theta(n) = \Theta(n^2)$$
$$\bullet\ 2n^2 + 3n + 1 = 2n^2 + \Theta(n) = \Theta(n^2)$$

100

# Un peu de vocabulaire

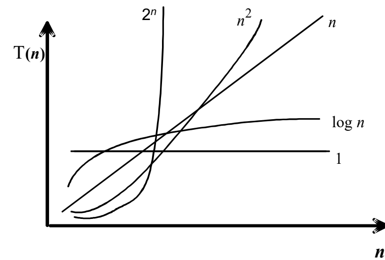
**Types de complexité**

- *O*(1) complexité constante
  (indépendante de la taille de la donnée)
- *O*(*log*(*n*)) complexité logarithmique
- *O*(*n*) complexité linéaire
- *O*(*nlog*(*n*)) complexité quasi-linéaire
- $O(n^2)$ complexité quadratique
- $O(n^3)$ complexité cubique
- $O(n^p)$ complexité polynomiale
- $O(n^{\log(n)})$ complexité quasi-polynomiale
- $O(2^n)$ complexité exponentielle
- *O*(*n*!) complexité factorielle

101

# Ordres de grandeur (exemples)

- 1, log *n*, *n*, *n*.log *n*, $n^2$, $n^3$, $2^n$




102

# Classification des algorithmes

- Il existe une classification grossière des algorithmes :

$$\text{algorithmes} \begin{cases} \text{polynomiaux} & (\log n, n^{0.5}, n, n\log n, n^2, \ldots) \\ \text{exponentiels} & (2^n, n^{\log n}, n!, n^n, \ldots) \end{cases}$$

- un bon algorithme est polynomial!

103

| taille / complexité | 20 | 50 | 100 | 200 | 500 | 1000 |
|---|---|---|---|---|---|---|
| $10^3$ n | 0.02 s | 0.05 s | 0.1 s | 0.2 s | 0.5 s | 1 s |
| $10^3$ n logn | 0.09 s | 0.3 s | 0.6 s | 1.5 s | 4.5 s | 10 s |
| 100 $n^2$ | 0.04 s | 0.25 s | 1 s | 4 s | 25 s | 2 mn |
| 10 $n^3$ | 0.02 s | 1s | 10 s | 1 mn | 21 mn | 27 h |
| $n^{logn}$ | 0.4 s | 1.1 h | 220 jours | 12500 ans | 5.$10^{10}$ ans | _ |
| $2^{n/3}$ | 0.0001 s | 0.1 s | 2.7 h | 3. $10^6$ ans | _ | _ |
| $2^n$ | 1 s | 36 ans | _ | _ | _ | _ |
| $3^n$ | 58 mn | 2.$10^{11}$ ans | _ | _ | _ | _ |
| n! | 77 100 ans | _ | _ | _ | _ | _ |

Opération : 1ms

104

## Problèmes faciles et difficiles

- On va considérer deux types de problèmes :
  - **Problème d'extraction**
    - on cherche dans un ensemble fini **S** un élément ou une configuration **s** qui vérifie une propriété donnée **Pr**
  - **Problème d'existence (PE)**
    - **Savoir si un problème d'extraction admet ou non une solution est un problème d'existence**

Attention! La taille de S peut être grande par rapport à la taille n des données en entrée.

- Exemple :
  - n nombres en entrée, et S l'ensemble des sous ensembles possibles que l'on peut faire avec ces n nombres
  - La taille de S est exponentielle $2^n$ par rapport à n

105

## Problèmes faciles et difficiles

- La réponse à un problème d'existence est Oui ou Non
  - *définition 1* : les PE qui admettent des algorithmes polynomiaux forment la **classe P**
  - *définition 2* : la **classe NP** est celle des PE
    - pour lesquels on ne sait pas forcément dire directement s'il existe un élément ou une configuration solution (satisfaisant **Pr**),
    - MAIS qui admettent un algorithme polynomial capable de tester si un élément ou une configuration **s** satisfait ou non **Pr**.

106

- Remarque :
  - Une manière de résoudre les problèmes dans *NP* consiste donc à énumérer l'ensemble des éléments **s** de **S** et de tester s'ils satisfont **Pr** à l'aide de l'algorithme polynomial
  - Si le problème est dans NP, cela signifie que la taille de **S** n'est donc pas polynomiale par rapport à la taille **n** des entrées!
- *Exemple* :
  - Etant donné un ensemble E de n entiers et un entier b, existe-t-il un sous-ensemble T de E tel que la somme des éléments de T soit égale à b ?
  - S est l'ensemble des sous-ensembles T de E

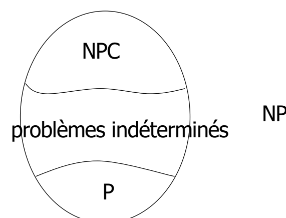
107

- Il n'existe pas d'algorithme polynomial connu à ce problème qui est donc dans NP.
- En effet :
  - une solution de problème d'extraction peut être proposée sous forme d'un sous-ensemble T d'éléments de E
  - un algorithme (linéaire) de vérification est le suivant
    - la somme des éléments de T est elle égale à b?
- *définition 3* : un problème **NP-complet** est un problème de NP en lequel se transforme polynomialement tout autre problème de NP
- on notera NPC l'ensemble des problèmes NP-complets

108

- *Exemple* : SAT (satisfaisabilité)
- Données :
  - n variables booléennes $x_i$
  - m clauses $C_j$ (combinaisons de ET et de OU)
- Question :
  - Peut-on affecter à chaque variable $x_i$ une valeur de façon à rendre vraies toutes les clauses ?
- Ce problème est NP-complet (Cook, 1970)

109




le problème de l'isomorphisme de 2 graphes est indéterminé

*conjecture* : $P \neq NP$

110

- Conclusion : il existe des problèmes qu'il ne faut pas s'acharner à résoudre de manière optimale ou de manière exacte
- Mais tout n'est pas perdu!
- Si on a un problème pour lequel on ne connaît pas d'algorithme polynomial, on peut essayer de résoudre le problème en tirant parti :
  - de la taille des données (petits cas, …)
  - de la nature des données (cas particuliers, …)
  - de la complexité moyenne (le pire cas est rare, …)
  - de méthodes diminuant la combinatoire (méthodes arborescentes, heuristiques …)
  - de méthodes approchées

111

## Spécificités des algorithmes itératifs et récursifs

- Outils de preuve
  - Algorithmes itératifs :
    - Invariant de boucle et condition d'arrêt
  - Algorithmes récursifs
    - Raisonnement par récurrence

112

- **Exemple de preuve d'un algorithme itératif**
- **procédure** rechercheDansSequence(
  **donnée** s : séquence, e: element,
  **résultat** i : position, b :booléen)
  {précondition : s, e initialisés
  postcondition : si b contient vrai
  alors e est en ième position de s,
  sinon e n'est pas dans s}
  **début**
  i←1, b ←faux
  **tantque** (i ≤ taille(s)) **etalors** (e≠s[i]) **faire**
  i++
  **fintantque**
  **si** i ≤ taille(s) **alors**
  b←true
  **finsi**
  **fin**

113

- Invariant de boucle
- i←1, b ←faux
  **tantque** (i ≤ taille(s)) **etalors** (e≠s[i]) **faire**
  {Assertion : Au kième passage éventuel dans la boucle, k=i et pour tout j t.que 1 ≤ j < k on a s[j] ≠ e}
  i++
  **fintantque**
- Preuve de l'invariant de boucle
- Récurrence
  - Vrai pour k=1
  - Supposons le vrai pour k=K
  - Si on passe une K+1 ième fois dans la boucle, cela signifie que K+1 ≤ taille(s) et que s[K+1] ≠ e donc s[j] ≠ e pour tout 1 ≤ j < K+1, ce qui clôt la récurrence

114

- Il faut également montrer que le corps de la boucle permet de progresser vers la condition d'arrêt
  - Au Kième passage dans la boucle, i augmente strictement : on ne pourra pas passer une infinité de fois dans la boucle, puisqu'une des conditions d'arrêt sera atteinte quand i excèdera taille(s)
- A la sortie de la boucle :
  - 1er cas : si i> taille(s) alors l'invariant du dernier passage dans la boucle assure que pour 1 ≤ j ≤ taille(s) on a s[j] ≠ e, donc e n'est pas dans s, ce qui est cohérent avec le fait que b contienne faux
  - 2ème cas : si i ≤ taille(s) alors s[i] vaut e ce qui est cohérent avec l'affectation de vrai à b

115

- **Exemple de preuve d'un algorithme récursif**
- Tri fusion d'un tableau
- **procédure** TriFusionRec(
  **donnée-résultat** tab : tableau[1..n] d'Elements,
  **données** p, r : 1..n)
  {précondition : aucune,
  postcondition : tab trié entre les indices p et r}
  **variable**
  q : 1..n
  **début**
  **si** p<r **alors**
  q←(p+r)/2
  TriFusionRec(tab,p,q) tri 1ère moitié
  TriFusionRec(tab,q+1,r) tri 2nde moitié
  Fusionner(tab,p,q,r)
  **finsi**
  **fin**

| | | | | p | | | q |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 24 | 56 | 2 | 26 | 10 | 100 | 45 | 9 |

116

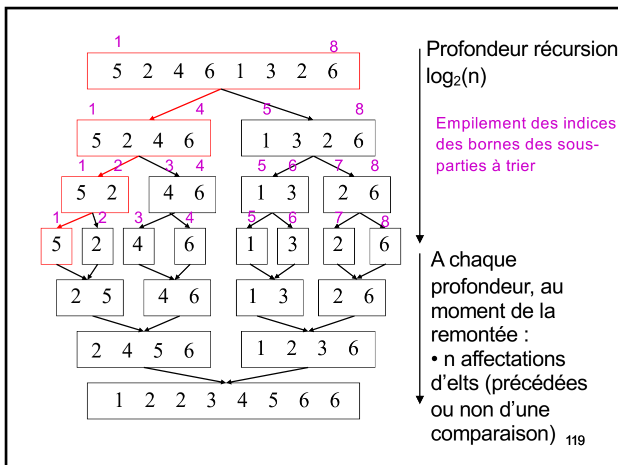
- Raisonnement par récurrence :
  - Pour un(e partie de) tableau de taille 0 ou 1, l'algorithme est correct
  - On suppose que l'algo est correct pour un tableau de taille m ≥1, montrons qu'il est correct pour un tableau de taille m+1
    - Comme m+1 ≥ 2 on a p≤q<r
      les 2 appels récursifs se font donc sur des tableaux de taille inférieure à m+1
    - Sous réserve que la procédure de fusion soit correcte, l'algorithme de tri fusion sera donc correct



# Complexité d'un algorithme récursif

- Le temps d'exécution d'un algorithme récursif est généralement solution d'une équation de récurrence
- Exemple : Analyse de la procédure TriFusionRec
  - Appelons T(n) le temps d'exécution de l'algorithme sur un tableau de taille n






Profondeur récursion $\log_2(n)$

Empilement des indices des bornes des sous-parties à trier

A chaque profondeur, au moment de la remontée :
- n affectations d'elts (précédées ou non d'une comparaison)



- T(1)=T(initialisation des paramètres formels) +T(comparaison d'indice)=$C_1$
- Pour n≥2
  T(n)= T(initialisation des paramètres formels)
  + T(comparaison d'indice)
  + T(affectation d'indice)
  + 2*T(n/2)
  + T(fusionner,n)
- Or T(fusionner,n)=θ(n)



$$T(n) = \begin{cases} C1 \text{ si } n=1 \\ 2T(n/2)+\theta(n) + C1 + C2 \text{ si } n>1 \end{cases}$$

$$T(n) = \begin{cases} C1 \text{ si } n=1 \\ 2T(n/2)+\theta(n) \text{ si } n>1 \end{cases}$$



# Résolution

Méthode par développement itératif

Soit n>1 et soit $k=\lg_2(n)$

(on supposera ici que n correspond exactement à $2^k$)

$T(n)=2T(n/2)+\theta(n)$

ie. $T(n) = 2(2T(n/2^2)+ \theta(n/2)) + \theta(n)$

ie. $T(n) = 2^2T(n/2^2)+2\theta(n/2) + \theta(n)$

ie. $T(n) = 2^kT(n/2^k)+ 2^{k-1}\theta(n/2^{k-1}) + 2^{k-2}\theta(n/2^{k-2}) \ldots +\theta(n)$

ie. $T(n) = 2^kT(n/2^k)+ \theta(n) + \ldots +\theta(n)$

ie. $T(n) = n\,C1 + \lg_2(n)\theta(n)$

ie. $T(n) = \theta(n \lg_2(n))$

- Il s'agit du temps d'exécution d'un algorithme qui divise un problème de taille n en a≥1 sous-problèmes de taille n/b avec b>1 (stratégie **« diviser pour régner »**)

$$T(n) = aT(\frac{n}{b}) + f(n) \quad n>0$$

*f(n)* décrit le coût de la division du problème en a sous-problèmes et de la recomposition des résultats

123

## Master Theorem

1. Si il existe $\varepsilon > 0$ t. que $f(n) = O(n^{\log_b(a)-\varepsilon})$ alors $T(n) = \Theta(n^{\log_b(a)})$
2. Si $f(n) = \Theta(n^{\log_b(a)})$ alors $T(n) = \Theta(n^{\log_b(a)} \log_b(n))$
3. Si il existe $\varepsilon > 0$ t.que $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ et il existe $c < 1$ t. que $af(\frac{n}{b}) \leq cf(n)$ pour n grand alors $T(n) = \Theta(f(n))$

Intuition de preuve au tableau

124

## Master Theorem

1. Si il existe $\varepsilon > 0$ t. que $f(n) = O(n^{\log_b(a)-\varepsilon})$ alors $T(n) = \Theta(n^{\log_b(a)})$ Coût de la décomposition et recomposition négligeable devant le coût lié à la récursion
2. Si $f(n) = \Theta(n^{\log_b(a)})$ alors $T(n) = \Theta(n^{\log_b(a)} \log_b(n))$ Coût de la décomposition et recomposition similaire à celui lié à la récursion
3. Si il existe $\varepsilon > 0$ t.que $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ et il existe $c < 1$ t. que $af(\frac{n}{b}) \leq cf(n)$ pour n grand alors $T(n) = \Theta(f(n))$ Coût de la décomposition et recomposition non négligeable devant le coût lié à la récursion, … mais maîtrisé!

125