# Algorithmique Avancée II

# Plan du cours

- Rappels
- Tables de hachage
- Dérécursivation
- Arbres binaires
- Arbres équilibrés
- B-Arbres
- graphes
- ...
- **Bibliographie :**
  - Introduction to Algorithms par Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest
  - Types de données et algorithmes par Christine Froidevaux, Marie-Claude Gaudel, Michèle Soria
  - Utiliser l 'internet!

# Rappels : algorithmique I

- **Qu'est-ce que l'algorithmique ?**
  - **Définition 1 (Algorithme).** *Un algorithme est suite finie d'opérations élémentaires constituant un schéma de calcul ou de résolution d'un problème.*
- **Double problématique de l'algorithmique ?**
  1. **Trouver une méthode** de résolution (exacte ou approchée) du problème.
  2. Trouver une méthode **efficace**.

  *=>Savoir résoudre un problème est une chose, le résoudre efficacement en est une autre, ou encore montrer qu 'il est correcte ...!!*

# Rappels : algorithmique I

- **Exemple 1:**
  - **problème : calculer $x^n$**

    **données : x : réel , n: entier**

    *Méthode 1 :* $x^0 = 1$; $x^i = x * x^{i-1}$ $i > 0$ $\quad T = n$

    *Méthode 2 :* $x^0 = 1$;

    $x^i = x^{i/2} * x^{i/2}$ , si i est pair; $\quad T = \log n$

    $x^i = x * x^{i/2} * x^{i/2}$ si i est impair

    ...

    **résultats** : y = $x^n$

  - Laquelle choisir? et pourquoi?
    - Plutôt la deuxième.

*=>Analyse de la complexité des algorithmes*

# Analyse de la complexité :

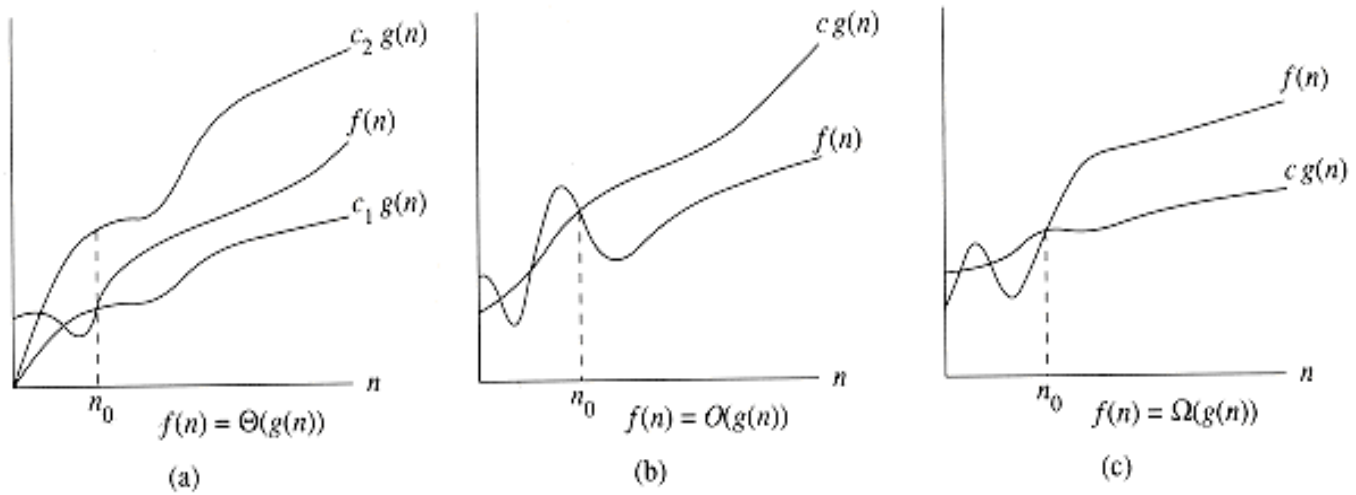
- Notation de Landau:
  - On ne s ’intéresse pas en général à la complexité exacte, mais à son ordre de grandeur.

=> besoin de notations asymptotiques.

$$\mathbf{O} \quad : \quad f = O(g) \quad \Leftrightarrow \quad \exists n_0, \exists c \geq 0, \forall n \geq n_0, f(n) \leq c \times g(n)$$

$$\mathbf{\Omega} \quad : \quad f = \Omega(g) \quad \Leftrightarrow \quad g = O(f)$$

$$\mathbf{\Theta} \quad : \quad f = \Theta(g) \quad \Leftrightarrow \quad f = O(g) \text{ et } g = O(f)$$



# Analyse de la complexité :

- Exemples :

0 : $n = O(n)$, (prendre $n_0 = 1$, $c = 1$)

$2n = O(3n)$ (prendre $n_0 = 1$ $c = 2/3$)

$n^2 + n - 1 = O(n^2)$ (prendre $n_0 = 1$, $c = 1$)

$\theta$ : $1/2\ n^2 - 3n = \theta(n^2)$

trouver n0 c1, c2 t.q.

$c1\ n^2 \leq 1/2\ n^2 - 3n \leq c2\ n^2$

$c1 \leq 1/2 - 3/n \leq c2$

la partie droite de l 'inéquation peut être satisfaite pour n0 = 1 et c2 = 1/2

la partie gauche de l 'inéquation peut être satisfaite pour n0 = 7 et c1 = 1/14

=> en choisissant n0 =7 , c1 =1/14, c2 = 1/2

=> $1/2\ n^2 - 3n = \theta(n^2)$

....

# Analyse de la complexité :

- **Complexité**

  **Définition 2 (Complexité).** *La complexité d'un algorithme est la mesure du nombre d'opérations fondamentales qu'il effectue sur un jeu de données. La complexité est exprimée comme une fonction de la taille du jeu de données.*

- *Nous notons Dn l'ensemble des données de taille n et C(d) le coût de l'algorithme sur la donnée d.*

- **Complexité au meilleur** : $T\min(n) = \min_{d \in Dn} C(d)$.
  C'est le plus petit nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée, ici à *n*.
  C'est une borne inférieure de la complexité de l'algorithme sur un jeu de données de taille *n*.

# Analyse de la complexité :

- **Complexité au pire** : $T\max(n) = \max_{d \in Dn} C(d)$. C'est le plus grand nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée, ici à $n$.
  - **Avantage** : il s'agit d'un maximum, et l'algorithme finira donc toujours avant d'avoir effectué $T\max(n)$ opérations.
  - **Inconvénient** : cette complexité peut ne pas refléter le comportement « usuel » de l'algorithme, le pire cas pouvant ne se produire que très rarement, mais il n'est pas rare que le cas moyen soit aussi mauvais que le pire cas.

# Analyse de la complexité :

**Complexité en moyenne** : $Tmoy(n) = \sum d \in Dn\ C(d) / |Dn|$

C'est la moyenne des complexités de l'algorithme sur des jeux de données de taille *n* (en toute rigueur, il faut bien évidemment tenir compte de la probabilité d'apparition de chacun des jeux de données).

**Avantage** : reflète le comportement « général » de l'algorithme si les cas extrêmes sont rares ou si la complexité varie peu en fonction des données.

**Inconvénient** : la complexité en pratique sur un jeu de données particulier peut être nettement plus importante que la complexité en moyenne, dans ce cas la complexité en moyenne ne donnera pas une bonne indication du comportement de l'algorithme.

En **pratique**, nous ne nous intéresserons qu'à la complexité au pire et à la complexité en moyenne.

# Analyse de la complexité :

- **Modèle de machine**

  Pour que le résultat de l'analyse d'un algorithme soit pertinent, il faut avoir un modèle de la machine sur laquelle l'algorithme sera implémenté (sous forme de programme). On prendra comme référence un modèle de **machine à accès aléatoire (RAM)** et à processeur unique, où les instructions sont exécutées l'une après l'autre, sans opérations simultanées.

# Analyse de la complexité :

## premier algorithme de tri

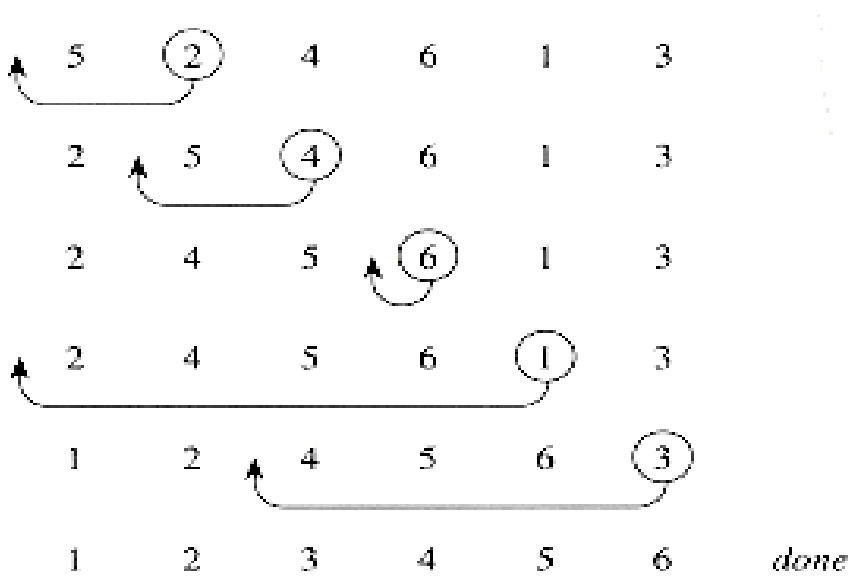
- **Illustration : cas du tri par insertion**
  - **Problématique du tri**

**données : une séquence de n nombre** *a1, a2, . . . , an*.

**résultats** : une permutation *<a ’1, a ’2, . . . , a ’n> des données t.q.*

*a ’1 ≤ a ’2 ≤, . . . , ≤ a ’n*

- exemple :



# Analyse de la complexité : algorithme de tri

- **Complexité**
  - attribution d ’un coût en temps ***ci*** à chaque instruction, et
  - compter le nombre d’exécutions de chacune des instructions.
  - Pour chaque valeur de $j \in [2..n]$, nous notons ***t j*** le nombre d’exécutions de la boucle **tant que** pour cette valeur de $j$.

    Il est à noter que la valeur de ***t j*** **dépend des données**…

| TRI-INSERTION | *Coût* | *Nombre d’exécutions* |
|---|---|---|
| **Pour** $j \leftarrow 2$ **à** $n$ **faire** | $c_1$ | $n$ |
| clé $\leftarrow$ A[$j$] | $c_2$ | $n-1$ |
| $i \leftarrow j-1$ | $c_3$ | $n-1$ |
| **tant que** $i > 0$ **et** A[$i$] > clé **faire** | $c_4$ | $\sum_{j=2}^{n} t_j$ |
| A[$i+1$] $\leftarrow$ A[$i$] | $c_5$ | $\sum_{j=2}^{n} (t_j - 1)$ |
| $i \leftarrow i-1$ | $c_6$ | $\sum_{j=2}^{n} (t_j - 1)$ |
| A[$i+1$] $\leftarrow$ clé | $c_7$ | $n-1$ |

  - Le temps d’exécution total de l’algorithme est alors :

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

# Analyse de la complexité : algorithme de tri

- **Complexité au meilleur** : le cas le plus favorable pour l'algorithme TRI-INSERTION est quand le tableau est déjà trié. Dans ce cas $t_j = 1$ pour tout $j$.

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7). \end{aligned}$$

- $T(n)$ peut ici être écrit sous la forme $T(n) = an+b$, $a$ et $b$ étant des *constantes* indépendantes des entrées, et
- $T(n)$ est donc une **fonction linéaire** de $n$.

# Analyse de la complexité : algorithme de tri

- **Complexité au pire** : le cas le plus défavorable pour l'algorithme TRI-INSERTION est quand le tableau est déjà trié dans l 'ordre inverse. Dans ce cas $tj = j$ pour tout $j$.

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

$$\downarrow$$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1)$$

$$= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right)n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right)n - (c_2 + c_3 + c_4 + c_7).$$

Rappel : $\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$. donc $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ et $\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$.

- $T(n)$ peut ici être écrit sous la forme $T(n) = ax^2 + bx + c$, $a$, $b$ et $c$ étant des constantes, et $T(n)$ est donc une fonction quadratique.

# Analyse de la complexité : algorithme de tri

- **Complexité en moyenne** : supposons que l'on applique l'algorithme de tri par insertion à *n* nombres choisis au hasard. Quelle sera la valeur de $t_j$ ? C'est-à-dire, où devra-t-on insérer A[ *j*] dans le sous-tableau A[1.. j-1]? En moyenne $tj = j/2$.
- *Si l 'on reporte cette valeur dans l 'équation on obtient également une fonction quadratique.*

  **meilleur cas** : $\Theta(n)$.

  **pire cas** : $\Theta(n^2)$.

  **en moyenne** : $\Theta(n^2)$.
- En général, on considère qu'un algorithme est plus efficace qu'un autre si sa complexité dans le pire cas a un ordre de grandeur inférieur.

# Analyse de la complexité : algorithme de tri

- **Classes de complexité**

Les algorithmes usuels peuvent être classés en un certain nombre de grandes classes de complexité :

– Les algorithmes ***sub-linéaires*** dont la complexité est en général en *O(logn)*.

– Les algorithmes ***linéaires*** en complexité *O(n)* et ceux en complexité en *O(nlogn)* sont considérés comme rapides.

– Les algorithmes ***polynomiaux*** en $O(n^k)$ pour $k > 3$ sont considérés comme lents, sans parler des algorithmes ***exponentiels*** (dont la complexité est supérieure à tout polynôme en *n*) que l'on s'accorde à dire impraticables dès que la taille des données est supérieure à quelques dizaines d'unités.

# La récursivité et le paradigme « diviser pour régner »

- **Récursivité**

  De l'art d'écrire des programmes qui résolvent des problèmes que l'on ne sait pas résoudre soi-même !

  - **Définition 4 (Définition récursive, algorithme récursif).** *Une définition récursive est une définition dans laquelle intervient ce que l'on veut définir. Un algorithme est dit récursif lorsqu'il est défini en fonction de lui-même.*

- **Récursivité simple**

  Revenons à la fonction puissance $x \rightarrow x^n$. Cette fonction peut être définie récursivement :

$$x^n = \begin{cases} 1 & \text{si } n = 0; \\ x \times x^{n-1} & \text{si } n \geq 1. \end{cases}$$

# La récursivité et le paradigme « diviser pour régner »

- **Récursivité multiple**

  Une définition récursive peut contenir plus d'un appel récursif. Nous voulons calculer ici les combinaisons $C^n_p$ en se servant de la relation de Pascal :

$$C^p_n = \begin{cases} 1 & \text{si } p = 0 \text{ ou } p = n; \\ C^p_{n-1} + C^{p-1}_{n-1} & \text{sinon.} \end{cases}$$

- **Récursivité mutuelle**

  Des définitions sont dites *mutuellement récursives* si elles dépendent les unes des autres. Ça peut être le cas pour la définition de la parité :

$$\text{pair}(n) = \begin{cases} \textit{vrai} & \text{si } n = 0; \\ \text{impair}(n-1) & \text{sinon;} \end{cases} \quad \text{et} \quad \text{impair}(n) = \begin{cases} \textit{faux} & \text{si } n = 0; \\ \text{pair}(n-1) & \text{sinon.} \end{cases}$$

# La récursivité
# et le paradigme « diviser pour régner »

- **Récursivité imbriquée**

  La fonction d'Ackermann est définie comme suit :

$$A(m,n) = \begin{cases} n+1 & \text{si } m = 0 \\ A(m-1,1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m-1,A(m,n-1)) & \text{sinon} \end{cases}$$

# La récursivité
# et le paradigme « diviser pour régner »

- **Principe et dangers de la récursivité**
  - **Principe et intérêt :** ce sont les mêmes que ceux de la démonstration par récurrence en mathématiques. On doit avoir :
    - un certain nombre de cas dont la résolution est connue, ces « cas simples » formeront les cas d'arrêt de la récursion ;
    - un moyen de se ramener d'un cas « compliqué » à un cas « plus simple ».
  - La récursivité permet d'écrire des algorithmes concis et élégants.

# La récursivité et le paradigme « diviser pour régner »

- **Difficultés :**
  - la définition peut être dénuée de sens :

    Algorithme A(n)

    **renvoyer** A(n)
  - il faut être sûrs que l'on retombera toujours sur un cas connu, c'est-à-dire sur un cas d'arrêt ; il nous faut nous assurer que la fonction est complètement définie, c'est-à-dire, qu'elle est définie sur tout son domaine d'applications.

# La récursivité et le paradigme « diviser pour régner »

- **Non décidabilité de la terminaison**

  **Question** : peut-on écrire un programme qui vérifie automatiquement si un programme donné *P* termine quand il est exécuté sur un jeu de données *D*?

  - **Entrée:** Un programme P et un jeu de données D.
  - **Sortie:** *vrai* si le programme *P* termine sur le jeu de données *D*, et *faux* sinon.

- **Démonstration de la non décidabilité**

  - Supposons qu'il existe un tel programme, nommé ***termine***, de vérification de la terminaison. À partir de ce programme on conçoit le programme Q suivant :

# La récursivité et le paradigme « diviser pour régner »

**programme Q**

résultat = termine(Q)

**tant que** résultat = *vrai* **faire** attendre une seconde **fin tant que**

**renvoyer** résultat

- Supposons que le programme Q —qui ne prend pas d'arguments— termine. Donc termine(Q) renvoie *vrai*, la deuxième instruction de Q boucle indéfiniment et Q ne termine pas. Il y a donc contradiction et le programme Q ne termine pas. Donc, termine(Q) renvoie *faux*, la deuxième instruction de Q ne boucle pas, et le programme Q termine normalement. Il y a une nouvelle fois contradiction : par conséquent, il n'existe pas de programme tel que termine.

- **Le problème de la terminaison est indécidable!!**

# La récursivité et le paradigme « diviser pour régner »

- **Diviser pour régner**
  - **Principe**

    Nombres d'algorithmes ont une structure récursive : pour résoudre un problème donné, ils s'appellent eux-mêmes récursivement une ou plusieurs fois sur des problèmes très similaires, mais de tailles moindres, résolvent les sous problèmes de manière récursive puis combinent les résultats pour trouver une solution au problème initial.

    Le paradigme « diviser pour régner » donne lieu à trois étapes à chaque niveau de récursivité :

    - **Diviser :** le problème en un certain nombre de sous-problèmes ;
    - **Régner :** sur les sous-problèmes en les résolvant récursivement ou, si la taille d'un sous-problème est assez réduite, le résoudre directement ;
    - **Combiner :** les solutions des sous-problèmes en une solution complète du problème initial.

# La récursivité et le paradigme « diviser pour régner »

- **Premier exemple : multiplication naïve de matrices**
  - Nous nous intéressons ici à la multiplication de matrices carrés de taille $n$.
  - **Algorithme naïf**

```
MULTIPLIER-MATRICES(A, B)
Soit n la taille des matrices carrés A et B
Soit C une matrice carré de taille n
Pour i ← 1 à n faire
    Pour j ← 1 à n faire
        cij ← 0
        Pour k ← 1 à n faire
                    cij ← cij +aik * bkj
renvoyer C
```

  - Cet algorithme effectue $\Theta(n^3)$ multiplications et autant d'additions.

# La récursivité et le paradigme « diviser pour régner »

- **Algorithme « diviser pour régner » naïf**

  Dans la suite nous supposerons que $n$ est une puissance exacte de 2. Décomposons les matrices $A$, $B$ et $C$ en sous-matrices de taille $n/2$ $*n/2$. L'équation $C = A *B$ peut alors se récrire :

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix}$$

  - En développant cette équation, nous obtenons :

$$r = ae + bf, \quad s = ag + bh, \quad t = ce + df \quad \text{et} \quad u = cg + dh.$$

    Chacune de ces **quatre** opérations correspond à **deux** multiplications de matrices carrés de taille $n/2$ et une addition de telles matrices. À partir de ces équations on peut aisément dériver un algorithme « diviser pour régner » dont la complexité est donnée par la récurrence :

$$T(n) = 8T(n/2) + \Theta(n^2),$$

  - l'addition des matrices carrés de taille $n/2$ étant en $\Theta(n^2)$.

# La récursivité et le paradigme « diviser pour régner »

- **Analyse des algorithmes « diviser pour régner »**
  - Lorsqu'un algorithme contient un appel récursif à lui-même, son temps d'exécution peut souvent être décrit par une équation de récurrence qui décrit le temps d'exécution global pour un problème de taille *n* en fonction du temps d'exécution pour des entrées de taille moindre.

# La récursivité et le paradigme « diviser pour régner »

- La récurrence définissant le temps d'exécution d'un algorithme « diviser pour régner » se décompose suivant les trois étapes du paradigme de base :

1. Si la taille du problème est suffisamment réduite, $n \leq c$ pour une certaine constante $c$, la résolution est directe et consomme un temps constant $\Theta(1)$.
2. Sinon, on divise le problème en $a$ sous-problèmes chacun de taille $1/b$ de la taille du problème initial. Le temps d'exécution total se décompose alors en trois parties :

   (a) $D(n)$ : le temps nécessaire à la division du problème en sous-problèmes.

   (b) $aT(n/b)$ : le temps de résolution des $a$ sous-problèmes.

   (c) $C(n)$ : le temps nécessaire pour construire la solution finale à partir des solutions aux sous-problèmes.

# La récursivité et le paradigme « diviser pour régner »

- La relation de récurrence prend alors la forme :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq c, \\ aT(n/b) + D(n) + C(n) & \text{sinon,} \end{cases}$$

- où l'on interprète $n/b$ soit comme $\lfloor n/b \rfloor$ soit comme $\lceil n/b \rceil$.

# La récursivité et le paradigme « diviser pour régner »

- **Résolution des récurrences :**

**Théorème 1 (Résolution des récurrences « diviser pour régner »).**

*Soient $a \geq 1$ et $b > 1$ deux constantes, soit $f(n)$ une fonction et soit $T(n)$ une fonction définie pour les entiers positifs par la récurrence :*

$$T(n) = aT(n/b) + f(n),$$

*où l'on interprète $n/b$ soit comme $\lfloor n/b \rfloor$, soit comme $\lceil n/b \rceil$.*

*$T(n)$ peut alors être bornée asymptotiquement comme suit :*

1. *Si $f(n) = O(n^{(\log_b a)-\varepsilon})$ pour une certaine constante $\varepsilon > 0$, alors $T(n) = \Theta(n^{\log_b a})$.*
2. *Si $f(n) = \Theta(n^{\log_b a})$, alors $T(n) = \Theta(n^{\log_b a} \log n)$.*
3. *Si $f(n) = \Omega(n^{(\log_b a)+\varepsilon})$ pour une certaine constante $\varepsilon > 0$, et si $af(n/b) \leq cf(n)$ pour une constante $c < 1$ et $n$ suffisamment grand, alors $T(n) = \Theta(f(n))$.*

Pour une démonstration de ce théorème voir Rivest-Carmen-etal

Il existe d 'autres méthodes de résolution des récurrences : par substitution, changement de variables etc.

# La récursivité et le paradigme « diviser pour régner »

- **Algorithmes de tri**
  - **Tri par fusion**

**Principe**

L'algorithme de tri par fusion est construit suivant le paradigme « diviser pour régner » :

1. Il divise la séquence de $n$ nombres à trier en deux sous-séquences de taille $n/2$.
2. Il trie récursivement les deux sous-séquences.
3. Il fusionne les deux sous-séquences triées pour produire la séquence complète triée.

La récursion termine quand la sous-séquence à trier est de longueur 1 car une telle séquence est toujours triée.

# La récursivité et le paradigme « diviser pour régner »

```
TRI-FUSION(A, p, r)
  si p < r alors q ← ⌊(p+r)/2⌋
                 TRI-FUSION(A, p, q)
                 TRI-FUSION(A, q+1, r)
                 FUSIONNER(A, p, q, r)
```

- **Complexité**
  - Pour déterminer la formule de récurrence qui nous donnera la complexité de l'algorithme TRI-FUSION, nous étudions les trois phases de cet algorithme « diviser pour régner » :
    - **Diviser :** cette étape se réduit au calcul du milieu de l'intervalle [p..r]
    - **Régner :** l'algorithme résout récursivement deux sous-problèmes de tailles respectives n/2
    - **Combiner :** la complexité de cette étape est celle de l'algorithme de fusion qui est de $\Theta(n)$ pour la construction d'un tableau solution de taille $n$.

# La récursivité et le paradigme « diviser pour régner »

- Par conséquent, la complexité du tri par fusion est donnée par la récurrence :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1, \\ 2T(\frac{n}{2}) + \Theta(n) & \text{sinon.} \end{cases}$$

- Pour déterminer la complexité du tri par fusion, nous utilisons de nouveau le théorème.
  - Ici a=2 et b=2 donc $\log_b a = 1$, et nous nous trouvons dans le deuxième cas du théorème $f(n) = \Theta(n^{\log_b a}) = \Theta(n)$.
  - par conséquent : $T(n) = \Theta(n \log n)$.
  - Pour des valeurs de *n* suffisamment grandes, le tri par fusion avec son temps d'exécution en $\Theta(n \log n)$ est nettement plus efficace que le tri par insertion dont le temps d'exécution est en $\Theta(n^2)$.

# Dérécursivation

**Dérécursiver**, c'est transformer un algorithme récursif en un algorithme équivalent ne contenant pas d'appels récursifs.

- **Récursivité terminale**

  **Définition** *Un algorithme est dit récursif terminal s'il ne contient aucun traitement après un appel récursif.*

# Récursivité terminale

**Exemple :**

ALGORITHME **P**(U)

**si** *C(U)*

**alors**

*D(U)*;**P**(α(*U*))

**sinon** *T(U)*

où :

– *U* est la liste des paramètres ;

– *C(U)* est une condition portant sur *U* ;

– *D(U)* est le traitement de base de l'algorithme (dépendant de *U*) ;

– α(*U*) représente la transformation des paramètres ;

– *T(U)* est le traitement de terminaison (dépendant de *U*).

# Récursivité terminale

- Avec ces notations, l'algorithme P équivaut à l'algorithme suivant :

ALGORITHME **P'**(*U*)

**tant que** *C(U)* **faire**

*D(U)*;

$U \leftarrow \alpha(U)$

**fintantque**

*T(U)*

- L'algorithme **P'** est une version **dérécursivée** de

# Récursivité non terminale

- Ici, pour pouvoir dérécursiver, il va falloir sauvegarder le contexte de l'appel récursif, typiquement les paramètres de l'appel engendrant l'appel récursif.
- Originellement, l'algorithme est :

ALGORITHME **Q**(U)

**si** $C(U)$ **alors**

$D(U)$;**Q**($\alpha(U)$);$F(U)$

**sinon** $T(U)$

# Récursivité non terminale

⇒Utilisation de **piles**

- Après dérécursivation on obtiendra donc :

ALGORITHME **Q'**(U)

*empiler(nouvel_appel, U)*

***tant que*** *pile non vide* ***faire***

*dépiler(état, V)*

***si*** *état = nouvel_appel* ***alors*** $U \leftarrow V$

***si*** *C(U)* ***alors*** *D(U)*

*empiler(fin, U)*

*empiler(nouvel_appel, α(U))*

***sinon*** *T(U)*

***si*** *état = fin* ***alors*** $U \leftarrow V$

*F(U)*

# Illustration de la dérécursivation de l'algorithme Q

- Exemple d'exécution de Q :

**Appel Q**($U0$)

$C(U0)$ *vrai*

$D(U0)$

**Appel Q**($\alpha(U0)$)

$C(\alpha(U0))$ *vrai*

$D(\alpha(U0))$

**Appel Q**($\alpha(\alpha(U0))$)

$C(\alpha(\alpha(U0)))$ *faux*

$T(\alpha(\alpha(U0)))$

$F(\alpha(U0))$

$F(U0)$

# Exemple d'exécution de l'algorithme dérécursivé.

- Appel Q'($U0$)

  *empiler(nouvel_appel, U))*

  pile = [(nouvel_appel, $U0$)]

  *dépiler(état, V))*

  état $\leftarrow$ nouvel_appel ; $V \leftarrow U0$ ; pile = []

  $U \leftarrow U0$

  **C(U0)** ***vrai***

  **D(U0)**

  *empiler(fin, U))*

  pile = [(fin, $U0$)]

  *empiler(nouvel_appel, $\alpha(U)$))*

  pile = [(fin, $U0$) ; (nouvel_appel, $\alpha(U0)$)]

# Exemple d'exécution de l'algorithme dérécursivé.

*dépiler(état, V))*

état $\leftarrow$ nouvel_appel ; $V \leftarrow \alpha(U0)$ ; pile = [(fin, $U0$)]

$U \leftarrow \alpha(U0)$

**C**($\alpha$(**U0**)) ***vrai***

**D**($\alpha$(**U0**))

*empiler(fin, U))*

pile = [(fin, $U0$) ; (fin, $\alpha(U0)$)]

*empiler(nouvel_appel, $\alpha(U)$))*

pile = [(fin, $U0$) ; (fin, $\alpha(U0)$) ; (nouvel_appel, $\alpha(\alpha(U0))$)]

# Exemple d'exécution de l'algorithme dérécursivé.

*dépiler(état, V))*

état ← nouvel_appel ; $V \leftarrow \alpha(\alpha(U0))$ ; pile = [(fin, $U0$) ; (fin, $\alpha(U0)$)]

$U \leftarrow \alpha(\alpha(U0))$

$\mathbf{C}(\alpha(\alpha(\mathbf{U0})))$ ***faux***

$\mathbf{T}(\alpha(\alpha(\mathbf{U0})))$

*dépiler(état, V))*

état fin ; $V \leftarrow \alpha(U0)$ ; pile = [(fin, $U0$)]

$\mathbf{F}(\alpha(\mathbf{U0}))$

*dépiler(état, V))*

état ← fin ; $V \leftarrow U0$ ; pile = []

$\mathbf{F}(\mathbf{U0})$

# Dérécursivation

- **Remarques**
  - Les programmes itératifs sont souvent plus efficaces,
  - mais les programmes récursifs sont plus faciles à écrire.
  - Les compilateurs savent, la plupart du temps, reconnaître les appels récursifs terminaux, et ceux-ci n'engendrent pas de surcoût par rapport à la version itérative du même programme.
  - Il est toujours possible de dérécursiver un algorithme récursif.