

Algorithmique et complexité

Mention : Master Informatique

Bibliographie

- Introduction à l'algorithmique, par Thomas Cormen, Charles Leiserson, Ronald Rivest et Clifford Stein.
- A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass., 1974.
- J. L. Bentley, Programming Pearls, Addison-Wesley, Reading, Mass., 1986.
- D. E. Knuth, The Art of Computer Programming, Vol 2 : Seminumerical Algorithms, 2d ed., Addison-Wesley, Reading, Mass., 1981.

Structure du cours

- Chap-0 : Outils mathématiques
- Chap-1: Introduction & motivations
- Chap-2: Complexité & optimalité
- Chap-3: Algorithmes de tri
- Chap-4: Récursivité
 - ✓ Différents types de récursivité
 - ✓ Dérécursivation d'algorithmes
 - ✓ Récursivité terminale et non terminale
 - ✓ Paradigme « diviser pour régner »
- Chap-5: Graphes et arbres
- Chap-6: Arbres binaires de recherche

Chapitre 0 – Outils mathématiques

Outils mathématiques : analyse élémentaire

$(U_k)_{k \in \mathbb{N}}$ suite de terme général U_k , $k \in \mathbb{N}$

$(U_k)_{k \in K}$ famille d'index $K \subset \mathbb{N}$; suite extraite de $(U_k)_{k \in \mathbb{N}}$

$\sum_{k=p}^q U_k$ somme des termes U_k où k vérifie $p \leq k \leq q$ (entiers);
lorsque $p > q$, la somme est *vide* et vaut 0

$\prod_{k=p}^q U_k$ produit des termes U_k où k vérifie $p \leq k \leq q$ (entiers);
lorsque $p > q$, le produit est *vide* et vaut 1

Identité sur les sommes et produits

$$\sum_{k=p}^q U_k = \left(\sum_{k=p}^{q-1} U_k \right) + U_q = U_p + \left(\sum_{k=p+1}^q U_k \right)$$

Plus généralement, si $P(n)$ est un prédicat :

$$\sum_{k=p}^q U_k = \sum_{\substack{k=p \\ P(k) \text{ est vrai}}}^q U_k + \sum_{\substack{k=p \\ P(k) \text{ est faux}}}^q U_k$$

Un exemple très courant :

$$\sum_{k=1}^n U_k = \sum_{\substack{k=1 \\ k \text{ est pair}}}^n U_k + \sum_{\substack{k=1 \\ k \text{ est impair}}}^n U_k$$

Idem pour les produits.

Outils mathématiques : arithmétique

opérateurs usuels :

$$+ \quad - \quad \times \quad / \quad < \quad \leq \quad \text{mod}$$

$\lfloor x \rfloor$ partie entière inférieure (ou *plancher*) du réel x : le plus grand entier $\leq x$

$\lceil x \rceil$ partie entière supérieure (ou *plafond*) du réel x : le plus petit entier $\geq x$

$n!$ la *factorielle* de n :

$$n! := \prod_{i=1}^n i = 1 \times 2 \times 3 \times \cdots \times n$$

Parties entières et égalités

Pour tout réel x , pour tout entier n :

- $\lfloor x \rfloor = n \iff n \leq x < n + 1;$

- $\lceil x \rceil = n \iff n - 1 < x \leq n;$

- $\lfloor x + n \rfloor = \lfloor x \rfloor + n;$

- $\lceil x + n \rceil = \lceil x \rceil + n.$

Pour tout entier n :

- $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil.$

Parties entières et inégalités

Pour tout réel x , pour tout entier n :

$$\blacksquare \lfloor x \rfloor < n \iff x < n;$$

$$\blacksquare \lceil x \rceil \leq n \iff x \leq n;$$

$$\blacksquare n < \lceil x \rceil \iff n < x;$$

$$\blacksquare n \leq \lfloor x \rfloor \iff n \leq x.$$

Pour tous réels x et y :

$$\blacksquare \lfloor x \rfloor + \lfloor y \rfloor \leq \lfloor x + y \rfloor \leq \lfloor x \rfloor + \lfloor y \rfloor + 1;$$

$$\blacksquare \lceil x \rceil + \lceil y \rceil - 1 \leq \lceil x + y \rceil \leq \lceil x \rceil + \lceil y \rceil.$$

Fonctions usuelles

\ln fonction logarithme népérien (ou naturel), de base e

\log_a fonction logarithme de base a : $\log_a(x) = \ln x / \ln a$

\log fonction logarithme sans base précise, à *une constante multiplicative près*

\log_2 fonction logarithme binaire, de base 2 :

$$\log_2(x) = \ln x / \ln 2$$

Complexités

Définitions (complexités temporelle et spatiale)

- *complexité temporelle : (ou en temps) : temps de calcul ;*
- *complexité spatiale : (ou en espace) : l'espace mémoire requis par le calcul.*

Définitions (complexités pratique et théorique)

- *La complexité pratique est une mesure précise des complexités temporelles et spatiales pour un modèle de machine donné.*
- *La complexité (théorique) est un ordre de grandeur de ces couts, exprimé de manière la plus indépendante possible des conditions pratiques d'exécution.*

Un exemple

Problème (plus grand diviseur)

Décrire une méthode de calcul du plus grand diviseur autre que lui-même d'un entier $n \geq 2$.

Notons $pgd(n)$ le plus grand diviseur en question.

On a :

- $1 \leq pgd(n) \leq n - 1$;
- $pgd(n) = 1 \iff n$ est premier.

Chapitre 1 – Introduction et motivations

Introduction

- Un algorithme = une suite ordonnée d'opérations ou d'instruction écrites pour la résolution d'un problème donné.
- ➡ Algorithme = une suite d'actions que devra effectuer un automate pour arriver à partir d'un état initial, en un temps fini, à un résultat
- L'algorithmique désigne le processus de recherche d'algorithme

Structures de données

- ❑ Une structure de données indique la manière d'organisation des données dans la mémoire.
 - ❑ Le choix d'une structure de données adéquate dépend généralement du problème à résoudre.
 - ❑ Deux types de structures de données :
 - Statiques : Les données peuvent être manipulées dans la mémoire dans un espace statique alloué dès le début de résolution du problème. Ex : les tableaux
 - Dynamiques : On peut allouer de la mémoire pour y stocker des données au fur et à mesure des besoins de la résolution du problème. Ex: liste chaînée, pile, file, ...
- ➡ notion des pointeurs ➡ nécessité de la gestion des liens entre les données d'un problème en mémoire.

Qualités d'un bon algorithme

- **Correct:** Il faut que le programme exécute correctement les tâches pour lesquelles il a été conçu
- **Complet:** Il faut que le programme considère tous les cas possibles et donne un résultat dans chaque cas.
- **Efficace:** Il faut que le programme exécute sa tâche avec efficacité c'est à dire avec un coût minimal. Le coût pour un ordinateur se mesure en termes de temps de calcul et d'espace mémoire nécessaire.

Exemple : Calcul de la valeur d'un polynôme

- Soit $P(X)$ un polynôme de degré n
- $P(X) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$
- Où,
 - ✓ n : entier naturel
 - ✓ $a_n, a_{n-1}, \dots, a_1, a_0$: les coefficients du polynôme

- Première version

début

$P=0$

Pour i de 0 à n faire

$P = P + a_i * x^i$

Finpour

fin

Coût de l'algorithme :

- $(n+1)$ additions
- $(n+1)$ multiplications
- $(n+1)$ puissances

- Deuxième version

début

Inter=1

P =0

Pour i de 0 à N faire

$P = P + \text{Inter} * a_i$

$\text{Inter} = \text{Inter} * X$

finpour

Fin

Coût de l'algorithme :

- (n+1) additions

- 2(n+1) multiplications

- Troisième version

$$P(x) = (....(((a_n x + a_{n-1})x + a_{n-2})x + a_{n-3})....)x + a_0$$

début

$$P = a_n$$

Pour i de n-1 à 0 (pas = -1) faire

$$P = P * X + a_i$$

Finpour

Fin

Coût de l'algorithme :

- n additions

- n multiplications

➡ *Nécessité d'estimer le coût d'un algorithme avant de l'écrire et l'implémenter*

Chapitre 2 – Complexité et optimalité

Définitions

- ❖ 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.
- ❖ La complexité d'un algorithme est souvent déterminée à travers une description mathématique du comportement de cet algorithme.

Définitions

- ❖ On note D_n l'ensemble des données de taille n et $T(d)$ le coût de l'algorithme sur la donnée d .
- ❖ On définit 3 types de complexité :
 - **Complexité au meilleur :**
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 .
 - **Complexité au pire :**
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 .

$$T_{max}(n) = \max_{d \in D_n} T(d)$$

- **Complexité en moyenne :**
C'est la moyenne des complexités de l'algorithme sur des jeux de données de taille n .

$$T_{moy}(n) = \frac{\sum_{d \in D_n} T(d)}{|D_n|}$$

Définitions

❑ C'est l'analyse pessimiste ou **au pire** qui est généralement adoptée.

❑ En effet, de nombreux algorithmes fonctionnent la plupart du temps dans la situation la plus mauvaise pour eux.

➡ l'analyse au pire des cas donne une limite supérieure de la performance et elle garantit qu'un algorithme ne fera jamais moins bien que ce qu'on a établi.

➡ Un algorithme est dit **optimal** si sa complexité est la complexité **minimale** parmi les algorithmes de sa classe.

❑ Même si on s'intéresse quasi-exclusivement à la complexité en temps des algorithmes. Il est parfois intéressant de s'intéresser à d'autres ressources, comme la complexité en espace (taille de l'espace mémoire utilisé), la largeur de bande passante requise, etc. .

Notations mathématiques

- ❑ La notation O est celle qui est le plus communément utilisée pour expliquer formellement les performances d'un algorithme.
- ❑ Cette notation exprime la limite supérieure d'une fonction dans un facteur constant.

$$f = O(g) \Leftrightarrow \exists n_0, \exists c \geq 0, \forall n \geq n_0, f(n) \leq c \times g(n)$$

➡ La notation O reflète la courbe ou l'ordre croissance d'un algorithme.

Notations mathématiques

□ Les règles de la notation O sont les suivantes :

- Les termes constants : $O(c) = O(1)$

- Les constantes multiplicatives sont omises :

$$O(cT) = cO(T) = O(T)$$

- L'addition est réalisée en prenant le maximum :

$$O(T1) + O(T2) = O(T1 + T2) = \max(O(T1); O(T2))$$

- La multiplication reste inchangée mais est parfois réécrite d'une façon plus compacte :

$$O(T1)O(T2) = O(T1T2)$$


Exemple

❑ On suppose qu'on dispose d'un algorithme dont le temps d'exécution est décrit par la fonction $T(n) = 3n^2 + 10n + 10$.
L'utilisation des règles de la notation O nous permet de simplifier en :

$$O(T(n)) = O(3n^2 + 10n + 10) = O(3n^2) = O(n^2)$$

❑ Pour $n = 10$ nous avons :

- ✓ Temps d'exécution de $3(n^2)$: $3(10^2) / 3(10^2) + 10(10) + 10 = 73,2\%$
- ✓ Temps d'exécution de $10n$: $10(10) / 3(10^2) + 10(10) + 10 = 24,4\%$
- ✓ Temps d'exécution de 10 : $10 / 3(10^2) + 10(10) + 10 = 2,4\%$

❑ Le poids de $3n^2$ devient encore plus grand quand $n = 100$, soit 96,7%  on peut négliger les quantités $10n$ et 10 . Ceci explique les règles de la notation O .

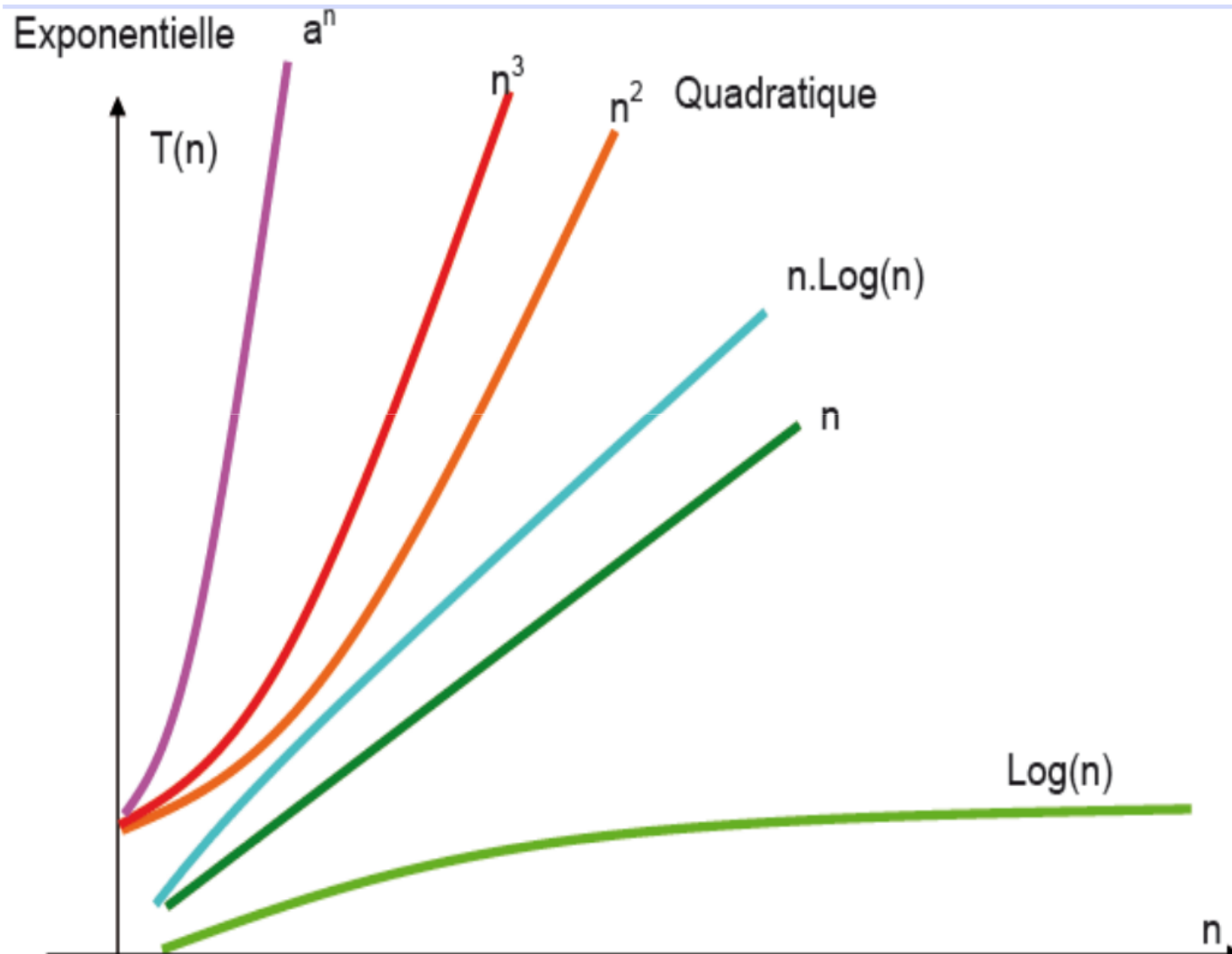
Classes de complexité

- ❑ Les algorithmes usuels peuvent être classés en un certain nombre de grandes classes de complexité.
- ❑ Les complexités les plus utilisées sont :
 - **Constante : $O(1)$** Accéder au premier élément d'un ensemble de données
 - **Logarithmique : $O(\log n)$** Couper un ensemble de données en deux parties égales, puis couper ces moitiés en deux parties égales, etc.
 - **Linéaire : $O(n)$** Parcourir un ensemble de données

Classes de complexité

- **Quasi-linéaire : $O(n \log n)$** Couper répétitivement un ensemble de données en deux et combiner les solutions partielles pour calculer la solution générale
- **Quadratique : $O(n^2)$** Parcourir un ensemble de données en utilisant deux boucles imbriquées
- **Polynomiale : $O(n^P)$** Parcourir un ensemble de données en utilisant P boucles imbriquées
- **Exponentielle : $O(2^n)$** Générer tous les sous ensembles possibles d'un ensemble de données

Classes de complexité



Calcul de la complexité

1. Cas d'une instruction simple : écriture, lecture, affectation

Dans le cas d'une suite d'instructions simples, on considère la complexité maximale.

```
x : entier;  
y : entier;  
x ← 0; /* instruction 1 en  $O(1)$  */  
y ← x2; /* instruction 2 en  $O(1)$  */
```

La complexité de cette séquence vaut $\max(O(1), O(1)) = O(1)$.

Calcul de la complexité

2. Cas d'un traitement conditionnel

```
Si (condition) Alors
  | Traitement1
Sinon
  | Traitement2
Fin Si
/*  $O(T_{condition}) + \max(O(T_{Traitement1}), O(T_{Traitement2}))$  */
```

3. Cas d'un traitement itératif : Boucle Tant Que

```
Tant que (condition) faire
  | Traitement
Fait
/* nombre d'itérations  $\times (O(T_{condition}) + O(T_{Traitement}))$  */
```

Calcul de la complexité

4. Cas d'un traitement itératif : Boucle Pour

Pour i de $indDeb$ à $indFin$ **faire**
 | Traitement
Fin Pour

$$\sum_{i=indDeb}^{indFin} O(T_{Traitement})$$

Chapitre 3 – Les algorithmes de Tri

Chapitre 4 – La récursivité

Définitions

- Un algorithme est dit récursif s'il est défini en fonction de lui-même.
 - La récursion est un principe puissant permettant de définir une entité à l'aide d'une partie de celle-ci.
- Chaque appel successif travaille sur un ensemble d'entrées toujours plus affinée, en se rapprochant de plus en plus de la solution d'un problème.

Evolution d'un appel récursif

- ❑ L'exécution d'un appel récursif passe par deux phases, la phase de descente et la phase de la remontée :
 - Dans la phase de descente, chaque appel récursif fait à son tour un appel récursif. Cette phase se termine lorsque l'un des appels atteint une condition terminale.
- ➡ condition pour laquelle la fonction doit retourner une valeur au lieu de faire un autre appel récursif.
- Ensuite, on commence la phase de la remontée. Cette phase se poursuit jusqu'à ce que l'appel initial soit terminé, ce qui termine le processus récursif.

Les types de récursivité

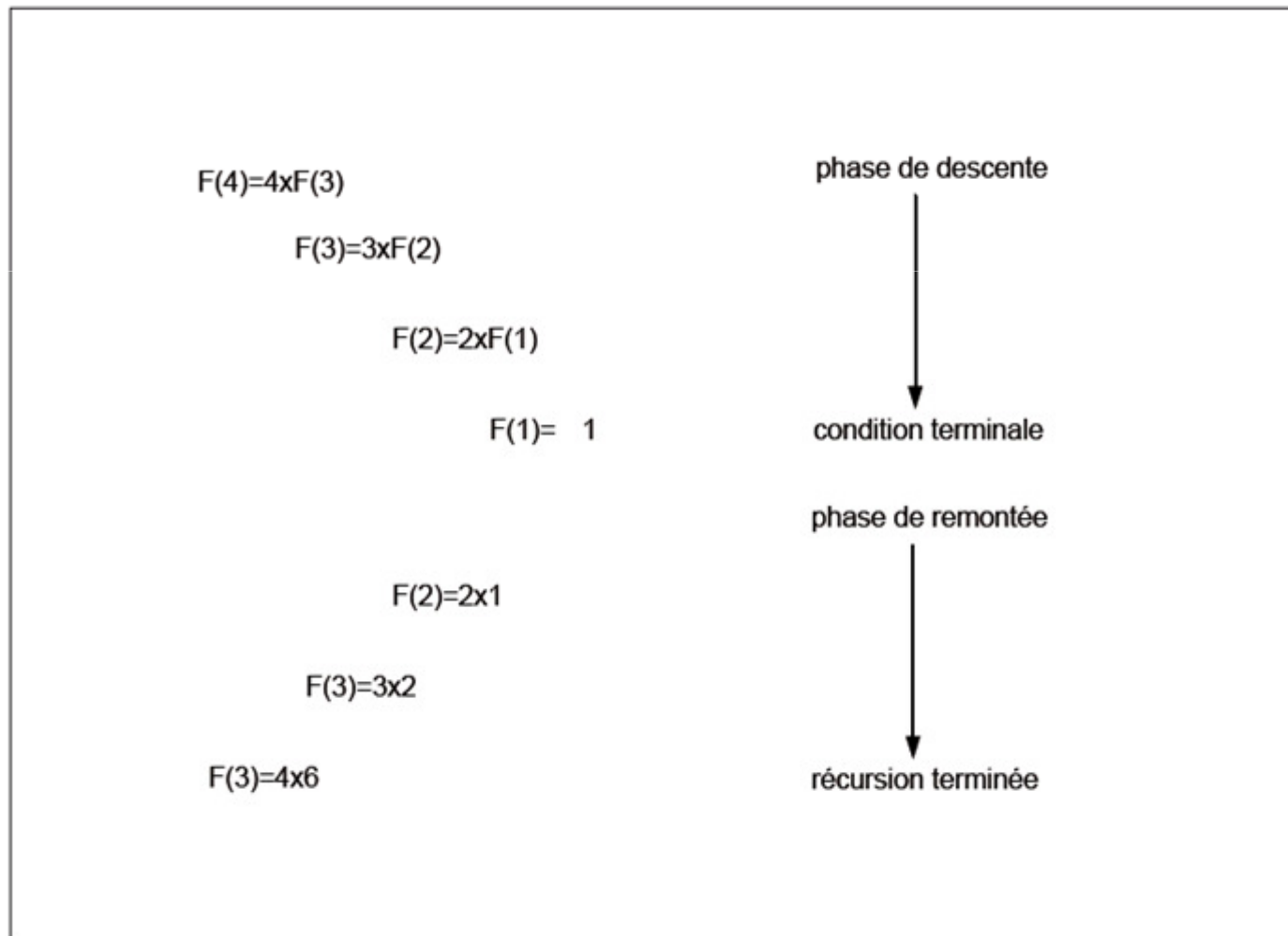
1/ La récursivité simple :

- récursivité simple → la fonction contient un seul appel récursif dans son corps.
- Exemple : la fonction factorielle

```
Fonction Factorielle( N : entier) : entier
    Si (N=1) Alors
        | Retourner 1 ;
    Fin Si
    Retourner N*Factorielle(N-1);
Fin
```

Les types de récursivité

- Trace d'exécution de la fonction factorielle (calcul de la valeur de 4!)



Les types de récursivité

2/ La récursivité multiple:

- récursivité multiple ➡ la fonction contient plus d'un appel récursif dans son corps
- Exemple : le calcul du nombre de combinaisons en se servant de la relation de Pascal :

$$C_n^p = \begin{cases} 1 & \text{si } p = 0 \text{ ou } p = n; \\ C_{n-1}^p + C_{n-1}^{p-1} & \text{sinon.} \end{cases}$$

```
Fonction Combinaisons( n : entier, p : entier) : entier
    Si (p = 0 OU p = n) Alors
        | Retourner 1;
    Fin Si
    Retourner Combinaisons (n - 1, p) + Combinaisons (n - 1, p - 1);
Fin
```

Les types de récursivité

3/ La récursivité mutuelle:

- Des fonctions sont dites mutuellement récursives si elles dépendent les unes des autres
- Par exemple la définition de la parité d'un entier peut être écrite de la manière suivante :

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

```
Fonction Pair( n : entier) : Boolean
| Si (n = 0) Alors
|   | Retourner vrai;
| Fin Si
| Retourner Impair(n - 1);
Fin
```

```
Fonction Impair( n : entier) : Boolean
| Si (n = 0) Alors
|   | Retourner faux;
| Fin Si
| Retourner pair(n - 1);
Fin
```

Les types de récursivité

4/ La récursivité imbriquée:

- Exemple : La fonction d'Ackermann

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

```
Fonction Ackermann( m : entier, n : entier) : entier
| Si (m = 0) Alors
|   | Retourner n+1 ;
| Fin Si
| Si (n = 0 ET m>0) Alors
|   | Retourner Ackermann(m - 1, 1);
| Sinon
|   | Retourner Ackermann(m - 1, Ackermann(m, n - 1));
| Fin Si
Fin
```

Récurtivité terminale vs. non terminale

- ❑ Une fonction récursive est dite récursive **terminale** si tous ses appels sont récursifs terminaux.
- ❑ Un appel récursif est terminal s'il s'agit de la dernière instruction exécutée dans le corps d'une fonction et que sa valeur de retour ne fait pas partie d'une expression.
- ❑ Les fonctions récursives terminales sont caractérisées par le fait qu'elles n'ont rien à faire pendant la phase de remontée.

Importance de l'ordre des appels récursifs

Proc. terminale

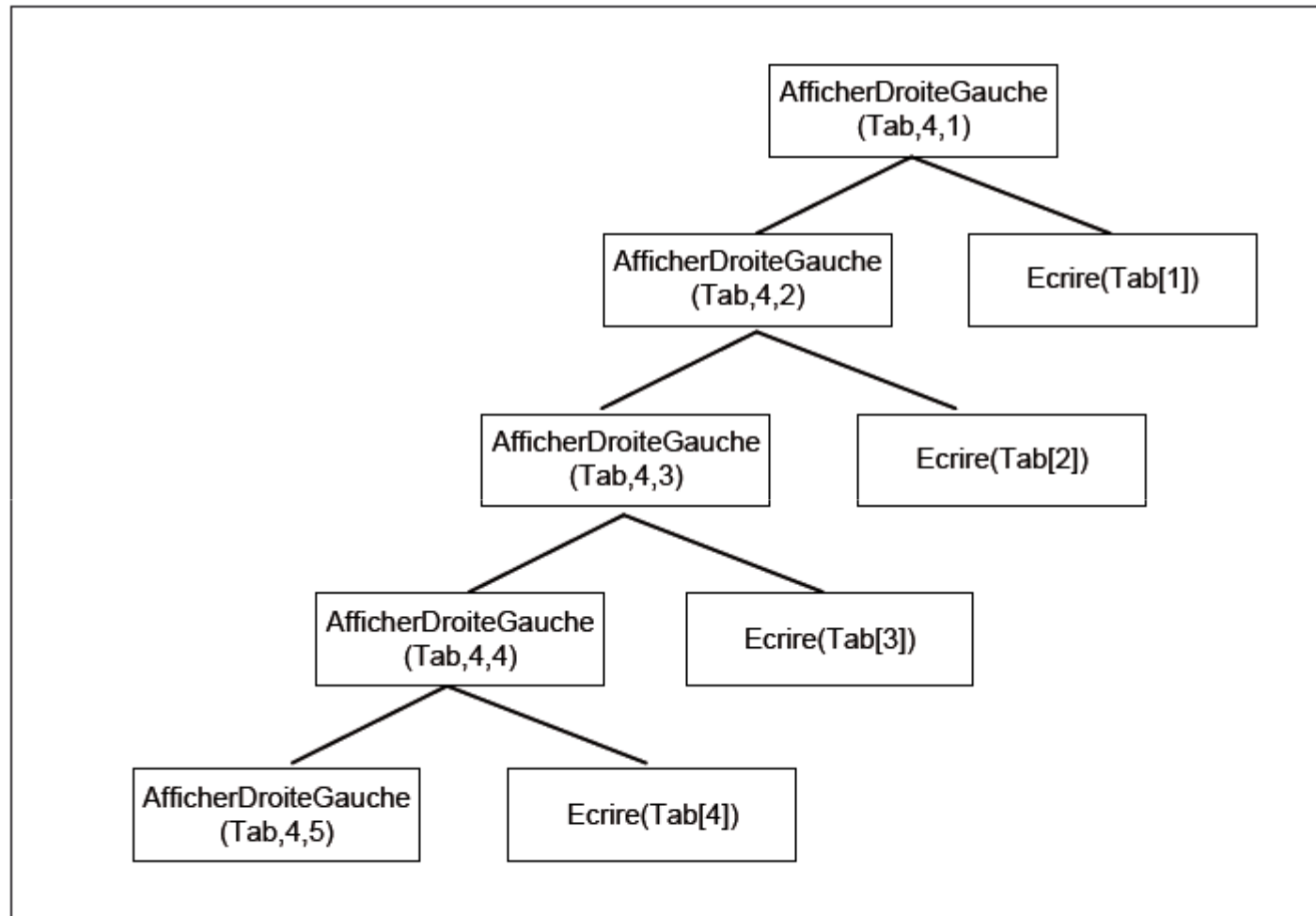
```
Procédure AfficherGaucheDroite ( Tab :  
Tableau entier, N : entier, i : entier)  
Si (i<=N) Alors  
Ecrire(Tab[i]) ; AfficherGaucheDroite  
(Tab,N,i+1) ;  
  
Fin Si  
Fin
```

Proc. non terminale

```
Procédure AfficherDroiteGauche ( Tab  
: Tableau entier, N : entier, i : entier)  
Si (i<=N) Alors  
AfficherDroiteGauche (Tab,N,i+1) ;  
Ecrire(Tab[i]) ;  
  
Fin Si  
Fin
```

➡ l'ordre des appels récursifs affecte la terminaison d'une fonction récursive

Importance de l'ordre des appels récursifs



Trace d'exécution de l'appel `AfficherDroiteGauche(Tab, 4, 1)`

Elimination de la récursivité

- ❑ Dérécursiver, c'est transformer un algorithme récursif en un algorithme équivalent ne contenant pas des appels récursifs.
- ❑ **Elimination de la récursivité terminale simple**
 - **Rappel** : Un algorithme est dit récursif terminal s'il ne contient aucun traitement après un appel récursif.
 - La récursivité terminale simple peut être remplacée par une solution itérative.

Elimination de la récursivité terminale simple

Algo. récursif

```
Procédure ALGOR( $X$ )  
Si ( $COND$ ) Alors  
   $TRAIT1$   
  ALGOR( $\beta(X)$ )  
Sinon  
   $TRAIT2$   
Fin Si  
Fin
```

Algo. itératif

```
Procédure ALGOI( $X$ )  
Tant que ( $COND$ ) faire  
   $TRAIT1$   
   $X \leftarrow \beta(X)$   
Fin tant que  
  
 $TRAIT2$   
Fin
```

- X est la liste des paramètres ;
- $COND$ est une condition portant sur X ;
- $TRAIT1$ est le traitement de base de l'algorithme (dépendant de X) ;
- $\beta(X)$ représente la transformation des paramètres ;
- $TRAIT2$ est le traitement de terminaison (dépendant de X).

Application : a est diviseur de b ?



Algo. récursif

```
Fonction Diviseur (a,b) : Bool  
Si (a <=0) Alors Retourner(Faux)  
Sinon  
Si (a>=b) Retourner (a=b)  
Sinon  
Retourner (Diviseur (a,b-a))  
Fin Si  
Fin Si  
Fin
```

Algo. itératif

```
Fonction Diviseur (a,b) : Bool  
Si (a <=0) Alors Retourner(Faux)  
Sinon  
Tant que (a<b) Faire  
b ← b-a  
Fin tant que  
Retourner (a=b)  
Fin Si  
Fin
```

Elimination de la récursivité non terminale simple

- ❑ Dans un algorithme récursif non terminal, l'appel récursif est suivi d'un traitement  il reste un traitement à reprendre après l'appel récursif
- ❑ Il va falloir donc sauvegarder, sur une pile, le contexte de l'appel récursif, typiquement les paramètres de l'appel engendrant l'appel récursif.
-  La récursivité non terminale simple peut être remplacée par une solution itérative utilisant une pile.

Elimination de la récursivité non terminale simple

Algo. récursif

```
Procédure ALGOR( $X$ )  
Si ( $COND$ ) Alors  
   $TRAIT1$   
  ALGOR( $\beta(X)$ )  
   $TRAIT2$   
Sinon  
   $TRAIT3$   
Fin Si  
Fin
```

Algo. itératif

```
Procédure ALGOI( $X$ )  
Pile.init()  
Tant que ( $COND$ ) Faire  
   $TRAIT1$   
  Pile.empiler( $X$ )  
   $X \leftarrow \beta(X)$   
Fin Tant que  
   $TRAIT3$   
Tant que (Non_vide_pile()) Faire  
  Pile.dépiler( $U$ )  
   $TRAIT2$   
Fin Tant que  
Fin
```

Elimination de la récursivité non terminale simple

Algo. récursif

```
Procédure AfficherDroiteGauche ( Tab
: Tableau entier, N : entier, i : entier)
Si (i<=N) Alors
AfficherDroiteGauche (Tab,N,i+1) ;
Ecrire(Tab[i]) ;
Fin Si
Fin
```

TRAIT1 = \emptyset

TRAIT2 = Ecrire(Tab[i])

TRAIT3 = \emptyset

Algo. itératif

```
Procédure ALGOI(X)
Pile.init()
Tant que (i<=N) Faire
/* TRAIT1 */
Pile.empiler(Tab[i])
i ← i+1
Fin Tant que
/* TRAIT3 */
Tant que (Non_vide_pile()) Faire
Ecrire(Pile.dépiler() )
/* TRAIT2 */
Fin Tant que
Fin
```

Exemple : Tours de Hanoi

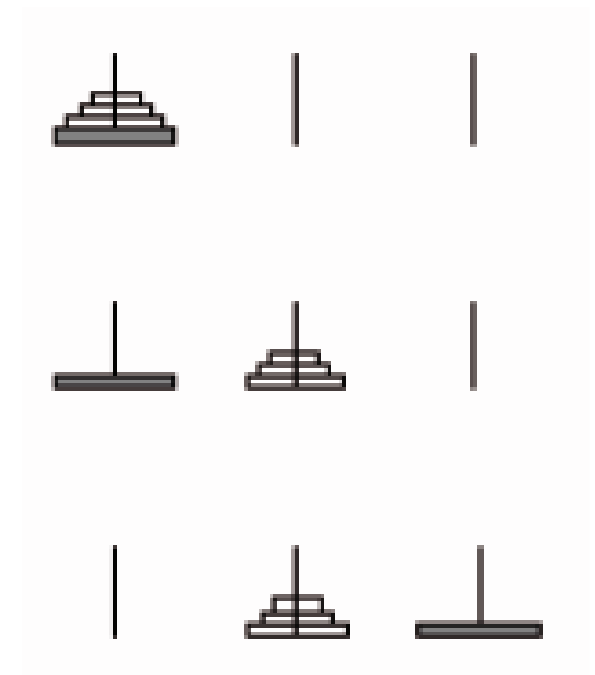
❑ Problème :

Le jeu est constitué d'une plaquette de bois où sont plantées trois tiges numérotées 1, 2 et 3. Sur ces tiges sont empilés des disques de diamètres tous différents. Les seules règles du jeu sont que l'on ne peut déplacer qu'un seul disque à la fois, et qu'il est interdit de poser un disque sur un disque plus petit. Au début, tous les disques sont sur la tige 1 (celle de gauche), et à la fin ils doivent être sur celle de droite.

Exemple : Tours de Hanoi

□ Résolution:

On suppose que l'on sait résoudre le problème pour $(n-1)$ disques. Pour déplacer n disques de la tige 1 vers la tige 3, on déplace les $(n-1)$ plus petits disques de la tige 1 vers la tige 2, puis on déplace le plus gros disque de la tige 1 vers la tige 3, puis on déplace les $(n-1)$ plus petits disques de la tige 2 vers la tige 3.



Exemple : Tours de Hanoi

□ Algorithme

Procédure Hanoi (n , *départ*, *intermédiaire*, *destination*)

Si $n > 0$ Alors

Hanoi ($n-1$, *départ*, *destination*, *intermédiaire*)

déplacer un disque de *départ* vers *destination*

Hanoi ($n-1$, *intermédiaire*, *départ*, *destination*)

Fin Si

Fin

Exemple : Tours de Hanoi

□ Trace d'exécution pour $n=3$

L'appel à `Hanoi(3,1,2,3)` entraîne l'affichage de :

1. Déplace un disque de la tige 1 vers la tige 3
2. Déplace un disque de la tige 1 vers la tige 2
3. Déplace un disque de la tige 3 vers la tige 2
4. Déplace un disque de la tige 1 vers la tige 3
5. Déplace un disque de la tige 2 vers la tige 1
6. Déplace un disque de la tige 2 vers la tige 3
7. Déplace un disque de la tige 1 vers la tige 3

Exemple : Tours de Hanoi

- ❑ Calcul de la complexité :
- ❑ On compte le nombre de déplacements de disques effectués par l'algorithme Hanoi invoqué sur n disques.
- ❑ On trouve :
 - $T(n) = T(n-1) + 1 + T(n-1)$
 - $T(n) = 2T(n-1) + 1$
- ➡ $T(n) = 2^n - 1$
- ➡ Complexité exponentielle

Paradigme « diviser pour régner »

- ❑ De nombreux 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** » parcourt trois étapes à chaque appel récursif à savoir :
 - **Diviser** : le problème en un certain nombre de sous-problèmes de taille moindre ;
 - **Régner** : sur les sous-problèmes en les résolvant d'une façon récursive ou le résoudre directement si la taille d'un sous-problème est assez réduite ;
 - **Combiner** : les solutions des sous-problèmes en une solution globale pour le problème initial.

Exemple 1

Recherche de l'indice du maximum dans un tableau d'entiers

Fonction maximum (Tab , indDeb, indFin)

Si (indDeb = indFin) alors

retourner (indDeb)

Sinon

$m = (\text{indDeb} + \text{indFin}) / 2$ // **division du problème en 2 sous-problèmes**

$k1 = \text{maximum}(\text{Tab}, \text{indDeb}, m)$ // **régner sur le 1er sous-problème**

$k2 = \text{maximum}(\text{Tab}, m+1, \text{indFin})$ // **régner sur le 2ème sous-problème**

if($\text{Tab}[k1] > \text{Tab}[k2]$) // **combiner les solutions**

retourner (k1)

else

retourner (k2)

FinSi

FinSi

Fin



$$T(n) = 2 T(n/2) + \text{cte}$$




Exemple2 : multiplication de matrices carrées

- Dans cet exemple, on se propose de multiplier 2 matrices carrées A et B de taille $n * n$ chacune, où n est une puissance exacte de 2. C est la matrice résultante.
- Si on décompose les matrices A, B et C en sous-matrices de taille $n/2 * n/2$, l'équation $C = AB$ peut alors se réécrire :

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

- Le développement de cette équation donne :
 $r = ae+bf$; $s = ag+bh$; $t = ce+df$ et $u = cg+dh$

Exemple2 : multiplication de matrices carrées

- Chacune de ces quatre opérations correspond à :
 - deux multiplications de matrices carrées de taille $n/2$ 
 - $2T(n/2)$
 - et une addition de telles matrices  $O(n^2/4)$
- A partir de ces équations on peut dériver un algorithme
- « diviser pour régner » dont la complexité est donnée par la récurrence :
 $T(n) = 8T(n/2) + O(n^2)$

Analyse des algorithmes « Diviser pour Régner »

- ❑ On peut souvent donner une relation de récurrence qui décrit le temps d'exécution d'un algorithme « diviser pour régner » en fonction des temps d'exécution des appels récurifs résolvant les sous-problèmes associés de taille moindre.
- ❑ Cette récurrence se décompose suivant les trois étapes du paradigme de base :
 - 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 $O(1)$.
 - Sinon, on divise le problème en a sous-problèmes chacun de taille n/b de la taille du problème initial. Le temps d'exécution total se décompose alors en trois parties :
 - ✓ $D(n)$: le temps nécessaire à la division du problème en sous-problèmes.
 - ✓ $aT(n/b)$: le temps de résolution des a sous-problèmes.
 - ✓ $C(n)$: le temps nécessaire pour construire la solution finale à partir des solutions aux sous-problèmes.

Analyse des algorithmes « Diviser pour Régner »

- La relation de récurrence prend alors la forme :

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

- Soit la fonction $f(n)$ la fonction qui regroupe $D(n)$ et $C(n)$. La fonction $T(n)$ est alors définie :

$$T(n) = a.T(n/b) + f(n)$$

$$T(n) = a.T(n/b) + c.n^k$$

Résolution des récurrence des algorithmes

« Diviser pour Régner »

□ Théorème de résolution de la récurrence :

- si $a > b^k \longrightarrow T(n) = O(n^{\log_b a})$
- si $a = b^k \longrightarrow T(n) = O(n^k \log_b n)$
- si $a < b^k \longrightarrow T(n) = O(f(n)) = O(n^k)$

□ Résolution de la relation de récurrence pour l'exemple de la multiplication de matrices :

- $T(n) = 8 T(n/2) + O(n^2)$
- $a = 8, b = 2, k = 2 \longrightarrow a > b^k$
 - $\longrightarrow \log_b a = 3$
 - $\longrightarrow T(n) = O(n^3)$

Analyse des algorithmes « Diviser pour Régner »

- Complexité de l'algorithme de recherche du maximum: $T(n) = 2 T(n/2) + 3$
- $a = 2$, $b = 2$, $k = 0 \Rightarrow a > b^k$
- $\log_b a = 1$
- $T(n) = O(n)$

Application : algorithme de recherche dichotomique

Fonction RechDicho(Tab :Tableau, borneinf :entier, bornesup :entier, elem :entier) : bool

Si (borneinf<=bornesup) alors

mil = (*borneinf+bornesup*) DIV 2 ;

Si (Tab[mil]=elem) Alors

retourner (vrai)

Sinon

Si (Tab[mil]>elem) Alors

Retourner (RechDicho(Tab, borneinf, *mil-1*, elem))

Sinon

Retourner(RechDicho(Tab, *mil+1*, bornesup, elem))

Fin Si

Fin Si

Sinon

Retourner (Faux)

FinSi

Application : algorithme de recherche dichotomique

□ Analyse de la complexité :

- $T(n) = T(n/2) + \text{cte}$
- $a = 1, b = 2, k = 0 \longrightarrow a = b^k$
- $a = b^k \longrightarrow T(n) = O(n^k \log_b n)$
- $T(n) = O(\log_2 n)$

Exemples d'application

□ Exemple 1 :

$$T(n) = 9T(n/3) + n$$

- $a = 9, b = 3, k = 1$

$$a > b^k$$

- $\log_b a = 2$
- $T(n) = O(n^2)$

□ Exemple 2 :

$$T(n) = T(2n/3) + 1$$

- $a = 1, b = 3/2, k = 0$
- $a = b^k$
- $T(n) = O(n^k \log n) = O(\log n)$

Exemples d'application

- Exemple 3 : $T(n) = 3T(n/4) + n \log n$

- $a = 3$, $b = 4$, $k = ??$

$$\text{or } n < n \log n < n^2 \quad \longrightarrow \quad 1 < k < 2$$

$$\longrightarrow 4 < b^k < 16 \quad \longrightarrow \quad a < b^k$$

- $T(n) = O(f(n)) = O(n \log n)$

Autres résolutions de récurrence

Equations de récurrence linéaires:

$$T(n) = aT(n-1) + f(n) \quad \longrightarrow \quad T(n) = a^n \left(T(0) + \sum_{i=1}^n \frac{f(i)}{a^i} \right)$$

Exemple : Les Tours de Hanoi

$$T(n) = 2 T(n-1) + 1 \quad \longrightarrow \quad T(n) = 2^n \left(0 + \sum_{i=1}^n \frac{1}{2^i} \right) = 2^n - 1$$

Autres résolutions de récurrence

- Equations de récurrence linéaires sans second membre ($f(n) = \text{cte}$)

$$T(n) - a_1 T(n-1) - a_2 T(n-2) - \dots - a_k T(n-k) = \text{cte}$$

- A une telle équation, on peut associer un polynôme:

$$P(x) = x^k - a_1 x^{k-1} - a_2 x^{k-2} - \dots - a_k$$

- La résolution de ce polynôme nous donne m racines r_i (*avec $m \leq k$*).
- La solution de l'équation de récurrence est ainsi donnée par :

$$T(n) = c_1 r_1^n + c_2 r_2^n + \dots + c_m r_m^n$$

 Cette solution est en général exponentielle

Autres résolutions de récurrence

Exemple : La suite de Fibonacci

$$F(n) = \begin{cases} 1, & n=0 \text{ ou } n=1. \\ F(n-1) + F(n-2), & n \geq 2. \end{cases}$$

$$T(n) = T(n-1) + T(n-2)$$

On pose $P(x) = X^2 - X - 1$

$$\begin{aligned} \rightarrow r_1 &= \frac{1+\sqrt{5}}{2} \text{ et } r_2 = \frac{1-\sqrt{5}}{2} \\ \rightarrow T(n) &= a r_1^n + b r_2^n = a \left(\frac{1+\sqrt{5}}{2} \right)^n + b \left(\frac{1-\sqrt{5}}{2} \right)^n \\ \rightarrow T(n) &= O\left(\left(\frac{1+\sqrt{5}}{2} \right)^n \right) \end{aligned}$$

Chapitre 5 – Graphes et arbres