Récursivité et Paradigme Diviser pour Régner

Master : Cyber Sécurité et Intelligence Artificielle
CSIA

Année universitaire : 2025 / 2026

Plan du chapitre

- Partie 1: La récursivité
- Partie 2: La paradigme « Diviser pour régner »
- Partie 3: La dérécursivation

PARTIE1: LA RÉCURSIVITÉ

PLAN DE LA PARTIE I

- Définitions
- Évolution d'un appel récursif
- Types de la récursivité
- Récursivité terminale vs non terminale
- Complexité des algorithmes récursifs
- Écrire un algorithme récursif
- Exemple: Tours de Hanoi

DÉFINITIONS

- Un algorithme est dit récursif s'il est défini en fonction de lui-même.
- Un programme est récursif est un programme qui s'appelle lui-même.

DÉFINITIONS

- Avantages :
  - La récursivité est une notion importante de la programmation
  - Elle permet de régler des problèmes complexes d'une manière très rapide.
  - Un mécanisme naturel qui permet a un sous-programme a faire appel a lui-même.
  - Outil simplifiant, lisible, efficace et souvent sous estimé.
- Inconvénients :
  - Cependant, une méthode avec laquelle il est facile de se perdre et d'avoir des résultats imprévisibles ou erronés.

Évolution d’un appel récursif

```
Procédure Récursivité(Paramètres)
Var : Déclaration des variables locales
Si Test d'arrêt alors
        Instruction du point d'arrêt
Sinon
        Instructions
        Récursivité(Paramètres changées)
Fin si
Fin
```

Évolution d’un appel récursif

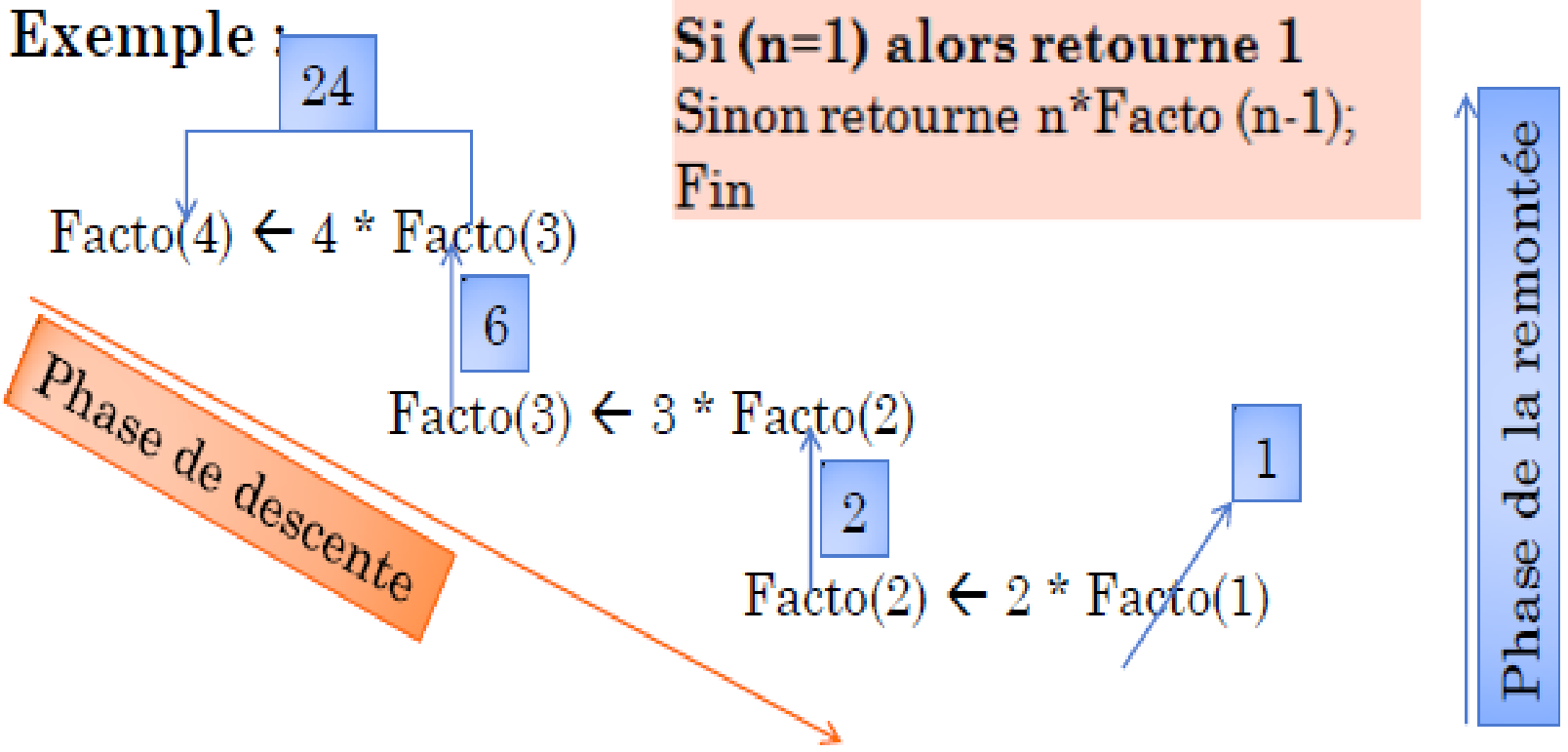
- Tout sous-programme récursif doit comporter un test d'arrêt.
- Le test d'arrêt indique quand est-ce que :
  - on exécute les instruction du point d'arrêt ;
  - on arrête les appels récursifs du sous programme.
- Au risque d'entrainer des appels infinis, les paramètres de l'appel récursif doivent changer à chaque appel.
- Grâce a ces changements de paramètres :
  - l'ordinateur va rencontrer un ensemble de paramètres vérifiant le test d'arrêt.
  - le sous-programme récursif va atteindre le point terminal.

Évolution 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.
- En arrivant à la condition terminale, on commence la phase de la remontée qui se poursuit jusqu'à ce que l'appel initial soit terminé, ce qui termine le processus récursif.

Évolution d'un appel récursif

```
Facto (n: entier): entier
Début
Si (n=1) alors retourne 1
Sinon retourne n*Facto (n-1);
Fin
```



Évolution d’un appel récursif

- Les problèmes récursifs peuvent toujours être résolus a l'aide d'algorithmes séquentiels (non-récursif).
  - Mais, souvent, la solution est plus complexe.
  - Inversement, certains problèmes itératifs (séquentiel) peuvent être résolus à l'aide de sous-programmes récursifs

Exemple de transformation de boucle en un algorithme récursif

Calcul de la somme des N premiers naturels

```
Fonction Addition(N : entier) : entier
Var : S, i : entier
S ← 0
Pour i ← 1 a N Faire
      S ← S + i
Fin Pour
Renvoyer S
Fin
```

```
Fonction Addition(N : entier) : entier
Si (N > 1) Alors
Renvoyer N + Addition(N - 1)
Sinon
Renvoyer 1
Fin Si
Fin
```

Types de récursivité

- **La récursivité simple** où l'algorithme contient un seul appel récursif dans son corps.
- Exemple : la fonction factorielle

```
Facto (n: entier): entier
Début
    Si (n=1) alors retourne 1
    Sinon retourne n*Facto (n-1);
 Fin
```

Types de récursivité

- **La récursivité multiple** où l'algorithme contient plus d'un appel récursif dans son corps.
- Exemple : Suite de Fibonacci

$$U_n \begin{cases} U_0 = U_1 = 1 \\ U_n = U_{n-1} - U_{n-2} \quad n \geq 2 \end{cases}$$

```
Fonction Fib(N : entier) : entier
Si (N  <=  1) Alors
   Renvoyer 1
Sinon
   Renvoyer Fib(N-1) -   Fib(N-2)
Fin Si
Fin
```

Types de récursivité

- **La récursivité mutuelle:** Des modules sont dits mutuellement récursifs s'ils dépendent les unes des autres.
- Exemple : Par exemple, deux fonctions A(x) and B(x) définies comme suit :

$$A(x) = \begin{cases} 1 & \text{si } x<=1 \\ B(x+2) & \text{si } x>1 \end{cases}$$

$$B(x) = A(x-3)+ 4$$

Types de récursivité

- **La récursivité imbriquée** consiste à faire un appel récursif à l'intérieur d'un autre appel récursif.
- 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
```

Récursivité terminale vs non terminale

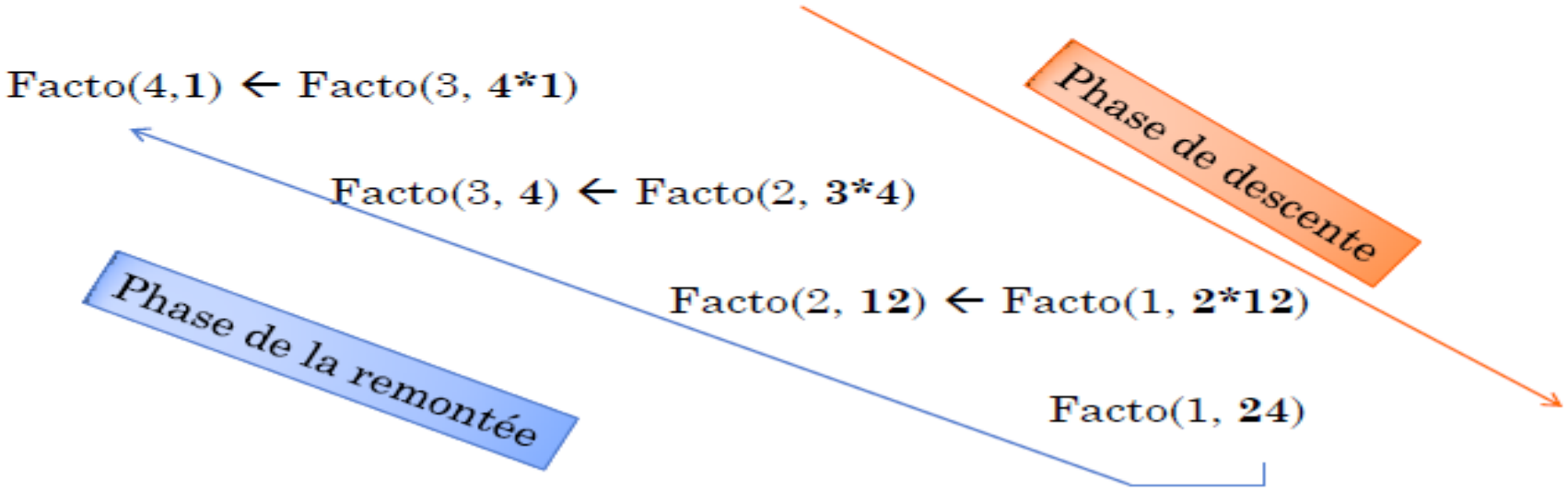
- Un module récursif est dit **terminal** si aucun traitement n'est effectué à la remontée d'un appel récursif (sauf le retour du module).
- Un module récursif est dit **non terminal** si le résultat de l'appel récursif est utilisé pour réaliser un traitement (en plus du retour du module).

Récursivité terminale vs non terminale

Récusivité non terminale	Récursivité Terminal
Facto (n: entier): entier Début Si (n=1) alors retourne 1 Sinon retourne n*Facto (n-1); Fin	**Facto (n: entier, resultat: entier): entier** Début si (n = 1) alors **retourne resultat;** sinon **retourne Facto (n-1, n * resultat);** Fin **// la fonction doit être appelée en mettant resultat à 1**

Récursivité terminale vs non terminale

```
Facto (n: entier, resultat: entier): entier
Début
si (n = 1) alors
retourne resultat;
sinon
retourne Facto (n-1, n * resultat);
Fin
// la fonction doit être appelée en mettant
resultat à 1
```



Complexité des algorithmes récursifs

- La complexité d'un algorithme récursif se fait par la résolution d'une équation de récurrence en éliminant la récurrence par substitution de proche en proche.
- Exemple : la fonction factorielle (T(n) le temps d'exécution nécessaire pour un appel à Facto(n))

Complexité des algorithmes récursifs

Facto (n: entier): entier

Début

Si (n=1) [O (1)]

alors retourne 1 [O (1)]

Sinon retourne n*Facto (n-1);

[T (n) dépend de T(n-1)]

Fin

$$T(n) = \begin{cases} a & si \quad n = 1 \\ b + T(n-1) & sinon \end{cases}$$

Complexité des algorithmes récursifs

- Pour calculer la solution générale de cette équation, on peut procéder par substitution :

$$
\begin{aligned}
T(n) &= b + T(n-1) \\
&= b + [b + T(n-2)] \\
&= 2b + T(n-2) \\
&= 2b + [b + T(n-3)] \\
&= \dots \\
&= ib + T(n-i) \\
&= ib + [b + T(n-i+1)] \\
&= \dots \\
&= (n-1)b + T(n-n+1) = nb - b + T(1) = nb - b + a \\
T(n) &= nb - b + a
\end{aligned}
$$

O (T) = O (n)

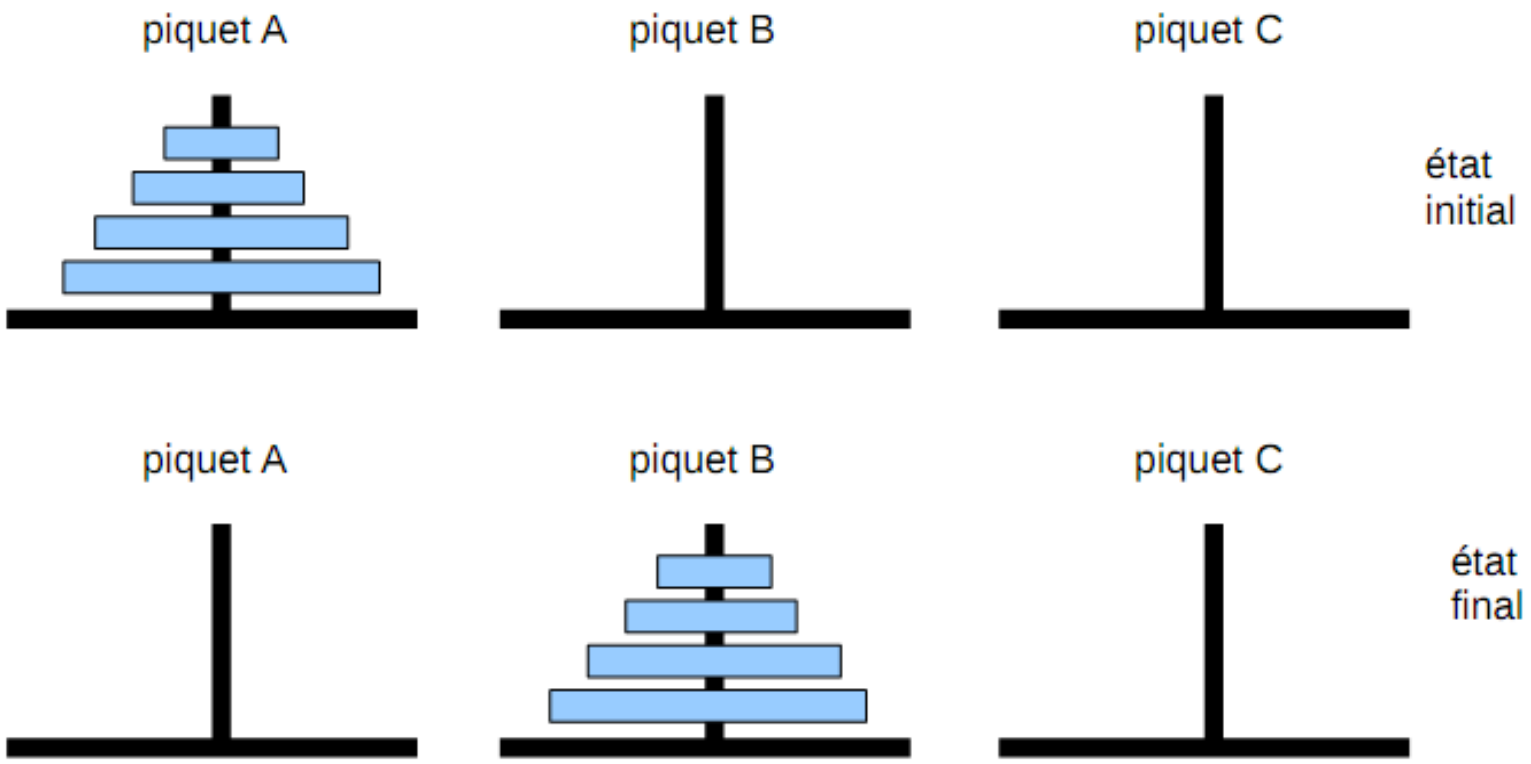
Écrire un algorithme récursif

- Dans un module récursif (procédure ou fonction) les paramètres doivent être clairement spécifiés
- Dans le corps du module il doit y avoir:
  - un ou plusieurs cas particuliers: ce sont les cas simples qui ne nécessitent pas d'appels récursifs
  - un ou plusieurs cas généraux: ce sont les cas complexes qui sont résolus par des appels récursifs
- L'appel récursif d'un cas général doit toujours mener vers un des cas particuliers

Exemple: Tours de Hanoi

- Déplacer n disques (empilés les uns sur les autres) d'un piquet (A) vers un autre piquet (B) en utilisant un troisième (C) comme intermédiaire **(ce sont des piles, on ne peut accéder qu'au disque du sommet)**
- **un disque ne peut jamais être placé au dessus d'un autre plus petit**

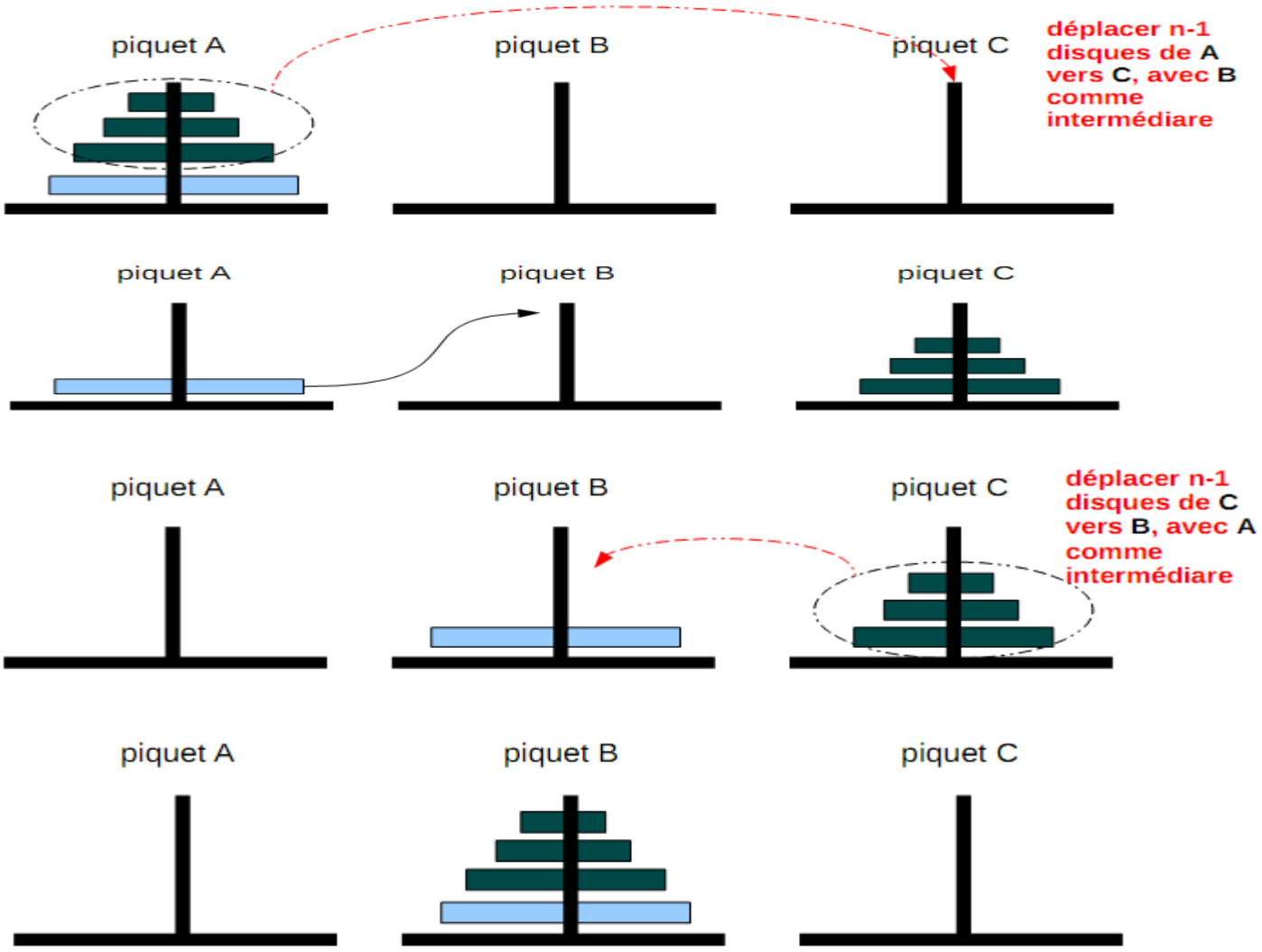
Exemple: Tours de Hanoi



Exemple: Tours de Hanoi

- Il faut pouvoir exprimer :
  - « le déplacement de n disques » en fonction d'un ou de plusieurs « déplacement de m disques » (avec m < n).
  - Par exemple (m=n-1) si on avait une solution pour déplacer n-1 disques, comment l'utiliser pour déplacer n disques ?

Exemple: Tours de Hanoi



Exemple: Tours de Hanoi

- Pour Transférer n disques de A vers B avec C comme intermédiaire, il faut :
- **Cas particulier (n = 1)**
  - déplacer le disque de X vers Y.
- **Cas général (n > 1)**
  - Transférer les n-1 premiers disques de A vers C, en utilisant B comme intermédiaire
  - déplacer l'unique disque qui reste dans A vers B
  - Transférer les n-1 premiers disques de C vers B, en utilisant A comme intermédiaire

Exemple: Tours de Hanoi

Algorithme 1

Hanoi(n:entier; A,B,C : caractères)
Debut
SI (n = 1)
écrire(« déplacer un disque de », A, « vers », B);
SINON
Hanoi (n-1, A, C, B);
écrire(« déplacer un disque de », A, « vers », B);
Hanoi(n-1, C, B, A);
FINS I
Fin

Exemple: Tours de Hanoi

Algorithme 2

Hanoi(n:entier; A,B,C : caractères)

SI (n > 0)

Hanoi (n-1, A, C, B);

écrire(« déplacer un disque de », A, « vers », B);

Hanoi(n-1, C, B, A);

FSI

Exemple: Tours de Hanoi

Hanoi(n:entier; A,B,C : caractères) **[T(n)]**

SI (n > 0)

Hanoi (n-1, A, C, B); **[T(n-1)]**

écrire(« déplacer un disque de », A, « vers », B); **[O(1)]**

Hanoi(n-1, C, B, A); **[T(n-1)]**

FSI

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

$2*2T(n-2) + 2a + a = \ldots\ldots\ldots = 2^n T(0) +$

$(2^0+2^1+2^2+\ldots\ldots+ 2^{n-1})\, a = 2^n\, 0 + (2^n - 1)\, a$

$O(T) = 2^n$