Programming et Algorithme III

Plan/objectifs du cours

- **Plan**
  - Algorithmes, Preuve, Complexité
  - Récursivité
  - Types abstraits, listes, ensembles
  - Classements (Tris)
  - Recherches
  - Graphes et leurs traitements
- **Objectifs**
  - Hard: Maîtriser les techniques scientifiques
  - **Soft: Comprendre des discours en français**

Références et sources

Références principales

- A.V. Aho, J.E. Hopcroft et J.D. Ullman, Structures de données et algorithmes, InterEditions, 1988.
- D. Beauquier, J. Berstel et Ph. Chrétienne, Éléments d'algorithmique, Masson, 1992, 463 pages.
- Cours adapté à partir des cours de Maxime Crochemore
- Structures de données, Licence d'informatique, UMLV
- Algorithmique Graphes et automates , Licence d'informatique, UMLV

Bibliothèque

- Beauquier Berstel, Chrétienne Masson, *Éléments d'algorithmique*, 2005.
- Sedgewick, *Algorithmes en C*, 1992.
- Cormen, Leiserson, Rivest, Dunod, *Algorithmes*, 1994.

Cours d’aujourd’hui

- Les définitions préliminaires
- Comment évaluer un algorithme? (Complexité)
- Comment calculer la complexité d’un algorithme?
- Algorithme récursive
  - Tours de Hanoï
  - Jeu des chiffres

Préliminaires

Algorithmique

- **Conception de méthodes pour la résolution de problèmes**
  - Description des données
  - Description des méthodes
  - Preuve de bon fonctionnement
- **Complexité des méthodes**
  - Efficacité : temps de calcul, espace nécessaire, ...
  - Complexité intrinsèque, optimalité
  - Solutions approchées
- **Réalisation - implémentation**
  - Organisation des objets
  - Opérations élémentaires

Modèle Mathématique	Algorithme informel ou abstrait

↓

Types de données abstraits	Algorithme formalisé *Super-Pascal*

↓

Structures de données	Programme *Pascal, C, Java, ...*

Définition d'algorithme

Al Khowarizmi, Bagdad IXè siècle.

Encyclopedia Universalis :

> « Spécification d'un schéma de calcul sous forme d'une suite finie d'opérations élémentaires obéissant à un enchaînement déterminé. »

DONNÉES → RÉSULTATS, ACTIONS

Composition d 'un nombre fini d'opérations dont chacune est :

- définie de façon rigoureuse et non ambiguë
- effective sur les données adéquates
  (exécution en temps fini)

A distinguer:

- **Spécification d ’un algorithme :**
  - ce que fait l’algorithme
  - cahier des charges du problème à résoudre
- **Expression d ’un algorithme :**
  - comment il le fait
  - texte dans un langage de type Pascal / C
- **Implémentation d ’un algorithme :**
  - traduction du texte précédent
  - dans un langage de programmation réel

Éléments de méthodologie

- Programmation structurée
- Modularité
- Programmation fonctionnelle
- Récursivité
- Types abstraits de données (héritage, polymorphism, …)
- Objets
- Réutilisabilité du code

Comment évaluer un algorithme?

- L'algorithme doit être correct
- Coût d'algorithm => Complexité
  - Temporelle
  - Espace

Temps de calcul

- Complexité en temps: T(*algo*, *d*) = temps d'exécution de l'algorithme *algo* appliqué aux données *d.*
- T(*algo, d*) dépend en général de *d*.
- **Complexité au pire**
  - T_{MAX} (*algo, n*) = max {T(*algo, d*) ; *d* de taille *n* }
- **Complexité au mieux**
  - T_{MIN} (*algo, n*) = min {T(*algo, d*) ; *d* de taille *n* }
- **Complexité en moye**
  - T_{MOY} (*algo, n*) = ∑ *p*
  - *p(d)* = probabilité

- Il est très difficile de prévoir le temps de calcul d'un programme.
- En revanche, on peut très bien **prévoir comment ce temps de calcul augmente quand la donnée augmente**

$\boxed{T_{MIN}(algo, n) \leq T_{MOY}(algo, n) \leq T_{MAX}(algo, n)}$

Temps de calcul dans les structures de contrôle

- Éléments de calcul de la complexité en temps
  - T(opération élémentaire) = constant :
  - T(**si** *C* **alors** *I* **sinon** *J*) $\leq$ T(*C*) + max (T(*I*), T(*J*))
  - T(**pour** $i \leftarrow e_1$ **à** e_2 **faire** I_i) = T(e_1) + T(e_2) + Σ T(I_i)
- procédures récursives→ solution d'équations de récurrence
- Complexité? à introduire à la suite.

Exemple: Tri (Classement)

Suite s = (7, 3, 9, 4, 3, 5)

Suite classée (en ordre croissant) c = (3, 3, 4, 5, 7, 9)

- **Spécification**
  - suite → suite classée
- **Méthode informelle** (Bulles)
  - **tant que** il existe i tel que si > si +1
  - **faire** échanger (si, si +1)
- **Opérations élémentaires**
  - comparaisons
  - transpositions d’éléments consécutifs

Formalisation

Algorithme Bulles1

répéter

{ i : = 1 ;

tant que $i < n$ et $s_i \leq s_{i+1}$ **faire** i := i + 1 ;

si $i < n$ **alors**

{échanger (s_i, s_{i+1}) ; inversion := vrai ; }

sinon

inversion := faux ;}

tant que inversion = vrai ;

Algorithme Bulles2

{ **pour** j := n-1 **à** 1 **pas** -1 **faire**
  pour i := 1 **à** j **faire**
   si $s_i > s_{i+1}$ **alors** échanger(s_i, s_{i+1}) ;
}

Temps d 'exécution T_{MAX} (Bulles2, n)

j donné $\longrightarrow$ temps $\leq k.j + k'$

$$T_{MAX}(\text{Bulles2}, n) \leq \sum_{j} (k.j + k') + k''$$

$$\leq k.\frac{n(n-1)}{2} + k'.(n-1) + k''$$

Nb comparaisons = $\frac{n(n-1)}{2}$ Nombre d'échanges $\leq \frac{n(n-1)}{2}$

Comparaisons de complexités

Retenir

- On ne veut pas comparer le nombre d'opération
- On va comparer aux constantes près la vitesse de croissance des fonctions qui comptent les nombres d'opérations.

Comparaisons de complexités

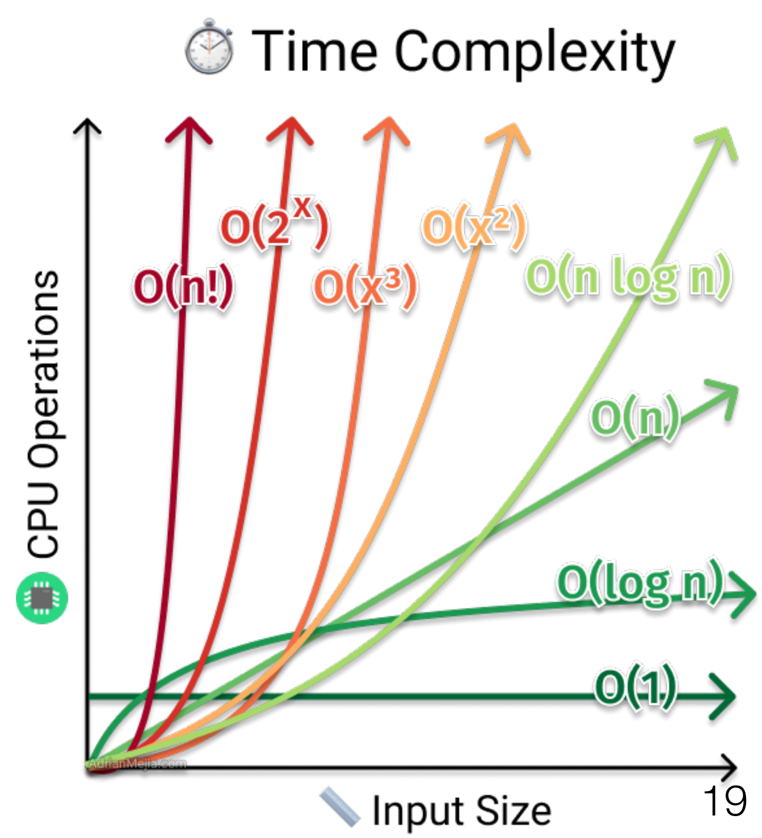
Notation asymptotique

- Soit $g : \mathbb{N} \rightarrow \mathbb{R}^{>0}$ une fonction positive.
  - Grand O: $O(g)$ est l'ensemble des fonctions positives f pour lesquelles il existe une constante strictement positive a et un entier n_0 tels que : $f(n) \leq ag(n)$, pour tout $n \geq n_0$
  - Grand Omega: $\Omega(g)$ est l'ensemble des fonctions positives f pour lesquelles il existe une constante strictement positive a et un entier n_0 tels que : $f(n) \geq ag(n)$, pour tout $n \geq n_0$.
  - Grand théta: $\Theta(g) = O(g) \cap \Omega(g)$.
- **Exemple (complexités au pire)**
  - T (BULLES2,n) = O(n2) = Ω(n2)   T (BULLES1,n) = O(n3) = Ω(n3)

Un tutoriel en détail: https://youtu.be/nA9C2_J76IM

Fonctions de référence

Définitions (désignations des complexités courantes)			
notation	désignation	notation	désignation
$\Theta(1)$	constante	$\Theta(n^2)$	quadratique
$\Theta(\log n)$	logarithmique	$\Theta(n^3)$	cubique
$\Theta(\sqrt{n})$	racinaire	$\Theta(nk)$, $k \in N$, $k \geq 2$	polynomiale
$\Theta(n)$	linéaire	$\Theta(a^n)$, $a > 1$	exponentielle
$\Theta(n \log n)$	quasi-linéaire	$\Theta(n!)$	factorielle



Une erreur très courante

- Attention!!! La complexité d'un algorithme ne parle pas du temps de calcul absolu des implémentations de cet algorithme mais de la vitesse avec laquelle ce temps de calcul augmente quand la taille des entrées augmente.

- On a effacé les constantes : $O(n) = O(2n)$. La complexité asymptotique ne nous permet pas de comparer deux algorithmes différents de même complexité. Une telle comparaison ne serait d'ailleurs pas forcément utile, à cause des différences entre les ordinateurs.

Relation temps-taille

	Évolution du temps quand la taille est **10 fois** plus grande	Évolution de la taille quand le temps est **10 fois** plus grand
1	t	∞
$\log_2 n$	$t + 3{,}32$	n^{10}
n	$10 \times t$	$10 \times n$
$n \log_2 n$	$(10 + \varepsilon) \times t$	$(10 - \varepsilon) \times t$
n^2	$100 \times t$	$3{,}16 \times n$
n^3	$1000 \times t$	$2{,}15 \times n$
2^n	t^{10}	$n + 3{,}32$

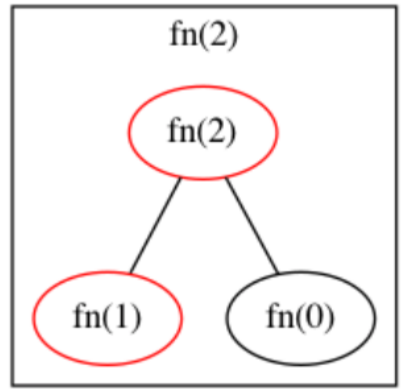
Calcul de complexité dans les structures de contrôle

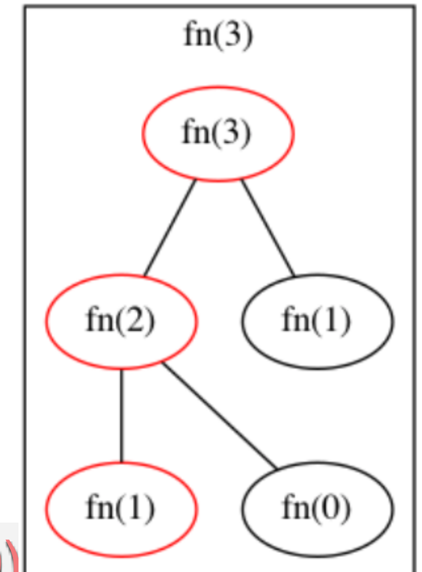
- Les instructions élémentaires (arithmétique affectations, comparaisons) sont en temps constant, soit en $\Theta(1)$.
- Tests (Conditionnel):
  - si a ∈ O(A), b ∈ O(B) et c ∈ O(C) alors (**si** a **alors** b **sinon** c) ∈ O (A + max(B , C))
  - si a ∈ Ω(A), b ∈ Ω(B) et c ∈ Ω(C) alors (**si** a **alors** b **sinon** c) ∈ Ω(A + min(B , C))
- Cas des boucles imbriquées
  - Boucles si $a_i \in O(A_i)$ (idem Ω, Θ) alors Boucles si $a_i \in O(A_i)$ (idem Ω, Θ) alors (**pour** i de 1 à n **faire** a_i)$\in O(\sum_{i=1}^{n}(A_i))$
  - Lorsque A_i est constant égal à A, on a (**pour** *i* **de** 1 **à** *n* **faire** a_i)$\in nO(A)$
- Cas de procédures récursives: Analyse avec un arbre de récursivité (Page suivante).

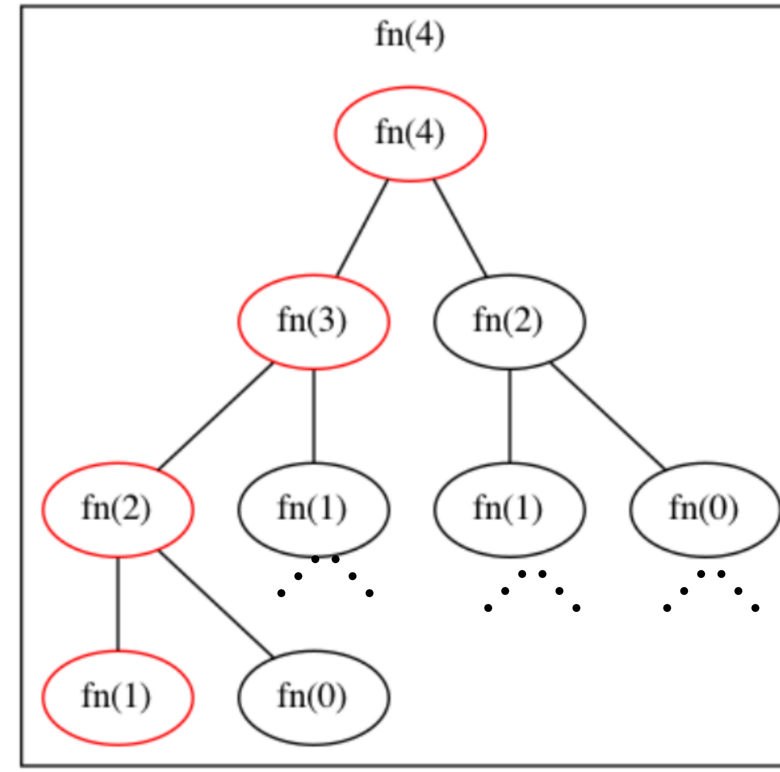
Arbre de récursivité

```
int fn(n) {
  if (n < 0) return 0;
  if (n < 2) return n;

  return fn(n - 1) + fn(n - 2);
}
```







- n = 2 -> la fonction est appelé 3 fois. Premièrement fn(2) qui appelle fn(1) et fn(0)
- n=3 -> la fonction est appelé 5 fois. Premièrement fn(3), qui appelle fn(2) et fn(1) etc.
- n=4 -> la fonction est appelé 9 fois. Premièrement fn(4), qui appelle fn(3) et fn(2) etc.

- Le profondeur est n
- Pour une arbre complète, le nombre de noeuds est 2^n-1. Mais dans cette arbre, le dernier niveau ne contient que 2 noeuds. Donc la nombre d'appels est $2^{n-1} +1$.
- $2^{n-1} +1 \in O(2^n)$
- (Attention: On ne décrit jamais une complexité par $O(2^{n-1})$ ou $O(2^{n-1}+1)$)

Optimalité

E (P) = ensemble des algorithmes qui résolvent un problème au moyen de certaines opérations élémentaires.

A ∈ *E (P)* est optimal en temps (par rapport à *E (P)* ssi. ∀ *B* ∈ *E (P),* ∀ donnée *d,* *T(B,d)* ≥ *T(A,d)*

A ∈ *E (P)* est asymptotiquement optimal ssi. *T(A,n)* = *O(T(B,n))*

Exemple (complexité au pire)

Rappelle: T(n) = O(f(n))
ssi. ∃ c > 0 ∃ N > 0 ∀ n > N T(n) ≤ c.f(n)

- BULLES 2 est asymptotiquement optimal parmi les algorithmes de classement qui utilisent les opérations élémentaires : comparaisons, transpositions d'éléments consécutifs.
- Preuve : Inv((n,n-1, …,1)) = $\frac{n(n-1)}{2}$ = O (n2)

Inv(s): nombre de transposition pour trier une séquence s

Plus Grand Diviseur Commun (PGDC)

Spécification

Calculer pgcd(n,m), plus grand diviseur commun aux entiers ≥0, n et m.

- **Algorithme 1** (n≥m)

  pour i : = m à 1 **pas** -1 **faire**

  si i divise n et m **alors** retour (i)

pgcd (21,9) = 3 [21 = 3x7, 9 = 3x3] 7 étapes

- **Algorithme d 'Euclide** (300 avant J-C)

  division entière n = q.m + r, 0 ≤ r < m /* n%m*/

  propriété : pgcd (n,m) = pgcd (m,r)

  si m = o **alors** pgcd(n,m) = n

  sinon pgcd(m, n mod m)

Preuve : si n<m première étape = échange, sinon propriété arithmétique

Terminaison : m décroît à chaque étape (strictement)

pgcd(9,21) = pgcd(21,9) = pgcd(9,3) = pgcd(3,0) = 3 3 étapes

Complexité de PGCD

- **Algorithme 1**

  T(Algo1, (n,m)) = O(min(m,n))

  moins de 2.min(n,m) divisions entières

- **Algorithme d'Euclide**

  Théorème de Lamé (1845) : (n≥m)

  Si l 'algorithme d 'Euclide nécessite *k* étapes pour calculer pgcd (n,m), on a n≥m≥ Fib_k

  [$Fib_0 = 0$, $Fib_1 = 1$, $Fib_k = Fib_{k-1} + Fib_{k-2}$, pour k>1] [0, 1, 1, 2, 3, 5, 8, 13, 21……]

pgcd(21,13) = pgcd(13,8) = pgcd(8,5) = pgcd(5,3) = pgcd(3,2) = pgcd(2,1) = pgcd(1,0) = 1

T(Euclide, (n,m)) = O(log min(n,m))

Version récursive

```
fonction  PGCD(n,m: entier): entier; {n>=0, m>=0}
début
si m = 0 alors retourner (n)
    sinon retourner (PGCD(m, n mod m));
fin.
```

Version itérative :

```
fonction  PGCD(n,m: entier): entier; {n>=0, m>=0}
 début
    var temp;
tant que m > 0 faire
        temp := m;
        m := n mod m;
        n := temp;
    fin tanque;
    retourner (n);
fin.
```

Version récursive :

```
int PGCD(int n, int m){
/* n>=0, m>=0 */
    if (m == o) return (n);
    else return ( PGCD(m, n % m));
}
```

Version itérative :

```
int PGCD(int n, int m){
/* n>=0, m>=0 */
    int temp;
    while (m > o){
        temp = m;
        m = n % m;
        n = temp;
    }
    return (n);
}
```

CERGY PARIS UNIVERSITÉ

浙江科技學院 ZHEJIANG UNIVERSITY OF SCIENCE & TECHNOLOGY

Récursivité terminale

fonction F(x) ;
début
  si C(x) **alors** { I; retour (y); }
  sinon {J; retour (F(z)); }
fin.

SEUL APPEL RÉCURSIF

Élimination (par copier-coller) :

fonction F(x) ;
début
  tant que non C(x) **faire**
    {J; $x := z$; }
  {I; retour (y); }
fin.

Factorielle

fonction FAC(*n*) ;

début
si *n*=0 alors retour (1)
sinon retour (*n** FAC(*n*-1)) ;
fin.
RÉCURSIVE

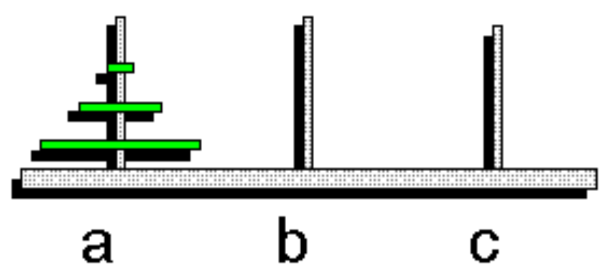
fonction FAC'(*n, m*)

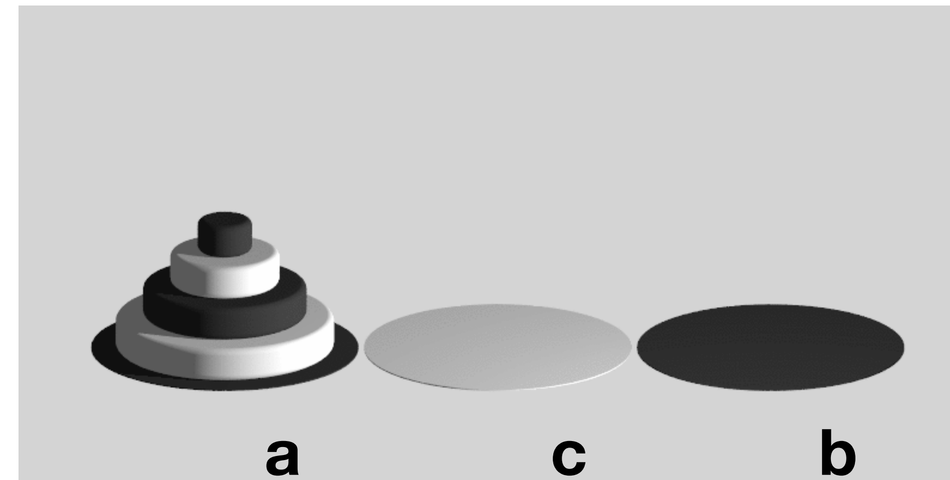
début
si *n*=0 **alors retour** (*m*)
sinon retour (FAC'(*n-1, n*m*)) ;
fin.
RÉCURSIVITÉ TERMINALE

fonction FAC(*n*) ;
début
m : = 1 ;
tant que *n* > 0 **faire**
{ *m* := *n** *m* ; *n* := *n*-1 ; }
retour (*m*) ;
fin.
ITÉRATIVE

fonction FAC'(*n, m*) ;
début
tant que *n* > 0 **faire**
{ *m* : = *n** *m* ; *n* : = *n*-1 ; }
retour (*m*) ;
fin.

Tours de Hanoï





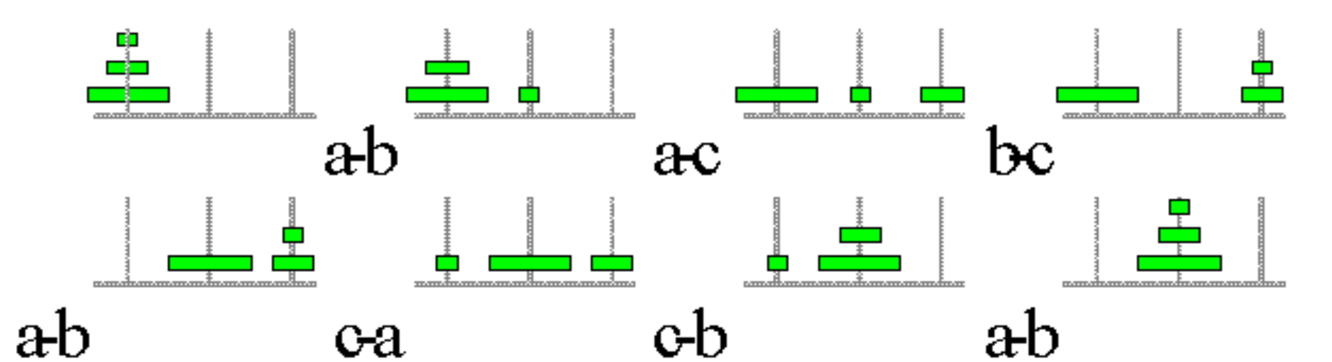
RÈGLES (opérations élémentaires)

1. déplacer un disque à la fois d'un bâton sur un autre
2. ne jamais mettre un disque sur un plus petit

BUT

transférer la pile de disques de a vers b

EXEMPLE (3 disques)



Spécification

Calculer H(n, x, y, z) = suite des coups pour transférer n disques de x vers y avec le bâton intermédiaire z ($n \geq 1$).

Relations

$$H(n, x, y, z) = \begin{cases} x\text{->}y & \text{si } n = 1 \\ H(n\text{-}1, x, z, y) \;;\; x\text{->}y \;;\; H(n\text{-}1, z, y, x) & \text{si } n > 1 \end{cases}$$

Algorithme ($n \geq 1$)

fonction H(n, x, y, z) ;

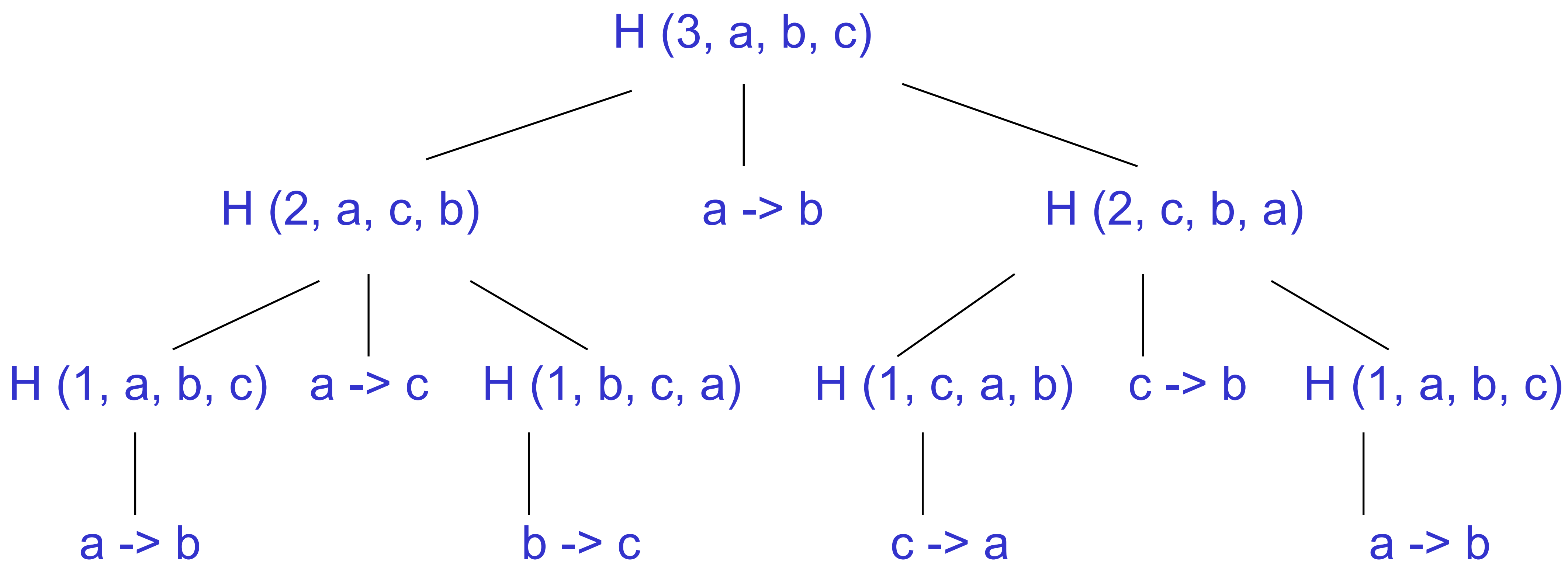
début

si $n = 1$ **alors** écrire (x->y) ;

sinon { H(n-1, x, z, y); écrire(x->y); H(n-1, z, y, x); }

fin.

Exécution



Temps d ’exécution

T(H, *n* disques) = $O(2^n)$

Longueur(*H(n, a, b, c)*) = $2^n - 1$

H est optimal (parmi les algorithmes qui respectent les règles)

Conclusion

- Révision de définitions préliminaires
- Pourquoi/**comment** calculer la complexité d'un algorithme
- Algorithme Récursive