Informatique de base (IB)
Récursivité

Martin Quinson <martin.quinson@loria.fr>

École Supérieure d'Informatique et Applications de Lorraine – 1ère année

2005-2006

(version du 25 décembre 2006)

Quatrième chapitre

La récursivité

- Introduction
- Fonctions et procédures récursives
  - Premier exemple : la factorielle
  - Différents schémas de récursion
  - Principes et dangers de la récursivité
  - Comment résoudre un problème par récursion ? Les tours de Hanoï
- Structures de données récursives
  - Exemple : un type chaîne
- Quelques fonctions récursives classiques
- Preuves de fonctions récursives
- Dérecursivation
  - Retour sur l'exécution en mémoire
  - Dérécursivation de récursivité terminale
  - Transformation en récursivité terminale
  - Algorithme générale utilisant une pile
- Back-tracking

Définition

Objet récursif : défini à partir de lui-même

- Exemples :
  - $U(n) = 3 \times U(n-1) + 1\,;\; U(0) = 1$
  - Chaîne de caractères = soit une lettre suivie d'une chaîne, soit chaîne vide
- Une condition terminale (ou base de récursivité) évite la boucle infinie
- Il est parfois possible de réécrire l'objet récursif autrement, sans récursion

En mathématiques : la récurrence

- Axiomatique de Peano (1880) : définition de l'ensemble des entiers naturels $\mathbb{N}$
  1. 0 est un nombre entier
  2. Si n est un nombre entier, son successeur (noté $n+1$) aussi
  3. Il n'existe pas de nombre x tel que $x+1=0$
  4. Des nombres distincts ont des successeurs distincts ($x \neq y \Leftrightarrow x+1 \neq y+1$)
  5. Si une propriété est vraie (i) pour 0 (ii) pour le successeur de chaque entier, elle est vraie pour tous les entiers
- Démonstration par récurrence
  - On démontre que la propriété est vraie pour 0 (ou autre)
  - On démontre qu'elle est vraie pour $n+1$ si elle est vraie pour n
  - On en déduit qu'elle est vraie pour tous les nombres

En informatique

Deux notions

- Fonctions et procédures définies de manières récursive
- Structures de données définies de manière récursive

Les fonctions récursives sont bien adaptées aux structures de données récursives

Quatrième chapitre

La récursivité

- Introduction
- Fonctions et procédures récursives
  - Premier exemple : la factorielle
  - Différents schémas de récursion
  - Principes et dangers de la récursivité
  - Comment résoudre un problème par récursion ? Les tours de Hanoï
- Structures de données récursives
  - Exemple : un type chaîne
- Quelques fonctions récursives classiques
- Preuves de fonctions récursives
- Dérecursivation
  - Retour sur l'exécution en mémoire
  - Dérécursivation de récursivité terminale
  - Transformation en récursivité terminale
  - Algorithme générale utilisant une pile
- Back-tracking

Fonctions et procédures récursives

Définition de fonction dite **récursive** ssi
elle est contient des **appels à la fonction elle-même**

Exemple

- la factorielle est définie mathématiquement par $\begin{cases} n! = n \times (n-1)! \\ 0! = 1 \end{cases}$
- Spécification du problème :
  - factorielle : entier $\longrightarrow$ entier
    Précondition : factorielle(n) définie ssi $n \geq 0$
    Postcondition : factorielle(n)$= n!$
  - factorielle : $n \mapsto r$
    P : $n \geq 0$
    Q : $r = n!$

L'algorithme récursif pour la factorielle

Il traduit littéralement la définition mathématique

FACTORIELLE(n) :
  si n $= 0$ **alors** $r \leftarrow 1$
  sinon $r \leftarrow n \times factorielle(n-1)$
  finsi

Remarques :

- $r \leftarrow 1$ est la condition terminale : ne fait pas d'appel récursif
- $r \leftarrow n \times factorielle(n-1)$ est le cas général : fait un appel récursif
- Pour que le calcul se termine, il faut arriver au cas terminal

Déroulement du calcul de la factorielle

```
FACTORIELLE(n) :
  si n = 0 alors  r ← 1
           sinon r ← n × factorielle(n − 1)
  finsi
```

$$factorielle(4) = 4 \times \overbrace{factorielle(3)}^{3 \times factorielle(2)} \overbrace{}^{2 \times factorielle(1)} \overbrace{}^{1 \times factorielle(0)}$$

Descente récursive

$$4 \times 3 \times 2 \times 1 \times \overbrace{1}$$

Condition terminale

$$\overbrace{}^{1} \quad \overbrace{}^{2} \quad \overbrace{}^{6} \quad \overbrace{}^{24}$$

Remontée récursive

$$factorielle(4) = 24$$

Schéma général d'une récursion

si cond **alors** TTER
sinon TGEN
finsi

- cond est une expression booléenne
- Si cond est vraie, exécuter cas terminal TTER (sans d'appel récursif)
- Si cond est fausse, exécuter cas général TGEN (avec des appels récursifs)
- Exemple : pour factorielle(n), on a :

  TTER : $r \leftarrow 1$

  TGEN : $r \leftarrow n \times \mathit{factorielle}(n-1)$

Autres schémas de récursion : récursivité multiple

Plus d'un appel récursif

Exemple : relations de Pascal et C_n^p

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

Algorithme correspondant :

PASCAL $(n,\ p)$

Si $p = 0$ ou $p = n$ **alors** $r \leftarrow 1$

sinon $r \leftarrow$ PASCAL $(n-1,\ p)$ + PASCAL $(n-1,\ p-1)$

Autres schémas de récursion : récursivité mutuelle

Des fonctions s'appellent les unes les autres

Exemple : une définition de la parité

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

Algorithmes correspondants :

PAIR (n)

Si $n = 0$ **alors** renvoyer *vrai*
sinon renvoyer IMPAIR $(n-1)$

IMPAIR (n)

Si $n = 0$ **alors** renvoyer *faux*
sinon renvoyer PAIR $(n-1)$

Rq : on parle aussi de récursivité *croisée*

Autres schémas de récursion : récursivité imbriquée

Appel récursif en paramètre

Exemple : la fonction d'Ackerman

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

D'où l'algorithme :

ACKERMAN(m, n)
  si $m = 0$
    alors $n + 1$
    sinon si $n = 0$ **alors** ACKERMAN($m - 1$, 1)
                sinon ACKERMAN($m - 1$, ACKERMAN(m, $n - 1$))

Attention, cette fonction croît vite :

$Ack(1,n) = n + 2$ $\qquad$ $Ack(2,n) = 2n + 3$

$Ack(3,n) = 8 \cdot 2^n - 3$ $\qquad$ $Ack(4,n) = 2^{2^{\cdots^{2}}\}n}$

$Ack(4,4) > 2^{65536} > 10^{80}$ (nombre estimé de particules dans l'univers)

Principes et dangers de la récursivité

- Intérêts
  - Moyen simple d'exprimer la solution d'un problème
  - Preuve de correction en général plus facile que pour une solution itérative
  - Mécanisme de base pour les langages fonctionnels (LISP) et logiques (Prolog)
- Inconvénients et difficultés
  - Inefficace dans les langages non adaptés (mais on peut toujours «dérécursiver»)
  - Garantie de terminaison : il faut retomber toujours sur une condition terminale
    Ordre bien fondé = suite des valeurs des arguments :
    strictement monotone et atteint toujours une valeur définie explicitement
- En résumé
  - Les usages algorithmiques de la récursivité sont nombreux
  - Une différence avec la récurrence : l'ordre de la suite
    - En mathématiques, on part du cas terminal et on augmente
      On montre que c'est vrai pour tout $\mathbb{N}$
    - En algorithmique, on part de l'argument passé, et on revient vers le cas terminal
      On calcule le résultat pour l'argument passé
  - Les algorithmes sont souvent simples à comprendre, et complexes à écrire

Quatrième chapitre

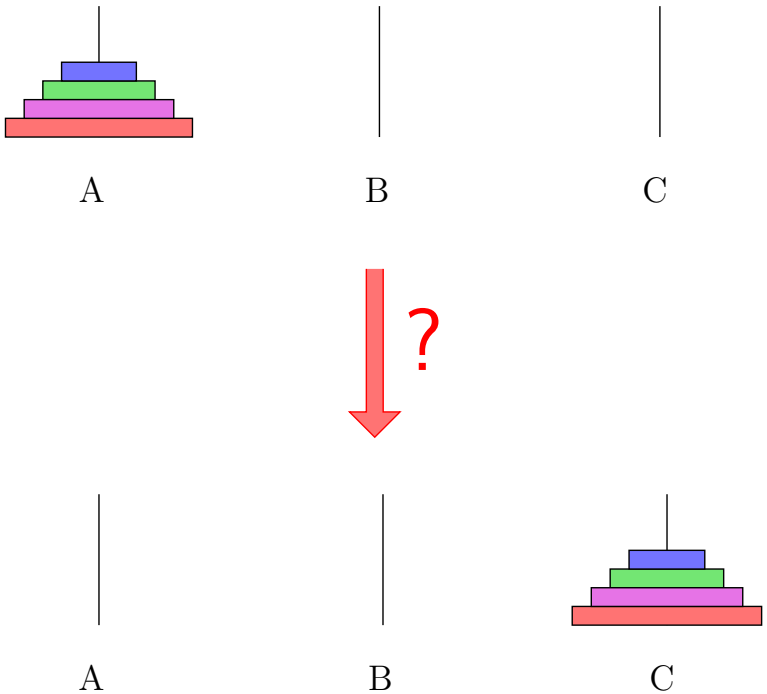
La récursivité

- Introduction
- Fonctions et procédures récursives
  - Premier exemple : la factorielle
  - Différents schémas de récursion
  - Principes et dangers de la récursivité
  - Comment résoudre un problème par récursion ? Les tours de Hanoï
- Structures de données récursives
  - Exemple : un type chaîne
- Quelques fonctions récursives classiques
- Preuves de fonctions récursives
- Dérecursivation
  - Retour sur l'exécution en mémoire
  - Dérécursivation de récursivité terminale
  - Transformation en récursivité terminale
  - Algorithme générale utilisant une pile
- Back-tracking

Comment résoudre un problème par récursion ?

1. Déterminer le paramètre portant la récursion :
   Entier ou type défini de manière récursive (cf. plus loin)
2. Résoudre les cas simples : ceux pour lesquels on a directement la réponse
   Ce sont les cas terminaux
3. Établir la récursion :
   On suppose savoir résoudre le problème pour une (ou plusieurs) valeur du paramètre strictement plus petite (dans un sens à préciser) que la valeur passée en argument
4. Écrire le cas général
   Exprimer la solution cherchée en fonction d'une solution supposée connue
5. Écrire les conditions d'arrêt
   Vérifier que la récursion parviendra à ces valeurs dans tous les cas

Un problème récursif classique : les tours de Hanoï



- Données : n disques de tailles différentes

  Un troisième piquet est disponible
- Problème : changer la pile de piquet
- Contrainte : pas de grand disque sur un petit

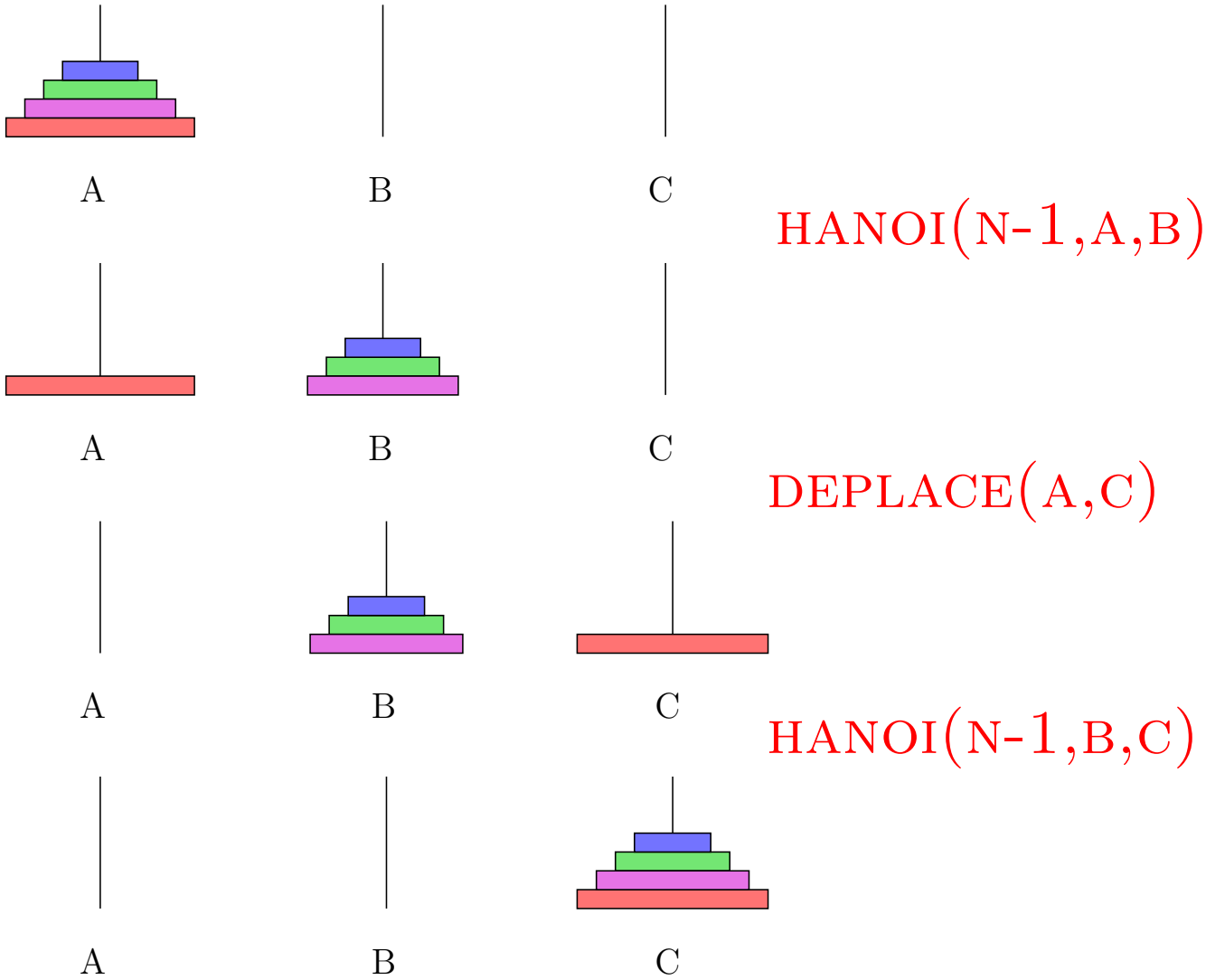
Étude du problème

- Paramètres :
  - le nombre n de disques empilés sur le piquet de départ
  - les piquets
- La récursion se fera sur l'entier n
- Comment résoudre le problème pour n disques quand on sait faire pour $n-1$?
- La décomposition se fait entre le plus grand disque et les (n-1) plus petits

- On veut écrire la procédure HANOI(N,DEP,ARR).
  Elle déplace les N disques du piquet de départ DEP vers le piquet d'arrivée ARR
- On introduit la procédure DEPLACER(DEP,ARR) (pour simplifier)
  Elle déplace le disque du dessus de DEP vers ARR (en vérifiant que l'on empile pas un grand sur un petit)

- Condition d'arrêt : quand il reste un seul disque, on utilise DEPLACER
  HANOI(1,X,Y)=DEPLACER(X,Y)

Décomposition possible pour calculer hanoi(n,a,c)



Algorithme correspondant

```
HANOI(n,a,b) :
  si n = 1 alors  deplacer(a,b)
           sinon  hanoi(n-1, a, c)
                  deplacer(a, b)
                  hanoi(n-1, c, b)
  finsi
```

Variante avec 0 comme cas terminal

```
HANOI(n,a,b) :
  si n ≠ 0 alors  hanoi(n-1, a, c)
                  deplacer(a, b)
                  hanoi(n-1, c, b)
  finsi
```

Quatrième chapitre

La récursivité

Structures de données récursives

- Définition : type récursif $\Leftrightarrow$ objet construit à partir d'objets du même type
- Exemples classiques :
  - liste = un élément suivi d'une liste, ou la liste vide
  - arbre binaire = valeur + fils droit + fils gauche, ou l'arbre vide
  - ...
  - Ces structures seront vues plus tard dans le cours

Exemple : un type chaîne

- Opérations définies :

chvide	*l'objet chaîne vide*	
addT	Chaîne $\times$ Caractère $\mapsto$ Chaîne	
	Adjonction d'un caractère en tête de la chaîne	
addQ	Chaîne $\times$ Caractère $\mapsto$ Chaîne	
	Adjonction d'un caractère en queue de la chaîne	
premier	Chaîne $\mapsto$ Caractère	*Premier caractère*
	Précondition premier(ch) est défini ssi non estvide(ch)	
dernier	Chaîne $\mapsto$ Caractère	*Dernier caractère*
	Précondition dernier(ch) est défini ssi non estvide(ch)	
début	Chaîne $\mapsto$ Chaîne	*Chaîne privée du dernier caractère*
fin	Chaîne $\mapsto$ Chaîne	*Chaîne privée du premier caractère*
estvide	Chaîne $\mapsto$ Booléen	*Teste si la chaîne est vide*

- Ce type est récursif car les chaînes sont construites à partir de chaînes
- Exemple : «hello»=addT(addQ(addQ(addQ(addQ(chvide,'e'),'l'),'l'),'o'),'h')

Implantation en Java

Element Classe représentant une lettre et la chaîne derrière (ie, les chaînes non-vides)

Chaine Classe représentant une chaîne (vide ou non)

La classe Element

```
public class Element {
  public char val;
  public Element suite;

  // Constructeur
  Element(char x, Element suite) {
      val = x;
      this.suite = suite;
  }
}
```

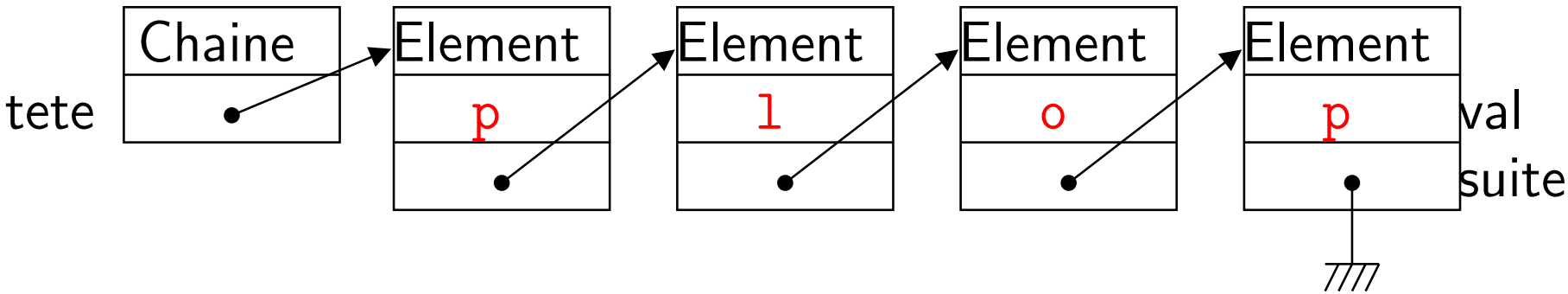
La classe Chaine

```
public class Chaine {
  public Element tete;

  // Constructeur -- donne la liste vide
  Chaine() {
      tete = null;
  }
  // Méthodes
  public boolean estVide() {
      return tete == null;
  }
  public void addT(char x) {
    // Créer le nouvel élément, et le raccorder
    Element nouveau = new Element(x,tete);
    // C'est la nouvelle tête
    tete = nouveau;
} }
```

(la distinction entre les deux permet de différencier la chaîne vide d'un objet de type chaîne qui ne serait pas affecté)

Schéma mémoire de la Chaine contenant «plop»



Exercice :

Écrire les méthodes `addQ`, `premier`, `dernier`, `début` et `fin`.

Quatrième chapitre

La récursivité

Fonctions récursives classiques : Fibonacci

Étude de la vitesse de reproduction des lapins (XII ième siècle)

- On part d'un couple
- Chaque couple produit un couple tous les mois
- Un couple est productif après deux mois
- $F_0 = 1 \quad F_1 = 1 \quad F_2 = 2 \quad F_3 = 3 \quad F_4 = 5 \quad F_6 = 8 \quad F_7 = 13 \quad \ldots$

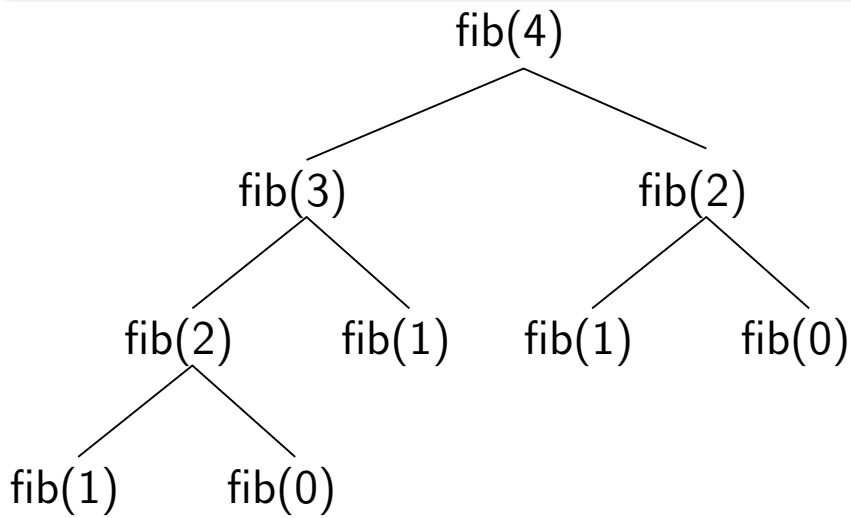
$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ \forall n, F_n = F_{n-1} + F_{n-2} \end{cases}$$

Exercice :

Calculer le nombre d'appels effectués.

Algorithme correspondant

```
static int fib(int n) {
  if (n <= 1)
    return n; // cas de base
  else
    return fib(n-1) + fib(n-2);
}
```



Fonction 91 McCarthy

Définition

$$M(n) = \begin{cases} n - 10 & \text{si } n > 100 \\ M(M(n+11)) & \text{si } n \leq 100 \end{cases}$$

Propriété amusante :

$\forall n \leq 101, M(n) = 91$

$\forall n > 101, M(n) = n - 10$

Preuve

- Pour $90 \leq k \leq 100$, on a $f(k) = f(f(k+11)) = f(k+1)$
  En particulier, $f(91) = f(92) = \ldots = f(101) = 91$
- Pour $k \leq 90$: Soit r tel que :$90 \leq k + 11r \leq 100$
  $f(k) = f(f(k+11)) = \ldots = f^{(r+1)}(k+11r) = f^{(r+1)}(91) = 91$

John McCarthy (1927-)

Prix Turing 1971, Inventeur du langage LISP, de l'expression «intelligence artificielle» et de l'idée d'application service provider (en 1961).

Fonction de Syracuse

```
SYRACUSE(n) :
  si n = 0 ou n = 1 alors 1
                    sinon si n mod 2 = 0 alors SYRACUSE(n/2)
                                         sinon SYRACUSE(3 × n + 1)
  finsi
```

- Question : Est ce que cette fonction s'arrête à coup sûr ?

  Difficile à dire : la suite n'est pas monotone
- Conjecture de Collatz : $\forall n \in \mathbb{N}$, $\text{SYRACUSE}(n) = 1$

  C'est un problème ouvert depuis 1937 (plusieurs primes disponibles)

Quatrième chapitre

La récursivité

Preuves de fonctions récursives

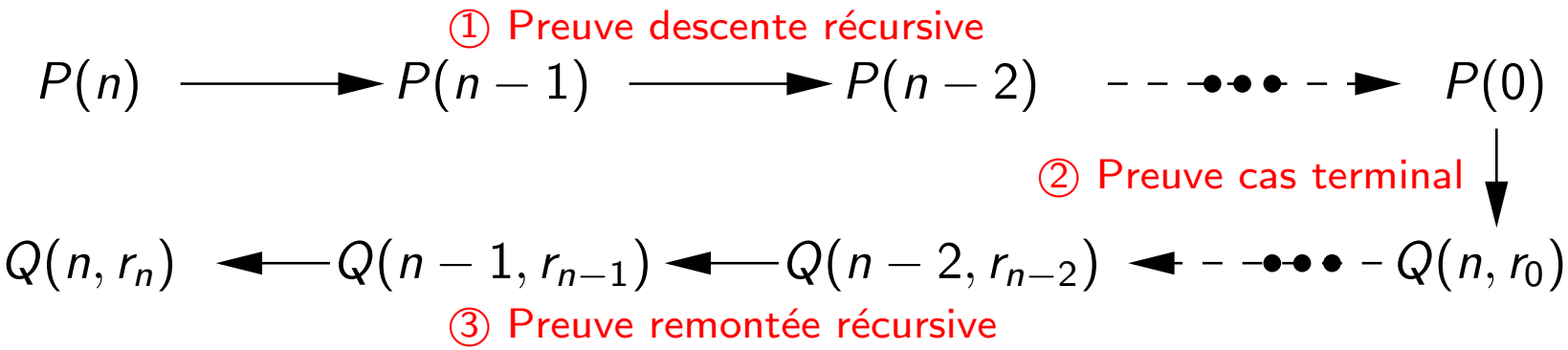
Deux choses à prouver

- Correction : respect des préconditions et postconditions
  Idée intuitive : preuve par récurrence (propagation des conditions entre étapes)
- Terminaison : le cas terminal est toujours atteint

Idée de la correction de fonctions récursives

$P(n)$: Précondition étape n ; $Q(n, r_n)$: Postcondition étape n avec résultat r_n

On veut montrer P(n) {TREC} Q(n, r_n)



Si $f(n)$ s'exprime en fonction de $f(n-1)$, il faut que :

- Dans le cas général
  - Précondition de $f(n)$ implique précondition de $f(n-1)$ ①
    Si non, le calcul est impossible
  - HdR : postcondition de $f(n-1)$ vraie. Prouver postcondition de $f(n)$ ③
- Dans le cas terminal
  - la précondition et le traitement permettent de prouver la postcondition ②

Preuve de la correction (1/2)

$$P(n)\ \{TREC\}\ Q(n, r_n) \quad (1)$$

$P(n)$ {**si** cond **alors** TTER **sinon** TGEN} $Q(n, r_n)$

Cas simple : TGEN et TTER sont des affectations

TGEN : $r \leftarrow G(n, f(n_{int}))$

TTER : $r \leftarrow v(n)$

Avec

- n_{int} Valeur de l'appel récursif
- $f(x)$ L'appel récursif
- $v(n)$ Fonction sans appel à $f(n)$
- $G(n, y)$ Fonction :
  - Sans appel récursif à $f(n)$
  - Définie $\forall n$ paramètre, $\forall y$

Exemple : Factorielle

TGEN : $r \leftarrow n \times facto(n-1)$

TTER : $r \leftarrow 1$

- $n_{int} = n - 1$
- $f(x)$: $facto(x)$
- v(n) $= 1$
- G(n,y) $= n \times y$
- $P(n)$: $n \geq 0$
- $Q(n, r)$: $r = n!$
- cond(n) : n=0

Preuve de la correction (2/2)

Cas simple : TTER et TGEN sont des affectations

- Algorithme calculant $r = f(n)$

  si cond(n) **alors** $r \leftarrow v(x)$
  sinon $r \leftarrow G(n, f(n_{int}))$

- Pour prouver (1), il suffit de prouver :
  - Dans le cas terminal : précondition et traitement impliquent postcondition
    $$P(n) \wedge cond(n) \Rightarrow Q(n, r)$$
  - Dans le cas général :
    - Descente récursive : précondition de $f(n)$ implique précondition de $f(n-1)$
      $$P(n) \wedge \neg cond(n) \Rightarrow P(n_{int})$$
    - Remontée récursive : postcondition de $f(n-1)$ implique postcondition de $f(n)$
      $$P(n) \wedge \neg cond(n) \wedge Q(n_{int}, r_{int}) \Rightarrow Q(n, r)$$

Cas plus général : plus dur

- Il faut combiner ceci avec les autres cours de preuve
  $$P(x) \Rightarrow P(x_{int}) \qquad\qquad Q(x_{int}, r_{int}) \Rightarrow Q(x, r)$$

Exemple de la factorielle

FACTORIELLE(n) :
si n $= 0$ **alors** $r \leftarrow 1$ (TTER)
sinon $r \leftarrow n \times factorielle(n-1)$ (TGEN)
finsi

$P(n) : n \geq 0$ cond(n) : $n = 0$ $Q(n, r) : r = n!$

- Cas terminal : $P(n) \wedge cond(n) \Rightarrow Q(n, r) \equiv n \geq 0 \wedge n = 0 \Rightarrow r = n!$
  Vrai (car $1 = 0!$ quoi qu'il arrive)
- Cas général :
  - $P(n) \wedge \neg cond(n) \Rightarrow P(n_{int}) \equiv (n \geq 0) \wedge (n \neq 0) \Rightarrow (n - 1 \geq 0)$
    Trivial
  - $P(n) \wedge \neg cond(n) \wedge Q(n_{int}, r_{int}) \Rightarrow Q(n, r)$
    $\equiv (n \geq 0) \wedge (n \neq 0) \wedge (r_{int} = n_{int}!) \Rightarrow (r = n!)$

    Vrai car :
    - $r = n \times r_{int}$ dans le cas général
    - $r_{int} = n_{int}! = (n-1)!$ par HdR
    - $n \times (n-1) = n!$

Preuve de terminaison

- Conditions suffisantes :
  - Valeurs successives du paramètre x : suite strictement monotone (pour un ordre éventuellement à préciser)
  - Existence d'un extrema x_0 vérifiant la condition d'arrêt
- Remarque : la suite de Syracuse semble se terminer sans ceci

- Exemple : la factorielle, bien sûr
  - $n \geq 0$
  - n strictement décroissant
  - $0 =$ condition d'arrêt

Quatrième chapitre

La récursivité

Retour sur l'exécution en mémoire

Actions réalisée lors d'un appel de fonction

1. Création d'un cadre de fonction dans la pile
2. Copie des valeurs des paramètres effectifs correspondants
3. Exécution de la fonction
4. Dépilement des paramètres formels (retour)
5. Destruction du cadre de fonction

La récursivité ne change rien à ce schéma

Exemple : le pgcd de deux entiers naturels

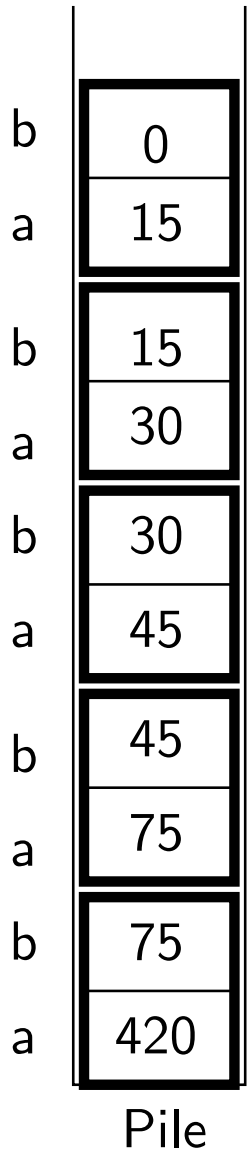
pgcd(a, b : Entier) = (r : Entier)

- Précondition : $a \geq b \geq 0$
- Postcondition : $(a \bmod r = 0)$ et $(b \bmod r = 0)$ et $\neg\,(\exists s, (s > r) \wedge (a \bmod s = 0) \wedge (b \bmod s = 0))$

Définition récursive

si $b = 0$ **alors** $r \leftarrow a$
sinon $r \leftarrow pgcd(b, a \bmod b)$

Calcul de pgcd(420,75)



si $b = 0$ **alors** $r \leftarrow a$
sinon $r \leftarrow pgcd(b, a \bmod b)$

- $pgcd(420, 75) = pgcd(75, 45) = \mathbf{15}$
- $pgcd(75, 45) = pgcd(45, 30) = \mathbf{15}$
- $pgcd(45, 30) = pgcd(30, 15) = \mathbf{15}$
- $pgcd(30, 15) = pgcd(15, 0) = \mathbf{15}$
- $pgcd(15, 0) = 15$
  C'est le cas terminal
- On dépile les paramètres
- $r \leftarrow r_{int}$ (pas d'autre traitement : $G(x, y) = y$)

- Le résultat de l'appel initial est connu dès le cas terminal
  On parle de **récursivité terminale**
- Factorielle : multiplication lors de la remontée
  $\Rightarrow$ récursivité **non-terminale**

Dérécursivation de fonctions

Transformer une fonction récursive en fonction itérative

Différentes méthodes selon les cas :

- Récursivité terminale : transformation très simple
- Récursivité non-terminale : deux méthodes (une seule générale)

- Les compilateurs utilisent ces méthodes d'optimisation
  (et bien d'autres)

Dérécursivation de récursivité terminale

- Soit l'algorithme récursif suivant :

```
f(x) :
  si cond(x) alors  TTER(x)
             sinon T(x) ; r ← f(x_int)
```

- L'algorithme itératif suivant est équivalent :

```
f(x) :
  u ← x
  jusqu'à cond(u) faire
    T(u)
    u ← u_int = h(u)
  fin jusqu'à
  TTER(u)
```

avec u_{int} étant une locale calculée par T(u)

Exemple : dérécursivation du pgcd

si $b = 0$ **alors** $r \leftarrow a$
sinon $r \leftarrow pgcd(b, a \bmod b)$

cond(a,b) : b=0

TTER(a,b) : $r \leftarrow a$

TGEN(a,b) : T(a,b) ; $r \leftarrow pgcd(a_{int}, b_{int})$

T(a,b) : $a_{int} \leftarrow b$

$b_{int} \leftarrow a \bmod b$

Version itérative déduite directement :

$pgcd(a, b)$:
  $u \leftarrow a$; $v \leftarrow b$
  jusqu'à v=0 **faire**
    $temp \leftarrow v$
    $v \leftarrow u \bmod v$
    $u \leftarrow temp$
  fin jusqu'à
  $r \leftarrow u$

(pgcd a deux arguments, d'où certains changements)

Transformation en récursivité terminale

- Soit **f(n)** fonction récursive non-terminale
- Si récursivité non-terminale, méthode précédente pas applicable

- On peut *parfois* définir une fonction **g()** à récursivité terminale équivalente
  - $g()$ a plus de paramètres que $f()$
  - On stocke les paramètres intermédiaires lors descente
  - On évite ainsi la remontée

- $f()$ doit avoir de bonnes propriétés (associativité, commutativité,...)

- Méthode : n opérations lors de remontée $\Rightarrow$ n paramètres supplémentaires
- Il faut vérifier :
  - Que l'on peut trouver le résultat de cette manière
  - Que l'algorithme obtenu est récursif terminal

Exemple : la factorielle en récursivité terminale

```
FACTORIELLE(n) :
  si n = 0 alors  r ← 1
           sinon r ← n × factorielle(n − 1)
  finsi
```

- On pose : $G(a, b) = a \times factorielle(b)$
- On vérifie :
  - Que $factorielle(n)$ se calcule avec $G()$: C'est $G(1, n)$
  - Que le calcul de $G()$ est récursif terminal

Calcul de $t = G(a, b)$:

```
si b = 0 alors  t ← 1
         sinon t ← a × (b × factorielle(b − 1))
finsi
```

Par associativité,
TGEN devient :

```
         sinon t ← (a × b) × factorielle(b − 1)
```

C'est à dire :

```
FACTORIELLE(n) :
  si b = 0 alors  t ← 1
           sinon t ← G(a × b; b − 1)
  finsi
```

Dérécursivation de la factorielle

```
FACTORIELLE(n) :
  r ← HELPER(1, n)

HELPER(a, b) :
  si b = 0 alors  t ← a
           sinon t ← HELPER(a × b, b − 1)
  finsi
```

- Cette fonction est récursive terminale, on peut la dérécursiver :

```
HELPER(a,b) :
  u ← a ; v ← b                  /* locales */
  jusqu'à v = 0 faire
    u ← u × v                    /* Attention à l'ordre */
    v ← v − 1
  fin jusqu'à
  r ← u
```

Algorithme générale utilisant une pile

- Idée : Les processeurs séquentiels exécutent toutes les fonctions récursives
  $\Rightarrow$ Il est toujours possible de dérécursiver
- Le principe est de simuler la pile d'appels des processus
- Exemple dans le cas d'un seul appel récursif

si cond(x) **alors** $r \leftarrow g(x)$
sinon T(x) ; $r \leftarrow G(x, f(x_{int}))$

```
p ← pileVide
a ← x /* a : variable locale */
/* empilements (descente) */
jusqu'à cond(a) faire
  empiler(p, a)
  a ← h(a)
fin jusqu'à
r ← g(a) /* cas terminal */
/* dépilements et calculs (remontée) */
jusqu'à pileEstVide(p) faire
  a ← sommet(p) ; dépiler(p) ; T(a)
  r ← G(a, r)
fin jusqu'à
```

Remarque :
Si $h()$ inversible, pas besoin de pile :
paramètre retrouvé par $h^{-1}()$

Condition d'arrêt : compter les appels

Dérécursivation des tours de Hanoï

```
HANOI(n,a,b) :
  si n > 0 alors hanoi(n-1, a, c)
                 deplacer(a, b)
                 hanoi(n-1, c, b)
  finsi
```

Il faut déplier l'appel comme ferait un processeur séquentiel

- H(4,a,b,c) = H(3,a,c,b)+D(a,b)+H(3,c,a,b)
- Calcule premier terme inconnu : H(3,a,c,b) = H(2,a,b,c)+D(a,c)+H(2,a,c,b)
- Calcule premier terme inconnu : H(2,a,b,c) = H(1,a,c,b)+D(a,c)+H(1,a,b,c)
- Calcule premier terme inconnu : H(1,a,c,b) = D(a,c)
- Je peux reprendre du travail laissé de coté : H(1,a,b,c) = D(a,b)
- (et ainsi de suite)

Dérécursivation des tours de Hanoï

```
HANOI(n,a,b) :
  si n > 0 alors hanoi(n-1, a, c)
                 deplacer(a, b)
                 hanoi(n-1, c, b)
  finsi
```

```
hanoi_derec(n, A, B) :
  empiler(n, A, B, 1)
  tant que (pile non vide)
    (n, A, B, TypeDAppel) ← depiler()
    si (n > 0)
      si (TypeDAppel == 1)
        empiler(n, A, B, 2) /* Reprendre du travail laissé de coté (après) */
        empiler(n-1, A, C, 1) /* Calcule du premier terme inconnu (maintenant) */
      sinon /* TypeDAppel == 2, donc */
        deplace(A, B)
        empiler(n-1, C, B, 1)
```

Rq : il existe des algos itératifs plus simples (n'étant pas des dérécursivations)

Quatrième chapitre

La récursivité

Recherche combinatoire

Méthode de résolution d'une large classe de problèmes

Problème

- On a de très nombreuses solutions
- On a un jeu de contraintes qui rendent certaines solutions invalides
- On cherche la solution qui maximise une fonction

Exemples

- Sac à dos : Ali-baba cherche la liste des objets qui tiennent dans son sac, et maximisant la valeur totale
- Arbre recouvrant minimal d'un graphe donné
- Voyageur de commerce : il doit passer par n villes dans un ordre indifférent de façon à minimiser la distance totale

Méthodes de résolutions

- Recherche exaustive : étude de *toutes* les solutions (souvent infaisable)
- Backtracking : étude restreinte aux solutions valides

Back-tracking (récursivité avec retour-arrière)

Caractérisation

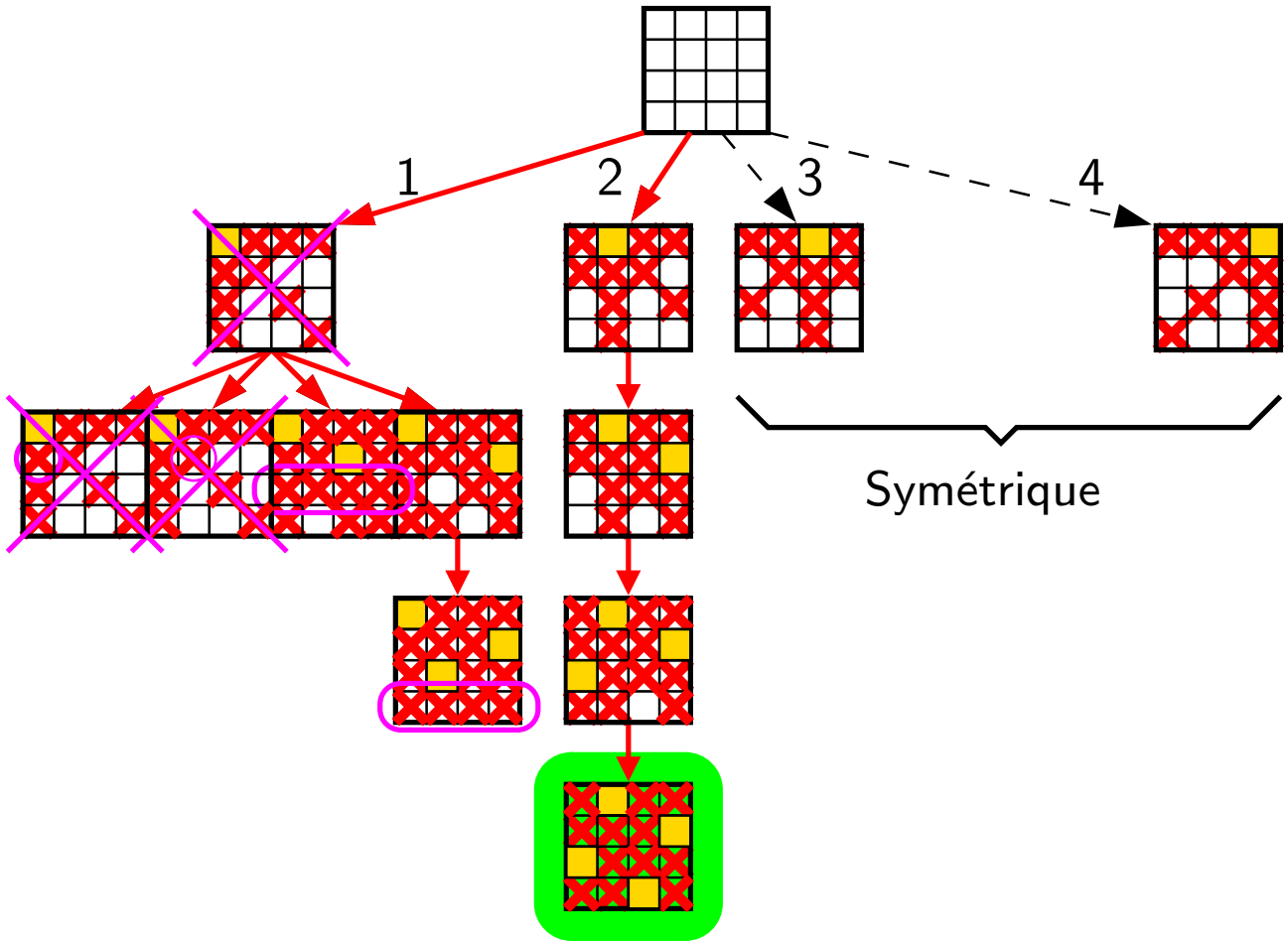
- Recherche de solution dans un espace donné de la sorte :
  - Choix d'une solution partielle valide
  - Appel récursif (pour le reste de la solution)
- Certaines solutions construites sont des impasses (impossible de faire une solution valide complète avec ce qu'on a choisi jusque là)
- Retour arrière alors nécessaire pour un *autre* choix
- Schéma général : un appel récursif à l'intérieur d'une itération

Exemple : le problème des n reines

- Objectif :
  Placer n reines sur un échiquier $n \times n$ sans qu'elles se menacent mutuellement
- Algorithme :
  - Placer une reine sur la première ligne
    Il y a n choix, chacun impliquent des impossibilités pour la suite
  - On fait un appel récursif pour la ligne suivante

Résolution du problème des n reines (pour n=4)

- À chaque étape de la récursion, itération sur les différentes solutions
- Chaque choix implique des impossibilités pour la suite
- Pour chaque itération, une descente
- En cas de blocage, remontée (et descente dans l'itération suivante)
- Jusqu'à trouver une solution (ou non)



Pseudo-code des n reines

```
bool Solution(bool plateau[][NUM_R], int ligne) {
  if (ligne >= NUM_R) // cas de base
     return true;

  for (int col = 0; col < NUM_R; col++) {
     if (PlacementLegal(plateau, ligne, col)) {
       PlaceReine(plateau, ligne, col);
       if (Solution(plateau, ligne + 1))
           return true;
       RetireReine(plateau, ligne, col);
     }
  }
  return false;
}
```

Quelques principes

- Examen « en profondeur d'abord » de l'arbre de décomposition
- Lors d'un retour arrière, restaurer l'état avant le dernier choix
  Immédiat ici (paramètres copiés lors des appels récursif), difficile en itératif
- Stratégie (ordre des branches) à établir
- Construction progressive d'une fonction booléenne
- Si la fonction retourne faux, il n'existe pas de solution

- Probable explosion combinatoire (4^4 échiquiers)
  $\Rightarrow$ Nécessité d'heuristiques pour limiter le nombre d'essais

Conclusion sur la récursivité

Outil algorithmique indispensable

- Récursivité en informatique, récurrence en mathématiques
- Les algorithmes récursifs sont fréquents car plus simples à comprendre...
  (et donc faciles à maintenir)

  ...mais moins aisés à écrire (c'est une question d'habitude)
- Les programmes récursifs sont légèrement moins efficaces...

  ...mais il est toujours possible de dérécursiver un programme
  (et les compilateurs le font)
- Fonctions classiques : Factorielle, pgcd, Fibonacci, Ackerman, Hanoï, Syracuse
- Preuve de correction : par récurrence

  Preuve de terminaison : paramètre = suite strictement monotone
- Back-tracking : recherche exhaustive dans espace des solutions *valides*
- En IB3, nombreuses structures récursives et algorithmes associés