LIFAP1 : ALGORITHMIQUE ET PROGRAMMATION RECURSIVE

Présentation de l'UE

Modalités de Contrôle des Connaissances

Présentation de l'UE LIFAP2

- Responsable de l'UE
  - Marie Lefevre – marie.lefevre@liris.cnrs.fr

- Responsables d'amphi
  - Marie Lefevre – bâtiment Nautibus
  - Nathalie Guin – bâtiment Nautibus

- Site Web de l'UE
  - http://liris.cnrs.fr/marie.lefevre/ens/LIFAP2/
  - Planning (salles, horaires)
  - Supports des CMs, TDs et TPs et corrigés des TPs
  - Modalité de Contrôle des Connaissances

Contenu

- Notions : récursivité, programmation fonctionnelle
  - Mémorisation, structure de données (listes, arbres)

- Complémentaire à l'UE LIFAP1
  - Programmation impérative et itérative
- Utile dans la suite des UE d'algorithmique et programmation (LIFAP3, LIFAP4)
  - Afin de choisir entre une approche impérative
    et une approche récursive pour résoudre un problème

Organisation

- Volume horaire
  - 9h de CM (séance d'1h30)
  - 9h de TD (séance d'1h30)
  - 18h de TP (séance de 3h)

- Plan « Réussite en Licence »
  - 1 séance de TD de révisions
  - 1 séance de soutien
    - Sur avis des intervenants de TD-TP
    - Présence obligatoire

Modalités de contrôle des connaissances

- Plusieurs épreuves :
  - Interrogations écrites en début de TD - coeff. 20%
    - 4 notes
  - TP noté en conditions d'examen - coeff. 40%
    - TP n°6, 3h, le 24/04/17
  - Epreuve écrite en amphi - coeff. 40%
    - Session 1 en mai - 90 min
    - Session 2 en juin - 60 min
- Harmonisation des notes en fin de semestre

LIFAP1 : ALGORITHMIQUE ET PROGRAMMATION RECURSIVE

Algorithme itératif / récursif

Programmation impérative / fonctionnelle

Qu'est-ce qu'un algorithme ?

- On souhaite résoudre le problème :

Trouver le minimum d'une liste de nombres :
par exemple (3 6,5 12 -2 0 7)

- Expliquer à une machine comment trouver le minimum de n'importe quelle liste de nombres

- Langage commun entre la machine et nous :
  Scheme

Définition d'un algorithme

- Un algorithme est une méthode
  - Suffisamment générale pour pouvoir traiter toute une classe de problèmes
  - Combinant des opérations suffisamment simples pour être effectuées par une machine

COMPLEXITÉ D'UN ALGORITHME

- Il faut :
  - que la machine trouve le plus vite possible
    - Complexité en temps
  - qu'elle trouve en utilisant aussi peu de place mémoire que possible
    - Complexité en espace

Rappel : la machine n'est pas intelligente

- L'exécution d'un algorithme ne doit normalement pas impliquer des décisions subjectives, ni faire appel à l'utilisation de l'intuition ou de la créativité

- Exemple : une recette de cuisine n'est pas un algorithme si elle contient des notions vagues comme « ajouter du sel »

MON PREMIER ALGORITHME

- Déterminer le minimum d'une liste de nombres
  - par exemple (3 6,5 12 -2 0 7)

MÉTHODE ITÉRATIVE

- Je parcours la liste de gauche à droite, en retenant à chaque pas le minimum provisoire

(3 6,5 12 -2 0 7)

  - Entre 3 et 6,5, c'est 3
  - Entre 3 et 12, c'est 3
  - Entre 3 et -2, c'est -2
  - Entre -2 et 0, c'est -2
  - etc.

DE QUOI AI-JE BESOIN POUR ÉCRIRE L'ALGORITHME ? (1)

- La séquence

  Début

  Instruction(s)

  Fin

- L'affectation

  Variable ← Expression
  - A ← 2
  - BX4 ← A+2*racine(15)

De quoi ai-je besoin pour écrire l'algorithme ? (2)

- La conditionnelle (1)

  Si Condition Alors

  Instruction(s)

  FinSi

- Exemple :

  Si (A > 27) Alors

  B ← A*3

  FinSi

De quoi ai-je besoin pour écrire l'algorithme ? (3)

- La conditionnelle (2)

```
Si Condition Alors
    Instruction(s)
Sinon
    Instruction(s)
FinSi
```

- Exemple :

```
Si ((A<10) et
      (B > racine(A*5))) Alors
    B ← A*3
    A ← A+B
Sinon
    A ← A+2
    B ← A*B
FinSi
```

De quoi ai-je besoin pour écrire l'algorithme ? (4)

La **condition** est :

- soit une condition élémentaire
- soit une condition complexe,
  c'est à dire une conjonction, disjonction, ou négation de conditions élémentaires et/ou complexes

De quoi ai-je besoin pour écrire l'algorithme ? (5)

- L'itérative, ou boucle

```
TantQue Condition Faire
    Instruction(s)
FinTantQue
```

- Exemple :

```
TantQue i<10 Faire
    a ← a*i
    i ← i+1
FinTantQue
```

Opérateurs booléens

- Pour écrire une conjonction, disjonction, ou négation de conditions, on a besoin des opérateurs booléens ET, OU, NON

- Une variable booléenne est une variable dont les valeurs peuvent être vrai ou faux

- Un opérateur booléen est un opérateur dont les arguments sont des variables booléennes et dont le résultat est booléen

L'OPÉRATEUR ET

X	Y	X ET Y
Vrai	Vrai	Vrai
Vrai	Faux	Faux
Faux	Vrai	Faux
Faux	Faux	Faux

L'OPÉRATEUR OU

X	Y	X OU Y
Vrai	Vrai	Vrai
Vrai	Faux	Vrai
Faux	Vrai	Vrai
Faux	Faux	Faux

L'OPÉRATEUR NON

X	NON X
Vrai	Faux
Faux	Vrai

ALGORITHME ITÉRATIF

- Soient
  - L : la liste
  - *premier*, *longueur*, *ième* et *écrire* : des primitives (i.e. algorithmes déjà définis)

Définition de minimum(L)

```
Début
    min ← premier(L)
    i ← 2
    TantQue i <= longueur(L) Faire
        Si ième(L,i) < min Alors
            min ← ième(L,i)
        FinSi
        i ← i+1
    FinTantQue
    Écrire(« Le minimum est »)
    Écrire(min)
Fin
```

VOCABULAIRE

- *minimum* et *écrire* sont des procédures,
  i.e. modifient l'environnement
- *premier*, *longueur* et *ième* sont des fonctions,
  i.e. retournent une valeur,
  mais ne modifient pas l'environnement

- L est le paramètre de la procédure minimum
- i est l'indice ou le compteur de la boucle

COMPLEXITÉ EN TEMPS DE L'ALGORITHME ITÉRATIF DU MINIMUM

```
Définition de minimum(L)
Début
    min ← premier(L)
    i ← 2
    TantQue i <= longueur(L) Faire
        Si ième(L,i) < min Alors
            min ← ième(L,i)
        FinSi
            i ← i+1
    FinTantQue
    Écrire(« Le minimum est »)
    Écrire(min)
Fin
```

Soit n la longueur de la liste L

1 affectation (initialisation)
1 affectation (initialisation)
n comparaisons
n-1 comparaisons
m affectations

n-1 affectation (incrémentation)

1 écriture
1 écriture

LE NOMBRE m DÉPEND DE LA LISTE

- Meilleur cas : $m=0$
  si le premier nombre de la liste est le minimum

- Pire cas : $m=n-1$
  si les nombres de la liste sont rangés en ordre décroissant

- Cas moyen : $m=(n-1)/2$
  s'ils respectent une distribution parfaitement aléatoire

COMPLEXITÉ EN ESPACE DE L'ALGORITHME

- Une variable *min* pour stocker le minimum provisoire
- Une variable *i* pour savoir où on en est dans la liste

MÉTHODE RÉCURSIVE (1)

- Pour trouver le minimum de la liste (3 6,5 12 -2 0 7)
- On suppose le problème résolu pour la liste privée de son premier élément i.e. (6,5 12 -2 0 7)
- Soit *min* le minimum de cette sous-liste, ici -2
- Le minimum de la liste complète s'obtient par comparaison entre le premier élément de la liste (ici 3) et *min* (ici -2)

Méthode récursive (2)

- Comment résout-on le problème pour la sous-liste ?
  - En faisant le même raisonnement.
- C'est la récursivité.

- Quand s'arrête-t-on ?
  - Quand on ne peut plus parler de sous-liste, i.e. quand la liste a un seul élément. C'est alors le minimum.

ILLUSTRATION DE LA MÉTHODE

Sur quoi faire l'appel récursif ?

Enlever le premier élément

(3 6,5 12 -2 0 7) → (6,5 12 -2 0 7)

minimum

appel récursif

minimum

-2 ← -2

Comparer -2 et 3

Comment passer du résultat de l'appel récursif au résultat qu'on cherche ?

Algorithme récursif

- Soient *vide?*, *reste* et *premier* des fonctions primitives

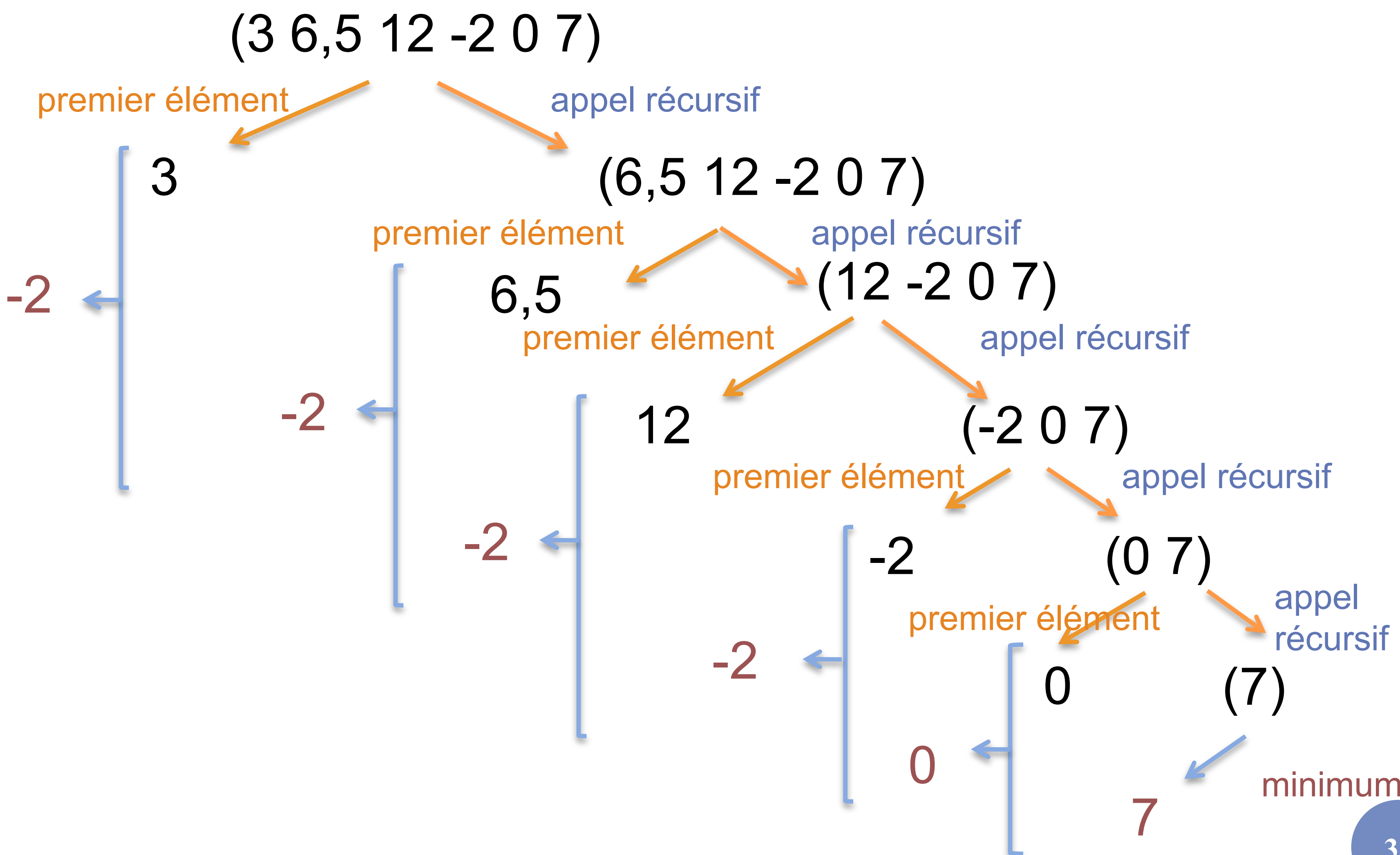
```
Définition de la fonction  minimum (L)

Si vide?(reste(L)) Alors
        retourne premier(L)
Sinon
        Si premier(L) < minimum(reste(L)) Alors
                retourne premier(L)
        Sinon
                retourne minimum(reste(L))
        FinSi
FinSi
```

Remarques

- *minimum* est ici une fonction,
  le mot *retourne* permet de dire quel est son résultat
- *minimum* est l'appel récursif

- En programmation fonctionnelle,
  on n'a plus besoin de la séquence
- En programmation récursive,
  on n'a plus besoin de la boucle

ILLUSTRATION DE L'ALGORITHME



COMPLEXITÉ EN TEMPS DE L'ALGORITHME RÉCURSIF DU MINIMUM

```
Définition de la fonction minimum(L)
Si vide?(reste(L)) Alors
    retourne premier(L)
Sinon
    Si premier(L) < minimum(reste(L))
    Alors
            retourne premier(L)
    Sinon
            retourne minimum(reste(L))
    FinSi
FinSi
```

1 test

1 comparaison
+ le nombre de comparaisons de l'appel récursif

COMPLEXITÉ EN TEMPS DE L'ALGORITHME (2)

- Si n est la longueur de la liste
- Si C(i) est le nombre de comparaisons de l'algorithme pour une liste de longueur i

- Alors $C(n) = 1+C(n-1)$
  $= 1+1+C(n-2)$
  $= \ldots$
  $= 1+1+\ldots+C(1)$
  $= 1+1+\ldots+0$
  $= n-1$

RÉSUMÉ SUR LA COMPLEXITÉ

- Choisir ce que l'on va compter
  - unité de comparaison des algorithmes
  - par exemple le nombre de comparaisons
- Ce qui importe est l'ordre de grandeur de la complexité
  - constant, log n, linéaire, n*log n, n^2, 2^n
- En LIFAP2 on s'intéressera essentiellement au nombre de fois où l'on parcourt une structure de donnée (liste, arbre)

POUR ÉCRIRE UN ALGORITHME RÉCURSIF

- Il faut choisir
  1. Sur quoi on fait l'appel récursif
  2. Comment on passe du résultat de l'appel récursif au résultat que l'on cherche
  3. Le cas d'arrêt

  - DANS CET ORDRE LÀ !!!

Structure type d'une fonction récursive

Si	je suis dans le cas d'arrêt
Alors	voila le résultat
Sinon	le résultat est le résultat de l'application d'une fonction sur le résultat de l'appel récursif

LES BUGS

- Mon algorithme est faux car son résultat n'est pas défini si la liste est vide ou si elle contient autre chose que des nombres
- Pour éviter les bugs il faut :
  - Définir le problème aussi bien que possible
    C'est la spécification
  - Tenter de prouver que son programme répond à la spécification
  - Passer des jeux d'essai, aussi divers et tordus que possible

Pour résumer

LIFAP1 : Programmation	**Impérative**	**Itérative**
Langage C	Séquence (faire les choses l'une après l'autre)	Boucle (répéter)
LIFAP2 : Programmation	**Fonctionnelle**	**Récursive**
Langage Scheme	Appliquer une fonction à des arguments pour obtenir un résultat	La répétition est assurée par l'enchaînement des appels récursifs
	Composer les fonctions pour enchaîner les traitements	Le test de la boucle est remplacé par le cas d'arrêt

Débuts en Scheme

Évaluer une expression

Définir une fonction

NOTION DE FONCTION

- Une fonction prend des arguments
  et retourne un résultat
- Arguments et résultat peuvent être de n'importe quel type :
  - Nombre
  - Booléen
  - Caractère
  - Chaîne de caractères
  - Liste
  - Fonction

ÉCRITURE DE L'APPEL À UNE FONCTION (1)

- Syntaxe :
  - Parenthèse ouvrante
  - Nom de la fonction
  - Espace
  - Premier argument
  - Espace
  - Deuxième argument
  - Etc.
  - Parenthèse fermante

(NomFct Arg1 Arg2 … Argn)

Écriture de l'appel à une fonction (2)

- Sémantique : il faut donner à la fonction le bon nombre d'arguments, et du bon type

- Exemples :
  - (+ 5 13) retourne 18
  - (- 10 b) retourne la différence
    si b a une valeur numérique, une erreur sinon
  - (+ (* 2 5) (- 3 1)) retourne 12
  - (* 5) n'est pas correct
  - (/ 5 "a") non plus

Évaluation de l'appel à une fonction

- Lorsqu'on lui fournit un appel de fonction, Scheme
  - Évalue chacun des arguments
  - Regarde s'il connaît la fonction,
    sinon affiche un message d'erreur
  - Applique la fonction aux résultats de l'évaluation des arguments
  - Affiche le résultat

- C'est un processus récursif

EXEMPLES D'ERREURS

- (1 + 2) → erreur : 1 n'est pas une fonction
- (toto (1 2 3)) → erreur : 1 n'est pas une fonction

- Dans certains cas particuliers, les arguments ne sont pas évalués avant l'application de la fonction.
  On parle alors de forme spéciale plutôt que de fonction

Les variables

- Dans le langage Scheme, une variable se nomme symbole
- L'affectation revient à attribuer une valeur à un symbole.
  On utilise pour cela la forme spéciale define

- Exemples :
  - (define a 5)
  - (define b 2)
  - (+ a b) → 7

LA FORME SPÉCIALE QUOTE

- La forme spéciale quote permet d'empêcher l'évaluation

- Exemples :
  - (define a 5)
  - a → 5

  - (quote a) → a
  - (quote (+ 1 2)) → (+ 1 2)
  - '(+ 1 2) → (+ 1 2)

La forme spéciale eval

- À l'inverse de quote, eval force l'évaluation

- Exemples :

```
(eval '(+ 3 2)) → 5

(define toto 5)
(define tata toto)
tata → 5

(define titi 'toto)
titi → toto
(eval titi) → 5
```

DÉFINITION D'UNE FONCTION

- Syntaxe :

```
(define fonction
        (lambda liste-des-arguments
                instructions))
```

- Exemple :

```
(define plus-1
        (lambda (x)
                (+ x 1)))
```

- Test : (plus-1 3) → 4

Spécification d'une fonction

```
; description de ce que fait la fonction
(define fonction ; → type du résultat
    (lambda liste-des-arguments ; type des arguments
        instructions))
```

Exemple :

```
    ; ajoute 1 à un nombre
        (define plus-1 ; → un nombre
            (lambda (x) ; x un nombre
                    (+ x 1)))
```

L'ALTERNATIVE

- L'alternative est définie en Scheme par la forme spéciale if

- Syntaxe :

  (if condition valeur-si-vrai valeur-si-faux)

- Exemples :
  - (if (> 3 2) 'yes 'no) → yes
  - (if (> 2 3) 'yes 'no) → no
  - (if (> 3 2) (- 3 2) (+ 3 2)) → 1

QUELQUES FONCTIONS PRÉDÉFINIES (1)

- Opérateurs arithmétiques :

  +, -, *, /, sqrt, min, max, abs, ...

  (sqrt 9) → 3
  (min 5 2 1 3) → 1

- Opérateurs booléens :

  not, or, and

  (not #t) → #f
  (and (> 3 2) (> 2 5)) → #f
  (or (> 3 2) (> 2 5)) → #t

Quelques fonctions prédéfinies (2)

- Opérateurs de comparaison :
  - =, <, >, <=, >= pour les nombres
  - eq? pour tout sauf les listes et les chaînes de caractères
  - equal? pour tout y compris les listes

- Pour tester le type d'un objet :
  boolean?, number?, symbol?, string?

- modulo : reste de la division entière
  - (modulo 12 5) → 2
  - (modulo 5 12) → 5

MA PREMIÈRE FONCTION RÉCURSIVE : CHOIX DE LA MÉTHODE

- On veut écrire une fonction qui étant donné un entier n calcule n!

Sur quoi faire l'appel récursif ?

appel récursif

$$\begin{tikzcd}
n \arrow[r, "-1"] \arrow[d, "\text{factorielle}"'] & n-1 \arrow[d, "\text{factorielle}"] \\
n! & (n-1)! \arrow[l, "\times n"]
\end{tikzcd}$$

Comment passer du résultat de l'appel récursif au résultat qu'on cherche ?

MA PREMIÈRE FONCTION RÉCURSIVE : ÉCRITURE

Cas d'arrêt : 0! = 1

Récursivité : n! = 1 x 2 x 3 x … x n = n x (n-1)!

```
(define factorielle ; → entier positif
  (lambda (n) ; n entier positif
      (if    (= n 0)
             1
             (* n (factorielle (- n 1))))))
```


M. Lefevre – N. Guin – F. Zara

UNE AUTRE FONCTION RÉCURSIVE : CHOIX DE LA MÉTHODE

- On veut écrire une fonction qui étant donné un nombre x et un entier positif n calcule x^n

Sur quoi faire l'appel récursif ?

$$
\begin{tikzcd}
x,n \arrow[r, "-1 \text{ sur } n"] \arrow[d, "\text{puissance}"'] & x,n-1 \arrow[d, "\text{puissance}"] \\
x^n & x^{n-1} \arrow[l, "\times x"]
\end{tikzcd}
$$

appel récursif

Comment passer du résultat de l'appel récursif au résultat qu'on cherche ?

Une autre fonction récursive : écriture

Cas d'arrêt : $X^0 = 1$

Récursivité : $X^n = X \times X \times \ldots \times X = X \times X^{(n-1)}$

```
(define puissance ; → nombre
  (lambda (x n) ; x nombre, n entier positif
      (if     (= n 0)
              1
              (* x (puissance x (- n 1))))))
```