Complexité des algorithmes

Complexité en temps

Complexité en espace

1

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

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

- Un mot pour stocker le « minimum pour l'instant » (min)
- Un mot pour savoir où on en est dans la liste (i)

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)
  = …
  = 1+1+…+C(1)
  = 1+1+…+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 LIF3 on s'intéressera essentiellement au nombre de fois où l'on parcourt une structure de donnée (liste, arbre)

Les listes en Scheme

car, cdr, cons

cond

list, append

Définition

- Une **liste** est une suite d'éléments rangés dans un certain ordre

```
'(alpha 3 beta  "delta" gamma)
```

Paires en Scheme

- Une **paire** est une structure d'enregistrement avec deux champs appelés car et cdr
  - car représente le premier élément
  - cdr représente le second élément
- Les champs car et cdr sont accédés par les fonctions car et cdr
- Une paire est représentée par la notation

  ' (element1 . element2)

  le . permet de faire la différence avec une liste de 2 éléments

Listes en Scheme

- Une liste peut être définie récursivement comme :
  - soit une liste vide : '()
  - soit une paire dont le cdr est une liste : ' (a . '(1 2 3))

Écriture d'une liste

- Les éléments d'une liste sont simplement compris entre parenthèses et séparés par des espaces
- La liste vide est écrite ' ()
- '(a b c d e) et ' (a . (b . (c . (d . (e . ()))))) sont deux notations équivalentes pour une liste de symboles

La fonction pair?

- (pair? x) retourne
  - #t si x est une paire,
  - #f sinon
- Exemples :
  - (pair? '(a . b)) → #t
  - (pair? '(a b c)) → #t
  - (pair? '()) → #f

Les fonctions car et cdr

- (car x) retourne le champ car de x
  - (car '(a b c)) → a
  - (car '((a) b c d)) → (a)
  - (car '(1 . 2)) → 1
  - (car '()) → error
- (cdr x) retourne le champ cdr de x
  - (cdr '((a) b c d)) → (b c d)
  - (cdr '(1 . 2)) → 2
  - (cdr '(a)) → ()
  - (cdr '()) → error
- Si x est une liste, la fonction car retourne le premier élément de la liste, et la fonction cdr retourne le reste de la liste

La fonction cons

- (cons x y) retourne une nouvelle paire dont le car est x et le cdr est y
  si y est une liste, elle met x au début de y
- Exemples :
  - (cons 'a '()) → (a)
  - (cons '(a) '(b c d)) → ((a) b c d)
  - (cons "a" '(b c)) → ("a" b c)
  - (cons 'a 3) → (a . 3)
  - (cons '(a b) 'c) → ((a b) . c)
- Pour que le résultat du cons soit une liste, il faut donc que le deuxième argument soit une liste

Fonctions de test

- (list? x) retourne #t si x est une liste, #f sinon
  - (list? '(a b c)) → #t
  - (list? '()) → #t
  - (list? '(a . b)) → #f
- (null? x) retourne #t si x est la liste vide, #f sinon

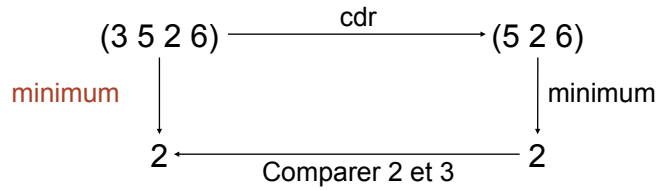
Implémentation de l'algorithme récursif du minimum

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
```

```
(define minimum ; → nombre
      (lambda (l) ; l liste de nombres non vide
   (if (null? (cdr l))
      (car l)
    (if  (<  (car l)(minimum (cdr l)))
                  (car l)
                  (minimum (cdr l))))))
```

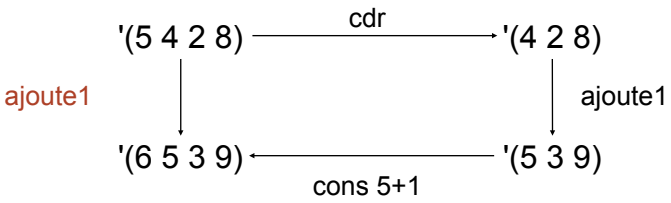
Illustration de la méthode



Une fonction qui retourne une liste

- Une fonction qui ajoute 1 à tous les éléments d'une liste de nombres

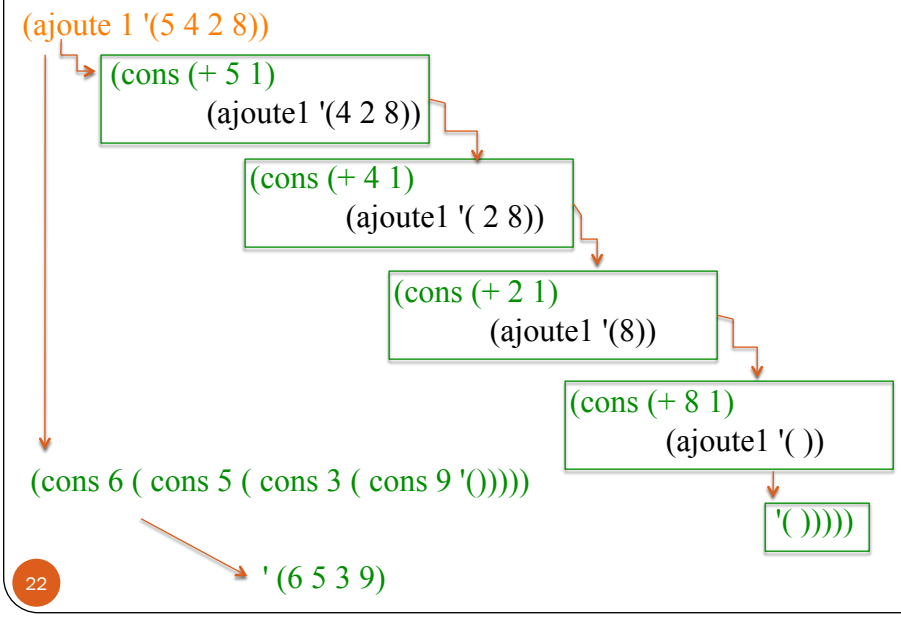
(ajoute1 '(5 4 2 8)) → (6 5 3 9)



Écriture de la fonction

```
(define ajoute1; → liste de nombres
   (lambda (l) ; l liste de nombres
         (if  (null? l)
                '()
                (cons (+ (car l) 1)
                        (ajoute1 (cdr l)))))
```


Illustration de la fonction



Relâcher des préconditions

Une fonction qui ajoute 1 à tous les nombres d'une liste

```
(define ajoute1 ; → liste
  (lambda (l) ; l liste
   (if  (null? l)
            '()
            (if (number? (car l))
                     (cons   (+ (car l) 1)
                              (ajoute1 (cdr l)))
                     (cons (car l)
                              (ajoute1 (cdr l)))))))
```

(ajoute1 '(2 "a" 1 (toto) 5)) → (3 "a" 2 (toto) 6)

La forme spéciale cond

- La forme spéciale cond permet d'écrire plus simplement des if imbriqués

```
(if test1 valeur1
        (if test2 valeur2
                ...
                        valeurN) ...))
```

→

```
(cond
        (test1 valeur1)
        (test2 valeur2)
        ...
        (else valeurN))
```

Application à ajoute1

```
(define ajoute1 ; → liste
  (lambda (l) ; l liste
    (cond
      ((null? l) '())
      ((number? (car l)) (cons (+ (car l) 1)
                               (ajoute1 (cdr l))))
      (else (cons (car l) (ajoute1 (cdr l)))))))
```


Mettre en facteur dans un if

```
(if (number? (car l))
    (cons (+ (car l) 1)
          (ajoute1 (cdr l)))
    (cons (car l)
          (ajoute1 (cdr l))))
```

```
(cons
   (if (number? (car l))
       (+ (car l) 1)
       (car l))
   (ajoute1 (cdr l)))
```

Ces deux expressions sont équivalentes

La fonction list

- La fonction list est une fonction qui permet de construire une liste, comme la fonction cons
- Elle prend un nombre quelconque d'arguments et les met tous dans une liste

(list 'a (+ 3 2) "toto" 'b) → (a 5 "toto" b)

Ne pas confondre list et list?

La fonction append

- La fonction append permet de concaténer des listes
- Elle prend un nombre quelconque d'arguments

(append '(a (b c) d) '(e f)) → (a (b c) d e f)

cons, list et append : les différences

Ces trois fonctions permettent toutes de construire des listes, mais ont chacune un comportement différent

- cons prend deux arguments, le deuxième étant obligatoirement une liste
- list prend un nombre quelconque d'arguments de types quelconques
- append prend un nombre quelconque d'arguments qui doivent tous être des listes

cons, list et append : exemples

- (cons '(a b) '(c d)) → ((a b) c d)
- (list '(a b) '(c d)) → ((a b) (c d))
- (append '(a b) '(c d)) → (a b c d)

Raccourcis pour composer les fonctions car et cdr

- Au lieu d'écrire (car (cdr L)), on peut écrire (cadr L)
- Au lieu d'écrire (cdr (cdr (cdr L))), on peut écrire (cdddr L)
- Et ainsi de suite (au maximum 4 caractères entre le c et le r)