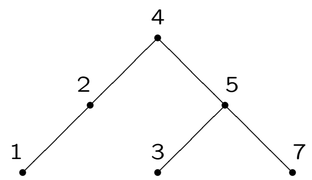
## Arbres binaires complètement étiquetés I

Un $K$-*arbre binaire complètement étiqueté* est soit l'arbre vide, soit un triplet comportant une *clef* ("key") élément de $K$, un sous-arbre de gauche et un sous-arbre de droite. On aura donc un constructeur sans argument pour l'arbre vide et un constructeur à trois arguments pour les arbres non vides ; on aura également trois accesseurs.

Exemple, $K = \mathbb{N}$




Représentation concrète simple : liste de trois éléments.

```
(define conc-tree '(4 (2 (1 () ()) ()) (5 (3 () ()) (7 () ()))))
```



## Arbres binaires complètement étiquetés II

Constructeurs :

```
(define mk-e-tree (lambda () '()))
(define mk-k-tree (lambda (k l r) (list k l r)))
```

Reconnaisseurs :

```
(define e-tree? null?)
(define k-tree?
  (lambda (x)  ;; x est un objet quelconque
    (and (pair? x) (key? (car x))
         (pair? (cdr x)) (ek-tree? (cadr x))
         (pair? (cddr x)) (ek-tree? (caddr x))
         (null? (cdddr x)))))
(define ek-tree? (lambda (x) (or (e-tree? x) (k-tree? x))))
(define key? (lambda (x) (and (integer? x) (>= x 0))))
```

Accesseurs :

```
(define key car)   (define left cadr)   (define right caddr)
```



## Arbres binaires complètement étiquetés III

Représentations concrète et abstraite :

```
(define conc-tree
  '(4 (2 (1 () ()) ()) (5 (3 () ()) (7 () ()))))

(define abst-tree  ;; corriger p. 189
  (mk-k-tree 4
           (mk-k-tree 2
                    (mk-k-tree 1
                             (mk-e-tree)
                             (mk-e-tree))
                    (mk-e-tree))
           (mk-k-tree 5
                    (mk-k-tree 3
                             (mk-e-tree)
                             (mk-e-tree))
                    (mk-k-tree 7
                             (mk-e-tree)
                             (mk-e-tree)))))
```



## Arbres binaires complètement étiquetés IV

Présence d'un nombre entier donné dans un $\mathbb{N}$-arbre donné

```
(define in-tree?
  (lambda (i tr)
    (if (e-tree? tr)
        #f
        (or (= i (key tr))
            (in-tree? i (left tr))
            (in-tree? i (right tr))))))
```

Ceci peut se récrire en :

```
(define in-tree?
  (lambda (i tr)
    (and (not (e-tree? tr))
         (or (= i (key tr))
             (in-tree? i (left tr))
             (in-tree? i (right tr))))))
```

## Arbres binaires conditionnés I

Un arbre non vide est dit *conditionné* ou *ordonné* si la clef de tout nœud interne est plus grande ou égale aux clefs de tous ses descendants de gauche, et plus petite ou égale aux clefs de tous ses descendants de droite.

Deux écueils à éviter.

L'approche "naïve" : Un arbre non vide serait conditionné si la clef de la racine est comprise entre les clefs des deux fils (s'il existent) et si les deux sous-arbres fils sont eux-mêmes conditionnés. La condition est nécessaire mais pas suffisante (voir exemple) ! ! !

L'approche "prudente" : un arbre non vide serait conditionné si la clef de la racine est supérieure à tous ses descendants de gauche et inférieure à tous ses descendants de droite et si les deux sous-arbres fils sont eux-mêmes conditionnés. La méthode est inefficace, parce que les mêmes comparaisons sont répétées plusieurs fois.

En fait, un arbre est conditionné si ses deux fils sont conditionnés et si sa racine est supérieure à tous les éléments de la branche la plus à droite du fils gauche, et inférieure à tous les éléments de la branche la plus à gauche du fils droit.



## Arbres binaires conditionnés II

Les prédicats auxiliaires `greq?` et `leeq?` testent les deux dernières conditions.

```
(define greq?
  (lambda (n tr)
    (or (e-tree? tr) (and (>= n (key tr)) (greq? n (right tr))))))

(define leeq?
  (lambda (n tr)
    (or (e-tree? tr) (and (<= n (key tr)) (leeq? n (left tr))))))

(define condit-1?
  (lambda (tr)
    (or (e-tree? tr)
        (and (condit-1? (left tr)) (greq? (key tr) (left tr))
             (condit-1? (right tr)) (leeq? (key tr) (right tr))))))
```

Ce programme n'est pas optimal ; une version plus efficace est possible si on dispose d'une borne supérieure `*max*` absolue pour les étiquettes des arbres.



## Arbres binaires conditionnés III

Version efficace

```
(define condit-2?
  (lambda (tr) (or (e-tree? tr) (tree-ok? 0 tr *max*))))

(define tree-ok?
  (lambda (min tr max)
    (and (<= min (key tr))
         (>= max (key tr))
         (or (e-tree? (left tr)) (tree-ok? min (left tr) (key tr)))
         (or (e-tree? (right tr)) (tree-ok? (key tr) (right tr) max)))))
```

`tr` est un arbre conditionné dont toutes les clefs sont comprises entre les naturels `min` et `max` ssi `(tree-ok? min tr max)` est vrai.



## Arbres binaires conditionnés IV

Si un arbre est conditionné, la liste de ses étiquettes est triée, à condition que dans cette liste toute étiquette se trouve entre les étiquettes de ses descendants de gauche et celles de ses descendants de droite.

La fonction `traversal` calcule cette liste :

```
(define traversal
  (lambda (tr)
    (if (e-tree? tr)
        '()
        (append (traversal (left tr))
                (cons (key tr)
                      (traversal (right tr)))))))
```

## Arbres binaires conditionnés V

On peut utiliser la technique des accumulateurs pour éviter l'usage de `append`, en écrivant une fonction auxiliaire `trav-a` telle que

[[`(trav-a tr acc)`]]

soit égal à

[[`(append (traversal tr) acc)`]]

```
(define trav-a
  (lambda (tr acc)
    (if (e-tree? tr)
        acc
        (trav-a (left tr)
                (cons (key tr)
                      (trav-a (right tr) acc))))))

(define traversal
  (lambda (tr) (trav-a tr '())))
```



## Arbres, tas et tri I

Une arbre est un *tas* si l'étiquette d'un nœud est supérieure aux étiquettes de ses descendants. La notion de tas est utile dans diverses applications. Le programme `heap?` teste si un arbre est un tas ("heap" en anglais) ; il est analogue au programme `condit-2`.

Les règles de portée empêchent toute confusion entre les liaisons locales et globales de `key`, `left` et `right` ; les liaisons locales sont des arbres, les liaisons globales sont des accesseurs.

```
(define heap?
  (lambda (tr)
    (or (e-tree? tr)
        (let ((key (key tr)) (left (left tr)) (right (right tr)))
          (and (greq? key left) (greq? key right)
               (heap? left) (heap? right))))))

(greq? n tr) :
(or (e-tree? tr) (>= n (key tr))
```



## Arbres, tas et tri II

Transformation d'un arbre en un tas

```
(define heapify
  (lambda (tr)
    (if (e-tree? tr)
        tr
        (let ((key (key tr)) (left (left tr)) (right (right tr)))
          (let ((lh (heapify left)) (rh (heapify right)))
    (adjust key lh rh))))))

(adjust 9 '(3 () (4 () ())) '(6 () ()))
       (9  (3 () (4 () ()))  (6 () ()))

(adjust 5 '(3 () (4 () ())) '(6 () ()))
       (6  (3 () (4 () ()))  (5 () ()))

(adjust 2 '(3 () (4 () ())) '(6 () ()))
       (6  (3 () (4 () ()))  (2 () ()))
```



## Arbres, tas et tri III

```
(define adjust
  (lambda (ky lh rh)
    (cond
      ((and (greq? ky lh) (greq? ky rh)) (mk-k-tree ky lh rh))
      ((greq? ky lh)
       (let ((krh (key rh)) (lrh (left rh)) (rrh (right rh)))
         (mk-k-tree krh lh (adjust ky lrh rrh))))
      ((greq? ky rh)
       (let ((klh (key lh)) (llh (left lh)) (rlh (right lh)))
         (mk-k-tree klh (adjust ky llh rlh) rh)))
      (else
       (let ((klh (key lh)) (krh (key rh)))
         (let ((llh (left lh)) (rlh (right lh))
               (lrh (left rh)) (rrh (right rh)))
           (if (> klh krh)
               (mk-k-tree klh (adjust ky llh rlh) rh)
               (mk-k-tree krh lh (adjust ky lrh rrh))))))))))
```

## Arbres, tas et tri IV

Si dans la liste des étiquettes d'un tas, l'étiquette d'un nœud vient toujours avant l'étiquette des descendants de ce nœud, alors la liste est "presque" triée par ordre décroissant. Une variante du prédicat `traversal` permet de le vérifier. Nous écrivons cette variante en utilisant un `letrec` et un accumulateur :

```
(define pre-trav
  (lambda (tr)
    (letrec
      ((pre-trav-a
        (lambda (tr acc)
          (if (e-tree? tr)
              acc
              (cons (key tr)
                    (pre-trav-a (left tr) (pre-trav-a (right tr) acc)))))))
      (pre-trav-a tr '()))))
```

Exercice : spécifier la fonction auxiliaire



## Arbres, tas et tri V

Considérons l'arbre dont la représentation concrète est

```
(5 (3 (5 (3 () (5 (4 () (7 () ())) (3 () ()))) (6 () ()))
      (4 (5 (3 () (4 () ())) (6 () ())) ()))
   (6 () (5 (4 () (7 () ())) (3 () ()))))
```

`heapify` transforme cet arbre en le tas

```
(7 (7 (6 (5 () (5 (4 () (3 () ())) (3 () ()))) (3 () ()))
      (6 (5 (4 () (3 () ())) (4 () ())) ()))
   (6 () (5 (5 () (4 () ())) (3 () ()))))
```

qui a même structure ; `pre-trav` fournit les listes

```
(3 4 7 5 3 5 6 3 3 4 5 6 4 5 6 4 7 5 3)
(7 7 6 5 5 4 3 3 3 6 5 4 3 4 6 5 5 4 3)
```

alors que la version triée de ces deux listes est

```
(7 7 6 6 6 5 5 5 5 5 4 4 4 4 3 3 3 3 3)
```

Ceci suggère qu'il devrait exister une variante de `pre-trav` qui, appliquée à un tas, fournirait la liste triée des étiquettes de ce tas.



## Arbres, tas et tri VI

```
(define hp-sort-trav  ;; liste triée des étiquettes
  (lambda (hp)
    (if (e-tree? hp)
        '()
        (let ((u1 (hp-sort-trav (left hp)))
              (u2 (hp-sort-trav (right hp))))
          (cons (key hp) (merge u1 u2))))))

(define merge  ;; fusion de deux listes triées
  (lambda (u1 u2)
    (cond ((null? u1) u2)
          ((null? u2) u1)
          (else (let ((a1 (car u1)) (a2 (car u2)))
                  (if (> a1 a2)
                      (cons a1 (merge (cdr u1) u2))
                      (cons a2 (merge u1 (cdr u2))))))))))
```

Cette technique de tri est raisonnablement efficace.



## Visualisation de la structure des arbres

```
(define space      ;; (space n) ecrit n blancs
  (lambda (n)
    (if (zero? n)
        (display "")
        (begin (display " ") (space (- n 1))))))

(define pr-tree
  (lambda (tr d)
    (if (e-tree? tr)
        (begin (space d) (display " -"))
        (let ((d1 (1+ d)))
          (begin (space d1) (display (key tr))
                 (newline) (pr-tree (left tr) d1)
                 (newline) (pr-tree (right tr) d1)))))))
```

## Enregistrements, réalisation concrète I

Les $K$-arbres binaires complètement étiquetés sont des cas particuliers d'enregistrements. Un *enregistrement* est une structure de donnée admettant un nombre fixé de composants, chacun d'eux ayant un type donné.

> *Le reconnaisseur* `record?` *prend comme arguments un objet* [[u]] *et une liste de propriétés* [[lp]] *et renvoie* `#t` *si* [[u]] *et* [[lp]] *sont des listes de même longueur $\ell$ et si pour tout $i = 1, \ldots, \ell$, le $i$ème objet de* [[u]] *satisfait la $i$ème propriété de* [[lp]].

*Remarque.* Une propriété est ici un prédicat à un argument.

Une solution simple et efficace est

```
(define record?
  (lambda (u lp)
    (or (and (null? u) (null? lp))
        (and (pair? u) (pair? lp) ((car lp) (car u))
             (record? (cdr u) (cdr lp))))))
```



## Enregistrements, réalisation concrète II

Le reconnaisseur `k-tree?` introduit plus haut, à savoir

```
(define k-tree?
  (lambda (x)  ;; x est un objet quelconque
    (and (pair? x) (key? (car x))
         (pair? (cdr x)) (ek-tree? (cadr x))
         (pair? (cddr x)) (ek-tree? (caddr x))
         (null? (cdddr x)))))
(define e-tree? null?)
(define ek-tree? (lambda (x) (or (e-tree? x) (k-tree? x))))
(define key? (lambda (x) (and (integer? x) (>= x 0))))
```

pourrait aussi se définir en utilisant `record?` :

```
(define k-tree?
  (lambda (u)
    (record?
     u
     (list (lambda (x) (and (integer? x) (>= x 0)))
           (lambda (v) (or (null? v) (k-tree? v)))
           (lambda (v) (or (null? v) (k-tree? v)))))))
```



## Les graphes I

```
(define *g0*
 '((a b c d e f) .
   ((b . a) (b . d) (c . b) (d . c) (d . f) (e . b) (e . f) (f . a))))

(define mk-graph (lambda (nodes arcs) (cons nodes arcs)))

(define nodes (lambda (gr) (car gr)))

(define arcs (lambda (gr) (cdr gr)))

(define mk-arc (lambda (org ext) (cons org ext)))

(define org (lambda (arc) (car arc)))

(define ext (lambda (arc) (cdr arc)))
```



## Les graphes II

```
(define *altg2*
  '((a . (b d g h k l o p))
    (b . (c e f p))
    (c . (a e i o))
    (d . ())
    (e . (b k m))
    (f . (c d g m n p))
    (g . (a e j p))
    (h . (b d k l n))
    (i . (b c e m o))
    (j . (c))
    (k . (a f g n p))
    (l . (b k))
    (m . (c e o))
    (n . (a i m))
    (o . (b d e))
    (p . (i j k))))
```

## Les graphes III

```
(define nodes
  (lambda (altgr) (map car altgr)))

(define arcs
  (lambda (altgr) (union-map gen-arcs altgr)))

(define union-map
  (lambda (f l)
    (if (null? l)
        '()
        (union (f (car l)) (union-map f (cdr l))))))

(define gen-arcs
  (lambda (nsuccs)
    (let ((n (car nsuccs)) (succs (cdr nsuccs)))
      (map (lambda (x) (mk-arc n x)) succs))))
```



## Les graphes IV

```
(mk-graph (nodes *altg2*) (arcs *altg2*))
==>
((a b c d e f g h i j k l m n o p)
 (a . b) (a . d) (a . g) (a . h) (a . k) (a . l) (a . o) (a . p)
 (b . c) (b . e) (b . f) (b . p)
 (c . a) (c . e) (c . i) (c . o)
 (e . b) (e . k) (e . m)
 (f . c) (f . d) (f . g) (f . m) (f . n) (f . p)
 (g . a) (g . e) (g . j) (g . p)
 (h . b) (h . d) (h . k) (h . l) (h . n)
 (i . b) (i . c) (i . e) (i . m) (i . o)
 (j . c)
 (k . a) (k . f) (k . g) (k . n) (k . p)
 (l . b) (l . k)
 (m . c) (m . e) (m . o)
 (n . a) (n . i) (n . m)
 (o . b) (o . d) (o . e)
 (p . i) (p . j) (p . k))
```



## Les graphes V

Successeurs d'un nœud dans un graphe

```
(define succs
  (lambda (nd gr)
    (let ((nodes (nodes gr)) (arcs (arcs gr)))
      (if (member nd nodes)
          (succs-arcs nd arcs)
          (error "unknown node" nd)))))

(define succs-arcs
  (lambda (nd arcs)
    (cond ((null? arcs) '())
          ((equal? (org (car arcs)) nd)
           (add-elem (ext (car arcs)) (succs-arcs nd (cdr arcs))))
          (else (succs-arcs nd (cdr arcs))))))

(define add-elem
  (lambda (x l) (if (member x l) l (cons x l))))
```



## Les graphes VI

```
(define offspring  ;; Descendance d'un noeud dans un graphe, version naive
  (lambda (nd gr) (off nd gr (length (nodes gr)))))

(define off
  (lambda (nd gr k)
    (if (= k 0) '() (add-elem nd (off* (succs nd gr) gr (- k 1))))))

(define off*
  (lambda (nd* gr k)
    (if (null? nd*)
        '()
        (union (off (car nd*) gr k) (off* (cdr nd*) gr k)))))

(define union
  (lambda (u v)
    (if (null? u) v (add-elem (car u) (union (cdr u) v)))))
```

## Les graphes VII

Descendance d'un nœud dans un graphe, deuxième solution

```
(define offspring-bis
  (lambda (nd gr)
    (off*-bis (list nd) gr '())))

(define off*-bis
  (lambda (nd* gr acc)
    (cond ((null? nd*) acc)
          ((member (car nd*) acc)
           (off*-bis (cdr nd*) gr acc))
          (else
           (off*-bis (append (succs (car nd*) gr) (cdr nd*))
                     gr
                     (cons (car nd*) acc))))))
```



## Les graphes VIII

Descendance d'un nœud dans un graphe, troisième solution

```
(define offspring-ter
  (lambda (nd gr) (off-ter nd gr '())))

(define off-ter
  (lambda (nd gr acc)
    (if (member nd acc) acc (off*-ter (succs nd gr) gr (cons nd acc)))))

(define off*-ter
  (lambda (nd* gr acc)
    (cond ((null? nd*) acc)
          ((member (car nd*) acc)
           (off*-ter (cdr nd*) gr acc))
          (else
           (off-ter (car nd*)
                    gr
                    (off*-ter (cdr nd*) gr acc))))))
```



## Les graphes IX

Essais.

Dans [[*g0*]], on a les arcs

```
c->b  b->a
      b->d d->c
           d->f f->a
```

donc la descendance de [[c]] dans [[*g0*]] comporte

[[a]], [[b]], [[c]], [[d]] et [[f]].

On a effectivement

```
(offspring     'c *g0*) (c b d f a)

(offspring-bis 'c *g0*) (f d a b c)

(offspring-ter 'c *g0*) (a f d b c)
```



# 11. Un exercice de programmation

*Buts :*
– *Raisonnement récursif*. Le type des arguments suggère souvent un schéma de récursion approprié.
– *Programmation "top-down"* . On définit d'abord la procédure principale, puis les procédures auxiliaires s'il y a lieu.
– *Abstraction sur les données*.

**Enoncé.** On a une collection d'objets ; chaque objet a un *poids* (naturel non nul) et une *utilité* (réel strictement positif). On se donne aussi un *poids maximal* (nombre naturel). Un *chargement* est une sous-collection d'objets ; le poids d'un chargement est naturellement la somme des poids des objets qu'il contient ; son utilité est la somme des utilités.
Le problème consiste à déterminer le chargement d'utilité maximale, dont le poids n'excède pas le poids maximal.

*Remarques*. Le problème du "sac à dos" (*knapsack*) est NP-complet.
Il a des applications en cryptographie.

## Stratégie : récursivité structurelle (mixte)

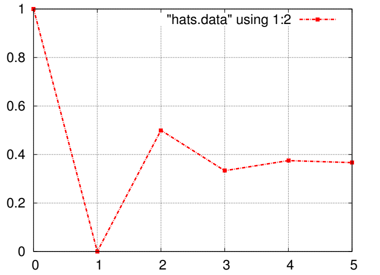
L'idée algorithmique utile ici est d'application fréquente dans les problèmes combinatoires. Elle consiste simplement à répartir les entités à considérer ou à dénombrer en deux classes, que l'on traite séparément (appels récursifs), puis à combiner les deux résultats partiels. On rappelle d'abord trois exemples classiques.

- Nombre $C(n,k)$ de choix de $k$ objets parmi $n$ ($0 \leq k \leq n$) ?
  Cas de base, $k = 0$ ou $k = n$, $C(n,k) = 1$
  Cas inductif, $0 < k < n$, soit $X$ un objet
  $X$ inclus : $C(n-1,k-1)$
  $X$ exclu : $C(n-1,k)$
  total : $C(n-1,k-1) + C(n-1,k)$ .

- Nombre de partages de $n$ objets distincts en $k$ lots non vides ?
  Cas de base, $k = n$, $P(n,k) = 1$
  $k = 0 < n$, $P(n,k) = 0$
  Cas inductif, $0 < k < n$, soit $X$ un objet
  $X$ isolé : $P(n-1,k-1)$
  $X$ non isolé : $k\,P(n-1,k)$
  total : $P(n-1,k-1) + k\,P(n-1,k)$ .



## Les dérangements




- Combien de "dérangements" de $(1,2,\ldots,n)$ ?
  (On doit avoir $p(i) \neq i$, pour tout $i$.)
  Cas de base, $n = 0$, $D(n) = 1$
  $n = 1$, $D(n) = 0$
  Cas inductif, $n \geq 2$, soit $p(n) = i$ $(i \in \{1,\ldots,n-1\})$.
  $p(i) = n$ : $(n-1)\,D(n-2)$ cas
  $p(i) \neq n$ : $(n-1)\,D(n-1)$ cas
  total : $(n-1)\,(D(n-2) + D(n-1))$ .

*Problème des chapeaux.* Si $n$ personnes mélangent leurs chapeaux puis se les réattribuent au hasard, la probabilité que personne ne récupère le sien est :

$$P(n) = D(n)/n!$$

On note une convergence rapide vers $e^{-1} = 0.367879...$ (voir graphique).



## Stratégie récursive mixte pour le problème "sac à dos"

*Cas de base.*
La collection est vide ou le poids maximal est nul.
La solution est le chargement vide, d'utilité nulle.

*Cas inductif.*
La collection $C$ n'est pas vide et le poids maximal $L$ est strictement positif.

Par rapport à un objet arbitraire $X$ de la collection, il y a deux types de chargements : ceux qui négligent $X$ (type I) et ceux qui contiennent $X$ (type II). Un chargement de type I est relatif à la collection $C \setminus \{X\}$ et au poids maximal $L$. Un chargement de type II comporte $X$, plus un chargement relatif à la collection $C \setminus \{X\}$ et au poids maximal $L - p(X)$ ; le type II n'existe pas si $L < p(X)$.

La tactique est donc de calculer, séparément, les deux solutions optimales, relatives à une collection amputée d'un élément (parfois une seule, si $L < p(X)$), puis d'en déduire le chargement optimal pour la collection donnée.



## Types abstraits de données

Le type "collection" est récursif. On a
– La constante de base `the-empty-coll` ;
– Le constructeur `add-obj-coll` (deux arguments) ;
– Les reconnaisseurs `coll?` et `empty-coll?` ;
– Les accesseurs `obj-coll` et `rem-coll`.

Pour le type non récursif "objet", on a
– Le constructeur `mk-obj` (deux arguments) ;
– Le reconnaisseur `obj?` ;
– Les accesseurs `poids` et `utilite`.

On écrit facilement les relations algébriques induites par ces définitions. Par exemple, dans un environnement où `x` et `c` sont liés respectivement à un objet et à une collection, si `(empty-coll? c)` a la valeur `#f`, les expressions `c` et `(add-obj-coll (obj-coll c) (rem-coll c))` ont même valeur.

On utilise aussi un type `solution` ; un objet de ce type comporte une collection avec son poids total et son utilité totale ; on aura notamment le constructeur `mk-sol` et les trois accesseurs `char`, `ptot` et `utot`.

## Développement du programme I

```
(define knap
  (lambda (pm c)
    (if (or (= 0 pm) (empty-coll? c)) ;; double cas de base
        (mk-sol the-empty-coll 0 0)
        (...))))
```

La partie à préciser concerne le cas inductif. Son traitement requiert la distinction d'un objet x de c et, au moins, le calcul récursif d'une solution de type I. On obtient

```
(define knap
  (lambda (pm c)
    (if (or (= 0 pm) (null? c))
        (mk-sol the-empty-coll 0 0)
        (let* ((x (obj-coll c))
               (rc (rem-coll c))
               (s1 (knap pm rc)))
          (...)))))
```



## Développement du programme II

Pour savoir si on devra aussi considérer une solution de type II, il faut comparer le poids de x au poids maximal ; on a

```
(define knap
  (lambda (pm c)
    (if (or (= 0 pm) (null? c))
        (mk-sol the-empty-coll 0 0)
        (let* ((x (obj-coll c)) (rc (rem-coll c))
               (s1 (knap pm rc))
               (px (poids x)) (ux (utilite x)))
          (if (>= pm px)
              (let ((s2 (knap (- pm px) rc)))
                (...)
              s1)))))
```



## Développement du programme III

Il reste à déterminer si la solution cherchée est s1, ou la solution obtenue en ajoutant x à s2.

```
(define knap
  (lambda (pm c)
    (if (or (= 0 pm) (null? c))
        (mk-sol the-empty-coll 0 0)
        (let* ((x (obj-coll c))
               (rc (rem-coll c))
               (s1 (knap pm rc))
               (px (poids x))
               (ux (utilite x)))
          (if (>= pm px)
              (let ((s2 (knap (- pm px) rc)))
                (if (> (+ (utot s2) (utilite x)) (utot s1))
                    (mk-sol (add-obj-coll x (char s2))
                            (+ px (ptot s2))
                            (+ ux (utot s2)))
                    s1))
              s1)))))
```



## Version finale

Il reste à déterminer si la solution cherchée est s1, ou la solution obtenue en ajoutant x à s2.

```
(define knap
  (lambda (pm c)
    (if (or (= 0 pm) (null? c))
        (mk-sol the-empty-coll 0 0)
        (let* ((x (obj-coll c)) (rc (rem-coll c))
               (s1 (knap pm rc))
               (px (poids x)) (ux (utilite x)))
          (if (>= pm px)
              (let ((s2 (knap (- pm px) rc)))
                (if (> (+ (utot s2) (utilite x)) (utot s1))
                    (mk-sol (add-obj-coll x (char s2))
                            (+ px (ptot s2))
                            (+ ux (utot s2)))
                    s1))
              s1)))))
```

## Structures de donnée

On peut réaliser le type abstrait `collection` par le type `list`, avec les correspondances suivantes :

```
the-empty-coll          '()
add-obj-coll            cons
coll?  empty-coll?      list?  null?
obj-coll  rem-coll      car  cdr
```

Le type `objet` est concrétisé par le type `pair` :

```
mk-obj                  cons
obj?                    pair?
poids  utilite          car  cdr
```

Pour le type `solution`, on utilise aussi le type `pair`, restreint au cas où la deuxième composante est aussi de type `pair` :

```
(mk-sol c p u)          (cons c (cons p u))
char  ptot  utot        car  cadr  cddr
```



## Essais

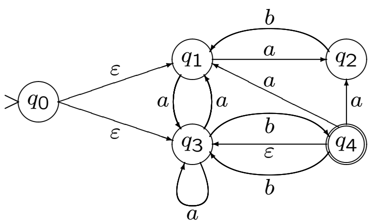
```
c       '((4 . 9) (3 . 8) (2 . 4) (1 . 1) (2 . 3)
          (6 . 9) (5 . 5) (4 . 8) (1 . 2) (2 . 1)
          (1 . 2) (1 . 1) (7 . 9) (6 . 6) (5 . 4)
          (4 . 5) (3 . 2) (2 . 3) (2 . 2) (3 . 3)))

(knap 0 c)      (()                                  0 . 0)

(knap 5 c)      ((3 . 8) (1 . 2) (1 . 2))            5 . 12)

(knap 10 c)     (((4 . 9) (3 . 8) (2 . 4) (1 . 2))  10 . 23)

(knap 15 c)     (((4 . 9) (3 . 8) (2 . 4) (4 . 8)
                  (1 . 2) (1 . 2))                  15 . 33)

(knap 20 c)     (((4 . 9) (3 . 8) (2 . 4) (6 . 9)
                  (4 . 8) (1 . 2))                  20 . 40)

(knap 25 c)     (((4 . 9) (3 . 8) (2 . 4) (2 . 3) (6 . 9) (4 . 8)
                  (1 . 2) (1 . 2) (2 . 3))          25 . 48)
```

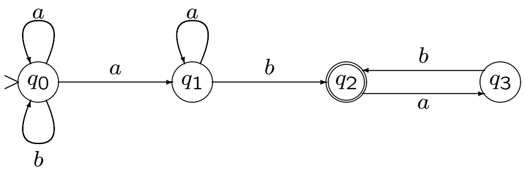
| $v$ | 5 | 10 | 15 | 20 | 25 |
|---|---|---|---|---|---|
| Temps | 1 | 6 | 32 | 141 | 317 |

Le comportement est typiquement exponentiel.



## Déterminisation d'un automate I






## Déterminisation d'un automate II

```
(define *autom01*
  (list->automaton
    '((q0 q1 q2 q3 q4)
      (a b)
      ((q0 () q1)  (q0 () q3)  (q1 (a) q2) (q1 (a) q3)
       (q2 (b) q1) (q3 (a) q1) (q3 (a) q3) (q3 (b) q4)
       (q4 (a) q1) (q4 (a) q2) (q4 () q3)  (q4 (b) q3))
      q0
      (q4))))

(define *autom02*
  (list->automaton
    '((q0 q1 q2 q3)
      (a b)
      ((q0 (a) q0) (q0 (b) q0) (q0 (a) q1) (q1 (a) q1)
       (q1 (b) q2) (q2 (a) q3) (q3 (b) q2))
      q0
      (q2))))
```

## Déterminisation d'un automate III

Réalisation du type automate

```
(define list->automaton (lambda (x) x))

(define mk-aut list)        (define mk-trans list)

(define states car)         (define orig car)
(define alph cadr)          (define word cadr)
(define trans caddr)        (define extr caddr)
(define init cadddr)
(define finals caddddr)

(define caddddr (lambda (x) (car (cddddr x))))
```



## Déterminisation d'un automate IV

```
(define filter
  (lambda (p? l)
    (cond ((null? l) '())
          ((p? (car l))
           (cons (car l) (filter p? (cdr l))))
          (else (filter p? (cdr l))))))

(define map-filter
  (lambda (f p? l)
    (if (null? l)
        '()
        (let ((rec (map-filter f p? (cdr l))))
          (if (p? (car l))
              (cons (f (car l)) rec)
              rec)))))
```



## Déterminisation d'un automate V

```
(define extend
  (lambda (aut q)
    (let ((states (states aut))
          (trans (trans aut)))
      (let ((arcs
             (map-filter
               (lambda (tr)
                 (mk-arc (orig tr) (extr tr)))
               (lambda (tr)
                 (null? (word tr)))
               trans)))
        (offspring-ter q
                       (mk-graph states arcs))))))

(define s-extend
  (lambda (aut q)
    (sort (extend aut q))))
```



## Déterminisation d'un automate VI

```
(define union*
  (lambda (l)
    (if (null? l) '() (union (car l) (union* (cdr l))))))

(define extend*
  (lambda (aut q*) (union* (map (lambda (q) (extend aut q)) q*))))

(define next
  (lambda (aut q x)
    (let ((trans (trans aut)))
      (map-filter
        extr
        (lambda (tr)
          (and (equal? (orig tr) q) (equal? (word tr) (list x))))
        trans))))

(define next*
  (lambda (aut q* x) (union* (map (lambda (q) (next aut q x)) q*))))
```

## Déterminisation d'un automate VII

```
(define del (lambda (aut q* x) (extend* aut (next* aut q* x))))

(define s-del (lambda (aut q* x) (sort (del aut q* x))))

(define determinize
  (lambda (aut)
    (let ((states (states aut)) (alph (alph aut)) (trans (trans aut))
          (init (init aut)) (finals (finals aut)))
      (let ((new-states (subsets states)))
        (mk-aut new-states
                alph
                (union-map
                  (lambda (x)
                    (s-map (lambda (ns) (mk-trans ns x (s-del aut ns x)))
                           new-states))
                  alph)
                (s-extend aut init)
                (filter (lambda (ns) (inter? finals ns)) new-states))))))
```



## Déterminisation d'un automate VIII

```
(define minimize
  (lambda (aut)
    (let ((states (states aut)) (alph (alph aut)) (trans (trans aut))
          (init (init aut)) (finals (finals aut)))
      (let ((graph
              (mk-graph states
                        (s-map (lambda (tr) (mk-arc (orig tr) (extr tr)))
                               trans))))
        (let ((m-states (offspring-ter init graph)))
          (let ((m-trans
                  (filter (lambda (tr) (in? (orig tr) m-states)) trans))
                 (m-finals
                  (filter (lambda (ns) (in? ns m-states)) finals)))
            (mk-aut m-states alph m-trans init m-finals)))))))
```



## Déterminisation d'un automate IX

```
(define *autom01*     ;; fig. 10.3, p. 218 (El. Prog)
  '((q0 q1 q2 q3 q4)
    (a b)
    ((q0 () q1) (q0 () q3) (q1 (a) q2) (q1 (a) q3)
     (q2 (b) q1) (q3 (a) q1) (q3 (a) q3) (q3 (b) q4)
     (q4 (a) q1) (q4 (a) q2) (q4 () q3) (q4 (b) q3))
    q0
    (q4)))

(define *det01* (determinize *autom01*))

(define *small01* (minimize *det01*))

(define *autom02*     ;; ex. 2.6, p. 29 (Calc)
  '((q0 q1 q2 q3)
    (a b)
    ((q0 (a) q0) (q0 (b) q0) (q0 (a) q1)
     (q1 (a) q1) (q1 (b) q2) (q2 (a) q3) (q3 (b) q2))
    q0
    (q2)))

(define *det02* (determinize *autom02*))

(define *small02* (minimize *det02*))
```



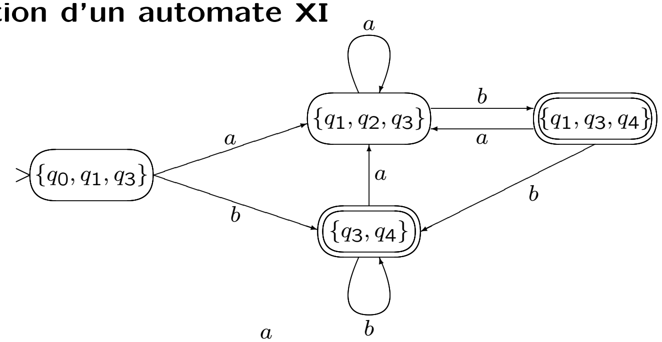
## Déterminisation d'un automate X

Si un automate non déterministe comporte $n$ états, et si l'alphabet contient $p$ symboles, l'automate déterministe correspondant comportera $2^n$ états et $p\,2^n$ transitions. Pour les deux exemples, on a respectivement 32 et 16 états, et 64 et 32 transitions. Les versions minimisées sont nettement plus petites, avec seulement 4 états et donc 8 transitions :

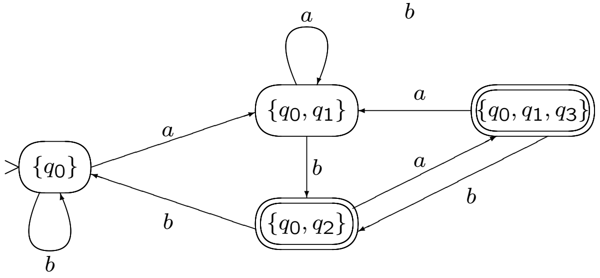
```
(automaton->list *small01*) ==>
(((q1 q3 q4) (q1 q2 q3) (q3 q4) (q0 q1 q3))
 (a b)
 (((q3 q4) a (q1 q2 q3)) ((q1 q3 q4) a (q1 q2 q3))
  ((q1 q2 q3) a (q1 q2 q3)) ((q0 q1 q3) a (q1 q2 q3))
  ((q3 q4) b (q3 q4)) ((q1 q3 q4) b (q3 q4))
  ((q1 q2 q3) b (q1 q3 q4)) ((q0 q1 q3) b (q3 q4)))
 (q0 q1 q3)
 ((q3 q4) (q1 q3 q4)))

(automaton->list *small02*) ==>
(((q0 q1 q3) (q0 q2) (q0 q1) (q0))
 (a b)
 (((q0) a (q0 q1)) ((q0 q2) a (q0 q1 q3)) ((q0 q1) a (q0 q1))
  ((q0 q1 q3) a (q0 q1)) ((q0) b (q0)) ((q0 q2) b (q0))
  ((q0 q1) b (q0 q2)) ((q0 q1 q3) b (q0 q2)))
 (q0)
 ((q0 q2)))
```

## Déterminisation d'un automate XI

small01




small02






# 12. Abstraction procédurale

*Principe.* La notion de procédure est la clef de la décomposition d'un problème en sous-problèmes. Le fait qu'en Scheme une procédure puisse accepter des procédures comme données et produire des procédures comme résultats rend le langage spécialement adapté à *l'abstraction procédurale*.

*Conséquence.* En programmation comme en mathématique, il est souvent opportun de reconnaître en un problème donné un cas particulier d'un problème plus général, et même de chercher d'emblée à résoudre le problème général, ce qui produira une procédure largement réutilisable dans des contextes variés.

*Application.* On va voir comment une procédure itérative de calcul de la racine carrée peut se généraliser en une procédure très générale de mise en œuvre d'un processus d'approximation.



## Méthode itérative de calcul de $\sqrt{x}$

$y_0 = 1 \qquad y_{n+1} = \frac{1}{2}\left(y_n + \frac{x}{y_n}\right)$

Si $x = 2$ : $1\,,\ \frac{1}{2}\left(1 + \frac{2}{1}\right) = 1.5\,,\ \frac{1}{2}\left(1.5 + \frac{2}{1.5}\right) = 1.4167\,,\ \ldots$

```
(define sqrt-iter
  (lambda (guess x)
    (display "  ") (write guess)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x) x))))

(define improve
  (lambda (guess x) (/ (+ guess (/ x guess)) 2)))

(define good-enough?
  (lambda (guess x) (< (abs (- (square guess) x)) 0.0000000001)))

(sqrt-iter 1.0 2.0)
1.  1.5  1.4166666666666665  1.4142156862745097  1.4142135623746899
;Value: 1.4142135623746899
```



## Abstraction et généralisation I

Généralisation élémentaire : passer de la racine carrée à la racine $p$ième.

Méthode itérative de calcul de $\sqrt[p]{x}$.

$y_0 = 1 \qquad y_{n+1} = \frac{1}{p}\left((p-1)y_n + \frac{x}{y_n^{p-1}}\right)$

```
(define sqrt-p-iter
  (lambda (guess x p) ...

(define improve
  (lambda (guess x p)
    (/ (+ (* (- p 1) guess)
          (/ x (expt guess (- p 1)))) p)))

(define good-enough?
  (lambda (guess x p)
    (< (abs (- (expt guess p) x)) 0.00000001)))

(sqrt-p-iter 1.0 729.0 3)    9.
(sqrt-p-iter 1.0 2.0 2)      1.4142135623746899
```

## Abstraction et généralisation II

Généralisation moins élémentaire :
passer de l'équation $y^p - x = 0$ à l'équation $f(y) = 0$.

$y_0 = 1 \qquad y_{n+1} = y_n - \frac{f(y_n)}{Df(y_n)}$

```
(define solve
  (lambda (guess f Df)
    (if (good-enough? guess f)
        guess
        (solve (improve guess f Df) f Df))))

(define improve
  (lambda (guess f Df) (- guess (/ (f guess) (Df guess)))))

(define good-enough?
  (lambda (guess f) (< (abs (f guess)) 0.1)))

(solve 1.0 (lambda (y) (- (expt y 3) 729.0))
           (lambda (y) (* 3 (expt y 2))))       9.0000220253469
```



## Abstraction et généralisation III

Résolution itérative de $f(y) = 0$ avec calcul approximatif de la dérivée

```
(define newton
  (lambda (gu f dx)
    (if (good-enough? gu f)
        gu
        (newton (improve gu f dx) f dx))))

(define deriv
  (lambda (f dx) (lambda (x) (/ (- (f (+ x dx)) (f x)) dx))))

(define improve
  (lambda (gu f dx) (- gu (/ (f gu) ((deriv f dx) gu)))))

(define good-enough?
  (lambda (gu f) (< (abs (f gu)) 0.1)))

(newton 1.0 (lambda (y) (- (expt y 3) 729.0)) 0.0001)  9.000022153425999
(newton 1.0 (lambda (y) (- y (cos y))) 0.0001)         0.7503675298583334
(cos 0.7503675298583334)                               0.731438296864949
```

(Ici, `good-enough?` ... ne mérite pas son nom.)



## Abstraction et généralisation IV

```
(define newton
  (lambda (gu f dx)
    (let* ((deriv
            (lambda (f dx)
              (lambda (x) (/ (- (f (+ x dx)) (f x)) dx))))
           (improve
            (lambda (gu f dx) (- gu (/ (f gu) ((deriv f dx) gu)))))
           (good-enough?
            (lambda (gu f) (< (abs (f gu)) 0.001))))
      (if (good-enough? gu f)
          gu
          (newton (improve gu f dx) f dx)))))

(newton 1.0 (lambda (y) (- y (cos y))) 0.0001)         0.7391131535431725
(cos 0.7391131535431725)                               0.7390662580950105
```

(`good-enough?` a été modifié.)



## Abstraction et généralisation V

Calcul itératif de point fixe : résoudre $x = f(x)$

```
(define fixpoint
  (lambda (gu f)
    (let ((good-enough?
           (lambda (gu f) (< (abs (- gu (f gu))) 0.001))))
      (if (good-enough? gu f) gu (fixpoint (f gu) f)))))

(fixpoint 1.0 cos)                      0.7395672022122561
(cos 0.7395672022122561)                0.7387603198742113
(fixpoint 1.0 (lambda (x) (/ 2 x)))     ...
```

Amélioration par lissage

```
(define fixpoint
  (lambda (gu f)
    (let ((good-enough?
           (lambda (gu f) (< (abs (- gu (f gu))) 0.001)))
          (improve
           (lambda (gu f) (/ (+ gu (f gu)) 2))))
      (if (good-enough? gu f) gu (fixpoint (improve gu f) f)))))

(fixpoint 1.0 (lambda (x) (/ 2 x)))     1.414...
```

## Abstraction et généralisation VI

Idée : `fixpoint` et `newton` sont deux instances de `iterative-improve`. `iterative-improve` prend comme arguments des procédures `good-enough?` et `improve` et renvoie comme valeur une procédure `f` telle que `(f gu)` soit `(if (good-enough? gu) gu (f (improve gu)))`

On doit donc

– définir `iterative-improve`
– écrire `fixpoint` et `newton` comme instances de `iterative-improve`

Ceci permettra des appels tels que

```
((fixpoint cos) 1.0)                              0.7392146118880453
((newton (lambda (x) (- x (cos x))) 0.001) 1.0)  0.7391155232281558
```



## Abstraction et généralisation VII

```
(define iterative-improve
  (lambda (good-enough? improve)
    (lambda (gu)
      (letrec
        ((f (lambda (g) (if (good-enough? g) g (f (improve g))))))
        (f gu)))))

(define fixpoint
  (lambda (f)
    (iterative-improve (lambda (gu) (< (abs (- gu (f gu))) 0.001))
                       (lambda (gu) (/ (+ gu (f gu)) 2)))))

(define newton
  (lambda (f dx)
    (let ((deriv
           (lambda (f) (lambda (x) (/ (- (f (+ x dx)) (f x)) dx)))))
      (iterative-improve
        (lambda (gu) (< (abs (- gu (f gu))) 0.001))
        (lambda (gu) (- gu (/ (f gu) ((deriv f) gu))))))))
```



## Itérateur I

On appelle $n$ième itérée de la fonction $f$ de $D$ dans $D$ la composée de $n$ fonctions égales à $f$.

> *Ecrire une fonction* `iter` *tel que pour tout naturel* `n`, `(iter n)` *soit la fonction qui à toute fonction* `f` *auto-composable associe la* `n`*ième itérée de* `f`.

La solution est immédiate, mais il faut veiller à respecter le type fonctionnel des objets manipulés.

```
(define iter                               ;; On définit iter
  (lambda (n)                              ;; une fonction à un argument
    (lambda (f)                            ;; (iter n) associe à f
      (if (zero? n)                        ;; si n vaut 0
          (lambda (x) x)                   ;; la fonction identité
          (compose f ((iter (- n 1)) f)))))) ;; sinon la composée de ...
```



## Itérateur II

```
(define iter      ;; variante
  (lambda (n)
    (lambda (f)
      (lambda (x)
        (if (zero? n) x (f (((iter (- n 1)) f) x)))))))

(define it        ;; scinder la difficulté
  (lambda (n f x)
    (if (= n 0) x (f (it (- n 1) f x)))))

(define iter-bis  ;; autre variante, équivalente
  (lambda (n)
    (lambda (f) (lambda (x) (it n f x)))))

(((iter     3) cos) 1)   .6542897904977791
(((iter-bis 3) cos) 1)   .6542897904977791
(cos (cos (cos 1)))      .6542897904977791
(it 3 cos 1)             .6542897904977791
```