# 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.

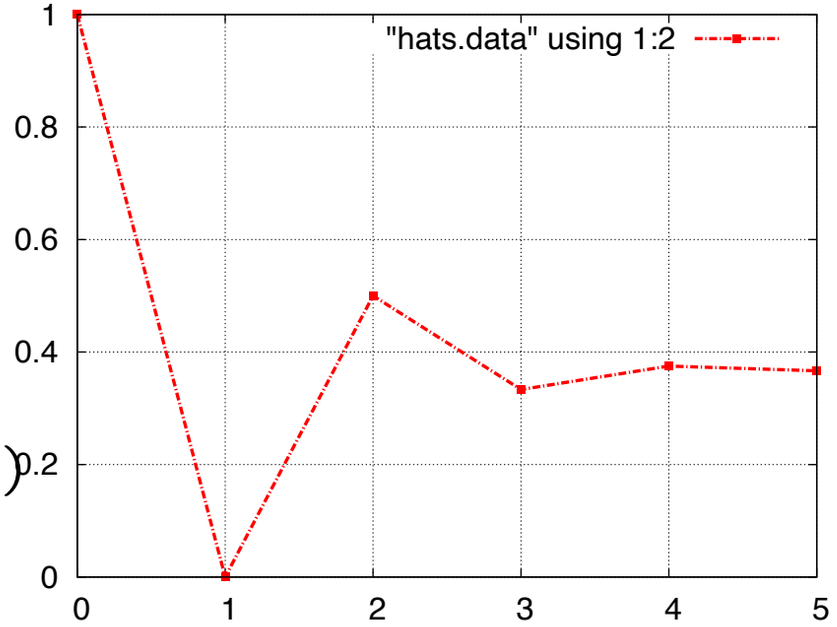
- Nombre $C(n,k)$ de choix de $k$ objets parmi $n$ $(0 \leq k \leq n)$ ?

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` |

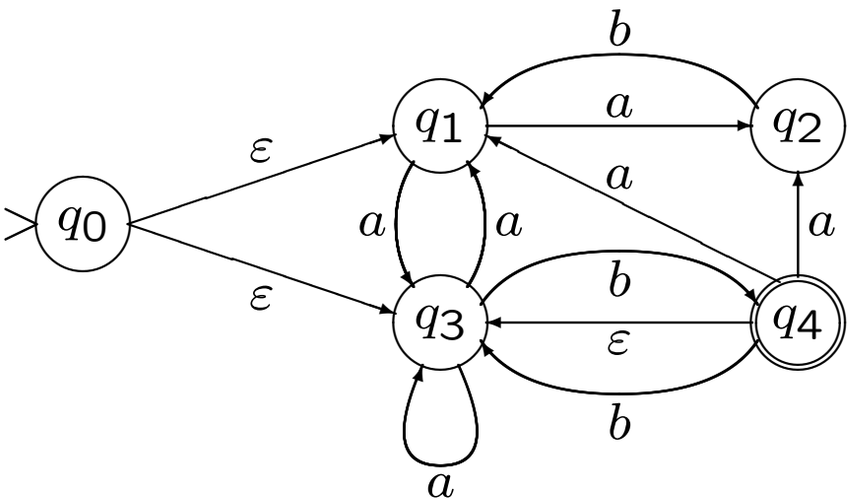
# Essais

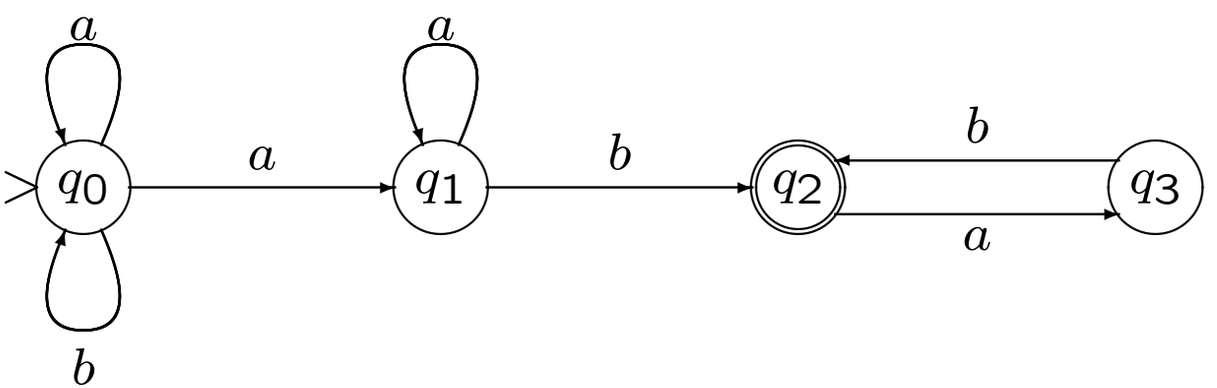
```
c       '((4 . 9) (3 . 8) (2 . 4) (1 . 1) (2 . 3)
          (6 . 9) (5 . 5) (4 . 8) (1 . 2) (2 . 1)
          (1 . 2) (1 . 1) (7 . 9) (6 . 6) (5 . 4)
          (4 . 5) (3 . 2) (2 . 3) (2 . 2) (3 . 3)))

(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*))
```

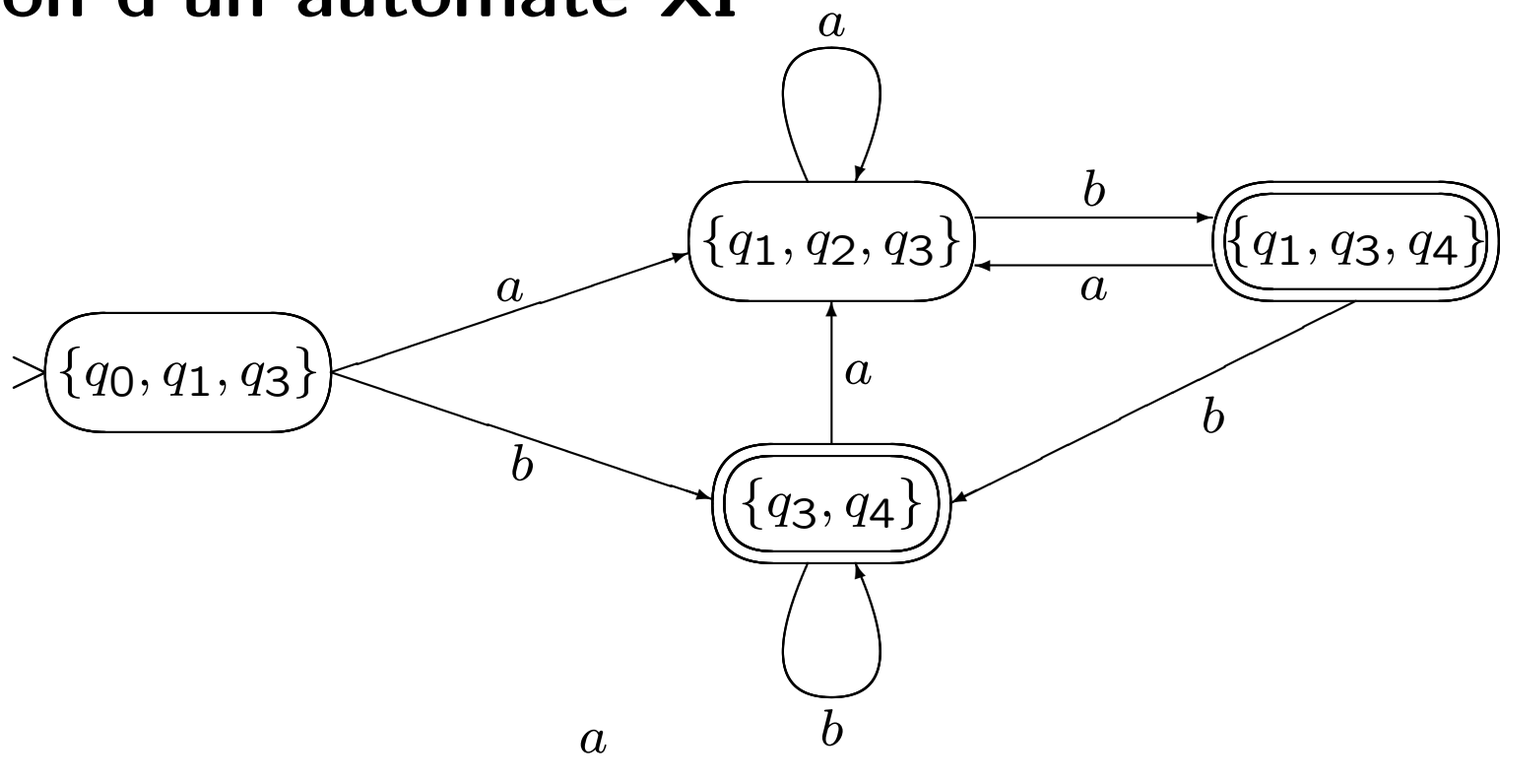
# Déterminisation d'un automate X

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