Algorithmes gloutons

*Nous abordons une grande classe d'algorithmes, les algorithmes gloutons (*greedy algorithmes*). Ils visent tous à trouver l'optimum global d'un problème d'optimisation par une suite d'étapes qui sont autant d'optimisations locales. Durant cette séance, nous allons aborder plusieurs grands "classiques", avant de finir par une version allégée d'un problème posé au concours Centrale-Supélec en 2002.*

1 Le problème du sac-à-dos

Il s'agit d'un problème extrêment célèbre en informatique, car la simplicité de son modèle le fait retrouver sous de nombreux visages.
D'abord, nous allons étudier une première version, qui est simple à résoudre.

Un voleur arrive à s'introduire dans une maison, et cherche, tout naturellement, à en repartir avec le plus d'objets de valeur possible. Malheureusement pour lui, il ne peut emporter dans son sac-à-dos que x kilos ; il va donc devoir sélectionner les objets qu'il va emporter.
L'optimisation globale qu'il cherche à réaliser sera donc le gain total qu'il emportera, alors que la suite d'optimisations locales sera constituée de sa décision en face de chaque objet, s'il l'emporte ou non.
Dans cette première version, nous allons supposer que tous les objets sont fractionnaires, i.e. il peut en prendre n'importe quelle quantité (comme si c'était de la poudre ou un liquide, par exemple). Il y a n produits différents, ayant chacun un prix au kilo $p.(i)$ ($0 \leq i \leq n-1$). Toutes les quantités sont limitées ; seuls $q.(i)$ kilos du produit i sont disponibles. Enfin, on supposera que tous les prix sont différents deux à deux et que p est trié par prix croissants.

▶ **Question 1** *Ecrivez en Caml un algorithme qui propose au cambrioleur les meilleurs choix, de façon à renvoyer la somme gagnée et la liste des choix à faire.*

```
voleur : int vect -> int vect -> int -> int * int vect
```

Maintenant, nous allons supposer que les objets ne sont plus fractionnaires. Chaque objet i sera représenté par un prix $p.(i)$ et une masse $m.(i)$.

▶ **Question 2** *Peut-on étendre l'algorithme précédent à ce nouveau cas ? Si oui, démontrez-le. Si non, exhibez un contre-exemple.*

2 Réservation SNCF

On suppose que n personnes veulent voyager en train un jour donné. La personne i veut prendre le train $p.(i)$, où p est un tableau d'entiers. Les trains sont numérotés de 0 à $k-1$ et partent dans l'ordre de leur numéro. Chaque train peut contenir au plus c personnes.

▶ **Question 3** *Écrivez une fonction* possible *qui teste si tout le monde peut prendre le train de son choix. Vous veillerez à ce que votre fonction ne modifie pas le tableau passé en argument.*

```
possible : int vect -> int -> int -> bool
```

On suppose maintenant que la personne i, si elle ne peut pas prendre le train $p.(i)$ parce qu'il est complet, accepte de prendre un des trains suivants (s'il y en a un).

▶ **Question 4** *Proposez et programmez un algorithme* sncf *pour affecter chaque personne à un train lorsque c'est possible.*

```
sncf : int vect -> int -> int -> int vect
```

3 Remplissage de boîtes

On dispose de n objets, de poids respectifs $p.(0), p.(1), ..., p.(n-1)$, où p est un tableau d'entiers strictement positifs. On supposera ce tableau trié dans l'ordre croissant. On désire remplir au maximum une boîte, sans toutefois dépasser une charge maximale x. De manière plus abstraite, on cherche en fait la plus grande valeur inférieure ou égale à x que l'on puisse obtenir en sommant les éléments d'un sous-ensemble de $\{p.(0), ..., p.(n-1)\}$ (on précise bien que c'est la valeur

en question que l'on cherche, et non pas le sous-ensemble correspondant).
On va concevoir un algorithme qui procède en n étapes : à l'étape i, on construira la liste de tous les poids inférieurs ou égaux à x que l'on peut obtenir en prenant des éléments dans $p.(0), ..., p.(i-1)$. On a besoin pour cela de quelques fonctions préliminaires.

▶ **Question 5** *Programmez une fonction* ajoute *qui prend pour argument un entier a ainsi qu'une liste d'entiers* $m = [m_0; ...; m_{k-1}]$ *et retourne la liste* $[m_0 + a; ...; m_{k-1} + a]$.

```
ajoute : int -> int list -> int list
```

▶ **Question 6** *Écrivez une fonction* retire *qui prend l'entier x et une liste m et retourne cette liste dans laquelle les éléments strictement supérieurs à x ont été supprimés.*

```
retire : 'a -> 'a list -> 'a list
```

▶ **Question 7** *Définissez enfin une fonction* list_max *qui calcule le maximum d'une liste.*

```
list_max : 'a list -> 'a
```

▶ **Question 8** *Déduisez-en une fonction* emboitement *qui résout le problème. Que pensez-vous de la complexité de votre algorithme ?*
Peut-on voir un lien avec le prermier problème (celui du cambrioleur) ?

```
emboitement : int vect -> int -> int
```

▶ **Question 9** *(bonus)*
Modifiez vos fonctions pour obtenir la solution exacte du problème du voleur.

```
ajoute2 : int -> int -> (int * int) list -> (int * int) list
retire2 : 'a -> ('a * 'b) list -> ('a * 'b) list
list_max2 : ('a * 'b) list -> 'b
emboitement2 : int vect -> int vect -> int -> int
```

4 Réservation d'un gymnase

On considère un gymnase dans lequel on souhaite organiser différentes épreuves. On souhaite en en "caser" le plus possible, sachant que deux épreuves ne peuvent pas avoir lieu en même temps, et qu'il n'y a qu'un seul gymnase disponible. Les événements, au nombre de n sont déterminés par leur heure de début d_i et leur heure de fin f_i. L'événement i recquiert donc le gymnase durant l'intervalle de temps $[d_i; f_i[$.

▶ **Question 10** *Indiquez comment modéliser la situation. Proposez un algorithme glouton optimal et écrivez une fonction qui résout le problème. On pourra supposer que le tableau est trié comme il faut.*
indication : 3 *idées sont possibles : on sélectionne en fonction de la durée des événements (en sélectionnant d'abord les plus courts), de la date de début (en prenant d'abord ceux qui commencent le plus tôt), de la date de fin (en prenant ceux qui finissent le plus tôt).*

```
gymnase : int vect -> int vect -> int list
```

5 Extraits du sujet Informatique posé à Centrale-Supélec en 2002

Note : le sujet est disponible dans son intégralité à l'adresse suivante : http ://centrale-supelec.sceiconcours.org/CentraleSupelec/2002/MP/sujets/info.pdf
Le sujet traite du problème du monnayeur : comment rendre la monnaie en utilisant le plus petit nombre possible de pièces ?

5.1 Formalisation du problème

Soit $c = (c_i)_{1 \leq i \leq m}$ un m-uplet d'entiers vérifiant $c_1 > c_2 > \ldots > c_m = 1$, qu'on appelle système. Les c_i sont les valeurs faciales des pièces utilisées, par exemple, le système utilisé en zone Euro est $(200, 100, 50, 20, 10, 5, 2, 1)$. On suppose qu'on dispose d'une quantité illimité de chaque pièce. Soit x un entier, correspondant à la monnaie à rendre. On peut alors représenter x dans le système c par un m-uplet $k = (k_i)_{1 \leq i \leq m}$ tel que : $x = \sum_{i=1}^{m} k_i c_i$

5.2 Systèmes de pièces

▶ **Question 11** *Rédiger en Caml une fonction* est_un_systeme *qui indique si une liste* c *est bien un système de pièces.*

```
est_un_systeme : int list -> bool
```

5.3 Représentations de poids minimal

Soit $c = (c_i)_{1 \leq i \leq m}$ un système et x un entier. Nous notons $M_c(x)$ (ou $M(x)$ s'il n'y a pas d'ambiguïté) le plus petit nombre de pièces nécessaires pour représenter x dans le système c : $M(x) = min\{\|k\| \text{ s.t. } k \in \mathbb{N}^m et (kc = x)\}$ (on a $\|k\| = \sum_{i=1}^{m} k_i$). Une représentation de poids minimale (ou représentation minimale) est une réprésentation qui vérifie $\|k\| = M(x)$.

▶ **Question 12** *Prouvez l'encadrement*

$$\left\lceil \frac{x}{c_1} \right\rceil \leq M(x) \leq x$$

▶ **Question 13** *Exhiber un système c et un entier x possédant plusieurs représentations dans c.* Soit $x > 1$, et s le plus petit indice i tel que $c_i \leq x$. On supposera démontrée l'égalité suivante :

$$M(x) = 1 + min_{s \leq i \leq m} M(x - c_i)$$

▶ **Question 14** *Rédigez une fonction* poids_minimaux x c *qui construit la liste des valeurs* $M_c(y)$ *pour* $1 \leq y \leq x$ *(dans l'ordre des y croissants).*
On pourra utiliser les fonctions list_of_vect , vect_of_list , make_vect.

```
poids_minimaux : int -> int list -> int list
```

6 L'algorithme glouton

Note : dans cette section, on travaillera obligatoirement sur des listes sans passer par des vecteurs.

▶ **Question 15** *Ecrivez en Caml un algorithme glouton* glouton *qui rend la monnaie en rendant d'abord le plus grand* c_i *possible, puis en rendant récursivement la monnaie sur* $x - c_i$.

```
glouton : int -> int list -> int list
```

Algorithmes gloutons

Un corrigé

▶ **Question 1** Quand les différentes matières sont fractionnaires, il suffit de sélectionner d'abord la matière la plus chère et d'en prendre le plus possible, avant de passer à la suivante, jusqu'à remplir complètement le sac-à-dos.

▶ Question 1

```
let voleur = fun p q x ->
let n = vect_length p and
place_restante = ref x and
magot = ref 0 in
let choix = make_vect n 0 in
for i = (n-1) downto 0 do (
choix.(i) <- min !place_restante q.(i);
place_restante := !place_restante - choix.(i);
magot := !magot + choix.(i) * p.(i);
)
done;
(!magot, choix);;
```

▶ **Question 2** Malgré sa simplicité et les nombreuses études menées, aucun algorithme avec une complexité en temps polynomiale en n n'a été trouvé pour résoudre ce problème de façon exacte. Si on cherche à sélectionner d'abord les objets qui ont le meilleur rapport $prix/masse$, un contre-exemple simple consiste à prendre $x = 10$, $m = (10, 1)$, $p = (5, 2)$. L'algorithme sélectionne le second objet et du coup ne peut pas sélectionner le premier. Le gain est alors de 2 alors qu'il aurait pu être de 5.
Si on sélectionne d'abord les objets qui ont la plus petite masse, le même contre-exemple fonctionne. Si on sélectionne d'abord les objets qui ont la plus grande valeur, un contre-exemple serait $x = 10$, $m = (10, 1, 1)$ et $p = (5, 4, 4)$.

▶ Question 3

```
let possible = fun p k c ->
let n = vect_length p and
trains = make_vect k 0 and
resultat = ref true in
for i = 0 to (n-1) do (
trains.(p.(i)) <- trains.(p.(i)) + 1;
resultat := !resultat && (trains.(p.(i)) <= c);
)
done;
!resultat
;;
possible [|0; 0; 0; 1|] 2 2;;
```

▶ Question 4

```
let sncf = fun p k c ->
let n = vect_length p and
occupation = make_vect k 0 and
t = ref 0 in
let affectation = make_vect n (-1) in
for i = 0 to (n-1) do (
t := p.(i);
while ((!t < k) && (occupation.(!t) >= c)) do (
t := !t + 1;
)
done;
if(!t < k) then (
affectation.(i) <- !t;
occupation.(!t) <- occupation.(!t) + 1;
)
)
done;
affectation;;
sncf [|0; 0; 0; 1|] 2 2;;
(* on met -1 comme numéro de train si on n'arrive pas à placer un voyageur*)
```

▶ Question 5

```
let rec ajoute = fun a m ->
match m with
| [] -> []
| t::q -> (t+a)::(ajoute a q)
;;
```

▶ Question 6

```
let rec retire = fun x m ->
match m with
| [] -> []
| t::q when t <= x -> t::(retire x q)
| t::q -> retire x q
;;
```

▶ Question 7

```
let rec list_max = fun list ->
match list with
| [] -> failwith "liste vide"
| t::[] -> t
| t::q -> let m = list_max q in max t m
;;
list_max [1; 4; 8; 3; 2; 1];;
```

▶ Question 8

```
let emboitement = fun p x −>
let n = vect_length p and
list = ref [0] in
for i = 0 to (n−1) do (
list := (retire x (ajoute (p.(i)) !list))@(!list);
)
done;
list_max !list
;;
```

▶ **Question 8** Comme dans le pire des cas la taille de la liste double à chaque itération, l'algorithme a une complexité exponentielle. Il n'est donc pas utilisable en pratique.

▶ Question 9

```
let rec ajoute2 = fun a b m −>
match m with
| [] −> []
| t::q −> ((fst t) +a, (snd t) + b)::(ajoute2 a b q)
;;
let rec retire2 = fun x m −>
match m with
| [] −> []
| t::q when (fst t) <= x −> t::(retire2 x q)
| t::q −> retire2 x q
;;

let rec list_max2 = fun list −>
match list with
| [] −> failwith "liste vide"
| t::[] −> snd t
| t::q −> let m = list_max2 q in max (snd t) m
;;
let emboitement2 = fun m p x −>
let n = vect_length p and
list = ref [(0, 0)] in
for i = 0 to (n−1) do (
list := (retire2 x (ajoute2 (m.(i)) (p.(i)) !list))@(!list);
)
done;
list_max !list
;;
emboitement2 [|1; 4; 8; 3; 2; 1|] [|1; 1; 1; 1; 1; 1|] 8;;
```

▶ Question 10

```
let gymnase = fun d f −>
let n = vect_length d in
let rec aux = fun heure i −>
if(i >= n) then []
else (
if d.(i) < heure then (aux heure (i+1))
else (i::(aux f.(i) (i+1)))
)
in
aux 0 0
;;
```

▶ Question 11

```
let rec est_un_systeme = fun m −>
let rec aux = fun m p −>
match m with
| [] −> false
| t::[] −> (t = 1)&&(t < p)
| t::q −> (t < p)&&(aux q t)
in
match m with
| [] −> false
| t::[] −> t = 1
| t::q −> aux q t
;;
est_un_systeme [7;5;2];;
```

▶ **Question 12** Une représentation possible de x est $k = (0, \cdots, 0, x)$, et on a $M(x) \leq \|k\| = x$.
Soit k une représentation de x. On a $kc = x$ et $kc \leq c_1\|k\|$, d'où le résultat (en se souvenant que $\|k\|$ est un entier).

▶ **Question 13** On peut prendre $c = (5, 2, 1)$, $x = 15$ ainsi que $k = (3, 0, 0)$ et $k' = (2, 2, 1)$.

▶ Question 14

```
let minimum = fun y s c pm −> (∗ pm correspond au tableau des M ∗)
let m = vect_length c in
let resultat = ref (pm.(y − c.(m−1))) in
for i = s to m do
resultat := min (!resultat) pm.(y − c.(i−1));
done;
!resultat;;
let poids_minimaux = fun x c −>
let pm = make_vect (x+1) 0 and
cc = vect_of_list c in
let m = vect_length cc in
let s = ref m in (∗ on sait que c(m) = 1 par definition ∗)
pm.(1) <− 1;
for y = 2 to x do (
(∗ on regarde si on peut décrémenter s ∗)
(∗ on a cc.(!s−1) == c(s) ∗)
if(!s > 1) && (cc.(!s − 2) <= y) then s := !s − 1;
(∗ on applique la formule donnée ∗)
pm.(y) <− 1 + (minimum y !s cc pm);
)
done;
list_of_vect (sub_vect pm 1 x);;
poids_minimaux 5 [5; 2; 1];;
```

▶ Question 15

```
let rec glouton = fun x c −>
match c with
| [] when x > 0 −> failwith "y a une erreur !"
| [] −> []
| t::q −> let a = x / t in (a::(glouton (x − (t ∗ a)) q))
;;
glouton 13 [5; 2; 1];;
```