Systèmes monétaires (d'après le concours Polytechnique)

1 Introduction

1. Il s'agit de sommer les éléments de la liste, par exemple avec une fonction récursive :

```
let rec valeur = function
  | [] -> 0
  | t::q -> t + valeur q
```

On pourrait aussi, éventuellement, utiliser un `List.fold_left` :

```
let valeur =
  List.fold_left (+) 0
```

2. On utilise le fait que listes soient triées pour une solution de complexité linéaire :

```
let rec est_extraite s1 s2 = match s1, s2 with
  | t1::q1, t2::q2 when t1=t2 -> est_extraite q1 q2
  | t1::q1, t2::q2 when t1<t2 -> est_extraite s1 q2
  | t1::q1, t2::q2 -> false
  | [], _ -> true
  | _, [] -> false
```

Notons que les trois derniers cas peuvent être regroupés en un seul cas :

```
let rec est_extraite s1 s2 = match s1, s2 with
  | t1::q1, t2::q2 when t1=t2 -> est_extraite q1 q2
  | t1::q1, t2::q2 when t1<t2 -> est_extraite s1 q2
  | _ -> s1 = []
```

2 Payer le compte exact

3. On peut montrer l'affirmation « pour tout $p \geqslant 0$, la stratégie gloutonne réussit » par exemple avec une recurrence forte sur p :

- c'est vrai pour $p = 0$;
- pour $p > 0$, supposons que c'est vrai pour tout p' vérifiant $0 \leqslant p' < p$ et considérons le cas de p : l'algorithme glouton peut toujours trouver une dénomination d inférieure ou égale à p (puisque le système contient une dénomination égale à 1), il se poursuit alors avec le prix $p - d$ vérifiant $0 \leqslant p - d < p$, qui réussit d'après l'hypothèse de récurrence.

Donc l'algorithme glouton réussit bien pour tout $p \in \mathbb{N}$.

On peut aussi remarquer que la somme restant à payer, dans l'algorithme, est une suite d'entiers naturels (p_n) strictement décroissante. Cette suite ne peut être infinie, donc l'algorithme va s'arrêter soit parce qu'il est arrivé à 0, soit parce qu'il n'a pas trouvé de dénomination suffisamment petite. Mais comme le système contient une dénomination égale à 1, ce second cas est impossible, donc l'algorithme parvient à payer le prix p.

4. Encore une fonction récursive, où si le prix p est plus petit que la plus grande des espèces du système monétaire `t`, on prend cette espèce et on continue à payer `p-t` avec le même système, et sinon on paye p avec les espèces restantes du système monétaire. On a utilisé ici à nouveau le fait que les dénominations dans le système sont ordonnées par ordre décroissant. La liste contenant les dénominations du système que l'on peut encore utiliser contient toujours au moins la dénomination 1, donc le dernier cas n'est présent que pour éviter un avertissement lors de la compilation.

```
let rec glouton p = function
  | _ when p=0 -> []
  | t::q when t>p -> glouton p q      (* t ne pourra plus servir *)
  | t::q -> t::glouton (p-t) (t::q)
  | [] -> failwith "Situation impossible";;
```

5. Payer 6 avec le portefeuille $[5;2;2;2]$ est possible, mais la stratégie gloutonne échouera. Il existe une infinité de contre-exemples (tel que 42 avec $[20;10;5;2;2;2;2;2;2]$), mais on s'efforcera dans ce genre de situation d'éviter un contre-exemple inutilement compliqué.

6. Il existe de nombreuses implémentations possibles. On prendra surtout garde à ce que, si l'on décide, pour payer une somme p, de choisir une dénomination t et de faire un appel récursif pour payer $t - p$ avec le reste du portefeuille, l'appel récursif peut échouer à trouver une solution et retourner une liste vide. On ne veut alors pas retourner la liste `[ t ]` ! Toutefois, une liste vide peut également ne pas être un échec mais simplement une façon de payer une somme nulle (si $p - t = 0$ par exemple).

Une solution possible est :

```
let paye_glouton p = function
  | [] -> []
  | h::t when h=p -> [h]
  | h::t when h>p -> paye_glouton p t
  | h::t -> match paye_glouton (p-h) q with (* p-h>0 *)
              | [] -> []
              | lst -> h::lst
```

On peut aussi utiliser utiliser la démarche gloutonne (sans réutilisation d'une espèce) et

contrôler que le résultat est bien celui recherché :

```
let paye_glouton p pf =
  let rec aux_glouton p = function
    | h::t when h>p -> glouton p t
    | h::t -> h::glouton (p-h) t
    | _ -> []
  in let sol = aux_glouton p pf
     in if valeur sol = p then sol else []
```

7. Pour toutes les espèces du portefeuille, on envisage de les utiliser ou non pour payer le prix p. Pour simplifier les choses, on peut utiliser une fonction auxiliaire récursive qui prend en entrée les deux mêmes arguments et qui renvoie un couple composé d'un booléen, indiquant si le prix est payable avec le portefeuille, et une liste qui sera la solution si elle existe, et une liste quelconque sinon. Cela peut s'écrire par exemple de la sorte :

```
let paye p pf =
  let rec tente_payer p pf = match p, pf with
    | 0, _ -> true, []      (* on a réussi à payer            *)
    | _, [] -> false, []    (* plus d'espèces, c'est un échec *)
    | _, h::t when h>p -> tente_payer p t (* h ne peut servir *)
    | _, h::t ->
      let b, lst = tente_payer (p-h) t in (* on tente avec h  *)
                   if b then (true, h::lst)
                        else tente_payer p t (* puis sans h
                                               si ça échoue  *)
  in let b, sol = tente_payer p pf
     in if b then sol else [];;
```

8. Dans le pire des cas, la solution envisagera 2^n façons de payer (chaque espèce pouvant ou non être prise), n étant le nombre d'espèces dans le portefeuille, en effectuant autant d'appels à `tente_payer`. Toutes les opérations effectuées sont de coût unitaire $O(1)$, la complexité de la fonction est donc $O(2^n)$.

3 Payer le compte exact et optimal

9. Il s'agit d'une fonction similaire à l'insertion d'un élément dans une liste triée, utilisée dans le tri par insertion. Cela s'écrit (voir cours) :

```
let rec ajout d = function
  | h::t when h>d -> h::ajout d t
  | lst -> d::lst;;
```

10. Ici encore, on utilise le fait que les listes sont triées !

```
let rec diff s sp = match s, sp with
  | [], _ -> []
  | _, [] -> s    (* Ce cas peut être omis *)
  | h::_, hp::tp when hp > h -> diff s tp
  | h::t, hp::tp when hp = h -> diff t tp
  | h::t, _ -> h::diff q tp;;
```

On notera que le second cas peut être librement omis.

11. On a :
- $\mathcal{T}(0) = [0]$,
- $\mathcal{T}(1) = [50;20;10;5;2]$ (les espèces de $\mathcal{P}f$)
- $\mathcal{T}(2) = [70;60;55;52;40;30;25;22;15;12;7]$ (les sommes de deux espèces *distinctes* de $\mathcal{P}f$ qui ne figurent pas dans $\mathcal{T}(1)$).

12. Il est bien précisé « *que l'on détaillera avec soin* » ! Dès qu'une fonction est délicate, les explications s'impose, mais ici la fonction est délicate *et* le sujet demande d'être très précis.

La fonction `suivant` prend en dernier argument une liste $\mathcal{T}_i = \{p_0, p_1, \ldots\}$ de prix pour lesquels on vient de trouver une solution optimale. On va donc prendre un à un chacun de ces prix (les p_j), récupérer dans la case `tab_m.(pj)` une solution optimale pour payer p_j, faire la différence entre le portefeuille et cette solution optimale pour déterminer les espèces restantes (notons-les d_k), et essayer d'ajouter chacune de ces espèces restantes à p_k.

Si la case d'index $p_j + d_k$ du tableau `tab_m` contient une liste vide (on veillera au passage à ne pas déborder du tableau), alors on vient de trouver une nouvelle solution optimale. Dans ce cas, on ajoute d_k à la solution optimale de p_j, on la place dans la case d'index $p_j + d_k$ du tableau `tab_m`, et on ajoute $p_j + d_k$ à l'ensemble $\mathcal{T}_{i+1}$.

Pour faciliter la construction de $\mathcal{T}_{i+1}$, on va utiliser une référence `t_ip1`, initialisée avec une liste vide. On va appeler la fonction `aux` sur toutes les valeurs p_j de $\mathcal{T}_i$. Pour chacune, on utilise `diff` pour déterminer les espèces d_k restantes, et pour chaque espèce d_k, on effectue le travail décrit précédemment.

L'écriture de la fonction proprement dite est délicate, car les itérations sur les listes ne sont pas particulièrement naturelles quand on est habitué aux langages impératifs. On peut utiliser `List.iter`, en mettant en premier argument une fonction qui sera appelée pour tous les éléments de la liste, placée en second argument.

Cela donne par exemple :

```
let suivant tab_m pf t_i =
  let t_ip1 = ref [] in       (* Contiendra T_{i+1} *)
  let aux pj =     (* Traitement d'une valeur p_j *)
      List.iter    (* Pour chaque espece d_k...   *)
        (fun dk -> if pj+dk < Array.length tab_m && tab_m.(pj+dk) = []
                   then begin
                     t_ip1:= (pj+dk) :: !t_ip1;
                     tab_m.(pj+dk) <- ajout dk tab_m.(pj)
                   end)
        (diff pf tab_m.(pj))  (* Calcul des espèces d_k restantes     *)
  in List.iter aux t_i;       (* Appel de aux avec chaque p_j dans T_i *)
     !t_ip1;;                 (* On retourne la liste construite      *)
```

On aurait évidemment pu donner un nom à « pj+dk », on a choisi sciemment ici de laisser la somme pour faciliter la lecture de la fonction.

4 Systèmes monétaires canoniques

13. Le contre-exemple le plus simple est probablement le prix $p = 48$, qui peut être payé avec $24+24$, alors que le glouton proposerait $30+12+6$.

14. Il s'agit desommer les cases du tableau, soit par exemple dans un style impératif :

```
let t_taille somme =
  let taille = ref 0
  and n = Array.length somme in
  for i = 0 to n - 1 do
    taille := !taille + somme.(i)
  done;
  !taille
```

Ou bien plus succintement :

```
let t_taille =
  Array.fold_left (+) 0;;
```

15. Là encore on applique la formule fournie :

```
let t_valeur systeme somme =
  let prix = ref 0
  and n = Array.length somme in
    for i = 0 to n-1 do
      prix := !prix + systeme.(i) * somme.(i)
    done;
    !prix;;
```

16. Notons s_k le nombre d'espèces de dénomination d_k dans $\mathcal{S}$ et $s'k$ celui dans $\mathcal{S}'$. Dans $\mathcal{S} \oplus \mathcal{S}'$, on a $s_k + s'_k$ espèces de dénomination d_k.

Ainsi, pour tout k, $s_k \leqslant s_k + s'_k$, donc $\mathcal{S} \leqslant_\ell \mathcal{S} + \mathcal{S}'$.

Note : puisque $\mathcal{S}'$ est non nul, on peut d'ailleurs affirmer qu'il existe un ou plusieurs k vérifiant $s_k < s_k + s'_k$. En prenant pour i le plus petit d'entre eux, on obtient $\mathcal{S} <_\ell \mathcal{S} + \mathcal{S}'$.

17. Rien de bien compliqué ici. On emploie un style impératif, puisque l'on travaille avec des tableaux. On crée un tableau de taille égale au nombre de dénominations, puis on remplit chaque case avec le nombre d'espèces nécessaires. Attention à bien utiliser une division entière pour obtenir la complexité demandée !

```
let t_glouton prix systeme =
  let n = Array.length systeme in
  let res = Array.make n 0
  and reste = ref prix in
    for i = 0 to n-1 do
      res.(i) <- !reste / systeme.(i);
      reste := !reste mod systeme.(i)
    done;
    res;;
```

18. Soient p et q tels que $p < q$. Soit $\mathcal{S}$ une somme quelconque de valeur $q - p > 0$.

On a $\mathrm{G}(p) <_\ell \mathrm{G}(p) + \mathcal{S}$

Or $\mathrm{G}(p) + \mathcal{S}$ a pour valeur $p + (q - p) = q$.

Par définition du glouton, on a donc $\mathrm{G}(p) + \mathcal{S} \leqslant_\ell \mathrm{G}(q)$.

Donc par transitivité $\mathrm{G}(p) <_\ell \mathrm{G}(q)$.

19. Si $\mathrm{S} = \mathrm{S}'$, alors $\mathrm{S} + \mathcal{I}_k = \mathrm{S}' + \mathcal{I}_k$, et donc $\mathrm{S} + \mathcal{I}_k <_\ell \mathrm{S}' + \mathcal{I}_k$

Sinon, il existe i tel que $\mathrm{S}_i < \mathrm{S}'_i$ et pour tout $0 \leqslant j < i$, $\mathrm{S}_j = \mathrm{S}'_j$. Trois cas sont à considérer :

- si $k > i$, alors $(\mathrm{S} + \mathrm{I}_k)_i < (\mathrm{S}' + \mathrm{I}_k)_i$ et pour tout $0 \leqslant j < i$, $(\mathrm{S} + \mathrm{I}_k)_j = (\mathrm{S}' + \mathrm{I}_k)_j$, donc $\mathrm{S} + \mathcal{I}_k <_\ell \mathrm{S}' + \mathcal{I}_k$;

- si $k = i$, alors $(S + d_k)_k < (S' + d_k)_k$ et pour tout $0 \leqslant j < k$, $(S + I_k)_j = (S' + I_k)_j$, donc $S + \mathcal{I}_k <_\ell S' + \mathcal{I}_k$;
- si $k \leqslant i$, alors $(S + I_k)_k < (S' + I_k)_k$ et pour tout $0 \leqslant j < k$, $(S + I_k)_j = (S' + I_k)_j$, donc $S + \mathcal{I}_k <_\ell S' + \mathcal{I}_k$.

20. $G(p) - I_k$ est bien un représentant de $p - d_k$.

Supposons par l'absurde qu'il ne s'agit pas de $G(p - d_k)$. On a alors, par définition du représentant glouton, $G(p) - \mathcal{I}_k <_\ell G(p - d_k)$.

Mais on a alors $G(p) - \mathcal{I}_k + \mathcal{I}_k <_\ell G(p - d_k) + \mathcal{I}_k$.

Soit $G(p) <_\ell G(p - d_k) + \mathcal{I}_k$, ce qui contrevient à la définition du glouton ($G(p)$ maximal parmi les représentants de p).

21. Supposons que $M(p - d_k) \neq M(p) \ominus \mathcal{I}_k$.

On alors par, définition de M, $M(p) \ominus \mathcal{I}_k \prec M(p - d_k)$. Deux cas sont possibles d'après la définition de $\prec$:

- $|M(p) \ominus \mathcal{I}_k| > |M(p - d_k)|$, mais alors $|M(p)| > |M(p - d_k) \oplus \mathcal{I}_k|$, donc $M(p) \prec M(p - d_k) \oplus \mathcal{I}_k$, ce qui est absurde ;
- $M(p) \ominus \mathcal{I}_k <_\ell M(p - d_k)$, mais alors $M(p) <_\ell M(p - d_k) \oplus \mathcal{I}_k$, donc $M(p) \prec M(p - d_k) \oplus \mathcal{I}_k$, ce qui est absurde également.

Donc si $M(p)$ contient une dénomination d_k, on a $M(p - d_k) = M(p) \ominus \mathcal{I}_k$.

22. Raisonnons par l'absurde en supposant qu'il existe un indice k tel que $m_k > 0$ et $g_k > 0$, et considérons la somme $w - d_k$.

On a $M(w - d_k) = G(w - d_k)$ par minimalité de w.

Mais si $m_k(w) \neq 0$, $M(w - d_k) = M(w) - \mathcal{I}_k$ (question 21).

De même, si $g_k(w) \neq 0$, $G(w - d_k) = G(w) - \mathcal{I}_k$ (question 20).

Par conséquent, $M(w) - \mathcal{I}_k = M(w - d_k) = G(w - d_k) = G(w) - \mathcal{I}_k$.

Et donc $M(w) = G(w)$, ce qui contrevient à la définition de w.

23. D'après la question précédente, si $m_0 \neq 0$, on a $g_0 = 0$, et donc $G(w) <_\ell M(w)$, ce qui est incompatible avec $G(w)$ maximum pour l'ordre $<_\ell$.

24. On a $M(w) <_\ell G(w)$ (par définition de w et du glouton).

Puisque $m_i \neq 0$ et $g_i = 0$, on a donc nécessairement $k < i$ tel que $g_k \neq 0$. $G(w)$ contient donc au moins une espèce de dénomination d_k, avec $d_k \geqslant d_{i-1}$ (puisque $k \leqslant i - 1$). On a donc $d_{i-1} \leqslant w$.

Il n'est pas possible d'avoir $w = d_{i-1}$ car sinon $G(w)$ ne contiendrait qu'une seule espèce, et on aurait nécessairement $G(w) = M(w)$. Donc $d_{i-1} < w$.

25. On a $M(w - d_j) = M(w) - I_j$ d'après la question 21.

Mais puisque w est minimal, $M(w - d_j) = G(w - d_j)$, donc $w - d_j < d_{i-1}$ (puisque la méthode gloutonne n'a pas pu sélectionner d'espèce de dénomination d_{i-1}), ce qui permet d'écrire $w < d_{i-1} + d_j$.

26. On déduit de l'encadrement fourni

$$G(d_{i-1} - 1) <_\ell M(w) \leqslant_\ell G(d_{i-1} - 1) + I_j$$

On a donc

$$\begin{cases} M_k(w) = G_k(d_{i-1} - 1) & \text{si} \quad k < j \\ M_j(w) = G_j(d_{i-1} - 1) + 1 \\ M_k(w) = 0 & \text{si} \quad k > j \end{cases}$$

27. Pour un i donné entre 1 et $n - 1$ (n étant le nombre de dénominations du système), on va déterminer $G(d_{i-1} - 1)$, noté g ci-dessous. Et, pour tout $j \geqslant i$, on va construire le potentiel contre-exemple $M(w)$ (appelé m dans la fonction qui suit) correspondant à ces valeurs de i et j grâce à la question précédente. Une fois $M(w)$ déterminé, on en déduit aisément w avec t_valeur, et on détermine $G(w)$ (noté g_w ci-après). Il ne reste alors qu'à vérifier si $G(w)$ a une longueur strictement plus grande que $M(w)$, ce qui signifierait que le système n'est *pas* canonique. Cela donne par exemple :

```
let est_canonique_i systeme i =
  let g = t_glouton (systeme.(i-1)-1) systeme
  and n = Array.length systeme in
  let rec aux j =
    if j = i-1 then true else
      (* On construit M(w) à partir de G *)
      let m = Array.make n 0 in
        for k = 0 to j-1 do
          m.(k) <- g.(k)
        done;
        m.(j) <- g.(j)+1;
        (* On en déduit le w correspondant *)
        let w = t_valeur systeme m in
        (* Et on regarde si le glouton est optimal *)
          let g_w = t_glouton w systeme in
            t_taille g_w <= t_taille m
            && aux (j-1)
  in aux (n-1);;
```

On termine en écrivant la fonction demandée, qui prend en argument un système monétaire et appelle la fonction précédente pour toutes les valeurs de i de $n - 1$ à 1 inclus, et retourne `true` si tous les appels retournent `true`, et `false` sinon :

```
let est_canonique systeme =
  let rec aux = function
    | 0 -> true
    | i -> est_canonique_i systeme i && aux (i-1)
  in aux (Array.length systeme - 1);;
```

28. Les boucles sur i et j vont effectuer, dans le pire des cas, $\Theta(n)$ constructions d'une solution gloutonne, $\Theta\left(n^2\right)$ constructions de $\mathrm{M}(w)$, $\Theta\left(n^2\right)$ déterminations de w, et $\Theta\left(n^2\right)$ calculs d'une solution gloutonne. L'algorithme a une complexité cubique $O\left(n^3\right)$.

Notons au passage que n est petit (moins d'une dizaine de dénominations en général), donc limiter à $\Theta\left(n^2\right)$ le nombre de possible contre-exemples à tester rend les choses extrêmement efficaces.