Polytechnique 2002. Option informatique. Étude des systèmes monétaires.

Corrigé pour serveur UPS de JL. Lamard (jean-louis.lamard@prepas.org)

Question 1.

Version itérative :

```
let valeur s = let v_prov = ref 0 and reste_s = ref s in
  while !reste_s <> [] do
    v_prov := !v_prov + hd !reste_s;
    reste_s := tl !reste_s
  done;
  !v_prov
;;
valeur : int list -> int = <fun>
```

Version récursive naïve non terminale :

```
let rec valeur s = match s with
  | [] -> 0
  | a :: suite -> a + valeur suite
;;
```

Version récursive terminale :

```
let rec valeur_aux valeur_prov reste = match reste with
  | [] -> valeur_prov
  | a :: suite -> valeur_aux (valeur_prov + a) suite
;;
let valeur = valeur_aux 0;;
```

Question 2.

L'algorithme glouton participe d'une boucle while : "tant que p est non nul".
À l'entrée d'une boucle (p est non nul), l'ensemble des espèces inférieures ou égales à p est non vide puisque l'une des espèces est égale à 1 et est fini donc admet un plus grand élément d. À la sortie, la nouvelle valeur de p : $p - d$ est strictement inférieure à la valeur d'entrée. Ainsi la valeur de p diminue strictement à chaque étape et donc la valeur de p finit par atteindre 0 en un nombre fini d'étapes. □

Version itérative :

```
let glouton sys p =
  let reste_p = ref p and reste_sys = ref sys and s_prov = ref [] in
    while (!reste_p > 0) do
      while hd(!reste_sys) > !reste_p do reste_sys := tl(!reste_sys) done;
      s_prov := hd(!reste_sys)::(!s_prov);
      reste_p := !reste_p - hd(!reste_sys)
    done;
    rev !s_prov
;;
glouton : int list -> int -> int list = <fun>
```

Version récursive :

```
let rec glouton sys p = match (sys,p) with
  | (sys,0) -> []
  | (a::suite,p) -> if a>p then glouton suite p
                else a::(glouton sys (p-a))
;;
glouton : int list -> int -> int list = <fun>
```

Question 3.

Le prix de 6 euros peut être payé avec le portefeuille $< 5, 2, 2, 2 >$ mais pas par la stratégie gloutonne.

Version itérative :

```
let paye_glouton pf p =
  let reste_pf = ref pf and reste_p = ref p and s_prov = ref [] in
    while (!reste_p > 0) & (!reste_pf <> []) do
      match hd !reste_pf with
        | a when a > !reste_p -> reste_pf := tl !reste_pf
        | _ -> reste_p := !reste_p - hd(!reste_pf);
               s_prov := hd (!reste_pf) :: !s_prov;
               reste_pf := tl !reste_pf
    done;
    if (!reste_p = 0) then rev !s_prov else []
;;
paye_glouton : int list -> int -> int list = <fun>
```

Version récursive :

```
let rec paye_glouton pf p = match pf with
  | [] -> []
  | a::suite -> if a>p then paye_glouton suite p
                else let s = paye_glouton suite (p-a) in
                  if (s<>[]) || (p-a=0) then a::s
                  else [];;
paye_glouton : int list -> int -> int list = <fun>
```

Question 4.

On notera bien que l'on cherche ici le cardinal de l'ensemble des suites (finies) extraites de la suite portefeuille dont la valeur est p. Ainsi si le portefeuille est $< 10, 10, 10 >$ et le prix 20, le cardinal cherché est 3 et non pas 6.

```
let rec compte_paiements pf p = match pf with
  | [] -> if p=0 then 1 else 0
  | a::suite -> match a with
      | _ when a>p -> compte_paiements suite p
      | _ -> compte_paiements suite (p-a) + compte_paiements suite p
;;
compte_paiements : int list -> int -> int = <fun>
```

Question 5.

- Fonction *ajoute*.

```
let rec ajoute s d = match s with
  | [] -> [d]
  | a::suite -> if d>=a then d::s else a::(ajoute suite d)
;;
ajoute : 'a list -> 'a -> 'a list = <fun>
```

- Fonction $diff$.

Version récursive non terminale :

```
let rec diff s1 s2 =
  match (s1,s2) with
    | _ when s2 = [] -> s1
    | _ when s1 = [] -> []
    | (a1::suite1,a2::suite2) -> match 1 with
        | _ when a1>a2 -> a1::diff suite1 s2
        | _ when a1=a2 -> diff suite1 suite2
        | _ -> diff s1 suite2
;;
diff : 'a list -> 'a list -> 'a list = <fun>
```

Version récursive terminale :

```
let rec diff_aux reste1 reste2 rev_diff_prov =
  match (reste1,reste2) with
    | _ when reste2 = [] -> (rev rev_diff_prov) @ reste1
    | _ when reste1 = [] -> rev rev_diff_prov
    | (a1::suite1,a2::suite2) -> match 1 with
| _ when a1>a2 -> diff_aux suite1 reste2 (a1::rev_diff_prov)
        | _ when a1=a2 -> diff_aux suite1 suite2 rev_diff_prov
        | _ -> diff_aux reste1 suite2 rev_diff_prov
;;
let diff s1 s2 = diff_aux s1 s2 [];;
```

Question 6.

$T(0) =< 0 >$, $M(0) =<>$,
$T(1) =< 10, 5, 1 >$, $M(10) =< 10 >$, $M(5) =< 5 >$ et $M(1) =< 1 >$,
$T(2) =< 20, 15, 11, 6 >$, $M(20) =< 10, 10 >$, $M(15) =< 10, 5 >$, $M(11) =< 10, 1 >$ et $M(6) =< 5, 1 >$.

Question 7.

L'algorithme est basé sur la remarque (non évidente à moins qu'une solution simple m'échappe) que chaque élément de $T(i+1)$ est la somme d'un élément de $T(i)$ (mettons *indice*) et d'une espèce de pf privé d'une somme optimale payant *indice* c'est à dire *tab*(*indice*).
En effet soit une somme $s =< e_1, e_2, \ldots, e_{i+1} >$ de taille $i+1$ et optimale pour un portefeuille pf. Alors au moins une somme obtenue en retirant une espèce est optimale. En effet si toutes les espèces d_k sont distinctes 2 à 2 alors la somme obtenue en retirant d_{i+1} est optimale (raisonnement par l'absurde pénible mais je ne vois pas plus simple) et sinon la somme obtenue en retirant une espèce figurant au moins deux fois est optimale (idem).

On parcourt $T(i)$. Pour chaque valeur rencontrée, mettons *indice* :
on parcourt pf privé de la somme *tab*(*indice*) qui est une somme optimale pour le prix *indice* et on ajoute chaque espèce rencontrée à *indice* ce qui fournit un prix *prix*.
Si $prix > p$ (on ne s'intéresse pas aux prix plus grands que p) ou si la case d'indice *prix* de *tab* est non vide (ce qui signifie que *prix* a déjà été payé par une somme extraite de taille plus petite), on ne fait rien.
Sinon on ajoute *prix* à la liste $T(i+1)$ et on remplit la case d'indice *prix* de *tab* par la somme obtenue en ajoutant l'espèce rencontrée à la somme dans *tab*(*indice*).

Pour ce qui est du tableau global *tab*, on notera bien qu'il doit bien entendu être défini avant la définition de *etape* et que, même si on redéfinit *tab* après, c'est la définition de *tab* au moment de la définitrion de *etape* qui est prise en compte par la fonction *etape*.

Pour la fonction *optimal* : il suffit une fois le tableau *tab* défini (de taille $p+1$ et initialisé par des listes vides), de définir la fonction *etape* puis la fonction *optimal* ci-dessous qui, commençant par $T(0) = [0]$, applique itérativement *etape* jusqu'à avoir un résultat vide et lit alors la case d'indice p du tableau *tab*.

```
let tab = make_vect 41 [];; (* par exemple *)

let etape pf p l = let reste_l = ref l and result = ref [] in
  while !reste_l <> [] do
    let indice = hd !reste_l in
    let reste_pf = ref (diff pf tab.(indice)) in
      while !reste_pf <> [] do
        let prix = indice + (hd !reste_pf) in
          if (prix <= p) & (tab.(prix) = []) then
            begin
              result := prix :: !result;
              tab.(prix) <- ajoute tab.(indice) (hd !reste_pf)
            end;
        reste_pf := tl !reste_pf
      done;
    reste_l := tl !reste_l;
  done;
  !result
;;
etape : int list -> int -> int list -> int list = <fun>
```

```
let optimal pf p =
  let l = ref [0] in
    while !l <> [] do
      l := etape pf p !l
    done;
    tab.(p)
;;
optimal : int list -> int -> int list = <fun>
```

Question 8.

L'algorithme glouton conduit à payer le prix de 48 avec la somme $< 30, 12, 6 >$ alors qu'elle est payable par la somme $< 24, 24 >$.

Question 9.

```
let tglouton sys p =
  let n = pred (vect_length sys) in
  let reste_p = ref p and indice = ref 1 and s = make_vect (n+1) 0 in
    while (!reste_p > 0) do
      while (sys.(!indice) > !reste_p) do incr indice done;
      s.(!indice) <- !reste_p / sys.(!indice);
      reste_p := !reste_p mod sys.(!indice);
      incr indice
    done;
    s
;;
tglouton : int vect -> int -> int vect = <fun>
```

Cet algorithme est linéaire en n taille du système car il effectue une boucle *while* avec au plus n étapes car à chaque étape l'indice de *sys* augmente d'une unité et s'il atteint l'étape n il s'arrête forcément alors puisque la somme restante est évidemment payable par l'espèce $d_n = 1$. En outre chaque étape est à temps borné (soit une incrémentation de l'indice soit idem et deux opérations arithmétiques). Ainsi cette fonction est au plus linéaire en la taille du système monétaire. □

Question 10.

• Soient $p < q$ et $U = G(p)$ et $V = G(q)$. Soit le i le plus petit indice tel que $u_i \neq v_i$. Il existe bien sinon on aurait $U = V$ et donc $p = q$.
Notons $r = \sum_{k=1}^{i-1} u_k d_k$ si $i \geqslant 2$ et $r = 0$ sinon.
Alors par l'algorithme glouton qui précède, on a :
$u_i = (p - r)/d_i$ et $v_i = (q - r)/d_i$ donc $u_i \leqslant v_i$ puisque $p - r < q - r$ et en fait $u_i < v_i$ puisque $u_i \neq v_i$. □

• Supposons que $G(p)$ contienne au moins une espèce de dénomination d_k et soient $U = G(p)$ et $V = G(p - d_k)$.
Notons $p_i = \sum_{j=1}^{i} u_i d_i$ et $q_i = \sum_{j=1}^{i} v_i d_i$.
Supposons $k > 1$. La division euclidienne fournit $p = u_1 d_1 + r_1$ (avec $0 \leqslant r_1 < d_1$) et surtout avec $r_1 \geqslant d_k$ car $G(p)$ contient au moins une espèce de dénomination d_k. Donc $p - d_k$ s'écrit $u_1 d_1 + (r_1 - d_k)$ avec $0 \leqslant r_1 - d_k < d_1$ ce qui prouve qu'il s'agit de la division euclidienne de $p - d_k$ par d_1 et donc que $v_1 = u_1$ et $p_1 = q_1$.
De même si $k > 2$ on a $u_2 = (p - p_1)/d_2 = (p - q_1)/d_2 = (p - q_1 - d_k)/d_2 = v_2$ car le reste de la division euclidienne de $p - q_1$ par d_2 est supérieur ou égal à d_k. Donc $p_2 = q_2$.
Par itération claire $u_i = v_i$ pour $i < k$ et $p_i = q_i$.
Alors $v_k = (p - d_k - q_{k-1})/d_k = (p - d_k - p_{k-1})/d_k = (p - p_{k-1})/d_k - 1 = u_k - 1$
et $q_k = p_k - d_k$ de sorte que $p - p_k = (p - d_k) - q_k$ et donc $u_i = v_i$ pour $i > k$.
Ce qui prouve bien que $V = U - I_k$ i.e. $G(p - d_k) = G(p) - I_k$. □

• Soit $U = M(p)$. Alors $V = U - I_k$ est un représentant de $p - d_k$. Il est nécessairement optimal sinon on disposerait d'un représentant W de taille strictement plus petite pour $p - d_k$ et alors $W + I_k$ serait un représentant de p de taille strictement plus petite que U ce qui est impossible.
Donc $V = M(p - d_k)$ i.e $M(p - d_k) = M(p) - d_k$. □

Question 11.

• Supposons qu'il existe k tel que $m_k > 0$ et $g_k > 0$ et considérons le prix $\omega' = \omega - d_k$ (qui est bien positif ou nul puisque $\omega \geqslant d_k$ puisque par exemple $m_k \geqslant 1$).

Comme $\omega' < \omega$ (à noter une erreur d'énoncé évidente à rectifier : $M(\omega') = G(\omega')$ pour $\omega' < \omega$ et non $\omega' > \omega$) on a $M(\omega') = G(\omega')$.

Mais alors, compte tenu de la question précédente on a $M(\omega) = M(\omega') + I_k = G(\omega') + I_k = G(\omega)$.

Contradiction. □

• Supposons $i = 1$. Alors $m_1 > 0$. Donc $\omega \geqslant d_1$. Donc $g_1 > 0$. Contradiction avec la question précédente car on aurait alors m_1 et g_1 strictement positifs. □

• a/ Notons déjà que comme $i > 1$ il est bien licite de considérer l'indice $i - 1$.

Comme $m_i > 0$ on a $\omega \geqslant d_i$.

Par ailleurs puisque $g_i = 0$ (question 11.a.), on a $\omega - \sum_{k=1}^{i-1} g_k d_k < d_i$ donc $\omega - \sum_{k=1}^{i-1} g_k d_k < \omega$ puisque $d_i \leqslant \omega$. Ainsi $\sum_{k=1}^{i-1} g_k d_k > 0$ donc $\sum_{k=1}^{i-1} g_k d_k \geqslant d_{i-1}$ puisque la suite (d_k) est décroissante.

Or $\omega = \sum_{k=1}^{n} g_k d_k \geqslant \sum_{k=1}^{i-1} g_k d_k$ donc $\omega \geqslant d_{i-1}$.

En outre si $\omega = d_{i-1}$ on a clairement $G(\omega) = M(\omega) = (x_k)$ avec $x_{i-1} = 1$ et $x_k = 0$ si $k \neq i - 1$ ce qui est exclu par définition de ω. Ainsi il vient $\omega > d_{i-1}$.

b/ Comme $m_j > 0$ il vient $\omega \geqslant d_J$ et il est donc licite de considérer le prix $\omega' = \omega - d_j$ pour lequel on a $G(\omega') = M(\omega')$ puisque $\omega' < \omega$. Or $M(\omega') = M(\omega) - I_j$ (question 10.c.).

Ainsi $G(\omega') = M(\omega') = \sum_{k=i}^{j-1} m_k d_k + (m_j - 1)d_j$ donc toutes les composantes de $G(\omega')$ d'indice strictement inférieur à i sont nulles ce qui prouve que $\omega' < d_{i-1}$ i.e. $\omega < d_{i-1} + d_j$.

c/ en conclusion on a $d_{i-1} < \omega < d_{i-1} + d_j$. □

Question 12.

• Notons $G(d_{i-1} - 1) = (0, \ldots, 0, g'_i, \ldots, g'_n)$ (les $i - 1$ premières composantes sont bien nulles).

L'inégalité de l'énoncé s'écrit alors :

$(0, \ldots, 0, m_i \ldots, m_{j-1}, m_j - 1, 0, \ldots, 0) \leqslant_\ell (0, \ldots, 0, g'_i, \ldots, g'_n) <_\ell (0, \ldots, 0, m_i \ldots, m_{j-1}, m_j, 0, \ldots, 0)$

Il en découle pour k de i à $j - 1$ que $m_k \leqslant g'_k \leqslant m_k$ donc $m_k = g'_k$.

Il vient alors $m_j - 1 \leqslant g'_j \leqslant m_j$. Or si $g'_j = m_j$ il vient, tenant compte de l'inégalité de droite que $g'_{j+1} \leqslant 0$ donc $g'_{j+1} = 0$ puis de même $g'_{j+2} = 0$ et finalement $g'_k = 0$ pour k de $j + 1$ à n ce qui signifie que $G(d_{i-1} - 1) = M(\omega)$ ce qui est contraire à l'inégalité stricte de droite.

En conclusion on a $m_k = g'_k$ pour k de i à $j - 1$ et $m_j = g'_j + 1$. □

• Notons que cela implique que pour un couple (i, j) donné il n'y a qu'un seul contre-exemple possible.

• L'algorithme consiste à essayer les différents contre-exemples possibles.

Le nombre de couples (i, j) possibles est $(n - 2) + (n - 1) + \cdots + 1 = O(n^2)$ (pour $i = 2$ il y a $n - 2$ valeurs possibles de j : de 3 à n, etc ...)

Une fois un couple (i, j) choisi, on calcule successivement $G(d_{i-1} - 1)$ (coût en $O(n)$ par la question 9), puis le vecteur $M(\omega)$ possible (coût constant) puis la valeur du ω candidat (coût en $O(n)$ pour le calcul de la somme) puis le vecteur $G(\omega)$ (coût en $O(n)$). Il reste alors à calculer la taille de $M(\omega)$ et $G(\omega)$ (coût en $O(n)$) et à tester l'inégalité stricte. Ainsi pour un couple (i, j) fixé le coût est-il un $O(n)$.

En conclusion, dans le cas le plus défavorable c'est à dire celui d'un système canonique, le coût est-il un $O(n^3)$.

Le programme suivant renvoie un couple de $bool \times int$ égal à $(true, 0)$ si le système est canonique et $(false, \omega)$ où ω est un contre-exemple sinon.

```
let canonique sys =
  let n = pred (vect_length sys) and reponse = ref true and contre_exemple = ref 0 in
  let i = ref n in let j = ref !i in
    while (!reponse = true) & (!i > 1) & (!j <= n) do
      let glouton = tglouton sys (sys.(!i-1)-1) in
        let omega = ref (glouton.(!i) * sys.(!i)) in
          for k = succ !i to !j do omega := !omega + (glouton.(k) * sys.(k)) done;
          omega := !omega + sys.(!j);
          let vecteur = tglouton sys !omega in
            let taille1 = ref 0 and taille2 = ref (glouton.(!i) + 1) in
              for k =1 to n do taille1 := !taille1 + vecteur.(k) done;
              for k = succ !i to !j do taille2 := !taille2 + glouton.(k) done;
              match (!taille2 < !taille1) with
                | true -> reponse := false ; contre_exemple := !omega
                | _ -> if !j <> n then incr j else begin decr i; j := !i end
    done;
    (!reponse,!contre_exemple)
;;
canonique : int vect -> bool * int = <fun>

let sys1 = [|0;500;200;100;50;20;10;5;2;1|];;
let sys2 = [|0;240;60;30;24;12;6;3;1|];;

canonique sys1;;
- : bool * int = true, 0

canonique sys2;;
- : bool * int = false, 48
```

FIN