X Informatique MP 2002 — Corrigé

Ce corrigé est proposé par Mathieu Giraud (ENS Lyon) ; il a été relu par Xavier Goaoc (ENS Cachan) et Sébastien Desreux (ENS Ulm).

Ce sujet comporte trois parties consacrées à l'étude des porte-monnaie et des systèmes monétaires. Chaque partie suit une progression logique même si quelques questions sont autonomes.

- La première partie ouvre le sujet par des questions traitant du paiement exact et de ses variantes. Elle est majoritairement composée de questions de programmation manipulant des listes.
- La deuxième partie s'intéresse à la recherche d'un paiement optimal. Elle commence par quelques questions sur des listes et l'étude d'un exemple concret.
- La troisième partie concerne la canonicité des systèmes monétaires par l'algorithme polynomial de Pearson. Des questions plus théoriques interviennent (algorithmes, complexités). Elles exigent une bonne compréhension des définitions, notamment au sujet de l'ordre lexicographique, ainsi que l'assimilation des notions des deux parties précédentes. La programmation met ici en jeu des tableaux.

La programmation est très présente dans ce sujet. Au début de chaque partie figurent des applications directes du cours, comme des parcours de listes ou de tableaux. Les programmes demandés dans les fins de partie sont plus conséquents et demandent plus d'initiative.

Les fonctions traitant des listes sont le plus souvent récursives. Le langage Caml est particulièrement adapté à ce type de fonctions et rend des solutions particulièrement élégantes, ce qui n'est pas toujours le cas avec Pascal.

L'épreuve d'informatique du concours Centrale-Supélec 2002 était similaire à celle-ci.

Indications

1 La récursivité fournit des solutions simples et efficaces lorsqu'on utilise des listes.

Partie I

2.a Quelle est la valeur de d_n ?

2.b Parcourir la liste.

3.a Chercher par exemple dans le portefeuille $\langle 50, 20, 20, 20\rangle$.

3.b Parcourir le portefeuille `pf` tout en construisant la liste demandée.

4 Écrire une fonction récursive qui explore l'arbre binaire dont les feuilles sont toutes les sommes possibles.

Partie II

6 Commencer par déterminer T(0), en déduire T(1) puis T(2). Observer avec attention le cas de la valeur 10.

7.a La technique de la question 6 peut se formaliser en

$$\mathrm{T}(i+1) = \{\, c = a + b \mid a \in \mathrm{T}(i) \,\mathrm{et}\, b \in \mathrm{P} \setminus \mathrm{M}_i(a) \,\} \setminus \mathrm{T}(i)$$

Effectuer de manière imbriquée les parcours de $\mathrm{T}(i)$ et de $\mathrm{P} \setminus \mathrm{M}_i(a)$. Utiliser les fonctions définies aux questions 5.a et 5.b.

7.b Faire boucler `etape` en arrêtant dès que le représentant optimal est déterminé ou impossible à réaliser.

Partie III

8 La dénomination 30 n'est pas le double de la précédente.

9 Utiliser la division entière.

10.a Observer le comportement de l'algorithme glouton lorsqu'il arrive à la première dénomination où $\mathrm{G}(p)$ et $\mathrm{G}(q)$ diffèrent.

10.b Remarquer que $\mathrm{G}(p - d_k) + \mathrm{I}_k$ est un représentant de p et que $\mathrm{G}(p) - \mathrm{I}_k$ est un représentant de $p - d_k$.

10.c Utiliser le même raisonnement qu'à la question 10.b.

11.a Il y a une erreur d'énoncé dans la définition du contre-exemple *minimal*: la seconde condition concerne les w' tels que $w' < w$ (et non $w' > w$). Raisonner par l'absurde et utiliser les résultats des questions 10.b et 10.c.

11.b Raisonner par l'absurde en considérant la première étape de l'algorithme glouton.

11.c Prouver séparément chaque inégalité.

12.a Réécrire l'inégalité admise en faisant apparaître toutes les composantes.

12.b Essayer la technique de la question 12.a sur toutes les paires (i, j) envisageables pour traquer un contre-exemple minimal. On pourra supposer l'existence de fonctions donnant la valeur et le poids d'une somme, et commencer par une fonction auxiliaire appliquant le résultat de la question 12.a. Utiliser aussi la fonction définie à la question 9.

12.c Mettre en œuvre l'algorithme de la question 12.b.

> Dans tout ce corrigé, les listes en Caml sont filtrées par le motif
> `t::q`
> (**t**ête et **q**ueue).

1

Version Caml

```
let rec valeur (s : somme) = match s with
  | [] -> 0
  | t::q -> t + valeur (q : somme) ;;
```

Version Pascal

```
function valeur(s:somme) : integer ;
begin
  if (s = nil)
  then valeur := 0
  else valeur := s.contenu + valeur(s.suivant) ;
end ;
```

I. Payer le compte exact

2.a Par hypothèse, la dernière dénomination d_n vaut 1. À sa dernière étape, l'algorithme glouton peut toujours utiliser p fois cette dernière espèce pour payer le prix p.

La stratégie gloutonne réussit toujours.

2.b On suppose ici que l'acheteur dispose toujours des espèces dont il a besoin et on parcourt la liste `sys` pour déterminer le paiement glouton. Puisque $d_n = 1$, cette stratégie réussit toujours d'après la question précédente : on ne teste donc pas le cas où la liste `sys` est vide.

Version Caml

```
let rec glouton sys p = match (sys, p) with
  |    _, 0 -> []
  | t::q, _ -> if (t <= p)
               then t::(glouton sys (p-t))
               else glouton q p ;;
```

Version Pascal

```
function glouton(sys:syteme; p:integer) : somme ;
begin
  if (p = 0)
  then glouton := nil
  else if (sys.contenu <= p)
       then glouton := cons(sys.contenu,
                            glouton(sys, p - sys.contenu))
       else glouton := glouton(sys.suivant, p) ;
end ;
```

3.a Dans le système européen, la stratégie gloutonne appliquée au prix de 60 euros pour le portefeuille $\langle 50, 20, 20, 20\rangle$ commence par utiliser le billet de 50 euros puis échoue, alors que la somme $\langle 20, 20, 20\rangle$ conviendrait. Ainsi :

> La stratégie gloutonne tenant compte des ressources de l'acheteur peut échouer.

3.b

> La version Caml présentée ici utilise les exceptions pour sortir de la boucle interne. La version Pascal est plus classique.

Version Caml

```
exception Echec_Glouton ;;

let rec paye_glouton_ex pf p = match (pf, p) with
  |    _, 0 -> []
  |   [], _ -> raise Echec_Glouton
  | t::q, _ -> if (t <= p)
               then t::(paye_glouton_ex q (p-t))
               else     paye_glouton_ex q   p  ;;

let paye_glouton pf p =
  try paye_glouton_ex pf p with
    Echec_Glouton -> [] ;;
```

Version Pascal

```
function paye_glouton(pf:somme; p:integer) : somme ;
var resultat : somme ;
begin
  resultat := nil ;
  while (pf <> nil and p > 0) do
    begin
      if (pf.contenu <= p) then
        begin
          p := p - pf.contenu ;
          resultat := cons(pf.contenu, resultat) ;
        end
      pf := pf.suivant ;
    end ;

  if (p > 0)
    then
      paye_glouton := nil
    else
      paye_glouton := resultat ;
end ;
```