# Devoir surveillé n° 4 — corrigé

## I Le problème du sac à dos

I.1) En notant $\ell$ et $p$ la longueur et le poids d'une corde, on remarque que

- $2p(D) = p(C)$ et $2\ell(D) > \ell(C)$ donc on ne choisira aucune corde $C$,
- $2p(B) = p(A)$ et $2\ell(B) < \ell(A)$ donc on choisira au plus une corde $B$,
- il est toujours préférable de prendre une corde $A$ avec les contraintes ci-dessus.

Afin de ne pas dépasser 10 kg, il y a alors deux possibilités :

- soit une corde $A$ et deux cordes $D$, ce qui donne une longueur de 48 m,
- soit une corde $A$ et une corde $B$, soit une longueur de 44 m.

On choisit donc une corde $A$ et deux cordes $D$.

I.2) Il est toujours préférable de prendre au moins deux cordes et on ne peut pas dépasser 3 cordes compte tenu des contraintes.

Avec trois cordes, l'optimum est atteint pour des cordes de poids total maximal, soit une corde $A$ et une corde $C$ pour une longueur de 46 m, et la seule possibilité pour trois cordes est de prendre les cordes $B$, $C$ et $D$ pour une longueur de 39 m.

On choisit donc une corde $A$ et une corde $C$.

I.3) Si $w = 0$, la capacité du sac est nulle, donc $s(0, j) = 0$.

Si $j = 0$, on ne prend pas d'objet, donc $s(w, 0) = 0$.

I.4) Soit $j \geqslant 1$. Dans $s(w', j-1)$ on est sûr de ne pas avoir pris l'objet $j$, donc il y a deux possibilités :

- soit on prend l'objet $j$, ce qui n'est possible que si $w \geqslant w_j$, et on obtient, avec cette hypothèse, la valeur optimale $s(w - w_j, j-1) + v_j$,
- soit on ne prend pas l'objet $j$ et on obtient, avec cette hypothèse, la valeur optimale $s(w, j-1)$.

On en déduit que

$$s(w,j) = \begin{cases} \max(s(w-w_j, j-1) + v_j, s(w, j-1)) & \text{si } w \geqslant w_j, \\ s(w, j-1) & \text{sinon.} \end{cases}$$

I.5) Une version avec une matrice :

```
let sac_a_dos capacite poids valeurs =
  let n = vect_length poids in
  let s = make_matrix (capacite + 1) (n + 1) 0 in
  for j = 1 to n do
    let w_j, v_j = poids.(j - 1), valeurs.(j - 1) in
    for w = 1 to w_j - 1 do
      s.(w).(j) <- s.(w).(j - 1)
    done;
    for w = w_j to capacite do
      let max_avec_j = s.(w - w_j).(j - 1) + v_j in
      s.(w).(j) <- max s.(w).(j - 1) max_avec_j
    done;
  done;
s.(capacite).(n)
;;
```

on peut aussi l'écrire avec un vecteur `s_j` qui à la fin de l'étape `j` va contenir le tableau $[|s(0,j); s(1,j); \cdots; s(W$

```
let sac_a_dos capacite poids valeurs =
  let n = vect_length poids in
  let s_j = make_vect (capacite + 1) 0 in
  for j = 1 to n do
    let s_j_moins_1 = copy_vect s_j in
    let w_j, v_j = poids.(j - 1), valeurs.(j - 1) in
    (* On ne met à jour que lorsque w >= w_j *)
    for w = w_j to capacite do
      let max_avec_j = s_j_moins_1.(w - w_j) + v_j in
      if max_avec_j > s_j_moins_1.(w) then
        s_j.(w) <- max_avec_j
    done;
  done;
s_j.(capacite)
;;
```

I.6) Il suffit d'initialiser la matrice à `false` puis, à chaque étape $(j,w)$, si on a $w \geqslant w_j$ (deuxième boucle **for** sur `w`) et $s(w-w_j, j-1) + v_j > s(w, j-1)$, de mettre `utilise.(w).(j)` à `true`.

I.7)

```
let rec retrouve_solution w j poids utilise =
  match j with
  | 0 -> []
  | _ ->
    if utilise.(w).(j) then
      let w_1 = w - poids.(j - 1) in
      j :: (retrouve_solution w1 (j - 1) poids utilise)
    else
      retrouve_solution w (j - 1) poids utilise
;;
```

I.8) La complexité temporelle de la fonction `sac_a_dos` est $\boxed{O(nW)}$ avec les notations du sujet, directement avec les deux boucles imbriquées.
La complexité spatiale est $\boxed{O(nW)}$ avec la version matrice (ou avec `utilise`), et $\boxed{O(W)}$ avec la version vecteur, mais alors il n'est plus possible a priori de reconstruire la solution optimale.

I.9) L'avantage d'une version récursive avec mémoïsation est qu'elle ne nécessite pas de remplir l'intégralité du tableau : on part des données initiales et on descend jusqu'à un cas de base seulement puis on remonte en remplissant le tableau pour les cas rencontrés seulement. Si toutes les données sont multiples de 100, on ne remplira que des cases sur des lignes multiples de 100.

I.10) On a $\boxed{k(w) = \max\left\{k(w-w_j) + v_j \,;\, 1 \leqslant j \leqslant n \text{ tel que } w_j \leqslant w\right\}.}$

I.11)
```
sac_a_dos_repetition(W, poids, valeurs) =
    Initialiser un tableau k de taille W + 1 avec des 0
    Pour w allant de 1 à W, faire :
        k[w] <- max (k[w - poids[j - 1]] + valeur[j - 1]
                              pour 1 <= j <= n tel que w_j <= w)
    Finpour
    Retourner k[W]
```

I.12) La complexité temporelle est $\boxed{O(nW)}$ (boucles sur `w` et sur `j`), la complexité spatiale est $\boxed{O(W).}$

I.13) Les données en entrée étant codées en binaire dans la machine, on peut choisir d'exprimer la complexité en fonction de la place nécessaire pour coder ces entiers et elle devient donc exponentielle.

I.14) Il suffit que chaque objet apparaisse autant de fois que son nombre dans `poids` et `valeurs`.

# II Arbres

## 1. Préliminaires : Arbre binaire d'entiers

II.1 On a $|a| = \begin{cases} -1 & \text{si } a = \emptyset \\ 1 + \max(|\mathcal{G}(a)|, |\mathcal{D}(a)|) & \text{sinon} \end{cases}$

II.2 Cette question est mal posée car il n'y a pas un seul et unique arbre qui minimise ni maximise la profondeur [1]. Un arbre pour $n$ étiquettes (et donc $n$ nœuds [2]) qui maximise la profondeur est un arbre filiforme (tout nœud sauf un est de degré 1) qui est de profondeur $n-1$. Je ne sais pas caractériser un arbre qui minimise la profondeur. Tout arbre de profondeur $\lceil \log_2(n+1) \rceil - 1$ convient. La réponse attendue est peut-être un arbre quasi-complet, c'est-à-dire que tous les niveaux sauf éventuellement le dernier sont remplis, mais cela n'est pas nécessaire. Avec 4 nœuds par exemple, tous les arbres de profondeur 2 conviennent (c'est-à-dire tous sauf les arbres filiformes) et il y a deux arbres non quasi-complets. De manière plus générale, si un des deux sous-arbres de la racine est complet, alors on minimise également la profondeur, mais encore une fois ceci n'est pas nécessaire.

II.3 Par récurrence (finie !) bien rédigée car c'est encore le début du sujet sur les niveaux de l'arbre (s'ils existent !). En substance, s'il y a $k$ nœuds à un niveau, il y en a $2k$ au suivant et il y a 1 nœud (la racine) au niveau 0.

II.4 On a alors $n = \sum_{k=0}^{p} 2^k = 2^{p+1} - 1$

II.5 On en déduit que $p = \log_2(n+1) - 1$.

## 2. Problème : Représentation de systèmes creux

II.11 La question doit se lire : « Montrer que l'ensemble des numéros possibles des nœuds à une profondeur $p$ est égal à l'intervalle $[\![2^p, 2^{p+1} - 1]\!]$ ». Montrons cette propriété par récurrence sur $p \in \mathbb{N}$, la vérification étant immédiate pour $p = 0$. Supposons donc que les $2^p$ nœuds (éventuels) à une profondeur $p$ aient des numéros décrivant $I_p = [\![2^p, 2^{p+1} - 1]\!]$. Quand $n$ parcourt $I_p$, $n + 2^p$ parcourt $I_p^g = [\![2^{p+1}, 2^{p+1} + 2^p - 1]\!]$ et $n + 2^{p+1}$ parcourt $I_p^d = [\![2^p + 2^{p+1}, 2^{p+2} - 1]\!]$. Ces deux ensembles sont disjoints et d'union $I_{p+1} = [\![2^{p+1}, 2^{p+2} - 1]\!]$, ce qui établit l'hérédité.

II.12 Comme il y a $2^p$ nœuds possibles à la profondeur $p$ dont les numéros décrivent un ensemble de $2^p$ éléments, les nœuds de profondeurs $p$ ont des numéros deux à deux distincts. De plus pour $p \neq q$ on a $I_p \cap I_q = \emptyset$ ce qui montre que la numérotation est unique.

II.13 Montrons cette propriété par récurrence sur $p \in \mathbb{N}$. Si $p = 0$, le seul nœud à cette profondeur est la racine d'étiquette $n = 1$ et d'occurrence $\langle\rangle$, la propriété est vérifiée. Soit $p \in \mathbb{N}$ tel que la propriété est vraie jusqu'au rang $p$ et considérons un nœud de profondeur $p+1$ et d'occurrence $\langle c_1, \dots, c_p, c_{p+1} \rangle$ (il peut n'y en avoir aucun mais dans ce cas la propriété est trivialement vérifiée). Par hypothèse de récurrence, le père de ce nœud d'occurrence $\langle c_1, \dots, c_p \rangle$ a pour numéro $m = 2^p + \sum_{i=0}^{p-1} c_{i+1} 2^i$. Il vient deux cas :

- Soit le nœud est le fils gauche de son père, alors $c_{p+1} = 0$ et

$$n = m + 2^p = 2^{p+1} + \sum_{i=0}^{p-1} c_{i+1} 2^i = 2^{p+1} + \sum_{i=0}^{p} c_{i+1} 2^i$$

- Soit le nœud est le fils droit de son père, $c_{p+1} = 1$ et

$$n = m + 2^{p+1} = 2^p + 2^{p+1} + \sum_{i=0}^{p-1} c_{i+1} 2^i = 2^{p+1} + \sum_{i=0}^{p} c_{i+1} 2^i$$

1. Sauf si $n = 1$.
2. J'imagine que l'on suppose les étiquettes distinctes et que l'on n'autorise pas une même étiquette à apparaître sur plusieurs nœuds, sinon on peut trouver des arbres arbitrairement profonds.

Dans les deux cas on vérifie la propriété pour le nœud de numéro $n$, ceci pour tout nœud (éventuel) à la profondeur $p+1$ et l'hérédité est prouvée.

II.14 La question précédente a montré que $\overline{1c_p \dots c_1}$ est la décomposition en binaire de $n$. On voit alors que $c_i$ est le reste de la division de $\left\lfloor \frac{n}{2^{i-1}} \right\rfloor$ par 2.

II.15 On propose sans problèmes

```
let rec taille arbre =
  match arbre with
   | Vide -> 0
   | Fourche (fg, fd) -> (taille fg) + (taille fd)
   | Noeud (fg, _, fd) -> 1 + (taille fg) + (taille fd)
;;
```

II.16 C'est sans doute la question la plus difficile de cette partie. Il faut connaître l'indice d'une position (*i.e.* le numéro d'un nœud) parcourue. On propose donc de parcourir l'arbre à l'aide d'une fonction auxiliaire `bornes_aux : int -> int -> arbre` qui connaît le numéro du nœud courant ainsi que la profondeur. Comme on n'a besoin que des puissances des profondeurs, on utilise directement celle-ci pour des raisons d'efficacité.

```
let bornes arbre =
  (* Bornes du sous-arbre enraciné au noeud de numero 'n'
     à une profondeur 'p' avec 'puiss = 2^p' *)
  let rec bornes_aux n puiss arbre =
    match arbre with
    | Vide -> (max_int, 0)
    | Fourche (fg, fd) ->
       let min_g, max_g = bornes_aux (n + puiss) (2 * puiss) fg in
       let min_d, max_d = bornes_aux (n + 2 * puiss) (2 * puiss) fd in
       (min min_g min_d, max max_g max_d)
    | Noeud (fg, _, fd) ->
       let min_g, max_g = bornes_aux (n + puiss) (2 * puiss) fg in
       let min_d, max_d = bornes_aux (n + 2 * puiss) (2 * puiss) fd in
       (min n (min min_g min_d), max n (max max_g max_d))
  in
  match arbre with
   | Vide -> failwith "Arbre vide"
   | _ -> bornes_aux 1 1 arbre
;;
```

II.17 Le dernier bit de la décomposition en binaire de l'indice recherché permet de savoir dans quel fils il faut aller.

```
let rec remplacer i elt arbre =
  match arbre with
  | Vide -> failwith "Pas d'élément à cet indice"
  | Fourche (fg, fd) ->
     if i = 1 then
       failwith "Pas d'élément à cet indice"
     else if i mod 2 = 0 then
       Fourche (remplacer (i / 2) elt fg, fd)
     else
       Fourche (fg, remplacer (i / 2) elt fd)
  | Noeud (fg, val, fd) ->
     if i = 1 then
       Noeud (fg, elt, fd)
     else if i mod 2 = 0 then
       Noeud (remplacer (i / 2) elt fg, val, fd)
     else
       Noeud (fg, val, remplacer (i / 2) elt fd)
;;
```

II.19 On propose de modifier la fonction `bornes_aux` pour renvoyer le quadruplet composé de l'indice minimum, de l'indice maximum, du nombre de nœuds et d'un booléen qui indique si l'arbre contient une fourche dont les deux fils sont vides ou une étiquette égale à `def`. On n'effectue bien qu'un seul parcours de l'arbre.

La fonction `valider` s'écrira alors immédiatement

```
let valider (imin, imax, taille, def, arbre) =
  let (mini, maxi, t, prob) = bornes_plus 1 1 def arbre in
  (imin = mini) && (imax = maxi)  && (taille = t) && (not prob)
;;
```

```
let rec bornes_plus n puiss def arbre =
  match arbre with
  | Vide -> (max_int, 0, 0, false)
  | Fourche (fg, fd) ->
     let min_g, max_g, taille_g, probleme_g =
       bornes_plus (n + puiss) (2 * puiss) def fg
     in
     let min_d, max_d, taille_d, probleme_d =
       bornes_plus (n + 2 * puiss) (2 * puiss) def fd
     in
     (min min_g min_d, max max_g max_d,
      taille_g + taille_d, fg = Vide && fd = Vide)
  | Noeud (_, val, _) when val = def ->
      (-1, -1, -1, true)
  | Noeud (fg, _, fd) ->
     let min_g, max_g, taille_g, probleme_g =
       bornes_plus (n + puiss) (2 * puiss) def fg
     in
     let min_d, max_d, taille_d, probleme_d =
       bornes_plus (n + 2 * puiss) (2 * puiss) def fd
     in
     (min n (min min_g min_d), max n (max max_g max_d),
      taille_g + taille_d + 1, false)
;;
```

II.20 C'est à peu de choses près la même chose que pour la fonction `rechercher`, sans oublier de renvoyer la valeur par défaut si on ne tombe pas sur un nœud, ce que l'on peut quelques fois détecter si l'on

sort de l'intervalle $[imin, imax]$.

```
let lire i (imin, imax, taille, def, arbre) =
  let rec trouver i arbre =
    match i, arbre with
    | _, Vide -> def
    | 1, Fourche _ -> def
    | _, Fourche (fg, fd) -> trouver (i / 2) (if i mod 2 = 0 then fg else
     ↪ fd)
    | 1, Noeud (_, elt, _) -> elt
    | _, Noeud (fg, _, fd) -> trouver (i / 2) (if i mod 2 = 0 then fg else
     ↪ fd)
  in
  if (i < imin) || (i > imax) then
    def
  else
    trouver i arbre
;;
```