Corrigé Option informatique - Centrale 2019
Cinquante nuances de Gray

Yann Hermans, Émeric Tourniaire

21 mai 2019

I Code binaire de Gray

Question 1 Il n'est en fait pas possible de répondre à cette question parce que les listes avec OCaml sont des structures persistantes.

Nous créons ici une fonction de signature `int list -> int list * bool` qui répond à la question. Nous convenons aussi que la fonction retournera un n-uplet représentant 0 lors d'un appel sur le dernier des n-uplets.

```
let rec suivant liste = match liste with
  | []     -> [], false
  | (t::q) -> let fin, b = suivant q in
             if b
             then (t::fin), b
             else ((1-t)::fin), t=0
;;
```

Question 2

```
let rec affiche_nuplet liste = match liste with
  | []     -> ()
  | [a]    -> print_int a ; print_newline ()
  | (t::q) -> print_int t ; print_string " ; " ; affiche_nuplet(q)
;;
```

Nous allons également programmer une fonction `init : int -> int list` qui renvoie le premier terme, soit une liste de n zéros.

```
let rec init n = match n with
  | 0 -> []
  | _ -> 0 :: init (n-1)
;;
```

On peut alors programmer la fonction demandée, de signature `tout_afficher : int -> unit`.

```
let tout_afficher n =
  let rec aux_affichage liste =
    affiche_nuplet liste ;
    let s, b = suivant liste in
      if b then aux_affichage s
  in
    aux_affichage (init n)
;;
```

La fonction `aux_affichage` prend en entrée une liste représentant un entier et affiche cette liste ainsi que toutes les suivantes.

Question 3 La fonction demandée est de type `'a -> 'a list list -> 'a list list`.

```
let rec ajout a l = match l with
  | []     -> []
  | (t::q) -> (a::t) :: (ajout a q)
;;
```

Question 4 On écrit deux fonction mutuellement récursives :

```
let rec monte n =
  if n = 0
  then [[]]
  else
    ajout 0 (monte (n-1)) @
    ajout 1 (descend (n-1))
and descend n =
  if n = 0
  then [[]]
  else
    ajout 1 (monte (n-1)) @
    ajout 0 (descend (n-1))
;;
```

Ces deux fonctions ont pour signature `int -> int list list`.

Question 5 Notons u_n le nombre d'appels de `monte n` (ou de `descend n` par symétrie). Dès lors $\forall n \geq 2,\ u_n = 2 + 2u_{n-1}$, $u_1 = 0$ et $u_0 = 0$. Ainsi $\forall n \geq 2,\ u_n + 2 = 2(u_{n-1} + 2)$ et $(u_n + 2)_n$ est une suite géométrique d'où :

$$\forall n \geq 2,\ u_n + 2 = 2.2^{n-1} \text{ et } u_n = 2^n - 2$$

Finalement le nombre d'appels à `monte` ou à `descend` qu'effectue un appel à `monte n` ou à `descend n` est 0 si $n = 0$ et $2^n - 2$ sinon.

Question 6 Notons déjà que la question n'est pas très explicite. En effet, si l'objectif est de réduire la complexité, cela sera difficile d'obtenir une complexité asymptotique en $o(2^n)$, vu que la longueur de la liste finale est 2^n. Si l'objectif est uniquement de proposer un moyen de réduire le nombre d'appels aux fonctions `monte` et `descend`, on peut aussi répondre de manière imbécile qu'il suffit de renommer les fonctions...

La réponse attendue était sans doute plutôt qu'on peut utiliser la mémoïzation dans l'algorithme de manière à calculer une fois pour toutes les listes `monte k` et `descend k` et réutiliser ces valeurs par la suite, mais signalons que cela n'économisera pas grand chose, parce que l'utilisation de la fonction `ajout` (ou la concaténation de listes) deviendront alors prépondérantes dans le calcul de la complexité globale.

Question 7 Signalons déjà que la valeur de $g(k)$ ne dépend pas du nombre n choisi (ce n'est pas évident !). En effet, si on considère n et k tels que $k < 2^n$, alors pour tout $n' > n$, la k-ème valeur dans l'écriture du codage de Gray à n' bits s'obtient en rajoutant successivement des 0 au début.

Supposons maintenant que n soit fixé. Dès lors l'écriture des nombres entre le rang 2^n et $2^{n+1} - 1$ dans le codage de Gray s'obtient en renversant la liste obtenue entre les rangs 0 et $2^n - 1$, et en remplaçant le premier bit par un 1. Les nombres $g(2^n + r)$ et $g(2^n - 1 - r)$ ont donc même représentation à l'exception de leur premier bit de valeur 1 pour $g(2^n + r)$ et 0 pour $g(2^n - 1 - r)$. Ainsi $g(k) = 2^n + g(2^n - 1 - r)$.

Question 8 Montrons la propriété par récurrence forte sur k. Elle est bien entendue vraie pour $k \in \{0;1\}$. Soit k qui s'écrit $b_n \cdots b_0$ en binaire, avec n le plus petit possible (c'est-à-dire que $b_n = 1$). On note $k = 2^n + r$. Notons tout d'abord que $2^n - 1 - r$ correspond au même nombre que r, mais dans lequel on a inversé tous les bits de 0 à $n-1$. Cette transformation laisse invariante les XOR de deux bits consécutifs $b_j \oplus b_{j+1}$, pour $j < n-1$.

Par hypothèse de récurrence, on a alors $\overline{\text{que } g(2^n - 1 - r}$) est constitué des $c_n \cdots c_0$, avec $c_j = b_j \oplus b_{j+1}$ pour tout $j < n-1$. On a également $c_{n-1} = \overline{b_{n-1}} \oplus 0 = \overline{b_{n-1}} \oplus \overline{1} = b_{n-1} \oplus b_n$, c'est-à-dire que $g(2^n - 1 - r)$ admet pour représentation binaire le nombre $a_{n-1} \cdots a_0$. Il ne reste qu'à ajouter que $a_n = b_n \oplus b_{n+1} = 1 \oplus 0 = 1$, et à remarquer alors que la représentation binaire de $2^n + g(2^n - 1 - r)$ est exactement $a_n \cdots a_0$. On obtient alors le résultat attendu d'après le principe de récurrence.

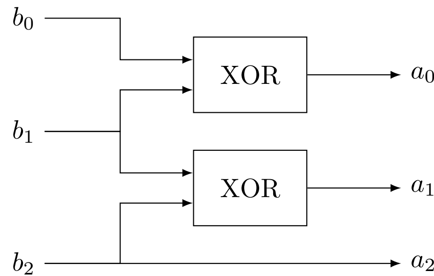
Question 9 D'après la question précédente, et en remarquant que la multiplication par deux décale une représentation binaire d'un cran vers la gauche, on a $g(k) = k \oplus \left\lfloor \frac{k}{2} \right\rfloor$.

Question 10 Montrons tout d'abord que g est injective. Supposons que l'on ait a représenté par $a_n \cdots a_0$ et b représenté par $b_n \cdots b_0$, deux nombres distincts, tels que $g(a) = g(b)$. Notons $a_{n+1} = b_{n+1} = 0$ et considérons k le plus grand indice tel que $a_k \neq b_k$ (donc $a_{k+1} = b_{k+1}$). Nécessairement $a_{k+1} \oplus a_k \neq b_{k+1} \oplus b_k$, ce qui n'est pas possible donc $a = b$ et g est injective.

Montrons maintenant la surjectivité. Soit n un entier naturel, en restreignant la fonction g à $[\![0, 2^{n+1} - 1]\!]$, g reste injective et un argument de cardinalité donne la surjectivité. Dès lors tout nombre $k \in [\![0, 2^{n+1} - 1]\!]$ possède un antécédent par g. Cette propriété étant vérifiée pour tout entier naturel n, g est surjective sur $\mathbb{N}$.

Finalement g réalise une bijection de $\mathbb{N}$ dans $\mathbb{N}$.

Question 11 Si on note b_2, b_1, b_0 les entrées et a_2, a_1, a_0 les sorties, il suffit d'avoir $a_0 = b_0 \oplus b_1$, $a_1 = b_1 \oplus b_2$ et $a_2 = b_2$, d'où le circuit :



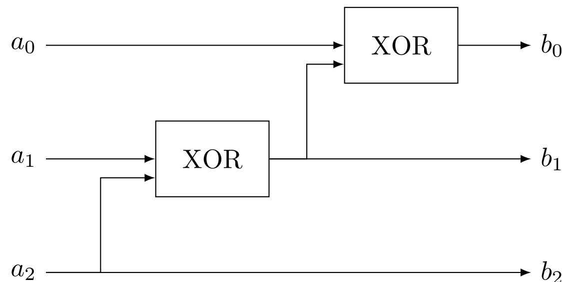
Question 12 Si on note a_2, a_1, a_0 les entrées, on sait d'après la question précédente qu'on a :

— $a_2 = \overline{b_2}$, donc $b_2 = a_2$

— $a_1 = b_1 \oplus b_2$, donc $b_1 = a_1 \oplus b_2$.

— $a_0 = b_0 \oplus b_1$, donc $b_0 = a_0 \oplus b_1$

Ce qui nous conduit au circuit suivant :



Question 13 Les deux questions précédentes se généralisent aisément, on obtient le premier circuit avec $n-1$ portes XOR. Le second circuit peut s'obtenir avec le même nombre de portes en le disposant en cascade.

II Énumération des combinaisons

Question 14 La fonction demandée a pour signature `combinaisons : int -> unit`. Elle utilise `aux_comb : int -> int -> int list list` telle que `aux_comb nb min` renvoie la liste des `nb`-uplets de valeurs comprises entre `min` (inclus) et `max` (exclus), triés par ordre lexicographique où `max` est l'argument de la fonction `combinaisons`.

```
let rec affiche_nuplets liste = match liste with
| []     -> ()
| (t::q) -> affiche_nuplet t; affiche_nuplets q
;;

let combinaisons max =
  let rec aux_comb nb min =
    if max - min < nb then []
    else if nb = 0 then [[]]
    else ajout min (aux_comb (nb-1) (min+1)) @ (aux_comb nb (min+1))
  in
    affiche_nuplets (aux_comb 3 0)
;;
```

Question 15 La première combinaison est toujours $[0; 1; \ldots; p-1]$, et la dernière est $[n-p; n-(p-1); \ldots; n-1]$.

Question 16 Nous supposons ici que l'indication de l'énoncé est : « On pourra commencer par chercher le plus grand indice j tel que $c_{j+1} > c_j + 1$ »

Nous commençons par définir la fonction `plus_grand_indice_j` : int array -> int| prenant en paramètre une combinaison et retournant le plus grand indice j tel que $c_{j+1} > c_j + 1$.

Nous proposons alors une fonction `comb_suivante : int array -> int -> bool` prenant en entrée une combinaison de E_n et l'entier n, qui transforme la combinaison en sa suivante lorsqu'elle existe. En cas d'existence la valeur de retour vaut `true` et `false` sinon.

```
let plus_grand_indice_j c =
  let p = Array.length c in
    let j = ref (p-2) in
      while !j>=0 && c.(!j+1)<=c.(!j)+1 do
          decr j
      done;
      !j
;;
```

```
let comb_suivante c n =
  let p = Array.length c and res = ref true in
    if c.(p-1) <= n-2 then
      c.(p-1) <- c.(p-1)+1
    else
    (
      let j = plus_grand_indice_j c in
        if j == -1 then
          res := false
        else
        (
          c.(j) <- c.(j)+1;
          for i=j+1 to p-1 do
            c.(i) <- c.(j)+i-j
          done
        )
    );
    !res
;;
```

Question 17 Pour énumérer l'ensemble des permutations, on part de la première combinaison et l'on passe successivement aux suivantes à l'aide de la fonction `comb_suivante` jusqu'à aboutir à la dernière.

La première combinaison de E_n est donnée par la fonction `premiere_combinaison : int -> int array` lors d'un appel sur le paramètre n. On se dote d'une fonction d'affichage d'une combinaison `affiche_combinaison : int array -> unit` et enfin de la fonction demandée `affiche_combinaisons : int -> int -> unit` attendant comme paramètres les entiers n et p.

```
let premiere_combinaison p =
    let combi = Array.make p 0 in
        for i=1 to p-1 do
          combi.(i) <- i
        done;
        combi
;;

let affiche_combinaison c =
    let p = Array.length c in
        for i=0 to p-1 do
            if (i>0) then
                print_string " ; ";
            print_int c.(i);
        done;
        print_newline ()
;;

let affiche_combinaisons p n =
    let c = premiere_combinaison p in
        affiche_combinaison c;
        while comb_suivante c n do
            affiche_combinaison c
        done;
;;
```

Question 18 Commençons par rappeler que le nombre total de combinaisons est $\binom{n}{p}$. On cherche à dénombrer toutes les combinaisons qui suivent $c_0 \ldots c_{p-1}$. Parmi celles-ci se trouvent toutes celles qui ne comportent aucun nombre entre 0 et c_0 (inclus), et il y en a $\binom{n-c_0-1}{p}$. Il y a également celles qui commencent par c_0 mais ne comportent alors aucun nombre entre c_0 et c_1 (inclus), et il y en a $\binom{n-c_1-1}{p-1}$. Plus généralement, celles qui commencent par $c_0, \ldots, c_{i-1}$ mais ne comportent aucun nombre entre c_{i-1} et c_i sont en nombre $\binom{n-c_i-1}{p-i}$ (il ne reste qu'à choisir les $p-i$ valeur restantes entre c_i et n).

Au total, il y a donc $\displaystyle\sum_{i=0}^{p-1} \binom{n-c_i-1}{p-i}$ permutations suivant celle que nous avons choisie.

Ainsi le nombre de combinaisons précédent $c_0 \ldots c_{p-1}$ est : $\displaystyle\binom{n}{p} - 1 - \sum_{i=0}^{p-1} \binom{n-c_i-1}{p-i}$ avant.

Question 19 Soit N un entier et soit (n,p) un couple d'entiers tel que $\binom{n}{p} > N$ (p peut être supposé fixé). On considère la combinaison $c_0 \ldots c_p$ de p éléments de E_n qui admet exactement N combinaisons après elle dans l'ordre lexicographique. Dès lors $\displaystyle N = \sum_{i=0}^{p-1} \binom{n-c_i-1}{p-i} = \sum_{k=1}^{p} \binom{n-c_{p-k}-1}{k}$.

Enfin par stricte croissance de la suite $(c_i)_i$ et donc stricte décroissance de la suite $(c_{p-k})_k$, on obtient que les nombres $n - c_{p-i} - 1$ vérifient bien les inégalités demandées.

Question 20 On sait que $\forall k \in [\![0, p-1]\!]$, $\displaystyle\binom{m-k}{p-k} + \binom{m-k}{m-p+1} = \binom{m-k}{m-p} + \binom{m-k}{m-p+1} = \binom{m-k+1}{m-p+1}$ d'après la formule de Pascal. Ainsi :

$$\sum_{k=0}^{p-1} \binom{m-k}{p-k} = \sum_{k=0}^{p-1} \left(\binom{m-k+1}{m-p+1} - \binom{m-k}{m-p+1} \right) = \left(\binom{m+1}{m-p+1} - \binom{m-p+1}{m-p+1} \right) = \binom{m+1}{p} - 1$$

Question 21 Soit $N \in \mathbb{N}$ le plus petit nombre disposant de deux décompositions combinatoires de degré p distinctes, que nous noterons $n_1 < \cdots < n_p$ et $n'_1 < \cdots < n'_p$. Par minimalité de N, on peut supposer $n_p \neq n'_p$ et sans perte de généralité nous supposerons que $n_p < n'_p$ et donc que $n_p + 1 \leq n'_p$.

Par stricte croissance de la suite d'entiers $(n_k)_k$, on obtient que $\forall k \in [\![0, n-1]\!]$, $n_{p-k} \leqslant n_p - k$. Dès lors

$$\sum_{k=1}^{p} \binom{n_k}{k} = \sum_{i=0}^{p-1} \binom{n_{p-i}}{p-i} \leqslant \sum_{i=0}^{p-1} \binom{n_p - i}{p-i} = \binom{n_p+1}{p} - 1 \leq \binom{n'_p}{p} - 1 < \binom{n'_p}{p} \leqslant \sum_{k=1}^{p} \binom{n'_k}{k}$$

On obtient donc $N < N$ ce qui est absurde, d'où l'unicité de la décomposition combinatoire de degré p.

Question 22 On peut l'utiliser pour énumérer toutes les combinaisons possibles. Pour chaque combinaison on évalue la masse totale, et si elle est inférieure ou égale à M, on calcule sa valeur totale. Il ne reste alors plus qu'à déterminer la plus grande valeur totale possible ce qui peut être fait au fur et à mesure de l'exploration des combinaisons.

Question 23 Le nombre d'additions d'un algorithme même un peu subtil (par exemple qui ne calcule pas la valeur d'une combinaison si sa masse dépasse M) semble complexe à évaluer, mais pour un algorithme *vraiment naïf*, on peut estimer que chaque combinaison nécessite $2p-2$ additions ($p-1$ pour les masses et $p-1$ pour les valeurs). Dans ce cas on obtient un total de $2(p-1)\binom{n}{p}$ additions.

Question 24 On peut coder une combinaisons $[a_0, \ldots, a_p]$ par un n-uplet $(x_0, \ldots, x_n)$ tel que $x_i = 1$ si et seulement si $i \in \{a_0, \ldots, a_p\}$, et $x_i = 0$ sinon. Ces n-uplets ont la particularité de contenir exactement p fois la valeur 1.

Question 25 On reprend les fonctions de la question 4 en leur adjoignant un nouveau paramètre `p` pour compter le nombre d'objets restant à choisir. Ces nouvelles fonctions ont donc pour signature : `int -> int -> int list list`

On suppose que l'ordre que les fonctions doivent respecter est celui du codage de Gray comme en question 4.

```
let rec monte n p =
  if p<0 || (n = 0 && p > 0) then []
  else if n = 0 then [[]]
  else
    ajout 0 (monte (n-1) p) @
    ajout 1 (descend (n-1) (p-1))
and descend n p =
  if p<0 || (n = 0 && p > 0) then []
  else if n = 0 then [[]]
  else
    ajout 1 (monte (n-1) (p-1)) @
    ajout 0 (descend (n-1) p)
;;
```

Question 26 Remarquons d'abord que `monte n p` renvoie la liste vide si $p > n$. Si $p \leq n$, notre algorithme va commencer par calculer successivement les valeurs de `monte (n-k) p`, pour rajouter successivement des 0 devant. Ainsi, la première valeur obtenue sera le n-uplet correspondant à la combinaison des p derniers éléments : $(\underbrace{0;\ldots;0}_{n-p \text{ fois}};\underbrace{1;\ldots;1}_{p \text{ fois}})$.

La dernière valeur s'obtient avec la fonction `ajout 1 (descend (n-1) (p-1))` qui fait donc des appels successifs ensuite à `ajout 0 (descend (n-1-k) (p-1))`. Pour la même raison que plus haut, la dernière de ces valeurs obtenues sera pour $n-1-k = p-1$, et il s'agira du $(p-1)$-uplet constitué de 1. L'algorithme ajoute ensuite autant de 0 devant que nécessaire, puis un seul 1. Le dernier n-uplet est donc $(1;\ \underbrace{0;\ldots;0}_{n-p-1 \text{ fois}};\underbrace{1;\ldots;1}_{p-1 \text{ fois}})$.

Question 27 Entre le premier et le dernier, si $p \neq 0$ et $p \neq n$, la propriété est toujours vraie et découle de l'expression de ces n-uplets à la question précédente (si $p = 0$ ou $p = n$, il y a une seule combinaison et dans ce cas, la propriété n'est pas vérifiée à strictement parler).

Nous démontrons maintenant par récurrence sur n la propriété suivante : pour tout n, tout p, deux combinaisons successives (dans l'ordre de Gray) de p bits parmi n diffèrent d'exactement deux bits. C'est vrai si $n = 1$, car que p soit égal à 0 ou 1, il n'y a qu'une seule combinaison, donc il n'existe pas de combinaisons successives.

Supposons le résultat vrai pour $n-1$ fixé, et toute valeur de p entre 0 et $n-1$. On regarde maintenant les combinaisons de p parmi n, avec une valeur particulière de p. Si $p = 0$ ou $p = n$, il y a une unique combinaison et la propriété est donc vraie. Sinon, rappelons que les valeurs du codage de Gray sont construites avec la relation de récurrence vue plus haut : on prend le codage de Gray à $n-1$ bits précédé d'un 0, puis la même précédée d'un 1, et écrite en sens opposé.

Considérons maintenant deux séquences consécutives correspondant à une combinaison de p éléments dans cette liste :

— Si ces deux séquences commencent par un zéro, alors elles correspondent à une combinaison de p éléments sur $n-1$ bits, et par récurrence on peut conclure que seuls deux bits sont changés de l'une à l'autre.

— Si ces deux séquences commencent par un 1, alors sans le bit du début, elles correspondent à une combinaison de $p-1$ éléments sur $n-1$ bits, et par récurrence on peut également conclure que seuls deux bits sont changés.

— Si la première séquence commence par un 0 et la seconde par un 1, alors la première est un zéro suivi de la dernière combinaison de p bits parmi $n-1$, soit $(0;1;\underbrace{0;\dots;0}_{n-p-2\text{ fois}};\underbrace{1;\dots;1}_{p-1\text{ fois}})$. La seconde est un 1 suivi de la première combinaison de $p-1$ bits parmi $n-1$, soit $(1;\underbrace{0;\dots;0}_{n-p-1\text{ fois}};\underbrace{1;\dots;1}_{p-1\text{ fois}})$. Il n'y a donc bien que deux bits à modifier (les deux premiers), et la propriété est donc toujours vraie.

Question 28 Puisque l'on sait que $\exists! j \in [\![1, \max(p, n-p)]\!],\ g^{-1}(a) - g^{-1}(a') \equiv 2^j \mod 2^n$, on en déduit que :

$$\exists! j \in [\![1, \max(p, n-p)]\!],\ a' = g((g^{-1}(a) - 2^j) \bmod 2^n)$$

Ainsi il suffit d'essayer successivement les différentes valeurs de j possibles jusqu'à obtenir une combinaison comportant exactement p fois le nombre 1.

```
let base2 k n = (* converti k en base 2 sur n bits *)
    let rec aux_b2 k l taille = match k with
    | 0 -> if taille==n then l
           else aux_b2 k (0::l) (taille+1)
    | _ -> aux_b2 (k/2) ((k mod 2)::l) (taille+1)
    in
      aux_b2 k [] 0
;;

let xor x y = if x!=y then 1
                      else 0
;;

let g k n = (* retourne g(k) sur n bits *)
  let k_base_2 = 0::(base2 k n) in
    let rec g_xor b = match b with
    | t1::t2::q -> (xor t1 t2)::(g_xor (t2::q))
    | _         -> []
    in
      g_xor k_base_2
;;

let inv_g b = (* retourne l'image de b par la reciproque de g *)
  let rec aux_inv_g v res b = match b with
  | []   -> res
  | t::q -> let x = xor v t in
              aux_inv_g x (2*res+x) q
  in
    aux_inv_g 0 0 b
;;

let rec nombre_de_un liste = match liste with
| []   -> 0
| t::q -> t + nombre_de_un q
;;
```

```
let suivant a =
  let p = nombre_de_un a and n = List.length a in
    let rec aux_suivant deux_puiss_j inv =
        let res = g ((inv + deux_puiss_j) mod 32) n in
            if nombre_de_un res == p then res
            else aux_suivant (2*deux_puiss_j) inv
    in
      aux_suivant 2 (inv_g a)
;;
```

Les signatures des fonctions proposées sont :

- `base2        : int -> int -> int list`
- `xor          : 'a -> 'a   -> int`
- `g            : int -> int -> int list`
- `inv_g        : int list   -> int`
- `nombre_de_un : int list   -> int`
- `suivant      : int list   -> int list`

Question 29 On propose de parcourir l'ensemble des combinaisons de p objets du sac. On part de la première combinaison $(\underbrace{0;\ldots;0}_{n-p \text{ fois}};\underbrace{1;\ldots;1}_{p \text{ fois}})$ puis on explore les suivantes à l'aide de la nouvelle fonction `suivant`. À chaque nouvelle combinaison, on recalcule le poids et la valeur du sac. Si le poids associé à cette combinaison ne dépasse pas le poids maximal M du sac et que la valeur obtenue est meilleure que les précédentes, on stocke cette combinaison. Notons qu'il serait possible de détecter quels sont les 2 bits qui ont changé pour réaliser l'évaluation du poids et de la valeur en un total de deux additions et deux soustractions. À l'issue du parcours, on retourne la dernière combinaison stockée. On n'oubliera pas de prendre garde au fait que le problème n'a pas de solution réalisable lorsque toutes les combinaisons de p objets sont trop lourdes pour le sac.

Une implémentation (non demandée) pourrait alors être la suivante :

```
let rec premier p n =
  if n>p then
    0::(premier p (n-1))
  else if n==p then
  (
    if n==0 then
      []
    else
      1::(premier (p-1) (n-1))
  )
  else failwith "premier : erreur p > n"
;;

let produit_scalaire x y =
    let rec sommer res u v = match u,v with
    | []     , _     -> res
    | tu::qu,tv::qv -> sommer (res+tu*tv) qu qv
    | _ -> failwith "produit_scalaire : x et y ne sont pas de même longueur"
    in
      sommer 0 x y
;;
```

```
let poids solution masses = produit_scalaire solution masses;;

let prix solution valeurs = produit_scalaire solution valeurs;;

let binom p n =
  let memoization = Array.make_matrix (p+1) (n+1) (-1) in
    for i=0 to n do
      memoization.(0).(i) <- 1
    done;
    for j=1 to p do
      memoization.(j).(0) <- 0
    done;
    let rec calcule a b =
      if memoization.(a).(b) == -1 then
      (
        memoization.(a).(b) <- calcule (a-1) (b-1) + calcule a (b-1)
      );
      memoization.(a).(b)
    in
      calcule p n
;;

let sac_a_dos valeurs masses p m_max =
  let n=List.length valeurs in
    let solution_courante = ref (premier p n) in
      let solution = ref !solution_courante
      and est_sol_valide = ref (poids !solution_courante masses <= m_max)
      and valeur_sol = ref (prix !solution_courante valeurs) in
        for i=2 to binom p n do
          solution_courante := suivant !solution_courante;
          if poids !solution_courante masses <= m_max then
          (
            if !est_sol_valide then
            (
              if (prix !solution_courante valeurs >= !valeur_sol) then
              (
                valeur_sol := prix !solution_courante valeurs;
                solution := !solution_courante
              )
            )
            else
            (
              est_sol_valide := true;
              valeur_sol := prix !solution_courante valeurs;
              solution := !solution_courante
            )
          )
        done;
        !est_sol_valide,!solution
;;
```