Question 1 –

```
(* suivant : int list -> int list * bool *)
(* Hypothèse: la liste donnée en argument ne contient que des 0 et des 1.
   Remarque: lorsque le n-uplet fourni en entrée est [1 ; ... ; 1], la fonction
   renvoie ([0 ; ... ; 0], false). *)
let rec suivant (li: int list): int list *  bool = match li with
  | [] -> [], false
  | a :: q -> let q2, b = suivant q in
              if b then a :: q2, true
              else  (1-a) :: q2, a = 0;;
```

Remarque. La consigne « Cette fonction modifie la liste qu'elle reçoit en argument » est étrange puisqu'une liste OCaml n'est pas modifiable. Le rapport du jury précise « Retourner un couple `bool*list` a été une solution régulièrement adoptée, ce qui est très bien ».

Question 2 –

```
(* affiche_nuplet : int list -> unit *)
let rec affiche_nuplet (li: int list): unit = match li with
  | [] -> ()
  | a :: q -> Printf.printf "%d " a;
              affiche_nuplet q;;
```

```
(* afficheQ2 : int -> unit *)
let afficheQ2 (n: int): unit =
  let c: (int list * bool) ref = ref (List.init n (fun _ -> 0) , true) in
  while snd !c do
    affiche_nuplet (fst !c);
    print_newline();
    c := suivant (fst !c)
  done;;
```

Question 3 –

```
(* ajout : int -> int list list -> int list list *)
let rec ajout (a: int) (li: int list list): int list list = match li with
  | [] -> []
  | uplet :: q -> (a :: uplet) :: (ajout a q);;
```

Question 4 –

```
(* monte : int -> int list list *)
let rec monte (n: int): int list list = match n with
  | n when n < 0 -> failwith "monte: n < 0"
  | 0 -> [[]]
  | n -> ajout 0 (monte (n-1)) @ ajout 1 (descend (n-1))
(* descend : int -> int list list *)
and descend (n: int): int list list = match n with
  | n when n < 0 -> failwith "descend: n < 0"
  | 0 -> [[]]
  | n -> ajout 1 (monte (n-1)) @ ajout 0 (descend (n-1));;
```

Question 5 – Pour tout $n \in \mathbb{N}$, notons a_n (resp. b_n) le nombre d'appels à la fonction `monte` (resp. `descend`) lors de l'exécution de « `monte n` ». On remarque que b_n (resp. a_n) est aussi le nombre d'appels à la fonction `monte` (resp. `descend`) lors de l'exécution de « `descend n` ». On a :

$$a_0 = 1 \qquad b_0 = 0 \qquad \forall n \in \mathbb{N}^* : \begin{cases} a_n = 1 + a_{n-1} + b_{n-1} \\ b_n = a_{n-1} + b_{n-1} \end{cases}$$

Une récurrence immédiate sur $n \in \mathbb{N}$ donne :

$$a_n = 2^n \qquad b_n = 2^n - 1$$

En résumé :

L'évaluation de « `monte n` » appelle 2^n fois `monte` et $2^n - 1$ fois `descend`.
L'évaluation de « `descend n` » appelle 2^n fois `descend` et $2^n - 1$ fois `monte`.

Question 6 – La liste « `monte n` » est égale à « `descend n` » en sens inverse. Ainsi, dans le cas où `n > 0`, on peut se passer de l'appel à « `descend (n-1)` » en réutilisant le résultat de l'appel à « `monte (n-1)` » :

```
(* monte_bis : int -> int list list *)
let rec monte_bis (n: int): int list list = match n with
  | n when n < 0 -> failwith "monte: n < 0"
  | 0 -> [[]]
  | n -> let li = monte_bis (n-1) in
         ajout 0 li @ ajout 1 (List.rev li);;
```

Cette fonction a une meilleure complexité puisque lors de l'évaluation de « `monte_bis n` », le nombre d'appels à `monte_bis` est $n+1$. La fonction `descend` peut être modifiée de la même façon.

Remarque. La complexité étudiée dans cette question n'est pas la complexité au sens usuel. La complexité usuelle consiste à compter le nombre d'opérations élémentaires, ce qui donne un $\Omega(2^n)$ pour la fonction `monte_bis`. En effet, la dernière concaténation et les deux derniers appels à `ajout` se font sur des listes de taille 2^{n-1}.

Question 7 – Pour tout $m \in \mathbb{N}$, notons L_m la liste dans l'ordre de Gray des 2^m m-uplets et $\overline{L_m}$ la liste L_m en sens inverse.

Par définition, $g(k)$ a pour représentation binaire le $(n+1)$-uplet d'indice k dans L_{n+1}. D'après la partie $I.B$, comme $2^n \leqslant k < 2^{n+1}$, la représentation binaire de k est un 1 suivi de la représentation binaire du n-uplet d'indice r dans $\overline{L_n}$. Par définition de g, le n-uplet d'indice r dans $\overline{L_n}$ est la représentation binaire de $g(2^n - 1 - r)$. Ainsi :

$$g(k) = 2^n + g(2^n - 1 - r)$$

Question 8 – On va montrer le résultat pour tout $n \in \mathbb{N}$ et tout $k \in [\![0; 2^{n+1} - 1]\!]$ (ce qui généralise l'hypothèse $k \in [\![2^n; 2^{n+1} - 1]\!]$ de l'énoncé). Reformulons les questions 7 et 8 :

- ⋆ Dans la question précédente, on a montré que pour tout $n \in \mathbb{N}$ et tout $(b_n, \dots, b_0) \in \{0,1\}^{n+1}$, si $b_n = 1$ alors :

$$g\left(\sum_{j=0}^{n} b_i 2^i\right) = 2^n + g\left(2^n - 1 - \sum_{j=0}^{n-1} b_i 2^i\right)$$

On en déduit que :

$$g\left(\sum_{j=0}^{n} b_i 2^i\right) = 2^n + g\left(\sum_{j=0}^{n-1} 2^i - \sum_{j=0}^{n-1} b_i 2^i\right) = 2^n + g\left(\sum_{j=0}^{n-1} (1 - b_i) 2^i\right) = 2^n + g\left(\sum_{j=0}^{n-1} (1 \oplus b_i) 2^i\right)$$

⋆ Dans cette question, on nous demande de montrer que pour tout $n \in \mathbb{N}$ et tout $(b_n, \dots, b_0) \in \{0,1\}^{n+1}$:

$$g\left(\sum_{j=0}^{n} b_i 2^i\right) = b_n 2^n + \sum_{j=0}^{n-1} (b_j \oplus b_{j+1}) 2^j.$$

Montrons ce résultat par récurrence forte sur $n \in \mathbb{N}$.

Initialisation. Pour $n = 0$:

→ Si $b_0 = 0$, alors $g(0) = 0$ et la formule est vérifiée.

→ Si $b_0 = 1$, alors $g(1) = 1$ et la formule est vérifiée.

Hérédité. Supposons le résultat jusqu'au rang $n-1$ et montrons le au rang n. Soit $(b_n, \dots, b_0) \in \{0,1\}^{n+1}$.

→ Si $b_n = 0$, alors l'hypothèse de récurrence appliquée à $(b_{n-1}, \dots, b_0)$ permet de conclure :

$$g\left(\sum_{j=0}^{n} b_i 2^i\right) = g\left(\sum_{j=0}^{n-1} b_i 2^i\right) = b_{n-1} 2^{n-1} + \sum_{j=0}^{n-2} (b_j \oplus b_{j+1}) 2^j = b_n 2^n + \sum_{j=0}^{n-1} (b_j \oplus b_{j+1}) 2^j$$

→ Sinon $b_n = 1$. On utilise la question précédente et on applique l'hypothèse de récurrence à $(1 \oplus b_{n-1}, \dots, 1 \oplus b_0)$:

$$\begin{aligned} g\left(\sum_{j=0}^{n} b_i 2^i\right) &= 2^n + g\left(\sum_{j=0}^{n-1} (1 \oplus b_i) 2^i\right) \\ &= 2^n + (1 \oplus b_{n-1}) 2^{n-1} + \sum_{j=0}^{n-2} (1 \oplus b_j \oplus 1 \oplus b_{j+1}) 2^j \\ &= b_n 2^n + \sum_{j=0}^{n-1} (b_j \oplus b_{j+1}) 2^j. \end{aligned}$$

Question 9 – À partir de maintenant, on note $(a_n \dots a_0)_2$ le nombre dont la représentation binaire est $a_n \dots a_0$.

La représentation binaire de $\lfloor k/2 \rfloor$ est la représentation binaire de k dans laquelle le bit de poids fort a été supprimé. En d'autres termes, avec les notations de la question 8 :

$$k = (b_n \dots b_0)_2 \qquad \lfloor k/2 \rfloor = (b_{n+1} \dots b_1)_2 \qquad g(k) = (a_n \dots a_0).$$

et donc :

$$\boxed{g(k) = k \oplus \lfloor k/2 \rfloor}$$

Question 10 –

⋆ Commençons par la deuxième partie de la question. Montrons pas itération finie descendante sur $j \in [\![0; n]\!]$ que :

$$\boxed{b_j = \bigoplus_{k=j}^{n} a_k}$$

Initialisation. Pour $j = n$, on a $a_n = b_n \oplus b_{n+1} = b_n$. D'où le résultat.

Hérédité. Soit $j \in [\![1; n]\!]$ tel que $b_j = \bigoplus_{k=j}^{n} a_k$. Étant donné que $a_{j-1} = b_{j-1} \oplus b_j$, on a :

$$b_{j-1} = a_{j-1} \oplus b_j = a_{j-1} \oplus \bigoplus_{k=j}^{n} a_k = \bigoplus_{k=j-1}^{n} a_k$$

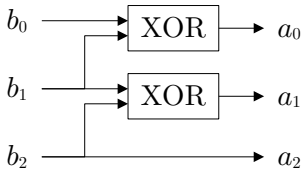
⋆ Le résultat ci-dessus impose que g soit injective. Pour tout $n \in \mathbb{N}$, soit $E_n = [\![0; 2^n - 1]\!]$, alors $g(E_n) \subset E_n$. Comme E_n est fini et que g est injective, g établit une bijection entre E_n et E_n. Ainsi, pour tout $n : E_n \subset g(\mathbb{N})$ et donc g est surjective.

$\boxed{g \text{ est bijective}}$

Question 11 – Par définition de $g(k)$:

$$a_2 = b_2 \qquad a_1 = b_2 \oplus b_1 \qquad a_0 = b_1 \oplus b_0$$

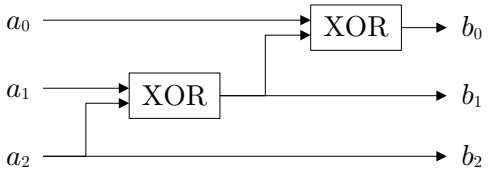
Le circuit suivant convient :



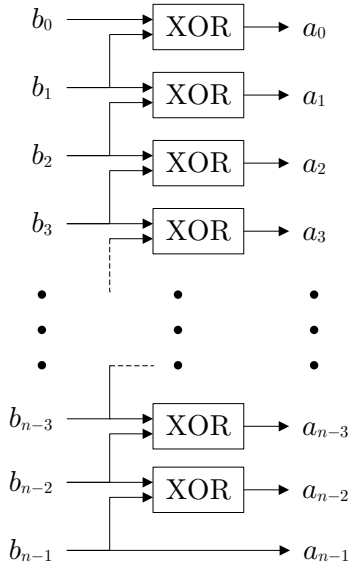
Question 12 – D'après la question 10 :

$$b_2 = a_2 \qquad b_1 = a_1 \oplus a_2 \qquad b_0 = a_0 \oplus a_1 \oplus a_2$$

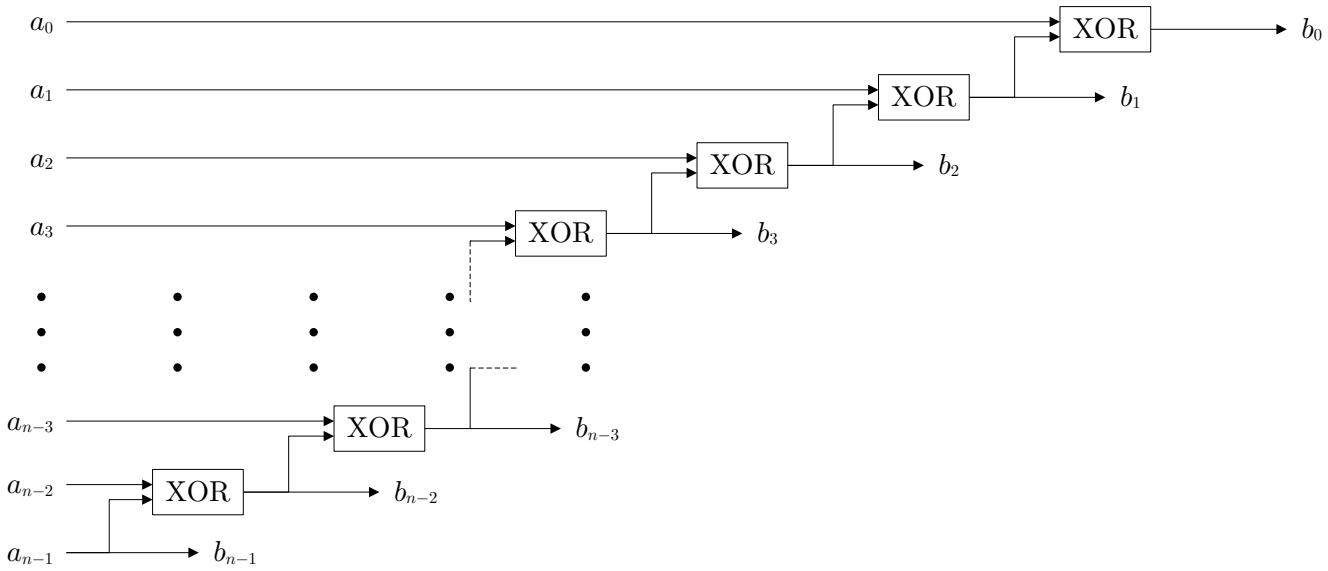
Le circuit suivant convient :



Question 13 – Pour g, on généralise le circuit de la question 11. On obtient un circuit avec $\boxed{n-1 \text{ portes}}$:



Pour g^{-1}, on généralise le circuit de la question 12. On obtient un circuit avec $\boxed{n-1 \text{ portes}}$:



Question 14 –

```
(* affiche_comb3 : int -> unit *)
let affiche_comb3 (n: int): unit =
  for c0 = 0 to n-1 do
    for c1 = c0+1 to n-1 do
      for c2 = c1+1 to n-1 do
        affiche_nuplet [c0; c1; c2]; (* Fonction de la question 2 *)
        print_newline();
      done;
    done;
  done;;
```

Question 15 – La première et la dernière combinaison dans l'ordre lexicographique sont respectivement :

$$\boxed{[0, 1, 2, \ldots, p-1] \text{ et } [n-p, n-p+1, n-p+2, \ldots, n-1].}$$

Question 16 – Il y a une erreur dans l'indication : il faut chercher le plus **grand** indice j tel que $c_{j+1} > c_j+1$.

```
(* indice_critique : int array -> int *)
(* Renvoie le plus grand indice j tel que c_{j+1} > c_j + 1 (erreur d'énoncé)
   Renvoie -1 si j n'existe pas *)
let indice_critique (comb: int array): int =
  let p = Array.length comb in
  let j = ref (p-2) in
  while !j >= 0 && comb.(!j+1) = comb.(!j) + 1 do
    decr j
  done;
  !j;;
```

```
(* comb_suivante : int -> int array -> unit *)
(* Hypothèse de l'énoncé: p > 0 *)
let comb_suivante (n: int) (comb: int array): unit =
  let p = Array.length comb in
  if comb.(p-1) < n-1 then
    comb.(p-1) <- comb.(p-1) + 1
  else begin
      let j = indice_critique comb in
      if j = -1 then failwith "comb_suivante: comb est la dernière combinaison";
      let c = comb.(j) in
      for k = j to p-1 do
        comb.(k) <- c + k - j + 1
      done;
    end;;
```

Question 17 –

```
(* print_int_array : int array -> unit *)
(* Affiche un tableau d'entiers *)
let print_int_array (tab : int array): unit =
  for i = 0 to Array.length tab - 1 do
    Printf.printf "%d " tab.(i);
  done;
  print_newline();;
```

```
(* afficher_comb : int -> int -> unit *)
(* Hypothèse de l'énoncé: p > 0 *)
let afficher_comb (n: int) (p: int): unit =
  let comb = Array.init p (fun i -> i) in
  print_int_array comb;
  while comb.(0) < n - p do
    comb_suivante n comb;
    print_int_array comb;
  done;;
```

Question 18 – On s'intéresse d'abord au nombre de combinaisons affichées après $c_0 \dots c_{p-1}$. Par définition de l'ordre lexicographique, $c'_0 \dots c'_{p-1} > c_0 \dots c_{p-1}$ si et seulement si :

$$\exists j \in [\![0, p-1]\!] : \mathcal{P}_j \qquad \text{où} \qquad \mathcal{P}_j : \begin{cases} \forall i \in [\![0, j-1]\!] : c'_i = c_i \\ c'_j > c_j \end{cases}$$

Une combinaison qui vérifie $\mathcal{P}_j$ est la concaténation de $c_0 \dots c_{j-1}$ et d'une combinaison de $p-j$ éléments de $[\![c_j+1; n-1]\!]$; soit $\binom{n-c_j-1}{p-j}$ possibilités. Au total, le nombre de combinaisons strictement après

$c_0 \dots c_{p-1}$ est :

$$\sum_{j=0}^{p-1} \binom{n-c_j-1}{p-j}$$

Finalement, le nombre de combinaisons affichées avant d'arriver à $c_0 \dots c_{p-1}$ est :

$$\boxed{\binom{n}{p} - 1 - \sum_{j=0}^{p-1} \binom{n-c_j-1}{p-j}.}$$

La combinaison elle-même n'est pas comptée

Question 19 – Soit $p \in \mathbb{N}^*$ un entier fixé et C l'ensemble de combinaisons de p éléments de E_{N+1}. On numérote les éléments de C en suivant l'ordre lexicographique par $1, 2, \dots, |C| = \binom{N+1}{p} > N$. Soit $c_0 \dots c_{p-1}$ la combinaison numéro $|C| - N$, alors d'après la question précédente :

$$\binom{N+1}{p} - \sum_{j=0}^{p-1} \binom{N-c_j}{p-j} = |C| - N$$

Pour tout $k \in [\![1, p]\!]$, on pose :

$$n_k = N - c_{p-k}$$

Alors :

$$N = \sum_{k=1}^{p} \binom{n_k}{k}$$

De plus, par définition d'une combinaison :

$$c_0 < \dots < c_{p-2} < c_{p-1} \leqslant N \qquad \text{donc} \qquad 0 \leqslant n_1 < n_2 < \dots < n_k$$

Question 20 –

⋆ Commençons par démontrer la formule de l'indication. Pour tout $k \in [\![0; p-1]\!]$, la relation de Pascal donne :

$$\binom{m-k+1}{m-p+1} = \binom{m-k}{m-p+1} + \binom{m-k}{m-p} = \binom{m-k}{m-p+1} + \binom{m-k}{p-k}$$

d'où le résultat.

⋆ La somme à calculer est donc télescopique :

$$\begin{aligned}\sum_{k=0}^{p-1} \binom{m-k}{p-k} &= \sum_{k=0}^{p-1} \left[\binom{m-k+1}{m-p+1} - \binom{m-k}{m-p+1} \right] \\ &= \binom{m+1}{m-p+1} - \binom{m-p+1}{m-p+1} \\ &= \binom{m+1}{p} - 1\end{aligned}$$

Question 21 – Montrons par récurrence sur $p \in \mathbb{N}^*$ que tout $N \in \mathbb{N}$ admet une unique décomposition combinatoire de degré p.

Initialisation. Pour $p = 1$ et $N \in \mathbb{N}$ quelconque, on doit avoir :

$$N = \sum_{k=1}^{1} \binom{n_k}{k} \qquad \text{avec} \qquad 0 \leqslant n_1$$

La seule possibilité est $n_1 = N$.

Hérédité. Soit $p \geqslant 2$. Supposons la propriété vraie pour $p - 1$.

⋆ Si $N = 0$, on doit avoir :

$$0 = \sum_{k=1}^{p} \binom{n_k}{k} \qquad \text{avec} \qquad 0 \leqslant n_1 < n_2 < \ldots < n_p$$

La seule possibilité est que $\binom{n_k}{k} = 0$ pour tout k et donc que :

$$n_1 = 0 \qquad n_2 = 1 \qquad \ldots \qquad n_p = p - 1$$

⋆ Sinon, $N \in \mathbb{N}^*$. Considérons deux décompositions combinatoires de N de degré p :

$$N = \sum_{k=1}^{p} \binom{n_k}{k} = \sum_{k=1}^{p} \binom{n'_k}{k} \qquad \text{avec} \qquad \begin{cases} 0 \leqslant n_1 < n_2 < \ldots < n_p \\ 0 \leqslant n'_1 < n'_2 < \ldots < n'_p \end{cases}$$

Si $n_p = n'_p$, l'hypothèse de récurrence appliquée à $N - \binom{n_p}{p}$ donne directement le résultat. Il suffit donc de démontrer $n_p = n'_p$. Supposons par l'absurde que $n_p + 1 \leqslant n'_p$ (le cas $n'_p + 1 \leqslant n_p$ se traite de la même façon). On a :

$$\forall k \in [\![1; p]\!] : n_k \leqslant n_p - p + k \qquad \text{et} \qquad n_p \geqslant p \text{ car } N > 0.$$

Ainsi :

$$N = \sum_{k=1}^{p} \binom{n_k}{k} \leqslant \sum_{k=1}^{p} \binom{n_p - p + k}{k} = \sum_{k'=0}^{p-1} \binom{n_p - k'}{p - k'}$$

Étant donné que $n_p \geqslant p$, par la question 20 :

$$N \leqslant \binom{n_p + 1}{p} - 1 \leqslant \binom{n'_p}{p} - 1 < \binom{n'_p}{p} \leqslant \sum_{k=1}^{p} \binom{n'_k}{k} = N$$

ce qui constitue une contradiction.

Question 22 – Pour résoudre ce problème, on peut générer toutes les combinaisons d'objets dont la masse totale est inférieure à M et rechercher parmi ces combinaisons celle dont la valeur est maximale.

Plus précisément, à l'aide de la fonction de la question 17, on génère toutes les combinaisons à p éléments de E_n et on conserve uniquement les combinaisons $c_0 \ldots c_{p-1}$ vérifiant $m_{c_0} + \ldots + m_{c_{p-1}} \leqslant M$. Les combinaisons solutions du problème sont celles qui maximisent $v_{c_0} + \ldots + v_{c_{p-1}}$.

Question 23 – Le nombre de combinaisons considérées est $\binom{n}{p}$. Pour chaque combinaisons $c_0 \ldots c_{p-1}$, on utilise $p - 1$ additions pour calculer $m_{c_0} + \ldots + m_{c_{p-1}}$ et éventuellement $p - 1$ additions pour calculer $v_{c_0} + \ldots + v_{c_{p-1}}$. Au total :

Le nombre d'additions entre m_i est $(p-1)\binom{n}{p}$.

Le nombre d'additions entre v_i est au plus $(p-1)\binom{n}{p}$.

Question 24 – La combinaison $c_0 \ldots c_{p-1}$ peut être représentée par le n-uplet $(a_0, a_1, \ldots, a_{n-1})$ où a_i indique si i fait partie de la combinaison :

$$\forall i \in [\![0, n-1]\!] : a_i = \begin{cases} 0 \text{ si } i \in \{c_0, \ldots, c_{p-1}\} \\ 1 \text{ si } i \notin \{c_0, \ldots, c_{p-1}\} \end{cases}$$

Question 25 –

```
(* monteQ25 : int -> int -> int list list *)
let rec monteQ25 (n: int) (p: int): int list list = match n,p with
  | _ when n < 0 -> failwith "monteQ25: n < 0"
  | 0,0 -> [[]]
  | 0,p -> []
  | _ -> ajout 0 (monteQ25 (n-1) p) @ ajout 1 (descendQ25 (n-1) (p-1))
(* descendQ25 : int -> int -> int list list *)
and descendQ25 (n: int) (p: int): int list list = match n,p with
  | _ when n < 0 -> failwith "descendQ25: n < 0"
  | 0,0 -> [[]]
  | 0,p -> []
  | _ -> ajout 1 (monteQ25 (n-1) (p-1)) @ ajout 0 (descendQ25 (n-1) p);;
```

Question 26 – Dans la suite, on note $M_{n,p}$ (resp. $D_{n,p}$) la liste renvoyée par « `monteQ25 n p` » (resp. « `descendQ25 n p` »).

Commençons par évoquer quelques résultats préliminaires. Ces résultats se démontrent par récurrence sur n en utilisant les codes de `monteQ25` et `descendQ25` (nous ne rédigerons pas ces preuves par soucis de concision) :

→ Pour tout $n \in \mathbb{N}$ et tout $p \in \mathbb{Z}$, la liste $D_{n,p}$ est égale à $M_{n,p}$ en sens inverse.

→ Pour tout $n \in \mathbb{N}$ et $p > n$ (resp. $p < 0$), $M_{n,p}$ et $D_{n,p}$ sont égales à la liste vide.

→ Pour tout $\mathtt{n} \in \mathbb{N}^*$, $M_{n,n}$ et $D_{n,n}$ ont pour unique élément le n-uplet $(1, \dots, 1)$. De même, $M_{n,0}$ et $D_{n,0}$ ont pour unique élément $(0, \dots, 0)$.

⋆ Montrons par récurrence sur $n \in \mathbb{N}^*$ que pour tout $p \in [\![0; n]\!]$:

> Le premier n-uplet renvoyé par « `monteQ25 n p` » est $(\underbrace{0; \dots; 0}_{n-p \text{ zéros}}; \underbrace{1; \dots; 1}_{p \text{ uns}})$

Initialisation. Pour $n = 1$, on a $p \in \{0, 1\}$ et :

$$M_{1,0} \text{ vaut } \texttt{[[0]]} \qquad\qquad M_{1,1} \text{ vaut } \texttt{[[1]]}$$

D'où l'initialisation.

Hérédité. Soit $n \geqslant 2$ et supposons la propriété vraie au rang $n-1$. Soit $p \in [\![0; n]\!]$. Le cas $p = n$ a été traité dans les résultats préliminaires. Sinon, $p \leqslant n-1$ et par hypothèse de récurrence, le premier élément de $M_{n-1,p}$ est $(\underbrace{0; \dots; 0}_{n-1-p \text{ zéros}}; \underbrace{1; \dots; 1}_{p \text{ uns}})$. D'après le code de la fonction `monteQ25`, le premier élément de $M_{n,p}$ est :

$$(0; \underbrace{0; \dots; 0}_{n-1-p \text{ zéros}}; \underbrace{1; \dots; 1}_{p \text{ uns}}) = (\underbrace{0; \dots; 0}_{n-p \text{ zéros}}; \underbrace{1; \dots; 1}_{p \text{ uns}})$$

D'où l'hérédité.

⋆ Montrons maintenant que :

> Le dernier n-uplet renvoyé par « `monteQ25 n p` » est $\begin{cases} (\underbrace{0; \dots; 0}_{n \text{ zéros}}) & \text{si } p = 0 \\ (1; \underbrace{0; \dots; 0}_{n-p \text{ zéros}}; \underbrace{1; \dots; 1}_{p-1 \text{ uns}}) & \text{si } p > 0 \end{cases}$

On traite plusieurs cas :

→ Si $n = 1$. Le résultat est vrai d'après l'initialisation de la récurrence précédente.

→ Si $p = 0$. Ce cas a été démontré dans les résultats préliminaires.

→ Sinon, $n \geqslant 2$ et $p \in [\![1, n]\!]$. Dans ce cas, on sait que le dernier élément de $D_{n-1,p-1}$ est égal au premier élément de $M_{n-1,p-1}$, c'est à dire $(\underbrace{0; \ldots; 0}_{n-p \text{ zéros}}; \underbrace{1; \ldots; 1}_{p-1 \text{ uns}})$. Ainsi, d'après le code de `monteQ25`, le dernier élément de $M_{n,p}$ est :

$$(1; \underbrace{0; \ldots; 0}_{n-p \text{ zéros}}; \underbrace{1; \ldots; 1}_{p-1 \text{ uns}})$$

Question 27 – Étant donné que `monteQ25` et `descendQ25` renvoient la même liste en sens inverse, il suffit de montrer le résultat pour `monteQ25`. Il s'agit de montrer que pour tout $n \in \mathbb{N}^*$ et tout $p \in [\![0, n]\!]$, si $M_{n,p}$ contient au moins deux éléments, alors :

- Entre deux éléments successifs de $M_{n,p}$, seuls deux bits changent.
- Entre le premier et dernier élément de $M_{n,p}$, seuls deux bits changent.

Le deuxième point est une conséquence directe des résultats de la question précédente. Montrons le premier point par récurrence sur $n \in \mathbb{N}^*$.

Soit $n \in \mathbb{N}^*$ et supposons le résultat vrai pour tout $n' < n$. Afin que $M_{n,p}$ contienne au moins deux éléments, on doit avoir $n \geqslant 2$ et $p \in [\![1; n-1]\!]$. Soient t_1, t_2 deux éléments successifs dans $M_{n,p}$:

→ Si t_1 et t_2 appartiennent à la liste « `ajout 0 (monteQ25 (n-1) p)` », alors il existe (t'_1, t'_2) deux éléments successifs de $M_{n-1,p}$ tels que t_1 (resp t_2) est le n-uplet contenant 0 suivit de t'_1 (resp. t'_2). $M_{n-1,p}$ contient au moins deux éléments, donc l'hypothèse de récurrence assure qu'entre t'_1 et t'_2, seuls deux bits changent. D'où le résultat.

→ La preuve est similaire lorsque t_1 et t_2 appartiennent à « `ajout 1 (descendQ25 (n-1) (p-1))` ».

→ Si t_1 appartient à « `ajout 0 (monteQ25 (n-1) p)` » et t_2 à « `ajout 1 (descendQ25 (n-1) (p-1))` », il faut montrer que :

> Entre le dernier élément de $M_{n-1,p}$ et le premier élément de $D_{n-1,p-1}$, seul un bit change.

c'est à dire que :

> Entre le dernier élément de $M_{n-1,p}$ et le dernier élément de $M_{n-1,p-1}$, seul un bit change.

En distinguant les cas $p = 1$ et $p > 1$, on se rend compte c'est bien le cas d'après la question précédente.

Remarque concernant la question 28 Il semble y avoir une erreur dans l'indication de l'énoncé. Il faut certainement échanger les rôles de $g^{-1}(a')$ et $g^{-1}(a)$:

$$\exists\, ! \, j \in [\![1; \max(p, n-p)]\!] : g^{-1}(a') - g^{-1}(a) \equiv 2^j \mod 2^n.$$

En effet, pour $n = 3$, les n-uplets dans l'ordre de Gray sont :

$$(0;0;0); (0;0;1); (0;1;1); (0;1;0); (1;1;0); (1;1;1); (1;0;1); (1;0;0)$$

Si on prend $p = 1$, $a = (0;0;1)$ et $a' = (0;1;0)$, on obtient $g^{-1}(a) = 1$ et $g^{-1}(a') = 3$. La relation de l'énoncé est alors fausse :

$$-2 \equiv 2 \mod 8 \qquad \text{ou} \qquad -2 \equiv 4 \mod 8.$$

Question 28 – Avec les notations de l'énoncé, on a $0 \leqslant g^{-1}(a) < g^{-1}(a') < 2^n$, ainsi il existe $j \in [\![1; \max(p, n-p)]\!]$ tel que :

$$g^{-1}(a') = g^{-1}(a) + 2^j.$$

De plus, d'après la question précédente, seuls deux bits changent entre a et a'. Ainsi pour passer de a à a', il s'agit de trouver le plus petit $j \geqslant 1$ vérifiant $\mathcal{P}$:

$$(\mathcal{P}) : \begin{cases} g(g^{-1}(a) + 2^j) \text{ a la même écriture binaire que } a \text{ sauf qu'un 0 a} \\ \text{été remplacé par un 1 et un 1 a été remplacé par un 0.} \end{cases}$$

Notons $000b_{n-1} \dots b_0$ et $000b'_{n-1} \dots b'_0$ les représentations binaires sur n-bits de $b = g^{-1}(a)$ et $b' = g^{-1}(a')$ précédées de trois zéros. Pour tout $j \in [\![1; \max(p, n-p)]\!]$, on note $i_j > j$ l'indice du bit se trouvant directement à gauche du premier zéro se trouvant à gauche de b_j. En particulier si $b_j = 0$, alors $i_j = j+1$. En distinguant les cas $b_j = 0$ et $b_j = 1$, montrons que :

$$j \text{ vérifie } \mathcal{P} \Leftrightarrow b_j \oplus b_{j-1} \oplus b_{i_j} = 1$$

⋆ Pour $b_j = 0$:

Numéro du bit	…	$i_j = j+1$	j	$j-1$	…
$g^{-1}(a)$	…	b_{i_j}	0	b_{j-1}	…
$a = g(g^{-1}(a))$	…	$b_{i_j} \oplus b_{i_j+1}$	b_{i_j}	b_{j-1}	…
$g^{-1}(a) + 2^j$	…	b_{i_j}	1	b_{j-1}	…
$g(g^{-1}(a) + 2^j)$	…	$b_{i_j} \oplus b_{i_j+1}$	$b_{i_j} \oplus 1$	$b_{j-1} \oplus 1$	…

La propriété $\mathcal{P}$ est vérifiée lorsque $b_{i_j} = 0$ et $b_{j-1} = 1$ ou inversement. C'est à dire lorsque $b_j \oplus b_{j-1} \oplus b_{i_j} = 1$.

⋆ Pour $b_j = 1$:

Numéro du bit	…	i_j	i_j-1	i_j-2	i_j-3	…	$j+1$	j	$j-1$	…
$g^{-1}(a)$	…	b_{i_j}	0	1	1	…	1	1	b_{j-1}	…
$a = g(g^{-1}(a))$	…	$b_{i_j} \oplus b_{i_j+1}$	b_{i_j}	1	0	…	0	0	$b_{j-1} \oplus 1$	…
$g^{-1}(a) + 2^j$	…	b_{i_j}	1	0	0	…	0	0	b_{j-1}	…
$g(g^{-1}(a) + 2^j)$	…	$b_{i_j} \oplus b_{i_j+1}$	$b_{i_j} \oplus 1$	1	0	…	0	0	b_{j-1}	…

La propriété $\mathcal{P}$ est vérifiée lorsque $b_{i_j} = 0$ et $b_{j-1} \oplus 1 = 1$ ou inversement. C'est à dire lorsque $b_j \oplus b_{j-1} \oplus b_{i_j} = 1$.

En résumé, pour la fonction `suivantQ28`, il suffit de trouver le plus petit $j \geqslant 1$ tel que $b_j \oplus b_{j-1} \oplus b_{i_j} = 1$, puis de calculer $g(g^{-1}(a) + 2^j)$.

```
(* xor : int -> int -> int *)
let xor (b1: int) (b2: int) =
  if b1 < 0 || b1 > 1 || b2 < 0 || b2 > 1 then failwith "xor";
  (b1 + b2) mod 2;;
```

```
(* fonction_g : int list -> int list *)
(* Renvoie [] pour [] *)
let rec fonction_g (t: int list): int list = match t with
  | [] -> []
  | b2 :: q1 -> match fonction_g q1 with
                | [] -> [b2]
                | b1 :: q2 -> b2 :: (xor b1 b2) :: q2;;
```

```
(* fonction_g_inv : int list -> int list *)
(* Renvoie [] pour [] *)
let rec fonction_g_inv (t: int list): int list = match t with
  | [] -> []
  | [b] -> [b]
  | b1 :: b2 :: q -> b1 :: fonction_g_inv ((xor b1 b2)::q);;
```

```
(* couper : int -> int list -> int list * int list *)
(* Cette fonction prend en entrée t l'écriture binaire de g^{-1}(a) et coupe
   cette liste en deux. La coupure est faite entre b_j et b_{j-1}.
   bij est le bit b_{i_j} défini dans les explications. *)
let rec couper (bij: int) (t: int list): int list * int list = match t with
  | [] -> failwith "couper: liste vide"
  | [b] -> [], [b]
  | b1 :: b2 :: q ->
     match couper (if b2 = 0 then b1 else bij) (b2 :: q) with
     | [], li when xor (xor b1 b2) bij = 1 -> [b1], li
     | [], li -> [], b1 :: li
     | li1 , li2 -> b1 :: li1, li2;;
```

```
(* suivant_image : int list -> int list * bool *)
(* Prend en entrée g^{-1}(a) et renvoie g^{-1}(a') = g^{-1}(a) + 2^j
   Renvoie également un booléen indiquant si la combinaison suivante existe. *)
let suivant_image (t: int list): int list * bool = match couper 0 t with
  | [], li -> t, false
  | li1, li2 ->
     match suivant li1 with
     | _  , false -> t, false
     | li3, true  -> li3 @ li2, true;;
```

```
(* suivantQ28 : int list -> int list * bool *)
let suivantQ28 (t: int list): int list * bool =
  let u = fonction_g_inv t in
  let v, b = suivant_image u in
  if b then fonction_g v, true else t, false;;
```

Question 29 – Pour résoudre ce problème, on peut générer toutes les combinaisons d'objets dont la masse totale est inférieure à M et rechercher parmi ces combinaisons celle dont la valeur est maximale.

Plus précisément, à l'aide de la fonction de la question 28, on génère toutes les combinaisons à p éléments de E_n en partant de $(\underbrace{0;\ldots;0}_{n-p \text{ zéros}};\underbrace{1;\ldots;1}_{p \text{ uns}})$ et on conserve uniquement les combinaisons $c_0 \ldots c_{p-1}$ vérifiant $m_{c_0} + \ldots + m_{c_{p-1}} \leqslant M$. Les combinaisons solutions du problème sont celles qui maximisent $v_{c_0} + \ldots + v_{c_{p-1}}$.

L'un des intérêts de cette méthode par rapport à celle de la question 22 est qu'il n'est pas nécessaire de recalculer entièrement la quantité $m_{c_0} + \ldots + m_{c_{p-1}}$ (resp. $v_{c_0} + \ldots + v_{c_{p-1}}$) à chaque étape. En effet, d'après la question 27, passer d'une combinaison à la suivante consiste à enlever un élément du sac et à en ajouter un autre. Il suffit donc de soustraire l'un des m_i (resp. v_i) et d'ajouter un $m_{i'}$ (resp. $v_{i'}$).