Compression entropique

I Comptage d'occurrences

Question I.1.

```
let n_max = 255;;

let occurrences l =
  let occ = Array.make n_max 0 in
  let rec lecture l =
    match l with
    |[] -> occ
    |t::q -> occ.(t) <- occ.(t) + 1;
        lecture q in
  lecture l;;
```

Question I.2.

On va lire deux fois le tableau ; une première fois pour compter le nombre de lettres dans la variable `nb` qui ont au moins une occurrence puis créer le tableau qu'une seconde lecteure remplira. On pouvait aussi ajouter les couples dans une liste qui est convertie en tableau à la fin.

```
let nonzero_occurences t =
  let n = Array.length t in
  let nb = ref 0 in
  for i = 0 to n -1 do
    if t.(i) > 0
    then incr nb
  done;
  let occ = Array.make !nb (0, 0) in
  let place = ref 0 in
  for i = 0 to n -1 do
    if t.(i) > 0
    then begin
      occ.(!place) <- (i, t.(i));
      incr place
    end
  done;
  occ;;
```

II Arbres binaires

Question II.1.

Un élément de $\mathcal{T}_{\{A,B,C,D\}}$ est un arbre à 4 feuilles donc 3 nœuds.
Il y a 5 arbres à 3 nœuds (on ne représente que les nœuds).
Pour chaque arbre il y 4! manières de nommer les feuilles avec les valeurs de $\{A, B, C, D\}$: on trouve bien $5 \cdot 24 = 120$ éléments dans $\mathcal{T}_{\{A,B,C,D\}}$

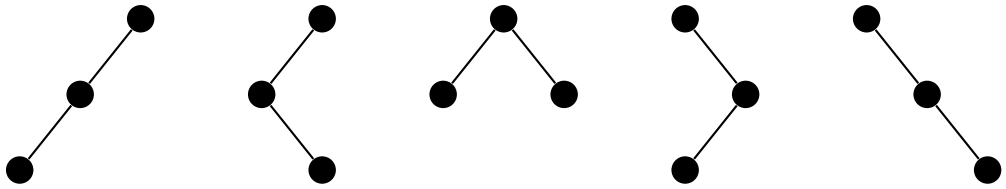

FIGURE 1 – Les arbres à 3 nœuds.

Question II.2.

On écrit une fonction auxiliaire récursive qui calcule la contribution de chaque fils à $c_q(t)$ et les additionne ou calcule la contribution d'une feuille ; cette fonction doit recevoir la profondeur de la racine pour calculer la hauteur des feuilles.

```
let cq arbre q =
  let rec calcul a prof =
    match a with
    |F k -> q.(k) * prof
    |N (g, d) -> (calcul g (prof + 1)) + (calcul d (prof +1))
  in calcul arbre 0;;
```

La fonction `calcul` effectue 3 opérations et 2 appels récursifs pour un nœud et une opération (et une lecture) pour une feuille : la complexité est donc linéaire en la taille de l'arbre. Comme la taille est égale à $2 \cdot |\mathcal{S}| - 1$, la complexité est en $O(|\mathcal{S}|)$.

Question II.3.

On remonte le chemin en ajoutant les branches parcourues depuis la feuille. On doit gérer le fils qui contient la lettre, pour cela la fonction auxiliaire renvoie un booléen signifiant l'appartenance de la lettre ainsi que le chemin vers la lettre, le chemin est quelconque si la lettre n'appartient pas au sous-arbre.

```
let get_path k arbre =
  let rec trouve k a =
    match a with
      |F p -> (k = p, [])
      |N (g, d) -> let bg, chemg = trouve k g in
                   if bg
                   then (true, 0 :: chemg)
                   else let bd, chemd = trouve k d in
                        (bd, 1 :: chemd) in
  let b, chem = trouve k arbre in
  if b
  then chem
  else failwith "Lettre non trouvée";;
```

Dans le cas pire on lit tous les nœuds et feuilles de l'arbre, la complexité est encore en $O(|\mathcal{S}|)$.

Question II.4.

On commence par un fonction qui crée un arbre correspondant à $(0|1)^k$ avec des valeurs consécutives aux feuilles qui commencent par k. La fonction renvoie aussi l'entier qui succède à la valeur de dernière feuille pour pouvoir construire les arbres suivants.

```
let rec complet debut hauteur =
  if hauteur = 0
  then F debut, debut + 1
  else let g, ng = complet debut (hauteur -1) in
       let d, nd = complet ng (hauteur -1) in
       N (g, d), nd;;
```

On construit alors l'arbre demandé, à chaque étape le fils gauche est un arbre complet et le fils droit reproduit la construction avec une hauteur d'arbre complet incrémentée ; quand cette hauteur est égale à k, on ne construit qu'une feuille.

```
let integers k_max =
  let rec construction debut hauteur =
    if hauteur = k_max
    then F debut
    else let g, n = complet debut hauteur in
      N (g, construction n (hauteur + 1))
  in construction 0 0 ;;
```

III Codes préfixes

Question III.1.

On suppose qu'une séquence de bits qui représente deux mots $\sigma_0\sigma_1\cdots\sigma_p$ et $\sigma'_0\sigma'_1\cdots\sigma'_q$.
On considère les lettres initiales communes : $\sigma_0 = \sigma'_0$, $\sigma_1 = \sigma'_1$, …, $\sigma_{i-1} = \sigma'_{i-1}$ sans pouvoir poursuivre. On est alors dans un des 4 cas suivants

- Si $i = 1 = p = q$, les deux mots sont égaux.
- Si $i - 1 = p < q$, le premier mot est un préfixe de l'autre. Dans ce cas $\sigma_0\sigma_1\cdots\sigma_p$ est représenté par la séquence et par une séquence strictement plus courte, ce qui est impossible.
- De même $i - 1 = q < p$ est impossible.
- Si $i - 1 < p$ et $i - 1 < q$ alors $\sigma_i \neq \sigma'_i$. On suppose par exemple que σ_i est représenté par k bits et σ'_i est représenté par k' bits avec $k \leqslant k'$. Alors, comme les bits antérieurs sont égaux, les k premiers bits de la représentation de σ'_i sont ceux de la représentation de σ_i. Quand on parcourt l'arbre avec les bits de la représentation de σ_i on aboutit à une feuille étiquetée par σ_i, elle n'est pas étiquetée par σ'_i et on ne peut pas prolonger le parcours car on est parvenu à une feuille, on ne pourra donc pas atteindre σ'_i et on aboutit aussi à une impossibilité.

La seulle possibilité est que les deux mots soient égaux.

Question III.2.

On commence par lire une lettre

```
let rec lire_lettre bits arbre =
  match bits, arbre with
  |_, F k -> k, bits
  |[], _ -> failwith "Il n'y a pas de lettre lisible"
  |0::suite, N (g, d) -> lire_lettre suite g
  |b::suite, N (g, d) -> lire_lettre suite d;;
```

On peut alors construire la mot.

```
let rec decomp1 bits arbre =
  match bits with
  |[] -> []
  |_ -> let s, suite = lire_lettre bits arbre in
        s :: (decomp1 suite arbre);;
```

On lit la liste des bits terme-à-terme, la complexité est donc en $O(n)$ où n est la longueur de la séquence de bits.

IV Arbres optimaux

Question IV.1.

On note de même $\mathcal{S}_d$ les éléments de $\mathcal{S}$ qui sont qui sont des étiquettes des feuilles de `d`.
La profondeur d'une feuille de `g` dans `t` est égale à sa profondeur dans `g` augmentée de 1 donc

$$\begin{aligned} c_q(t) &= \sum_{\sigma\in\mathcal{S}_g} q(\sigma)\ell_t(\sigma) + \sum_{\sigma\in\mathcal{S}_d} q(\sigma)\ell_t(\sigma) \\ &= \sum_{\sigma\in\mathcal{S}_g} q(\sigma)(\ell_g(\sigma)+1) + \sum_{\sigma\in\mathcal{S}_d} q(\sigma)\ell_t(\sigma) \\ &= \sum_{\sigma\in\mathcal{S}_g} q(\sigma)\ell_g(\sigma) + \sum_{\sigma\in\mathcal{S}_g} q(\sigma)) + \sum_{\sigma\in\mathcal{S}_d} q(\sigma)\ell_t(\sigma) \\ &= c_q(g) + \sum_{\sigma\in\mathcal{S}_g} q(\sigma)) + \sum_{\sigma\in\mathcal{S}_d} q(\sigma)\ell_t(\sigma) \end{aligned}$$

Si `g` n'était pas optimal pour $(q, \mathcal{S}_g)$ il existerait `g'` tel que $c_q(g') < c_q(g)$.
On a $\mathcal{S}_{g'} = \mathcal{S}_g$ et, pour l'arbre `t' = N (g', d)`, on aurait

$$c_q(t') = c_q(g') + \sum_{\sigma\in\mathcal{S}_{g'}} q(\sigma)) + \sum_{\sigma\in\mathcal{S}_d} q(\sigma)\ell_t(\sigma) < c_q(g) + \sum_{\sigma\in\mathcal{S}_g} q(\sigma)) + \sum_{\sigma\in\mathcal{S}_d} q(\sigma)\ell_t(\sigma) = c_q(t)$$

ce qui contredit l'optimalité de `t`. Ainsi `g` est optimal pour $(q, \mathcal{S}_g)$

Question IV.2.

On va montrer la contraposée : tout arbre dans lequel la feuille $F(\sigma_0)$ n'est pas à profondeur 1 ne peut être optimal. Soit $\texttt{t} \in \mathcal{T}_\mathcal{S}$ dans lequel la feuille $F(\sigma_0)$ est à profondeur $k > 1$.
On suppose, par exemple, que la feuille $F(\sigma_0)$ est dans le fils droit ; on considère alors l'arbre `t'` obtenu en échangeant dans `t` le fils gauche `g` et la feuille $F(\sigma_0)$.
On a $\ell_{t'}(\sigma_0) = \ell_t(\sigma_0) - k + 1$ et, pour tout $\sigma \in \mathcal{S}_g$, $\ell_{t'}(\sigma) = \ell_t(\sigma) + k - 1$. On a donc

$$\begin{aligned} c_q(t') &= c_g(t) - (k-1)q(\sigma_0) + (k-1)\sum_{s\in\mathcal{S}_g} q(\sigma) \\ &\leqslant c_g(t) - (k-1)q(\sigma_0) + (k-1)\sum_{s\in\mathcal{S}} q(\sigma) \\ &< c_g(t) - (k-1)q(\sigma_0) + (k-1)q(\sigma_0) = c_q(t) \end{aligned}$$

donc `t` n'est pas optimal.

Question IV.3.

1. On remarque pour commencer que, dans un arbre optimal, si deux feuilles $F(\sigma)$ et $F(\sigma')$ sont a des profondeur k et k' avec $k' > k$ alors on doit avoir $q(\sigma) \geqslant q(\sigma')$ car l'arbre `t'` obtenu en échangeant ces deux feuilles vérifie

$$c_q(t') = c_q(t) + (k' - k)q(\sigma) + (k - k')q(\sigma') = c_q(t) + (k' - k)(q(\sigma) - q(\sigma')) \leqslant c_q(t)$$

Ainsi, dans un arbre optimal, si $\ell(\sigma) > \ell(\sigma_1)$ alors $q(\sigma) \leqslant q(\sigma_1)$ d'où $q(\sigma) = q(\sigma_1)$.
En considérant un nœud de profondeur maximale, il a pour fils deux feuilles $F(\sigma_a)$ et $F(\sigma_b)$ qui sont de profondeur maximale L.
   - Si $\ell(\sigma_1) < L$ ou $\ell(\sigma_2) < L$ alors la remarque ci-dessus impose qu'on ait $q(\sigma_a) = q(\sigma_b) = q(\sigma_1) = q(\sigma_2)$ donc on peut échanger σ_a et σ_1 ainsi que σ_b et σ_2 sans changer la valeur de c_q et on a bien un nœud $N(F(\sigma_1), F(\sigma_2))$.
   - Si $\ell(\sigma_1) = \ell(\sigma_2) = L$ on peut encore échanger σ_a et σ_1 ainsi que σ_b et σ_2 sans changer la valeur de c_q et on a bien un nœud $N(F(\sigma_1), F(\sigma_2))$.
2. Quand on passe de `t` à `t'` en remplaçant $F(\sigma_3)$ par $N(F(\sigma_1), F(\sigma_2))$ on a, en conservant la même notation pour la profondeur dans les deux arbres,
   $\ell(\sigma_1) = \ell(\sigma_2) = \ell(\sigma_3) + 1$ donc
   $c_q(t) = c_{q'}(t') - q'(\sigma_3)\ell(\sigma 3) + (q(\sigma_1) + q(\sigma_2))(\ell(\sigma_3) + 1) = c_{q'}(t') + q(\sigma_1) + q(\sigma_2)$
   On peut considérer un arbre optimal pour $(q, \mathcal{S})$, t_1 qui, d'après la question précédente, contient un nœud $N(F(\sigma_1), F(\sigma_2))$. Si on remplace de nœud par une feuille $F(\sigma_3)$ on obtient un arbre $t'_1 \in \mathcal{T}_{\mathcal{S}'}$ avec, par le même calcul $c_q(t_1) = c_{q'}(t'_1) + q(\sigma_1) + q(\sigma_2)$.
   On a $c_q(t) \geqslant c_q(t_1)$ car t_1 est optimal et $c_{q'}(t') \leqslant c_{q'}(t'_1)$ car t' est optimal donc
   $c_q(t) = c_{q'}(t') + q(\sigma_1) + q(\sigma_2) \leqslant c_{q'}(t'_1) + q(\sigma_1) + q(\sigma_2) = c_q(t_1)$ d'où $c_q(t) = c_q(t_1)$.
   Ainsi t est bien optimal.

Question IV.4.

1.
```
let rec insert (k, a) l =
  match l with
  |[] -> [(k, a)]
  |(k', a') :: reste when k <= k' -> (k, a) :: l
  |(k', a') :: reste -> (k', a') :: (insert (k, a) reste);;
```

2. On sépare la conversion de la liste qui définit q en une liste de feuilles ; on n'a pas besoin d'utiliser `insert` car la liste est déjà triée selon les valeurs de q.

```
let rec convert l =
  match l with
  |[] -> []
  |(s, q) :: reste -> (q, F s) :: (convert reste);;
```

```
let optimal l =
  let rec creer la =
    match la with
    |[] -> failwith "L'alphabet est vide"
    |[(k, a)] -> a
    |(k1, a1) :: (k2, a2) :: reste -> creer (insert (k1 + k2,
       N(a1, a2)) reste)
  in creer (convert l);;
```

3. La fonction `insert` a un complexité au pire linéaire en la taille de la liste passée en argument, la fonction `convert` a un complexité linéaire en la taille de la liste, donc en $O(|\mathcal{S}|)$ et la fonction `optimal` utilise la fonction `insert` sur des liste de taille variant de 0 à $|\mathcal{S}|$ puis utilise `convert`. La complexité de `insert` est donc en $O(|\mathcal{S}|^2)$

Question IV.5.

Lorsque l'on retire les deux couples (k_1, t_1) et (k_2, t_2) de la liste dans `insert` tous les couples restants (k, t) on une valeur de k supérieure ou égale à $k_2 \geqslant k_1$ et on ajoute le couple $(k_1 + k_2, N(t_1, t_2))$ avec $k_1 + k_2 \geqslant k_2$. On va ensuite retirer (k_3, t_3) et (k_4, t_4) avec $k_2 \leqslant k_3 \leqslant k_4$.
Ainsi, pour $t = N(t_1, t_2)$ et $t' = N(t_3, t_4)$ on a $\bar{q}(t) = k_1 + k_2 \leqslant k_3 + k_a = \bar{t'}$.
Les arbres sont bien ajoutés à l'ensemble E par valeur croissante de $\bar{q}(t)$.

Question IV.6.

On peut alors modifier l'algorithme en n'insérant pas les arbres crées dans l'ensemble des arbres mais en les stockant dans une file d'attente, celle-ci sera alors croissante. Si on implémente cette file par un tableau de taille $|\mathcal{S}|$ qui majore le nombre d'arbres crées, on peut obtenir des complexité d'ajout et de retrait constants donc l'algorithme sera de complexité linéaire en $|\mathcal{S}|$.
Dans cet algorithme on doit choisir les deux arbres à réunir en regardant les deux arbres avec la valeur de $\bar{q}(t)$ minimale parmi les premiers éléments restants de E et ceux de la file.

V Arbres canoniques

Question V.1.

On construit l'arbre de la gauche vers la droite donc le fils gauche avant le fils droit. À chaque étape les paramètres sont le nombre de sommets déjà écrits, ici notés par `ind`, et la profondeur actuelle, ici notée `h`. S'il reste des feuilles à définir à la hauteur `h`, on renvoie une feuille dont l'étiquette est celle d'indice `ind` sinon on construit un arbre, le fils gauche, à partir de `ind` et à la profondeur $h + 1$ puis on construit le fils droit, lui aussi à la profondeur $h + 1$. On a besoin de connaître la position où démarrer le fils droit donc la fonction de création doit renvoyer aussi cette position.
Lorsque l'on construit une feuille, on doit décrémenter le nombre de feuilles à la profondeur h, on va travailler sur une copie du tableau des hauteurs. J'utilise `Array.copy` bien que cela ne fasse pas partie des rappels.

```
let canonical haut elem =
  let ht = Array.copy haut in
  let rec creer ind h =
    if ht.(h) > 0
    then begin
      ht.(h) <- ht.(h) -1;
      F elem.(ind), ind + 1 end
    else let g, i1 = creer ind (h + 1) in
      let d, i2 = creer i1 (h+1) in
      N(g, d), i2
  in fst (creer 0 0);;
```

VI Arbres alphabétiques

Question VI.1.

Si $t = N(g,d)$, on a vu qu'on avait $\ell_t(\sigma) = \ell_g(\sigma) + 1$ pour les étiquettes de g et de même pour d. On en déduit qu'on a $c_q(t) = c_q(g) + \sum_{\sigma\in\mathcal{S}_g} q(\sigma) + c_q(d) + \sum_{\sigma\in\mathcal{S}_d} q(\sigma) = c_q(g) + \bar{q}(g) + c_q(d) + \bar{q}(d)$.
Pour déterminer $m_{i,j}$ avec $i < j$ on considère tous les arbres de $\mathcal{A}_{[i,j]}$; ils sont de la forme $N(g,d)$ avec $g \in \mathcal{A}_{[i,k]}$ et $d \in \mathcal{A}_{[k+1,j]}$ avec $i \leqslant k < j$. On a donc, en posant $\bar{q}_{i,j} = \sum_{k=i}^{j} q(k)$,

$$m_{i,j} = \min\{m_{i,k} + \bar{q}_{i,k} + m_{k+1,j} + \bar{q}_{k+1,j} \ ;\ i \leqslant k < j\}$$

Cette propriété de récurrence permet d'utiliser la programmation dynamique : on peut calculer les termes $\bar{q}_{i,j}$, $m_{i,j}$ et des arbres alphabétiques-optimaux pour $\mathcal{A}_{[i,j]}$ en les mémorisant dans des matrices. Dans le calcul du minimum, on compare les termes $m_{i,k} + m_{k+1,j}$ car $\bar{q}_{i,k} + \bar{q}_{k+1,j} = \bar{q}_{i,j}$ est indépendant de k.

```
let alpha_optimal tab_q =
  let n = Array.length tab_q in
  let mat_q = Array.make_matrix n n 0 in
  let mat_m = Array.make_matrix n n 0 in
  let mat_a = Array.make_matrix n n (F 0) in
  for i = 0 to (n-1) do
    mat_q.(i).(i) <- tab_q.(i);
    mat_a.(i).(i) <- F i done;
  for i = (n-2) downto 0 do
    for j = (i+1) to (n-1) do
      let m_min = ref mat_m.(i+1).(j) in
      let k_min = ref i in
      mat_q.(i).(j) <- mat_q.(i).(i) + mat_q.(i+1).(j);
      for k = (i+1) to (j-1) do
        if mat_m.(i).(k)  + mat_m.(k+1).(j) < ! m_min
        then begin
          m_min := mat_m.(i).(k)  + mat_m.(k+1).(j);
          k_min := k end
      done;
      mat_m.(i).(j) <- !m_min + mat_q.(i).(j);
      mat_a.(i).(j) <- N (mat_a.(i).(!k_min), mat_a.(!k_min+1)
         .(j))
    done
  done;
  mat_a.(0).(n-1);;
```

Question VI.2.

Le principe reste le même ; on doit gérer les lettres à placer dans les feuilles, ici cela est fait par une variable référencée, cela évite de passer un paramètre supplémentaire à la fonction de création et au retour de la fonction.
Quand la séquence ne représente pas un arbre ce peut être quil n'y a pas assez d'information pour terminer ou qu'il reste des bits non utilisés.

```
let alpha pref =
  let n = Array.length pref in
  let s = ref 0 in
  let rec creer pos =
    if pos >= n
    then failwith "La sequence est trop courte";
    if pref.(pos) = 1
    then begin  incr s;
      F (!s - 1), (pos + 1) end
    else let g, i1 = creer (pos + 1) in
      let d, i2 = creer i1 in
      N (g, d), i2;
  in let a, m = creer 0 in
  if m < n
  then failwith "La sequence est trop longue";
  a;;
```

VII Codes arithmétiques

Question VII.1.

1. L'algorithme de la partie **IV.** fournit l'arbre de la figure 2

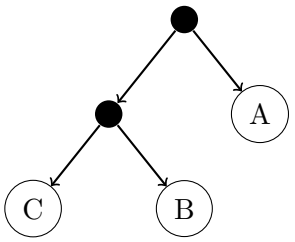

FIGURE 2 – Arbre pour l'exemple de la question**VII.1.**

   A est codé par 1 bit et B et C par 2 : un mot de $\mathcal{S}^q$ nécessite $15 \cdot 1 + 4 \cdot 2 + 1 \cdot 2 = 25$ bits.

2. Pour un mot de$\mathcal{S}^q$ on choisi les 15 emplacements de A, ce qui fait $\binom{20}{15}$ choix puis la place de C parmi les 5 places restantes.

$$|\mathcal{S}^q| = \binom{20}{15} \cdot 5 = \frac{20 \cdot 19 \cdot 18 \cdot 17 \cdot 16 \cdot 5}{5 \cdot 4 \cdot 3 \cdot 2} = 2^3 \cdot 3 \cdot 5 \cdot 17 \cdot 19$$

3. On a donc $|\mathcal{S}^q| == 2^3 \cdot 15 \cdot 17 \cdot 192^3 \cdot (2^8 - 1) \cdot 19$ avec $19 < 2^5$ donc $|\mathcal{S}^q| < 2^{17}$.
   Ainsi $\mathcal{S}^q$ est codable avec 17 bits.

On se restreint maintenant au cas où $\mathcal{S} = [0, n-1]$. Soit $N = \sum_{\sigma \in \mathcal{S}} q(\sigma)$. La fonction $E_q : \mathbb{N} \times \mathcal{S} \to \mathbb{N}$ est définie de la façon suivante pour $q(\sigma) \neq 0$:

$$E_q(x, \sigma) = \lfloor x/q(\sigma) \rfloor \cdot N + (x \bmod q(\sigma)) + \sum_{0 \leqslant k < \sigma} q(k)$$

où $\lfloor a \rfloor$ désigne la partie entière de a.
La fonction E_q peut être étendue en une fonction $C_q : \mathbb{N} \times \mathcal{S}^* \to \mathbb{N}$ qui travaille sur les mots de la façon suivante :

$$C_q\left(x, \sigma_0\sigma_1\ldots\sigma_k\right) = E_q\left(\ldots E_q\left(E_q\left(x, \sigma_0\right), \sigma_1\right), \ldots \sigma_k\right).$$

Ainsi, après avoir choisi x arbitrairement, un mot $\sigma_0\ldots\sigma_{N-1} \in \mathcal{S}^q$ peut être compressé en un entier $y = C_q\left(x, \sigma_0\ldots\sigma_{N-1}\right)$. Il est alors possible de retrouver le mot original à partir de y en calculant progressivement $\sigma_{N-1}, \sigma_{N-2}$, etc, jusqu'à σ_0.
Considérons par exemple le mot « 0120000 » ($N = 7, q = [\![5, 1, 1]\!]$). Si l'on part de $x = 0$, sa version compressée sera l'entier 151 :

$$0 \xrightarrow{0} 0 \xrightarrow{1} 5 \xrightarrow{2} 41 \xrightarrow{0} 57 \xrightarrow{0} 79 \xrightarrow{0} 109 \xrightarrow{0} 151$$

où $x \xrightarrow{\sigma} y$ signifie $E_q(x, \sigma) = y$.

Question VII.2.

On note $y = E_q(x, \sigma)$. On a $0 \leqslant x \bmod q(\sigma) < q(\sigma)$ donc

$$0 \leqslant \sum_{0\leqslant k<\sigma} q(k) \leqslant (x \bmod q(\sigma)) + \sum_{0\leqslant k<\sigma} q(k) < q(\sigma) + \sum_{0\leqslant k<\sigma} q(k) = \sum_{0\leqslant k\leqslant\sigma} q(k) \leqslant N$$

L'écriture $y = \lfloor x/q(\sigma)\rfloor \cdot N + (x \bmod q(\sigma)) + \sum_{0\leqslant k<\sigma} q(k)$ avec
$0 \leqslant (x \bmod q(\sigma)) + \sum_{0\leqslant k<\sigma} q(k) < N$ est ainsi celle de la division euclidienne de y par N :

$$(x \bmod q(\sigma)) + \sum_{0\leqslant k<\sigma} q(k) = y \bmod N \text{ et } \left\lfloor \frac{x}{q(\sigma)} \right\rfloor = \left\lfloor \frac{y}{N} \right\rfloor$$

On a vu qu'on a $\sum_{0\leqslant k<\sigma} q(k) \leqslant y \bmod N < \sum_{0\leqslant k\leqslant\sigma} q(k)$ donc σ est la première lettre telle que $y \bmod N < \sum_{0\leqslant k\leqslant\sigma} q(k)$.
On peut ainsi déterminer σ puis $x \bmod q(\sigma) = y \bmod N - \sum_{0\leqslant k<\sigma} q(k)$ puis x car on connaît aussi son quotient euclidien par $q(\sigma)$.
On calcule le tableau des sommes $\sum_{0\leqslant k\leqslant\sigma} q(k)$ qu'on utilise dans une fonction séparée pour calculer un antécédent à la fois. N en est le dernier terme.

```
let invE y q sumQ =
  let p = Array.length sumQ in
  let n = sumQ.(p-1) in
  let reste = y mod n in
  let quot = y / n in
  let sigma = ref 0 in
  while reste <= sumQ.(!sigma) do incr sigma done;
  let x = quot * q.(!sigma) + reste - sumQ.(!sigma) + q.(!
     sigma) in
  x, !sigma;;
```

```
let decomp2 q code k =
  let p = Array.length q in
  let sumQ = Array.make p 0 in
  sumQ.(0) <- q.(0);
  for i = 1 to (p-1) do
    sumQ.(i) <- sumQ.(i-1) + q.(i) done;
  let mot = Array.make k 0 in
  let y = ref code in
  for i = (k-1) downto 0 do
    let x, s = invE !y q sumQ in
    y := x;
    mot.(i) <- s done;
  !y, mot;;
```

Question VII.3.

On considère une opération élémentaire $x \to y = \left\lfloor \frac{x}{2^k} \right\rfloor \to x' = E(y, \sigma)$.

On a $x \leqslant p \iff \lfloor x \leqslant p \rfloor$ pour p entier et $\left\lfloor \frac{x'}{N} \right\rfloor = \left\lfloor \frac{y}{q(\sigma)} \right\rfloor$. On veut $x' < 2^K$

$$\begin{aligned}
x' < 2^K &\iff \frac{x'}{N} < \frac{2^K}{N} = 2^{K-B} \\
&\iff \left\lfloor \frac{x'}{N} \right\rfloor = \left\lfloor \frac{y}{q(\sigma)} \right\rfloor < 2^{K-B} \\
&\iff \frac{y}{q(\sigma)} < 2^{K-B} \\
&\iff y = \left\lfloor \frac{x}{2^k} \right\rfloor < q(\sigma) \cdot 2^{K-B} \\
&\iff \frac{x}{2^k} < q(\sigma) \cdot 2^{K-B} \\
&\iff 2^k > \frac{x}{q(\sigma)} \cdot 2^{K-B} \\
&\iff k > \log_2 \left(\frac{x}{q(\sigma)} \right) + B - K
\end{aligned}$$

Si on impose $x < 2^k$ aussi, il suffit d'avoir $k > \log_2 \left(\frac{2^K}{q(\sigma)} \right) + B - K = B - \log_2(q(\sigma)) \geqslant 0$ car $q(\sigma) \leqslant N = 2^B$. Pour être certain d'avoir $x' < 2^K$, on peut choisir $k = \lceil B - \log_2(q(\sigma)) \rceil$.
Ainsi, après avoir calculé y_i et σ_i à partir de x_{i+1} comme à la question **VII.2** , on peut calculer k_i qui ne dépend que de σ_i pour déterminer x_i.
Il est nécessaire de choisir $x_0 < 2^K$ pour que la méthode fonctionne.

VIII Code OCaml pour la question IV.6.

On commence par les fonctions de file d'attente ; la création demande un modèle et un nombre maximal, les fonctions de lecture renvoient un type optionnel pour gérer le cas d'une file vide.

```
type 'a queue = {tableau : 'a array;
                 mutable debut : int;
                 mutable libre : int};;
```

```
let create x0 n_max =
  {tableau = Array.make n_max x0; debut = 0; libre = 0};;
```

```
let is_empty qu = qu.debut = qu.libre;;
```

```
let add x qu =
  let l = qu.libre in
  qu.tableau.(l) <- x;
  qu.libre <- l + 1;;
```

```
let peek qu =
  if is_empty qu
  then None
  else Some qu.tableau.(qu.debut);;
```

```
let take qu =
  if is_empty qu
  then None
  else let d = qu.debut in
       qu.debut <- d + 1;
       Some qu.tableau.(d);;
```

On modifie `convert` pour calculer aussi la longueur, c'est-à-dire $|\mathcal{S}|$,

```
let rec convert1 l =
  match l with
  |[] -> [], 0
  |(s, q) :: reste -> let la, k = convert reste
                      in ((q, Fs) :: la, k+1);;
```

Une fonction qui choisit l'arbre pour lequel $\bar{q}$ est minimal.

```
let suivant la qu =
  match la, peek qu with
  |(k, a)::reste, Some (k', a') when k >= k' -> take qu, la
  |(k, a)::reste, _ -> Some (k, a), reste
  |[], Some (k, a) -> take qu, []
  |[], None -> None, [];;
```

On peut alors écrire la recherche d'optimum

```
let optimal1 l =
  let la, n = convert1 l in
  let qu = create (0, F 0) n in
  let rec creer la =
    let x1, la1 = suivant la qu in
    let x2, la2 = suivant la1 qu in
    match x1, x2 with
    |None, _ -> failwith "L'alphabet est vide"
    |Some (k1, a1), None -> a1
    |Some (k1, a1), Some (k2, a2) -> add (k1 + k2, N(a1, a2))
       qu;
        creer la2
  in creer la;;
```